Sublinear Approximation of Centrality Indices in Large Graphs

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of

Doctor of Philosophy

by

Benjamin W. Priest

Thayer School of Engineering
Dartmouth College
Hanover, New Hampshire

June 2017

Examining Committee:

Chairman _____
George Cybenko

Member _____
Eugene Santos

Member _____
Amit Chakrabarti

Member _____
Roger Pearce

_____
F. Jon Kull, Ph.D.
Dean of Graduate and Advanced Studies

Acknowledgements

I am priviledged to work with my advisor George Cybenko. The subject of this thesis would never have come to my attention without his prompting, and its contents would never have come to fruition without his guidance. George has been a patient mentor and a good friend.

I would like to thank Eugene Santos, Amit Chakrabarti, and Roger Pearce for agreeing to serve on my thesis committee, and for their time and guidance during the process. I would like to thank Amit Chakrabarti in particular, who introduced me to the data stream model and changed the trajectory of my career in the process.

I have been fortunate to work with my excellent co-authors and colleagues Kate Farris, Luan Huy Pham, Massamilano Albanese, Roger Pearce, Geoffrey Sanders, Keita Iwabuchi, Trevor Steil, and Jordan Katz. I would especially like to thank Roger Pearce and Geoffrey Sanders for their mentorship during my time at Lawrence Livermore National Laboratory.

I must extend my sincere gratitude to Ellen Woerta for her administrative assistance, without whom I would have been lost and confused during much of my time at Dartmouth.

Finally, I would like to thank my friends and family for bearing with me and keeping me sane during this strange journey. I appreciate especially everyone I called out of the blue to talk for hours about nothing in particular. It helped more that they can realize. I would especially like to thank Ryan Andreozzi with whom I deadlifted 500 pounds, a milestone of which I am as proud as any of my academic achievements.

This thesis is dedicated to Jennifer Lay. Whatever my destination, I hope to travel with her.

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. Traditional computational models assume that the entire input fits into working memory, which is impractical for very large graphs. Exact algorithms on very large graphs in practice hold the graph in distributed memory, where a collection of processors partition the graph. Distributed graph algorithms must optimize communication in addition to execution time. The data stream model is an alternative approach to large data scale that 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 and local triangle counts. 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 vertex- and edge-local triangle counts, with special attention paid to heavy hitter recovery. We also utilize ℓ_p sampling sketches for the adjacency information of high degree vertices to boost the performance of the sampling of random walks and subtrees. We present hybrid exact-approximating distributed algorithms for sublinearly sampling random walks, simple paths, and subtrees from scale-free graphs. We use these algorithms to approximate K-path centrality as a proxy for recovery the top-k betweenness centrality elements.

Contents

Chapter 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.

1.1 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 σ after reading t updates as $\sigma(t)$. A streaming algorithm $\mathcal{A}$ accumulates a data structure $\mathcal{S}$ while reading over σ. We will sometimes use the notation $\mathcal{D}(\sigma)$ to indicate the data structure state after $\mathcal{A}$ has accumulated σ. 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 γ [FKM+05, M+05].
The frequency vector and dynamic streams: A stream σ is often thought of as updates to a hypothetical frequency vector $\mathbf{f}(\sigma)$, which holds a counter for each element in $\mathcal{U}$. We will drop the parameterization of $\mathbf{f}$ where it is clear. We will sometimes parameterize $\mathbf{f}(t)$ to refer to $\mathbf{f}$ after reading $t \in [m]$ updates from σ. $\mathcal{D}$ can be thought of as a lossy compression of $\mathbf{f}$ that approximately preserves some statistic thereof. The stream σ's elements are of the form (i, c), where i is an index of $\mathbf{f}$ (an element of $\mathcal{U}$), $c \in [-L, \ldots, L]$ for some integer L, and (i, c) indicates that $\mathbf{f}_i \leftarrow \mathbf{f}_i + c$. Such a stream σ is called a *turnstile* or sometimes *dynamic* stream. In the *cash register* model only positive updates are permitted, whereas in the *strict* turnstile model all elements of $\mathbf{f}$ are guaranteed to retain nonnegativity.
Data sketching: Let $\circ$ be the concatenation operator on streams. For $\mathcal{A}$ a streaming algorithm, we call its data structure $\mathcal{S}$ a *sketch transform* if there is an operator $\oplus$ such that, for any streams σ_1 and σ_2,

$$\mathcal{S}(\sigma_1) \oplus \mathcal{S}(\sigma_2) = \mathcal{S}(\sigma_1 \circ \sigma_2). \tag{1.1}$$

$\mathcal{S}$ is usually determined by sampling hash functions from a suitable hash family. We will call the distribution over such samples Π a *sketch distribution*. The accumulated object $\mathcal{S}(\sigma)$ is a sketch data structure. We will sometimes abuse terms and refer to Π, $\mathcal{S}$, and $\mathcal{S}(\sigma)$ as *sketches*. The space of all possible sketch data structures given a sketch transform $\mathcal{S}$ is the sketch space $\mathbb{S}_{\mathcal{S}}$. $(\mathbb{S}_{\mathcal{S}}, \oplus)$ forms a commutative monoid, where the operator identity is the result of applying $\mathbb{S}$ to an empty stream.
Linear sketching: A sketch distribution Π is a *linear sketch* distribution if $\mathcal{S} \sim \Pi$ is a linear function of frequency vectors $\mathbf{f}(\cdot)$ of fixed dimension. For streams σ_1 and σ_2 with frequency vectors $\mathbf{f}_1$ and $\mathbf{f}_2$, scalars a and b, and linear sketch transform $\mathcal{S}$,

$$a\mathcal{S}(\mathbf{f}_1) + b\mathcal{S}(\mathbf{f}_2) = \mathcal{S}(a\mathbf{f}_1 + b\mathbf{f}_2). \tag{1.2}$$

The sketch space and the $+$ operator form a commutative group, where the operator identity is the sketch transform applied to a zero vector.
The graph stream model: In this model, the stream σ consists of m edge insertions on n vertices. This is sometimes termed the *insert only* model in the literature. The *dynamic graph stream model* also allows edge deletions. If the graph is weighted, the stream updates the weight of the corresponding edge, possibly bringing it into existence or, in the case of dynamic streams, deleting it.

1.2 Serial Graph Stream Algorithms

Streaming Degree Centrality: Classic degree centrality is still one of the most commonly applied centrality measures. Indeed, indegree centrality is known to correlate well with PageRank, and so can be used as a proxy in some scenarios [UCH03]. We demonstrate streaming $(1+\varepsilon, \delta)$-approximations for degree centrality of a graph that recovers the degree centrality heavy hitters.

These algorithms are a straightforward application of CountMinSketch, where we interpolate a graph stream as updates to a vector $\mathbf{d} \in \mathbb{R}^n$ of the degrees of the vertices. The algorithms accept a threshold fraction $\phi \in (0, 1)$ and attempt to recover all vertices $x \in \mathcal{V}$ such that $\mathbf{d}_x \geq \phi\|\mathbf{d}\|_1$. We obtain the following results:

1. In a cash register stream, we demonstrate an algorithm that recovers every vertex with degree $\geq \phi\|\mathbf{d}\|_1$, and avoids returning any vertices with degree $< (\phi - \varepsilon)\|\mathbf{d}\|_1$ with probability $(1 - \delta)$). This algorithm requires space $O(\varepsilon^{-1} \log \frac{n}{\delta})$ with update time $O(\log \frac{n}{\delta})$.

2. In a strict turnstile stream, we demonstrate an algorithm that returns every vertex with degree $\geq (\phi+\varepsilon)\|\mathbf{d}\|_1$, and avoids returning any vertices with degree $< \phi\|\mathbf{d}\|_1$ with probability $1-\delta$. The turnstile

algorithm has the increased requirements of $O(\varepsilon^{-1} \log n \log \frac{2 \log n}{\phi \delta})$ space and $O(\log n \log \frac{2 \log n}{\phi \delta})$ update time.

Constant-pass semi-streaming closeness centrality: The closeness centrality of a vertex is the inverse of all of its shortest paths, and is a common centrality index of interest in applications. We describe a semi-streaming $(r^{\log 5} - 1, \delta)$-approximation algorithm for the vertex centrality of a graph that uses $O(n^{1+1/r})$ space and $\log r$ passes, where r is an accuracy parameter. The algorithm builds on an algorithm developed by Ahn, Guha, and McGregor that builds a $(r^{\log 5} - 1)$-spanning sparsifying subgraph of an input graph [AGM12b]. Ahn, Guha, and McGregor's algorithm depends upon building random subtrees by iteratively querying ℓ_p-sampling sketches in $\log r$ passes. We then compute the closeness centrality of this subgraph, which bounds the shortest paths distance error by design. For an almost-linear runtime, we can instead employ the fast, scalable algorithm due to Cohen et al. at the expense of some additional error [CDPW14].

1.3 Vertex-Centric Distributed Streaming Graph Algorithms

Hybrid pseudo-asynchronous communication for vertex-centric distributed algorithms: Modern distributed graph algorithms must partition the adjacency matrix to processors, which communicate as required. The most natural partitioning involves assigning vertices to processors, which can be thought of as partitioning columns of the adjacency matrix. [MWM15] summarizes much of the associated literature. The iconic Pregel system executes communication as global rounds of communication between all processors [MAB+10]. However, Pregel-like systems have trouble with scale-free graphs, as high-degree vertices create storage, processing, and communication imbalances. Meanwhile, one successful approach addresses this concern by sub-partitioning high-degree vertices across processors [PGA14]. However, this approach introduces development and computational overhead due to the nature of peer-to-peer routing in implementations, such as MPI. We implement a hybrid "pseudo-asynchronous" approach where rounds of point-to-point communication occur over partitions of processors, taking advantage of modern hybrid distributed-shared memory architectures. Once a node's participation in its partitions is complete, it can drop out of the communication exchange and continue computation. We propose three protocols that route messages from source to destination over MPI.

1. **Node Local Routing**. Cores exchange messages in shared memory then forward messages to other nodes.
2. **Node Remote Routing**. Cores forward messages to other nodes then exchange messages in shared memory.
3. **Node Local Node Remote Routing** (**NLNR**). Cores locally exchange messages, followed by a remote exchange via a lattice, followed by a second local exchange.

We show in rigorous experiments that these routing exchanges exhibit better scaling characteristics than Pregel- or fully-asynchronous-style communication protocols on distributed algorithms over large graphs.

DegreeSketch and local triangle count heavy hitters: Counting global and local triangles is a canonical problem in both the random access and data stream models. In the data stream model it is known that $\Omega(n^2)$ space is required to even decide whether a graph has any triangles [BYKS02]. Vertex-local triangle counts are similarly fraught in the data stream model, as they require $\Omega(n)$ space to even write them down. Recent advances in the graph stream literature achieve low variance via clever sampling from edge streams [BBCG08, LK15, SERU17], and have been distributed to achieve better variance [SHL+18, SLO+18].

We examine a different line of analysis altogether: utilizing *cardinality sketches* to estimate local triangle counts via a sublinearization of the set intersection method. We discuss trade-offs between different estimators of the intersection of cardinality sketches, as well as different cardinality sketches themselves. We develop DegreeSketch, a distributed sketch data structure composed of cardinality sketches that can answer point queries about unions and intersections of vertex neighborhoods in a manner reminiscent of CountSketch.

We implement DegreeSketch using the famous HyperLogLog sketch [FFGM07]. We explore various algorithmic optimizations to HyperLogLog, including improved estimators [HNH13, QKT16, Lan17,

Ert17], sparse register implementation [HNH13], and compressed registers [XZC17]. We introduce novel improvements to the compressed register implementation described in [XZC17], affording lossless merging of compressed sketches. We also discuss the tradeoffs of using different cardinality sketches in implementations, such as MinCount [Gir09].

We demonstrate DegreeSketch as a tool for estimating the edge- and vertex-local triangle count heavy hitters of large graphs.

1. **Edge-local triangle count heavy hitters**. We estimate the edge-local triangle count heavy hitters of a graph using linear time and $O(n(\varepsilon^{-2}\log\log n + \log n))$ worst-case distributed memory
2. **Vertex-local triangle counts**. We estimate the vertex-local triangle counts (or, alternately, the heavy hitters) of a graph using the same asymptotic time and memory requirements.

We analyze the performance of these algorithms on numerous large graphs, and discuss many practical optimizations to the underlying algorithms. We also discuss the limitations of the approach relating to high variance on small intersections.

Distributed sampling of random walks and simple paths via fast ℓ_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 fast ℓ_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. **Hybrid sublinear random walk sampling**. We ingest a graph using some partitioning and store sparse columns of the adjacency matrix. Once a column grows sufficiently large, we write it as a stream to NVRAM and construct a logarithmic number of ℓ_p sampling sketches, where $p \in \{0, 1\}$. All subsequent additions to the row are appended to the NVRAM stream and inserted into the sketches. When sampling a neighbor from a high degree vertex, we consume one of the sampling sketches to yield the next neighbor. Once all sampling sketches of a particular vertex are consumed, we take another pass over its recorded stream and accumulate new sketches. The algorithm requires
$$O\left(\sum_{x\in\mathcal{V}} \min\left\{\mathbf{d}_x, \log^3 n \log(1/\delta)\right\}\right)$$
space for unweighted graphs, and
$$O\left(\sum_{x\in\mathcal{V}} \min\left\{\mathbf{d}_x, \varepsilon^{-1}\log(1/\varepsilon)\log^3 n \log(1/\delta)\right\}\right)$$
space for weighted graphs, where $\mathbf{d}_x$ is the degree of vertex x.
2. **Hybrid sublinear random simple path sampling**. In a manner similar to random walk sampling, we construct ℓ_p sampling sketches on demand, while forwarding sampled path histories. We use these path histories when accumulating sampling sketches so as to ignore edges that would result in loops. The resulting scheme has the same asymptotic costs as random walk sampling, but does not permit the a priori accumulation of sampling sketches, which must be accumulated from passes over recorded streams as needed.

Sublinear distributed K-path centrality: The κ-path centrality of a vertex is the probability that a random simple path (a non-self-intersecting random walk) of length $\leq \kappa$ will include it. κ-path centrality was introduced as a cheaper, more scalable alternative to betweenness centrality [ATK+11], and is shown to empirically agree on heavy hitters [KAS+13]. κ-path centrality can be approximated via a Monte Carlo simulation of $T = 2\kappa^2 n^{1-2\alpha}\ln n$ random simple paths, where α is an accuracy parameter. We sublinearize this simulation algorithm to estimate the κ-path centrality of an input graph using our hybrid sublinear random simple path sampling scheme. This affords us an algorithm that can empirically recover the betweenness centrality heavy hitters of a graph using space sublinear in the size of the graph.

Chapter 2

Background and Notation

This chapter introduces the basic concepts and notation that we will use throughout this document. Section 2.1 lays out the basic graph definitions and notation conventions that will be referenced later. Section **??** defines the vector and matrix definitions used throughout the rest of the document. Section **??** describes k-universal hash families, an important concept for many sketches. Section **??** describes some of the sketching concepts that we will reference throughout this document.

2.1 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 xth 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

$$B_{xy,z} = \begin{cases} 1 & \text{if } xy \in \mathcal{E} \text{ and } x = z \\ -1 & \text{if } xy \in \mathcal{E} \text{ and } y = z \\ 0 & \text{else.} \end{cases} \tag{2.1}$$

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

$$L = BWW^T B^T. \tag{2.2}$$

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.

2.2 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.

2.3 Vector and Matrix definitions and notation

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

$$\|v\|_p = \left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p}. \tag{2.3}$$

As $p \to 0$, this quantity converges to the special case of the ℓ_0 norm:

$$\|v\|_0 = \sum_{i=1}^{n} v_i^0 = |\{i \in [n] \mid v_i \neq 0\}|. \tag{2.4}$$

Here we define $0^0 = 0$. Throughout this document we will mostly be concerned with the ℓ_0 and ℓ_1 norms of matrix rows and columns. The p-th frequency moment F_p of a vector v is related to its ℓ_p norm in the following way:

$$F_p(v) = \|v\|_p^p = \sum_{i=1}^{n} v_i^p. \tag{2.5}$$

We will also sometimes be interested in matrix norms. For $i \in [n]$ and $j \in [m]$, we will write the i, jth element of M as $M_{i,j}$. We will also write the ith row and jth 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:

$$\|M\|_F = \left(\sum_{i=1}^{n} \sum_{j=1}^{m} M_{i,j}^2 \right)^{1/2}. \tag{2.6}$$

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 Σ. 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.

2.4 k-Universal Hash Families

Many critical results in the sketching literature depend on k-universal hash families.

Definition 2.4.1. *A hash family from sets $\mathcal{X}$ to $\mathcal{Y}$ is a set of functions $\mathcal{H}$ such that for all $h \in \mathcal{H}$, $h : \mathcal{X} \to \mathcal{Y}$. Such a family $\mathcal{H}$ is k-universal if for all $x_1, \ldots, x_k \in \mathcal{X}$ and for all $y_1, \ldots, y_k \in \mathcal{Y}$, the following holds:*

$$\Pr_{h \in_R \mathcal{H}} [h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k] = \frac{1}{|\mathcal{Y}|^k}.$$

A k-universal hash family $\mathcal{H}$ has the property that for any collection of $x_1, \ldots, x_k \in \mathcal{X}$, if $h \in_R \mathcal{H}$ then $(h(x_1), \ldots, h(x_k))$ is distributed uniformly in $\mathcal{Y}^k$. This property is of critical importance. If an algorithm requires a k-wise independent uniform projection over elements, then a k-universal hash function suffices. Moreover, so long as $k \ll n$, a sampled hash function is relatively efficient to store.

Such hash functions underly many of the fundamental results enabling sketching algorithms, some of which we discuss in detail in subsequent chapters.

2.5 Approximation Paradigms

Throughout this document we will consider several different approximation problems. For the purpose of discussion, we will consider a nonspecific centrality index $\mathcal{C}$ and a graph $\mathcal{G}$. The general approach to approximate C is to generate a function $\widetilde{\mathcal{C}} : \mathcal{V} \to \mathbb{R}_{\geq 0}$ such that $\widetilde{\mathcal{C}}(x)$ is a "good approximation" of $\mathcal{C}(x)$ for every $x \in \mathcal{V}$. However, in many applications it is not necessary to exhaustively list the centrality for every vertex in the graph. Instead, many applications require only the top k vertices with respect to $\mathcal{C}$, for $k \ll n$. Consequently, we will also be interested in the problem of maintaining approximations of the top k vertices with respect to $\mathcal{C}$.

For $x, y \in \mathbb{R}$, $\varepsilon > 0$, we will use the compact notation $x = (1 \pm \varepsilon)y$ to denote the situation where $(1 - \varepsilon)y \leq x \leq (1 + \varepsilon)y$. We will refer to an algorithm as an (ε, δ)-approximation of quantity Q if it is guaranteed to output $\widetilde{Q}$ such that $\widetilde{Q} = (1 \pm \varepsilon)Q$ with probability at least $1 - \delta$.

2.5.1 Total Centrality Approximation

We will consider the problem ApproxCentral$(\mathcal{C}, \mathcal{G}, \varepsilon)$ as the problem of producing a function $\widetilde{\mathcal{C}_{\mathcal{G}}}$ such that, for all $x \in \mathcal{V}$, $\widetilde{\mathcal{C}_{\mathcal{G}}}(x) = (1 \pm \varepsilon)\mathcal{C}_{\mathcal{G}}(x)$. It is important to note that such strong approximations may not be attainable for some indices in the streaming or even semi-streaming model. In such cases we will be forced to accept unbounded approximations that exhibit good empirical performance. We will call this relaxed problem, with no specified worst case bounds, UBApproxCentral$(\mathcal{C}, \mathcal{G})$.

2.5.2 Top-k Centrality Approximation

It is worth repeating the estimation of the centrality of every vertex in a graph is unnecessary for many applications. Indeed, in many cases a set of the top k central vertices suffices, for a reasonable choice of k. We will consider the problem ApproxTopCentral$(\mathcal{C}, \mathcal{G}, k, \varepsilon)$ as the problem of producing a list $\mathcal{V}_k$ of k vertices such that, if v_{i_k} is the k-th largest index of $\mathcal{V}$ with respect to $\mathcal{C}_{\mathcal{G}}$, then for every $v \in \mathcal{V}_k$, $\mathcal{C}_{\mathcal{G}}(v) \geq (1 - \varepsilon)\mathcal{C}_{\mathcal{G}}(v_{i_k})$. We will similarly relax the requirements to UBApproxTopCentral$(\mathcal{C}, \mathcal{G}, k)$ to allow for unbounded approximations.

Chapter 3

Streaming Degree Centrality

In this chapter we present streaming algorithms for approximating degree centrality. The algorithms are straightforward applications of CountMinSketch to the degree counting problem, and so instead of maintaining $O(n)$ counters we maintain a $\widetilde{O}(1)$ sketch that can be queried for degrees. These algorithms also address sublinear streaming heavy hitter recovery.

3.1 Introduction and Related Work

In an undirected and unweighted graph, the degree centrality of a vertex is simply calculated as the number of adjoining edges in the graph. If the graph is weighted, degree centrality is usually generalized to the sum of the weights of the adjoining edges. In either case, the degree centrality of vertex $x \in [n]$ is equal to the sum of the xth row of the adjacency matrix A. In a directed graph, the indegree (outdegree) centrality of vertex x is the number of incoming (outgoing) edges to (from) x, conventionally corresponding to the xth column (row) of A.

$$\mathcal{C}^{\text{DEG}}(x) = |\{(u,v) \in \mathcal{E} \mid x \in \{u,v\}\}| = \|A_{x,:}\|_1 = \|A_{:,x}\|_1 \tag{3.1}$$

$$\mathcal{C}^{\text{IDEG}}(x) = |\{(u,v) \in \mathcal{E} \mid x = v\}| = \|A_{x,:}\|_1 \tag{3.2}$$

$$\mathcal{C}^{\text{ODEG}}(x) = |\{(u,v) \in \mathcal{E} \mid x = u\}| = \|A_{:,x}\|_1 \tag{3.3}$$

Though simple, degree centrality is still widely used as a benchmark in many applications [BV14]. Indeed, it is competitive with more sophisticated notions of centrality in some contexts [UCH03]. Moreover, degree centrality is known to correlate well with PageRank, making it a decent proxy when computing PageRank is not practical [UCH03]. Moreover, even a naïve streaming implementation of degree centrality is efficient to compute compared to other centrality indices. However, the naïve implementation requires the maintenance of $\Omega(n)$ counters over a graph stream.

We will improve upon this constraint using the famous CountMinSketch, an early and important sketch performing approximate counting [CM05]. CountMinSketch maintains $t = O(\log(1/\delta))$ sets of $r = O(\varepsilon^{-1})$ counters, using 2-universal hash functions $h_1, \ldots, h_t : [n] \to [r]$. These function define matrices $C^{(1)}, \ldots, C^{(t)} \in \mathbb{R}^{r \times n}$ as follows. Initialize $C^{(1)} = \cdots = C^{(t)} = \{0\}^{tr \times n}$. Then, for each $i \in [t]$ and each $j \in [n]$, set $C^{(i)}_{h_i(j),j} = 1$. $C^{(1)}, \ldots, C^{(t)}$ are sparse matrices with 1 nonzero entry in every column. For streaming frequency vector $\mathbf{f} \in \mathbb{R}^n$ let $\mathcal{S}(\mathbf{f}) \in \mathbb{R}^{t \times r}$ be the matrix whose ith column is given by $C^{(i)}\mathbf{f}$. We will drope the parameterization of $\mathcal{S}$ where it is obvious. $\mathcal{S}$ is the CountMinSketch object, and can be computed in $\widetilde{O}(\text{nnz}(\mathbf{f}) \log(1/\delta))$ time.

Theorem 3.1.1 shows that the minimum of $\{\mathcal{S}_{1,h_1(j)}, \mathcal{S}_{2,h_2(j)}, \ldots \mathcal{S}_{t,h_t(j)}\}$ is a biased estimator of $\mathbf{f}_j$ with bounded error. Given a vector $\mathbf{f} \in \mathbb{R}^n$, we use the notation $\mathbf{f}_{-i}$ to denote the vector whose elements are all the same as $\mathbf{f}$ aside from the ith element, which is zero.

Theorem 3.1.1 (Theorem 1 of [CM05])**.** *Let $\mathbf{f} \in \mathbb{R}^n$ be the frequency vector of a strict turnstile stream and let $\mathcal{S}$ be its accumulated* CountMinSketch *with parameters (ε, δ). For all $j \in [n]$ the following holds with*

probability at least $1-\delta$*:*

$$0 \leq \widetilde{\mathbf{f}}_j - \mathbf{f}_j \leq \varepsilon \|\mathbf{f}_{-j}\|_1$$

CountMinSketch guarantees error in terms of the ℓ_1 norm, which is not as tight as the ℓ_2 bounds offered by the more general CountSketch, which has the added benefit of operating on turnstile streams [CCFC02]. However, the CountSketch data structure depends upon a second set of 2-universal hash functions and adds a $1/\varepsilon$ multiplicative factor to the definition of r, as well as involving larger constants [CM05]. All of the results in the following are stated in terms of CountMinSketch as they yield more practical implementations, but similar results could be obtained using CountSketch.

3.2 Streaming Degree Centrality

Given a stream updating adjacency matrix A, it is a simple to interpolate it as a stream updating a vector $\mathbf{d}$ storing the degree of every vertex in the graph. If computing indegree, simply convert an update (x, y, c) to (y, c), where (y, c) is interpreted as "add c to $\mathbf{d}_y$". Outdegree instead converts (x, y, c) to (x, c), and if $\mathcal{G}$ is undirected one applies both (x, c) and (y, c). Here $c \in \{1, -1\}$ in all cases if $\mathcal{G}$ is unweighted, where $c = 1$ implies an edge insertion and $c = -1$ implies an edge deletion. We assume a strict turnstile model where each edge can only be inserted or deleted, possibly more than once. Thus accumulated vector $\mathbf{d}$ is such that $\mathbf{d}_x = \mathcal{C}_{\mathcal{G}}^{\text{DEG}}(x)$.

We can generalize this model to account for weighted $\mathcal{G}$. In this case edge weights could change after insertion, and the vector $\mathbf{d}$ is such that $\mathbf{d}_x = \|A_{:,x}\|_1$ for indegree centrality, or $\mathbf{d}_x = \|A_{x,:}\|_1$ for outdegree centrality. If $\mathcal{G}$ is unweighted, both versions are equivalent. Rather than the number of edges incident upon x, $\mathbf{d}_x$ denotes the amount of weight incident upon x. As the treatment for weighted and unweighted $\mathcal{G}$ is the same, we will not distinguish in the following.

An immediate consequence of this formulation is that one can accumulate CountMinSketch on a strict turnstile graph stream to obtain $\mathcal{S}$ and perform point queries as to the degrees of vertices. However, the error guarantees of Theorem 3.1.1 are tight only for $x \in \mathcal{V}$ where $\mathbf{d}_x \geq \phi \|d\|_1$ for a nontrivial fraction ϕ. It is desirable in particular to compute and return these heavy hitters. While this can be achieved trivially with a second pass, we show that single pass algorithms suffice. In the following we assume that $\mathcal{G}$ in undirected. Simple modifications suffice for directed $\mathcal{G}$.

Cash Register Model. We describe an algorithm similar to the algorithm of Theorem 6 of [CM05]. We maintain a heap H of candidate heavy hitters and $\|\mathbf{d}(t)\|$ for update t. The latter is monotonically increasing because the stream is insert only. Upon receiving (x_t, y_t, c_t) from the graph stream σ, update $\mathcal{S}(t)$ as usual using updates (x_t, c_t) and (y_t, c_t) and then query it for $\widetilde{\mathbf{d}(t)}_x$ and $\widetilde{\mathbf{d}(t)}_y$. If $\widetilde{\mathbf{d}(t)}_x \geq \phi \|\mathbf{d}(t)\|_1$, add $\left(x, \widetilde{\mathbf{d}(t)}_x\right)$ to H, and similarly for y. Let $\left(z, \widetilde{\mathbf{d}(t^*)}_z\right) = \min(H)$ and remove it from H if $\widetilde{\mathbf{d}(t^*)}_z < \phi \|\mathbf{d}(t)\|_1$. Once done reading σ, we iterate through $\left(z, \widetilde{\mathbf{d}}_z\right) \in H$ and using $\mathcal{S}$ output z such that $\widetilde{\mathbf{d}}_z \geq \phi \|d\|_1$.

Theorem 3.2.1. *Let $\mathcal{G}$ with degree vector $\mathbf{d}$ be given in a cash register stream. An algorithm can return every vertex with degree $\geq \phi \|\mathbf{d}\|_1$, and avoid returning any vertices with degree $< (\phi - \varepsilon)\|\mathbf{d}\|_1$ with probability $1 - \delta$. The algorithm requires space $O\left(\varepsilon^{-1} \log \frac{n}{\delta}\right)$ and update time $O\left(\log \frac{n}{\delta}\right)$.*

Proof. Note that $\|\mathbf{d}(t)\|_1$ increases monotonically with t in a cash register stream. If $t < t^*$ and a vertex's estimate is smaller than $\phi \|\mathbf{d}(t)\|_1$, it cannot be larger than $\phi \|\mathbf{d}(t^*)\|_1$ without its estimate due to $\mathcal{S}$ increasing by time t^*. We check the estimate for each vertex when an incident edge is updated and the estimates do not underestimate, so no heavy hitters are omitted.

Call the event where $\widetilde{\mathbf{d}}_x > \phi \|\mathbf{d}_x\|_1 \wedge \mathbf{d}_x < (\phi - \varepsilon)\|\mathbf{d}\|_1$ for some $x \in \mathcal{V}$ a *miss*. If $x \in \mathcal{V}$ participates in a miss, it is falsely output as a heavy hitter by the algorithm. By Theorem 3.1.1, an accumulated

CountMinSketch data structure $\mathcal{S}$ with parameters ε and δ^* guarantees that

$$\begin{aligned}
n\delta^* &> \sum_{x\in\mathcal{V}} \Pr\left[\widetilde{\mathbf{d}}_x - d_x > \varepsilon\|\mathbf{d}_{-x}\|_1\right] \\
&\geq \sum_{x\in\mathcal{V}} \Pr\left[\widetilde{\mathbf{d}}_x - (\phi-\varepsilon)\|\mathbf{d}\|_1 > \varepsilon\|d\|_1 \wedge \mathbf{d}_x < (\phi-\varepsilon)\|\mathbf{d}\|_1\right] \\
&= \sum_{x\in\mathcal{V}} \Pr\left[\widetilde{\mathbf{d}}_x > \phi\|\mathbf{d}\|_1 \wedge \mathbf{d}_x < (\phi-\varepsilon)\|\mathbf{d}\|_1\right] && \text{i.e. sum of misses} \\
&\geq \Pr\left[\bigvee_{x\in\mathcal{V}} \widetilde{\mathbf{d}}_x > \phi\|\mathbf{d}\|_1 \wedge \mathbf{d}_x < (\phi-\varepsilon)\|\mathbf{d}\|_1\right]. && \text{Union bound}
\end{aligned}$$

We have shown that the probability that *any* miss occurs is less than $n\delta^*$. By setting $\delta^* = \frac{\delta}{n}$ we guarantee that a miss occurs with probability at most δ, obtaining our desired result. This also fixes the space and update time complexities. □

Strict Turnstile Model. The presence of negative updates necessitates a more complex algorithm, as $\|\mathbf{d}(t)\|_1$ is no longer monotonic. We describe an algorithm similar to Theorem 7 of [CM05]. Assume that n is a power of two for convenience of notation. For $j \in \{0, 1, \ldots, \log n\}$ we maintain a CountMinSketch sketch $\mathcal{S}^{(j)}$. For each degree update (x, c), apply the update $\left(\left\lfloor \frac{x}{2^j} \right\rfloor, c\right)$ to $\mathcal{S}^{(j)}$ for each j. Note that $\mathcal{S}^{(j)}$ receives at most $2^{\log n - j}$ distinct elements, which correspond to the dyadic ranges $\{1, \ldots, 2^j\}$, $\{2^j + 1, \ldots, 2 \cdot 2^j\}$, $\ldots$, $\{(2^{\log n - j} - 1)2^j, \ldots, 2^{\log n - j} \cdot 2^j\}$. Moreover the sketches form a hierarchy, where the range elements of $\mathcal{S}^{(j)}$ subdivide those of $\mathcal{S}^{(j+1)}$, forming a natural binary search structure. This structure will allow us to query for heavy hitters.

Algorithm 1 Strict Turnstile Degree Centrality Heavy Hitters

Input: σ - stream of edge updates
ϕ - threshold
ε, δ - approximation parameters

Accumulation:

for $j \in \{0, 1, \ldots, \log n\}$ **do**
  $\mathcal{S}^{(j)} \leftarrow$ empty independent CountMinSketch(ε, δ^*)
$D \leftarrow 0$
for $(x, y, c) \in \sigma$ **do**
  $D \leftarrow D + 2c$
  for $j \in \{0, 1, \ldots, \log n\}$ **do**
    Insert $\left(\left\lfloor \frac{x}{2^j} \right\rfloor, c\right)$ and $\left(\left\lfloor \frac{y}{2^j} \right\rfloor, c\right)$ into $\mathcal{S}^{(j)}$

Query:

return AdaptiveSearch$(\log n, 0, (\phi + \varepsilon)D)$

Functions:

function AdaptiveSearch$(j, r, thresh)$
  $\widetilde{d}_r \leftarrow$ query $\mathcal{S}^{(j)}$ for r
  if $\widetilde{d}_r \geq thresh$ **then**
    if $j = 0$ **then**
      return $\left(r, \widetilde{d}_r\right)$
    else
      AdaptiveSearch$(j - 1, 2r, thresh)$
      AdaptiveSearch$(j - 1, 2r + 1, thresh)$

Algorithm 1 summarizes this approach. We perform a recursive binary search, starting with $\mathcal{S}^{\log n}$. For the dyadic range sums at layer j that are greater than $(\phi+\varepsilon)\|\mathbf{d}\|_1$, we query the partitioned halves of the dyadic ranges at layer $j-1$, until we obtain the heavy hitters at layer 0. If a queried range sum is below the threshold, it is discarded. Elements are only returned when a recursive chain of queries makes it all the way to layer 0, which is a normal CountMinSketch over the stream.

Theorem 3.2.2. *Let $\mathcal{G}$ with degree vector $\mathbf{d}$ be given in a strict turnstile stream. Algorithm 1 can return every vertex with degree $\geq (\phi+\varepsilon)\|\mathbf{d}\|_1$, and avoid returning any vertices with degree $< \phi\|\mathbf{d}\|_1$ with probability $1-\delta$. The algorithm requires space $O\left(\varepsilon^{-1}\log n \log\frac{2\log n}{\phi\delta}\right)$ and update time $O\left(\log n\log\frac{2\log n}{\phi\delta}\right)$.*

Proof. First note that for each $x\in\mathcal{V}$ where $\mathbf{d}_x \geq (\phi+\varepsilon)\|d\|_1$, each dyadic range of which it is a part must have a sum no less than $(\phi+\varepsilon)\|\mathbf{d}\|_1$. Consequently, Algorithm 1 must return this vertex as CountSketches cannot underestimate any of these sums.

Call the event where $\widetilde{\mathbf{d}}_x > (\phi+\varepsilon)\|\mathbf{d}_x\|_1 \wedge \mathbf{d}_x < \phi\|\mathbf{d}\|_1$ for some $x\in\mathcal{V}$ a *miss*. Again, if $x\in\mathcal{V}$ participates in a miss Algorithm 1 falsely reports it as a heavy hitter. Clearly, for each $j\in\{0,1,\ldots,\log n\}$ there are at most $1/\phi$ true range sums greater than $\phi\|\mathbf{d}\|_1$. Consequently the algorithm makes at most twice this number of queries at any particular layer, assuming there are no misses. This makes for a maximum of $\frac{2\log n}{\phi}$ queries in total. The sketches are independent, so these queries all have the same probability of failure δ^*. Assume that $\widetilde{\mathbf{d}}_x^{(j)}$ is the queried output of some range x from $\mathcal{S}^{(j)}$ in a run of Algorithm 1, and let $Q=\{(x,j)\mid \mathcal{S}^{(j)} \text{ is queried for } x\}$. Recall from the above that $|Q|=\frac{2\log n}{\phi}$. By Theorem 3.1.1, the CountMinSketch data structures guarantee that

$$
\begin{aligned}
\frac{2\log n}{\phi}\delta^* &> \sum_{(x,j)\in Q}\Pr\left[\widetilde{\mathbf{d}}_x^{(j)} - d_x > \varepsilon\|\mathbf{d}_{-x}\|_1\right] \\
&\geq \sum_{(x,j)\in Q}\Pr\left[\widetilde{\mathbf{d}}_x^{(j)} - \phi\|\mathbf{d}\|_1 > \varepsilon\|\mathbf{d}\|_1 \wedge \mathbf{d}_x < \phi\|\mathbf{d}\|_1\right] \\
&= \sum_{(x,j)\in Q}\Pr\left[\widetilde{\mathbf{d}}_x^{(j)} > (\phi+\varepsilon)\|\mathbf{d}\|_1 \wedge \mathbf{d}_x < \phi\|\mathbf{d}\|_1\right] && \text{i.e. sum of misses} \\
&\geq \Pr\left[\bigvee_{(x,j)\in Q}\widetilde{\mathbf{d}}_x^{(j)} > (\phi+\varepsilon)\|\mathbf{d}\|_1 \wedge \mathbf{d}_x < \phi\|\mathbf{d}\|_1\right]. && \text{Union bound}
\end{aligned}
$$

We have shown that the probability that *any* miss occurs is less than $\frac{2\log n}{\phi}\delta^*$. By setting $\delta^* = \frac{\phi\delta}{2\log n}$ we guarantee that a miss occurs with probability at most δ, obtaining our desired result. This also fixes the space and update time complexities. □

Chapter 4

Semi-Streaming Closeness Centrality

In this chapter we present a $O(1)$-pass semi-streaming algorithm for the approximation of closeness centrality over strict turnstile streams. The algorithm draws upon recent advances in graph sparsification using ℓ_p sampling sketches. We are unable to gain asymptotic improvements by only returning the heavy hitters, as writing the sketches down requires superlinear memory in n. However, it is trivial to maintain a heap of the heavy hitters during computation.

4.1 Introduction and Related Work

Closeness centrality is a commonly used centrality measure in the analysis of networks [Bav48, WF94, BV14, CDPW14, WC14]. Closeness centrality, itself an early measure of graph centrality, defines centrality of a vertex in terms of the reciprocal of the vertex's distance to the graph's other vertices [BV14]. Unlike degree centrality, which is based on vertex-local measurements, the closeness centrality depends on the full path structure of $\mathcal{G}$. Consequently, a logarithmic amount of memory appears unlikely to suffice.

A vertex with high closeness can be thought of as one that can communicate with the other vertices in the graph relatively quickly. Put precisely, one calculates the closeness centrality of a vertex x as

$$\mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x) = \frac{1}{\sum\limits_{y \in \mathcal{V} \setminus \{x\}} d(x,y)}. \tag{4.1}$$

Consequently, an implementation of exact closeness centrality requires one to solve the AllSourcesShortestPaths problem, making it expensive to compute on a large dense graph. Consequently, on a large evolving graph maintaining up-to-date measures of closeness centrality requires a more sophisticated approach. It is worth pointing out that Eq. (4.1) is not defined if the underlying graph is not strongly connected, and indeed it was probably not intended for such graphs [BV14]. While there are generalizations to the definition that account for disconnected graphs, their details are out of the scope of this document .

Computing the closeness centrality of an evolving graph is an interesting problem, but one that is eclipsed by the attention paid to its cousin betweenness centrality. We discuss some of this literature in Section **??**. However, online solutions have been suggested in the literature [WC14]. In particular, given the information required to solve betweenness centrality for all vertices one can also solve the closeness centrality. Thus, it might be expected that a solution for betweenness centrality can be adapted into a solution for closeness centrality. However, computing closeness centrality is strictly easier than betweenness centrality, so some solutions that work for closeness centrality in pass- or space-constrained settings may not generalize to betweenness centrality.

Cohen et al. describe the best known approximation algorithm for $\mathcal{C}^{\textsc{Cloes}}$ in terms of speed, producing approximations in nearly linear time [CDPW14]. Like many approximation algorithms for path-based centrality indices, their algorithm depends upon solving SingleSourceShortestPaths for a set of $k = O(\log n)$ uniformly sampled source vertices, $C \subseteq \mathcal{V}$. For $x \in \mathcal{V}$, we can estimate closeness centrality using $\widetilde{\mathcal{C}}^{\textsc{Close}}(x) = \sum_{y \in C \setminus \{x\}} \frac{d(x,y)}{k}$. However, this estimate can have high variance. In order to improve error

bounds, Cohen et al. use the *pivot* of v for v that are "far" from C , $c(v) = \arg\min_{u \in C} d(uv)$ to estimate $\mathcal{C}^{\text{CLOSE}}(v)$, as $\widetilde{\mathcal{C}}^{\text{CLOSE}}(v) \coloneqq \widetilde{\mathcal{C}}^{\text{CLOSE}}(c(v)) + d(u, c(v))$. Their algorithm satisfies the following theorem:

Theorem 4.1.1. *(Theorem 2.1 of [CDPW14]). There is an algorithm (Algorithm 1 of [CDPW14]) that produces, for every $x \in \mathcal{V}$, an approximation of closeness centrality satisfying $|\mathcal{C}^{Close}(x) - \widetilde{\mathcal{C}}^{Close}(x)| \leq O(\varepsilon)\mathcal{C}^{Close}(x)$ with high probability. This algorithm requires $O((m + n \log n) \log \frac{n}{\varepsilon^3})$ time and $O(m)$ memory.*

We will also utilize graph sparsification methods stemming from work by Ahn, Guha and McGregor to produce spanning subgraphs using semi-streaming methods. Section 4.1.1 describes ℓ_p sampling sketches, while Section 4.1.2 discusses their application to constructing spanning subgraphs.

4.1.1 ℓ_p Sampling Sketches

First, we define a ℓ_p-sampling sketch on vector $v \in \mathbb{R}^n$ as follows.

Definition 4.1.1. *Let Π be a distribution on real $r \times n$ matrices, where $r = \text{poly}(1/\varepsilon, \log(1/\delta))$. Suppose $\mathbf{f} \in \mathbb{R}^n$ is a streaming frequency vector and sample $S \sim \Pi$. Suppose further that there is a procedure that, given $S\frac{2 \log n}{\phi}\delta^*$, can output (i, P) where $i \in [n]$ is an index of $\mathbf{f}$ sampled with probability $P = (1 \pm \varepsilon)\frac{|\mathbf{f}_i|^p}{\|\mathbf{f}\|_p^p}$, where $p \geq 0$, failing with probability at most δ. Then Π is an ℓ_p sampling sketch distribution, $S \sim \Pi$ is an ℓ_p sketch transform, and $S\mathbf{f}$ is an ℓ_p sampling sketch.*

Monemizadeh and Woodruff proved the existence of such sampling sketches and provided upper bounds for $p \in [0, 2]$ [MW10]. Jowhari et al. [JST11] improved upon these space bounds. In particular, they proved that ℓ_0 and ℓ_1 sampling sketches require $O(\log^2 n \log \delta^{-1})$ and $O(\varepsilon^{-1} \log \varepsilon^{-1} \log^2 n \log \delta^{-1})$ space, respectively. Note that both have only a polylogarithmic dependence on n, and ℓ_0-sampling sketches can achieve arbitrary sampling precision at no additional cost. Further note that we make use of the convention that $0^0 = 0$ in Definition 4.1.1. Vu improves these sampling sketches with algorithms that allow the simultaneous accumulation of s sampling sketches for a single vector $\mathbf{f}$ in the same asymptotic update times, i.e. without dependence upon s so long as $s = o(\|\mathbf{f}\|_0)$ [Vu18].

The full details are verbose and out of the scope of this document, as we will be using ℓ_p samplers as black box building blocks. It is important to note that as these sampling sketches are linear, they can be added together. That is, if we have two vectors of the same length $\mathbf{f}, \mathbf{f}' \in \mathbb{R}^n$ and S is a sketch matrix drawn from such a sampling distribution, then $S\mathbf{f} + S\mathbf{f}' = S(\mathbf{f} + \mathbf{f}')$. This observation is key for many sublinear graph sparsification algorithms, such as the spanner construction discussed in Section 4.1.2.

4.1.2 ℓ_p-Sampling Graph Sparsification

ℓ_p sampling sketches, particularly of the ℓ_0 and ℓ_1 variety, have proven popular for solving various problems on dynamic graph streams under semi-streaming memory constraints. Applications include testing connectivity and bipartiteness, approximating the weight of the minimum spanning tree in one pass and in multiple passes computing sparsifiers, the exact minimum spanning tree, and approximating the maximum weight matching [AGM12a], as well as computing in several passes cut sparsifiers, approximate subgraph pattern counting, and sparsifying subgraphs [AGM12b], spectral sparsification [AGM13], and identifying densest subgraphs [Vu18].

We briefly summarize the general approach taken in the literature. Consider the columns of the vertex-edge incidence matrix B (see Equation (2.1)), and a series of ℓ_0 sampling sketches $S_1, S_2, \ldots, S_t$, for some $t = O(\log n)$. First, read the stream defining B (which can be adapted from a stream defining A) and sketch each column of B using each of $S_1, S_2, \ldots, S_t$. If $x \in [n]$ is a vertex of $\mathcal{G}$, then $B_{:,x}$ is a vector whose nonzero entries correspond to edges of which it is an endpoint. Thus, $S_1(B_{:,x})$ can recover a uniformly sampled neighbor of x, say y, with high probability. Then, $S_2(B_{:,x}) + S_2(B_{:,y}) = S_2(B_{:,x} + B_{:,y})$ can sample a neighbor of the supervertex $(x + y)$, since the row values indexed by the edge (x, y) are cancelled out when adding the two row vectors by the definition of B. New sketches are required for each contraction, as otherwise the samples will not be independent and the guarantees of the sampling fails. The sparisification of the graph so obtained may then be used, possibly over several passes, to learn nontrivial information about G using a semi-streaming algorithm. Algorithms using ℓ_1-sampling sketches are similar.

For the purposes of this chapter we are interested in their construction of sparse graph spanners as discussed in [AGM12b].

Definition 4.1.2. *An α-spanner of a graph $\mathcal{G}$ is a sparse subgraph $\mathcal{H}$ of $\mathcal{G}$ such that for all $x, y \in \mathcal{V}$, the following holds:*

$$d_{\mathcal{H}}(x, y) \leq \alpha d_{\mathcal{G}}(x, y). \tag{4.2}$$

If one is able to construct such a spanner $\mathcal{H}$ using small memory in a small number of passes over $\mathcal{G}$, then one can estimate vertex-vertex distances on $\mathcal{G}$ in the semi-streaming model. Unfortunately, there is presently no known algorithm that can do so in a single pass. The following Theorem summarizes the result.

Theorem 4.1.2 (Theorem 5.1 of [AGM12b])**.** *Given an unweighted graph $\mathcal{G}$, there is a randomized algorithm that constructs a $(k^{\log 5} - 1)$-spanner in $\log k$ passes using $\widetilde{O}(n^{1+1/k})$ space.*

The rough idea of the algorithm is as follows. Over the course of $\log k$ passes, iteratively contract $\mathcal{G}$ by cleverly choosing vertices from whom to sample ℓ_0 neighbors. Let $\widetilde{\mathcal{G}}_0 = \mathcal{G}$. In pass i, contract graph $\widetilde{\mathcal{G}}_{i-1}$ by partitioning its vertex set into $\widetilde{O}(n^{2^i/k})$ subsets, using an ℓ_0 sampling sketch for each partition. These sampled edges give us a graph $\mathcal{H}_i$. Finally, perform a clever clusting procedure to collapse sampled vertices in the neighborhood of a high degree vertex into a single supervertex. This compression defines $\widetilde{\mathcal{G}}_i$. In every pass, we ensure that the graph is compressed by a certain amount, ensuring favorable memory use. For full details, see [AGM12b].

4.2 Semi-Streaming Constant-Pass Closeness Centrality

A natural avenue of inquiry is whether the ℓ_p-norm sampling approach discussed in Section 4.1.2 can be applied to approximating centrality. Theorem 4.1.2 provides an algorithm for computing a $(k^{\log 5}-1)$-spanner using $\widetilde{O}(n^{1+1/k})$ space taking $\log k$ passes over a stream defining B [AGM12b]. This is of particular note for approximating closeness centrality, as it is defined in terms of shortest path distances.

An α-spanner $\mathcal{H}$ of $\mathcal{G}$ can then be used to approximate the closeness centrality of the vertices in $\mathcal{G}$. Discounting the time spent constructing the sketched spanner, the time needed to compute $\mathcal{C}_{\mathcal{H}}^{\textsc{Close}}(x)$ (Equation (4.1)) will be substantially less than the time required to compute $\mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x)$, particularly if $\mathcal{G}$ is dense.

Though this approximation of closeness centrality is a fairly obvious application of Ahn, Guha, and McGregor's work, to our knowledge it has not be formulated in prior literature. We prove the following lemma:

Lemma 4.2.1. *Let $\mathcal{H}$ be an α-spanner of $\mathcal{G}$. Then for all $x \in \mathcal{V}$,*

$$\mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x) - \mathcal{C}_{\mathcal{H}}^{\textsc{Close}}(x) \leq \left(1 - \frac{1}{\alpha}\right) \mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x). \tag{4.3}$$

Proof. Note that if $\mathcal{H}$ is an α-spanner of $\mathcal{G}$, then for any $x \in \mathcal{V}$, the following holds:

$$\begin{aligned}
\mathcal{C}_{\mathcal{H}}^{\textsc{Close}}(x) &= \frac{1}{\sum_{y \in \mathcal{V} \setminus \{x\}} d_{\mathcal{H}}(x, y)} && \text{Eq. (4.1)} \\
&\geq \frac{1}{\sum_{y \in \mathcal{V} \setminus \{x\}} \alpha d_{\mathcal{G}}(x, y)} && \text{Eq. (4.2)} \\
&= \frac{1}{\alpha} \mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x).
\end{aligned}$$

Furthermore, an α-spanner cannot underestimate distances by construction, as it only prunes edges. This implies Eq. (4.3). □

Lemma 4.2.1 immediately implies that the algorithm corresponding to Theorem 4.1.2 can be used to approximate the closeness centrality of $\mathcal{G}$ up to a multiplicative factor of $\left(1 - \frac{1}{k^{\log 5} - 1}\right)$. As k increases, the guaranteed bound becomes less tight, but the space complexity of $\mathcal{H}$ improves. As the spanner and analysis applies for both weighted and unweighted graphs, we have the following Theorem.

Theorem 4.2.2. *Let $\mathcal{H}$ be the subgraph spanner of $\mathcal{G}$ output by the algorithm corresponding to Theorem 4.1.2. Then for all $x \in \mathcal{V}$*

$$\mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x) - \mathcal{C}_{\mathcal{H}}^{\textsc{Close}}(x) \leq \left(1 - \frac{1}{k^{\log 5} - 1}\right) \mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x). \tag{4.4}$$

The implicit algorithm uses $\log k$ passes over $\mathcal{G}$ and $\widetilde{O}(n^{1+1/k})$ space. Computing $\mathcal{C}_{\mathcal{H}}^{\textsc{Close}}(\mathcal{V})$ requires $\widetilde{O}(n^{2+1/k})$ time.

Theorem 4.2.2 yields an approximation algorithm, although one that is still rather expensive in the output stage, as we must compute closeness centrality on $\mathcal{H}$. At the expense of possible additional error, however, we can improve the computational cost of this output by utilizing a closeness centrality approximation algorithm, such as that of Theorem 4.1.1. The application thereof is straightforward, and yields the following Theorem where we trade off precision for decreased computation time.

Theorem 4.2.3. *Let $\mathcal{H}$ be the subgraph spanner of $\mathcal{G}$ output by the algorithm corresponding to Theorem 4.1.2. For $x \in \mathcal{V}$, let $\widetilde{\mathcal{C}}_{\mathcal{H}}^{\textsc{Close}}(x)$ be the approximation output by the algorithm of Theorem 4.1.1. Then for all $x \in \mathcal{V}$,*

$$\mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x) - \widetilde{\mathcal{C}}_{\mathcal{H}}^{\textsc{Close}}(x) \leq \left(1 - \frac{1 - O(\varepsilon)}{k^{\log 5}}\right) \mathcal{C}_{\mathcal{G}}^{\textsc{Close}}(x). \tag{4.5}$$

The implicit algorithm uses $\log k$ passes over $\mathcal{G}$ and $\widetilde{O}(n^{1+1/k})$ space. Computing $\widetilde{\mathcal{C}}_{\mathcal{H}}^{\textsc{Close}}(\mathcal{V})$ requires $\widetilde{O}\left(\left(n^{1+1/k} + n \log n\right) \log \frac{r}{\varepsilon}\right.$ time.

Chapter 5

Pseudo-Asynchronous Communication for Vertex-Centric Distributed Algorithms

In this chapter we present general-purpose communication protocols for the vertex-centric distributed graph algorithms. The subject matter in this chapter is much more applied than the rest of this document, and is principally concerned with practical communication management for distributed codes expecting a skewed distribution of work across processors. In addition to motivating and describing the protocols, we discuss our implementation and provide empirical justification.

5.1 Introduction and Related Work

5.1.1 Graph Partitioning

Due to the structured nature of graphs, partitioning them across a universe of processors $\mathcal{P}$ presents challenges not present in many other data structures. A *vertex-centric* or sometimes 1D partitioning is the most natural approach. In such a model the information local to each vertex $x \in \mathcal{V}$, e.g. its sparse adjacency vector $A_{:,x}$, is assigned to a specific processor, e.g. [MAB+10]. A vertex-centric partitioning of $\mathcal{G}$ assumes a partitioning function $f : \mathcal{V} \to \mathcal{P}$. However, unlike many other common data structures in HPC applications, many natural graphs are scale-free, i.e. the vertex degree distribution asymptotically follows the power law distribution. Consequently, there is typically a lot of skewness present in the distribution of information size across $\mathcal{V}$, where high degree matrices or *hubs* account for an outsized amount of partition memory. A paritioning f might overwhelm the memory of some $P \in \mathcal{P}$, or incur significant communication overhead during overehead. Consequently, load-balancing f is an active area of research with a rich history. See [MWM15]. and [BMS+16] for good surveys of this literature.

Sub-vertex centric or 2D partitioning is an alternative scheme where the information in A is instead partitioned into submatrices in a checkerboard pattern. 2D partioning has been applied successfully to several HPC problems [BM11, YBP+11, SKW+14]. However, such schemes are susceptible to hypersparsity in some partitions, wherein a processor may be assigned more vertices than edges [BG08]. This results in poor scaling in practice. Some approaches avoid this by joining 1D and 2D partitioning schemes [BDR13].

Still another approach to handling hubs is the use of vertex delegates, wherein a 1D partitioning is augmented by subpartitioning hubs over a certain size [PGA14]. In this way the computational load and some of the communication load of hubs is shared, at the expense of some additional communication between the *controller* processor and the *delegate* processors when solving problems.

In addition to all of this literature, streaming and semi-streaming algorithms have been proposed for graph partitioning [SK12, TGRV14, XCC+15]. [GHC+17] gives an overview and experimental comparison of many of these technologies. These algorithms seek to limit the time cost of optimizing a graph partition, which has been noted as approaching or eclipsing that of subsequent computation.

We will limit our analysis in this work to vertex-centric partitioning, which we will assume has already been performed. We note that the communication protocols in this chapter can be extended to sub-vertex-centric or hybrid partitioning as well, at the cost of a reduction in legibility. As we are most concerned with optimizing bandwidth utilization, we will also not go into great detail about the partitioning functions f, and assume that it is given. We consider the optimization and computation of f as important but out of scope.

5.1.2 Synchronous and Asynchronous Communication

As many graph algorithms are pathing-centric, *graph traversal* is a first class function of any reasonable graph processing library. This results in hubs costing much more time than low-degree vertices in practice. In many implementations of graph algorithms, even using optimized load-balancing on f, there is an unequal distribution of computation and communication labor across $\mathcal{P}$.

This unequal distribution of labor poses problems for handling communication that do not arise in many other HPC applications where there is a roughly symmetric amount of computation on each processor. Consider, for example, the famous Google Pregel framework [MAB+10]. Pregel was the first proposed vertex-centric processing frameworks, where communication is handled in a series of synchronous communication rounds. Others have noted that this synchronicity results in poor performance in practice, as processing moves at the speed of the slowest processor [PGA14, JPNR17]. In some pathological cases synchronous implementations of algorithms are entirely unable to complete in reasonable amounts of time compared to asynchronous implementations [JPNR17]. Consequently, not only is bandwidth not utilized during processing rounds, but many processors may remain idle while waiting for communication.

A different approach uses fully asynchronous communication, wherein processors communicate with one another in a point-to-point fashion as required by the algorithm. This approach potentially avoids the processor underutilization problem present in Pregel-like systems. However, in typical graph traversals message sizes are small. For instance, computing a random walk generally requires only the path so far to be forwarded to the next processor.

It is conventional wisdom within the practice of HPC to avoid codes that transmit many small messages. This is because each message requires a constant amount of information be transmitted in the form of headers, and also incurs a computational cost at the sender and receiver to handle transmission and receipt. This cost is usually invisible to the application writer, and is incurred within a message handling service such as MPI - Message Passing Interface, the de facto standard HPC communication protocol. If messages contain a small amount of information relative to this overhead, then bandwidth utilization is said to be poor.

To summarize, it is desirable to batch messages for most graph algorithms, as they tend to be small. For synchronous, Pregel-like systems, this is simple as all messages are sent within rounds. Messages with matching sources and destinations can be simply packaged together at transmission. Asynchronous systems are much more complex, as the designer must make decisions balancing message size against delivery promptness. These decisions are application specific, as the needs of a SingleSourceAllShortestPaths computation are different than those of, say, second neighborhood size estimation or random walk sampling. An optimized fully asynchronous communication scheme demands significant designer overhead and may not be generalizable. Consequently, these is a trade-off between synchronous and asynchronous communication, both leaving something to be desired.

5.2 Pseudo-Asynchronous Communication Protocols

In this chapter we discuss pseudo-asynchronous communication protocols - a middle ground between synchronous and asynchronous protocols. These protocols attempt to join the desirable features of synchronous communication - batching of messages, pushback against prolific senders - with the desirable features of asynchronous communication - processors can enter and leave communication context when ready - while retaining generality, easy of use and ease of maintenance.

We describe messaging protocols that provide a mailbox abstraction. Algorithm implementations place messages in a mailbox during normal computation and continue with local tasks. Once a mailbox reaches

a threshold size, the processor enters communication context and begins the process of communicating - whether or not its communication partners have entered communication context themselves. The algorithm designer can also manually enter communication context. Once the mailbox receives confirmation that it is to receive no more communications in the current exchange, it drops out of the communication context and returns to the local algorithm computation. An algorithm implementation need only specify when to send messages and the behavior upon receipt, and may otherwise be agnostic to the communication. We also ensure standard MPI guarantees - such as that messages arrive in the order they are sent.

We discuss three related routing protocols - *Node Local*, *Node Remote*, and *Node Local Node Remote* or *NLNR*. Each of these variants take advantage of *local* and *remote* routing in hybrid distributed memory systems. We use the term *local* to refer to communications wherein the source and destination processors exist on different processors (cores) on the same compute node. We meanwhile use *remote* to refer to the converse situation, where the endpoints of a message exist on different nodes.

The key insight to this analysis is that remote communication requires the transmission of messages over a wire, whereas local communication is handled in shared memory of a single machine. The former is generally more costly. Consequently, we seek to minimize the number of discrete remote messages and channels. Aggregating messages improves bandwidth utilization and reduces overhead, while partitioning routing into channels increases asynchronicity.

Throughout, we assume that there are N compute nodes participating in a hybrid distributed memory instance. We identify each node with an offset in $[N]$. Here we use the notation $[z] = \{1, 2, \ldots, z\}$ for $z \in \mathbb{Z}_+$. We will further assume that each participating node holds the same number of cores C, similarly identified with an offset in $[C]$. We address a processor identified with the cth core on the nth node by the tuple $(n, c) \in [N] \times [C]$. We call c the processor's *core offset* and n the processors *node offset*. We will refer to the universe of $N \times C$ processors as $\mathcal{P}$.

We assume without loss of generality that N is a multiple of C. We will refer to partitions of $[N]$ into the C-sized chunks $\{1, \ldots, C\}$, $\{C+1, \ldots, 2C\}$, $\ldots$ $\{N-(C-1), \ldots, N\}$ as the *layers* of the instance. Assume that there are $L = N/C$ such layers. We will sometimes sub-address a core c on the nth node in layer l using the tuple $(l, n, c) \in [L] \times [C] \times [C]$. In this case, we call l the processor's *layer* and n the processor's *layer offset*. The same core can be addressed using $((l-1)C + n, c) \in [N] \times [C]$, as above.

Each protocol proceed in a series of *exchanges*. An exchange consists of a subset of processes passing messages between themselves. All messages due to each destination core, as well as any messages routed through said core, are aggretated into a single message and transmitted. Each member of an exchange may be responsible for forwarding communication in a later exchange. We call these forwarding processes intermediaries. At the end of an exchange phase, we assume that each process holds all outbound messages intended either for it or for one of the exterior processes for which it is an intermediary. Exchanges are triggered upon entering a communication context in a distributed program. This occurs either due to reaching a maximum number of batched messages, or is manually flushed by the high level program. In this document we consider two types of exchanges: *local* and *remote*.

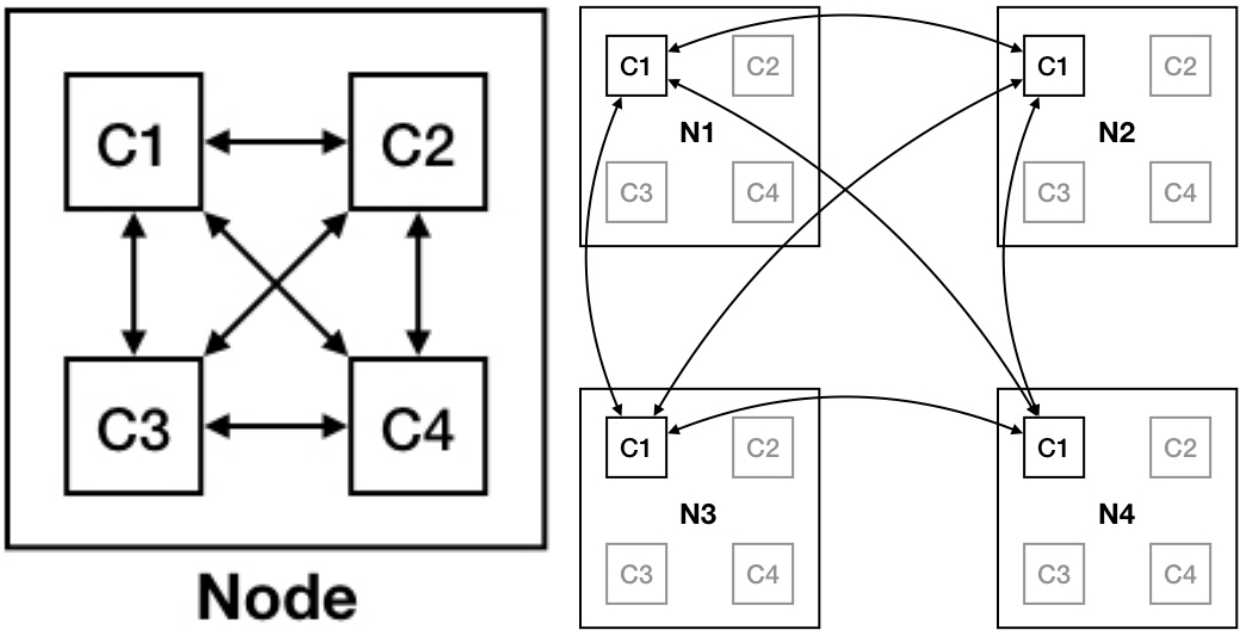
A local exchange consists of all of the processes on a single compute node, and occurs in shared memory. There are N such exchanges. Figure 5.1a illustrates a local exchange where $C = 4$. A particular exchange involves the transmission of at most $\binom{C}{2}$ messages. An implementation might utilize MPI, or instead handle the information exchange directly in shared memory.

A simple remote exchange consists of all of the processors that share a given core offset, and involves remote communication. There are C such exchanges. Figure 5.1b illustrates a remote exchange for core offset 1 where $N = C = 4$. A particular remote exchange involves the transmission of at most $\binom{N}{2}$ messages. An implementation must utilize MPI or a similar remote message passing protocol.

Node Local

We call the protocol consisting of a local exchange on each node, followed by C remote exchanges that occur in parallel *Node Local*. At the beginning of the local exchange, the process on core (n, c) holds a set of messages to be transmitted. Each message with destination (n', c') is forwarded to (n, c') in a local exchange, unless $c' = c$, in which case the process holds onto the message. At the end of this local exchange, each core (n, c) holds messages with addresses of the form (n', c). If $n' = n$, then the message has arrived at its destination and is processed. Otherwise, it is to be communication in a subsequent remote exchange.

Once the processor on (n, c) completes its local communication phase, it enters the program context for



(a) Local exchange

(b) Remote exchange

Figure 5.1: Local and remote exchange diagrams where $N = C = 4$.

a remote exchange. This remote exchange consists of all of the cores with local offset c, each of which also hold, or will hold, messages bound for some participant in the exchange. Once a process has sent all messages routed through it and received all messages sent to it during this "round" of communication, it is finished. It can safely move on to a different program context, even if others are still working.

In total, the node local protocol consists of N local exchanges and C remote exchanges, which occur in parallel. All messages destined for a particular remote process are accumulated at a single intermediary at each node prior to remote transmission, saving on remote overhead.

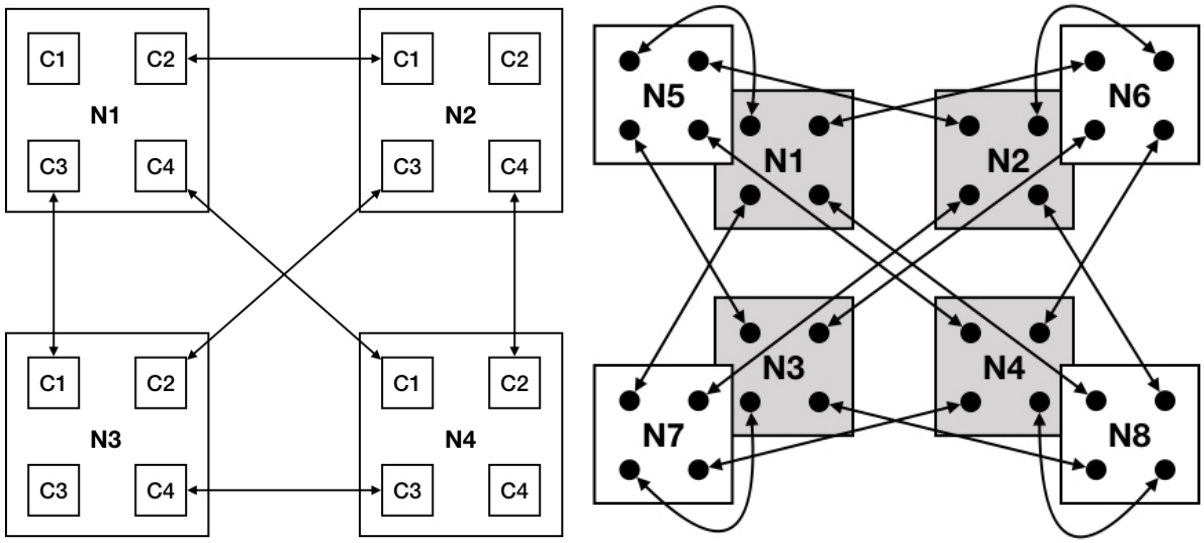
While this local aggregate, remote send policy appears adequate for handling point-to-point messages, it is not robust to one-to-many broadcasts. Such a broadcast results in a total of NC remote messages, which can clearly be improved.

Node Remote

Consider the reverse protocol, which we call *Node Remote*. That is, each process participates in a remote exchange with all cores matching its core offset in the first round of communication, followed by a local exchange at each node in the second. For each message held by the process at (n, c) with destination (n', c'), the process forwards the message to (n', c) in the remote exchange. Once all remote exchanges have completed, each message is held on a node matching its destination node offset. A local exchange in the second phase ensures that each message arrives at its destination core.

Whereas the node local protocol accumulates all messages to a particular process in a single intermediary before remote transmission, the node remote protocol instead forwards all messages from a particular process destined for the same *node*, allowing for a similar bundling of messages in shared memory. If the distribution over sender-receiver pairs is roughly uniform in an application, then the two protocols should exhibit similar performance. However, node remote performs much better in the presence of a large number of broadcasts. In node remote, a broadcast generates only $N - 1$ remote messages. This means that Node Remote uses less bandwidth per broadcast than Node Local, at a gain of $O\left(\frac{1}{C}\right)$. The broadcasting work is pushed onto the (typically much faster) shared memory local exchanges.

NLNR



(a) Intra-layer communication

(b) Inter-layer communication

Figure 5.2: Intra- and inter-layer exchange diagrams where $N = 8$ and $C = 4$.

Both the Node Local and Node Remote protocols exhibit some weaknesses depending upon the distribution of message recipients. Moreover, each round of each protocol results in $O\left(N^2\right)$ remote messages, as each processor has $N - 1$ possible remote communication partners. These messages are sent along C parallel communication channels, each including N participating processors.

The final protocol we discuss in this document improves upon both the messages per round as well as the communication channel size. The NLNR protocol reduces the number of remote communication channels to the theoretical minimum, while still allowing each node to communicate directly with every other node by eliminating redundancy. Consider that a message originating at node n_1 might be transmitted along any of C different remote channels to node n_2 using node local or node remote. In NLNR, there is only one channel connecting each such pair of nodes.

In order to facilitate this reduction in channels, the protocol need occur in three stages: an initial local exchange, a more complex remote exchange, and a final local exchange. This more complex remote exchange can be described by taking a clique connecting each node to every other node and assigning to each edge a single core on each incident node. Figure 5.2 illustrates an example remote exchange where $N = 8$, $C = 4$ and $L = 2$.

It is helpful to visualize the topology of the remote exchange to explain the whole process. We must ensure that each node has an intermediary core responsible for each remote node, so that when viewed as a graph the topology of any subset of nodes and their communication channels forms a clique. This constraint leads to a natural "layering" of the nodes, where each notional layer consists of C nodes. For convenience, in addition to their node offset $n \in [N]$, we assign to each node a *layer offset* $\ell = n \mod C$. Further, we enforce the rule that $(n, c) \in [N] \times [C]$ is an intermediary for all cores on node n', where $c = n' \mod C$ with corresponding intermediaries (n', c') where $c' = n \mod C$. Fig. 5.3 depicts such a topology for a single layer, whose connections form a clique. Note that cores with addresses of the form (n, c) where $c = n \mod C$ do not participate in any communication within a layer. These cores only communicate with their corresponding cores in nodes whose layer offsets match their own. Fig. 5.4 depicts an example inter-layer communication topology for two layers. Intra-layer edges are suppressed for clarity. The edges in Fig. 5.4 mirror those of Fig. 5.3 aside from the addition of the self-offset edges missing from Fig. 5.3. Note that by adding the edges from Fig. 5.3 to both layers, we have a clique.

In the first local exchange on node n, a message from (n, c) with destination (n', c') is forwarded to $(n, n' \mod C)$. In the remote exchange, this message is sent to $(n', n \mod C)$. In the final local exchange, the message arrives at (n', c'). If any of these intermediate cores are the destination, it is received there and not

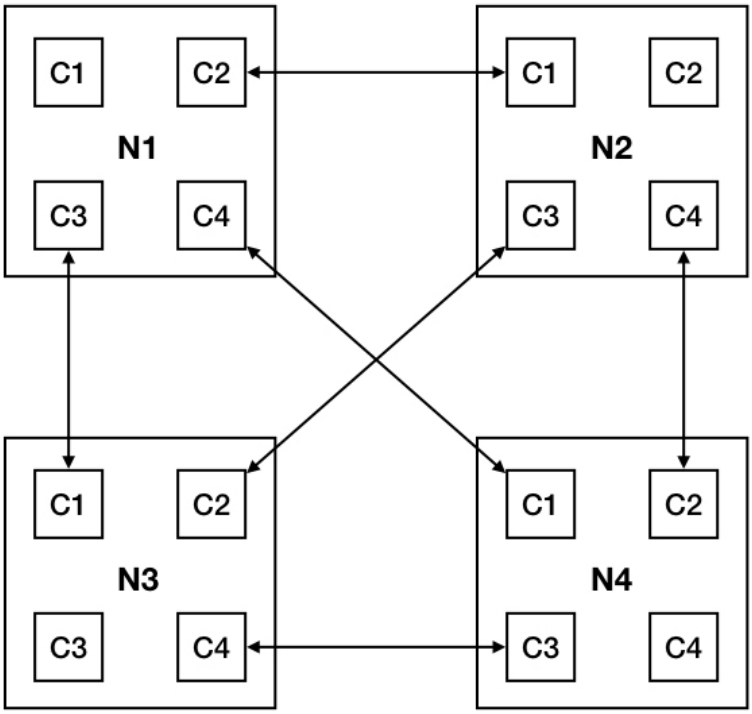


Figure 5.3: Example of the remote communication topology within a single layer of 4-core machines in the nlnr protocol.

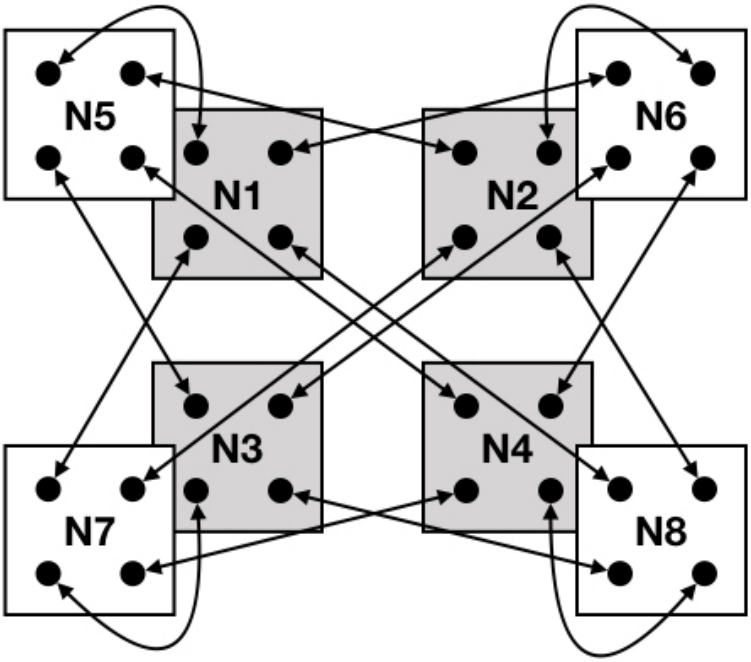


Figure 5.4: Example of the remote communication topology between two layers of 4-core machines in the nlnr protocol.

forwarded.

If a process needs to broadcast to all other processes, then it sends this message to each of its local neighbors, who forward it along to each of their local partners, who in turn distribute it on each remote node. Like the node remote protocol, a broadcast over NLNR results in $N-1$ remote messages.

While point to point messages are transmitted at most twice in node local and node remote, they might be transmitted up to 3 times in NLNR. While the local shared memory transmissions are less costly than remote transmissions over a wire, they are still not trivial and so this can result in overhead not seen in the other two protocols. However, in exchange we significantly reduce the number of channels over which remote messages traverse. Recall that node local and node remote each require C communication channels, each of which includes the N cores with matching core offset c for each $c \in [C]$. NLNR, however, requires $\binom{C}{2} + C$ channels, each including $2\frac{N}{C}$ cores (aside from the self-offset channels, which include $\frac{N}{C}$ cores each). Such a channel consists of all the address pairs (n, c), (n', c') where $c = \ell' = n' \mod C$ and $c' = \ell = n \mod C$ for some $(\ell, \ell') \in [C]^2$.

It is worth noting that there are many other specific assignments of cores to channels possible. We did not find that other assignments exhibited sufficiently different behavior, and so we will forgo a discussion thereof.

5.3 Experiments

Chapter 6

DegreeSketch with Applications to Neighborhood Approximation and Local Triangle Count Heavy Hitter Estimation

In this chapter we present DegreeSketch, a semi-streaming distributed sketch datastructure and demonstrate its utility for estimating local triangle count heavy hitters. DegreeSketch consists of vertex-centric cardinality sketches distributed across a set of processors. While other semi-streaming approaches to estimating local triangle counts depend on sampling, DegreeSketch is a persistent queryable data structure. We discuss the advantages and limitations of this approach, and present empirical results.

6.1 Introduction and Related Work

Counting the number of triangles in graphs is a canonical problem in both the RAM and streaming models. A "triangle" is a trio of co-adjacent vertices, and is the smallest nontrivial community structure. Consequently, triangles and triangle counting arise often in applications. Both the global count of triangles and the vertex-local counts, i.e. the number of triangles incident upon each vertex, are key to network analysis and graph theory topics such as cohesiveness [LK15], global and local clustering coefficients [Tso08], and trusses [Coh08]. These counts are also directly useful in many applications, such as spam detection [BBCG10], community discovery [WZTT10, BHLP11], and protein interaction analysis [MSOI+02].

Although not traditionally considered a centrality index, one can think of the number of triangles incident upon a vertex as a generalization of its degree centrality. To wit, we define the *triangle count centrality* of a vertex $x \in \mathcal{V}$ as

$$\mathcal{C}^{\text{TRI}}(x) = |\{yz \in \mathcal{E} \mid xy, zx \in \mathcal{E} \wedge |\{x, y, z\}| = 3\}|. \tag{6.1}$$

Similarly, we can consider the triangle count centrality of edges as well, defined similarly as

$$\mathcal{C}^{\text{TRI}}(xy) = |\{z \in \mathcal{V} \setminus \{x, y\} \mid yz, zx \in \mathcal{E}\}|. \tag{6.2}$$

Although many exact algorithms have been proposed for the triangle counting problem [Tso08, BBCG10, CC11, SV11, WDB+17], their time complexity is superlinear in the number of edges, $O(m^{\frac{3}{2}})$. Consequently, these analyses are expensive on very large graphs, particularly if they are dense. While there is a rich literature centered on the exact global and local triangle counting problem, we will not exhaustively recall it in this document.

In order to avoid the dreaded superlinear scaling of exact algorithms, many researchers have turned to approximation. Several streaming local triangle counting algorithms have been proposed in recent years. These serial streaming algorithms maintain a limited number of sampled edges from an edge stream. Streaming global triangle estimation algorithms have arisen that sample edges with equal probability [TKMF09],

sample edges with probability relative to counted adjacent sampled edges and incident triangles [ADWR17], and sample edges along with paths of length two [JSP13]. The first proposed semi-streaming local triangle estimation algorithm relies upon min-wise independent permutations accumulated over a logarithmic number of passes [BBCG08]. More recently, true single-pass algorithms have arisen such as MASCOT [LK15], which maintains local estimates that are updated whether an observed edge is sampled or not. Similarly, TRIÉST [SERU17] utilizes reservoir sampling, possibly disposing of sampled edges when a new edge is observed if system memory is saturated. This affords robustness to dynamic streams, as well as reducing variance. None of these algorithms perform edge-local triangle counting.

While many distributed global and vertex-local triangle counting algorithms have been proposed, the overwhelming majority store the graph in distributed memory and return exact solutions using MAPREDUCE [SV11] or distributed memory [AKM13, Pea17]. Recently, the study of distributed streaming vertex-local triangle counting was intiated in earnest with the presentation of TRY-FLY [SHL+18], which maintains parallel instantiations of $\text{TRIÉST}_{\text{IMPR}}$. A controller node feeds partitions of the graph stream to each of these instances in order to boost estimation accuracy and speed. DISLR [SLO+18] improves upon TRY-FLY by introducing limited redundancy into the graph stream partitions. Although TRY-FLY and DISLR are the most similar examples in the literature to our approach, we use distinct methods.

Our approach is fundamentally different, depending upon sketching rather than sampling as its core primitive. While the sampling approaches produce estimates, we produce a leave-behind queryable data structure similar in interface to COUNTMINSKETCH.

6.2 DEGREESKETCH via Distributed Cardinality Sketches

We take a different approach to estimating local triangle counts, relying upon sketches instead of sampling. We introduce DEGREESKETCH, a cardinality sketch-based distributed data structure trained on graph streams. We will first describe DEGREESKETCH at a high level, describe algorithms that utilize it in Sections 6.3, 6.4, and 6.5 and then discuss implementation details using the HYPERLOGLOG cardinality sketch in Section 6.6.

DEGREESKETCH maintains a data structure $\mathcal{D}$ that can be queried for an estimate of a vertex's degree, similar in interface to the celebrated COUNTSKETCH [CCFC02] and COUNTMINSKETCH [CM05]. This data structure is organized not unlike the sampling sketch approach to graph sparsification discussed in Section 4.1.2, with sketches summarizing the local information for each $x \in \mathcal{V}$. For each $x \in \mathcal{V}$ we maintain a cardinality sketch $\mathcal{D}[x]$, which affords the approximation of $\mathbf{d}_x$, the degree of x.

It is known that any data structure that provides relative error guarantees for the cardinality of a multiset with n unique elements requires $O(n)$ space [AMS99]. Consequently, investigators have developed many so-called *cardinality sketches* that provide such relative error guarantees while admitting a small probability of failure, such as PCSA [FM85], MinCount [BYJK+02], LogLog [DF03], Multiresolution Bitmap [EVF03], HyperLogLog [FFGM07], and the space-optimal solution of [KNW10]. While all these cardinality sketches have a natural union operation that allows one to combine the sketches of two multisets into a sketch of their union, most have no closed intersection operation.

For the purposes of discussion, we will abstract the particulars of cardinality sketches until Section 6.6. We assume that the sketches provide a ε-approximation of the number of unique items in a stream using $\Omega(\varepsilon^{-2})$ space. We assume that the sketches support an INSERT operation to add elements, a MERGE operation to combine sketches, an ESTIMATE operation to estimate cardinalities, and an ESTIMATEINTERSECTION operation to estimate intersection cardinalities. For reasons that will be described in Section 6.6.4, we do not assume that the ESTIMATEINTERSECTION procedure has the same error and variance properties as the guarantees for ESTIMATE.

We will describe the accumulation of a DEGREESKETCH instance on a universe of processors $\mathcal{P}$. We assume that the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is given by a stream $\boldsymbol{\sigma}$. $\boldsymbol{\sigma}$ is further partitioned by some unknown means into $|\mathcal{P}|$ substreams. We assume that each processor $P \in \mathcal{P}$ has send and receive buffers $\mathcal{S}[P]$ and $\mathcal{R}[P]$, respectively. We will make no assumptions in the following algorithms how processors handle switching context between processing and sending and receiving communication. In our implementations we use the software package YGM described in Chapter 5. We further assume that there is some partitioning of vertices to processors $f : \mathcal{V} \to \mathcal{P}$. We will occasionally abuse notation and use $f(P)$ to describe the set

of vertices that map to $P \in \mathcal{P}$. We make no assumptions about the particulars of f, noting that vertex partitionings are a subject of intense academic scrutiny as discussed in Section 5.1.1.

Algorithm 2 describes the distributed accumulation of a DegreeSketch instance. In a distributed pass over the partitioned stream, processors use the partition function f to send edges to the cognizant processors for each endpoint. These processors each maintain cardinality sketches for their assigned vertices. When $P \in \mathcal{P}$ receives an edge $xy \in \mathcal{E}$ where $f(x) = P$, it performs Insert$(\mathcal{D}[x], y)$. Once all processors are done reading and communicating, $\mathcal{D}$ is accumulated.

Algorithm 2 DegreeSketch Accumulation

Input: $\boldsymbol{\sigma}$ - edge stream divided into $|\mathcal{P}|$ substreams
$\mathcal{P}$ - 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: $\mathcal{D}$ - accumulated DegreeSketch

Send Context for $P \in \mathcal{P}$:
while $\mathcal{S}[P]$ is not empty **do**
  $(W, xy) \leftarrow \mathcal{S}[P]$.pop()
  $\mathcal{R}[W]$.push (xy)
Receive Context for $P \in \mathcal{P}$:
while $\mathcal{R}[P]$ is not empty **do**
  $xy \leftarrow \mathcal{R}[P]$.pop()
  if $!\exists\mathcal{D}[x]$ **then**
    $\mathcal{D}[x] \leftarrow$ empty sketch
  Insert$(\mathcal{D}[x], y)$
Accumulation Context for $P \in \mathcal{P}$:
$\mathcal{D} \leftarrow$ empty DegreeSketch dictionary
while σ_P has unread element uv **do**
  $U \leftarrow f(u)$
  $\mathcal{S}[P]$.push (U, uv)
  $V \leftarrow f(v)$
  $\mathcal{S}[P]$.push (V, vu)
return $\mathcal{D}$

DegreeSketch can be implemented with any cardinality sketch that admits some form of union and intersection estimation. In fact, the algorithms in Sections 6.4 and 6.5 do not even require a closed merge operation. In our experiments, we focus on the well-known HyperLogLog or HLL cardinality sketches. We use HLLs for implementation due to several attractive features:

1. Small register size
2. Simple merge operation
3. Sparse register representation
4. Adequate intersection estimator

We introduce HLL and discuss these features in greater detail in Section **??**. FIrst, however, we describe algorithms utilizing DegreeSketch for neighborhood size estimation in Section 6.3, recovering edge-local triangle count heavy hitters in Section 6.4, and finally recovering vertex-local triangle count heavy hitters in Section 6.5.

6.3 Neighborhood Size Estimation

Let the neighborhood function $\mathcal{N}_{\mathcal{G}}(t)$ be defined by

$$\mathcal{N}_{\mathcal{G}}(t) = |\{(x, y) \in \mathcal{V} \times \mathcal{V} | d_{\mathcal{G}}(x, y) < t\}|. \tag{6.3}$$

This function provides data about how fast the "average ball" around each $x \in \mathcal{V}$ expands, permitting the estimation of G's properties such as the effective diameter [PGF02]. For $t \in \mathbf{N}$, let $\mathcal{N}^t_{\mathcal{G}}(x)$ be the *local* neighborhood function defined as

$$\mathcal{N}^t_{\mathcal{G}}(x) = |\{y \in \mathcal{V} | d_{\mathcal{G}}(x, y) < t\}|. \tag{6.4}$$

We will drop the subscripts where they are clear. In particular, we have that

$$\mathcal{N}(t) = \sum_{x \in \mathcal{V}} \mathcal{N}^t(x). \tag{6.5}$$

The ANF [PGF02] and HyperANF [BRV11] algorithms estimate $\mathcal{N}^t(x)$ for each $x \in \mathcal{V}$ using Flajolet-Martin and HyperLogLog cardinality sketches, respectively, and produce estimates of $\mathcal{N}(t)$ of the form Eq. (6.5).

Let $\mathcal{D}$ be an instance of DegreeSketch as described, so that for $x \in \mathcal{V}$, $\mathcal{D}[x]$ is a cardinality sketch of the adjacency set of x. Assume that there is an approximate union operator $\widetilde{\cup}$. If we have $A_{:,x}$, then we can compute an estimate of $\mathcal{N}^2(x)$ by computing

$$\widetilde{\mathcal{N}}^2(x) = \widetilde{\bigcup}_{y : A_{y,x} \neq 0} \mathcal{D}[y]. \tag{6.6}$$

Higher-order merge operations described by Eq. (6.6) form the core of the ANF [PGF02] and HyperANF [BRV11] algorithms. We will restrict further analysis to HyperANF. HyperANF uses HyperLogLog sketches similar to DegreeSketch to estimate the t-hop neighborhood sizes of all vertices by way of a iterative sketch merging procedure, which is useful for applications such as edge prediction in social networks [GGL+13] and probabilistic distance calculations [BRV11, MSGL14]. However, HyperANF also requires storing the whole graph in memory in addition to the DegreeSketch data structure, and is optimized for shared but not distributed memory. It also does not take advantage of recent advances in HyperLogLog joint estimation that permit reasonable estimation of sketch intersections as well as unions.

Algorithm 3 uses DegreeSketch to recreate the behavior of HyperANF. After accumulating $\mathcal{D}^1$, an instance of DegreeSketch, the algorithm takes a number of additional passes over $\boldsymbol{\sigma}$. For t starting at 2, we accumulate

$$\mathcal{D}^t[x] = \widetilde{\bigcup}_{y : xy \in \mathcal{E}} \mathcal{D}^{t-1}[y] \tag{6.7}$$

by way of a message-passing scheme similar to Algorithm 2. When $P \in \mathcal{P}$ receives an edge $xy \in \mathcal{E}$ where $f(x) = P$, it forwards $\mathcal{D}^{t-1}[x]$ to $Y = f(y)$. When Y receives this sketch, it merges it into its next layer local sketch for y, $\mathcal{D}^t[y]$, computing Eq. (6.7) once all messages are processed. By construction, we have that

$$\mathcal{D}^t[x] = \widetilde{\bigcup}_{y : d(x,y) = s < t-1} \mathcal{D}^s[y]. \tag{6.8}$$

Ergo, the set of elements inserted into $\mathcal{D}^t[x]$ consists of all $y \in \mathcal{V}$ such that $d(x, y) < t$, which is to say that $\mathcal{D}^t[x]$ directly approximates $\mathcal{N}^t(x)$ (Eq. (6.4)). Ergo, the summations over all sketches in lines 18 and 28 of Algorithm 3 estimate $\mathcal{N}(t)$ in the same way as does the equivalent procedure in HyperANF. Note that these summations are performed as distributed Reduce operations. In addition to returning estimates of the neighborhood function, this procedure can be used to return estimates of the local t-degree neighborhoods of vertices, which have their own uses. Furthermore, as the actual estimates produced by Algorithm 3 as the same as those produced by HyperANF, all of the statistical results of [BRV11] apply. We reproduce the following theorem, although the other theorems and corollaries of [BRV11] also apply.

Theorem 6.3.1. *The output $\widetilde{\mathcal{N}}(t)$ of Algorithm 3 at the t-th iteration satisfies*

$$\frac{\mathbb{E}\left[\widetilde{\mathcal{N}}(t)\right]}{\mathcal{N}(t)} = 1 + \delta_1(n) + o(1) \text{ for } n \to \infty,$$

Algorithm 3 DegreeSketch Neighborhood Approximation

Input: $\boldsymbol{\sigma}$ - edge stream divided into $|\mathcal{P}|$ substreams
$\mathcal{P}$ - universe of processors
$\mathcal{D}^1$ - accumulated DegreeSketch
$\mathcal{S}$ - 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: $\mathcal{N}(t)$ for all $t > 1$

Send Context for $P \in \mathcal{P}$:
while $\mathcal{S}[P]$ is not empty **do**
  if next message is an Edge **then**
    $(W, xy, t) \leftarrow \mathcal{S}[P]$.pop()
    $\mathcal{R}[W]$.push (Edge, xy, t)
  else if next message is a Sketch **then**
    $(W, \mathcal{D}[x], y, t) \leftarrow \mathcal{S}[P].pop()$
    $\mathcal{R}[W]$.push (Sketch, y, t)
Receive Context for $P \in \mathcal{P}$:
while $\mathcal{R}[P]$ is not empty **do**
  if next message is an Edge **then**
    $(xy, t) \leftarrow \mathcal{R}[P]$.pop()
    $Y \leftarrow f(y)$
    $\mathcal{S}[P]$.push $\big(\text{Sketch}, (Y, \mathcal{D}^{t-1}[x], y, t)\big)$
  else if next message is a Sketch **then**
    $(\mathcal{D}^{t-1}[x], y, t) \leftarrow \mathcal{R}[P]$.pop()
    $\mathcal{D}^t[y] \leftarrow \text{Merge}(\mathcal{D}^t[y], \mathcal{D}^{t-1}[x])$
Execution Context for $P \in \mathcal{P}$:
$t \leftarrow 1$
$\mathcal{N}(0) \leftarrow |\mathcal{V}|$
$\mathcal{N}_P(1) \leftarrow \sum_{x \in f(P)} \text{Estimate}(\mathcal{D}^1[x])$
$\mathcal{N}(1) \leftarrow \text{Reduce}(\text{Sum}, \mathcal{N}_P(1))$
while $\mathcal{N}(t) \neq \mathcal{N}(t-1)$ **do**
  $t \leftarrow t + 1$
  $\mathcal{D}^t \leftarrow$ empty DegreeSketch dictionary
  while σ_P has unread element uv **do**
    $U \leftarrow f(u)$
    $\mathcal{S}[P]$.push $(\text{Edge}, (U, uv, t))$
    $V \leftarrow f(v)$
    $\mathcal{S}[P]$.push $(\text{Edge}, (V, vu, t))$
  $\mathcal{N}_P(t) \leftarrow \sum_{x \in f(P)} \text{Estimate}(\mathcal{D}^t[x])$
  $\mathcal{N}(t) \leftarrow \text{Reduce}(\text{Sum}, \mathcal{N}_P(t))$
  replace $\mathcal{D}^{t-1}$ with $\mathcal{D}^t$

where $\delta_1(n)$ is the same as in [FFGM07] Theorem 1, and $|\delta_1(x)| < 5 \cdot 10^{-5}$ when $r \geq 16$.

Furthermore, let $\widetilde{\mathcal{N}}^t(\cdot)$ be the local output of Algorithm 3 (i.e. $\widetilde{\mathcal{N}}^t(x) = \textsc{Estimate}(\mathcal{D}^t[x])$. Each of these estimates shares the standard deviation η_r given by [FFGM07], which is also shared by $\widetilde{N}(t)$. That is,

$$\frac{\sqrt{\text{Var}\left[\widetilde{\mathcal{N}}(t)\right]}}{\mathcal{N}(t)} \leq \eta_r.$$

Proof. The proof of the first two claims is the same as the proof of Theorem 1 in [BRV11], so we will not reproduce it here. □

Unlike HyperANF, Algorithm 3 is distributed and streaming, and does not require the storage of $\mathcal{G}$ in memory. However, it is also not optimized to take advantage of shared memory task decomposition or multicore optimizations using *broadword programming* like HyperANF. It is possible to design an algorithm that supports these features, but we do not produce it here as it would be out of scope. We will instead describe it at a high level. Rather than each core on each node operating independently, each node would act as an instance of HyperANF on a partition of the graph, communicating sketches to other nodes as needed like Algorithm 3. Merges and estimates would make use of all of the shared memory cores, utilizing the broadword programming approach given in Section 3 of [BRV11].

6.4 Edge-Local Triangle Count Heavy Hitters

In addition to estimating local neighborhood sizes, DegreeSketch affords an analysis of local triangle counts using intersection estimation. Furthermore, while sampling-based streaming algorithms are limited to vertex-local triangle counts, DegreeSketch affords the analysis of edge-local triangle counts. Edge-local triangle counts, i.e. the number of triangles in which each edge participate, can be thought of as a generalization of vertex-local triangle counts. Given the edge-local triangle counts for each edge incident upon a vertex, we can easily compute its vertex-local triangle count. Specifically,

$$\mathcal{C}^{\text{Tri}}(x) = \frac{1}{2} \sum_{xy \in \mathcal{E}} \mathcal{C}^{\text{Tri}}(xy) \tag{6.9}$$

The reverse is not true.

Edge-local triangle counts have understandably not received much attention in the streaming literature, considering that even enumerating them requires $\Omega(m)$ space. Given an accumulated DegreeSketch $\mathcal{D}$ and intersection operator $\widetilde{\cap}$, for $xy \in \mathcal{E}$ we can estimate $\mathcal{C}^{\text{Tri}}(xy)$ using

$$\widetilde{\mathcal{C}}^{\text{Tri}}(xy) = \mathcal{D}[x] \widetilde{\cap} \mathcal{D}[y]. \tag{6.10}$$

This procedure is similar to the well-known intersection method for local triangle counting. Indeed, we can estimate the total number of triangles $\mathcal{T}$ in the graph by computing

$$\widetilde{\mathcal{T}} = \frac{1}{3} \sum_{xy \in \mathcal{E}} \widetilde{\mathcal{C}}^{\text{Tri}}(xy) = \frac{1}{3} \sum_{xy \in \mathcal{E}} \mathcal{D}[x] \widetilde{\cap} \mathcal{D}[y]. \tag{6.11}$$

Unfortunately, while most cardinality sketches have a native and closed $\widetilde{\cup}$ operation, they all lack a satisfactory intersection operation. This is not surprising, as sketches are in effect lossy compressions. Indeed, it is known that the detection of a trivial intersection is impossible in sublinear memory. Hence, we must instead make use of unsatisfactory intersection operations in practice, which has been a focus of recent research [Tin16, CKY17, Ert17]. We will discuss these in more detail in Section 6.6.4, and their shortcomings in Section 6.7. For our purposes, we will suppose that $\widetilde{\cap}$ is reliable only where intersections are large. Consequently, we will attempt only to recovery the heavy hitters of $\mathcal{C}^{Tri}$.

Algorithm 4 provides a chassis for Algorithms 5 and 6, which differ only in their communication behavior. In Algorithm 4, all processors read over their edge streams and forward edges to one of their endpoints, similar to the behavior in the **Accumulation Context** of Algorithm 3. They also initialize a counter $\widetilde{\mathcal{T}}$ and a

Algorithm 4 Local Triangle Count Heavy Hitters Chassis

Input: $\boldsymbol{\sigma}$ - edge stream divided into $|\mathcal{P}|$ substreams
k - integral heavy hitter count
$\mathcal{P}$ - universe of processors
$\mathcal{D}$ - accumulated DegreeSketch
$\mathcal{S}$ - distributed dictionary mapping $\mathcal{P}$ to send queues
$\mathcal{R}$ - distributed dictionary mapping $\mathcal{P}$ to receive queues
f - function mapping $\mathcal{V} \to \mathcal{P}$

Accumulation Context for $P \in \mathcal{P}$:
1: $\widetilde{\mathcal{H}}_k \leftarrow$ empty k-heap
2: $\widetilde{\mathcal{T}} \leftarrow 0$
3: **while** σ_P has unread element uv **do**
4: $\quad U \leftarrow f(u)$
5: $\quad \mathcal{S}[P]$.push $(\textsc{Edge}, (U, uv))$
6: Reduce $\widetilde{\mathcal{T}}$
7: $\widetilde{\mathcal{T}} \leftarrow \frac{1}{3}\widetilde{\mathcal{T}}$

min heap with a maximum size of k, $\widetilde{\mathcal{H}}_k$. These values are modified in the send and receive contexts of Algorithms 5 and 6.

Algorithm 5 issues a chain of messages for each read edge, not unlike the procedure in Algorithm 3. P reads uv, and issues a message of type Edge containing uv to $U = f(u)$. Upon receipt, U issues a message of type Sketch containing $(\mathcal{D}[u], uv)$ to $V = f(v)$. When V receives this message, it computes $\widetilde{\mathcal{C}}^{\text{Tri}}(uv)$ via Eq. (6.10) and updates $\widetilde{\mathcal{T}}$ and $\widetilde{\mathcal{H}}_k$. Once computation is complete and all receive queues are flushed, the algorithm computes a global Reduce sum to find $\widetilde{\mathcal{T}}$ and similarly finds the global top k estimates via a reduce on $\widetilde{\mathcal{H}}_k$. The algorithm returns $\widetilde{\mathcal{T}}/3$ (each triangle is counted 3 times) and $\widetilde{\mathcal{H}}_k$.

Algorithm 5 addresses edge–local triangle count heavy hitter recovery using memory sublinear in the size of $\mathcal{G}$. It requires $\widetilde{O}(\varepsilon^{-2}m)$ time and communication, given our assumptions, and a total of $O(\varepsilon^{-2}|\mathcal{V}| \log\log|\mathcal{V}| + \log|\mathcal{V}|)$ space, where DegreeSketch is implemented using HyperLogLog sketches with accuracy parameter ε. Unfortunately, we are unable to provide an analytic bound on the error of this algorithm, due to the nature of sublinear intersection estimation. We will explore this problem in Section 6.7 and provide experimental analysis of Algorithm 6.8.

6.5 Vertex-Local Triangle Count Heavy Hitters

Given access to a trained DegreeSketch $\mathcal{D}$ and $A_{:,x}$, we can compute an estimate of $\mathcal{C}^{\text{Tri}}(x)$ using

$$\widetilde{\mathcal{C}}^{\text{Tri}}(x) = \frac{1}{2}\sum_{y:A_{y,x}\neq 0} \widetilde{\mathcal{C}}^{\text{Tri}}(xy) = \frac{1}{2}\sum_{y:A_{y,x}\neq 0} \mathcal{D}[x]\widetilde{\cap}\mathcal{D}[y]. \tag{6.12}$$

While we are not faced with the space constraint present in Section 6.4 when contending with simply writing down vertex-local triangle counts, we instead must contend with the dreaded small intersection problem discussed in Section 6.7. Consequently, we limit our scope to the recovery of vertex-local triangle count heavy hitters.

Algorithm 6 performs vertex-local triangle count estimation in a manner similar to Algorithm 5 with some additional steps. We maintain $\widetilde{\mathcal{C}}^{\text{Tri}}(x)$ for each $x \in \mathcal{V}$, which are of course distributed so that $X = f(x)$ computes $\widetilde{\mathcal{C}}^{\text{Tri}}(x)$. It performs similar work for $xy \in \mathcal{E}$ up to the point processor $Y = f(y)$ estimates $\widetilde{\mathcal{C}}^{\text{Tri}}(xy)$. Instead of inserting this estimate into a local max heap, we add it to $\widetilde{\mathcal{C}}^{\text{Tri}}(y)$, and forward $(\widetilde{\mathcal{C}}^{\text{Tri}}(xy), x)$ to $X = f(x)$ so that it can add it to $\widetilde{\mathcal{C}}^{\text{Tri}}(x)$. This message has the Est type, to distinguish it from Edge and Sketch messages.

Algorithm 5 DegreeSketch Edge-Local Triangle Count Heavy Hitters

Output: $\widetilde{\mathcal{T}}$, $\widetilde{\mathcal{C}}^{\text{Tri}}(xy)$ for top k edges xy

Send Context for $P \in \mathcal{P}$:
while $\mathcal{S}[P]$ is not empty **do**
  if next message is an Edge **then**
    $(W, xy) \leftarrow \mathcal{S}[P].\text{pop}()$
    $\mathcal{R}[W].\text{push}\,(\text{Edge}, xy)$
  else if next message is a Sketch **then**
    $(W, \mathcal{D}[x], y) \leftarrow \mathcal{S}[P].pop()$
    $\mathcal{R}[W].\text{push}\,(\text{Sketch}, xy)$
Receive Context for $P \in \mathcal{P}$:
while $\mathcal{R}[P]$ is not empty **do**
  if next message is an Edge **then**
    $xy \leftarrow \mathcal{R}[P].\text{pop}()$
    $Y \leftarrow f(y)$
    $\mathcal{S}[P].\text{push}\,(\text{Sketch}, (Y, \mathcal{D}[x], xy))$
  else if next message is a Sketch **then**
    $(\mathcal{D}[x], xy) \leftarrow \mathcal{R}[P].\text{pop}()$
    $\widetilde{\mathcal{C}}^{\text{Tri}}(xy) \leftarrow \text{EstimateIntersection}(\mathcal{D}[y], \mathcal{D}[x])$
    $\widetilde{\mathcal{T}} \leftarrow \widetilde{\mathcal{T}} + \widetilde{\mathcal{C}}^{\text{Tri}}(xy)$
    if $\widetilde{\mathcal{C}}^{\text{Tri}}(xy) > \min \mathcal{H}_k$ **then**
      insert $\left(xy, \widetilde{\mathcal{C}}^{\text{Tri}}(xy)\right)$ into $\mathcal{H}_k$
      if $|\mathcal{H}_k| > k$ **then**
        remove $\min \mathcal{H}_k$
Execution for $P \in \mathcal{P}$:
Run Algorithm 4 using these communication contexts
Reduce $\widetilde{\mathcal{H}}_k$
return $\widetilde{\mathcal{T}}, \widetilde{\mathcal{H}}_k$

Algorithm 6 DegreeSketch Vertex-Local Triangle Count Heavy Hitters

Output: $\widetilde{\mathcal{T}}$, $\widetilde{\mathcal{C}}^{\text{Tri}}(x)$ for top k vertices x

Send Context for $P \in \mathcal{P}$:
while $\mathcal{S}[P]$ is not empty **do**
  if next message is an Edge **then**
    $(W, xy) \leftarrow \mathcal{S}[P]$.pop()
    $\mathcal{R}[W]$.push (Edge, xy)
  else if next message is a Sketch **then**
    $(W, \mathcal{D}[x], xy) \leftarrow \mathcal{S}[P].pop()$
    $\mathcal{R}[W]$.push (Sketch, xy)
  else if next message is an Est **then**
    $(Y, \widetilde{\mathcal{C}}^{\text{Tri}}(xy), y) \leftarrow \mathcal{S}[P].pop()$
    $\mathcal{R}[T]$.push $\left(\text{Est}, (\widetilde{\mathcal{C}}^{\text{Tri}}(xy), y)\right)$

Receive Context for $P \in \mathcal{P}$:
while $\mathcal{R}[P]$ is not empty **do**
  if next message is an Edge **then**
    $xy \leftarrow \mathcal{R}[P]$.pop()
    $Y \leftarrow f(y)$
    $\mathcal{S}[P]$.push $(\text{Sketch}, (Y, \mathcal{D}[x], xy))$
  else if next message is a Sketch **then**
    $(\mathcal{D}[x], xy) \leftarrow \mathcal{R}[P]$.pop()
    $Y \leftarrow f(y)$
    $\widetilde{\mathcal{C}}^{\text{Tri}}(xy) \leftarrow \text{EstimateIntersection}(\mathcal{D}[y], \mathcal{D}[x])$
    $\widetilde{\mathcal{C}}^{\text{Tri}}(x) \leftarrow \widetilde{\mathcal{C}}^{\text{Tri}}(x) + \widetilde{\mathcal{C}}^{\text{Tri}}(xy)$
    $\widetilde{\mathcal{T}} \leftarrow \widetilde{\mathcal{T}} + \widetilde{\mathcal{C}}^{\text{Tri}}(xy)$
    $\mathcal{S}[P]$.push $\left(Y, (\widetilde{\mathcal{C}}^{\text{Tri}}(xy), y)\right)$
  else if next message is an Est **then**
    $(\widetilde{\mathcal{C}}^{\text{Tri}}(xy), y) \leftarrow \mathcal{R}[P]$
    $\widetilde{\mathcal{C}}^{\text{Tri}}(y) \leftarrow \widetilde{\mathcal{C}}^{\text{Tri}}(y) + \widetilde{\mathcal{C}}^{\text{Tri}}(xy)$

Execution for $P \in \mathcal{P}$:
$\widetilde{\mathcal{C}}^{\text{Tri}}(x) \leftarrow 0$ for each $x \in f(P)$
Run Algorithm 4 using these communication contexts
for $x \in f(P)$ **do**
  if $\widetilde{\mathcal{C}}^{\text{Tri}}(x) > \min \widetilde{\mathcal{H}}_k$ **then**
    insert $\left(x, \widetilde{\mathcal{C}}^{\text{Tri}}(x)\right)$ into $\widetilde{\mathcal{H}}_k$
    if $|\widetilde{\mathcal{H}}_k| > k$ **then**
      remove $\min \widetilde{\mathcal{H}}_k$
Reduce $\widetilde{\mathcal{H}}_k$
return $\widetilde{\mathcal{T}}, \widetilde{\mathcal{H}}_k$

Algorithm 6 addresses vertex–local triangle count heavy hitter recovery using the same asymptotic computation, memory and communication costs as Algorithm 5. Unfortunately, we are similarly unable to provide an a priori analytic bound on the error of this algorithm. We do, however, have the following theorem using the subadditivity of the standard deviation (i.e. If A and B have finite variance, $\sqrt{\text{Var}\left[A+B\right]} \leq \sqrt{\text{Var}\left[A\right]} + \sqrt{\text{Var}\left[B\right]}$).

Theorem 6.5.1. *Let $\widetilde{\mathcal{C}}^{\text{TRI}}(x)$ be the estimated output of Algorithm 6 for $x \in \mathcal{V}$, and that $\widetilde{\mathcal{C}}^{\text{TRI}}(xy)$ is the estimated edge triangle count for each $xy \in \mathcal{E}$. Assume further that for each xy, we know a standard deviation bound η_{xy} so that*

$$\frac{\sqrt{\text{Var}\left[\widetilde{\mathcal{C}}^{\text{TRI}}(xy)\right]}}{\mathcal{C}^{\text{TRI}}(xy)} \leq \eta_{xy}. \tag{6.13}$$

Furhermore, let $\eta_* = \max_{xy \in \mathcal{E}} \eta_{xy}$. Then, $\widetilde{\mathcal{C}}^{\text{TRI}}(x)$ has at most twice this maximum standard deviation. That is,

$$\frac{\sqrt{\text{Var}\left[\widetilde{\mathcal{C}}^{\text{TRI}}(x)\right]}}{\mathcal{C}^{\text{TRI}}(x)} \leq 2\eta_*.$$

Proof.

$$\begin{aligned}
\frac{\sqrt{\text{Var}\left[\widetilde{\mathcal{C}}^{\text{TRI}}(x)\right]}}{\mathcal{C}^{\text{TRI}}(x)} &= \frac{\sqrt{\text{Var}\left[\sum_{xy \in \mathcal{E}} \widetilde{\mathcal{C}}^{\text{TRI}}(xy)\right]}}{\mathcal{C}^{\text{TRI}}(x)} \\
&\leq \frac{\sum_{xy \in \mathcal{E}} \sqrt{\text{Var}\left[\widetilde{\mathcal{C}}^{\text{TRI}}(xy)\right]}}{\mathcal{C}^{\text{TRI}}(x)} && \text{subadditivity} \\
&\leq \frac{\sum_{xy \in \mathcal{E}} \eta_{xy} \mathcal{C}^{\text{TRI}}(xy)}{\mathcal{C}^{\text{TRI}}(x)} && \text{Eq. (6.13)} \\
&\leq \frac{\eta_* \sum_{xy \in \mathcal{E}} \mathcal{C}^{\text{TRI}}(xy)}{\mathcal{C}^{\text{TRI}}(x)} \\
&= 2\eta_* && \text{Eq. (6.9)}
\end{aligned}$$

□

Theorem 6.5.1 shows that if we can bound the standard deviation of the edge-local triangle count estimates produced using DegreeSketch, we can also bound the standard deviation of the vertex-local triangle count estimates produced by Algorithm 6. Unfortunately, we are unable to provide these bounds a priori, as they depend upon the sizes of all of the the sets and their intersections, which are unknown. It does, however, show how the variance of the vertex-local estimates depends upon those of the edge-local estimates.

6.6 HyperLogLog Cardinality Sketches

While many different cardinality sketches have been proposed, the HyperLogLog sketch is undoubtedly the most popular of these data structures in practice, and has attained widespread adoption [FFGM07]. The sketch relies on the key insight that the binary representation of a random machine word starts with $0^{j-1}1$ with probability 2^{-j}. Thus, if the maximum number of leading zeros in a set of random words is $j-1$, then 2^j is a good estimate of the cardinality of the set [FM85]. However, this estimator clearly has high variance. The variance is traditionally minimized using stochastic averaging to simulate parallel random trials [FM85].

Assume we have a stream σ of random machine words of a fixed size W. For a $W = (p+q)$-bit word w, let $\xi(w)$ be the first p bits of w, and let $\rho(w)$ be the number of leading zeros plus one of its remaining q bits. We pseudorandomly partition elements e of σ into $r = 2^p$ substreams of the form $\sigma_i = \{e \in \sigma | \xi(e) = i\}$. For each of these approximately equally-sized streams, we maintain an independent estimator of the above form. Each register $\mathbf{r}_i$, $i \in [m]$, accumulates the value

$$\mathbf{r}_i = \max_{x \in \sigma_i} \rho(x). \tag{6.14}$$

After accumulation, $\mathbf{r}_i$ stores the maximum number of leading zeroes in the substream σ_i, plus one. The authors of HyperLogLog show in [FFGM07] that the normalized bias corrected harmonic mean of these registers,

$$\widetilde{D} = \alpha_r r^2 \left(\sum_{i=0}^{r-1} 2^{-\mathbf{r}_i} \right)^{-1}, \tag{6.15}$$

where the bias correction term α_r is given by

$$\alpha_r := \left(r \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^r du \right)^{-1}, \tag{6.16}$$

is a good estimator of the number of unique elements in σ. If the true cardinality of the streamed multiset is D, the error of estimate $\widetilde{D}$, $|D - \widetilde{D}|$, has standard error $\approx 1.04/\sqrt{r}$. In expectation, $\widetilde{D}$ satisfies

$$|D - \widetilde{D}| \leq (1.04/\sqrt{r})D. \tag{6.17}$$

with high probability. A comprehensive analysis of the properties of the HLL sketch, the quality of (6.15), and a proof of (6.17) can be found in [FFGM07]. In particular, if N is the number of possible machine words, a HyperLogLog sketch satisfied Eq. (6.17) using space $O(\varepsilon^{-2} \log \log N + \log N)$, where $r = \Theta(\varepsilon^2)$.

Of course, practical streams do not consist of random 64-bit numbers. In practice, we simulate this randomness by way of hash functions, of which there are many alternatives. The fast, non-cryptographic hash functions Murmurhash3 [App] and xxhash [Col] are often utilized in implementations. We will assume throughout that algorithms have access to such a hash function $h : 2^{64} \to 2^{64}$.

A particular HyperLogLog sketch, S, consists of such a hash function h, a prefix size p (typically between 4 and 16), a maximum register value q, and an array of $r = 2^p$ registers, $\mathbf{r}$, all of which are initialized to zero. We summarize references to such a sketch as $\text{HLL}(p, q, h)$. Algorithm 7 describes the accumulation and functions supported by the vanilla HyperLogLog sketch. We will add features in the next few sections.

Note that HLLs support a natural merge operation: taking the element-wise maximum of each index of a pair register vectors. This requires that the two sketches were generated using the same hash function. We will assume that all sketches share a hash function throughout the rest of the chapter.

6.6.1 HLL Improvements: Sparse Register Format

Heule et al. suggest a sparse representation for HyperLogLog sketches consisting of a list of the set index-value pairs of a HLL's register list [HNH13]. Mathematically, the sparsification procedure is tantamount to maintaining the set $R = \{(i, \mathbf{r}_i) | \mathbf{r}_i \neq 0\}$. R requires less memory than $\mathbf{r}$ when the cardinality of the underlying multiset is small. Moreover, it is straightforward to saturate a sparse sketch into a dense one once it is no longer cost effective to maintain it by instantiating $\mathbf{r}$ while assuming all registers not set in R are zero. We will assume that R is implemented as a map, where an element $R[j] = z$ if $(j, z) \in R$ and is zero otherwise. Algorithm 8 describes the changes and additions to Algorithm 7 needed to implement sparse registers.

We use sparse registers in our algorithms, although we will not go into the details of their maintenance in the interest of clarity. The sparse sketch representation R can be implemented efficiently by maintaining a list sorted by register index, containing at most one entry per register, and a set of unsorted pairs that is periodically folded into the sorted list. In a practical implementation, these pairs are encoded into a single value that must be decoded when read. We obfuscate the details of the efficient implementation in our algorithms for the sake of clarity, and invite the interested reader to investigate Figures 6 & 7 of [HNH13].

Algorithm 7 HLL(p, q, h) Operations

State Variables for HLL(p, q, h) S :

- p integral prefix size
- q integral maximum register value
- h hash function mapping universe $U \to [0, 2^{64} - 1]$
- $r \coloneqq 2^p$
- $\mathbf{r}$ array of m registers, initially all zeroes

Accumulation:

1: $S \leftarrow$ empty HLL(p, q, h)
2: **for** $e \in \sigma$ **do**
3: $\quad x \leftarrow h(e)$
4: $\quad$ INSERT$(S, \xi(x), \rho(x))$

Functions:

5: **function** INSERT(S, j, z)
6: $\quad \mathbf{r}_j \leftarrow \max(\mathbf{r}_j, z)$
7: **function** MERGE$(S^{(0)}, S^{(1)}, \ldots, S^{(\ell)})$
8: $\quad S^* \leftarrow$ empty HLL(p, q, h)
9: $\quad$ **for** $j \in [0, r)$ **do**
10: $\quad\quad \mathbf{r}^*_j \leftarrow \max_{i \in [0,\ell]} \mathbf{r}^{(i)}_j$
11: $\quad$ **return** S^*
12: **function** ESTIMATE(S)
13: $\quad$ **return** $\alpha_r r^2 \left(\sum_{j=0}^{r-1} 2^{-\mathbf{r}_j} \right)^{-1}$

Algorithm 8 HLL(p, q, h) Operations Update - sparsification

State Variables for HLL(p, q, h) S :
- ν mode $\in \{\text{SPARSE}, \text{DENSE}\}$, initially SPARSE
- R sparse register set, initially $\emptyset$

1: **function** INSERT(S, j, z)
2: **if** $\nu = \text{DENSE}$ **then**
3: $\mathbf{r}_j \leftarrow \max(\mathbf{r}_j, z)$
4: **else if** $\nu = \text{SPARSE}$ **then**
5: $R[j] \leftarrow \max\{z, R[j]\}$ (see Figures 6 & 7 of [HNH13])
6: **if** $|R| > 6 * r$ **then**
7: SATURATE(S)
8: **function** SATURATE(S)
9: $\nu \leftarrow \text{DENSE}$
10: **for** $(j, z) \in R$ **do**
11: INSERT(S, j, z)
12: $R \leftarrow \emptyset$
13: **function** MERGE$(S^{(0)}, S^{(1)}, \ldots, S^{(\ell-1)})$
14: $S^* \leftarrow$ empty HLL(p, q, h)
15: **for** $j \in [0, r)$ **do**
16: $z \leftarrow \max_{i \in [0,\ell)} \left(\max\left\{\mathbf{r}_j^{(i)}, R^{(i)}[j]\right\}\right)$
17: **if** $z \neq 0$ **then**
18: INSERT(S^*, j, z)
19: **return** S^*
20: **function** ESTIMATE(S)
21: **if** $\nu = \text{DENSE}$ **then**
22: **return** $\alpha_r r^2 \left(\sum_{j=0}^{r-1} 2^{-\mathbf{r}_j}\right)^{-1}$
23: **else**
24: **return** $\alpha_r r^2 \left(\sum_{(j,z) \in R} 2^{-z}\right)^{-1}$

The authors of [HNH13] also describe a procedure by which the sparse elements of R might be stored at higher precision than the elements of $\mathbf{r}$. Although this method may be practical for applications we will avoid it in our work to avoid unnecessary confusion.

6.6.2 HLL Improvements: Reduced Register Size

In practice, $N = 2^{64}$, and so the registers require 6 bits apiece. This small register size is one of the advantages of HLLs over other cardinality sketches. For example, MinCount requires the storage of the same number of 64-bit hashes for the same universe size. Cardinality sketches are known to require $\Omega(\varepsilon^{-2} + \log N)$ space, although the known optimal algorithm is not considered practical [KNW10]. This reliance upon $\Omega(\varepsilon^{-2})$ registers implies that HLLs are about as optimal as we can get in implementations without sacrificing performance.

The authors of HyperLogLog-TailCut reduced the footprint of the HyperLogLog algorithm by reducing the registers to 4 bits [XZC17]. In order to avoid overflow, HyperLogLog-TailCut adds a base register b, initialized to zero, and changes the update rule (6.14) and estimator (6.15) so that each register $\mathbf{r}_j$ stores notional value $\mathbf{r}_j + b$. When $\mathbf{r}_j$ would experience an overflow event, HyperLogLog-TailCut instead increases b to the minimum set register value and decreases all registers by the same quantity. This yields an insert rule given by Algorithm 9.

Algorithm 9 HyperLogLog-TailCut Insert

$e \leftarrow$ next element of σ
$x \leftarrow h(e)$
if $\rho(x) - b > 15$ **then**
    $\Delta b \leftarrow \min_{i \in [0,r)} \mathbf{r}_i$
    if $\Delta b > 0$ **then**
        $b \leftarrow b + \Delta b$
        for $j \in [0, r)$ **do**
            $\mathbf{r}_j \leftarrow \mathbf{r}_j - \Delta b$
$\mathbf{r}_{\xi(x)} \leftarrow \max(\mathbf{r}_{\xi(x)}, \min(\rho(x) - b, 15))$

Note that this modification to the register update procedure is lossy. If $\rho(x) - b > 15$ even after updating b, then the algorithm notionally stores $b + 15$ in $\mathbf{r}_{\xi(x)}$. We henceforward refer to this event as a "tail cut". The authors show that tail cuts occur infrequently enough that it does not impact the Monte Carlo $1.04/\sqrt{m}$ error bound using the following modification to (6.15):

$$\tilde{D} = \alpha_r r^2 \left(\sum_{i=0}^{r-1} 2^{-(b+\mathbf{r}_i)} \right)^{-1}. \tag{6.18}$$

However, this tail-cutting introduces bias when performing merge operations on sketches. Concatenations of two streams may not yield the same answer as merging their individual sketches, which violates Eq. (1.1). Furthermore, tail-cutting is order dependent, so sketches accumulated over the same reordered stream may not agree, which violates a core assumption one usually makes about sketches.

The validity of the estimator based upon the union of sketches depends upon the property that the element-wise maximum of a set of sketches over streams is identical to the sketch accumulated from their concatentation. The tail-cutting procedure introduced in Algorithm 9 violates this property, as some hashes can be cut in the operands that would not be cut in the sketch over the concatenated stream. For a small number of sketches, the resulting error is small enough that it might go without notice. However, when merging many sketches, the additional error introduced by the tail cuts results in an estimator that does not maintain the desired error bound property (6.17).

We solve this problem by maintaining a set E of these cut elements. This set is of the same (index, value) form as the set of sparse registers R discussed above, and in practice is small as tail cutting events are uncommon. Where b is set, the probability of generating a hash x that causes an overflow is given by

$$\Pr[\rho(x) - b > 15] = 2^{-15-b} \tag{6.19}$$

using an idealized hash function. The probability that this hash gets cut is more difficult to characterize, and depends on the elements read thus far. We found approximately 4 tail cut events per 10^9 distinct element insertions in our experiments.

We add a subroutine that attempts to reinsert the cut elements into the registers. This subroutine gets called whenever the base register increases, the estimate procedure is called, the sketch is merged with another, or the set reaches a given size bound. These changes result in the updated INSERT, MERGE, and ESTIMATE procedures given in Algorithm 10. Note that these procedures are able to coexist with the sparse register format, allowing us to combine the two approaches.

Algorithm 10 guarantees order-invariance in the produced sketches, as well as maintaining a true sketch merge. These results come at the cost of some additional space, as the cut set E must be stored. In a pathological stream E could hold as many as $m-1$ elements, i.e. there is one holdout register index that prevents b from growing in spite of cut insertions. Fortunately, the likelihood of such an event is vanishingly unlikely. Furthermore, even this worst case at most doubles the size of the sketch, as so maintains the same asymptotic bounds as the vanilla HLL.

6.6.3 HLL Improvements: Maximum Likelihood Estimation

The estimator (6.15) is known to have several practical problems, many of which are discussed at length in [FFGM07] and [HNH13]. Subsequent work has refined HyperLogLog by modifying the estimator (6.15) to reduce bias on high and low values [FFGM07, HNH13, QKT16], reducing the register size by a constant [XZC17], and replacing the estimator (6.15) entirely with a maximum likelihood estimator [XZC17, Lan17, Ert17]. We adopt the latter of these approaches, which yields the added benefit of a maximum likelihood estimator for the intersection of two sketches. We will sketch the ideas for the estimators here, although the full treatment is somewhat involved. We direct the interested reader to [Ert17] for details.

The maximum likelihood estimator in [Ert17] uses a Poisson model, assuming that the cardinality itself is drawn from a Poisson distribution with parameter λ, and that the observed register values after accumulation are independent. This yields the following loglikelihood function for λ given the observed register list $\mathbf{r}$:

$$\mathcal{L}(\lambda \mid \mathbf{r}) = -\frac{\lambda}{r}\sum_{k=0}^{q}\frac{\mathbf{c}_k}{2^k} + \sum_{k=1}^{q}\mathbf{c}_k \log\left(1 - e^{-\frac{\lambda}{r2^k}}\right) + \mathbf{c}_{q+1}\log\left(1 - e^{-\frac{\lambda}{r2^q}}\right). \tag{6.20}$$

Here

$$\mathbf{c}_k = |\{\mathbf{r}_i = k \mid i \in \{0, \ldots, p-1\}\}| \tag{6.21}$$

is the count of occurrences of the value k in the register list $\mathbf{r}$ for $k \in \{0, 1, \ldots, q+1\}$. The author shows in [Ert17] that given an unbiased estimator $\hat{\lambda}$ for λ, we can leverage depoissonization [JS98] to yield an estimator for the fixed-size set. That is, $\mathbb{E}[\hat{\lambda} \mid D] = D$, where D is the cardinality of the input set. The count statistic $\mathbf{c}$ suffices to iteratively find the optimum of (6.20), yielding a maximum likelihood estimator. See Algorithm 8 of [Ert17] for the full algorithm description.

6.6.4 HLL Improvements: Intersection Estimation

A naïve approach to estimating an intersection of two sets A and B using cardinality sketches might involve computing the intersection via the inclusion-exclusion principle:

$$|A \cap B| = |A \cup B| - |A| - |B|. \tag{6.22}$$

Given sketches $S^{(A)}$ and $S^{(B)}$ for A and B, we might be tempted to estimate the intersection via

$$\widetilde{|A \cap B|} = \text{ESTIMATE}\left(S^{(A)}\right) + \text{ESTIMATE}\left(S^{(B)}\right) - \text{ESTIMATE}\left(\text{MERGE}\left(S^{(A)}, S^{(B)}\right)\right). \tag{6.23}$$

However, the approach in Eq. (6.23) suffers from several failings. In particular, it might be negative! Furthermore, due to the error noise in each estimate, if the true intersection is small relative to the set sizes, or if one set is much larger than the other, the variance of Eq. (6.23) will be quite high.

We describe a better intersection estimator due to Ertl [Ert17]. This estimator is similar to the maximum likelihood estimator Eq. (6.20), instead focusing on the joint distribution of a pair of sketched sets A and

Algorithm 10 HLL(p, q, h) Operations - sparsification + tail cut

State Variables for HLL(p, q, h) S :
- b base register, initially 0
- E cut set, initially $\emptyset$

Functions:

function INSERT(S, j, z)
  if $\nu =$ DENSE **then**
    $\mathbf{r}_j \leftarrow \max(\mathbf{r}_j, z)$
    if $z - b > 15$ **then**
      $\Delta b \leftarrow \min_{i \in [0,r)} \mathbf{r}_i$
      if $\Delta b > 0$ **then**
        $b \leftarrow b + \Delta b$
        for $i \in [0, r)$ **do** $\mathbf{r}_i \leftarrow \mathbf{r}_i - \Delta b$
        FLUSHCUTS(S)
    $\mathbf{r}_j \leftarrow \max(\mathbf{r}_j, \min(z - b, 15))$
    if $z - b > 15$ **then**
      $E[j] \leftarrow \max\{z, E[j]\}$
  else if $\nu =$ SPARSE **then**
    $R[j] \leftarrow \max\{z, R[j]\}$ (see Figures 6 & 7 of [HNH13])
    if $|R| > 4 * r$ **then**
      SATURATE(S)
function FLUSHCUTS(S)
  for $(j, z) \in E$ **do**
    $E \leftarrow E \setminus \{(j, z)\}$
    INSERT(j, z)
function MERGE$(S^{(0)}, S^{(1)}, \ldots, S^{(\ell)})$
  $S^* \leftarrow$ empty HLL(p, q, h)
  for $j \in [0, r)$ **do**
    $b^* \leftarrow \max_{i \in [0,\ell]} b^{(i)}$
    $z \leftarrow \max_{i \in [0,\ell]} \left(\max \left\{ \mathbf{r}_j^{(i)} + b^{(i)} - b^*, E^{(i)}[j], R^{(i)}[j] \right\} \right)$
    if $z \neq 0$ **then**
      INSERT(S^*, j, z)
  return S^*
function ESTIMATE(S)
  if $\nu =$ DENSE **then**
    return $\alpha_r r^2 \left(\sum_{j=0}^{r-1} 2^{-\max\{\mathbf{r}_j + b, E[j]\}} \right)^{-1}$
  else
    return $\alpha_r r^2 \left(\sum_{(j,z) \in R} 2^{-z} \right)^{-1}$

B. Accordingly, the estimator yields estimates of $|A \setminus B|$, $|B \setminus A|$, and $|A \cap B|$. The algorithm depends on a similar optimization of a Poisson model application, where it is assumed that $|A \setminus B|$ is drawn from a Poisson distribution with parameter λ_a, and similarly $|B \setminus A|$ and $|A \cap B|$ use Poisson parameters λ_b and λ_x. These parameters can be related to the observed HyperLogLog register lists corresonding to A and B, $\mathbf{r}^{(A)}$ and $\mathbf{r}^{(B)}$, via a loglikelihood function $\mathcal{L}(\lambda_a, \lambda_b, \lambda_x \mid \mathbf{r}^{(A)}, \mathbf{r}^{(B)})$. This is Eq. (70) in [Ert17], which we reproduce in Eq. (6.24) in this work.

$$
\begin{aligned}
\mathcal{L}(\lambda_a, \lambda_b, \lambda_x \mid \mathbf{r}^{(A)}, \mathbf{r}^{(B)}) = \\
&\sum_{k=1}^{q} \log\left(1 - e^{-\frac{\lambda_a+\lambda_x}{r2^k}}\right) \mathbf{c}_k^{(A),<} + \log\left(1 - e^{-\frac{\lambda_b+\lambda_x}{r2^k}}\right) \mathbf{c}_k^{(B),<} \\
&+ \sum_{k=1}^{q+1} \log\left(1 - e^{-\frac{\lambda_a}{r2^{\min\{k,q\}}}}\right) \mathbf{c}_k^{(A),>} + \log\left(1 - e^{-\frac{\lambda_b}{r2^{\min\{k,q\}}}}\right) \mathbf{c}_k^{(B),>} \\
&+ \sum_{k=1}^{q+1} \log\left(1 - e^{-\frac{\lambda_a+\lambda_x}{r2^{\min\{k,q\}}}} - e^{-\frac{\lambda_b+\lambda_x}{r2^{\min\{k,q\}}}} + e^{-\frac{\lambda_a+\lambda_b+\lambda_x}{r2^{\min\{k,q\}}}}\right) \mathbf{c}_k^{=} \\
&- \frac{\lambda_a}{r} \sum_{k=0}^{q} \frac{\mathbf{c}_k^{(A),<} + \mathbf{c}_k^{=} + \mathbf{c}_k^{(A),>}}{2^k} - \frac{\lambda_b}{r} \sum_{k=0}^{q} \frac{\mathbf{c}_k^{(B),<} + \mathbf{c}_k^{=} + \mathbf{c}_k^{(B),>}}{2^k} - \frac{\lambda_x}{r} \sum_{k=0}^{q} \frac{\mathbf{c}_k^{(A),<} + \mathbf{c}_k^{=} + \mathbf{c}_k^{(B),<}}{2^k}
\end{aligned}
\tag{6.24}
$$

Like Eq. (6.20), this function depends on the statistics:

$$
\begin{aligned}
\mathbf{c}_k^{(A),<} &= |\{i \mid k = \mathbf{r}_i^{(A)} < \mathbf{r}_i^{(B)}\}|, \\
\mathbf{c}_k^{(A),>} &= |\{i \mid k = \mathbf{r}_i^{(A)} > \mathbf{r}_i^{(B)}\}|, \\
\mathbf{c}_k^{(B),<} &= |\{i \mid k = \mathbf{r}_i^{(B)} < \mathbf{r}_i^{(A)}\}|, \\
\mathbf{c}_k^{(B),>} &= |\{i \mid k = \mathbf{r}_i^{(B)} > \mathbf{r}_i^{(A)}\}|, \\
\mathbf{c}_k^{=} &= |\{i \mid k = \mathbf{r}_i^{(A)} = \mathbf{r}_i^{(B)}\}|,
\end{aligned}
\tag{6.25}
$$

which capture the differences in register list distribution. The Naïve inclusion-exclusion estimator implemented using the maximum likelihood optimization of Eq. (6.20) depends on the count statistics

$$
\begin{aligned}
\mathbf{c}^{(A)} &= \mathbf{c}^{(A),<} + \mathbf{c}^{=} + \mathbf{c}^{(A),>}\mathbf{c}, \\
\mathbf{c}^{(B)} &= \mathbf{c}^{(B),<} + \mathbf{c}^{=} + \mathbf{c}^{(B),>}\mathbf{c}, \\
\mathbf{c}^{(A \cup B)} &= \mathbf{c}^{(A),>} + \mathbf{c}^{=} + \mathbf{c}^{(B),>}\mathbf{c},
\end{aligned}
\tag{6.26}
$$

which loses information present in the more detailed count statistics statistics in Eq. (6.25). Algorithm 9 of [Ert17] describes the estimation of $|A \setminus B|$, $|B \setminus A|$, and $|A \cap B|$ by accumulating the sufficient statistic (6.25) and using it to find the maximum of Eq. (6.24) via maximum likelihood estimation. The author shows extensive simulation evidence indicating that this method significantly improves upon the estimation error of a naïve estimator. They also show evidence suggesting that this estimator can be used to obtain a better estimate of $|A \cup B|$ than the naïve method, which involves composing a new sketch by computing the elementwise maximum of each register and taking its estimate. In the parlance of Algorithm 7 this is tantamount to computing $\textsc{Estimate}\left(\textsc{Merge}\left(S^{(A)}, S^{(B)}\right)\right)$. We reaffirm these conclusion with our findings in Section 6.8, where we use this algorithm to estimate the intersection of sets underlying pairs of sketches.

Algorithm 11 summarizes the various estimation procedures that we have described. In practice we use the machinery of Algorithm 10 along with the estimators of Algorithm 11.

6.7 Intersection Estimation Limitations: Dominations and Small Intersections

We have noted that there are limitations to the sketch intersection estimation in Section 6.6.4. There appear to be two main sources of large estimation error in practice.

Algorithm 11 HyperLogLog Maximum Likelihood Estimators

function EstimateMLE(S)
    Compute a statistic $\mathbf{c}$ of form Eq. (6.21)
    return MLE of (6.25) (e.g. Algorithm 8 of [Ert17])
function NaïveIntersection($S^{(A)}, S^{(B)}$)
    return EstimateMLE $\left(S^{(A)}\right)$ + EstimateMLE $\left(S^{(B)}\right)$
        −EstimateMLE $\left(\text{Merge}\left(S^{(A)}, S^{(B)}\right)\right)$
function EstimateUnion($S^{(A)}, S^{(B)}$)
    Compute statistics $\mathbf{c}^{(A),<}, \mathbf{c}^{(A),>}, \mathbf{c}^{(B),<}, \mathbf{c}^{(>),>}$ and $\mathbf{c}^{=}$ of form (6.25)
    return sum of MLEs of (6.24) (e.g. Algorithm 9 of [Ert17])
function EstimateIntersection($S^{(A)}, S^{(B)}$)
    Compute statistics $\mathbf{c}^{(A),<}, \mathbf{c}^{(A),>}, \mathbf{c}^{(B),<}, \mathbf{c}^{(>),>}$ and $\mathbf{c}^{=}$ of form (6.25)
    return Intersection MLE of (6.24) (e.g. Algorithm 9 of [Ert17])

The first is the phenomenon where $\mathbf{r}_i^{(A)} > \mathbf{r}_i^{(B)}$ for all i where $\mathbf{r}_i^{(B)} > 0$, resulting in $\mathbf{c}_k^{(A),<} = \mathbf{c}_k^{(B),>} = 0$ for all k and $\mathbf{c}_k^{=} = 0$ for all $k > 0$. We say that such an A *strictly dominates* B. In this case, Eq. (6.24) can be rewritten as the sum of functions depending upon λ_a and $\lambda_b + \lambda_x$. This means that the optimization relative to λ_a does not depend upon λ_x or λ_b, and is given by $\widetilde{\lambda}_{(A)} = \text{EstimateMLE}(S^{(A)})$. The optimization relative to $\lambda_b + \lambda_x$ is similarly independent of λ_a, and thus is given by $\widetilde{\lambda}_{(B)} = \text{EstimateMLE}(S^{(B)})$. Consequently, Eq. (6.24) does not specify an estimator for λ_x in this case, as it could be anything between 0 and $\widetilde{\lambda}_{(B)}$ without affecting the optimum.

We also consider the phenomenon where $\mathbf{r}_i^{(A)} \geq \mathbf{r}_i^{(B)}$ for all i, resulting in $\mathbf{c}_k^{(A),<} = \mathbf{c}_k^{(B),>} = 0$ for all k. We say that such an A *dominates* B. We are unable to make the same analytic statements about Eq. (6.24), as the terms dependent upon $\mathbf{c}^{=}$ are not eliminated. Consequently, the optimum estimate for λ_a depends upon λ_b and λ_x. If A dominates B, the count statistics given by Eq. 6.25 are unable to distinguish whether B is subset of A. Given the construction of Eq. (6.24), many and large nonzero values for $\mathbf{c}_k^{=}$ for large k will bias the optimization towards larger intersections, whereas the converse is true if $\mathbf{c}_k^{=}$ is nonzero for only a few small values of k. If $|A| \gg |B|$, then the latter might occur whether $|A \cap B|$ is large or small. Furthermore, note that if B is a subset of A, then A will (possibly strictly) dominate B.

If A dominates B, then $S^{(A \cup B)} = \text{Merge}(S^{(A)}, S^{(B)}) = S^{(A)}$. Ergo, the inclusion-exclusion naïve intersection estimator produces the estimate NaïveIntersection = EstimateMLE$(S^{(B)}) = \lambda_{(B)}$. This estimate is dubious, given that we have no evidence that the sets A and B hold any elements in common. This is especially true if $|A| \gg |B|$. Hence, both the naïve and maximum likelihood estimators may suffer from bias when a domination event occurs.

Consequently, it might be safest to disregard dominations in practice, as doing so can greatly reduce mean relative error. However, this poses a problem for our applications, as we will frequently have to compare the sketches of high degree vertices with those of comparatively low degree.

We have also noted the problem of small intersections. As discussed above, the intersection estimate optimized by Eq. (6.24) is proportional to the number (and size) of the nonzero $\mathbf{c}_k^{=}$ for $k > 0$, where larger k biases the estimate toward larger intersections. If the ground truth intersection is small relative to $|A|$ and $|B|$, however, Eq. (6.24) might exhibit high variance.

6.8 Experiments

Chapter 7

Distributed Sublinear Random Walk Simulation with Applications to Betweenness Centrality Heavy Hitter Recovery

In this chapter we discuss schemes for the distributed parallel sampling of random walks using sublinear memory. We extend this analysis to include random walk variants, with a focus on random simple paths. We employ the same fast ℓ_p sampling sketches introduced in Section 4.1.1 to bound the amount of working memory required to sample from each vertex. By storing the adjacency stream for each hub in fast storage, we can accumulate ℓ_p sampling sketches on demand. We are then able to afford the distributed, parallel sampling of random walks using sublinear distributed memory at the cost of I/O access.

We also demonstrate how this distributed framework can be applied to estimating κ-path centrality. κ-path centrality is notable for its empirical agreement with betweenness centrality on heavy hitters, as well as its relative ease of computation compared to the notoriously resource-hungry betweenness centrality.

7.1 Introduction and Related Work

Random walks are a fundamental primitive in the study of graphs, and participate as a core primitive in many graph algorithms in computer science. Random walks, and variants thereof, feature heavily in both theory and practice. Applications of random walks on graphs include connectivity testing [Rei08], clustering [AP09], graph partitioning [COP03, ACL07, ST13], load balancing [KR04], search [ALPH01, LCC+02], generating random spanning trees [Bro89], among other many other topics.

A random walk in an unweighted graph is a series of random steps, beginning at a given start vertex, that forms a path. In each step, the walk hops from the current vertex x to one of its neighbors y, where y is sampled uniformly. That is,

$$\Pr[y \mid x] = \frac{|A_{x,y}|^0}{\|A_{x,:}\|_0}. \tag{7.1}$$

Here we once again employ the convention that $0^0 = 0$. In a weighted graph, the neighbors are sampled with probability relative to their weight, i.e.

$$\Pr[y \mid x] = \frac{|A_{x,y}|^1}{\|A_{x,:}\|_1}. \tag{7.2}$$

These definitions are equivalent whether $\mathcal{G}$ is directed or undirected. Typically in applications a length t and a start vertex x_0 are given, after which t samples of the form Eq. (7.1) or Eq. 7.2 occur.

The total probability of a particular random walk of length t starting from $s \in \mathcal{V}$ is given by

$$\Pr_{\mathrm{rw}(s,t)}[(x_0, x_1, \ldots, x_t)] = \mathbf{1}_{[x_1 = s]} \prod_{i=0}^{t-1} \frac{A_{x_{i+1}, x_i}}{\|A_{:,x_i}\|_1}. \tag{7.3}$$

For two distributions P and Q over the same set $\mathcal{U}$, their ℓ_1 distance is given by

$$\|P - Q\|_1 = \sum_{u \in \mathcal{U}} |P(u) - Q(u)|. \tag{7.4}$$

Let $\widetilde{\Pr}_{\mathrm{rw}(s,t)}$ be the output distribution over $\mathcal{V}^{t+1}$ of a randomized algorithm $\mathcal{A}$ simulating t-length random walks starting from s. We say that $\mathcal{A}$ simulates random walks within error ε if $\|\widetilde{\Pr}_{\mathrm{rw}(s,t)} - \Pr_{\mathrm{rw}(s,t)}\|_1 \leq \varepsilon$. If $\varepsilon = 0$, we way $\mathcal{A}$ is a *perfect* simulation.

In distributed memory naïve implementation of random walk sampling using some vertex partitioning generates $O(t)$ communications, where each communication consists of the path generated thus far and is sent to the owner of the most recently sampled vertex. If we are locked in a Pregel-style synchronous communication scheme, this corresponds to $O(t)$ rounds required to sample any number of walks of length t. Das Sarma et al. proposed distributed algorithms for performing random walks in fewer rounds [DSNPT13]. By starting a larger number of walkers and joining them together, the authors prove results that permits $\widetilde{O}(\sqrt{tD})$ rounds for a single random walk of length t. Here D is the diameter of $\mathcal{G}$, i.e. it is the longest shortest path $D = \max_{x,y \in \mathcal{V}} d_{\mathcal{G}}(x, y)$. This matches, up to polylogarithmic factors, the corresponding lower bound proved in [NDSP11]. The authors also demonstrate an algorithm requiring $\widetilde{O}(\sqrt{ktD} + k)$ rounds for k parallel random walks of length t. As these bounds depend on the diameter of $\mathcal{G}$, they are practical only when $t \gg D$. Furthermore, although these algorithms reduce the maximum number of *sequential* computations required to sample random walks in such a situation, they may result in much more actual bandwidth usage than the naïve approach.

We will instead focus on augmenting the naïve distributed random walk sampling algorithm, by reducing the amount of working memory demanded by hub vertices. We do so by leveraging reservoir sampling and the ℓ_p sampling sketches discussed in Section 4.1.1 in a distributed scheme similar to DegreeSketch, discussed in Chapter 6 utilizing an asynchronous communication protocol such as YGM discussed in Chapter 5.

7.1.1 Streaming Random Walks

Das Sarma et al. demonstrated the first random walk sampling algorithm in the literature, using $O(n)$ memory and $O(\sqrt{t})$ passes over the graph [SGP11]. The authors questioned whether random walks could be simulated using fewer passes in the semi-streaming model.

If $\mathcal{G}$ is given in a dynamic stream, the hop probabilities Eq. (7.1) and Eq. 7.2 naturally correspond to ℓ_p sampling on the columns of A, as discussed in Section 4.1.1. Recall that a single ℓ_p sampling sketch of a frequency vector $\mathbf{f}$ can be used to obtain (i, P), where i is sampled with probability $P = (1 \pm \varepsilon)\frac{|\mathbf{f}_i|^p}{\|\mathbf{f}\|_p}$ with failure probability δ, where ℓ_0 sampling sketches require $O\left(\log^2 n \log \frac{1}{\delta}\right)$ space and ℓ_1 sampling sketches require $O\left(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon} \log^2 n \log \frac{1}{\delta}\right)$ space. Hence, it is natural to apply ℓ_p sampling sketches to the problem of simulating random walks.

As discussed in Section 4.1.1, each sketch can be used to obtain one sample only. Subsequent samples require "fresh" sketches in order to maintain the independence of the samples. So, consider the naïve algorithm that accepts a turnstile graph stream σ defining $\mathcal{G}$. Consider that each update (x, y, c) to σ can be interpreted as an update to $A_{x,y}$. Sample $t = \widetilde{O}(1)$ ℓ_0 sketch transforms $S_1, \ldots, S_t$. In a pass over σ accumulate $S_1 A_{:,x}, \ldots, S_t A_{:,x}$ For each $x \in \mathcal{V}$. Given start vertex x_1, then, sample x_{i+1} from $S_i A_{:,x_i}$.

One might expect that this procedure obtains a path $(x_1, \ldots, x_t)$, however each sample suffers from a probability of failure. Accounting for these failures by applying a union bound solves this problem at the cost of inflating the memory overhead of each sketch. We formalize this result in Theorem 7.1.1.

Theorem 7.1.1. *Let unweighted $\mathcal{G}$ with adjacency matrix A be given in a strict turnstile stream. A randomized algorithm can perfectly simulate a random walk of length t starting at vertex v_0, which need not be known ahead of time, using $O(nt \log^2 n \log(t/\delta))$ space, where δ is the probability of failure.*

Proof. As described above, sample t ℓ_0 sampling transforms $S^{(0)}, S^{(1)}, \ldots, S^{(t-1)}$, each having failure probability δ^* to be specified later. In a pass over $\mathcal{G}$, accumulate $S^{(0)}A_{:,x}, S^{(1)}A_{:,x}, \ldots, S^{(t-1)}A_{:,x}$ sketches for each vertex $x \in \mathcal{V}$. Note that if a sparse representation of $A_{:,x}$ is smaller than the size of these sketches, then we can store that instead and not bother with the sketches. After accumulation, we sample from a specified starting vertex's first sketch, sample from the resulting vertex's second sketch, and so on.

Call an event where some sampled sketch of $x \in \mathcal{V}$ outputs FAIL a *failure*. This can only happen when d_x is sufficiently large, but as this depends on the degree distribution of the graph we will forgo this detail in the subsequent analysis. We will adopt the assumption in this proof that the random variables $(v_0, v_1, \ldots, v_j) \in \mathcal{V}^{k+1}$ are sampled in a simulation until either $j = t$ or v_j fails. We say that $x \in \mathcal{V}$ is *chosen* if $v_i = x$ for some $0 \leq i < t$ during a simulation. Notably, x can fail only if it is chosen, as otherwise its sketch will never be queried. If a failure occurs during the simulation of a random walk starting at v_0 we output the path so far accumulated and FAIL. We know that the probability that a particular vertex x fails on a random walk simulation starting at vertex s is given by

$$\begin{aligned}
\Pr[x \text{ fails} \mid v_0 = s] &= \Pr[x \text{ fails and } x \text{ is chosen} \mid v_0 = s] \\
&= \Pr[x \text{ fails} \mid v_0 = s \text{ and } x \text{ is chosen}] \cdot \Pr[x \text{ is chosen} \mid v_0 = s] \\
&\leq \Pr[x \text{ fails}] \cdot \Pr[x \text{ is chosen} \mid v_0 = s] \\
&\leq \delta^* \cdot \Pr[x \text{ is chosen} \mid v_0 = s]. \qquad (7.5)
\end{aligned}$$

We have shown that the probability that a particular vertex fails is at most δ times the probability that it occurs as a non-final vertex in a simulated walk. We extend this result to obtain the probability that *any* failure occurs on a random walk starting with s.

$$\begin{aligned}
\Pr[\text{a failure occurs} \mid v_0 = s] &\leq \Pr\left[\bigvee_{x \in \mathcal{V}} x \text{ fails} \mid v_0 = s\right] \\
&\leq \sum_{x \in \mathcal{V}} \Pr[x \text{ fails} \mid v_0 = s] && \text{Union bound} \\
&\leq \delta^* \sum_{x \in \mathcal{V}} \Pr[x \text{ is chosen} \mid v_0 = s] && \text{Eq. 7.5} \\
&\leq \delta^* t. && |\{v_0, v_1, \ldots, v_{t-1}\}| \leq t
\end{aligned}$$

We have shown that the probability that a failure occurs in a simulation is less than $t\delta^*$. By setting $\delta^* = \frac{\delta}{t}$ we guarantee that a miss occurs with probability at most δ, obtaining our desired result. This also fixes the space and update time complexities. □

A similar algorithm suffices for simulating weighted walks up to ε error due to error inherent in the sketch.

We have shown a rudimentary serial algorithm for sublinearly sampling random walks from turnstile streams. However, unlike the application in Chapter 4, here there is no need to combine sampling sketches. Indeed, consider that the naïve algorithm for simulating t-step random walks we discussed above accumulates t sketches for each vertex and iteratively samples vertices, using a total of $\widetilde{O}(nt)$ space. Thus, the *sketch*-ness of the data structures is unimportant, and so we can utilize more general streaming data structures that might not admit a merge operator. However, as demonstrated by Li et al., streaming algorithms on turnstile streams possess a linear sketch equivalent [LNW14].

If $\mathcal{G}$ is given in an insert-only stream (i.e. σ lists $\mathcal{E}$), then the above result does not apply. Indeed, we can apply the well-studied *reservoir sampling* in the place of ℓ_p sampling sketches [Vit85]. Reservoir sampling maintains a reservoir of t items as follows. Upon reading the sth item, if fewer than t items are held then it is added to the reservoir. If t items are currently held, then with probability $\frac{t}{s}$ add it to the reservoir and discard one of the held items uniformly at random. We borrow the following lemma:

Lemma 7.1.2. *Given an insert-only stream σ, reservoir sampling uniformly samples t items without replacement in a single pass using $O(t)$ space.*

Furthermore, Efraimidis and Spirakis extended reservoir sampling to admit sampling from weighted streams [ES06]. For each index i of $\mathbf{f}$ we randomly sample $u_i \in (0,1)$ and compute a key $k_i = u_i^{\frac{1}{\mathbf{f}_i}}$. In a pass over the stream, we maintain the top k elements, and return it as a sample. We summarize this result in the following lemma:

Lemma 7.1.3. *Given a weighted insert-only stream σ with index weights $\mathbf{f}_i$ and total weight $\|\mathbf{f}\|_1$, reservoir sampling samples t items without replacement, sampling index i with probability proportional to $\frac{\mathbf{f}_i}{\|\mathbf{f}\|_1}$ in a single pass using $O(t)$ space.*

However, these weighted reservoir samplers are not robust to changes in weight, even cash register-style monotoic changes, and a suitable only for weighted insert-only streams. A similar approach that is robust to changes was developed by Indyk and Woodruff to estimate F_k moments [IW05] and was subsequently formalized and streamlined as *precision sampling* [AKO11] and applied to many other problems in the streaming literature, such as estimating entropy [BG06], cascading norms [JW09], Earth-mover distance [ADBIW09], and even ℓ_p sampling in [MW10, JST11]. At a high level, so-called precision sampling involves executing precision sampling with an internal CountSketch data structure that estimates the frequency vector whose indices are modulated by the random values sampled in precision sampling.

It if further worth noting that reservoir sampling can also be use to sample t items *with replacement* by maintaining t parallel reservoir samplers each of size 1. However, this increases the amount of work required to process each element in the stream by a factor of t. We can instead simulate sampling with replacement, which we summarize in Lemmas 7.1.2 and 7.1.3. This procedure maintains the update time up to constant factors at the cost of a logarithmic increase in the space overhead. We refer to this procedure as replacement reservoir sampling.

Lemma 7.1.4. *Given an insert-only stream σ consisting of n insertions, there is a procedure allowing reservoir sampling to uniformly sample $t \leq \frac{n}{2}$ items with replacement in a single pass using $O(t \log(n/t))$ bits of space.*

Proof. Run a reservoir sampler with capacity t on the stream, obtaining t samples stored in the list S while also recording c, the total count of elements in the stream. Let U be an initially empty set of used samples. When sampling from the sampler for the first time, select a random element from S and remove it, adding it to U before returning it. For subsequent samples, with probability $\frac{|U|}{c}$ instead sample a used sample uniformly from U and return it. Otherwise, select an element from S and add it to U as before. In this way, each returned item is selected with probability $\frac{1}{c}$ with replacement, as desired, which follows from the correctness of Lemma 7.1.2.

The sampler might consist of any subset of elements drawn from $\mathcal{V} \setminus \{x\}$ of size between 0 and t. Since the universe consists of n items, there are $\sum_{i=0}^{t} \binom{n-1}{i}$ total states that the sampler might assume. If $t < n/2$, then we are able to say $\sum_{i=0}^{t} \binom{n-1}{i} \leq \sum_{i=0}^{t} \frac{(n-1)^i}{i!} = \sum_{i=0}^{t} \frac{k^i}{i!} \left(\frac{n-1}{k}\right)^i \leq \left(\frac{e(n-1)}{t}\right)^t$, where e is Euler's constant. Thus, the state can be encoded using $\left\lceil t \log\left(\frac{e(n-1)}{t}\right)\right\rceil = O(t \log(n/t))$ bits of space. □

Lemma 7.1.5. *Given a weighted insert-only stream σ with index weights $\mathbf{f}_i$ and total weight $\|\mathbf{f}\|_1$, reservoir sampling samples t items without replacement, sampling index i with probability proportional to $\frac{\mathbf{f}_i}{\|\mathbf{f}\|_1}$ in a single pass using $O(t \log(n/t))$ words of memory.*

Proof. The proof is similar to that of Lemma 7.1.4. Collect S as before using weighted reservoir sampling, where the elements of S are tuples $(i, \mathbf{f}_i)$ consisting of element descriptors i and their weights $\mathbf{f}_i$. Simultaneously record the sum of weights in the stream, $\|\mathbf{f}\|_1$. Let U be an initially empty list of used samples. When sampling from the sampler for the first time, sample an element from S with probability proportional to its weight and remove it, adding it to U before returning it. Say that $U = ((i_0, \mathbf{f}_{i_0}), (i_1, \mathbf{f}_{i_1}), \ldots (i_j, \mathbf{f}_{i_j}))$ after performing some number of samples. Upon querying for a new sample, sample a random number $y \sim (0, \|f\|_1)$. If $y \leq \mathbf{f}_{i_0}$, return i_0. If $y \in \left(\sum_{k=0}^{\ell-1} \mathbf{f}_{i_k}, \sum_{k=0}^{\ell} \mathbf{f}_{i_k}\right]$ for some $0 < \ell \leq j$, then return i_ℓ. Else return the next element from S, removing it and placing it in U as $(i_{j+1}, \mathbf{f}_{j+1})$. As before, each item is returned with probability $\frac{\mathbf{f}_i}{\|\mathbf{f}\|_1}$ with replacement, as desired, which follows from the correctness of Lemma 7.1.3. Furthermore, the sampler has the same number of states with respect to sampled items as the sampler in Lemma 7.1.4.

However, each of these states possesses a list of $O(t)$ weights as well. Assuming that these weights can be stored in a machine word, we have proven the $O(t \log(n/t))$ bound. □

In particular, it is simple to reproduce the behavior of Theorem 7.1.1 for insert-only streams, which we do with Theorem 7.1.6.

Theorem 7.1.6. *Let unweighted $\mathcal{G}$ with adjacency matrix A be given in an insert-only stream. A randomized algorithm can simulate a random walk of length $t < n/2$ starting at vertex v_0, which need not be known ahead of time, using $O(nt \log(n/t))$ bits of memory.*

Proof. For every $x \in \mathcal{V}$, use Lemma 7.1.4 to uniformly sample t of x's neighbors. If $\mathbf{d}_x < t$ edges are incident upon x, then we can instead store $A_{:,x}$ explicitly as before. In a single pass over $\mathcal{G}$, we sample up to t neighbors uniformly with replacement from each vertex. Let N_x be this set of sampled neighbors for vertex x. We also compute the degree $\mathbf{d}_x$ of each vertex x. Like in Theorem 7.1.1 we simulate a random walk from given starting vertex v_0 using the sampled vertices. Unlike that theorem, however, the only additional work required is to read from N_x, as there is no sketch data structure. We are guaranteed not to run out of samples for any vertex, and the correctness of the distribution over random walks follows from Lemma 7.1.4. As there are n such replacement reservoir samplers operating in parallel, we have proven the $O(nt \log(n/t)))$ bit memory bound.

□

Theorem 7.1.7. *Let unweighted $\mathcal{G}$ with adjacency matrix A be given in an insert-only stream. A randomized algorithm can simulate a random walk of length $t < n/2$ starting at vertex v_0, which need not be known ahead of time, using $O(nt \log(n/t))$ words of memory.*

Proof. The proof is similar to Theorem 7.1.6, but uses Lemma 7.1.5 instead of Lemma 7.1.4. □

Jin explores the simulation of random walks from both insert-only and dynamic graph streams for directed and undirected graphs [Jin18]. The directed graph algorithms are similar to those corresponding to Theorems 7.1.1 and 7.1.6. The undirected algorithms boast only a $O(\sqrt{t})$ dependence on t by sampling $O(\sqrt{t})$ neighbors from each vertex's adjacency list, as well as computing all of the vertex degrees. The insert-only algorithm stores $A_{:,v}$ for each $v \in \mathcal{V}$ sparsely in memory until the vector reaches a size threshold, upon which it begins maintaining reservoir samplers for v. The turnstile algorithm follow a similar procedure, instead using ℓ_1 samplers when vectors become too large. Vertices that can be stored wholly in memory are recorded in a central map. Vertices with degree greater than $\sqrt{t}$ might maintain only a subset of their edges. Once accumulated, a random walk is simulated on this sparsified subgraph. When sampling from a high degree vertex, the central map is used with probability relative to the proportion of neighbors therein over the degree of the source. This is similar to the uniform sampling approach described in Theorem 7.1.6. Samples are consumed only when the central map is unused, and failures occur only when a vertex runs out of untouched samples.

7.2 Sublinear Simulation of Many Random Walks

We described serial algorithms for the sampling of single random walks in Theorems 7.1.1 and 7.1.6. 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 7.1.6 and 7.1.1 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.

7.2.1 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 7.2.1. *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 7.2.2. *For* $t = O(n^2)$*, 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_js 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 jth 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 jth 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 pth random walk output by $\mathcal{A}$ corresponds to the random variables $\left(v_0^{(p)}, v_1^{(p)}, \ldots v_{t^*}^{(p)}\right)$. Consider,

$$\begin{aligned}
\Pr\left[\left(v_{\ell+2}^{(p)}, v_{\ell+3}^{(p)}\right) = (a,b) \mid v_\ell^{(p)} \in A_j\right] &\leq \Pr\left[v_{\ell+1}^{(p)} \in V_0 \wedge v_{\ell+2}^{(p)} = a \wedge v_{\ell+3}^{(p)} = b \mid v_\ell^{(p)} \in A_j\right] \\
&= \Pr[v_{\ell+1}^{(p)} \in V_0 \mid v_\ell^{(p)} \in A_j] \cdot \Pr[v_{\ell+2}^{(p)} = a \mid v_{\ell+1}^{(p)} \in V_0] \cdot \Pr[v_{\ell+3}^{(p)} = b \mid v_{\ell+2}^{(p)} = a] \\
&\leq \frac{|V_0|}{|V_0| + |B_j|} \cdot \frac{1}{|A_j|} \cdot \frac{1}{|V_0| + |B_j|} \\
&= \frac{2}{9kt}. \qquad (7.6)
\end{aligned}$$

Thus, in ever four hops on the pth 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:

$$\begin{aligned}
\Pr\left[\text{walker } p \text{ fails} \mid v_0^{(p)} \in V_0\right] &\leq \Pr\left[\text{walker } p \text{ passes at every opportunity} \mid v_0^{(p)} \in V_0\right] \\
&\leq \prod_{s=1}^{t^*} \mathbf{1}_{[s \mod 4 = 1]} \Pr\left[\text{walker } p \text{ passes} \mid v_s^{(p)} \in A_j\right] \\
&\leq \prod_{s=1}^{t^*} \mathbf{1}_{[s \mod 4 = 1]} \left(1 - \Pr\left[\left(v_{\ell+2}^{(p)}, v_{\ell+3}^{(p)}\right) = (a,b) \mid v_\ell^{(p)} \in A_j\right]\right) \\
&\leq \left(1 - \frac{2}{9kt}\right)^{\left\lfloor \frac{t^*}{4} \right\rfloor}. \qquad \text{Eq. (7.6)}
\end{aligned}$$

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

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

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 7.2.1, $\mathcal{A}$ requires $\Omega(n\sqrt{kt})$ memory, and we have the result. □

7.2.2 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 7.1.1, 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 7.1.4 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 12 describes this accumulation procedure in pseudocode. Algorithm 13 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.

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

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

Algorithm 12 Insert-Only Streaming k Random Walk Accumulation

Input: σ - 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:
function InitVertex(x)
  if $\exists!\mathbf{d}[x]$ **then**
    $\mathbf{d}[x] \leftarrow 0$
    $\mathcal{L}[x] \leftarrow \emptyset$
    $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \emptyset$
    $\mathcal{N}_{\mathcal{B}}[x] \leftarrow$ empty sampler
function FeedSampler(x, y)
  if $\mathbf{d}[x] > c$ **then**
    Feed (x, y) into $\mathcal{N}_{\mathcal{B}}$
  else
    $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \mathcal{N}_{\mathcal{S}}[x] \cup \{(x, y)\}$
function InsertArc(x, y)
  InitVertex(x), InitVertex(y)
  $\mathbf{d}[y] \leftarrow \mathbf{d}[y] + 1$
  if $\mathbf{d}[y] = c + 1$ **then**
    for $u \in \mathcal{L}[y]$ **do**
      $\mathcal{N}_{\mathcal{S}}[u] \leftarrow \mathcal{N}_{\mathcal{S}}[u] \setminus (u, y)$
      FeedSampler(u, y)
  if $\mathbf{d}[y] \leq c$ **then**
    $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \mathcal{N}_{\mathcal{S}}[x] \cup \{(x, y)\}$
    $\mathcal{L}[y] \leftarrow \mathcal{L}[y] \cup \{x\}$
  else
    FeedSampler(x, y)
Accumulation:
for $xy \in \sigma$ **do**
  InsertArc(x, y)
  InsertArc(y, x)
return $\mathcal{N}_{\mathcal{S}}$, $\mathcal{N}_{\mathcal{B}}$

Algorithm 13 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:
function SimulateRandomWalk(v_0)
  for $i = 0, 1, \ldots, t-1$ **do**
    $a \sim_U [\mathbf{d}[v_i]]$
    if $a \leq |\mathcal{N}_{\mathcal{S}}[v_i]|$ **then**
      $v_{i+1} \sim_U \mathcal{N}_{\mathcal{S}}[v_i]$
    else
      if $|\mathcal{N}_{\mathcal{B}}[v_i]| > 0$ **then**
        $v_{i+1} \leftarrow$ next item from $\mathcal{N}_{\mathcal{B}}[v_i]$
        $\mathcal{N}_{\mathcal{B}}[v_i] \leftarrow \mathcal{N}_{\mathcal{B}}[v_i] \setminus \{v_{i+1}\}$
      else
        return $(v_0, v_1, \ldots, v_i)$, FAIL
  return $(v_0, v_1, \ldots, v_t)$
Accumulation:
parallel for $j \in [k]$ **do**
  $\left(v_0^{(j)}, v_1^{(j)}, \ldots, v_t^{(j)}\right) \leftarrow$ SimulateRandomWalk $\left(v_0^{(j)}\right)$
return $\left(v_0^{(j)}, v_1^{(j)}, \ldots, v_t^{(j)}\right)$ for all $j \in [k]$

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.

Lemma 7.2.3. *Suppose for every $x \in \mathcal{V}$, $\Pr\left[x \text{ fails} \mid v_0^{(1)} = x \wedge \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \leq \delta$. Then for any starting vertex $x \in \mathcal{V}$, $\Pr\left[\text{any vertex fails} \mid v_0^{(1)} = s \wedge \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \leq tk\delta$.*

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 $x \in \mathcal{V}$,

$$\begin{aligned}
&\Pr\left[x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \\
&= \Pr\left[x \text{ fails and } x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \\
&= \Pr\left[x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right) \text{ and } x \text{ is chosen}\right] \cdot \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \\
&\leq \Pr\left[x \text{ fails} \mid v_0^{(1)} = x, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \cdot \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \\
&\leq \delta \cdot \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right]. \qquad (7.7)
\end{aligned}$$

We are now able to show that

$$\begin{aligned}
&\Pr\left[\text{a failure occurs} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \\
&\leq \Pr\left[\bigvee_{x \in \mathcal{V}} x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] \\
&\leq \sum_{x \in \mathcal{V}} \Pr\left[x \text{ fails} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] && \text{Union bound} \\
&\leq \delta \sum_{x \in \mathcal{V}} \Pr\left[x \text{ is chosen} \mid v_0^{(1)} = s, \left(v_0^{(2)}, \ldots, v_0^{(k)}\right)\right] && \text{Eq. 7.5} \\
&\leq \delta kt && \leq kt \text{ vertices chosen.}
\end{aligned}$$

□

We now must show that a bound of the type supposed in Lemma 7.2.3 exists. We do so by setting c appropriately. The analysis unfortunately depends upon μ, the steady-state distribution of $\mathcal{G}$. μ corresponds to the left dominant eigenvector of A.

Lemma 7.2.4. *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*

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

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 $x \in \mathcal{V}$,

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

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 $v_1^{(k)}, v_2^{(k-1)}, \ldots, v_1^{(1)}$, followed by $v_2^{(k)}, v_2^{(k-1)}, \ldots, v_2^{(1)}$, and so on. Assume that x fails on the ℓth step of the wth 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 $\left(v_0^{(p)}, v_1^{(p)}, \ldots, v_\ell^{(p)}\right)$ of the pth walk where $p \in [w, k]$, and $\left(v_0^{(p)}, v_1^{(p)}, \ldots, v_{\ell-1}^{(p)}\right)$ 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

$$\Gamma_{i,j}\begin{pmatrix}\left(v_0^{(k)}, v_2^{(k)} \ldots, v_i^{(k)}\right), \\ \vdots \\ \left(v_0^{(j)}, v_2^{(j)} \ldots, v_{i-1}^{(j)}\right), \\ \vdots \\ \left(v_0^{(1)}, v_2^{(1)} \ldots, v_{i-1}^{(1)}\right)\end{pmatrix} = \Pr_{\mu}\left[v_0^{(2)}, \ldots, v_0^{(k)}\right]\left(\prod_{i^*=0}^{i-1}\prod_{j^*=1}^{k}\frac{1}{\mathbf{d}\left[v_{i^*}^{(j^*)}\right]}\right)\prod_{j^*=j+1}^{k}\frac{1}{\mathbf{d}\left[v_i^{(j^*)}\right]} \tag{7.8}$$

be the probability that $v_0^{(2)}, \ldots, v_0^{(k)}$ are sampled from μ 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

$$\begin{aligned}
&\Pr\left[x \text{ fails} \mid v_0^{(1)} = x \wedge v_0^{(2)}, \ldots, v_0^{(k)} \sim \mu \wedge (v_0, v_1) \in \mathcal{E}_{\mathcal{B}}\right] \\
&= \sum_{i=1}^{t}\sum_{j=k}^{1} \sum_{\substack{\left(v_0^{(k)}, v_2^{(k)} \ldots, v_i^{(k)}\right) \\ \vdots \\ \left(v_0^{(j)}, v_2^{(j)} \ldots, v_i^{(j)}\right) \\ \vdots \\ \left(v_0^{(1)}, v_2^{(1)} \ldots, v_{i-1}^{(1)}\right)}} \mathbf{1}\left[\begin{array}{c} v_0^{(1)} = v_{i-1}^{(j)} = x \wedge \left(v_0^{(1)}, v_1^{(1)}\right), \left(v_{i-1}^{(j)}, v_i^{(j)}\right) \in \mathcal{E}_{\mathcal{B}} \wedge \\ \left(\left|\left\{(i^*, j^*) \in [0, i-2] \times [k] \mid v_{i^*}^{(j^*)} = x \wedge \left(v_{i^*}^{(j^*)}, v_{i^*+1}^{(j^*)}\right) \in \mathcal{E}_{\mathcal{B}}\right\}\right| \right. \\ \left. + \left|\left\{j^* \in [j, k] \mid v_i^{(j^*)} = x \wedge \left(v_{i-1}^{(j^*)}, v_i^{(j^*)}\right) \in \mathcal{E}_{\mathcal{B}}\right\}\right| = c+1 \right) \end{array}\right] \frac{1}{\mathbf{d}_{\mathcal{B}}[x]}\Gamma_{i,j} \\
&= \sum_{i=1}^{t}\sum_{j=k}^{1} \sum_{\substack{\left(v_0^{(k)}, v_2^{(k)} \ldots, v_i^{(k)}\right) \\ \vdots \\ \left(v_0^{(j)}, v_2^{(j)} \ldots, v_{i-1}^{(j)}\right) \\ \vdots \\ \left(v_0^{(1)}, v_2^{(1)} \ldots, v_{i-1}^{(1)}\right)}} \mathbf{1}\left[\begin{array}{c} v_0^{(1)} = v_{i-1}^{(j)} = x \wedge \left(v_0^{(1)}, v_1^{(1)}\right) \in \mathcal{E}_{\mathcal{B}} \wedge \\ \left(\left|\left\{(i^*, j^*) \in [0, i-2] \times [k] \mid v_{i^*}^{(j^*)} = x \wedge \left(v_{i^*}^{(j^*)}, v_{i^*+1}^{(j^*)}\right) \in \mathcal{E}_{\mathcal{B}}\right\}\right| \right. \\ \left. + \left|\left\{j^* \in [j+1, k] \mid v_i^{(j^*)} = x \wedge \left(v_{i-1}^{(j^*)}, v_i^{(j^*)}\right) \in \mathcal{E}_{\mathcal{B}}\right\}\right| = c \right) \end{array}\right] \Gamma_{i,j}
\end{aligned} \tag{7.9}$$

Recall that $v_{\ell-1}^{(w)} = x$ is the point at which we assume the simulation fails. In Eq. (7.9) we threw away the ℓth step of the wth walk. At this point we have simulated the kth through $(w+1)$th walks up to ℓ 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. 7.8, but parameterized by these reversed walks. This allows

us to set the summation Eq. (7.9) equal to

$$\sum_{i=1}^{t}\sum_{j=k}^{1}\sum_{\substack{\left(v_0^{(k)'},v_2^{(k)'}\ldots,v_i^{(k)'}\right)\\ \vdots \\ \left(v_0^{(j)'},v_2^{(j)'}\ldots,v_{i-1}^{(j)'}\right)\\ \vdots \\ \left(v_0^{(1)'},v_2^{(1)'}\ldots,v_{i-1}^{(1)'}\right)}} \mathbf{1}\left[\begin{array}{c} v_0^{(1)'}=v_{i-1}^{(j)'}=x\wedge\left(v_{i-1}^{(j)'},v_i^{(j)'}\right)\in\mathcal{E}_{\mathcal{B}}\wedge \\ \left(\left|\left\{(i^*,j^*)\in[1,i-1]\times[k]|v_{i^*}^{(j*)'}=x\wedge\left(v_{i^*}^{(j^*)'},v_{i^*-1}^{(j^*)'}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|\right) \\ +\left|\left\{j^*\in[j+1,k]|v_i^{(j*)}=x\wedge\left(v_i^{(j^*)'},v_{i-1}^{(j^*)'}\right)\in\mathcal{E}_{\mathcal{B}}\right\}\right|=c \end{array}\right]\Gamma'_{i,j} \tag{7.10}$$

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

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 $\left(v_0^{(j)'},v_2^{(j)'}\ldots,v_{i-1}^{(j)'}\right)$, we have that

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

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

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

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 $\Pr\left[x\text{ fails}\mid v_0^{(1)}=x\wedge v_0^{(2)},v_0^{(3)}\ldots,v_0^{(k)}\sim\mu\right]<\delta$ 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 7.2.5. *There is a randomized algorithm that can simulate k t-step random walks where each source is drawn with replacement from μ in a single pass over an insert-only stream defining an undirected graph within error ε 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 7.2.3 and Lemma 7.2.4, where $\delta = \frac{\varepsilon}{2kt}$. □

Unfortunately this dependence upon μ 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 μ! Das Sarma et al. demonstrate an algorithm that can sample from μ 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 7.2.5 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 7.2.4, 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 12 and 13 easily, but their proofs of correctness would still depend upon μ. Thus, an improvement upon the constraints of Lemma 7.2.4 also translates to analogous results for streaming multigraphs and turnstile graphs.

7.2.3 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 6, we will assume a mailbox abstraction on communication implemented using an asynchronous communication protocol like that described in Chapter 5. 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 7.2.3. As the same operations will be performed, Theorem 7.2.5 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 14 describes the procedure of accumulating these data structures in a pass over a partitioned stream σ. It is similar to Algorithm 12, 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 15 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 13 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 14 and 15 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 14 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:
function InitVertex(x)
  if $\exists!\mathbf{d}[x]$ **then**
    $\mathbf{d}[x] \leftarrow 0$
    $\mathcal{L}[x] \leftarrow \emptyset$
    $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \emptyset$
    $\mathcal{N}_{\mathcal{B}}[x] \leftarrow$ empty sampler
  function FeedSampler(x, y)
    if $\mathbf{d}[x] > c$ **then**
      Feed (x, y) into $\mathcal{N}_{\mathcal{B}}$
    else
      $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \mathcal{N}_{\mathcal{S}}[x] \cup \{(x, y)\}$
Send Context $P \in \mathcal{P}$:
while $\mathcal{S}[P]$ is not empty **do**
  $(\xi, (x, y)) \leftarrow \mathcal{S}[P]$.pop()
  if $\xi =$ Edge **then**
    $W \leftarrow f(y)$
  else
    $W \leftarrow f(x)$
  $\mathcal{R}[W]$.push $(\xi, (x, y))$
Receive Context $P \in \mathcal{P}$:
while $\mathcal{R}[P]$ is not empty **do**
  $(\xi, (x, y)) \leftarrow \mathcal{R}[P]$.pop()
  if $\xi =$ Edge **then**
    InitVertex(y)
    $\mathbf{d}[y] \leftarrow \mathbf{d}[y] + 1$
    if $\mathbf{d}[y] = c + 1$ **then**
      for $u \in \mathcal{L}[y]$ **do**
        $\mathcal{S}[P]$.push (Promote, (u, y))
    if $\mathbf{d}[y] \leq c$ **then**
      $\mathcal{S}[P]$.push (Small, (x, y))
      $\mathcal{L}[y] \leftarrow \mathcal{L}[y] \cup \{x\}$
    else
      $\mathcal{S}[P]$.push (Big, (x, y))
  else if $\xi =$ Promote **then**
    $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \mathcal{N}_{\mathcal{S}}[x] \setminus (x, y)$
    FeedSampler(x, y)
  else if $\xi =$ Small **then**
    $\mathcal{N}_{\mathcal{S}}[x] \leftarrow \mathcal{N}_{\mathcal{S}}[x] \cup \{(x, y)\}$
  else if $\xi =$ Big **then**
    FeedSampler(x, y)
Accumulation $P \in \mathcal{P}$:
for $xy \in \boldsymbol{\sigma}_P$ **do**
  $\mathcal{S}[P].push$(Edge, (x, y))
  $\mathcal{S}[P].push$(Edge, (y, x))
return $\mathcal{N}_{\mathcal{S}}$, $\mathcal{N}_{\mathcal{B}}$

Algorithm 15 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 $P \in \mathcal{P}$:
while $\mathcal{R}[P]$ is not empty **do**
  $(v_0, v_1, \ldots, v_j) \leftarrow \mathcal{R}[P].\text{pop}()$
  $Q \leftarrow f(v_j)$
  $R[Q].\text{push}\,(v_0, v_1, \ldots, v_j)$
Receive Context $P \in \mathcal{P}$:
while $\mathcal{R}[P]$ is not empty **do**
  $(v_0, v_1, \ldots, v_i) \leftarrow \mathcal{R}[P].\text{pop}()$
  $a \sim_U [\mathbf{d}[v_i]]$
  if $a \leq |\mathcal{N}_{\mathcal{S}}[v_i]|$ **then**
    $v_{i+1} \sim_U \mathcal{N}_{\mathcal{S}}[v_i]$
  else
    if $|\mathcal{N}_{\mathcal{B}}[v_i]| > 0$ **then**
      $v_{i+1} \leftarrow$ next item from $\mathcal{N}_{\mathcal{B}}[v_i]$
      $\mathcal{N}_{\mathcal{B}}[v_i] \leftarrow \mathcal{N}_{\mathcal{B}}[v_i] \setminus \{v_{i+1}\}$
    else
      return $(v_0, v_1, \ldots, v_i)$, FAIL
  if $i + 1 = t$ **then**
    return $(v_0, v_1, \ldots, v_{i+1})$
  else
    $S[P].\text{push}\,(v_0, v_1, \ldots, v_{i+1})$
Execution $P \in \mathcal{P}$:
parallel for $j \in [k]$ **do**
  if $f\left(v_0^{(j)}\right) = P$ **then**
    $\mathcal{R}[P].\text{push}\left(v_0^{(j)}\right)$

algorithm 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 7.2.4.

7.2.4 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 may have its own memory bank, in which 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, $\mathcal{M}[x]$, allocated to $x \in \mathcal{V}_P$. In Algorithm 14, while it is 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, $M[x] = \{(u, v) \in \mathcal{E}_{\mathcal{B}} \mid u = x\}$ is a stream recorded in fast storage, but not held in working memory. 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.

Say, for example, that we run this algorithm where we sample $O(\sqrt{kt})$ neighbors per vertex, same as before. However, where simulation would have resulted in FAIL, we now take another pass over the relevant substream and record a new set of $O(\sqrt{kt})$ 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 $\omega(\sqrt{k})$ playbacks on x. Then the algorithm considers $\omega(k\sqrt{t})$ 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 by the single source algorithm, it would have output fail. We have shown that $\omega(\sqrt{k})$ playbacks occurring on one vertex implies that some walk is simulated that would have failed using the single source algorithm. Thus, the probability that $\omega(\sqrt{k})$ 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 7.2.6. *If δ is the probability that a single source streaming random walk simulation of length t fails, then the distributed algorithm with playback using the same parameters will generate $O(\sqrt{k})$ playbacks on any particular vertex with probability at most $(1-\delta)^k$.*

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 7.2.4 in the distributed case. 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 $q' = 2 + \frac{\log 1/\varepsilon}{\sqrt{k}}$. 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. For practical graphs, it is likely that most vertices will trigger far fewer than $\sqrt{k}$ playbacks, which indicates much less time spent in I/O and communication than taking an equivalent number of passes over $\boldsymbol{\sigma}$.

7.2.5 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.

7.3 Application: Semi-Streaming Estimation of Betweenness Centrality Heavy Hitters via κ-Path Centrality

7.3.1 Betweenness Centrality

Betweenness Centrality is a popular centrality index that is related to closeness centrality [Fre77]. The betweenness of a vertex is defined in terms of the proportion of shortest paths that pass through it. Thus, a vertex with high betweenness is one that connects many other vertices to each other - such as a boundary vertex connecting tightly-clustered subgraphs. For $x, y, z \in \mathcal{V}$, suppose that $\lambda_{y,z}$ is the number of shortest paths from y to z and $\lambda_{y,z}(x)$ is the number of such paths that include x. Then the betweenness centrality of x is calculated as

$$\mathcal{C}^{\text{BTW}}(x) = \sum_{\substack{x \notin \{y,z\} \\ \lambda_{y,z} \neq 0}} \frac{\lambda_{y,z}(x)}{\lambda_{y,z}}. \tag{7.11}$$

An implementation of betweenness centrality must solve the AllPairsAllShortestPaths problem, which is strictly more difficult than the AllPairsShortestPaths solution required by closeness centrality. Moreover, there is no known algorithm for computing the betweenness centrality of a single vertex using less space or time than the best algorithm for computing the betweenness centrality for all of the vertices. The celebrated Brandes algorithm, the best known algorithm for solving betweeness centrality, requires $\Theta(nm)$ time ($\Theta(nm + n^2 \log n)$ for weighted graphs) and space no better than that required to store the graph [Bra01].

A significant number of algorithms that attempt to alleviate this time cost by approximating the betweenness centrality of some or all vertices have been proposed. Some of these approaches depend on adaptively sampling and computing all of the single-source shortest paths of a small number of vertices [BKMM07, BP07], while others sample shortest paths between random pairs of vertices [RK16]. A recent advancement incrementalizes the latter approach to handle evolving graphs [BMS14].

Fortunately, researchers have directed much effort in recent years toward maintaining the betweenness centrality of the vertices of evolving graphs [GMB12, WC14, KMB15]. The most recent of these approaches keep an eye toward parallelization across computing clusters to maintain scalability.

The most fruitful recent line of research into approximating betweenness centrality involves the Hypergraph Sketch data structure, which maintains sampled shortest path DAGs between pairs of vertices [Yos14]. Hayashi et al. expanded upon the Hypergraph Sketch, showing how to maintain it in distributed memory on dynamic graphs [HAY15]. Riondato and Upfal recent improved upon this work further, applying Rademacher Averages to reduce the number of samples needed [RU18].

If the evolving graph in question is sufficiently small that storing it in working memory is feasible, then existing solutions suffice to solve the problem in a reasonably efficient fashion. However, none of the existing solutions adapt well to the semi-streaming model, as they each require $\Omega(m)$ memory. Indeed, directly approximating betweenness centrality seems likely to be infeasible using sublinear memory.

It is unclear how to sublinearize approximating the betweenness centrality of a vertex. Indeed, each solution to the SSSP problem requires $\Omega(m)$ memory. As this is the basis of the on- and off-line betweenness centrality approximation algorithms discussed in Section 7.3.1, variations on their approach is unlikely to yield a semi-streaming solution. Moreover, the spanner-based approach deployed in Section 4.2 to approximate closeness centrality does not apply. This is because betweenness centrality is defined in terms of the number of all shortest paths between two vertices, information which is lost when building the sparse spanner.

7.3.2 κ-Path Centrality

Kourtellis et al. attempt to approximate the top-k betweenness central vertices by way of relaxing its computation with the related κ-path centrality [ATK+11, KAS+13]. The authors define the κ-path centrality of a vertex x as the sum, over all $y \in \mathcal{V} \setminus \{x\}$, of the probability that a random simple path beginning at y of length at most k passes through x. In notation, this is

$$\mathcal{C}_{\kappa}^{\text{PATH}}(x) = \sum_{y \in \mathcal{V} \setminus \{x\}} \Pr_{p: p \text{ a simple path, } |p| \leq \kappa} [x \in p] \,. \tag{7.12}$$

Kourtellis et al. provide a randomized algorithm for approximating the κ-path centrality of the vertices of a graph, and demonstrate that on real-world networks vertices with high approximate κ-path centrality empirically correlate well with high betweenness-centrality vertices. Moreover, this algorithm is much more efficient than other attempts to approximate betweenness centrality, as it sidesteps the need to approximate the AllPairsAllShortestPaths problem. Finally, κ-Path centrality is desirable in that it depends only on κ-local features of the graph when considering any particular vertex.

7.3.3 Approximation of Heavy Hitters via κ-Path Centrality

In order to find a way forward, it is helpful to step back and assess what betweenness centrality purports to measure. Vertices with relatively high betweenness centrality scores are those that connect communities - the same vertices should also have high κ-path centrality scores, as discussed in Section 7.3.2. Moreover, the existing κ-path centrality approximation algorithm operates by way of sampling simple paths from $\mathcal{G}$. This algorithm looks suspiciously similar to the vertex-edge adjacency matrix sampling algorithms developed by Ahn, Guha, and McGregor discussed in Section 4.1.2.

While these results are promising, they do not solve the problem of approximating betweenness centrality in the semi-streaming turnstile model. First, the κ-path centrality approximation algorithm requires a static graph. Second, there is at present no known algorithm that can approximate κ-path centrality using $o(m)$ space. Third, as discussed by Kourtellis et al. in [KAS+13], the sampling algorithm requires $2\kappa n^{1-2\alpha} \ln n$ independent samples, where α is an accuracy parameter. Finally, while empirical correlation is promising, there are no theoretical guarantees that κ-path centrality will accurately capture the top-k betweenness central vertices, which is a highly desirable result. If an algorithm were to arise that simultaneously solves the first three problems, it might be used to approximate the betweenness centrality of the vertices of an arbitrary evolving graph in the semi-streaming model.

It is the authors' belief that ℓ_p-sparsification sketches of the type discussed in Section **??** might afford such an algorithm. Such methods certainly allow for randomly sampling trees (or, with some minor modifications, simple paths) in G in the semi-streaming model. The κ-path centrality approximation algorithm given by Kourtellis et al. is a simple Monte Carlo simulation, which affords a loose additive error. It may well be that a modified algorithm, or indeed a slightly different measure of centrality, suffices to easily sublinearize the approximation problem.

Even in this best case, however, it is important to note that the best we can hope is a semi-streaming algorithm that solves UBApproxCentral$(\text{BC}, \mathcal{G})$ well in practice. A solution to UBApproxTopCentral$(C, \mathcal{G}, k)$ is also desirable, but would require a single-pass algorithm that simultaneously maintains a heap - similar to the CountSketch heap algorithm discussed in Section **??**. Hence, even once such an algorithm is derived it will require extensive empirical evaluation on both real and synthetic data. Similar to the evaluation proposed in Section **??**, we propose an extensive performance comparison between such an algorithm once derived and existing techniques. Even if the empirical accuracy is substandard compared to state-of-the-art techniques, significant time and space improvements represent a contribution toward improving the state of the art.

7.4 Sublinear Distributed κ-Path Centrality

Bibliography

[ACL07] Reid Andersen, Fan Chung, and Kevin Lang. Using pagerank to locally partition a graph. *Internet Mathematics*, 4(1):35–64, 2007.

[ADBIW09] Alexandr Andoni, Khanh Do Ba, Piotr Indyk, and David Woodruff. Efficient sketches for earth-mover distance, with applications. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 324–330. IEEE, 2009.

[ADWR17] Nesreen K Ahmed, Nick Duffield, Theodore L Willke, and Ryan A Rossi. On sampling from massive graph streams. *Proceedings of the VLDB Endowment*, 10(11):1430–1441, 2017.

[AGM12a] Kook Jin Ahn, Sudipto Guha, and Andrew 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.

[AGM12b] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.

[AGM13] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 1–10. Springer, 2013.

[AKM13] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 529–538. ACM, 2013.

[AKO11] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Streaming algorithms via precision sampling. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 363–372. IEEE, 2011.

[ALPH01] Lada A Adamic, Rajan M Lukose, Amit R Puniyani, and Bernardo A Huberman. Search in power-law networks. *Physical review E*, 64(4):046135, 2001.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58:137–147, 1999.

[AP09] Reid Andersen and Yuval Peres. Finding sparse cuts locally using evolving sets. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 235–244. ACM, 2009.

[App] Austin Appleby. MurmurHash3. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp. Accessed: 2018-12-20.

[ATK+11] Tharaka Alahakoon, Rahul Tripathi, Nicolas Kourtellis, Ramanuja Simha, and Adriana Iamnitchi. K-path centrality: A new centrality measure in social networks. In *Proceedings of the 4th workshop on social network systems*, page 1. ACM, 2011.

[Bav48] Alex Bavelas. A mathematical model for group structures. *Applied anthropology*, 7(3):16–30, 1948.

[BBCG08] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24. ACM, 2008.

[BBCG10] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient algorithms for large-scale local triangle counting. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):13, 2010.

[BDR13] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12. IEEE, 2013.

[BG06] Lakshminath Bhuvanagiri and Sumit Ganguly. Estimating entropy over data streams. In *European Symposium on Algorithms*, pages 148–159. Springer, 2006.

[BG08] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11. IEEE, 2008.

[BHLP11] Jonathan W Berry, Bruce Hendrickson, Randall A LaViolette, and Cynthia A Phillips. Tolerating the community detection resolution limit with edge weighting. *Physical Review E*, 83(5):056119, 2011.

[BKMM07] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[BM11] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.

[BMS14] Elisabetta Bergamini, Henning Meyerhenke, and Christian L Staudt. Approximating betweenness centrality in large evolving networks. In *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 133–146. SIAM, 2014.

[BMS+16] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[BP07] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[Bro89] Andrei Broder. Generating random spanning trees. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 442–447. IEEE, 1989.

[BRV11] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. Hyperanf: Approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th international conference on World wide web*, pages 625–634. ACM, 2011.

[BV14] Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[BYJK+02] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.

[BYKS02] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[CC11] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[CCFC02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[CDPW14] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F Werneck. Computing classic closeness centrality, at scale. In *Proceedings of the second ACM conference on Online social networks*, pages 37–50. ACM, 2014.

[CKY17] Reuven Cohen, Liran Katzir, and Aviv Yehezkel. A minimal variance estimator for the cardinality of big data set intersection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 95–103. ACM, 2017.

[CM05] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[Coh08] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16, 2008.

[Col] Yann Collet. xxHash. https://github.com/Cyan4973/xxHash. Accessed: 2018-12-20.

[COP03] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 30–39. ACM, 2003.

[DF03] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.

[DSNPT13] Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. Distributed random walks. *Journal of the ACM (JACM)*, 60(1):2, 2013.

[Ert17] Otmar Ertl. New cardinality estimation algorithms for hyperloglog sketches. *arXiv preprint arXiv:1702.01284*, 2017.

[ES06] Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[EVF03] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166. ACM, 2003.

[FFGM07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[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.

[FM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[Fre77] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[GGL+13] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. ACM, 2013.

[GHC+17] Yong Guo, Sungpack Hong, Hassan Chafi, Alexandru Iosup, and Dick Epema. Modeling, analysis, and experimental comparison of streaming graph-partitioning policies. *Journal of Parallel and Distributed Computing*, 108:106–121, 2017.

[Gir09] Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

[GMB12] Oded Green, Robert McColl, and David A Bader. A fast algorithm for streaming betweenness centrality. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 11–20. IEEE, 2012.

[HAY15] Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. Fully dynamic betweenness centrality maintenance on massive networks. *Proceedings of the VLDB Endowment*, 9(2):48–59, 2015.

[HNH13] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.

[IW05] Piotr Indyk and David Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 202–208. ACM, 2005.

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

[JPNR17] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. Management and analysis of big graph data: current systems and open challenges. In *Handbook of Big Data Technologies*, pages 457–505. Springer, 2017.

[JS98] Philippe Jacquet and Wojciech Szpankowski. Analytical depoissonization and its applications. *Theoretical Computer Science*, 201(1-2):1–62, 1998.

[JSP13] Madhav Jha, Comandur Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 589–597. ACM, 2013.

[JST11] Hossein Jowhari, Mert Sağlam, and Gábor 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.

[JW09] Thathachar S Jayram and David P Woodruff. The data stream space complexity of cascaded norms. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 765–774. IEEE, 2009.

[KAS+13] Nicolas Kourtellis, Tharaka Alahakoon, Ramanuja Simha, Adriana Iamnitchi, and Rahul Tripathi. Identifying high betweenness centrality nodes in large social networks. *Social Network Analysis and Mining*, 3(4):899–914, 2013.

[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.

[KMB15] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. Scalable online betweenness centrality in evolving graphs. *IEEE Transactions on Knowledge and Data Engineering*, 27(9):2494–2506, 2015.

[KNW10] Daniel M Kane, Jelani Nelson, and David P Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 41–52. ACM, 2010.

[KR04] David R Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43. ACM, 2004.

[Lan17] Kevin J Lang. Back to the future: an even more nearly optimal cardinality estimation algorithm. *arXiv preprint arXiv:1708.06839*, 2017.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM, 2002.

[LK15] Yongsub Lim and U Kang. Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 685–694. ACM, 2015.

[LNW14] Yi Li, Huy L Nguyen, and David P Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 174–183. ACM, 2014.

[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.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

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

[MSGL14] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network?: the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 493–498. ACM, 2014.

[MSOI+02] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[MW10] Morteza Monemizadeh and David P Woodruff. 1-pass relative-error lp-sampling with applications. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1143–1160. SIAM, 2010.

[MWM15] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[NDSP11] Danupon Nanongkai, Atish Das Sarma, and Gopal Pandurangan. A tight unconditional lower bound on distributed randomwalk computation. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 257–266. ACM, 2011.

[Pea17] Roger Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–4. IEEE, 2017.

[PGA14] Roger Pearce, Maya Gokhale, and Nancy M Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 549–559. IEEE, 2014.

[PGF02] Christopher R Palmer, Phillip B Gibbons, and Christos Faloutsos. Anf: A fast and scalable tool for data mining in massive graphs. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 81–90. ACM, 2002.

[QKT16] Jason Qin, Denys Kim, and Yumei Tung. Loglog-beta and more: A new algorithm for cardinality estimation based on loglog counting. *arXiv preprint arXiv:1612.02284*, 2016.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):17, 2008.

[RK16] Matteo Riondato and Evgenios M Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30(2):438–475, 2016.

[RU18] Matteo Riondato and Eli Upfal. Abra: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(5):61, 2018.

[SERU17] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. Trièst: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(4):43, 2017.

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

[SHL+18] Kijung Shin, Mohammad Hammoud, Euiwoong Lee, Jinoh Oh, and Christos Faloutsos. Tri-Fly: Distributed estimation of global and local triangle counts in graph streams. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 651–663. Springer, 2018.

[SK12] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[SKW+14] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[SLO+18] Kijung Shin, Euiwoong Lee, Jinoh Oh, Mohammad Hammoud, and Christos Faloutsos. Dislr: Distributed sampling with limited redundancy for triangle counting in graph streams. *arXiv preprint arXiv:1802.04249*, 2018.

[ST13] Daniel A Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM Journal on Computing*, 42(1):1–26, 2013.

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[TGRV14] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014.

[Tin16] Daniel Ting. Towards optimal cardinality estimation of unions and intersections with sketches. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1195–1204. ACM, 2016.

[TKMF09] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

[Tso08] Charalampos E Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 608–617. IEEE, 2008.

[UCH03] Trystan Upstill, Nick Craswell, and David Hawking. Predicting fame and fortune: PageRank or indegree. In *Proceedings of the Australasian Document Computing Symposium, ADCS*, pages 31–40, 2003.

[Vit85] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[Vu18] Hoa Vu. Data stream algorithms for large graphs and high dimensional data. 2018.

[WC14] Wei Wei and Kathleen Carley. Real time closeness and betweenness centrality calculations on streaming network data. In *Proceedings of the 2014 ASE Big-Data/SocialCom/Cybersecurity Conference, Stanford University*, 2014.

[WDB+17] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

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

[WZTT10] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony KH Tung. On triangulation-based dense neighborhood graph discovery. *Proceedings of the VLDB Endowment*, 4(2):58–68, 2010.

[XCC+15] Ning Xu, Bin Cui, Lei Chen, Zi Huang, and Yingxia Shao. Heterogeneous environment aware streaming graph partitioning. *IEEE Transactions on Knowledge & Data Engineering*, (1):1–1, 2015.

[XZC17] Qingjun Xiao, You Zhou, and Shigang Chen. Better with fewer bits: Improving the performance of cardinality estimation of large data streams. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2017.

[YBP+11] Andy Yoo, Allison H Baker, Roger Pearce, et al. A scalable eigensolver for large scale-free graphs using 2d graph partitioning. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 63. ACM, 2011.

[Yos14] Yuichi Yoshida. Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1416–1425. ACM, 2014.