Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science &
Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

EFFICIENT AND SCALABLE TRIANGLE CENTRALITY ALGORITHMS IN THE ARKOUDA FRAMEWORK

by
Joseph Thomas Patchett

Graph data structures provide a unique challenge for both analysis and algorithm development. These data structures are irregular in that memory accesses are not known *a priori* and accesses to these structures tend to lack locality.

Despite these challenges, graph data structures are a natural way to represent relationships between entities and to exhibit unique features about these relationships. The network created from these relationships can create unique local structures that can describe the behavior between members of these structures. Graphs can be analyzed in a number of different ways including at a high level in community detection and at the node level in centrality. Both of these are difficult to quantitatively define because a "correct" answer is not readily apparent. The centrality of a node can be subjective; what does it mean central in an amorphous data structure? Further, even when centrality or community detection can be defined, there are typically trade offs in detection and analysis. A fine grained method may yield a more precise method but the run time may scale exponentially or even beyond. For small datasets this may not be a concern but for graph datasets this can make analysis prohibitive considering a social media networks where there are millions of people with millions of connections. Based on these two criteria, we implement several versions of a recently designed centrality measure called Triangle Centrality which is a centrality metric that considers both connectivity of a node with other nodes and the connectivity of a node's neighbors. The connectivity is aptly measured through the triangles formed by nodes. There are two ways to implement triangle centrality; a graph based approach and an approach that utilizes linear algebra and

matrix operations. This implementation is done with graph based data structures and to optimize this, we implement several versions of triangle counting based on prior research into the high performance computing framework, Arkouda. We implement an edge list intersection, a minimized search kernel method, a path merge method, and a small set intersection method. To compare these methods, we include a naive method and a comparison to a linear algebra implementation that uses the SuiteSparse GraphBLAS library.

Our implementation utilizes an open-source framework called Arkouda which is a distributed platform for data scientists and developers. It simplifies complex parallel algorithms and the storage of datasets onto a back end Chapel server and allows users to access these from an intuitive pythonic interface. Our results demonstrate the scalability of the platform and are analyzed against different graph properties to see how these affect the implementation.

EFFICIENT AND SCALABLE TRIANGLE CENTRALITY ALGORITHMS IN THE ARKOUDA FRAMEWORK

by
Joseph Thomas Patchett

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

August 2022

APPROVAL PAGE

EFFICIENT AND SCALABLE TRIANGLE CENTRALITY ALGORITHMS IN THE ARKOUDA FRAMEWORK

Joseph Thomas Patchett

David A. Bader, Dissertation Advisor | Date
Distinguished Professor of Data Science, New Jersey Institute of Technology

---

Ioannis Koutis, Committee Member | Date
Associate Professor, New Jersey Institute of Technology

Zhihui Du, Committee Member | Date
Research Scientist, New Jersey Institute of Technology

BIOGRAPHICAL SKETCH

Author: Joseph Thomas Patchett

Degree: Master of Science

Date: August 2022

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Master of Science in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 2022
- Bachelor of Science in Materials Science and Engineering
  The Pennsylvania State University, University Park, PA, 2016

Major: Computer Science

Presentations and Publications:

Z. Du, O. Alvarado Rodriguez, J. Patchett, D. Bader, "Interactive Graph Stream Analytics in Arkouda," *Algorithms*, vol. 14, 2021.

Joseph Patchett, "K-Truss Implementation in Arkouda (Extended Abstract)." *Conference*, IEEE HPEC, Virtual, 2021.

This work is dedicated to several people. First and foremost, to my family who have supported me in all of my endeavors and have helped me get to where I am today. Their unquestioning support and love are beyond measure and I cannot even begin to thank them. Secondly, my beloved girlfriend, Leah, who has been with me the entire way and helped me work many long nights on my school and research work. Her support has meant the world to me.

ACKNOWLEDGMENT

I would like to thank my advisor Dr. Bader for his patience, providing constructive feedback, and giving me direction. I would like to thank Dr. Zhihui Du from my research group and committee. Zhihui Du always gave me excellent technical knowledge and was happy to jump on calls with me. Bouncing ideas off of him was invaluable in all research efforts I worked on. I want to thank Dr. Ioannis Koutis, who provided guidance and a unique perspective on my research. I would like to thank Timothy Hart at NJIT for guiding me in this journey. I could not have done anything I did here without his help and support. Balancing all three of those is not an easy task but Dr. Bader manages it deftly. I would like to thank everyone in our research group as well especially Oliver Alvarado Rodriguez and Fuhuan Li. They have been great friends for me at NJIT; not only are extremely intelligent individuals who helped greatly with my research but they are also kind and warmhearted. Finally, I want to acknowledge all the supporting administrative staff at NJIT whose unsung (yet highly valuable) work has helped me get here.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1
  1.1 Motivation 1
  1.2 Graph Definitions 2
  1.3 Arkouda 2
    1.3.1 Architecture 3
  1.4 Graph Representations 3
  1.5 Centrality as a Metric 5
  1.6 Graph Topology 8
2 GRAPH DATASETS 11
  2.1 Background 11
  2.2 Graph Datasets 11
    2.2.1 Real World Networks 12
    2.2.2 Synthetic Datasets 16
    2.2.3 Dataset Properties 16
3 RELATED WORK 18
  3.1 Graphs in Arkouda 18
  3.2 Triangle Counting 20
  3.3 Triangle Centrality 24
4 EXPERIMENTS AND RESULTS 27
  4.1 Algorithms 27
    4.1.1 Naive Method 29
    4.1.2 Minimized Triangle Search 30
    4.1.3 Edge List Intersection 31
    4.1.4 Path Merge Method 33
    4.1.5 Small Set Intersection 35

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4.2 Experimental Setup 35

4.3 Results and Discussion 36

5 CONCLUSION 42

APPENDIX A GRAPH PROPERTIES RELATIONSHIP FIGURES 44

APPENDIX B DEGREE DISTRIBUTIONS OF GRAPHS 47

B.0.1 Real World Graphs 47

B.0.2 Delaunay Graphs 54

BIBLIOGRAPHY 58

LIST OF TABLES

Table **Page**

2.1 Dataset Descriptions 17

4.1 Graph Processing Times (sec) 37

LIST OF FIGURES

Figure		Page
1.1	Double index data structure.	4
2.1	Log-log plot of ca-Astro.	12
2.2	Log-log plot of ca-CondMat.	13
2.3	Log-log plot of ca-GrQc.	13
2.4	Log-log plot of ca-HepPh.	14
2.5	Log-log plot of ca-HepTh.	14
2.6	Log-log plot of email-enron.	15
2.7	Log-log plot of loc-brightkite.	16
3.1	Delaunay build efficiency.	18
3.2	Rgg build efficiency.	19
3.3	Kron build efficiency.	19
4.1	Log of edge processing speed vs. assortativity.	38
4.2	Log of edge processing speed vs. fraction of closed triangles.	39
4.3	Arkouda edge processing speed vs. assortativity.	40
4.4	Arkouda edge processing speed vs. fraction of closed triangles.	40
4.5	Arkouda edge processing speed vs. 90 percent diameter.	41
A.1	Log of edge processing speed vs. average cluster coefficient.	44
A.2	Log of edge processing speed vs. power law exponent.	44
A.3	Log of edge processing speed vs. diameter.	45
A.4	Arkouda edge processing speed vs. average cluster coefficient.	45
A.5	Arkouda edge processing speed vs. power law exponent.	46
B.1	p2p-gnutella04 degree distribution.	47
B.2	p2p-gnutella05 degree distribution.	48
B.3	p2p-gnutella06 degree distribution.	48
B.4	p2p-gnutella08 degree distribution.	49

LIST OF FIGURES
(Continued)

Figure **Page**

B.5 p2p-gnutella09 degree distribution. 49

B.6 p2p-gnutella24 degree distribution. 50

B.7 p2p-gnutella30 degree distribution. 50

B.8 p2p-gnutella31 degree distribution. 51

B.9 Oregon1_010331 degree distribution. 51

B.10 Oregon1_010407 degree distribution. 52

B.11 Oregon1_010414 degree distribution. 52

B.12 Oregon1_010421 degree distribution. 53

B.13 Oregon1_010428 degree distribution. 53

B.14 Delaunay_n10 degree distribution. 54

B.15 Delaunay_n11 degree distribution. 54

B.16 Delaunay_n12 degree distribution. 55

B.17 Delaunay_n13 degree distribution. 55

B.18 Delaunay_n14 degree distribution. 56

B.19 Delaunay_n15 degree distribution. 56

B.20 Delaunay_n16 degree distribution. 57

B.21 Delaunay_n17 degree distribution. 57

CHAPTER 1

INTRODUCTION

1.1 Motivation

The world has become exponentially more connected over the past few decades. From Usenet, the early days of the internet to its rapid adoption culminating in the rise of social media, personal cell phones, and every other advancement in technology devices, the amount of data we create every day has exploded. From this explosion of data comes opportunity; software engineers are needed to build out phone and web apps using the ever expanding library of technology, marketing managers now have different ways to measure and reach out to new audiences, and the entire career of data scientists have come to being.

Correspondingly, datasets have increased exponentially if not even more relative to the growth of technology. Handling the massive datasets created by all these new technological marvels can prove to be a challenge given that devices are gated by their storage capacity and memory. Data scientists in particular may run into problems when working with large datasets; their personal devices will not be able to store these large and growing datasets in memory, let alone handle the space complexity that comes with advanced machine learning algorithms like deep learning. Even for datasets that fit on their machine, algorithms like those used in analytics for graphs in cybersecurity, traffic, and knowledge graphs may take a prohibitive amount of time to run.

Distributed systems solve both of these problems, a supercomputer will be able not only to load large datasets, but also be able to rapidly perform operations on that dataset. Executing operations in a parallel fashion can reduce the running time by a significant margin. With respect to graph data, calculating the centrality of

every node in a graph could be prohibitive if done sequentially, but in parallel those operations can be spread across multiple machines operating on multiple nodes in a graph at once.

1.2 Graph Definitions

The graph data structure is made of two components: edges and nodes. The latter are defined by an index and edges are connections between nodes. Consider two nodes, u and v, if there is an edge between these two nodes that edge is defined by those two nodes. In undirected graphs, this edge is bidirectional meaning that the edge goes from both u to v and from v to u. In a directed graph, this edge is unidirectional such that an edge (u, v) is only an edge from u to v. Additionally, edges can have weights associated with them. The degree of a node is the number of edges that contain that node. This definition of degree is important to understand for real world graphs later on.

The amorphous nature of this results in many different types of graph topologies. These will be explained in detail later on.

1.3 Arkouda

Arkouda is an open source framework designed to bridge the gap between the limitations of personal computers and the ever-growing size of datasets. Originally developed by Michael Merrill and William Reus [28], Arkouda is guided by the principles of scalability and performance, an intuitive user interface, and productivity. This leads to productivity not just for the data scientist but also for the developer of algorithms within Arkouda. Arkouda's back end server utilizes Chapel to speed development of parallel algorithms. Chapel [12] is also an open-source compiler and runtime system originally developed under the DARPA High Productivity Computing

Systems (HPCS) program. The front end Python and the back end Chapel are coupled by a ZeroMQ [23] middleware.

1.3.1 Architecture

As discussed above, Arkouda's architecture has 3 main components; a back end Chapel server, a ZeroMQ middleware, and a Python front end. Datasets have grown massively over time, from megabytes to terabytes and soon some may reach the petabyte range. Traditionally, many data scientists have worked from their laptops using Python frameworks such as NumPy and Jupyter notebooks. Arkouda leverages the paradigms in those frameworks to build a front end that is intuitive and similar to other tools. The massive datasets and complex parallel algorithms are abstracted away from the practitioner allowing for seamless access from their personal device. Chapel is a high level programming language with built in abstractions for parallel programming without sacrificing performance. Chapel was designed with several characteristics in mind: productivity, scalability, speed, portability, and open-source usage. ZeroMQ is a framework for controlling the conversation between the Python front-end and the Chapel back-end of Arkouda. This middleware allows for the server to respond with the results of the back end and completely abstracts away the parallel algorithms and the massive datasets.

1.4 Graph Representations

Graphs can be represented in a number of ways including adjacency lists, adjacency matrices, compressed sparse rows, and from these, many different graph data structures arise. The adjacency matrix is an n by n matrix where n is the number of nodes. Each (i, j) value in this matrix refers to a potential edge in the graph. For an undirected graph with no edge weights, a 1 at (i, j) indicates an edge between the two nodes, (i, j). In the case of a directed graph, value at (i, j) only indicates an edge

from (i, j). For weighted edges, the cell in the matrix can hold the weight of an edge in both directed and undirected graphs.

An adjacency list maintains the neighbors for each edge in a linked list format. Each node, u in a graph G will be linked only to its neighbors. This limits the space complexity caused by a sparse graph and allows for new nodes to be added much faster but increases the complexity of adding in an edge between existing nodes.

I will focus primarily on the data structure in Arkouda and how this compares to other implementations. Du *et al* [19] implemented a double index data structure which forms the backbone for graphs in Arkouda.

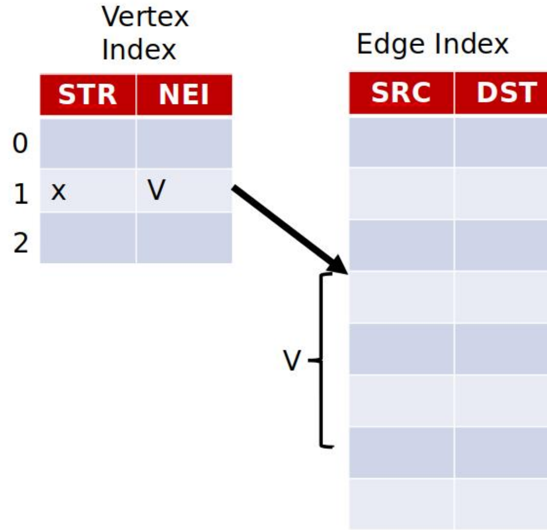

Figure 1.1 Double index data structure.

The first double vector contains every node in the graph G. The nodes are represented by the indices of the these vectors. The first entry contains the starting index for that node's edges in the edge index array. The next value is the number of neighbors V for each u in G. These two values together gives a direct reference to that node's edge list in the edge array. To be explicit, for a node u, the edges that contain that node start at the index x in the edge index array. That edge index then has a number of edges V which contain u.

There are a number of benefits to using this method. The adjacent edges of a node can be accessed in $O(1)$ time which allows for quick triangle counting. Because the edge list is also stored, operations on edges can be load balanced on the edge list instead of the nodes. This strategy will improve the efficiency with real world graphs that tend to have a skewed distribution.

1.5 Centrality as a Metric

Measuring the importance of a node in a network has been extensively researched. The simplest being degree centrality; which in an undirected graph is ranking each node by the number edges that node is included in. This calculation was originally by Nieminen *et al* [30] done using an adjacency matrix and is the sum of the number of edges for that node. There are some problems with this heuristic; the existence of an edge is not a guarantee of importance. Consider a negative example: movie actors and their agents. A movie actor may have a massive network of people that are influenced by their actions and they have an agent that manages their brand. These agents may not have broad influence themselves, but they may be the connected to many actors and they may be the only connection between actors. A node with a low degree is not guaranteed to have a low impact. A fake profile on a social network could send out tons of friend requests and acquire a large friend network just for the sake of having a high friend count. Further, one of the hallmarks of a community is interaction within that community; if friends of friends don't interact, is that really a good measure of community structure? Network topology plays a strong role in identifying centrality; structure of the surrounding graph plays a role in the importance of a node. This paradigm is proven in later work.

The large number of different centrality metrics emerges from the many differences in centrality paradigms. The above example illustrates an analytical reason why the choice of centrality metric is a challenge. The second challenge is finding

an algorithm that either has a low computational complexity or at least is easily parallelizable. Within these, centrality metrics can be categorized into geometric distance like closeness centrality, spectral measures like PageRank, and path metrics that utilize the paths going through a node like betweenness centrality.

The closeness centrality [4] takes into account the nearness of other nodes. In other words, the closeness centrality considers the distance between a given node and the other nodes in the network. This is formally represented as the inverse of the sum of all the distances shown below.

$$CC(u) = \frac{n-1}{\sum_{w \in V} d(u,w)}$$

This metric considers the entire graph now and the impact of each node on another. There are still several questions that arise from this method; calculating the closenesss centrality for single node is done in $O(mn)$ time where m is the number of edges and n is the number of nodes, and we still see that high degree nodes have a disproportionate impact on the centrality ranking. In Saxena *et al* [34] this complexity can be reduced to approximately a scalar multiple of $O(m)$ but is an estimation and not an exact metric. Even with the powerful tools available to data scientists in Arkouda, an $O(mn)$ run time can be prohibitive.

Building off of the work done for closeness centrality, Freeman [22] developed betweenness centrality as a new metric. Instead of calculating the distance between each node, the betweenness centrality considers the number of shortest paths through two other nodes i and j and the number of times node u is in those paths given that $u \neq i \neq j$.

$$BC(u) = \sum_{u,i,j \in V} \frac{\sigma_{ij}(u)}{\sigma_{ij}}$$

This property calculates the flow of a network and potentially penalizes nodes with higher degrees but more redundancy. The idea is similar to percolation theory in power networks; how likely will removing a node change the flow of

electricity/information? Nodes with high ratios will disrupt more shortest paths if removed. Again though, while this may capture network topology better than other metrics, calculations are still done in $O(mn)$ time which can be prohibitive.

Utilizing adjacency lists allows for matrix operations on graphs. The Eigenvector centrality, developed by Bonacich [9], considers the centrality of each other node. For each node u, in an adjacency matrix A and with eigenvalues λ, its centrality is defined as the value in u-th index of a vector x where x is defined as follows.

$$Ax = \lambda x$$

Vertices are penalized for connections to low degree vertices and rewarded for connections with high degree vertices As with the other metrics, high degree nodes have a disproportionate impact on their neighbor's centrality. Further, eigenvectors are calculated on adjacency matrices and in scale-free graphs these require matrices of massive dimensions. Eigenvector centrality provides the basis for the next extremely famous centrality metric: PageRank.

Powering Google's search engine; PageRank [31] is a normalized form of eigenvector centrality. Illustrating this method is best done using a web link example. The centrality or authority of a page is determined by the rank of each incoming page into it divided by the number of outgoing links from each respective page. Additionally there is a damping factor which intuitively can be understood as an individual's propensity to traverse through a limited number of pages before they reach their desired web page. The formula is shown below.

$$PR(u) = \frac{1-d}{N} + d * \sum v \in N(u) \frac{PR(v)}{k_v}$$

Because of the changing influence of a web page on another, this metric does not converge in a single iteration. The harmonic centrality [8] of a node u is the sum of the reciprocal distances from every other vertex v in the connected component to u. This metric was developed to solve the problem of multiple connected components in

graph data sets and bounds the centrality into an easily understandable maximum of $n - 1$. Formally, this is defined as follows:

$$HC(u) = \sum_{v \in V and v \neq u} \frac{1}{d(u,v)}$$

Higher values indicate greater centrality. In graphs with a large number of connected components, harmonic centrality penalizes smaller components because of the smaller impact of each node. This method has the same time complexity, $O(mn)$ of the closeness centrality because it calculates the distance for every node to each other within the component.

1.6 Graph Topology

We employ several intrinsic features of a graph to examine how graph properties can affect the performance of different implementations of Triangle Centrality.

The assortativity [29] of a graph is defined at a high level is the proclivity of a vertex to have an edge with other vertices with similar degrees. Formally this is the Pearson correlation coefficient, r, of the degrees of each edge over all edges and is defined as follows:

$$r = \frac{1}{\sigma_q^2} \sum_{jk} jk(e_{jk} - q_j q_k)$$

Where j and k are the degrees of each vertex in an edge, q_k and q_j are the normalized degree distribution sans the opposite vertex, e_{jk} is the normalized probability distribution of the two vertices in an edge, and σ^2 is the variance of the distribution. A positive value indicates that high degree vertex will be comparatively more likely to form an edge with another vertex of a similar degree, i.e. high degree vertex to high degree vertex and low degree vertex to low degree vertex. A negative value indicates the opposite; vertices preferentially form an edge with vertices with disparate degrees, i.e. high degree to low degree and vice versa.

Many degree distributions in real-world graphs follow a power law distribution [3]. This means that for any vertex in a graph and a degree k, the probability that this vertex has this degree is given by the following:

$$p(k) = k^{-\gamma}$$

Where γ is the degree exponent of the graph and a higher value results in a tighter distribution. Practically, this results in a skewed distribution of degrees. A large number of vertices will have a small degree and a small number of degrees will have a large degree. This creates some load balancing challenges because of the skewed distribution.

We also investigate three other much popular graph properties. The average clustering coefficient measures how likely vertices in a wedge form a triangle or more broadly, how likely vertices are to group together. Higher values means a higher proclivity towards this behavior. The fraction of closed triangles is the ratio of closed triangles over the number of 2 paths in a graph. These are measures of density in a graph. The diameter of a graph is maximum shortest path between two vertices in a graph. The 90th percentile diameter captures the smallest distance such that 90 percent of nodes are that distance from each other. This metric gives a more complete view of a graph's density because it avoids the impact of outliers. For graphs with multiple connected components, only the largest is used in the metric.

In this thesis, we explore the insights of how graph algorithm and property can affect the normalized performance by employing four different triangle centrality algorithms and five typical graph properties.

There has been some work done on the effect of graph topology on graph algorithms but not directly on the properties investigated here. Blanco *et al* [7] explore how k-truss performs on different types of graphs based on the number of edges, maximum degree, average degree, and several other aspects. The k-truss [16]

of a graph is the maximal subgraph such that each edge is incident to $k-2$ triangles. The k-truss is a strong comparison since triangle counting comprises the bulk of the computations similar to triangle centrality. They also demonstrate that tuning an algorithm to the specific types of graphs can improve the performance. Pearce *et al* [33] use edge list partitioning to implement k-core, triangle counting and breadth first search (BFS). To analyze their results they graph the edges processed per second as a function of diameter in BFS and the run time as a function of the maximum vertex degree in triangle counting. These are very intuitive metrics to consider and we extend some of these paradigms into our own work.

CHAPTER 2

GRAPH DATASETS

2.1 Background

Before diving in to the specific datasets used, it is important to take a brief foray into the some of the characteristics of real world graphs. The degree distribution of a graph is one of the ways that real world graphs are distinguished. The degree distribution of a graph is a histogram of the number of nodes that have an equivalent degree.

Random degree values in this distribution will follow what is known as a Poisson distribution. However, this was shown not to be the case for networks like the World Wide Web in by Jeong *et al* [2]. Instead of following a Poisson distribution, the degree distribution of the graph followed a power law and this is the case for most real world graphs [35, 1, 21]. In a log-log plot, this can be seen as a straight line with a negative slope.

Practically, this results in a smaller number of nodes having a large degree and a large number of nodes having a small degree. This presents unique optimization problems especially when counting triangles, the choice of how to iterate through adjacency lists is important. Some lists will have a large amount of edges which increases the number of iterations required.

2.2 Graph Datasets

Our experiments use a number of real world and synthetic datasets. Within these datasets there are a few categories of types of datasets that will be utilized.

2.2.1 Real World Networks

The first set of real world datasets are collaboration networks where edges are created when authors or coauthors write a paper together. The first is ca-AstroPh which is a dataset that contains the contributions to the Astro Physics category of arxiv [25]. Authors and coauthors represent nodes in the graph and the edges represent collaboration in papers between them. There are several others of these types of citation networks including ca-CondMat which is Condense Matter, ca-GrQc which is General Relativity and Quantum Cosmology, ca-HepPh which is High Energy Physics - Phenomenology, and ca-HepTh which is High Energy Physics - Theory.

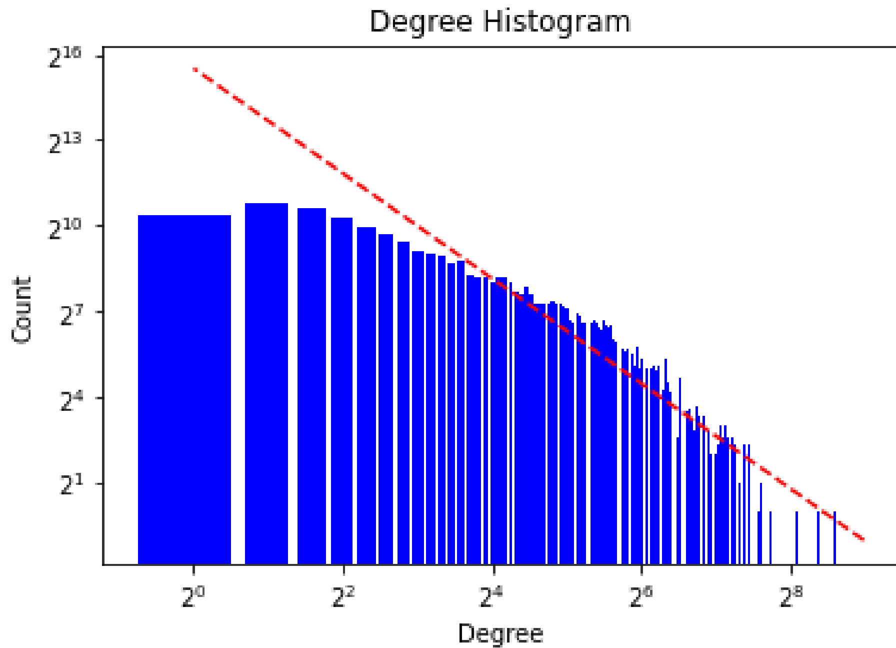

Figure 2.1 Log-log plot of ca-Astro.

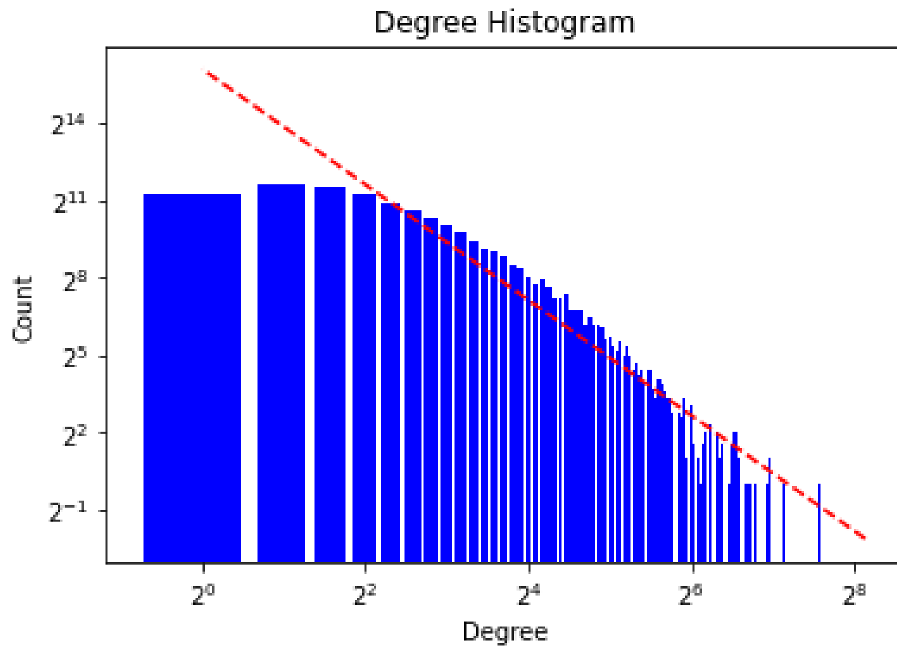

Figure 2.2 Log-log plot of ca-CondMat.

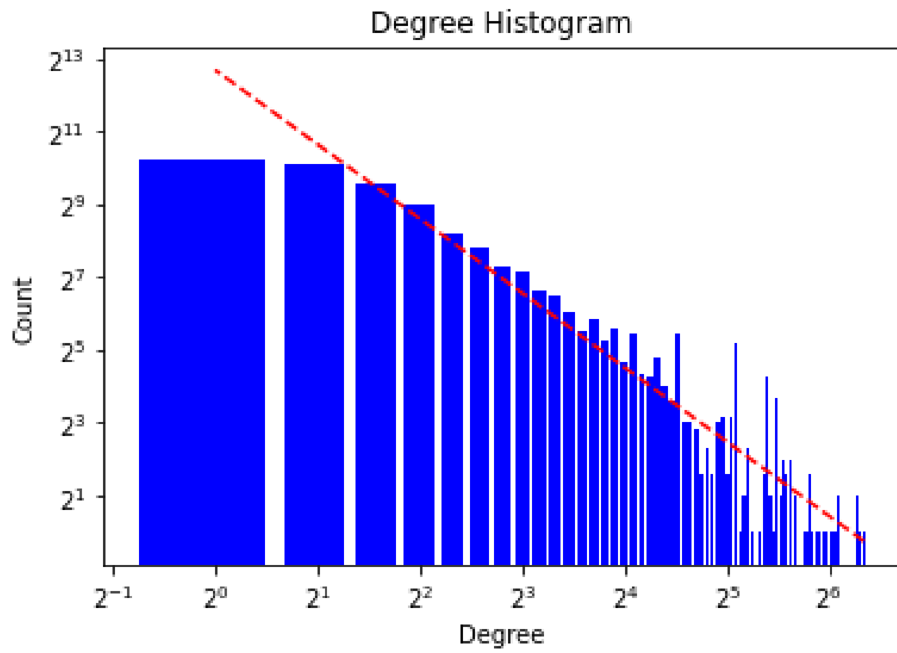

Figure 2.3 Log-log plot of ca-GrQc.

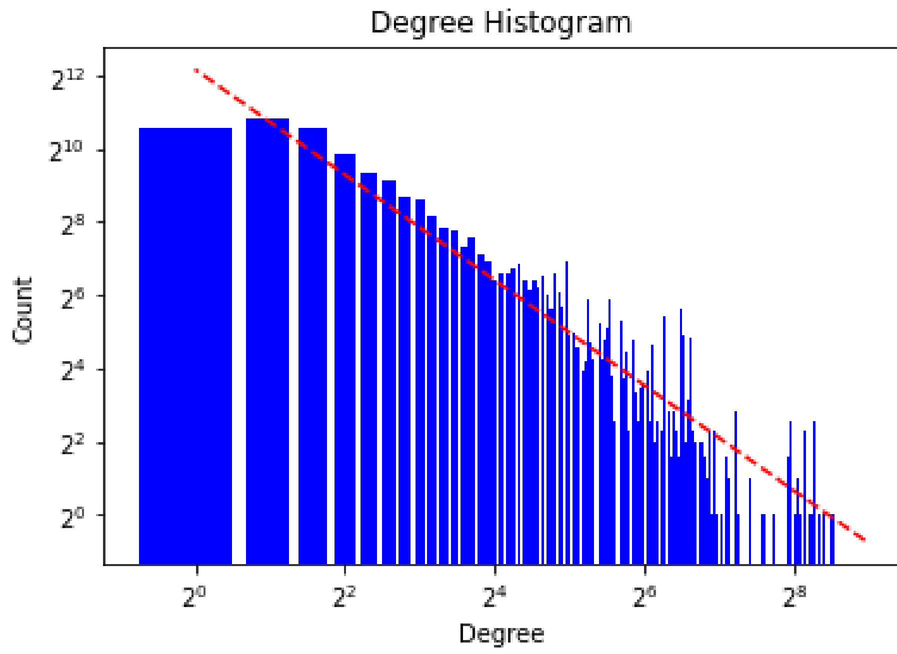

Figure 2.4 Log-log plot of ca-HepPh.

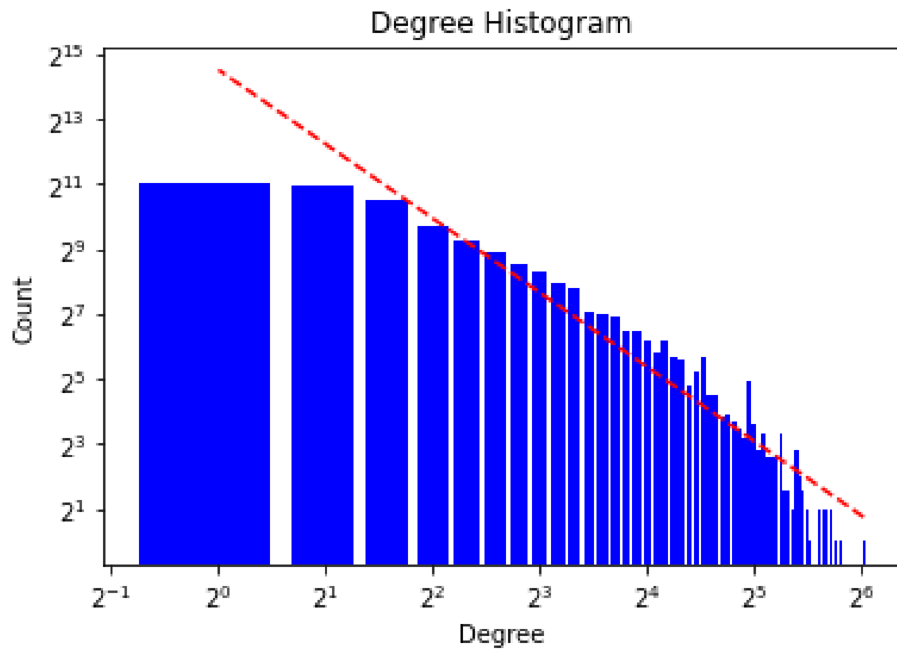

Figure 2.5 Log-log plot of ca-HepTh.

For each of these figures 2.1, 2.2, 2.3, 2.4, 2.5, the count of each nodes at a certain degree is plotted. Note that each plot is a log-log plot; for power datasets, when the degree distribution is plotted in this manner, the slope is a straight line.

The Email-Enron dataset [26] is derived from the collection of emails that were made public by the Federal Energy Regulatory Commission during its investigation of Enron. In this dataset, people represent nodes, and the emails between them represent edges.

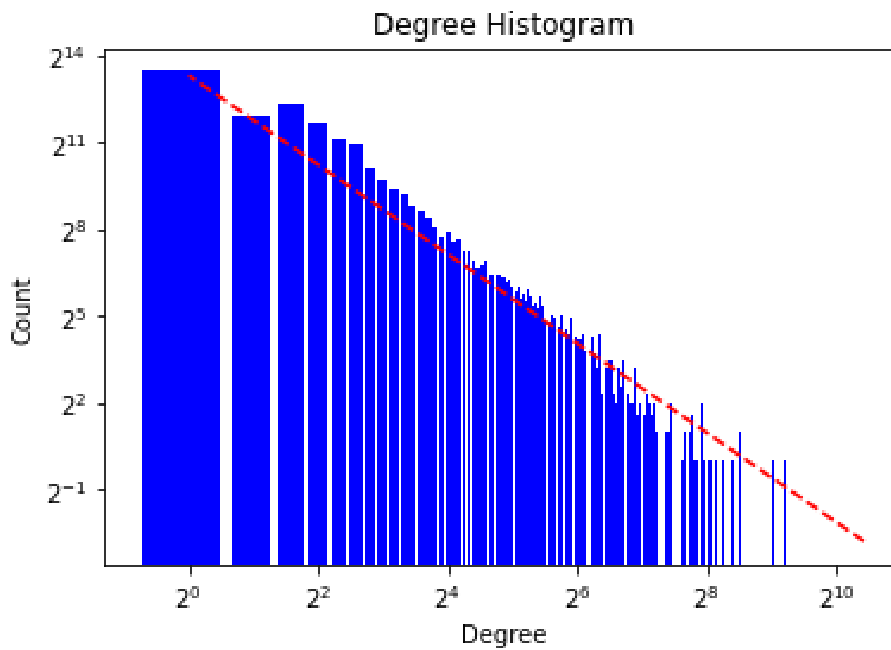

Figure 2.6 Log-log plot of email-enron.

The loc-brightkite dataset[13] was created from an old social network called “Brightkite” that connected users based on their location. Friendships in this social media network were directed, but an undirected network was derived and provides the basis for this dataset.

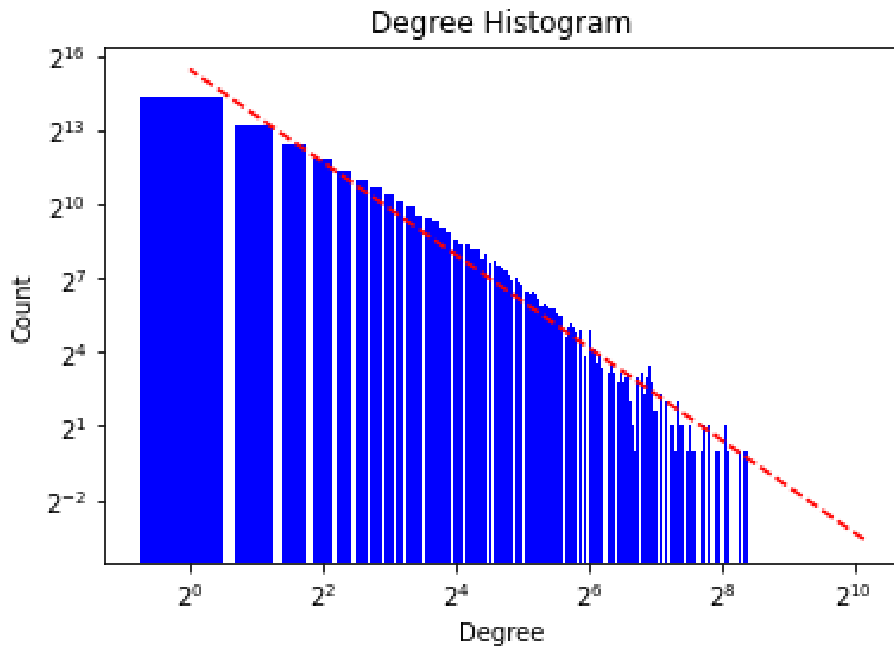

Figure 2.7 Log-log plot of loc-brightkite.

The as-caida[24] dataset was built from autonomous systems. This maps business arrangements between ISPs and each individual system is a node. Edges are derived from several relationships, customer to customer, peer to peer, and sibling to sibling.

2.2.2 Synthetic Datasets

Synthetic datasets are those that are generated algorithmically and as such, graphs generated with the same algorithm will often have the same structure. For the purposes of our experiments, we utilize the Delaunay graphs for comparison of run times and provide another level of comparison for the algorithms.

2.2.3 Dataset Properties

Table 2.1 Dataset Descriptions

Graph Name	Edges	Vertices	Average Cluster Coefficient	Fraction of Closed Triangles	90th Percentile Diameter	Assortativity	Power Law Exponent
as-caida20071105	53381	26475	0.2082	0.002452	4.7	-0.194646	2.50865
ca-AstroPh	198050	18772	0.6306	0.1345	5	0.205129	1.494
ca-CondMat	93439	23133	0.6334	0.107	6.5	0.133955	2.23
ca-GrQc	14484	5242	0.5296	0.3619	7.6	0.659325	1.44
ca-HepPh	118489	12008	0.6115	0.3923	5.8	0.632275	2.29
ca-HepTh	25973	9877	0.4714	0.1168	7.4	0.267821	2.04
email-Enron	183831	36692	0.497	0.03015	4.8	-0.167768	2.651
facebook_combined	88234	4039	0.6055	0.2647	4.7	-0.668214	1.54
loc-brightkite_edges	214078	58228	0.1723	0.03979	6	0.0108158	1.88
Oregon1_010331	22002	10670	0.297	0.003121	4.4	-0.1863	1.14
Oregon1_010407	21999	10729	0.2921	0.00286	4.5	-0.1889	1.11
Oregon1_010414	22469	10790	0.2954	0.00316	4.4	-0.1937	1.11
Oregon1_010421	22747	10859	0.2968	0.003258	4.4	-0.1932	1.12
Oregon1_010428	22493	10886	0.294	0.002991	4.4	-0.195	1.11
p2p-Gnutella04	39994	10876	0.005	0.0018	5.5	-0.013	1.66
p2p-Gnutella05	31839	8846	0.008	0.0025	5.5	0.014	1.67
p2p-Gnutella06	31525	8717	0.008	0.0027	5.5	0.051	1.67
p2p-Gnutella08	20777	6301	0.02	0.0069	5.5	0.035	1.753
p2p-Gnutella09	26013	8114	0.017	0.0058	5.7	0.033	1.77
pnp-Gnutella24	65369	26518	0.0055	0.001371	6.1	-0.007728	1.985
pnp-Gnutella25	54705	22687	0.0053	0.001516	6.3	-0.172839	1.993
pnp-Gnutella30	88328	36682	0.0063	0.001727	6.6	-0.10337	2.03568
p2p-Gnutella31	147892	62586	0.0038	0.0013	6.8	-0.092	2.06

CHAPTER 3

RELATED WORK

3.1 Graphs in Arkouda

There has been significant work on Arkouda. Due to its novelty, the first steps were establishing the viability of the framework. Du *et al* [19] introduce two breadth first search papers; a high level algorithm and a low level one. The most important conclusion that can be drawn is that the DI data structure and data structures intrinsic to Chapel can be exploited to get high performance.

For Triangle Centrality and other Arkouda algorithms, they demonstrate how graphs are built in Arkouda. Building the graph incurs overhead on all Arkouda algorithms and efficient build times are important to prove. The efficiencies of Arkouda is shown on these synthetic graphs.

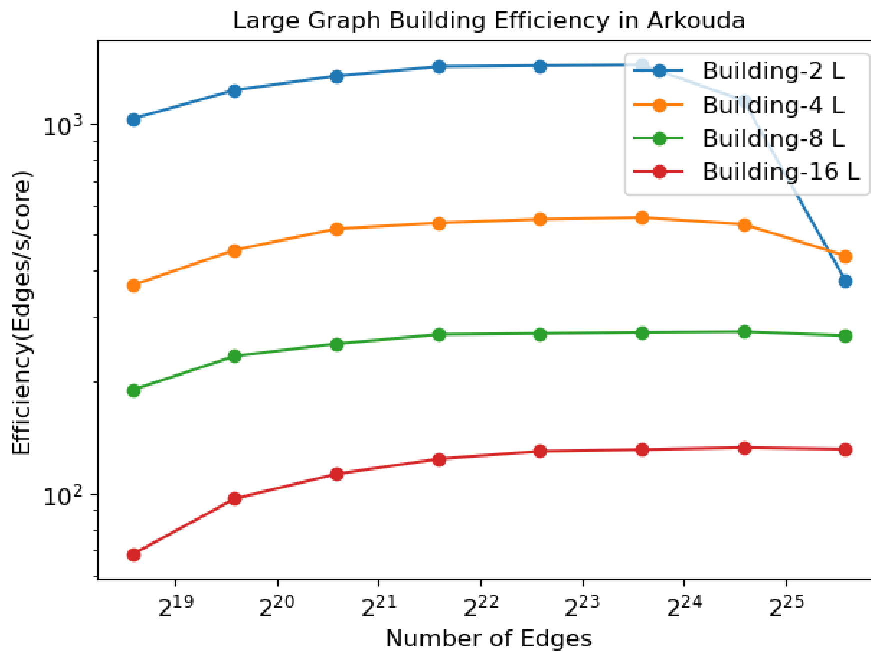

Figure 3.1 Delaunay build efficiency.

Source: [19]

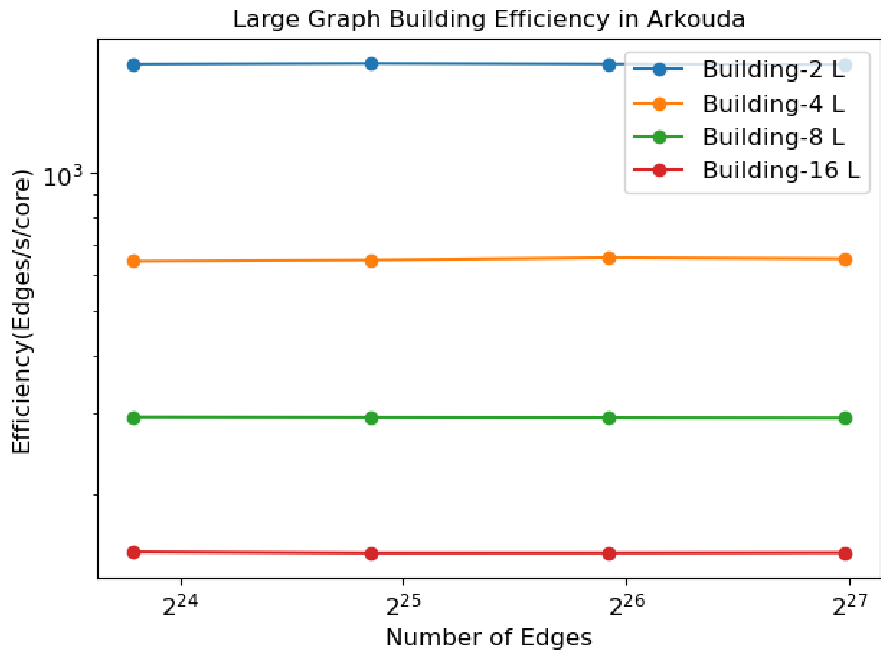

Figure 3.2 Rgg build efficiency.

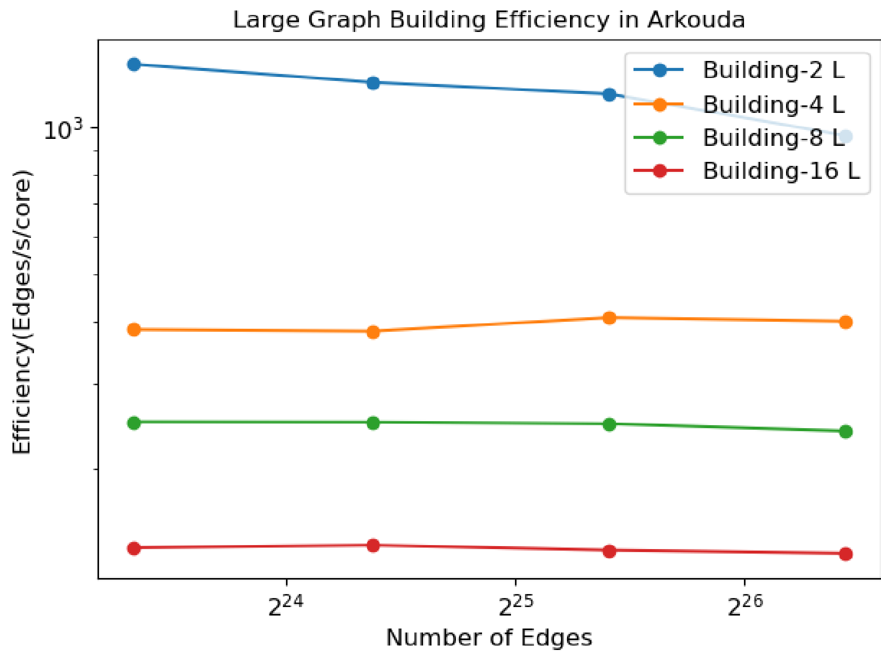

Figure 3.3 Kron build efficiency.

In Fig. 3.1 as the number of edges increase, the efficiency of the build stays constant across larger locales. Larger locales have a smaller efficiency but this can be explained because there are more communications across locales required as the number increases. Despite this, larger number of locales are more stable as higher number of edges are added as shown in the decrease in efficiency from the 2 and 4 locales.

The Figures 3.2 and 3.3, the difference is not quite as stark, but the 2-locale line in the Kron build graph begins to show this behavior at larger edge counts. Because Arkouda was built to handle terabytes and beyond of data, this behavior is encouraging.

3.2 Triangle Counting

Some graph algorithms include a form of graph isomorphism which would include comparing a simple structure across a graph. Due to their simplicity, triangles are often used for this type of isomorphism. Much work has been done on efficient triangle counting in Arkouda and otherwise. To start, I will go through other work and end with the current work done in Arkouda.

The GraphChallenge[1] provides many benchmarks for triangle counting and many other types of algorithms in different types of environments including shared memory, distributed systems, and GPUs.

Shared memory methods while fast and efficient can't handle larger graphs simply because these graph can't be held in memory. Nevertheless, exploring these may yield some insight into how optimizations can be done on distributed systems. Wolf *et al* [36] use a linear algebra based approach. According to their results, by leveraging the KokkosKernel they can achieve up to 670,000 times speed up from the C++ reference and 10,000 times speed when compared to Python. Improving upon

[1]https://graphchallenge.mit.edu/challenges

this using Cilk, Yaşar *et al* [37] get up to 7x efficiency compared state of the art implementations and show a $O(n)$ scalability when there is a momefnt of 4/3.

Other methods involve an efficient I/O method of triangle counting when the graph cannot be fully loaded into main memory. To counteract this problem and ensure fast output, Chu *et al* [14][15] utilize a graph partitioning method to handle massive graphs and requires only 1 or 2 passes of the graph to get a complete picture.

Chapel and Arkouda do not yet support GPUs natively but some inspiration may be taken from works in this area. Bisson *et al* [5] implement two algorithms: a k-truss method and a triangle counting method. The former requires reassessing the number of triangles for each edge to determine which edges can be removed and because of this an efficient triangle counting method is required for optimal performance of the k-truss algorithm. Utilized in this operation are set operations on bitmaps. After this, in an update [6] they utilized a new GPU architecture and a new method for adjacency list compaction to get significant improves upon their old method. Their novel compaction method reduces the variance in the adjacency lists and improves their ability to load balance critical steps of triangle counting in both methods. Pandey *et al* [32] utilize an h-index sorting method, they use a comparatively larger neighbor list M and a smaller neighbor list N to create hash buckets from which the larger neighbor list can be iterated through. Their method avoids the overhead required by preprocessing of sorting a neighbor list.

There has been significant research in distributed triangle counting. Burkhardt [10] utilizes shared memory in a MapReduce model for an $O(m\sqrt{m})$ run time done on an adjacency matrix.

There are a couple of challenges when comparing these to an Arkouda implementation. The first is the difference between shared memory systems and distributed systems; shared memory methods do not have the overhead that results from communication between locales. The second limitation is that due to limitations

in Arkouda's data structures, arrays are not possible, so methods can only be implemented with 1 dimensional vectors.

The edge list intersection is the simplest algorithm to calculate the number of triangles in a graph. Simply, by comparing the adjacency lists of a node u and a node v, and the intersecting nodes, w, of this result make up the triangles created by edges of these 3 nodes.

Algorithm 1: High level Edge List Intersection

```
1 TriangleCount(G(N, E))
2 TCount = 0
3 for Node u in E do
4 |   for Node v, in N_π(u) do
5 |   |   for w > v in N_π(u) do
6 |   |   |   if < w, v > in E then
7 |   |   |   |   TCount+ = 1;
8 return TCount
```

Where in 1, the $N_\pi(u)$ is the adjacency list of u. This represents all nodes to which u has an edge. For any node v in this adjacency list, if there is also a node w in $N_\pi(u)$ There have already been methods to implement fast triangle counting in Arkouda [20]. Because triangle counting is critical to triangle centrality, a fast counting method is required. This method utilizes the double index data structure to provide an exact count of triangles in the graph.

In the above algorithm, the authors utilize the parallel abstractions inherent to Chapel. Specifically, this means utilizing the $coforall$ and the $forall$ key words to indicate that a process will be done in parallel. This is a form of an edge list intersection method where the adjacency lists of two nodes are combined and if a node matches, then a triangle is added. Concretely, for an edge $< u, v >$, they check

Algorithm 2: Edge Iterator Vertex Algorithm

```
1  Triangle_Count(G =< E, V >)
   /* G_sk is the given graph sketch partition, E_sk, V_sk are edge and
      vertex sets.                                                      */
2  var localeTriSum=0: [0..numLocales-1] int;//store each locale's number of
    triangles
3  coforall (loc in Locales) do
4     if (current loc is my locale) then
5        var triCount=0:int; //Local triangle count
6        Adjust StartVer and EndVer based on locale to cover all vertices
          and avoid overlapping
7        forall (u in startVer..endVer with (+ reduce triCount)) do
8           u_adj = {x| < u, x >∈ E_sk ∧ (x > u)} //build the u adjacency
             vertex set in parallel
9           forall v ∈ u_adj do
10             v_adj = {x| < v, x >∈ E_sk ∧ (x > v)} //build the v adjacency
                vertex set in parallel
11          triCount+ = |u_adj ∩ v_adj|;
12       localeTriSum[here.id] = triCount;
13 return sum(localeTriSum)
```

the adjacency list of u and the adjacency list of v. Then for all nodes in the adj_u if the same node exists in the adj_v then a triangle is added to the locale and then finally summed together with the other locales.

There are ways to improve upon this method. Referencing Fig. 2.4 and Fig. 2.5, we can see that a small number of nodes in the power law graph have a very high degree and most nodes have a much smaller degree. We can exploit this using a minimized search kernel. By choosing the smallest edge list of the 3 candidates, we can dramatically reduce the number of operations that are performed while counting triangles.

Algorithm 3: Minimized search kernel triangle counting

1 **forall** *(edge* $e_1 = \{u, v\} \in E$*) && (*e_1 *is local)* **do**
2 Let L_{uv} be the lower degree vertex and H_{uv} higher degree vertex in $\{u, v\}$
3 Let $Adj_{L_{uv}}$ be the adjacent vertices set of L_{uv}
4 **forall** *(*$x \in Adj_{L_{uv}}$*)* **do**
5   Let $e_2 = \{L_{uv}, x\}$
6   **if** *(*$\exists\ e_3 = \{x, H_{uv}\}$*)* **then**
7   Increment the support of u,v,w, and G by 1
8 **return** *Count of all triangles for all N in G, and triangle count in G*

3.3 Triangle Centrality

The seminal work for Triangle Centrality was done by Paul Burkhardt [11] and details the development in their paper. Triangle Centrality is a metric of how involved a vertex is in the triangles of the graph. In an undirected graph G represented as $G = \{V, E\}$, with respective vertex and edge sets, the Triangle Centrality of a node v is given by Burkhardt as:

$$TC(v) = \frac{\frac{1}{3}\sum_{u \in N_{\Delta}^{+}} \Delta(u) + \sum_{w \in \{\frac{N(v)}{N_{\Delta}(v)}\}} \Delta(w)}{\Delta G}$$

This metric falls within the closed range of [0,1] which is achieved by damping the counting of triangles by one third to avoid over counting and then normalizing that by all the triangles in the graph G. Burkhardt also introduces this method in linear algebra form but that particularly paradigm will not be available to us in Chapel as of writing this. Additionally Burkhardt proposes several theorems that prove that the triangle counting can be done in $O(m\sqrt{m})$ time when done on an adjacency matrix.

In contrast to other definitions of centrality, Burkhardt proposes a time complexity of $O(m\sqrt{m})$ which depending on the graph topology, may yield better performance. Critically in triangle centrality, the value of a node is not given just by the number of triangles in its adjacency but also by the triangles of its neighbors. This allows this centrality metric to rank highly nodes that are connected to highly cohesive nodes i.e. like the agents of movie stars.

According to Burkhardt, the advantages of triangle Centrality are 4 fold.

1. No need to make iterative updates and guaranteed convergence

2. Grants weight to nodes that are directly in a lot of triangles themselves if their neighbors are highly connected

3. Is robust for each node because it takes consideration from not only the neighbors, but also indirectly the neighbors of neighbors through triangle counting

4. Efficient run time

Li and Bader [27] describes a succinct and novel implementation of Triangle Centrality in GraphBLAS [18][17]. GraphBLAS is an API that represents graphs as sparse matrices and allows developers to build graph algorithms using linear algebra. This method was built in the SuiteSparse implementation of GraphBLAS and provides the basis for our comparisons. Because this method is done GraphBLAS

with precise matrix methods, the algorithm should approach the optimal $O(m\sqrt{m})$ time complexity that Burkhardt describes.

CHAPTER 4

EXPERIMENTS AND RESULTS

4.1 Algorithms

Much of the work for Triangle Centrality lies in counting triangles of which there are three requirements for this method: triangle counts for the set of triangles belonging to the neighborhood of each node which include said node, the number of triangles in the adjacency of a node but do not include the node, and the total number of triangles in the graph. We compare the results of 6 different implementations in Arkouda using the Chapel back end for parallel computing. At a high level, each algorithm will have similar data structures for the triangle centrality calcuations.

Algorithm 4: High Level Triangle Centrality

1 Let $NeiTrNum$ be an array of length N
2 Let $NeiNonTriNum$ be an array of length N
3 Let $TriangleCentrality$ be an array of length N
4 $TriNum$, $NeiAry$, $TriCount$ = CountTriangles(G)
5 co**forall** *edges in locale* **do**
6   **if** $NeiAry[edge]$ *is True* **then**
7   Update $NeiTriNum$ of u with v
8   Update $NeiTriNum$ of v with u
9   **else**
10   Update $NonNeiTriNum$ of u with v
11   Update $NonNeiTriNum$ of v with u
12 co**forall** $u \in G$ **do**
13   **forall** $vinN(u)$ **do**
14   Update Curnum with the neighbor's triangles
15     $TriangleCentrality[u] =$
       $(NeiNonTriNum[u] + ((NeiTriNum[u] + TriNum[u])) * 1/3)/\Delta G$
16 **return** *Triangle Centrality of all vertices in G*

In Alg. 4, the data structures utilized are common to all algorithms because these are used for the calculations in Triangle Centrality. The variable *triCount* is just a counter for the total count of triangles in the graph, the $NeiAry$ is a binary array which indicates if a triangle exists for an edge in G. The arrays $NeiTriNum$ and $NeiNonTriNum$ are arrays of length E that maintain the number of triangles for the open and closed set of triangles for each node v respectively. The array $TriNum$ maintains the triangle count for each vertex in the G. Because all of these methods rely on the double index data structure, all of the aforementioned data structures are used in the methods presented.

4.1.1 Naive Method

To illustrate the performance of each algorithm, we start with a naive method. This method, like all of the others presented here utilize the same double index data structure and all of the parallel abstractions inherent to Chapel. The only difference is the actual triangle counting method. In the naive method, we simply iterate through the edge list which contains edges (u, v). Then, we iterate through the adjacency list of each of those vertices if we find a third edge that forms a triangle then we make updates to three data structures. These data structures are common to all algorithms because these are used for the calculations in Triangle Centrality. The variable $triCount$ is just a counter for the total count of triangles in the graph, the $NeiTriNum$ is a binary array which indicates if a triangle exists for an edge in G. The array $TriNum$ maintains the triangle count for each vertex in the G. Because all of these methods rely on the double index data structure, all of the aforementioned data structures are used in the methods presented.

Algorithm 5: Naive Triangle Centrality

```
1  Let NeiAry be a vector of length E
2  triCount = 0
3  Let TriNum be a vector of length N
4  coforall edges e_1 = (u, v) ∈ E && (e_1 is local) do
5  |   forall edge e_2 (u, w) ∈ N(u) do
6  |   |   if edge e_3 (w, v) ∈ E then
7  |   |   |   Increment TriCount
8  |   |   |   Update NeiAry
9  |   |   |   Update TriNum
10 |   forall edges e_2 (v, w) ∈ N(v) do
11 |   |   if edge e_3 (w, u)inE then
12 |   |   |   Increment TriCount
13 |   |   |   Update NeiAry
14 |   |   |   Update TriNum
```

4.1.2 Minimized Triangle Search

Notice that in Alg. 5 the search is done exhaustively searching both adjacency lists of u and v to find triangles. If real world graphs had even degree distributions meaning that every node had the same sized adjacency list then searching both lists would be an acceptable assumption but this is not the case. Due to the distribution inherent to a power law graph, some nodes will have a much higher adjacency list than others. When searching for triangles within the graph, it may be advantageous to exploit this distribution and search the smaller adjacency lists. This is the motivation behind the Minimized Triangle Search.

Algorithm 6: Minimized Triangle Search

```
1  Let NeiAry be a vector of length E
2  triCount = 0
3  Let TriNum be a vector of length N
4  coforall (edge e_1 = (u,v) ∈ E) && (e_1 is local) do
5      Let uadj and vadj be sets
6      Let Adj_{L_uv} be the adjacent vertices set of L_uv
7      small_adj = node with min(N(u), N(v))
8      large_adj = node with max(N(u), N(v))
9      forall (w ∈ N(min_adj))) do
10         if (adj(N(w)) < adj(e)) then
11             e = findEdge(adj(e_big, w)
12         else
13             e = findEdge(adj(w, e_big)
14         if e ≠ −1 then
15             Increment TriCount
16             Update NeiAry for all edges
17             Update TriNum for all vertices
18 return TriCount, NeiAry, TriNum
```

In Alg. 6 iterations through both adjacency lists are done through the shortest possible lists. In a highly skewed graph, this may reduce a large number of computations.

4.1.3 Edge List Intersection

This is the traditional method for counting triangles and it is a good comparison for our other methods. Like 5 the method iterates through both adjacency lists to find potential pairings. The difference comes when finding if a triangle exists due to a

connecting third edge; in the naive method there is an explicit search for that third edge. In the edge list intersection, potential connections are held in a set and then compared later.

Algorithm 7: Edge List Intersection

```
1  Let NeiAry be a vector of length E
2  triCount = 0
3  Let TriNum be a vector of length N
4  coforall (edge e_1 = (u, v) ∈ E) && (e_1 is local) do
5      Let uadj and vadj be sets
6      forall (w ∈ N(u)) do
7          if (∃ e = (u, w)) then
8              Add w into uadj
9      forall (w ∈ N(v)) do
10         if (∃ e = (v, w) then
11             Add w into vadj
12     forall w ∈ uadj do
13         if w ∈ vadj then
14             Increment triCount
15             Update NeiAry
16             Update TriNum
17 return TriCount, NeiAry, TriNum
```

This is handled through Chapel's parallel abstraction for a set. The structures *uadj* and *vadj* contain the nodes that are adjacent to u and v respectfully and these structures are then compared. If there exists a w in both *uadj* and *vadj* then this indicates that there are edges (u, w) and (v, w) in G meaning there is a triangle.

4.1.4 Path Merge Method

This method exploits the sorting done in Arkouda's graph processing. Edges are sorted by value and this avoids the need to search for edges within a graph. Instead, the comparisons of the two adjacency lists is done side by side and stops when the counter on one side is greater than the length of its adjacency list. This does not inherently exploit the power law nature of the graphs however; even if the adjacency list of one is short, if the final node is a comparatively high value, it will be compared all the other nodes in the other adjacency list. This method does avoid the second for loop needed to compare v's adjacency list however.

Algorithm 8: Path Merge Method

```
1  Let NeiAry be a vector of length E
2  Let TriCount = 0
3  Let TriNum be a vector of length N
4  coforall (edge e_1 = (u, v) ∈ E) && (e_1 is local) do
5      set LNI and RNI to the start of the left, right edge adjacency list
6      Iterate through the adjacency list
7      while (LNI < N(u) and RNI < N(v)) do
8          if dst[LNI] == v then
9              Increment LNI
10             Continue
11         if dst[RNI] == u then
12             Increment RNI
13             Continue
14         if dst[LNI] == dst[RNI] then
15             Increment TriCount
16             Update NeiAry
17             Update TriNum
18             Increment LNI and RNI
19         else
20             if dst[LNI] > dst[RNI] then
21                 Increment RNI
22             else
23                 Increment LNI
24 return TriCount, NeiAry, TriNum
```

4.1.5 Small Set Intersection

Paradigms from Alg. 7 and Alg. 6 can be utilized to create a new method. First this method iterates through the smaller adjacency list and then utilizes binary search to find a matching edge if it exists.

Algorithm 9: Small Set Intersection Method

```
1  Let NeiAry be a vector of length E
2  Let triCount = 0
3  Let TriNum be a vector of length N
4  coforall (edge e_1 = (u,v) ∈ E) && (e_1 is local) do
5      Let small_adj be the node of min(N(u), N(v))
6      Let large_adj be the node of max(N(u), N(v))
7      Let s_set be an empty set
8      forall edge e_2 = {small_adj, x} ∈ N(small_adj) do
9          if x ≠ large_adj then
10             Add x into s_set
11     forall w ∈ s_set do
12         if findEdge(w, large_adj) then
13             Update NeiAry for all edges
14             Increment TriNum for all vertices
15             Increment TriCount
16 return TriCount, NeiAry, TriNum
```

4.2 Experimental Setup

The experiments are run on an Ubuntu 20.04.3 LTS desktop with an Intel i7-10700K on a single core with 16GB of RAM. Based on the capabilities of Arkouda and Chapel, multilocale functionality is possible but slower because of the inter-locale communications required especially on these smaller datasets.

The comparisons to the other implementation done by Li *et al* was done on the same machine. The implications of this comparison will be discussed in the results section. After running all of the experiments the results will be considered in two ways. The first is raw result time: each of the methods implemented in Arkouda are run on the same machine and so direct comparisons are applicable. Because the other method is not a distributed system implementation it will not have some of the overhead Arkouda has but running on a single locale will provide some equalization. Each of these algorithms will be run multiple times and the average over the runs will be taken.

In addition to a raw run time comparison, the edge processing speed will be compared as a function of some of the graph properties mentioned. Algorithms that perform well on certain types of graphs may not perform well on others, and showing how each performs based on a certain property may help in future run time decisions. The edge processing speed will be plotted as a function of properties including the assortativity, the power law exponent, 90th percentile diameter, the average cluster coefficient, and the fraction of closed triangles.

4.3 Results and Discussion

Results will be in interpreted as described previously, by the actual run time and an examination of the edge processing speed based on different graph properties.

On initial view of Table 4.1 there is not a clear "best" algorithm. Both Alg. 6 and Alg. 8 have runs where one outperformed the other. On occasion, these algorithms even ran faster than the shared memory method which seems to further exemplify the speed of Chapel and by extension, Arkouda. On the aggregate, the shared memory implementation the Arkouda methods which would be expected because Arkouda is expected to handle distributed machines and communications overhead. To see how each method performs relative to types of graphs, the processing

Table 4.1 Graph Processing Times (sec)

Graph	*GraphBLAS*	*Naive*	*Minimum Search*	*Path Merge*	*Small Search*	*List Intersection*
as-caida20071105	1.30e-4	7.69e-1	1.11e-1	8.45e-2	1.84e-1	1.57e+0
ca-AstroPh	3.96e-1	2.76e-1	1.76e-1	2.49e-1	3.69e-1	7.02e-1
ca-CondMat	1.08e-1	1.31e-1	8.87e-2	4.74e-2	1.56e-1	2.77e-1
ca-GrQc	1.66e-2	2.03e-2	1.46e-2	7.93e-3	2.51e-2	4.24e-2
ca-HepPh	2.83e-1	2.28e-1	2.01e-1	3.94e-1	4.55e-1	7.45e-1
ca-HepTh	3.16e-2	3.65e-2	2.61e-2	1.18e-2	4.33e-2	7.35e-2
email-Enron	1.20e-4	3.80e-1	1.71e-1	4.10e-1	3.35e-1	1.07e+0
facebook_combined	1.20e-1	1.87e-1	1.34e-1	2.32e-1	2.90e-1	5.02e-1
loc-brightkite_edges	5.73e-3	3.43e-1	2.04e-1	2.70e-1	3.65e-1	7.80e-1
Oregon1_010331	2.48e-1	4.75e-1	2.40e-2	7.45e-2	3.98e-2	6.80e-1
Oregon1_010407	2.34e-1	4.72e-1	2.39e-2	7.44e-2	3.93e-2	6.79e-1
Oregon1_010414	3.12e-1	4.55e-1	2.41e-2	7.99e-2	4.01e-2	6.73e-1
Oregon1_010421	2.52e-1	4.15e-1	2.44e-2	7.82e-2	4.06e-2	6.80e-1
Oregon1_010428	2.05e-4	4.76e-1	2.43e-2	7.88e-2	4.00e-2	6.99e-1
p2p-Gnutella04	2.05e-4	5.42e-2	3.98e-2	1.61e-2	7.02e-2	1.25e-1
p2p-Gnutella05	3.09e-4	4.26e-2	3.12e-2	1.32e-2	5.47e-2	9.88e-2
p2p-Gnutella06	3.13e-4	4.17e-2	3.07e-2	1.34e-2	5.40e-2	9.76e-2
p2p-Gnutella08	1.11e-3	2.84e-2	2.06e-2	1.07e-2	3.57e-2	6.54e-2
p2p-Gnutella09	1.91e-4	3.49e-2	2.59e-2	1.23e-2	4.49e-2	8.14e-2
p2p-Gnutella24	2.49e-4	9.03e-2	6.66e-2	2.96e-2	1.11e-1	2.03e-1
p2p-Gnutella25	1.24e-4	7.50e-2	5.53e-2	2.41e-2	9.14e-2	1.68e-1
p2p-Gnutella30	1.52e-4	1.23e-1	9.06e-2	4.06e-2	1.49e-1	2.75e-1
p2p-Gnutella31	1.14e-4	2.05e-1	1.53e-1	6.76e-2	2.49e-1	4.60e-1
delaunay_n10	1.93e-4	4.45e-3	3.28e-3	1.07e-3	5.35e-3	8.54e-3
delaunay_n11	1.50e-4	8.87e-3	6.53e-3	2.15e-3	1.06e-2	1.69e-2
delaunay_n12	1.18e-4	1.77e-2	1.30e-2	4.16e-3	2.13e-2	3.38e-2
delaunay_n13	2.82e-4	3.55e-2	2.60e-2	8.51e-3	4.23e-2	6.78e-2
delaunay_n14	2.45e-4	7.09e-2	5.18e-2	1.63e-2	8.46e-2	1.36e-1
delaunay_n15	4.03e-4	1.42e-1	1.07e-1	3.63e-2	1.72e-1	2.75e-1
delaunay_n16	2.93e-4	2.83e-1	2.08e-1	6.62e-2	3.38e-1	5.46e-1
delaunay_n17	3.74e-4	5.68e-1	4.16e-1	1.33e-1	6.78e-1	1.15e+0

speed of each method will be compared to different graph properties. Not all of the properties were insightful but the figures are included in the appendix for reference.

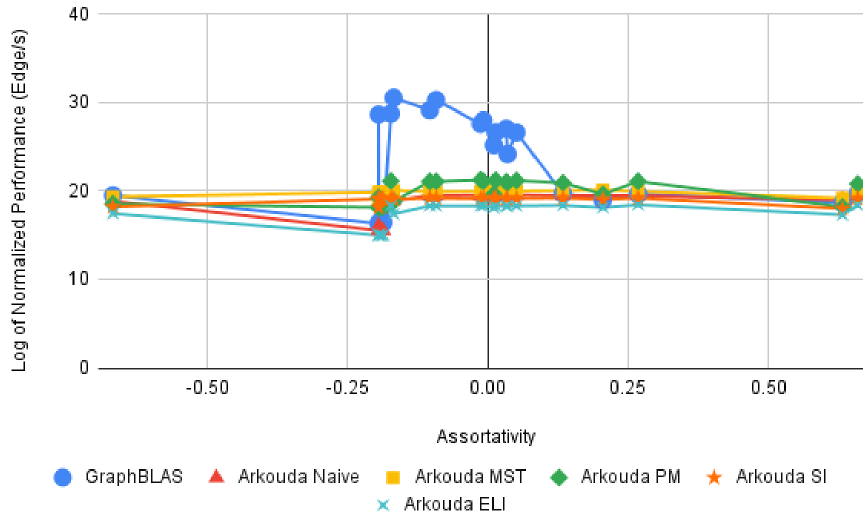

Figure 4.1 Log of edge processing speed vs. assortativity.

In Fig. 4.1 the performance for GraphBLAS clusters around low assortativity and is higher elsewhere. Because GraphBLAS had very high performance on certain graphs, the edge processing speeds are scaled by a logarithm of base 2. Additionally, when reading this, it is important to remember that depending on the property the slope of the line is not necessarily important. In this result, the GraphBLAS implementation performs much better on graphs with low assortativity meaning that it performs well on graphs with moderate levels of mixing between high and low degree vertices.

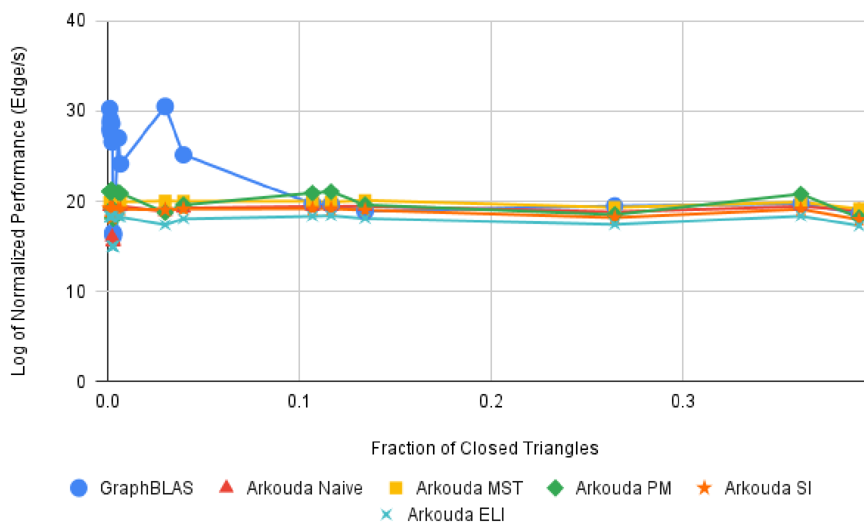

Figure 4.2 Log of edge processing speed vs. fraction of closed triangles.

The same interpretation is valid here in Fig. 4.2 where the processing speed of GraphBLAS clusters around a low fraction of closed triangles. This might imply that GraphBLAS is better suited to sparser graphs than Arkouda. The Arkouda implementations have to search for triangles in all implementations as opposed to the GraphBLAS implementation which utilizes matrix operations.

For the Arkouda implementations, the results are close enough that they can be compared without the logarithm. Because these methods are implemented in the same environment and framework, these can be compared directly.

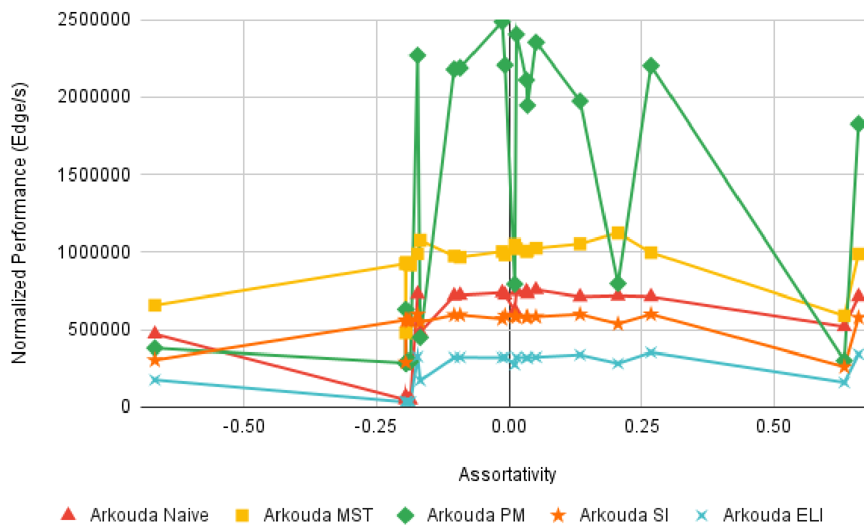

Figure 4.3 Arkouda edge processing speed vs. assortativity.

Like Fig. 4.1, Fig. 4.3 shows a similar result for the Path Merge method, it performs comparatively better at lower levels of assortativity than the other methods. This said, there may be not enough data at high and low levels of assortativity to say for sure.

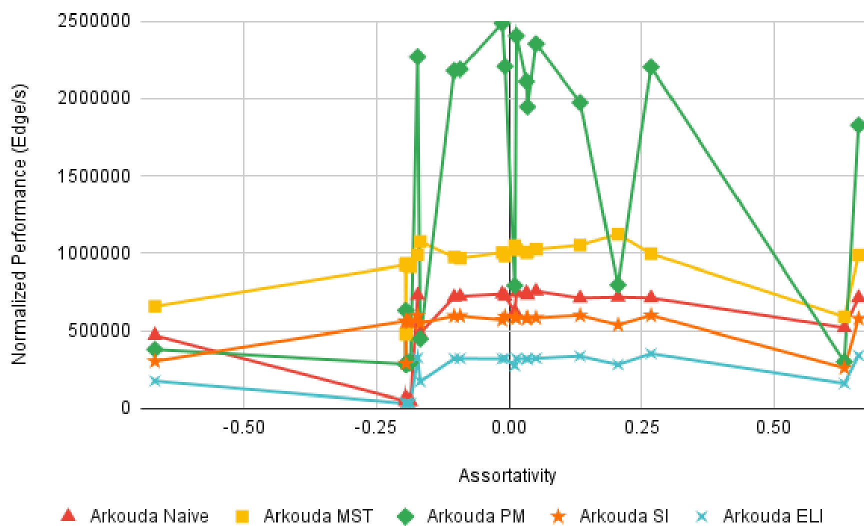

Figure 4.4 Arkouda edge processing speed vs. fraction of closed triangles.

In Fig. 4.4 the Path Merge method performs significantly better in sparser graphs. This may be due to the lack of searches used in the method since all the other Arkouda methods utilize an edge search in some respect.

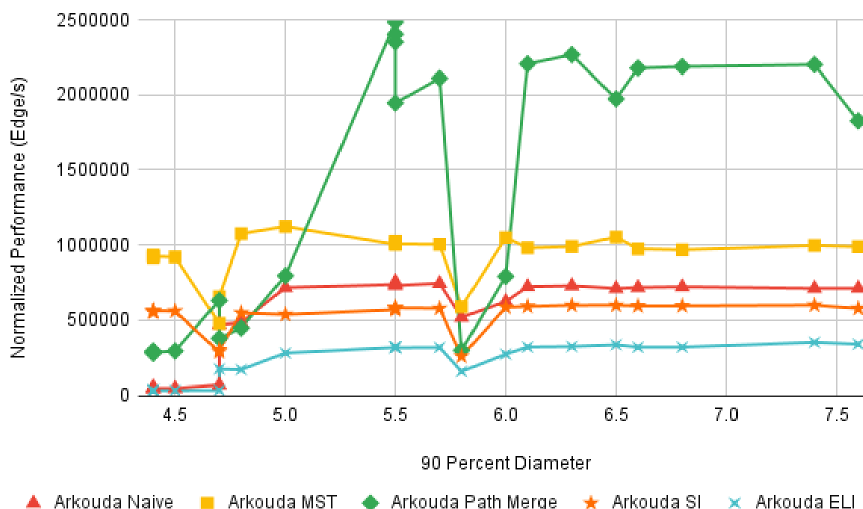

Figure 4.5 Arkouda edge processing speed vs. 90 percent diameter.

The 90 percent diameter is another metric for density; as stated it contains the minimal distance such that 90 percent of nodes in graph are within that value; lower values may imply a denser graph. Path Merge consistently outperforms other methods at higher values but the minimum search method is stronger at lower values which suggests again that path merge works better in sparser graphs.

Based purely on these results though, it seems that Alg. 8 is superior for a larger number of graphs. In certain cases, particular denser graphs, Alg. 6 may be a better option but additional research may be necessary to prove absolutely. It may also be possible that the edge processing speed maybe better modeled through a combination of multiple graph properties.

CHAPTER 5

CONCLUSION

An implementation of several different triangle counting methods have been implemented for a novel centrality metric. Five methods for triangle counting were presented, a naive method, a minimal search method, a path merge method, a small set intersection method, and an edge list intersection method. These methods were compared to each other and another open source implementation of this method on GraphBLAS. Because the latter was done in a shared memory environment, generally it performed better but in denser graphs the Arkouda methods performed nearly equally or better. Of the Arkouda methods alone, the path merge method performed the best most consistently in sparser graphs but the minimized search method performed better in denser graphs.

The edge processing speed for each were then compared to different graph properties including assortativity, average clustering coefficent, the 90 percent diameter, the fraction of closed triangles, and the power law exponent. Based on these metrics, the GraphBLAS implementation and the path merge method performed best on sparser graphs however because the former is a shared memory implementation it is not feasible for massive datasets.

Further research might be done as Arkouda expands their capabilities into arrays with dimensions greater than one dimension. Having multidimensional arrays would allow exploration into methods with adjacency matrices and applying linear algebra methods for counting triangles. Regardless, due to the novelty of Arkouda there are plenty of other centrality metrics that can be implemented in Arkouda.

This work further demonstrates the performance of Arkouda relative to even a local implementation and adds a new tool for practitioners of Arkouda. Our work is

openly available at https://github.com/Bears-R-Us/arkouda-njit and all components necessary to run our methods are open-source.

APPENDIX A

GRAPH PROPERTIES RELATIONSHIP FIGURES

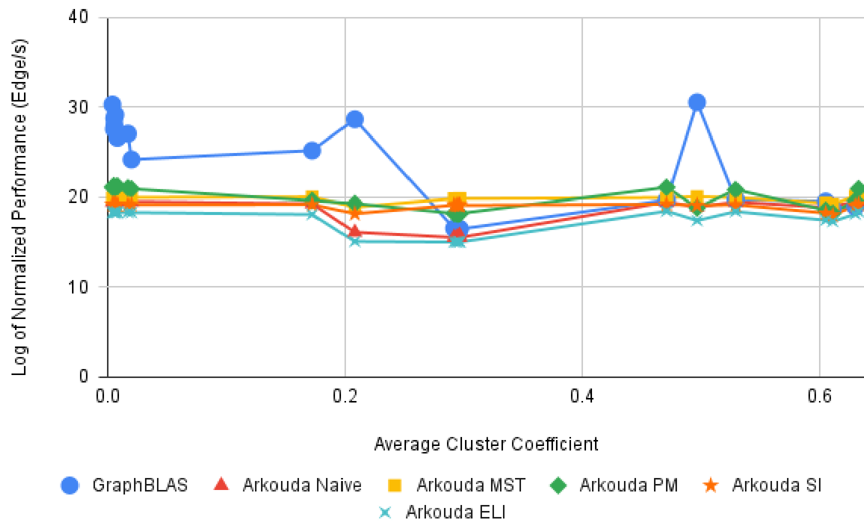

Figure A.1 Log of edge processing speed vs. average cluster coefficient.

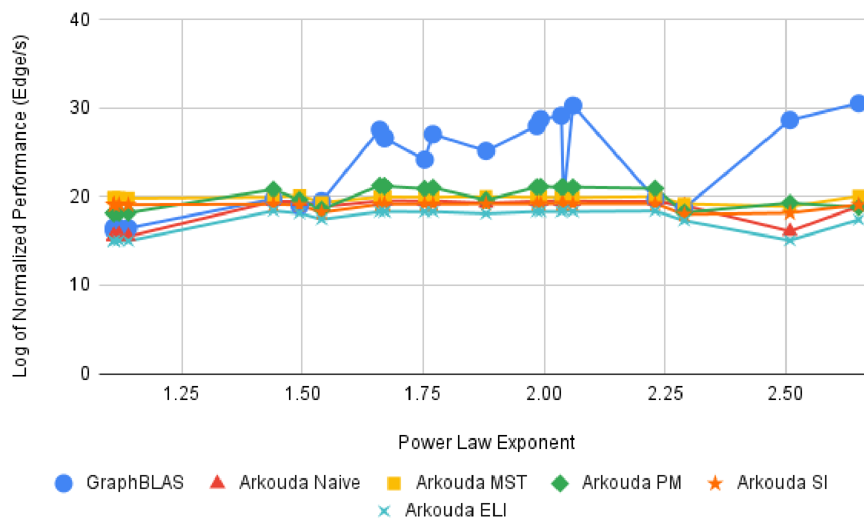

Figure A.2 Log of edge processing speed vs. power law exponent.

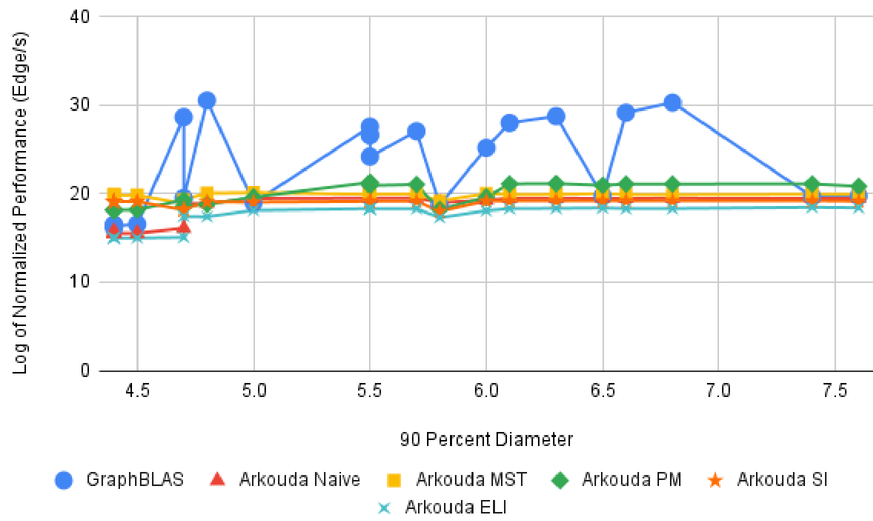

Figure A.3 Log of edge processing speed vs. diameter.

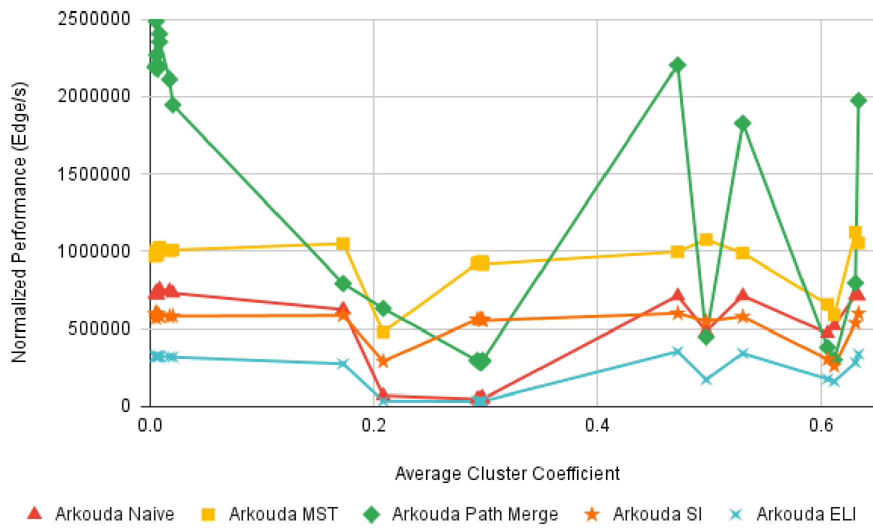

Figure A.4 Arkouda edge processing speed vs. average cluster coefficient.

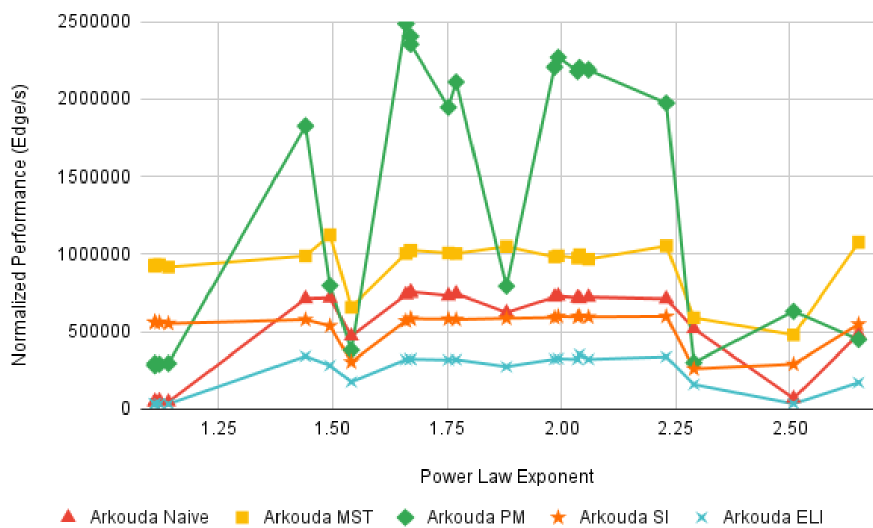

Figure A.5 Arkouda edge processing speed vs. power law exponent.

APPENDIX B

DEGREE DISTRIBUTIONS OF GRAPHS

B.0.1 Real World Graphs

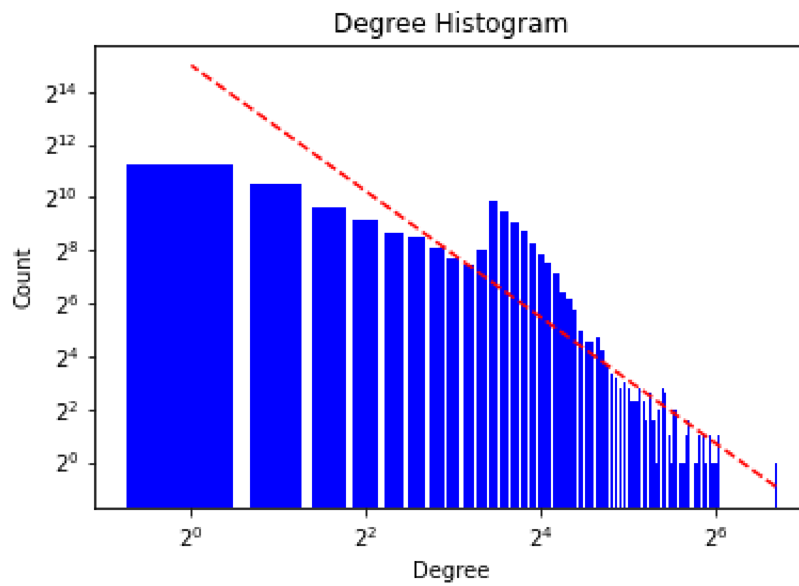

Figure B.1 p2p-gnutella04 degree distribution.

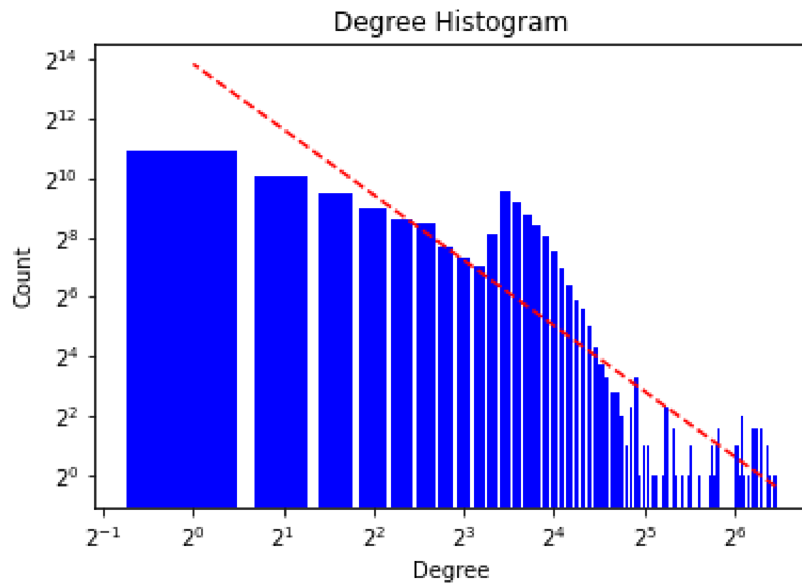

Figure B.2 p2p-gnutella05 degree distribution.

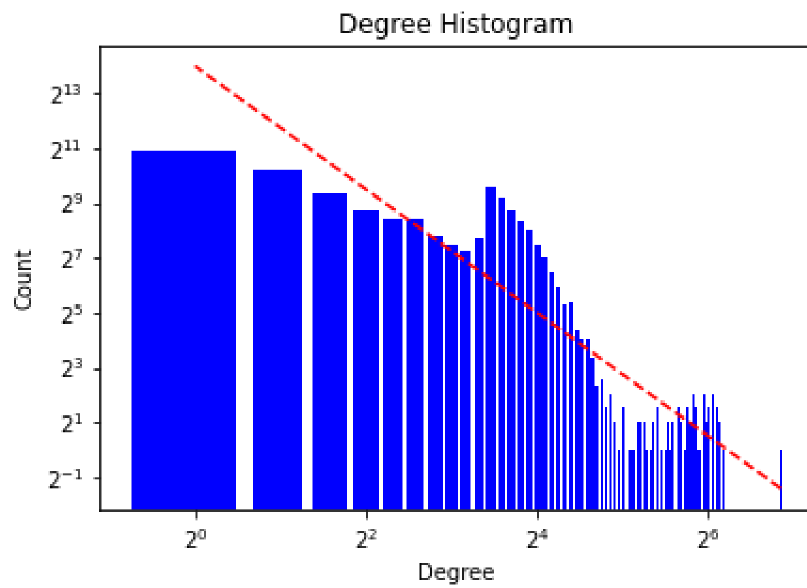

Figure B.3 p2p-gnutella06 degree distribution.

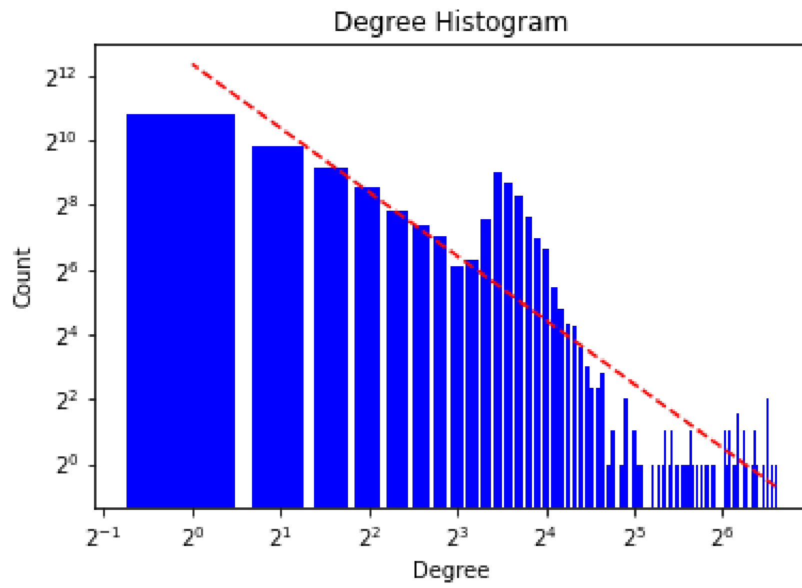

Figure B.4 p2p-gnutella08 degree distribution.

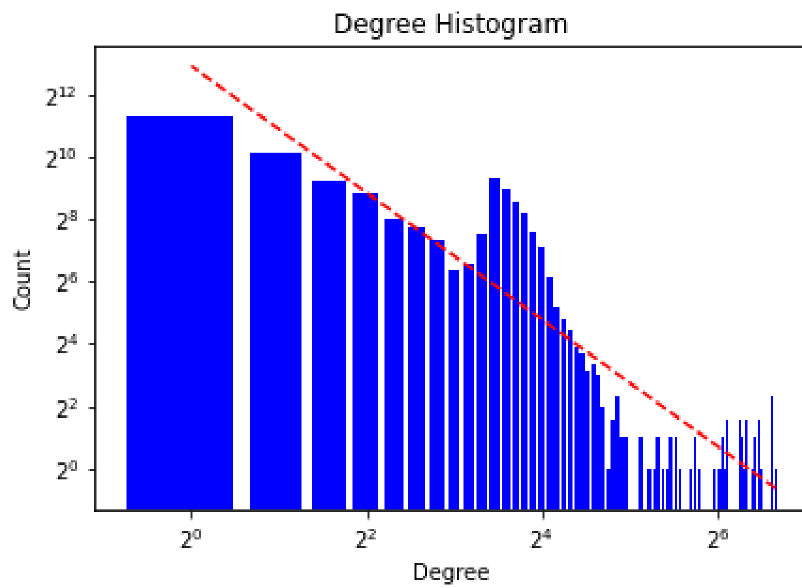

Figure B.5 p2p-gnutella09 degree distribution.

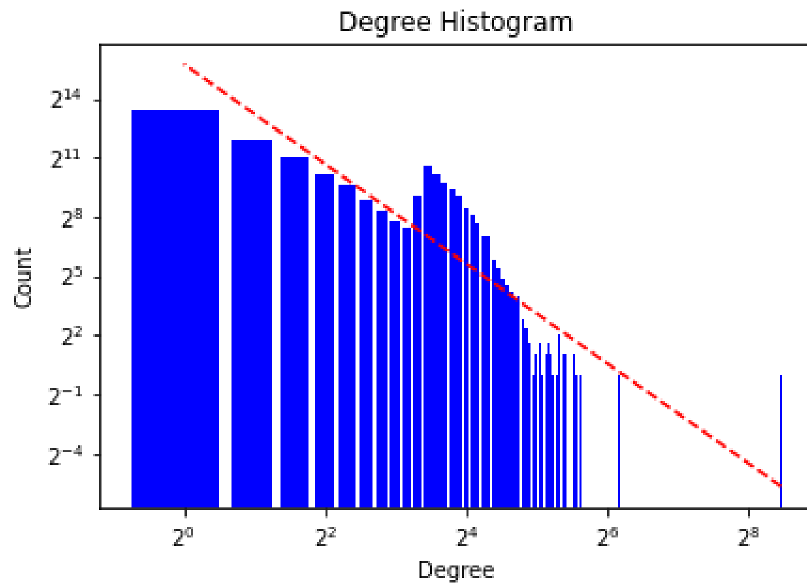

Figure B.6 p2p-gnutella24 degree distribution.

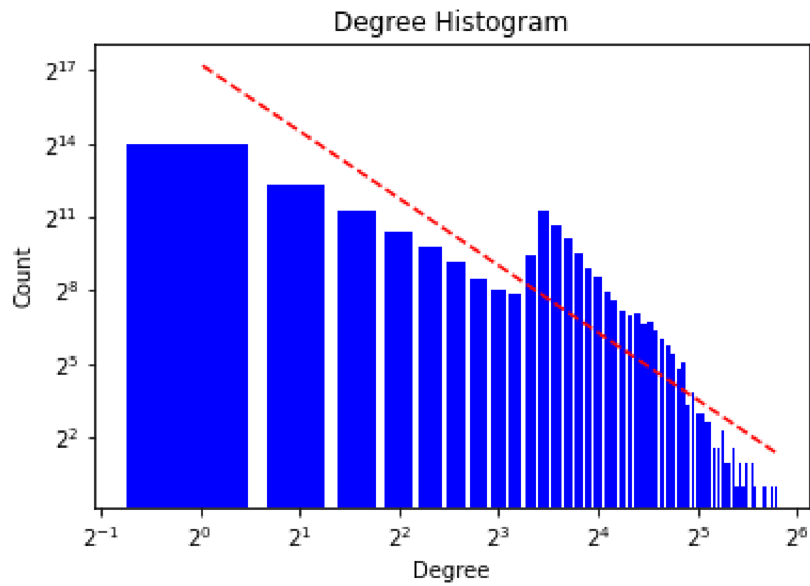

Figure B.7 p2p-gnutella30 degree distribution.

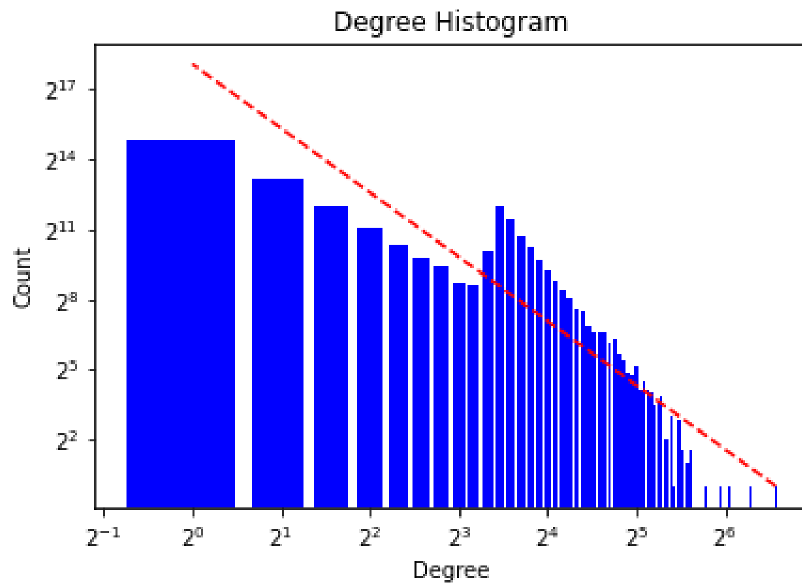

Figure B.8 p2p-gnutella31 degree distribution.

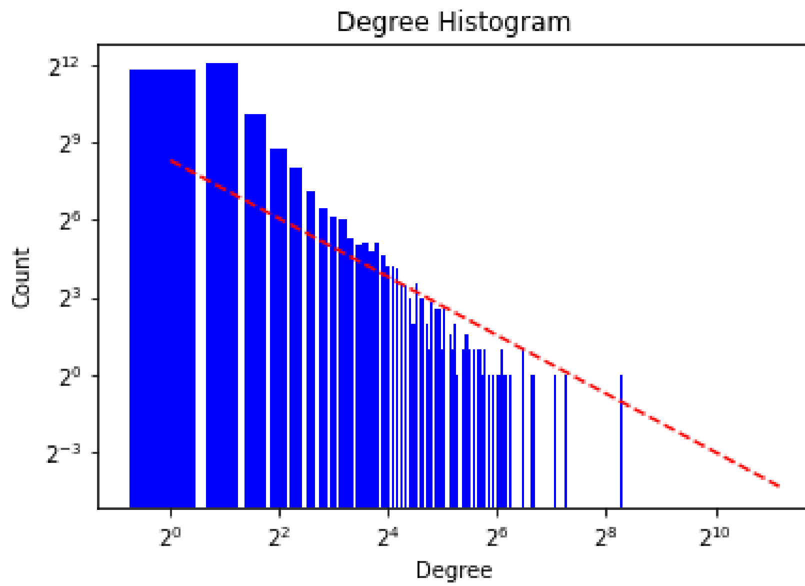

Figure B.9 Oregon1_010331 degree distribution.

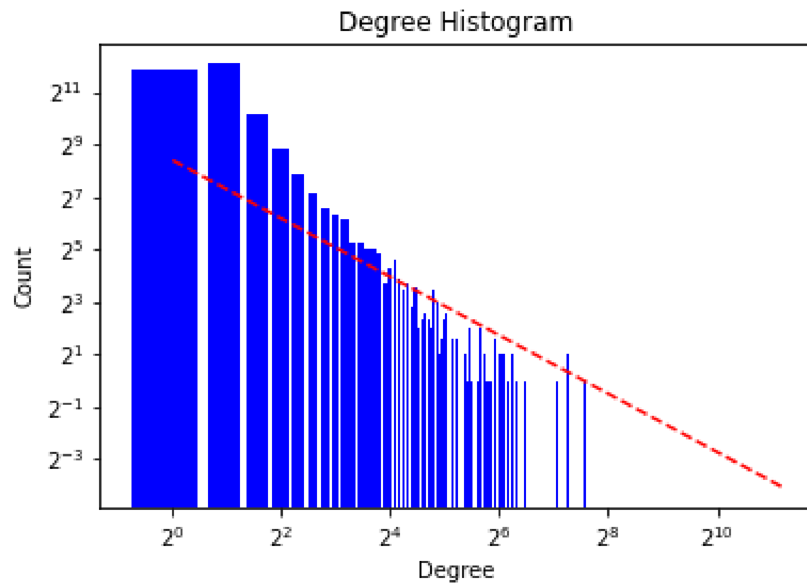

Figure B.10 Oregon1_010407 degree distribution.

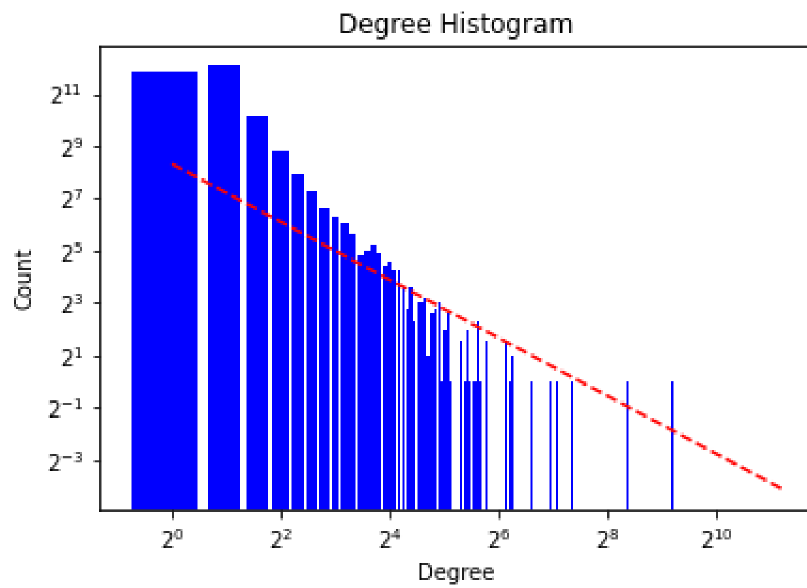

Figure B.11 Oregon1_010414 degree distribution.

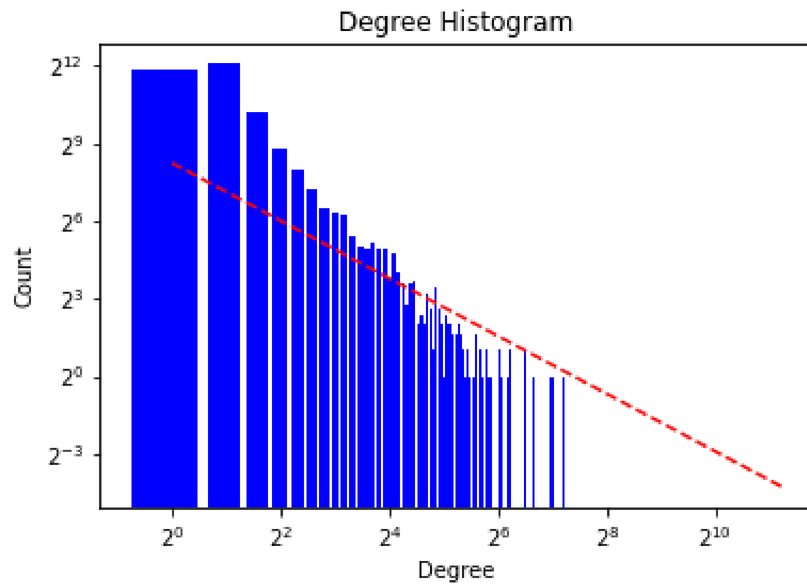

Figure B.12 Oregon1_010421 degree distribution.

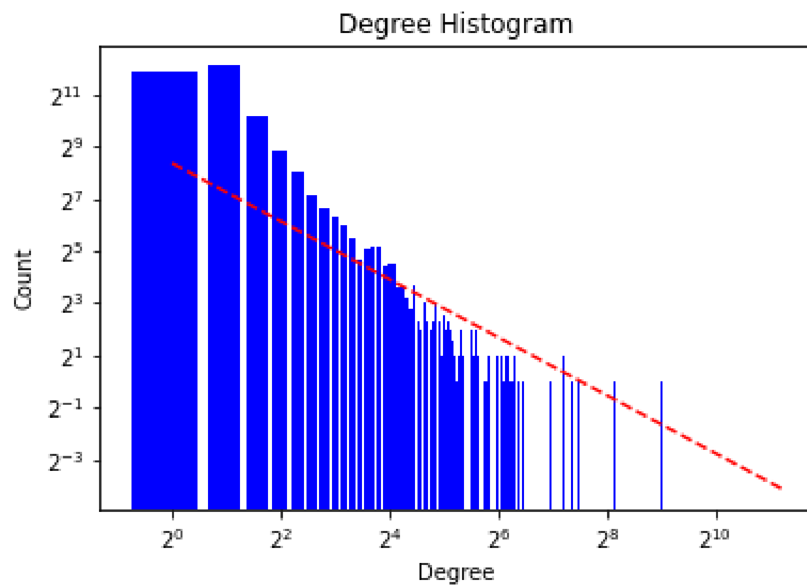

Figure B.13 Oregon1_010428 degree distribution.

B.0.2 Delaunay Graphs

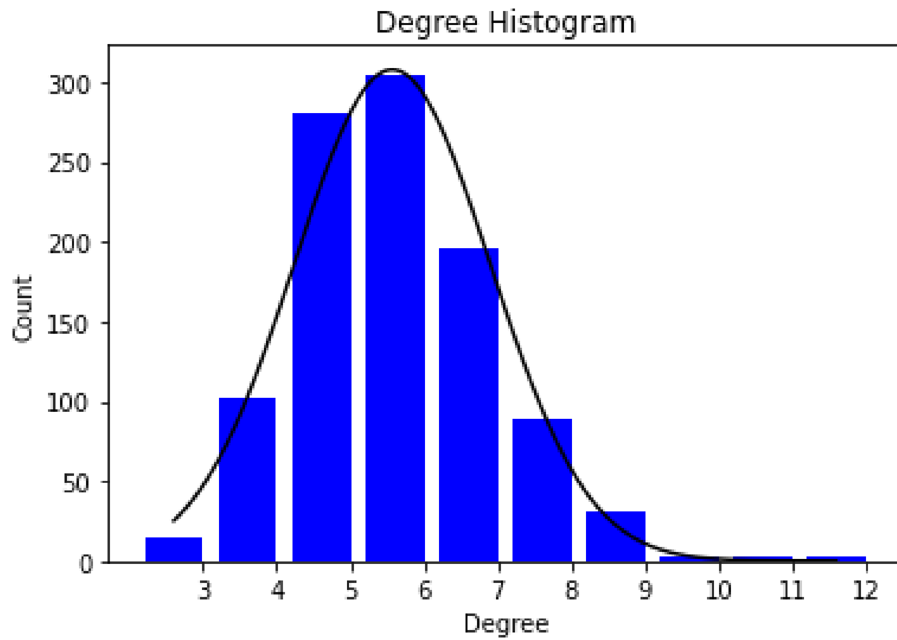

Figure B.14 Delaunay_n10 degree distribution.

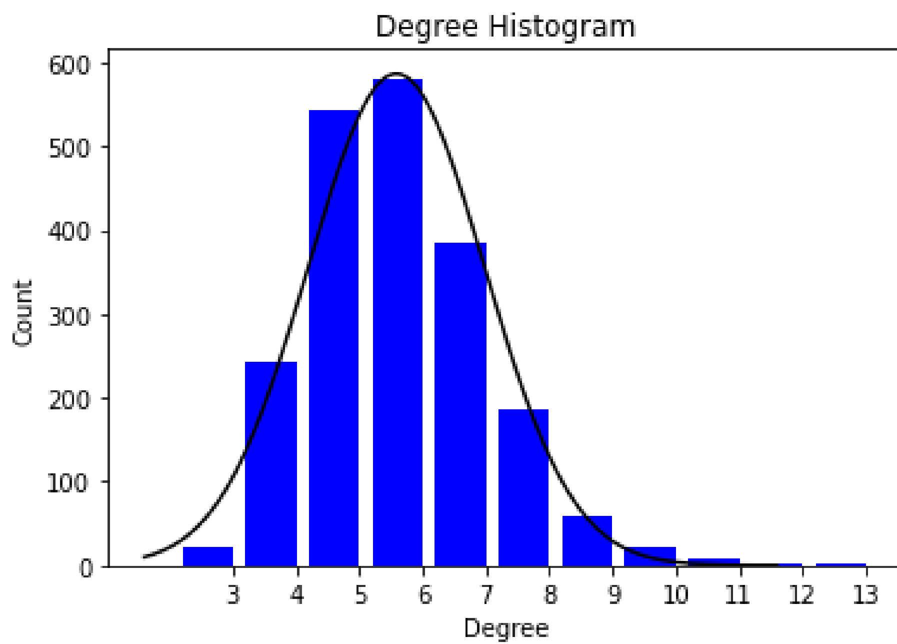

Figure B.15 Delaunay_n11 degree distribution.

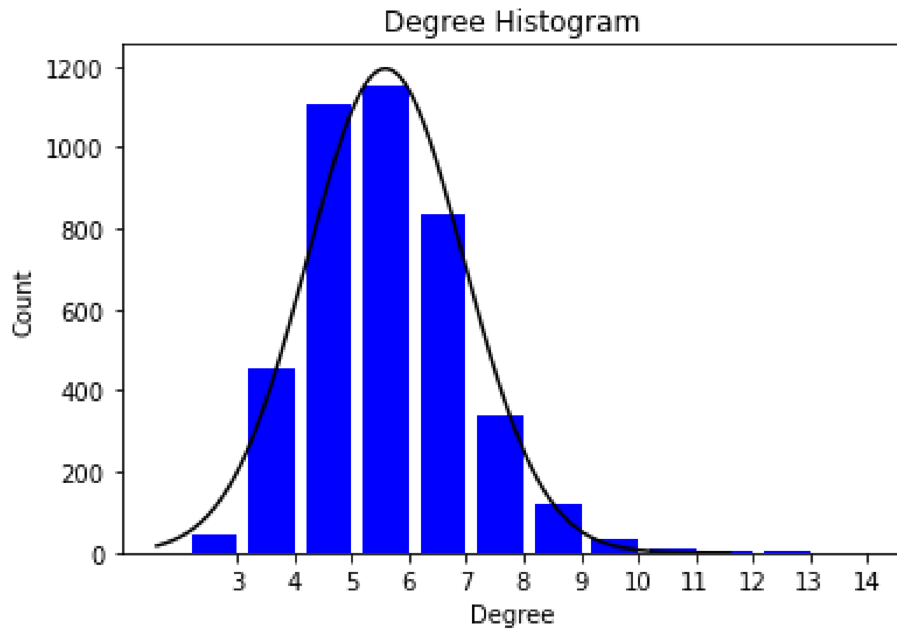

Figure B.16 Delaunay_n12 degree distribution.

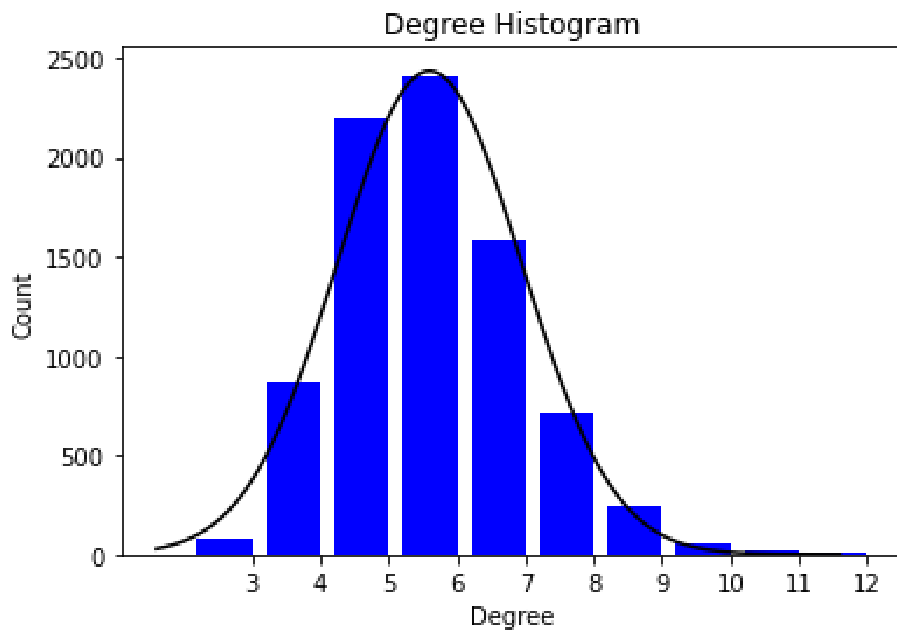

Figure B.17 Delaunay_n13 degree distribution.

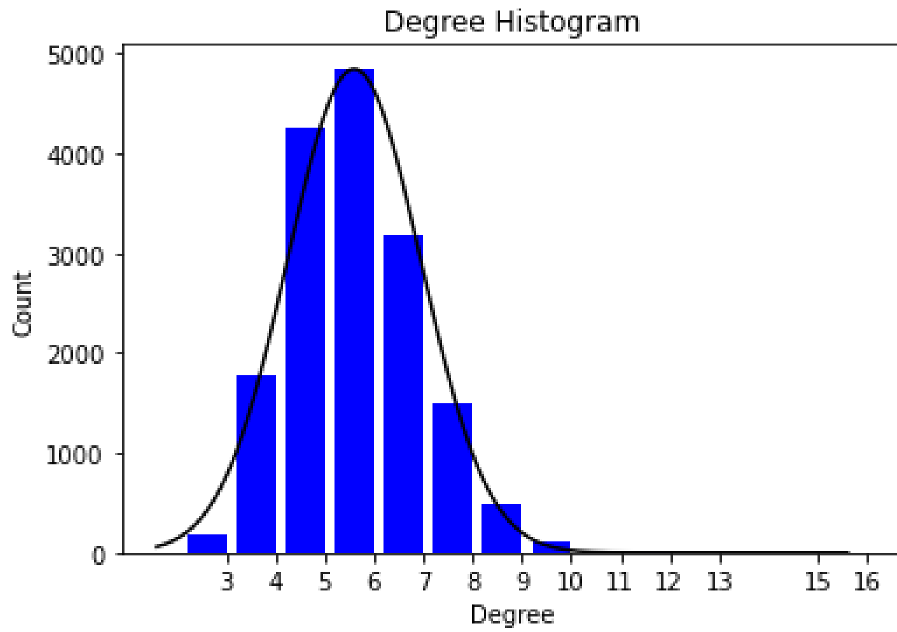

Figure B.18 Delaunay_n14 degree distribution.

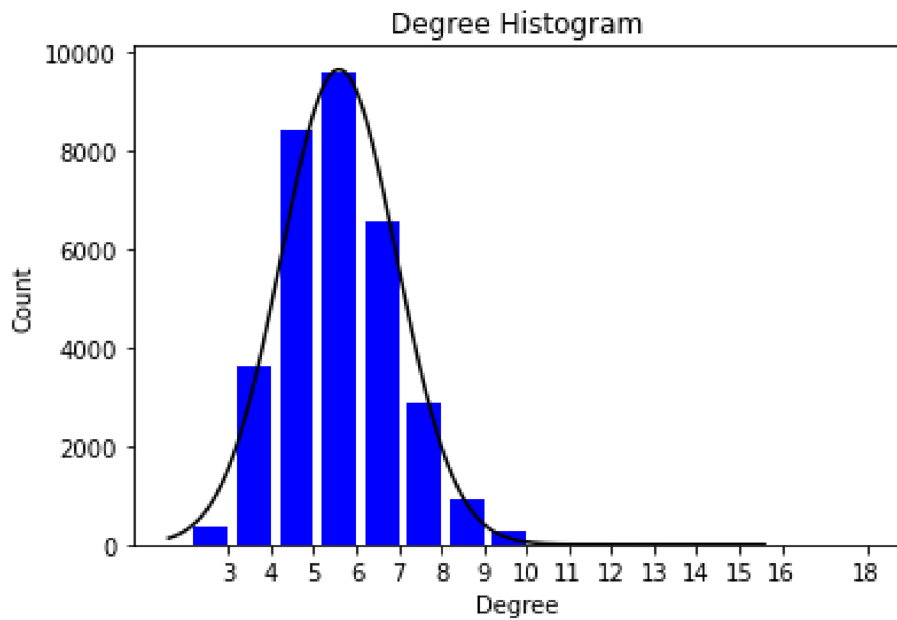

Figure B.19 Delaunay_n15 degree distribution.

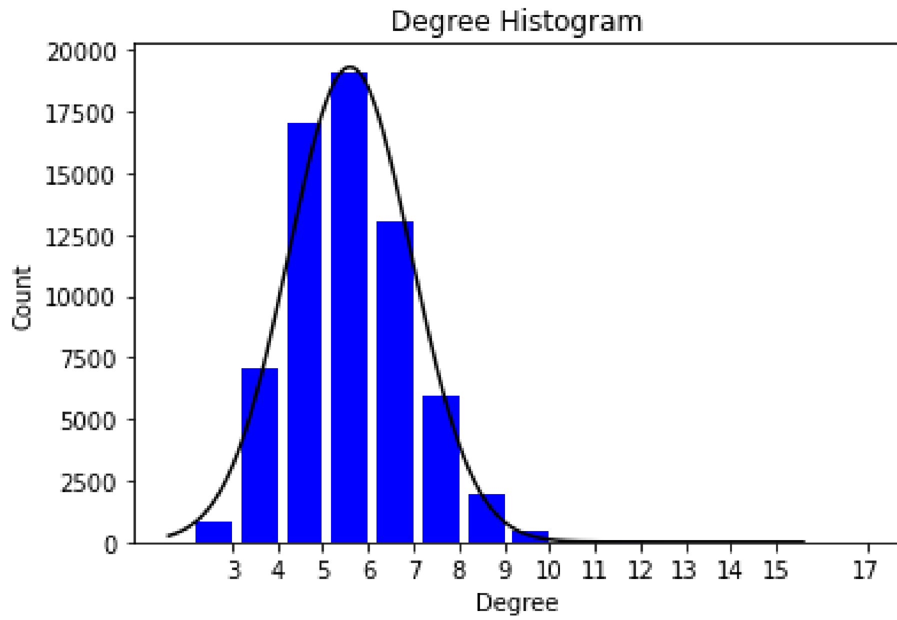

Figure B.20 Delaunay_n16 degree distribution.

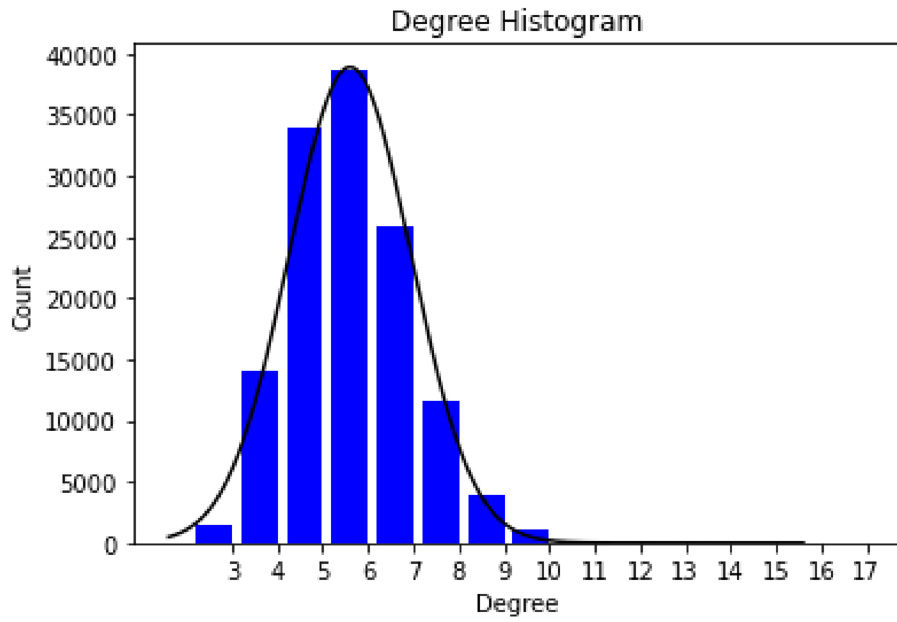

Figure B.21 Delaunay_n17 degree distribution.

BIBLIOGRAPHY

[1] Lada A Adamic, Bernardo A Huberman, AL Barabási, R Albert, H Jeong, and G Bianconi. Power-law distribution of the world wide web. *Science*, 287(5461):2115–2115, 2000.

[2] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the world-wide web. *Nature*, 401(6749):130–131, sep 1999.

[3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] Alex Bavelas. Communication patterns in task-oriented groups. *The journal of the acoustical society of America*, 22(6):725–730, 1950.

[5] Mauro Bisson and Massimiliano Fatica. Static graph challenge on GPU. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2017.

[6] Mauro Bisson and Massimiliano Fatica. Update on static graph challenge on GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2018.

[7] Mark Blanco, Tze Meng Low, and Kyungjoo Kim. Exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[8] Paolo Boldi and Sebastiano Vigna. Axioms for centrality, 2013.

[9] P Bonacich. Factoring and weighing approaches to clique identification. *Journal of Mathematical Sociology*, 92:1170–1182, 1971.

[10] Paul Burkhardt. Graphing trillions of triangles. *Information Visualization*, 16(3):157–166, 2017. PMID: 28690426.

[11] Paul Burkhardt. Triangle centrality, 2021.

[12] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus. Chapel comes of age: Making scalable programming productive. *Cray User Group*, 2018.

[13] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 1082–1090, New York, NY, USA, 2011. Association for Computing Machinery.

[14] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 672–680, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Shumo Chu and James Cheng. Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data*, 6(4), dec 2012.

[16] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report*, 16(3.1), 2008.

[17] Timothy A. Davis. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[18] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), dec 2019.

[19] Zhihui Du, Oliver Alvarado Rodriguez, David A Bader, Michael Merrill, and William Reus. Exploratory large scale graph analytics in Arkouda. 2021.

[20] Zhihui Du, Oliver Alvarado Rodriguez, Joseph Patchett, and David A. Bader. Interactive graph stream analytics in arkouda. *Algorithms*, 14(8), 2021.

[21] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM Computer Communication Review*, 29(4):251–262, 1999.

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

[23] Pieter Hintjens. *ZeroMQ: messaging for many applications.* O'Reilly Media, Inc., 2013.

[24] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, page 177–187, New York, NY, USA, 2005. Association for Computing Machinery.

[25] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. 2006.

[26] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, 2008.

[27] Fuhuan Li and David A. Bader. A GraphBLAS implementation of triangle centrality. In *The 25th Annual IEEE High Performance Extreme Computing Conference (HPEC)*, 2021.

[28] Michael Merrill, William Reus, and Timothy Neumann. Arkouda: interactive data exploration backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, pages 28–28, 2019.

[29] Mark EJ Newman. Assortative mixing in networks. *Physical review letters*, 89(20):208701, 2002.

[30] Juhani Nieminen. On the centrality in a graph. *Scandinavian journal of psychology*, 15(1):332–336, 1974.

[31] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[32] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. H-index: Hash-indexing for parallel triangle counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[33] Roger Pearce, Maya Gokhale, and Nancy M Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 825–836. IEEE, 2013.

[34] Akrati Saxena, Ralucca Gera, and S. R. S. Iyengar. A faster method to estimate closeness centrality ranking, 2017.

[35] Andrew T Stephen and Olivier Toubia. Explaining the power-law degree distribution in a social commerce network. *Social Networks*, 31(4):262–270, 2009.

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

[37] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V Çatalyürek. Fast triangle counting using Cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.