**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, and Trevor ???. 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 $\ell_p$ 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 $\kappa$-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 $\sigma$ after reading $t$ updates as $\sigma(t)$. A streaming algorithm $\mathcal{A}$ accumulates a data structure $\mathcal{S}$ while reading over $\sigma$. We will sometimes use the notation $\mathcal{D}(\sigma)$ to indicate the data structure state after $\mathcal{A}$ has accumulated $\sigma$. Authors generally assume $|\mathcal{D}| = \widetilde{O}(1) = O(\log m + \log n)$, where here the tilde suppresses logarithmic factors. Except where noted otherwise, we will assume the base 2 logarithm in our presentation.
**The semi-streaming model**: Unfortunately, logarithmic memory constraints are not always possible. In particular, it is known that many fundamental properties of complex structured data such matrices and graphs require memory linear in some dimension of the data [M+11, McG09]. In such cases, the logarithmic requirements of streaming algorithms are sometimes relaxed to $O(n \operatorname{polylog} n)$ memory, where $\operatorname{polylog} n = \Theta(\log^c n)$ for some constant $c$. In the case of matrices, here $n$ refers to one of the matrix's dimensions, whereas for graphs $n$ refers to the number of vertices. This is usually known as the *semi-streaming model*, although some authors also use the term to refer to $O(n^{1+\gamma})$ for small $\gamma$ [FKM+05, M+05].
**The frequency vector**: A stream $\sigma$ is often thought of as updates to a hypothetical frequency vector $f(\sigma)$, which holds a counter for each element in $\mathcal{U}$. We will drop the parameterization of $f$ where it is clear. We will sometimes parameterize $f(t)$ to refer to $f$ after reading $t \in [m]$ updates from $\sigma$. $\mathcal{D}$ can be thought of as a lossy compression of $f$ that only preserves some statistic thereof.
**Dynamic streams**: If $f$ is subject to change, then we call $\sigma$ a *turnstile* or *dynamic* stream. Such a $\sigma$'s elements are of the form $(i, c)$, where $i$ is an index of $f$ (an element of $\mathcal{U}$), $c \in [-L, \ldots, L]$ for some integer $L$, and $(i, c)$ indicates that $f_i \leftarrow f_i + c$. In the *cash register* model only positive updates are permitted, whereas in the *strict* turnstile model all elements of $f$ 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* if there is an operator $\oplus$ such that, for any streams $\sigma_1$ and $\sigma_2$,

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

**Linear sketching**: A sketch $\mathcal{S}$ is a *linear sketch* if it is a linear function of $f(\sigma)$ of fixed dimension. For streams $\sigma_1$ and $\sigma_2$ with frequency vectors $f_1$ and

$f_2$, scalars $a$ and $b$, and linear sketch transform $\mathcal{S}$,

$$a\mathcal{S}(f_1) + b\mathcal{S}(f_2) = \mathcal{S}(af_1 + bf_2). \tag{1.2}$$

**The graph stream model**: In this model, the stream $\sigma$ consists of edge insertions on $n$ vertices. 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 $\phi$ and attempt to recover all vertices $x \in \mathcal{V}$ such that $\mathbf{d}_x \geq \phi\|d\|_1$. In a cash register stream, we demonstrate an algorithm that recovers every vertex with degree $\geq \phi\|d\|_1$, and avoids returning any vertices with degree $< (\phi-\varepsilon)\|d\|_1$ with probability $(1-\delta)$). This algorithm requires space $O(\varepsilon^{-1}\log\frac{n}{\delta})$ with update time $O(\log\frac{n}{\delta})$. We also give an algorithm that accepts strict turnstile streams. This algorithm returns every vertex with degree $\geq (\phi + varepsilon)\|d\|_1$, and avoids returning any vertices with degree $< \phi\|d\|_1$ 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 $\ell_p$-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 partition the vertices of input graphs to processors, which communicate as required. 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, fully asynchronous approaches address 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 propose 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 on the same compute node exchange messages destined for the target core offset, followed by cores at the same core offset but different nodes exchanging messages to their final destinations.

2. **Node Remote Routing**. Cores with the same core offset but different nodes exchange messages destined for the target node, followed by cores at the same core offset but different nodes exchanging messages.

3. **Node Local Node Remote Routing**. 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 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 recover 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 recover the verte-local triangle counts (or, alternately, the heavy hitters) of a graph using linear time and $O(n(\varepsilon^{-2}\log\log n + \log n))$ worst-case distributed memory

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

1. **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 $\ell_p$ 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

$$O\left(\sum_{x\in\mathcal{V}} \min\left\{d_x, \log^3 n \log(1/\delta)\right\}\right)$$

space for unweighted graphs, and

$$O\left(\sum_{x\in\mathcal{V}} \min\left\{d_x, \varepsilon^{-1}\log(1/\varepsilon)\log^3 n \log(1/\delta)\right\}\right)$$

space for weighted graphs, where $d_x$ is the degree of vertex $x$.

2. **Hybrid sublinear random simple path sampling**. In a manner similar to random walk sampling, we construct $\ell_p$ 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.

3. **Hybrid sublinear random subtree sampling**. In a manner similar to random simple path sampling, we construct $\ell_p$ sampling sketches of sparse column vectors of the vertex-edge incidence matrix. Exact or sketched columns are transmitted along with walk histories so that they can be summed together with the next sampled column, allowing us to sample from tree neighbors without forming loops.

**Sublinear distributed $\kappa$-path centrality**: The $\kappa$-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. $\kappa$-path centrality was introduced as a cheaper, more scalable alternative to betweenness centrality, and is shown to empirically agree on heavy hitters [KAS+13]. $\kappa$-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 $\alpha$ is an accuracy parameter. We sublinearize this simulation algorithm to estimate the $\kappa$-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. 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 $A_{x,y} = \mathbf{w}_{xy}$. Let $D \in \mathbb{R}^{n \times n}$ be a diagonal matrix, where $D_{x,x}$ is the *degree* or *valency* of vertex $x \in \mathcal{V}$, which can be computed as the row sum of the $x$th row of $A$. We define $L = D - A$ as the *Laplace Matrix* or *Laplacian* of $\mathcal{G}$.

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

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

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. 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 with an edge with head $x$ and ends with an edge with tail $y$. There may be more than one such path. If the graph is clear from context, we may omit the subscripts and write $d(x, y)$. If there is no path connecting $x$ to $y$ in $\mathcal{G}$, then $d_{\mathcal{G}}(x, y) = \infty$. 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 $\ell_p$ 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 $\ell_0$ 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 $\ell_0$ and $\ell_1$ norms of matrix rows and columns. The $p$-th frequency moment $F_p$ of a vector $v$ is related to its $\ell_p$ norm in the following way:

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

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

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

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 $\Sigma$. $A_k$ is known to solve the optimization problem

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

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, discussed in detail in Chapter **??**.

## 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 $(\varepsilon, \delta)$-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 $x$th 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 $x$th 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 a 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 $v \in \mathbb{R}^n$ let $\mathcal{S}(v) \in \mathbb{R}^{t \times r}$ be the matrix whose $i$th column is given by $C^{(i)}v$. $\mathcal{S}(v)$ is the CountMinSketch object, and can be computed in $\widetilde{O}(\text{nnz}(v))$ 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 $v_j$ with bounded error. Given a vector $v \in \mathbb{R}^n$, we use the notation $v_{-i}$ to denote the vector whose elements are all the same as $v$ aside from the $i$th element, which is zero.

**Theorem 3.1.1** (Theorem 1 of [CM05])**.** *Let $v \in \mathbb{R}^n$ be the frequency vector of a strict turnstile stream and let $\mathcal{S}$ be its accumulated* CountMinSketch *with parameters $(\varepsilon, \delta)$. For all $j \in [n]$ the following holds with probability at least $1 - \delta$:*

$$0 \leq \widetilde{v}_j - v_j \leq \varepsilon \|v_{-j}\|_1$$

CountMinSketch guarantees error in terms of the $\ell_1$ norm, which is not as tight as the $\ell_2$ 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 $d$ 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 $d_y$". Outdegree is similar, and if $\mathcal{G}$ is undirected the two measures are equivalent. 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 $d$ is such that $d_x$ is exactly the degree centrality of vertex $x$.

We can generalize this model to account for weighted $\mathcal{G}$. In this case edge weights could change after insertion, and the vector $d$ is such that $d_x = \|A_{x,:}\|_1$ or $d_x = \|A_{x,:}\|_1$. Rather than the number of edges incident upon $x$, $d_x$ 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 $d_x \geq \phi\|d\|_1$ for a nontrivial fraction $\phi$. It is desirable in particular to compute and return these heavy hitters. While this can be achieved trivially with a second pass, single pass algorithms suffice.

**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 $\|d(t)\|$ for update $t$. The latter is monotonically increasing because the stream is insert only. Upon receiving $(x_t, y_t, c_t)$, update $\mathcal{S}(t)$ as usual and then query it for $\widetilde{d(t)}_x$ and $\widetilde{d(t)}_y$. Instead perform these updates for only the head or tail in out or in degree centrality, respectively. If $\widetilde{d(t)}_x \geq \phi\|d(t)\|_1$, add $\left(x, \widetilde{d(t)}_x\right)$ to $H$, and similarly for $y$. Let $\left(z, \widetilde{d(t^*)}_z\right) = \min(H)$ and remove it from $H$ if $\widetilde{d(t^*)}_z < \phi\|d(t)\|_1$. Once done, we iterate through $z \in H$ and using $\mathcal{S}$ output $z$ such that $\widetilde{d}_z \geq \phi\|d\|_1$.

**Theorem 3.2.1.** *Let $\mathcal{G}$ with degree vector $d$ be given in a cash register stream. An algorithm can return every vertex with degree $\geq \phi\|d\|_1$, and avoid returning any vertices with degree $< (\phi - \varepsilon)\|d\|_1$ 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.* Recall that $\|d(t)\|_1$ increases monotonically with $t$. If $t < t^*$ and a vertex's estimate is smaller than $\phi\|d(t)\|_1$, it cannot be larger than $\phi\|d(t)^*\|_1$ 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 $\widetilde{d}_x > \phi\|d_{-x}\|_1 \wedge d_x < (\phi - \varepsilon)\|d\|_1$ for some $x \in \mathcal{V}$ a *miss*. By Theorem 3.1.1, an accumulated CountMinSketch data structure $\mathcal{S}$ with parameters $\varepsilon$ and $\delta^*$ guarantees that

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

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 $\delta$, 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 $\|d(t)\|_1$ 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 $j \in \{0, 1, \ldots, \log n\}$ a CountMinSketch sketch $\mathcal{S}^{(\ell)}$. 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, allowing us to query for heavy hitters.

**Algorithm 1** Strict Turnstile Degree Centrality Heavy Hitters

**Input:** $\sigma$ - stream of edge updates
$\phi$ - threshold
$\varepsilon, \delta$ - approximation parameters

**Accumulation**:

**for** $j \in \{0, 1, \ldots, \log n\}$ **do**
  $\mathcal{S}^{(j)} \leftarrow$ empty independent CountMinSketch$(\varepsilon, \delta^*)$
$D \leftarrow 0$
**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** $(r, \widetilde{d}_r)$
    **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 $(\phi+\varepsilon)\|d\|_1$, 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 $d$ be given in a strict turnstile stream. Algorithm 1 can return every vertex with degree $\geq(\phi+\varepsilon)\|d\|_1$, and avoid returning any vertices with degree $<\phi\|d\|_1$ 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 $d_x\geq(\phi+\varepsilon)\|d\|_1$, each dyadic range of which it is a part must have a sum no less than $(\phi+\varepsilon)\|d\|_1$. Consequently, Algorithm 1 must return this vertex as CountSketches cannot underestimate any of these sums.

Call the event where $\widetilde{d}_x>(\phi+\varepsilon)\|d_{-x}\|_1\wedge d_x<\phi\|d\|_1$ for some $x\in\mathcal{V}$ a *miss*. Clearly, for each $j\in\{0,1,\ldots,\log n\}$ there are at most $1/\phi$ range sums greater than $\phi\|d\|_1$. 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 $\delta^*$. Assume that $\widetilde{d}_x^{(j)}$ is the queried output of some range $x$ from $\mathcal{S}^{(j)}$ in a run of Algorithm 1, and let $Q=\{(x,j)\mid\mathcal{S}^{(j)}\text{ is queried for }x\}$. By Theorem 3.1.1, the CountMinSketch data structures guarantee that

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

We have shown that the probability that *any* miss occurs is less than $\frac{\phi\delta^*}{2\log n}$. By setting $\delta^*=\frac{2\log n}{\phi}$ we guarantee that a miss occurs with probability at most $\delta$, 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 $\ell_p$ 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 [**?**, 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

$$\mathrm{CC}(x) = \frac{1}{\sum_{y \in \mathcal{V} \setminus \{x\}} d(x, y)}. \tag{4.1}$$

Consequently, an implementation of exact closeness centrality requires one to solve the AllSourcesAllShortestPaths 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 8.1. 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 $\mathcal{C}^{\text{CLOES}}$ 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 $k = O(\log n)$ uniformly sampled source vertices, $C \subseteq \mathcal{V}$. For $x \in \mathcal{V}$, we can estimate closeness centrality using $\widetilde{\mathcal{C}}^{\text{CLOSE}}(x) = \sum_{y \in C \setminus} \frac{d(x,y)}{k}$. 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 $C$ , $c(v) = \text{argmin}_{u \in C} d(uv)$ 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 $\widetilde{\mathcal{C}}^{Close}(x) - \mathcal{C}^{Close}(x) \leq O(\varepsilon)\mathcal{C}^{Close}(x)$ 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 $\ell_p$ sampling sketches, while Section 4.1.2 discusses their application to constructing spanning subgraphs.

### 4.1.1 $\ell_p$ Sampling Sketches

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

**Definition 4.1.1.** *Let $\Pi$ be a distribution on real $r \times n$ matrices, where $r = \text{poly}(1/\varepsilon, \log(1/\delta))$. Suppose $v \in \mathbb{R}^n$ and sample $S \sim \Pi$. Suppose further that there is a procedure that, given $Sv$, can output $(i, P)$ where $i \in [n]$ is an index of $v$ sampled with probability $P = (1 \pm \varepsilon)\frac{|v_i|^p}{\|v\|_p^p}$, failing with probability at most $\delta$. Then $\Pi$ is an $\ell_p$ sampling sketch distribution, $S \sim \Pi$ is an $\ell_p$ sketch transform, and $Sv$ is an $\ell_p$ 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 $\ell_0$ and $\ell_1$ 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 $\ell_0$-sampling sketches can achieve arbitrary sampling precision at no additional cost. Vu improves these sampling sketches with algorithms that allow the simultaneous accumulation of $s$ sampling sketches for a single vector $\mathbf{f}$ the same asymptotic update times, i.e. without dependence upon $s$ so long as $s = o(\|\mathbf{f}\|_0)$.

The full details are verbose and out of the scope of this document, as we will be using $\ell_p$ 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 $\mathbf{v}, \mathbf{v}' \in \mathbb{R}^n$ and $S$ is a sketch mat rix drawn from such a sampling distribution, then $S\mathbf{v} + S\mathbf{v}' = S(\mathbf{v} + \mathbf{v}')$. This observation is key for many sublinear graph sparsification algorithms, such as the spanner construction discussed in Section 4.1.2.

### 4.1.2 $\ell_p$-Sampling Graph Sparsification

$\ell_p$ sampling sketches, particularly of the $\ell_0$ and $\ell_1$ 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 [**?**], and identifying densest subgraphs [**?**].

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 $\ell_0$ 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 $\ell_1$-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 $\alpha$-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 $\ell_0$ 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 $\ell_0$ 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 $\ell_p$-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 $\alpha$-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 $\alpha$-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 $\alpha$-spanner of $\mathcal{G}$, then for any $x \in \mathcal{V}$, the following holds:

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

This immediately implies Eq. (4.5). □

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 factor of $1 - \frac{1}{k^{\log 5}}$. 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 $x \in \mathcal{V}$*

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

*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 $\widetilde{O}(n^{1+1/k} \log \frac{n}{\varepsilon^3})$ 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 [**?**]. and [**?**] 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 [**?**, **?**, **?**]. However, such schemes are susceptible to hypersparsity in some partitions, wherein a processor may be assigned more vertices than edges [**?**]. This results in poor scaling in practice. Some approaches avoid this by joining 1D and 2D partitioning schemes [**?**].

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 [**?**, **?**, **?**]. [**?**] 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, **?**]. In some degenerate cases synchronous implementations of algorithms are entirely unable to complete in

reasonable amounts of time compared to asynchronous implementations [**?**]. 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.

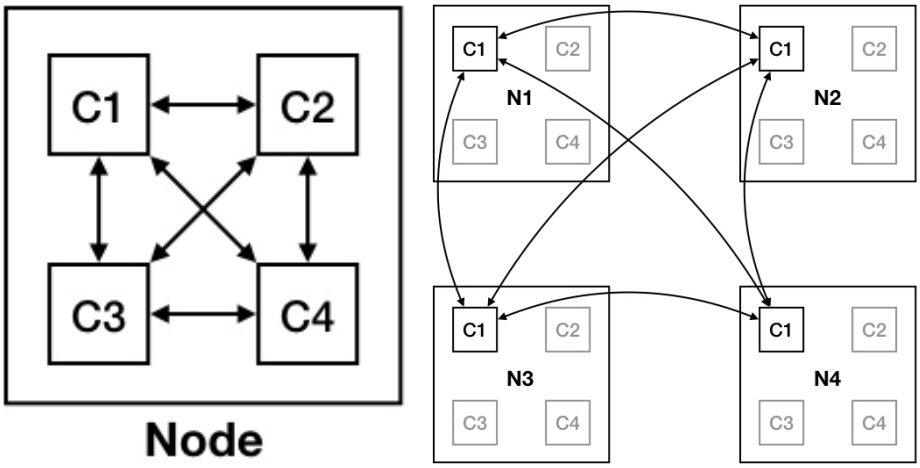
We discuss three related routing protocols - *Node Local*, *Node Remote*, and *Node Local Node Remote* or *NLNR*. Each of these variants take advantage of *local* and *remote* routing in hybrid distributed memory systems. We use the term *local* to refer to communications wherein the source and destination 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 latter 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 $c$th core on the $n$th 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 $n$th 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




(a) Local exchange (b) Remote exchange

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

are triggered upon entering a communication context in a distributed program, either due to reaching a maximum number of out 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 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**

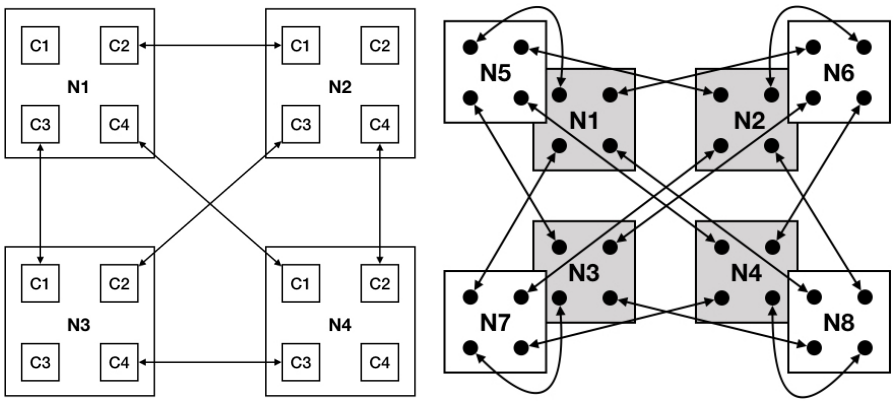
Consider the reverse protocol, which we call *Node Remote*. That is, each process participates in a remote exchange with all cores matching its core offset in the first round of communication, followed by a local exchange at each node in the second. For each message held by the process at $(n, c)$ with 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. However, the opposite relationship is true for many-to-one communications, such as those of a gather or reduce operation. We will illustrate these relationships in greater detail in Section 5.3.

**NLNR**

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




(a) Intra-layer communication

(b) Inter-layer communication

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

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 the a message originating at node $n_1$ to 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 assigns to each edges a single core on each incident node. It is helpful to separate this clique Figure 5.2 illustrates a sample

It is helpful to visualize how 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 and 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 for $(n, c) \in [N] \times [C]$ 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.

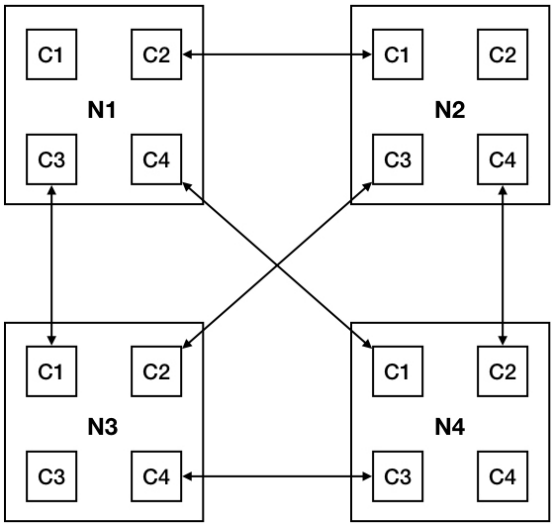



Figure 5.3: Example of the remote communication topology within a single layer of 4-core machines in the nlnr protocol.

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 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 $(\ell, \ell') \in [C]^2$.

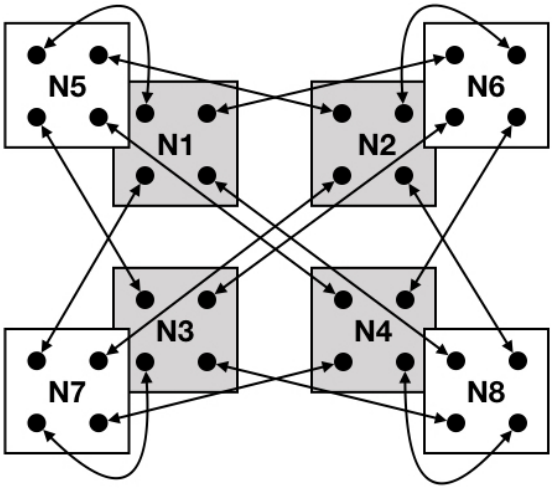



Figure 5.4: Example of the remote communication topology between two layers of 4-core machines in the nlnr protocol.

It turns out that this is, in fact, the minimum achievable channel size given our constraint that each node communications directly with each other node.

## 5.3 Experiments

# Chapter 6

# DegreeSketch and local triangle count heavy hitters

# Chapter 7

# Distributed sampling of random walks, simple paths, and subtrees via fast $\ell_p$-sampling sketches

# Chapter 8

# Sublinear distributed $\kappa$-path centrality

## 8.1 Introduction and Related Work

### 8.1.1 Betweenness Centrality

Betweenness is a more recent, though still relatively old, measure of centrality 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

$$\mathrm{BC}(x) = \sum_{\substack{x \notin \{y,z\} \\ \lambda_{y,z} \neq 0}} \frac{\lambda_{y,z}(x)}{\lambda_{y,z}}. \tag{8.1}$$

An implementation of betweenness centrality must solve the all pairs *all* shortest paths problem, which is strictly more difficult than the all pairs shortest paths 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 com-

puting 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.

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.

### 8.1.2 $\kappa$-Path Centrality

Kourtellis et al. attempt to approximate the top-$k$ betweenness central vertices by way of relaxing its computation with the related $\kappa$-path centrality [KAS+13]. The authors define the $\kappa$-centrality of a vertex $x$ as the probability that a random simple path of length at most $k$ over all other source vertices $y$ passes through $x$. Kourtellis et al. provide a randomized algorithm for approximating the $\kappa$-path centrality of the vertices of a graph, and demonstrate that on real-world networks vertices with high approximate $\kappa$-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 all pairs all shortest paths problem. Finally, $\kappa$-Path centrality is desirable in that it depends only on $\kappa$-local features of the graph when considering any particular vertex.

### 8.1.3 Semi-Streaming Approximation of Betweenness Centrality Heavy Hitters via $\kappa$-Path Centrality

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 8.1.1, 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.

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 $\kappa$-path centrality scores, as discussed in Section 8.1.2.

Moreover, the existing $\kappa$-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 $\kappa$-path centrality approximation algorithm requires a static graph. Second, there is at present no known algorithm that can approximate $\kappa$-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 $\alpha$ is an accuracy parameter. Finally, while empirical correlation is promising, there are no theoretical guarantees that $\kappa$-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 $\ell_p$-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 $\kappa$-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(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.

## 8.2 Sublinear Distributed $\kappa$-Path Centrality

# Bibliography


[A+13] Alexandr Andoni et al. Eigenvalues of a matrix in the streaming model. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1729–1737. Society for Industrial and Applied Mathematics, 2013.

[AC06] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 557–563. ACM, 2006.

[Ach01] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 274–281. ACM, 2001.

[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.

[AH14] Farhad Pourkamali Anaraki and Shannon Hughes. Memory and computation efficient pca via very sparse random projections. In *International Conference on Machine Learning*, pages 1341–1349, 2014.

[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.

[BKMM07] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *International Work-*

*shop on Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[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.

[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.

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

[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.

[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.

[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.

[CW09] Kenneth L Clarkson and David P Woodruff. Numerical linear algebra in the streaming model. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 205–214. ACM, 2009.

[CW17] Kenneth L Clarkson and David P Woodruff. Low-rank approximation and regression in input sparsity time. *Journal of the ACM (JACM)*, 63(6):54, 2017.

[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.

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

[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.

[GPW12] Anna C Gilbert, Jae Young Park, and Michael B Wakin. Sketched SVD: Recovering spectral features from compressive measurements. *arXiv preprint arXiv:1211.0361*, 2012.

[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.

[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.

[Kle99] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[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.

[KN14] Daniel M Kane and Jelani Nelson. Sparser johnson-lindenstrauss transforms. *Journal of the ACM (JACM)*, 61(1):4, 2014.

[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. On sketching matrix norms and the top singular vector. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1562–1581. Society for Industrial and Applied Mathematics, 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.

[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.

[NN14] Jelani Nelson and Huy L Nguyên. Lower bounds for oblivious subspace embeddings. In *International Colloquium on Automata, Languages, and Programming*, pages 883–894. Springer, 2014.

[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.

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

[Sar06] Tamas Sarlos. Improved approximation algorithms for large matrices via random projections. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 143–152. IEEE, 2006.

[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.

[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.

[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.

[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.

[W+14] David P Woodruff et al. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 10(1–2):1–157, 2014.

[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.

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