University of South Florida
Scholar Commons

Graduate Theses and Dissertations | Graduate School

June 2018

Borromean: Preserving Binary Node Attribute Distributions in Large Graph Generations

Clayton A. Gandy
University of South Florida, cagandy3@gmail.com

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation

Gandy, Clayton A., "Borromean: Preserving Binary Node Attribute Distributions in Large Graph Generations" (2018). *Graduate Theses and Dissertations.*
https://scholarcommons.usf.edu/etd/7293

This Thesis is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.

Borromean: Preserving Binary Node Attribute Distributions in Large Graph Generations

by

Clayton A. Gandy

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Adriana Iamnitchi, Ph.D.
Yicheng Tu, Ph.D.
John Skvoretz, Ph.D.

Date of Approval:
June 11, 2018

Keywords: graph generators, social networks, dk graph models, binary node attribute networks

Copyright © 2018, Clayton A. Gandy

DEDICATION

This work is dedicated to those whose perseverance helped bring it to fruition and to those I lost along the journey. My parents were invaluable for their support and steadfastness over the years. Thank you to my advisor Adriana Iamnitchi for teaching me how to be a storyteller.

In memoriam Roger Timothy Gandy (1968-2002). It is not the great American novel, but it is a start.

ACKNOWLEDGMENTS

This research was partially supported by the U.S. National Science Foundation under grant IIS 1546453.

TABLE OF CONTENTS

LIST OF TABLES iii

LIST OF FIGURES v

ABSTRACT vi

CHAPTER 1: INTRODUCTION 1
- 1.1 Motivation 1
- 1.2 DK-2 Graphs 2
- 1.3 Objectives and Contributions 5

CHAPTER 2: RELATED WORK 7
- 2.1 Synthetic Graph Generation 7
  - 2.1.1 Kronecker Generation 8
  - 2.1.2 Block Two-Level Erdos-Renyi 9
  - 2.1.3 ERGMs 9
  - 2.1.4 Related DK Graph Generators 10
    - 2.1.4.1 DK 2.5 10
    - 2.1.4.2 2K Simple 10
- 2.2 Parallel Graph Processing Frameworks 11
  - 2.2.1 Giraph 11
  - 2.2.2 Darwini 12
- 2.3 Graph Generators for Graph Anonymity 13
  - 2.3.1 Differential Privacy and DK-2 Graphs 15
  - 2.3.2 Pygmalion: Differentially Private DK-2 Graph Anonymization 16

CHAPTER 3: DATASETS 18

CHAPTER 4: ORBIS DK-2 TOPOLOGY GENERATOR 20
- 4.1 dK Rewiring 21
- 4.2 Limitations of the Orbis DK-2 Pseudo-Graph Generation Model 21

CHAPTER 5: BORROMEAN-SEQUENTIAL LABELED GRAPH GENERATION ALGORITHM . . 25
5.1 Comparison of Original Graphs vs. Borromean-S DK-2 Generations 29

CHAPTER 6: BORROMEAN-PARALLEL LABELED GRAPH GENERATION ALGORITHM 31
6.1 Stage 2: Partitioning the DK-2 Distribution 32
6.2 Stage 4: Stitching the Subgraphs 36
6.3 Performance of Borromean-P-Python 42

CHAPTER 7: BORROMEAN-P SPARK: PARALLEL DESIGN AND IMPLEMENTATION 52
7.1 Spark 52
7.2 GraphX 54
7.3 Tuning and Optimization 55
7.4 Cluster Configuration 55
7.5 Adapting the Parallel Merge Algorithm for Spark 56
7.6 Performance of the Parallel Borromean 57

CHAPTER 8: DISCUSSIONS AND FUTURE WORK 60
8.1 Discussion of Results for All Algorithm Variants 61
8.2 Future Work 64

REFERENCES 65

LIST OF TABLES

Table 3.1 Datasets Used in this Study 19

Table 3.2 Labeled Attribute Percentages of the Datasets 19

Table 4.1 Properties of Orbis-Generated DK-2 Unlabeled Graphs 23

Table 5.1 Borromean-S Transitivity Comparison 27

Table 5.2 Properties of Borromean-S Generated Labeled Graphs 30

Table 5.3 Borromean-S Attribute Percentages 30

Table 6.1 Sweden 5K Labeled 4-Partition Merge 37

Table 6.2 Sweden 5K Four Partition Comparison 38

Table 6.3 Sweden 5K Four Partition Comparison - Triads 39

Table 6.4 Sweden 5K Ten Partition Comparison - Fractional and K^{th} 40

Table 6.5 Sweden 5K Ten Partition Comparison - Fractional and K^{th}- Triads 41

Table 6.6 Sweden 5K Ten Partition Comparison - Cross-Sectional and Random Shuffle 42

Table 6.7 Sweden 5K Ten Partition Comparison - Cross-Sectional and Random Shuffle - Triads 43

Table 6.8 Sweden 50K Four Partition Comparison 44

Table 6.9 Sweden 50K Four Partition Comparison - Triads 45

Table 6.10 Sweden 50K Ten Partition Comparison - Cross-Sectional and Random Shuffle 46

Table 6.11 Sweden 50K Ten Partition Comparison - Cross-Sectional and Random Shuffle - Triads . . 47

Table 6.12 Sweden 50K Ten Partition Comparison - Kth and Fractional 48

Table 6.13 Sweden 50K Ten Partition Comparison - Kth and Fractional - Triads 49

Table 6.14 Properties of Labeled Graphs Generated with Borromean-P Python 49

Table 6.15 Borromean-P Python Node Attribute Percentages 50

Table 6.16 Borromean-P Python Performance (Time) 50

Table 7.1 Properties of Borromean-P-Spark Graphs Generated in Parallel 58

Table 7.2 Borromean-P-Spark Node Attribute Percentages 59

Table 7.3 Borromean-P Spark - Parallel Performance (Time) 59

Table 8.1 Label A Attribute Percentage Comparison 61

Table 8.2 Label B Attribute Percentage Comparison 61

Table 8.3 Edge Type A-A Percentage Comparison 62

Table 8.4 Edge Type B-B Percentage Comparison 62

Table 8.5 Edge Type A-B Percentage Comparison 62

Table 8.6 Clustering Coefficient (Transitivity) Comparison 63

LIST OF FIGURES

Figure 1.1 The DK Series Representation of a Sample Graph 3

Figure 1.2 Plot of the Degree Distribution of the Sweden Graph and Its Borromean-S Generation . . . 4

Figure 1.3 Plot of the Transitivity of the Sweden Graph and Its Borromean-S Generation 4

Figure 1.4 Labeled Sample Graph 5

Figure 5.1 The Borromean-S (Labeled DK Series) Representation of a Sample Graph 26

Figure 6.1 System Design Overview 32

ABSTRACT

Real graph datasets are important for many science domains, from understanding epidemics to modeling traffic congestion. To facilitate access to realistic graph datasets, researchers proposed various graph generators typically aimed at representing particular graph properties. While many such graph generators exist, there are few techniques for generating graphs where the nodes have binary attributes. Moreover, generating such graphs in which the distribution of the node attributes preserves real-world characteristics is still an open challenge.

This thesis introduces Borromean, a graph generating algorithm that creates synthetic graphs with binary node attributes in which the attributes obey an attribute-specific joint degree distribution. We show experimentally the accuracy of the generated graphs in terms of graph size, distribution of attributes, and distance from the original joint degree distribution. We also designed a parallel version of Borromean in order to generate larger graphs and show its performance.

Our experiments show that Borromean can generate graphs of hundreds of thousands of nodes in under 30 minutes, and these graphs preserve the distribution of binary node attributes within 40% on average.

CHAPTER 1: INTRODUCTION

Real social graph datasets are fundamental to understanding a variety of phenomena, such as epidemics, adoption of behavior, crowd management and political uprisings [1, 2, 3]. At the same time, many such datasets capturing these phenomena are often recorded now by individual researchers or by organizations. However, due to privacy concerns, many such datasets cannot be publicly shared. Moreover, for experimental investigations, there is always a need for more such datasets than are available. This work focuses on techniques for generating large graphs with binary node attributes and a specified joint-degree attribute-specific edge distribution.

1.1 Motivation

Many research problems and applications that involve graph datasets benefit from knowledge of specific node characteristics. Characteristic attributes of individual nodes may designate a state, condition, or affiliation that the node exhibits. Coupling these attributes with the traditional node-edge relationship information provided by graphs allows for the study of information flow and detailed analysis of node interaction.

A wealth of descriptive metadata may be available in the underlying data sets, but it is often useful to distill node characteristics into representative binary attributes, e.g. infected/non-infected, liberal/conservative, or student/teacher. Binary node attributes are important for the simulations of information dissemination scenarios, such as adoption of behavior or epidemic diffusion. Such labeling can also be used to predict the spread of other types of contagion [1].

Characterization of real data sets showed different particularities on such attributes. This work is motivated by work done by Blackburn et al. [4, 5] that showed that node attribute values associate with node degree distribution and homophily. Specifically, users labeled as cheaters in the Steam Community online social network of gamers have a high level of homophily with other cheaters. That is, the more cheater

neighbors a player has, the more likely they are to be a cheater. Translated into graph metrics, the node attribute value is correlated with the number of edges connecting it with nodes with the same attribute value.

While many graph generators exist, few focus on graphs with node attributes. The Erdos-Renyi [6] model generates unlabeled graphs connected with a given probability. Block Two-Level Erdos Renyi [7, 8] modifies the ER model through partitioning to generate graphs that conform to a given degree distribution and clustering coefficient.

The Barabasi-Albert [9] model generates graphs with power-law characteristics by means of preferential-attachment, such that new edges will be more likely to connect to already well-connected hub nodes. The Kronecker [10] model also generates power-law graphs, but through a recursive matrix product.

To enable attribute-aware applications, it is crucial to encapsulate the needed attribute information along with the node and edge structure in the graph representation. Graph generators have been proposed that incorporate node attributes. Skvoretz et al. [2] propose a model that assigns labels to nodes as a function of a homophily parameter. However, while it preserves in aggregate the concentration of edges connecting nodes with particular combinations of attributes, it does not conform precisely to the joint-degree distribution of the graph. In fact, each of these generators lacks a distribution of binary attributes to nodes that follows a particular joint-degree attribute-specific distribution.

1.2 DK-2 Graphs

An important family of graphs and their generators have been proposed by Mahadevan et al. in [11]. The dK series random graph model is based on graph feature extraction and random graph synthesis. The dK series is a graduated extraction of the degree distribution of connected components of size K within a graph.

$$dK(G) : G^n \rightarrow d_x, d_y; k \tag{1.1}$$

Equation 1.1 is the formal function for the dK series, where G is the input graph to be transformed [12]. n is the number of nodes in the graph. d is the degree of the respective connected components, and k is the number of connected components of size $K = \{0, 1, 2, 3\}$ having that degree combination. Figure 1.1

illustrates the successive dK representations of a sample graph. In the dK-2 instance, the joint degree distributions of all two-node subgraphs are extracted. The dK-3 instance extracts the degrees of the three node triad distribution, which is helpful in accurately capturing the number of triangles in a graph, an important property for many real social networks referred to as the *clustering coefficient* in network analysis.

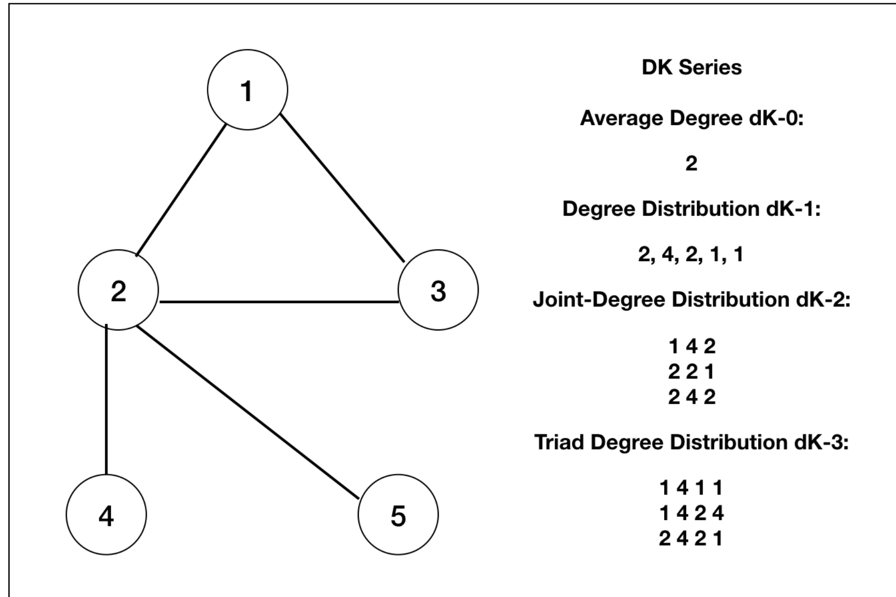

Figure 1.1: The DK Series Representation of a Sample Graph

Graphs generated from the dk series closely match the degree-based properties of their original graphs. As mentioned above and shown in Figure 1.1, each successive level of the series retains more property information about the degree distribution of nodes and their connected neighbors.

The dk-2 level gives a sufficient representation of the degree distributions of its original counterpart, as illustrated in Figure 1.2, though not of the clustering coefficient, shown in Figure 1.3. To accurately represent the clustering coefficient, the dk-3 level would be needed, but it is computationally intractable for large graphs [11].

The primary generator for the dk-2 level is the configuration [13] or pseudo-graph model. The configuration model pre-assigns edge end stubs to nodes equal to their degree and connects the edge ends at random. This technique reproduces given degree distributions exactly, but can produce self-loops and multi-edges.

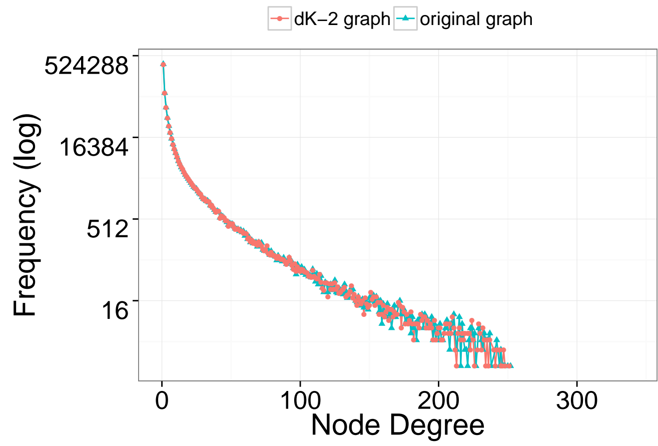

Figure 1.2: Plot of the Degree Distribution of the Sweden Graph and Its Borromean-S Generation. The graph is a subset of the Steam gaming community player friendship graph consisting of Swedish players.

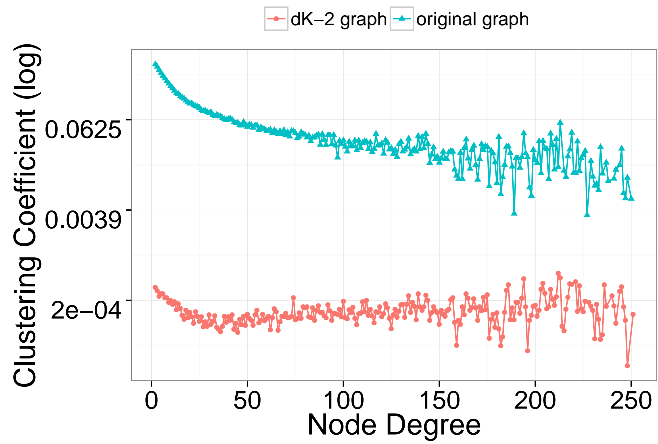

Figure 1.3: Plot of the Transitivity of the Sweden Graph and Its Borromean-S Generation. The graph is a subset of the Steam gaming community player friendship graph consisting of Swedish players. Transitivity is the global clustering coefficient.

Mahadevan et al. [11] modify the configuration model to respect dk-2 joint-degree distributions and disallow self-loops and multi-edges.

The graph generation technique used in this work is based on the dk-2 level of the series and modifications to the configuration model. We choose dk-2 for our work because of the prospect of developing a reasonable implementation and because it affords the opportunity to retain and study relationships between nodes.

1.3 Objectives and Contributions

The objectives of this thesis are twofold:

First, we want to generate labeled graphs with binary node attributes that follow a specific attribute-based joint-degree distribution. For example, let us assume we want to generate a graph in which a black node of degree 4 is connected with 2 black nodes of degrees 1 and 2 and 2 white nodes of degrees 1 and 2, as in Figure 1.4. In this case, white and black are the node binary attributes whose distribution we are interested in generating.

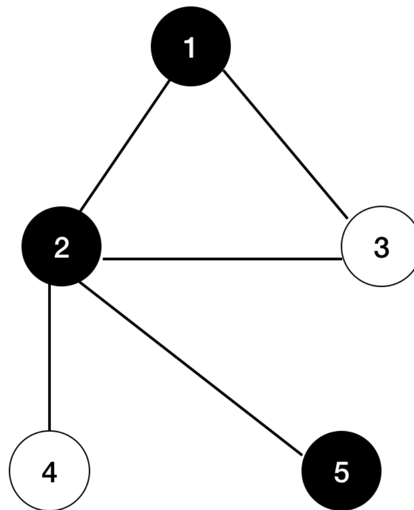

Figure 1.4: Labeled Sample Graph

Second, we want to be able to generate large graphs (in the order of hundreds of thousands nodes).

The contributions of this thesis are:

1. We propose an algorithm, Borromean, for generating graphs with node attributes by modifying the dK-2 model. We are motivated to use dK-2 because it provides fidelity to the degree-based properties of the original graph. We implemented Borromean in C++ and refer to this sequential implementation as Borromean-S. We provide thorough experimental evaluations on six datasets from real networks.
2. We propose a parallel version of the algorithm to be able to generate larger graphs in reasonable time. There are two key ideas in the design of the parallel version: the partitioning of the problem and the termination condition. The latter is meant to more accurately reproduce a desired average path length.
3. We propose the parallel algorithm in two versions. For testing the accuracy of the results we designed and implemented the parallel algorithm for Python. We refer to this version as Borromean-P-Python. For testing its performance in terms of execution time and scalability, we also designed and implemented the parallel Borromean algorithm for Apache Spark. We refer to this version as Borromean-P-Spark.
4. We also provide experimental evaluations on six datasets from real networks with binary node attributes.

The rest of the thesis is organized as follows:

- Chapter 2 provides the context for this work.
- Chapter 3 introduces the real datasets used for experiments. They are introduced early to allow us to describe experimental results related to particular topics addressed in subsequent chapters.
- Chapter 4 describes the dk-2 generation utility suite Orbis.
- Chapter 5 introduces our proposed sequential algorithm, Borromean-S, that is based on dk-2 models and generates node attribute-specific joint-degree distribution obeying graphs.
- In Chapter 6 we introduce and evaluate a parallel version of this algorithm that we name Borromean-P.
- Chapter 7 describes and evaluates the parallel implementation in Spark [14] of Borromean-P.
- Finally, Chapter 8 concludes with a discussion of lessons and future work.

CHAPTER 2: RELATED WORK

Significant effort has been invested into collecting, purchasing, or publishing social data for research. For example, Kwak et al. [15] collected the entire Twitter network as of 2010: 41.7 million user profiles, 1.47 billion social relations, 4.262 trending topics, and 106 million tweets. The Federal Energy Regulatory Commission published a repository of approximately 500,000 email messages of Enron Corporation, which has been frequently analyzed for research [16, 17]. AOL [18], Netflix, and Flickr [19] have all publicly released samples of their user graphs as part of crowd-sourced data-mining experiments.

These periodic releases do not provide sufficient data to model and simulate all possible structural characteristics and information flow needed to adequately address the multitude of research problems that require data. Researchers have turned in part to synthetic graph generation to fill the gap and provide more flexibility. We discuss synthetic generation models in Section 2.1, with particular focus on generating dK graphs in Section 2.1.4.

The fact that the real graph data sets released were sampled and anonymized and still proved to be vulnerable to re-identification from various modes of attack poses a privacy risk to users. We discuss anonymity models and their effects on the dk series in Section 2.3.

Graph data sets in the billions of nodes are so large that they cannot reasonably be processed by sequential methods. Algorithms are needed that allow them to be efficiently analyzed in parallel when distributed across a cluster of worker servers. We discuss parallel generation models in section 2.2.

2.1 Synthetic Graph Generation

Several models have been developed to allow these real world data sets to be synthetically mapped in graph relationships between nodes (i.e., users, emails, tweets, pages, locations, etc.). The synthetic models

attempt to capture various structural properties that can then be speculatively manipulated to service a myriad of hypothetical situations without compromising the original data.

Ultimately, graph generation is a question of intent and tuning of parameters. As previously stated, no synthetic generator can perfectly reproduce an original graph. Each has strengths and weaknesses with regard to individual properties, so a model must be chosen based on research interest.

For a simple example, the Erdos-Renyi [6] model generates graphs where nodes are connected with some probability p. Such random construction cannot capture degree distribution or clustering coefficient making it of little use structurally. Yet, many models augment ER with additional conditions, thereby generating graphs relatively closer to the degree distribution and clustering. Other models generate scale-free graphs where edge formation is not random but tends to concentrate from specific nodes. The most relevant of these models are described below.

Barabasi-Albert [9] generates graphs with power-law characteristics through probabilistic preferential-attachment. New edges have a higher probability to connect to nodes with greater degree. It does not respect fixed degree or attribute distributions.

2.1.1 Kronecker Generation

Lescovec et al. [10, 20] propose a means to mathematically model graphs with power-law characteristics through the recursive Kronecker product. The Kronecker product is the matrix direct product of two adjacency matrices, in this case, the product of the graph with itself.

In the resulting Kronecker graph, the original graph appears as interconnected sub-graphs up to k times in the larger matrix. Stochastic Kronecker graphs form the edges of the adjacency matrix with a probability, p, in order to produce more continuous property distributions.

For standard power-law graphs, the Kronecker model captures degree and eigenvalue distribution, as well as densification, the gradual shortening of effective diameter as the graph grows. However, like Barabasi-Albert the Kronecker model does not accurately conform to graphs that do not follow power-law behavior, even for social networks with a fixed maximum degree.

The Kronecker model is related to work with densification in the forest fire model [21]. Forest Fire joins new nodes to the graph using a random breadth first search of existing edges, adding a one node then its neighbors, continuing recursively through the graph.

2.1.2 Block Two-Level Erdos-Renyi

BTER [7, 8] extends the Erdos-Renyi model to capture degree distribution and clustering coefficient through partitioning. $d+1$ nodes of degree d are grouped into community partitions. The community members are connected with a probability based on their maximum degree to form triangles approximating the local clustering coefficient. The communities are interconnected at random by edges between the remaining degree stubs in each community.

The BTER model is more flexible than the Kronecker in its ability to produce varying degree distributions departing from the power-law, but it cannot model dissociative graphs or graphs where the local clustering coefficient varies among nodes of the same degree.

2.1.3 ERGMs

Exponential random graph generation models are a family of probability distributions originally proposed to model directed graphs [22, 23]. ERGMs can regressively generate both directed and undirected graphs with a range of network properties from given distribution values such as degree or triangle count. While ERGMs can reproduce assortativity values, they cannot accurately reproduce joint degree distributions.

Each network property can correspond to a space of possible graphs. The aim of ERGMs is to be able to generate as much of this space as statistically feasible. However, some portions of any space are not realizable, either because they do not produce stable graphs or because they would be the result of linear dependencies between properties.

2.1.4 Related DK Graph Generators

The dk graph model naturally preserves degree distributions and properties dependent thereupon, but does not preserve clustering related properties. Additionally, pseudo-graph matching generation loses edges that should be in the distribution due to the saturation of nodes with required degrees. The algorithms described in this section attempt to solve for these deficiencies.

2.1.4.1 DK 2.5

Gjoka et al. [24] propose the dk-2.5 model which improves the utility of dk-2 by specifically targeting the average clustering coefficient of the original graph. In the dk-2.5 algorithm, the joint degree distribution and clustering coefficient are estimated through a random walk of a sample of the original graph.

This process over-estimates lower degree nodes that are not easily surveyed by a random walk. The algorithm attempts to correct this using gaussian kernel smoothing; however, smoothing introduces floating-point values for degrees that must be rounded to integers and numbers of edges that cannot be matched given the existing nodes.

The algorithm uses a stochastic process (Markov Chain Monte Carlo sampling) during its swapping phase where it adds or deletes certain combinations of edges in order to minimize the error. The paper claims that this affects 3% of edges. However, MCMC is time consuming and may not converge at the targeted clustering coefficient [25].

2.1.4.2 2K Simple

2k_Simple [25] attempts to modify the Mahadevan et al. dk-2 compatible configuration model [26] in order to generate dk-2 graphs that exactly match the joint-degree distribution of the input graph. The most notable contribution of the work is a proposed solution to the edge starvation problem induced by earlier configuration approach. If a edge saturated vertex is encountered, the NeighborSwap function in moves an edge from the vertex with no open stubs to a vertex of equal potential degree and a free stub. The target node remains the same. Thus, the joint-degree distribution is completely preserved.

2K_Simple_Attributes extends 2K_Simple to add attribute awareness by matching source and target node degree along with their attributes to the joint-degree distribution. 2K_Simple_Attributes is the closest proposed alogrithm to Borromean in the literature. 2K_Simple_Clustering attempts to improve fidelity to the clustering coefficient by assigning nodes to a coordinate grid and connecting them by order of distance.

The implementation used in 2K_Simple differs from Borromean-S, though both build from the configuration model. 2K_Simple uses a custom graph class and NeighborSwap. The performance for 2K_Simple_Attributes cannot be directly compared, as the source code for 2K_Simple_Attributes and 2K_Simple_-Clustering has not been published.

2.2 Parallel Graph Processing Frameworks

2.2.1 Giraph

Apache Giraph is a graph analysis platform based on the vertex-centric bulk synchronous processing model similar to Pregel [27]. Facebook published a VLDB paper [28] describing the contributions they made to Giraph such that it could handle their production workloads of up to a trillion nodes. They also published a comparison [29] of the modified version of Giraph with Apache Spark's GraphX library. Spark was originally designed to be the successor to Giraph.

Chiefly, the contributions of the paper improved memory overhead by storing worker data as byte arrays rather that Java objects, improved parallelism by instituting multi-threading per worker and a sharded aggregation model where random workers are chosen to gather the reduction results of their neighbors for global variables rather than bottleneck at a single central driver. The processing of incoming edge messages was also split so that multiple workers can process the traffic of a single well-connected vertex if needed.

Giraph performs the PageRank calculation 3 times faster than GraphX on a 1.5 billion edge sample of the Twitter graph with 16 workers, one per machine, and 80 GB of RAM. On a UK web graph with 3.7 billion edges, Giraph is 5 times faster. Giraph finds the connected components of the Twitter graph 5 times faster than GraphX. Giraph is also 3 times faster on average when calculating PageRank on a synthetic graph of 2 billion edges and 6 times faster finding connected components.

GraphX can complete PageRank on a 10 billion edge in approximately 18 minutes, and connected components in 12. Giraph is 4 and 8 times faster, respectively.

GraphX is limited in instances above 10 billion edges by high memory usage. Even with 2 billion edges, GraphX requires at least 20 GB per worker over sixteen machines.

The performance for triangle counts was also compared. Giraph completed the count in 40 minutes with 32 workers on the UK graph, but GraphX could not complete a count for a graph over 2 billion edges.

With 50 workers, page rank calculation using Giraph on a graph of 200 billion edges takes approximately 6 minutes. The page rank calculation exhibits a linear time increase as edges are added with a fixed number of workers. The horizontal scalability curve approaches 2 minutes when 300 workers are added. A page rank calculation over 1 trillion edges and 1.39 billion nodes was achieved in 3 minutes using 200 machines.

While the modified Giraph shows better performance with basic graph analysis using the Pregel/BSP algorithms it is designed for, it is still not well-suited for graph transformations such as those we were able to perform in GraphX/Spark and are needed for Borromean. In Borromean, the graph must be deconstructed, manipulated, and remade. Spark is a wider, general purpose platform that allows greater customization and manipulation of data.

2.2.2 Darwini

Darwini [30] is a synthetic graph generator similar to BTER that runs on top of Giraph. Darwini marks each node with its target degree and target clustering coefficient before partitioning into communities. The algorithm groups vertices that participate in the same number of triangles into community partitions. Communities are connected, not at completely at random as in BTER, but according to the target distributions. The triangles are completed by adding edges according to the Erdos-Renyi model with a probability derived from the the clustering. After this stage, the vertices in each community are still under-connected. The degree distribution is completed by connecting random vertices between partitions. Communities can be formed with vertices having a heterogeneous number of triangles participated in. This may result in an incorrect clustering coefficient. The paper claims this is preferable to incomplete partitions. Thus, the gen-

erated degree and clustering coefficients are closer to the original than BTER, but the generated effective diameter cannot be controlled [31].

The algorithm uses these communities as a basis for parallelism. It processes each community in a parallel, distributed manner using the vertex-centric bulk synchronous parallel approach in GraphInc, giving linear scalability. Very large graphs are split into super-communities that span the highest density of nodes in a region. Processing proceeds within super-communities, then between, recursively. The algorithm is particularly memory intensive in instances where entire super-communities must be loaded, even though the graph as a whole need not be loaded. The algorithm can accommodate graph data sets with trillions of edges. The entire Facebook graph was processed in 7 hours on an industrial cluster.

2.3 Graph Generators for Graph Anonymity

As more property information, and therefore utility for research, is retained in synthetic models, the space of random graphs that can be generated while fitting the model is increasingly restrained. The generated graph will closely resemble the original input graph if all the structural information required to faithfully represent every property is retained.

However, researchers working with social graphs need to exercise particular sensitivity when publishing their data sets as they may contain a wealth of user identifiable information in both their nodes and structure [32]. To this end, many graph generation models at least partially predicate their motivation on the need to privatize data, but anonymization methods such as k-anonymization [33], partitioning [34], differential privacy [35], which we discuss in the next section, and dk-2 anonymized graph generation [11, 12] all retain some degree of vulnerability to deanonymization attacks [32, 36], particularly those aided by machine-learning [19].

K-anonymization [33] algorithms attempt to increase the size and homogeneity of candidate sets through various permutations such as addition, deletion or randomization of edges. A natural complication to re-identification are regions of the graph that are structurally isomorphic. When analyzing isomorphic regions, the attacker's finest level of discernment is a candidate set that contains all isomorphic nodes. The attacker's counter potential lies in his or her ability to refine the candidate set. K-anonymization ensures each member

node within a possible candidate set is homogeneous in terms of degree such that a node would be isometric to $k-1$ neighbors. Therefore, although not targeted to social network data publishing, k-anonymity protection ensures that the information for each person contained in a released data cannot be distinguished from at least $k-1$ other individuals in the data [37].

Partition or class based anonymization algorithms [34] partition the graph according to similarity of structural features. Interaction between these partitions is limited and any inter-connection edges are consolidated. The partitions in this way form candidate sets difficult for attackers to distinguish. Partitioning also provides anonymity by clustering nodes and edges into groups that are then represented in the anonymized graph as super-nodes [38].

While preserving user privacy, the published graph must also maintain as much utility as possible for structural analysis and research. Yet, the process of anonymization often distorts structural properties of relevance to researchers along with node identity. For example, existing k-anonymous graph models degrade the utility of degree-based metrics and in addition have prohibitive run times for large data sets. Even statistical generation models like differential privacy can obscure the utility of graph properties that are especially resistant to edge manipulation with the injection of copious noise. Thus, there is no comprehensive solution that provides unassailable privacy and utility while allowing unfettered distribution.

In many cases, data sets are released in an *identity-scraped* form, where personally identifying information associated with each user is either removed altogether or substituted with a random ID [39]. Yet sharing real social graphs, even with node-identifiable information removed, has been proven over and over again to be dangerous for the privacy of the individuals represented by the nodes of such graphs because naive anonymization does not in itself alter or obscure the structure of the graph. .

For example, in 2006 AOL released an anonymized data set of twenty million search keywords for over 650,000 users [40]. Despite the fact that the data released was identity-scraped, users' privacy was compromised. To make the point, the New York Times identified an individual from this data set by cross referencing users with phone book listings [18].

In other cases, Narayanan et al. [19, 36, 41] demonstrated the feasibility of a large-scale machine-learning based de-anonymization attack under the assumption that the attacker has background knowledge of a

different network whose membership partially overlaps with the target network. Machine-learning attacks bolster the attacker's knowledge by training with this existing data. This training generates a vast array of background information which can be compared to potential target nodes. If the attacker has even partial knowledge of the adjacent edges to the target node, then it may well be possible to re-identify the target and its neighbors. The external information risk is high because basic identifying data is often public. Narayanan et al. showed that a third of the common users of Flickr and Twitter can be recognized in the completely anonymous Twitter graph with only 12% error rate.

Also, structural anonymization compromises the utility of the original graph. Aggarwal et al. [42] demonstrated that utility degrades very quickly while privacy is achieved very slowly in real, social networks with approaches that randomly rewire the graph, due to hidden structural signatures in large, sparse networks.

2.3.1 Differential Privacy and DK-2 Graphs

Differential privacy is an anonymization method that injects statistical noise into a data set such that any one user will be indistinguishable from any other. Differential privacy algorithms [43, 35, 44] perturb the entries of a data set, e.g. the rows of a table or the edges of the graph, with probabilistic random noise such that the data of any given entry matches any other within a certain small limit, ε.

The perturbation attempts to address the problem of preserving the attributes and properties of a data set as a whole, in the original formulation a statistical database, while maintaining the anonymity of individual users. This technique is adapted in later literature [12, 45] to anonymize graphs as well. In each graph, structural properties have specific sensitivities to perturbation. The lower the sensitivity, the greater the noise required for anonymization of that property for the graph.

It is useful to understand how dk-2 graphs are affected by the differential privacy model. Sala et al. [12] apply the anonymization techniques of the model to dk-2 graphs. The dk-2 series is used as a non-interactive structural property query on graph data sets. The query extraction is executed, then the anonymization occurs once for the entire set as opposed to the interactive model where selected parts of the data are dynamically

anonymized as they are requested. Multiple dK models can be generated from the same data set without losing privacy.

The goal of the differential privacy condition is to produce an anonymized graph that is probabilistically close to the original graph within a factor ε. Since the dK series extraction is the query in this instance, the probability of the output graph having the same structures as the original input graph should be within ε. ε is inversely proportional to data fidelity; as ε grows, the similarity between the two graphs decreases. Assume the neighbors of G are similar graphs each differing by an edge. The sensitivity of the dK-2 series is the maximum number of changes in the set among neighbors. The sensitivity of dK-2, S_{dK-2} , is bounded by the maximum degree, d_{max}, specifically $4d_{max}+1$.

In order to ensure anonymity with this method, the probability distributions of statistical property queries on both the original database and its counterpart must be statistically close. To meet the indistinguishability condition, the absolute value of the natural log of the ratio of the two distributions must be less than or equal to a small constant. In practice, the noise needed by differentially private anonymization is much too large to ensure accurate utility. Also, differentially private graphs are still vulnerable to machine-learning attacks.

2.3.2 Pygmalion: Differentially Private DK-2 Graph Anonymization

Sala et al.[12] propose Pygmalion, a differentially private graph model emphasizing edge privacy that statistically extracts the structure of the original graph as a degree correlation set using the dK-2 series model [11], then partitions this set using a degree-based clustering algorithm in order to minimize the noise needed to reach the differentially private condition.

The algorithm reduces the degree variance in each cluster. It is claimed that this reduction lessens the noise needed by an order of magnitude. The calculated noise is then injected into the set. Isotonic regression is used to evenly distribute the noise, mitigating the effective error, it is claimed, by 50%. Further, it is claimed that the generated graph is a close match to the standard metrics and experimental utility of the original graph.

Sala et al. use the Laplace mechanism to generate random noise. Unfortunately, the noise grows polynomially with node degree, so before noise is injected the dK-2 series set needs to be sorted and clustered. The expected noise error is far too large to produce graphs with any accuracy.

CHAPTER 3: DATASETS

We chose six publicly available datasets from four different contexts and generated networks with binary node attributes. The datasets include selections from the Facebook 100 university network collection labeled by faculty/student status [46], a political blogging network labeled by liberal/conservative affiliation [47], and a portion of the Amazon product network labeled by book/music product type [48]. The following details these data sets.

- `polblogs` [47] is an interaction network between political blogs during the lead up to the 2004 US presidential election. This dataset includes ground-truth labels identifying each blog as either conservative or liberal.

- `fb-dartmouth`, `fb-michigan`, and `fb-caltech` [46] are Facebook social networks extant at three US universities in 2005. A number of node attributes such as dorm, gender, graduation year, and academic major are available. We chose the occupation node attribute occupation, represented by the values "student" and "faculty".

- `amazon-products` [48] is a bi-modal projection of categories in an Amazon product co-purchase network. Nodes are labeled as "book" or "music", edges signify that the two items were purchased together.

- `steam sweden` [49] is a subset of the Steam online gaming service player friendship graph. The subset corresponds to the Swedish population of players.

Table 3.1: Datasets Used in this Study. Column Trans presents the clustering coefficient of the graph, and column Avg Path presents the average path length. Column Trans is the transitivity, or global clustering coefficient.

Data Set	#Nodes	# Edges	Trans	Avg Path
Polblogs	1224	16718	0.22	2.49
FB-Caltech	769	16656	0.29	1.33
FB-Dartmouth	7694	304076	0.15	2.76
FB-Michigan	30147	1176516	0.13	3.05
Amazon Products	303551	835326	0.21	17.42
Steam Sweden	749878	2008476	0.13	6.10

Table 3.2: Labeled Attribute Percentages of the Datasets. Column A-A presents the percentage of edges that connect nodes of type A, B-B presents the percentage of edges that connect nodes of type B, and A-B the percentage of edges that connect nodes of different types.

Dataset	Label A %	Label B %	A-A %	B-B %	A-B %
Polblogs	48	52	44	48	8
FB-Caltech	72	28	69	8	23
FB-Dartmouth	62	37	58	18	24
FB-Michigan	78	22	72	9	19
Amazon Products	81	18	83	2	16
Steam Sweden	97	2	84	0.9	14

CHAPTER 4: ORBIS DK-2 TOPOLOGY GENERATOR

The Orbis graph topology suite [11, 50] is a collection of utilities designed to generate and manipulate the dK series and its respective random, synthetic graphs. Orbis is written in C++ and requires the Boost graph library [51]. Graphs are represented by Boost AdjacencyList graph data structures, and the degree distributions are represented by nested map data structures from the C++ standard template library.

Orbis uses dK degree distributions to produce graphs exhibiting properties conforming to dK levels 0, 1, 2, and 3. Orbis can only directly generate dK-0, dK-1, dK-2 graphs. It uses edge rewiring algorithms to approximate dk-3.

In the dk-2 instance, the distribution list is of the form $[d_u, d_v, t]$, where d_u is the degree of the first vertex in the pair, d_v is the degree of the second vertex in the pair, and t is the total number of edge occurrences of pairs with those degrees in the graph. There is only one line per unique degree pair.

The dkDist utility extracts the dk degree distribution from the original graph. The algorithm for dkDist is shown as Listing 4.1.

Orbis' primary means of dK-2 generation is a pseudo-graph matching, or configuration, algorithm implemented in dkTopoGen2k. The algorithm for dkTopoGen2k is shown as Listing 4.2.

dkTopoGen2k labels each node with edge ends in accordance with its degree. The ends match each edge in the edge set. The edge ends are randomly connected with the connect stubs procedure listed as Algorithm 4.3.

The edge stubs are connected with restrictions following the degree distribution of the original graph or the joint degree distribution in the dK-2 instance. For example, if the tuple in the JDD is $(d, d') = m$, then an edge end corresponding to a node of degree d must be connected to a target end on a node of degree d'. The

process will be repeated for m edges and in turn for all tuples in the JDD. Edge ends will not be connected if it would result in a self-loop to the same node or a multi-edge with an extant edge between two nodes.

4.1 dK Rewiring

dK preserving rewiring is a random graph generation technique that moves edges between random pairs of nodes so long as such a move would preserve the properties of the given dK distribution. No edges are added or removed. For example, dK-2 rewiring must preserve the joint degree distribution and dK-3 rewiring must preserve wedge and triad distributions. Section 4.1.4 of [11] discusses the general approaches.

Movement of edges in this manner continues until the graph converges on a stable state where further rewirings do not alter dK distribution properties. The basic preserving algorithm requires the original graph and cannot create a random graph from a dK tuple list alone. It has a runtime of $O(m)$.

dK targeted rewiring moves from a dK graph to a higher, more descriptive level based on a target (tuple list) distribution. For example, a dK-2 graph can be rewired to a dK-3 graph based on a dK-3 tuple list. The algorithm is essentially a version of the Metropolis-Hastings algorithm, using Metropolis dynamics as an acceptance function.

Some graphs representing higher order dK levels are nonergodic, i.e. they do not converge to a desired stable state. This is chiefly exhibited in graphs having dK-4 or higher.

4.2 Limitations of the Orbis DK-2 Pseudo-Graph Generation Model

In combination with randomization, the degree matching, self-loop, and multi-edge restrictions of the pseudo-graph generation model can cause an edge starvation condition. The condition occurs because all eligible target edge ends have already been occupied before all edges in the JDD are formed. For example, the dk-2 graphs of each of the network samples of the Sweden subgraph differ in edge set cardinality from their original counterparts by approximately 1.1% fewer edges.

The pseudo-graph matching method becomes computationally impractical for dk-3 and above [12]. There is currently no graph generator algorithm for K greater than or equal to 3, as each level of fidelity

demands higher computation and storage requirements. This difficulty is concerning because social network research requires the preservation of community structures and metrics including the relationship of nodes with specific attributes to their neighbors. The dk-2 distribution shows this in part, but to study how cheating behavior propagates through the graph, it is also useful to measure the global clustering coefficient, or transitivity, as shown in Equation 4.1. The dk-2 random graph generation model preserves degree-based properties but obscures clustering-based properties. For example, we have empirically confirmed that the dK-2 distribution does not accurately preserve clustering coefficients. Table 4.1 shows the structural properties including the transitivity of graphs generated by the original Orbis unlabeled dk-2 pseudo-graph matching algorithm.

$$C = (number\ of\ triangles) * 3/(number\ of\ 2\ paths) \tag{4.1}$$

Algorithm 4.1 DK-2 Distribution Generation (dkDist)

1: Input Graph G
2: Output Joint Degree Distribution
3: **procedure** READ INPUT GRAPH (Edge List)
4:     **for** edge $\in E$ **do**
5:         set node degree in Boost graph structure
6:     **end for**
7: **end procedure**

8: **procedure** GET 2K DISTRIBUTION
9:     **for** edge $\in$ Boost graph G **do**
10:         retrieve Boost degree count for each node
11:         increment JDD map (NKKMap) count based on degree combination
12:     **end for**
13: **end procedure**

Algorithm 4.2 DK-2 Graph Generation (dkTopoGen2k)

```
 1: procedure READ 2K DISTRIBUTION
 2:     for tuple ∈ dK-2 distribution do
 3:         recreate NKKMap
 4:     end for
 5: end procedure

 6: procedure DKTOPOGEN2K
 7:     for level in NKKMap do
 8:         approximate node count per degree (degree distribution NKMap)
 9:         round(edges / degree + 0.5)
10:     end for
11:     for degree of each node in NKMap do
12:         create edge end stub structure
13:         create list of stubs ordered by node degree
14:     end for
15:     shuffle stub list

16:     initialize vector matrix of available stubs (all available)
17:     create working copy from NKKMap
18:     create randomly shuffled list of all stubs
19: end procedure
```

Table 4.1: Properties of Orbis-Generated DK-2 Unlabeled Graphs. The APL column is the effective diameter of the graph.

Data Set (GCC)	Nodes	Edges	Transitivity	APL
Polblogs	1219	16667	0.17	2.93
FB-Caltech	760	16615	0.16	2.76
FB-Dartmouth	7679	304033	0.03	2.90
FB-Michigan	30027	1174804	0.01	3.33
Amazon Products	291274	803831	0.0001	7.27
Steam Sweden	739817	1992747	0.0001	5.87

Algorithm 4.3 DK-2 Graph Generation (dkTopoGen2k Connect Stubs)

```
 1: procedure CONNECT STUBS
 2:     while ¬randomStubIds.empty() do
 3:         stubId = randomStubIds.front()
 4:         randomStubIds.pop_front()
 5:         if available[stubId] then
 6:             get2kRandomDegree(stub.edge_type, stub.degree, workingMap)
 7:             total edges from workingMap[source degree]
 8:             target edge = rand() mod total edges
 9:             increment through workingMap[source degree] until target edge
10:             targetDegree = second degree of target edge tuple
11:             targetList = DegreeStubListMap[targetDegree]
12:             for stub ∈ targetList do
13:                 if edge type match && available[stub] && edge not connected then
14:                     add_edge to Boost graph
15:                     set stubs as not available
16:                     decrement edge in workingMap
17:                 end if
18:             end for
19:         end if
20:     end while
21: end procedure
```

CHAPTER 5: BORROMEAN-SEQUENTIAL LABELED GRAPH GENERATION ALGORITHM

We have developed an algorithm for the generation of labeled graphs, Borromean-Sequential, that preserves the node attribute typed edge relationships from an original labeled graph within its randomly synthesized counterpart. Borromean-S generates labeled dk-2 graphs by modifying the algorithms used by Orbis. We selected the dK-2 distribution [26] because it provides a reasonable compromise between computational cost and fidelity with regard to graph metrics. The relevant procedures of the labeled algorithm are listed as Algorithms 5.1, 5.2, 5.3, and 5.4.

The first phase is found in the Borromean-S version of the *dkDist* utility, whose unlabeled Orbis implementation is discussed in Chapter 4. The Algorithm 5.1 reads the original graph, notes the node attribute for each node, and counts the degree for each node. It then creates a nested map data structure, *Labeled_NKKMap*, containing the unique edge types as keys. Edge type is a string signifying the combination of node attributes possessed by the edge ends. Specifically, it could be A-A, B-B, or A-B if the binary attributes for the graph were A and B. The algorithm uses an additional mirrored B-A edge type to distinguish the node attribute of the first node in the pairing found in the graph. The nested map is indexed as a multi-dimensional array with indexes i and j. Index i is the degree of the first node in the pairing; index j is the degree of the second. The corresponding value to each set of indexes is the total number of edges present for each combination. The complete map represents the labeled dk-2 joint-degree distribution. Finally, the distribution is written to a file. Figure 5.1 presents an overview of the Borromean-S labeled dk series distributions.

The second phase takes the distribution map file as input to the Borromean-S version of the *dkTopoGen2k* utility. The Algorithm 5.2 reads the labeled dk-2 distribution file to recreate the *Labeled_NKKMap* data structure. As in Chapter 4, Algorithm 5.3 then approximates the node count per degree by summing the total number of edges in the distribution and dividing by each degree. The algorithm creates the nodes of this count and assigns each node a number of edge ends, or stubs, equal to its degree. Algorithm 5.4 matches

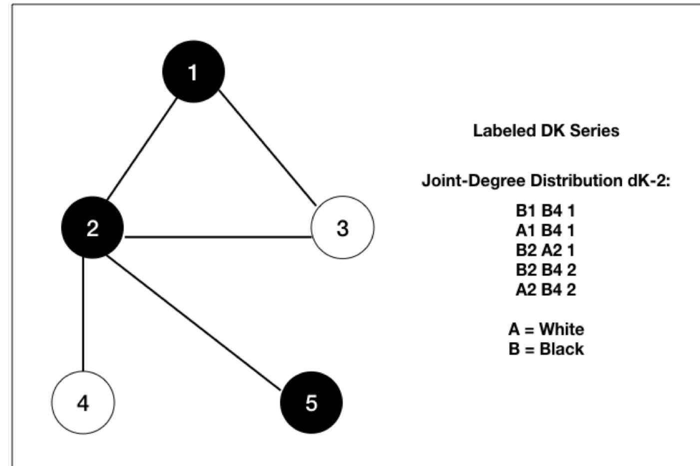

Figure 5.1: The Borromean-S (Labeled DK Series) Representation of a Sample Graph

edge ends at random according to the distribution map. Edge ends must match both the node attributes and the degrees of the distribution pairing. Specifically, if the distribution pairing line is *A 3 B 5 10*, then the first node must be of attribute type A with degree 3, and the second node must be of attribute type B with degree 5. The algorithm connects matching ends with edges. Edge end matching continues until all eligible ends have been connected.

The resulting ratios of edges connected between node attribute types are statistically similar to the original graph. As shown in Table 5.2, respecting labeled attributes adds additional restrictions and further exacerbates edge starvation problem seen in Chapter 4 in some cases where the program is prevented from creating multi-edges between nodes. Type and degree appropriate target edge ends are randomly selected; it may be that the only remaining type appropriate edge end is on a node to which the source node is already connected.

Theoretically, the additional restrictions on graph structure with regard to the labeling should improve the clustering coefficient. In general, our experiments show this to be true. Table 5.1 shows the clustering coefficient (transitivity) comparison for the original graph, the Orbis dk-2 generation, and the Borromean-S labeled generation. Borromean-S improves upon the transitivity versus Orbis in all data sets except for the Amazon products graph. The product co-purchase relationships do not represent a social network as the

other graphs do. The graph has relatively fewer edges for its node size, and the average path length is much greater.

Table 5.1: Borromean-S Transitivity Comparison. Clustering Coefficient (Transitivity) Comparison Between the original graph, the Orbis dk-2 generation, and the Borromean-S generation.

Data Set	Original	Orbis DK-2	Borromean-S
Polblogs	0.22	0.17	0.19
FB-Caltech	0.29	0.16	0.18
FB-Dartmouth	0.15	0.03	0.05
FB-Michigan	0.13	0.01	0.02
Amazon Products	0.21	0.0001	0.00005
Steam Sweden	0.13	0.0001	0.0002

Our algorithm further restricts edge formation to nodes of specific labels or types based on attributes. In the Steam player friendship graph, VAC banned users are labeled cheaters (C) and non-VAC banned users are labeled non-cheaters (NC). The Steam graph as a whole as approximately 7% cheaters [49]. Thus, the graph has three types of edges: cheater to cheater (C-C), cheater to non-cheater (C-NC), and non-cheater to non-cheater (NC-NC). The algorithm uses four edge designations, adding the NC-C mirror of C-NC so as to distinguish mixed edge types and correctly identify the needed node attribute type of the target node.

Algorithm 5.1 Borromean-S Labeled Distribution Generation (dkDist)

1: Input Graph G
2: Output Joint Degree Distribution
3: **procedure** READ INPUT GRAPH (Edge List)
4: **for** $edge \in E$ **do**
5: set node label and degree attributes in Boost graph structure
6: **end for**
7: **end procedure**

8: **procedure** GET 2K DISTRIBUTION
9: **for** $edge \in Boost\ graph\ G$ **do**
10: retrieve Boost degree count for each node
11: increment JDD map (labeled NKKMap) count based on degree combination and edge type
12: **end for**
13: **end procedure**

Algorithm 5.2 Borromean-S DK-2 Labeled (Read 2K Distribution)

```
1: procedure READ 2K DISTRIBUTION
2:     for tuple ∈ labeled dK − 2 do
3:         recreate labeled NKKMap
4:     end for
5: end procedure
```

Algorithm 5.3 Borromean-S DK-2 Labeled Graph Generation (dkTopoGen2k)

```
1: procedure DKTOPOGEN2K
2:     for level ∈ labeled NKKMap do
3:         approximate node count per degree (degree distribution labeled NKMap)
4:         round(edges/degree + 0.5)
5:     end for
6:     for degree of each node in labeled NKMap do
7:         create and label edge end stub structure
8:         create list of stubs ordered by node label and degree
9:     end for
10:    shuffle stub list

11:    initialize vector matrix of available stubs (all available)
12:    create working copy from labeled NKKMap
13:    create randomly shuffled list of all stubs
14: end procedure
```

Algorithm 5.4 Borromean-S DK-2 Labeled Graph Generation (Connect Stubs)

1: **procedure** CONNECT STUBS
2:     **while** $\not$randomStubIds.empty() **do**
3:         stubId = randomStubIds.front()
4:         randomStubIds.pop_front()
5:         **if** available[stubId] **then**
6:             get2kRandomDegree(stub.edge_type, stub.degree, workingMap)
7:             total edges from workingMap[type][source degree]
8:             target edge = rand() mod total edges
9:             increment through workingMap[type][source degree] until target edge
10:             targetDegree = second degree of target edge tuple
11:             targetList = labeledDegreeStubListMap[label][targetDegree]
12:             **for** stub $\in$ targetList **do**
13:                 **if** edge type match && available[stub] && edge not connected **then**
14:                     add_edge to Boost graph
15:                     set stubs as not available
16:                     decrement edge in workingMap
17:                 **end if**
18:             **end for**
19:         **end if**
20:     **end while**
21: **end procedure**

5.1 Comparison of Original Graphs vs. Borromean-S DK-2 Generations

In Table 5.2, we give the properties for the full Borromean-S labeled dk2 graphs generated from the data sets used. As described in Chapter 4, there is a loss of nodes and edges from the edge starvation of the pseudo-graph matching/configuration construction algorithm and the restriction of the graph to its greatest connected component. Such a restriction is reasonable since most social networks have a giant component that holds 50-90% of their nodes [52].

As expected, the accuracy of the clustering coefficient suffers greatly and the average path length is affected. The percentage of the second labeled attribute also drops, though the Facebook Dartmouth and Amazon product networks hold well.

Since dk-2 generation through psuedo-graph matching, and subsequently Borromean-S, damages the clustering coefficient, in the following iterations of the Borromean algorithm we specifically avoid breaking triangles.

Table 5.3 presents the labeled attribute node and edge type percentages for the Borromean-S dk-2 generated graphs. With the exception of the FB-Dartmouth and Amazon dk-2 graphs, there is a general loss of label A nodes and a gain in label B nodes. There is also a general loss of the label A-A edge type and a gain of the mixed label A-B edge type.

Table 5.2: Properties of Borromean-S Generated Labeled Graphs

Data Set (GCC)	Nodes	Edges	Transitivity	APL
Polblogs	617	7839	0.19	2.77
FB-C	517	11508	0.18	2.48
FB-D	4076	176635	0.05	2.79
FB-M	20902	847570	0.02	2.97
Amazon	239515	673449	0.00005	6.92
Steam SE	643966	1681319	0.000185	5.97 (ED)

Table 5.3: Borromean-S Attribute Percentages

Data Set	Label A %	Label B %	A-A %	B-B %	A-B %
Polblogs	66	34	31	48	19
FB-C	79	20	53	7	39
FB-D	63	36	39	13	46
FB-M	65	34	42	12	46
Amazon	81	18	57	5	37
Steam SE	88	12	53	6	40

CHAPTER 6: BORROMEAN-PARALLEL LABELED GRAPH GENERATION ALGORITHM

Borromean-S takes many hours to generate labeled dk-2 analogs for large graphs. Specifically, it takes more than 19 hours to generate a labeled dk-2 analog for the 750,000 node and 2 million edge Steam Sweden graph on a server with 32 GB of RAM. Since the generation and analysis of large graphs is a computationally intensive endeavor, we propose a parallel design and implementation for our labeled dk-2 synthetic random graph generation algorithm, Borromean-P. We chose Python for the initial implementation of Borromean-P because it is a highly extensible language with support for a large number of API modules providing graph transformation and analysis. Python is not a parallel language; however, we implement our algorithm in Python first to serve as a basis for further experiments in intrinsically parallel languages.

The four stages of the algorithmic design are illustrated in Figure 6.1 and described in detail below. The algorithm first extracts the dk-2 joint-degree distribution of an input graph using Borromean-S. The distribution is then subdivided into a given number of partitions. A dk-2 graph is generated from each partition using Borromean-S. Next the partition subgraphs are merged or stitched together by swapping edges between partitions. Edges with matching attribute types and degrees are identified in both subgraphs. The edges are swapped such that their terminal nodes are in different subgraphs. Specifically, an edge with an originating node of attribute A and degree 5 and a terminating node of attribute B and degree 7 in subgraph X is swapped with an edge of the same originating and terminating properties in subgraph Y. The originating node in subgraph X is connected to the terminating node in subgraph Y. Similarly, the originating node in subgraph Y is connected to the terminating node in subgraph X. The original connections are removed. In order to mitigate any further loss to the clustering of the graph, candidate edges for swapping must not be a part of existing triads. Swapping terminates when a given average path length is reached.

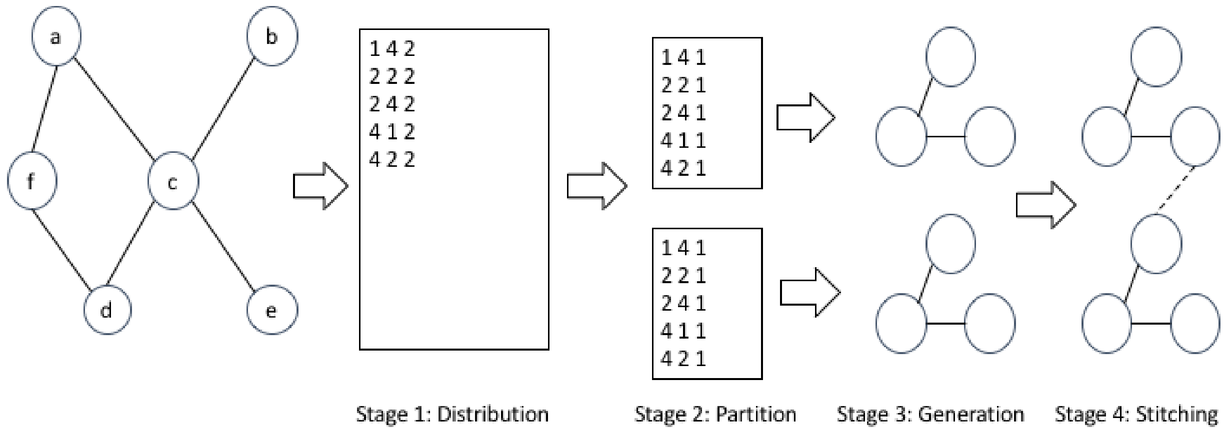

Figure 6.1: System Design Overview

In the first stage, the original graph is processed through the Borromean-S version of the *dkDist* utility to extract its signature as a listing of the joint degree distribution as detailed in Chapter 5. The procedure is illustrated in Stage 1 of Figure 6.1.

6.1 Stage 2: Partitioning the DK-2 Distribution

In the second stage of Borromean-P, we partition the dk-2 joint degree distribution before the any graphs are generated. It is useful to partition the graph generation into separate tasks to improve the tractability of our generation algorithm. The partitions provide a basis for parallel implementation allowing for faster processing on smaller portions of data and requiring less communication across workers. In addition, partitioning provides access to a larger merged vertex set than does the un-partitioned GCC dk-2 generation. The dk-2 distribution is partitioned into *N* sub-partitions of similar sizes.

We used 5,000 and 50,000 node samples of the Steam Sweden graph to test various methods of partitioning the distribution. The graphs are sampled via breadth first search from randomly chosen cheater nodes

until the node size is reached and the percentage of cheater nodes reaches that of the original Steam Sweden graph.

Tables 6.2, 6.4, 6.6, 6.8, 6.10, and 6.12 present our results in evaluating partitioning schemes. The tables also serve as a numerical comparison between the structural properties of the dk-2 graphs of the 5,000 and 50,000 node variations of the Sweden data set and their original analog. The relative differences between network properties are also quantified. We present results for four and ten partitions, respectively. The four schemes evaluated are cross-sectional, random shuffle, k^{th} partition, and fractional. Our goal is to find a partitioning method that optimizes the balance of structural properties across partitions

Simple cross-sectional partitioning divides the dk-2 distribution across a given number of partitions. Random shuffle partitioning randomly sorts the listing, then divides the distribution as in cross-sectional. A Kth partitioning division moves distribution lines into partitions in a round robin fashion. The fractional method divides the total edges of each pair by the number of partitions N and distributes the integer ceiling of this edge quotient to each partition such that the distribution lines become $[d_u, d_v, \lceil t/N \rceil]$.

In cross-sectional partitioning, the distribution is sorted in ascending order with larger total edge counts at the bottom of the listing resulting in the later partitions having a greatly imbalanced share of edges. Table 6.2 presents the nodes, edges, transitivity, average clustering coefficient, and average path length for the 4-way partitioning of the Sweden 5K Borromean-S labeled dk-2 graph. Cross-sectional partitioning results in one partition with 4602 nodes, one with 855, and an overall 69.86% increase from the original graph in nodes summed across partitions. The number of edges increases 61.78% overall. The transitivity drops 91.32%. Table 6.3 presents the triad properties for the Sweden 5K graph. The closed triads drop 93.23% from the original.

Table 6.6 presents the properties for the 10-way cross-sectional and random shuffle partitioning of the Sweden 5K graph. The nodes and edges are unevenly distributed as in the 4-way partitioning. One partition has 5957 edges while another has 976. The number of nodes increases 80.30%, and the number of edges increases 60.97%. The transitivity drops 90.61%, and the number of closed triads drops 90.75% as presented in Table 6.7.

Table 6.8 presents the properties for the 4-way partitioning of the Sweden 50K graph. The cross-sectional nodes and edges remain unevenly distributed with 48,488 nodes and 223,115 edges in partition subgraph 1. The number of nodes increases 46.65%, less than the 5K graph, but still significant. The number of edges increases 47.85%, and the transitivity decreases 89.35%, slightly less than the 5K. The closed triads decrease 88.15% as presented in Table 6.9. Table 6.10 presents the properties of the 10-way cross-sectional and random shuffle partitioning of the Sweden 50K graph. The cross-sectional nodes increase 73.20%, and the edges increase 68.60%. Partition subgraph 1 has 43,304 nodes and 151,489 edges. The transitivity decreases 89.70%, and the closed triads decrease 89.73% as presented in Table 6.11.

Random shuffle partitioning exhibits large total edge counts that may still imbalance the partitions considering that distribution lines are placed in partitions with their whole count. The distribution of nodes in the random shuffle 4-way partitioning of the Sweden 5K graph is more even than the cross-sectional with the largest partition having 2115 nodes and the smallest having 1882. However, the total node count increases 61.90% from the original graph. The number of edges increases 51.44% overall. The transitivity drops 91.32%, and closed triads drop 90.14%.

In the random shuffle 10-way partitioning of the Sweden 5K graph, the largest partition has 1092 nodes and the smallest has 593. The number of nodes increases 62.46%, and the number of edges increases 38.41%. The transitivity drops 80.36%, and the number of closed triads drops 86.00%.

In the random shuffle 4-way partitioning of the Sweden 50K graph, the largest partition has 22,612 nodes, and the smallest has 19,980. The number of nodes increases 70.45%; the number of edges increases 66.42%. The transitivity decreases 93.43%, and the closed triads decrease 90.02%. In the random shuffle 10-way partitioning of the Sweden 50K graph, the nodes increase 83.87%, and the edges increase 74.18%. The largest partition has 10,291 nodes, and the smallest has 7,732. The transitivity decreases 87.60%, and the closed triads decrease 82.59% as presented in Table 6.11.

K^{th} partitioning suffers imbalances due to high total edge counts. The distribution of nodes in the Kth 4-way partitioning of the Sweden 5K graph is similar to the random shuffle with the largest partition having 2094 nodes and 6130 edges. The smallest partition has 1976 nodes and 5844 edges. The total node count

increases 62.68% from the original graph. The number of edges increases 55.38% overall. The transitivity drops 90.51%, and closed triads drop 89.18%.

Table 6.4 presents the properties for the Kth and fractional 10-way partitioning of the Sweden 5K graph. In the Kth 10-way partitioning of the Sweden 5K graph, the largest partition has 926 nodes and the smallest has 618. The number of nodes increases 54.20%, and the number of edges increases 32.71%. The transitivity drops 82.21%, and the number of closed triads drops 87.95% as presented in Table 6.5.

In the Kth 4-way partitioning of the Sweden 50K graph, the largest partition has 22058 nodes, and the smallest has 21142. The number of nodes increases 73.05%; the number of edges increases 68.46%. The transitivity decreases 93.56%, and the closed triads decrease 90.03%. Table 6.12 presents the properties for the Kth and fractional 10-way partitioning of the Sweden 50K graph. In the Kth and fractional 10-way partitioning of the Sweden 50K graph, the nodes increase 83.11%, and the edges increase 73.61%. The largest partition has 9833 nodes, and the smallest has 8571. The transitivity decreases 88.11%, and the closed triads decrease 83.41% as presented in Table 6.13.

Fractional cross-sectioning yields the best balance for all properties across partitions and the best fidelity to the clustering coefficient of the original graph as a whole of the partitioning methods analyzed. The distribution of nodes in the fractional 4-way partitioning of the Sweden 5K graph is even with one partition having 1154 nodes, two having 1156, and one having 1159. The total node count decreases 7.50% from the original graph where the node count increases in the other methods. The distribution of edges is also even with two partitions having 3590, one having 3592, and one having 3595. The number of edges decreases 6.08% overall. The transitivity does drop 75.86%, but this is the lowest decrease of all the methods. The closed triads drop 85.67%.

In the fractional 10-way partitioning of the Sweden 5K graph, the partitions range in node set size from 530 to 537. The partition edge set sizes range from 1658 to 1678. The number of nodes increases 6.72%, and the number of edges increases 8.86%. The transitivity drops 42.25%, and the number of closed triads drops 66.61%.

In the fractional 4-way partitioning of the Sweden 50K graph, two partitions have 12474 nodes, one has 12494, and one has 12496. The partition edge sizes range from 66034 to 66110. The number of nodes

decreases 0.12%; the number of edges increases 3.34%. The transitivity decreases 83.05%, and the closed triads decrease 83.02%. In the fractional 10-way partitioning of the Sweden 50K graph, the nodes increase 6.97%, and the edges increase 20.41%. The partition node set sizes range from 5251 to 5358. The edge set sizes range from 30759 to 30816. The transitivity decreases 42.91%, and the closed triads decrease 27.07%.

We choose the fractional method to partition the Borromean-S dk-2 distribution before any graphs are generated because it yields balanced partitions having relatively equal counts for both nodes and edges. In addition, it is the closest method to the original graph clustering coefficient overall.

In the third stage, dk-2 graphs are generated from the partitions using Borromean-S.

6.2 Stage 4: Stitching the Subgraphs

The final stage of the algorithm reconstitutes the full graph from the partitions. We take this opportunity to try to not to damage the transitivity and average path length of the resulting dk-2 graph further. The partition graphs are merged, or stitched, by swapping eligible edges across partitions. Each partition maintains a list of its eligible edges. Eligible edges must:

- Maintain the dK-2 distribution
- Not disconnect connected components
- Not be a part of an existing triad

Edges are swapped across partitions so long as the candidate for swapping is not in an existing triad and not a bridge between connected components. The objective is to preserve as much of the clustering and community structure as possible. Triads are identified by searching for common neighbors between two adjacent nodes. Bridge edges are detected by breath first search in a variant of Trajan's bridge finding algorithm [53]. Table 6.1 presents the bridge, non-triad, and candidate edge counts for the 4-partition merge of the labeled Sweden 5K graph specifically.

Table 6.1: Sweden 5K Labeled 4-Partition Merge. The 4-partition merge of the labeled Sweden 5K graph in Borromean-P-Python. The Bridges 1 column presents the number of bridge edges in the first partition subgraph of that merge phase, partition 1 for merge phase 1 & 2, partition 3 for phase 3 & 4, and the merge of 1 & 2 for the Full merge phase. The Bridges 2 column presents the number of bridge edges in the second partition subgraph of that merge phase, partition 2 for merge phase 1 & 2, partition 4 for phase 3 & 4, and the merge of 3 & 4 for the Full merge phase. Likewise, Non-Triad 1 and 2 present the number of non-triad edges. Cand 1 and 2 present the number of candidate edges. The APL column is the average path length of the partial result from that phase, and Time is the time taken in seconds.

	Bridges 1	Bridges 2	Non-Triad 1	Non-Triad 2	Cand 1	Cand 2	APL	Time (s)
1 & 2	325	325	3483	3475	3163	3156	4.73	7
3 & 4	325	325	3505	3473	3185	3152	4.73	8
Full	649	649	6955	7007	6307	6359	4.74	1140

Eligible edges are swapped from their current target node to a compatible node in a neighboring partition, creating a path that spans the partitions. Each partition attempts swapping with every other partition linearly, $O(n)$. This strategy also aims to match the average path length of the original graph.

When the average shortest path length of the merged graph is within a user specified, asymptotic range of the original path length as sampled from the original graph, stitching halts. Stitching may also halt when the population of eligible edges is exhausted.

Table 6.2: Sweden 5K Four Partition Comparison. Column CC presents the transitivity (global clustering coefficient) results. Row Frac presents results for fractional partitioning. Row C-S presents the cross-sectional results. Row R-S presents the random shuffle results.

		Node	**#Node** %Var	Edge	**#Edge** %Var	CC	**#CC** %Var	Avg CC	**#CC** %Var	Avg Path
Orig		5000	5000	15297	15297	0.093	0.093	0.268	0.268	4.59
dK-2		4926	-1.48	15060	-1.55	0.045	-51.68	0.010	-96.12	3.98
	P1	1154	**4625**	3592	**14367**	0.021	**0.0223**	0.025	**0.021**	3.58
	P2	1156		3590		0.024		0.021		3.59
	P3	1156	-7.50	3595	-6.08	0.022	-75.86	0.022	-92.03	3.59
Frac	P4	1159		3590		0.023		0.018		3.60
	P1	4602	**8493**	11846	**24747**	0.001	**0.008**	0.003	**0.011**	4.33
	P2	1337		5738		0.009		0.007		3.48
	P3	855	69.86	3281	61.78	0.014	-91.32	0.016	-95.98	3.40
C-S	P4	1699		3882		0.007		0.017		3.77
	P1	1882	**8095**	5607	**23166**	0.009	**0.0084**	0.014	**0.013**	3.84
	P2	2115		5906		0.008		0.009		4.01
	P3	2068	61.90	5842	51.44	0.009	-90.95	0.013	-95.34	4.07
R-S	P4	2030		5811		0.008		0.015		3.84
	P1	2094	**8134**	6130	**23768**	0.009	**0.0088**	0.015	**0.013**	3.86
	P2	1976		5844		0.009		0.015		3.85
	P3	2019	62.68	5915	55.38	0.009	-90.51	0.012	-95.02	3.82
K^{th}	P4	2045		5879		0.008		0.012		3.86

Table 6.3: Sweden 5K Four Partition Comparison - Triads. Row Frac presents results for fractional partitioning. Row C-S presents the cross-sectional results. Row R-S presents the random shuffle results.Column #C %Var presents the sum of closed triads across partitions and the percentage variation from the original.

		Node	**#Node** %Var	Edge	**#Edge** %Var	Open Triad	**#Open** %Var	Closed Triad	**#C** %Var
Orig		5000	5000	15297	15297	338006	338006	11485	11485
dK-2		4926	-1.48	15060	-1.55	3651188	980.21	547	-95.24
	P1	1154	**4625**	3592	**14367**	53991	**215711**	395	**1646**
	P2	1156		3590		53868		433	
	P3	1156	-7.50	3595	-6.08	54030	-36.18	401	-85.67
Frac	P4	1159		3590		53822		417	
	P1	4602	**8493**	11846	**24747**	237560	**447908**	85	**778**
	P2	1337		5738		70116		231	
	P3	855	69.86	3281	61.78	70116	32.51	231	-93.23
C-S	P4	1699		3882		70116		231	
	P1	1882	**8095**	5607	**23166**	94227	**401167**	276	**1132**
	P2	2115		5906		81745		231	
	P3	2068	61.90	5842	51.44	114677	18.69	339	-90.14
R-S	P4	2030		5811		110518		286	
	P1	2094	**8134**	6130	**23768**	112414	**418900**	329	**1243**
	P2	1976		5844		102216		330	
	P3	2019	62.68	5915	55.38	104723	23.93	309	-89.18
K^{th}	P4	2045		5879		99547		275	

Table 6.4: Sweden 5K Ten Partition Comparison - Fractional and K^{th}. Column CC presents the transitivity (global clustering coefficient) results. Row Frac presents results for fractional partitioning.

		Node	**#N** %Var	Edge	**#E** %Var	CC	**#CC** %Var	Avg CC	**#CC** %Var	Avg Path
Orig		5000	5000	15297	15297	0.092	0.092	0.268	0.268	4.59
dK-2		4926	-1.48	15060	-1.55	0.044	-51.68	0.010	-96.12	3.98
	P1	530	**5336**	1660	**16652**	0.050	**0.053**	0.037	**0.038**	3.42
	P2	537		1665		0.054		0.033		3.47
	P3	531		1678		0.052		0.047		3.43
	P4	537		1662		0.053		0.034		3.46
	P5	532		1666		0.055		0.034		3.44
	P6	535	6.72	1669	8.86	0.056	-42.25	0.035	-85.92	3.42
	P7	535		1668		0.048		0.033		3.43
	P8	532		1666		0.051		0.040		3.41
	P9	532		1660		0.056		0.047		3.44
Frac	P10	535		1658		0.055		0.035		3.45
	P1	837	**7710**	2209	**20300**	0.025	**0.016**	0.029	**0.018**	3.89
	P2	695		1907		0.001		0.012		3.84
	P3	618		1826		0.022		0.024		3.76
	P4	665		1850		0.010		0.017		3.79
	P5	903		2106		0.013		0.011		4.19
	P6	767	54.20	1981	32.71	0.028	-82.21	0.024	-93.31	3.97
	P7	926		2274		0.015		0.012		3.92
	P8	761		2099		0.022		0.019		3.89
	P9	688		1900		0.018		0.021		3.90
K^{th}	P10	850		2148		0.008		0.005		4.01

Table 6.5: Sweden 5K Ten Partition Comparison - Fractional and Kth- Triads. Row Frac presents results for fractional partitioning.

		Node	**#N** %Var	Edge	**#E** %Var	Open Triad	**#Open** %Var	Closed Triad	**#Closed** %Var
Orig		5000	5000	15297	15297	338006	338006	11485	11485
dk-2		4926	-1.48	15060	-1.55	3651188	980.21	547	-95.24
	P1	530	**5336**	1660	**16652**	20351	**203662**	363	**3835**
	P2	537		1665		20284		388	
	P3	531		1678		20632		382	
	P4	537		1662		20494		388	
	P5	532		1666		20351		401	
	P6	535	6.72	1669	8.86	20339	-39.75	401	-66.61
	P7	535		1668		20425		346	
	P8	532		1666		20458		368	
	P9	532		1660		20328		405	
Frac	P10	535		1658		20000		393	
	P1	837	**7710**	2209	**20300**	28352	**230385**	239	**1384**
	P2	695		1907		19187		81	
	P3	618		1826		19645		151	
	P4	665		1850		19427		66	
	P5	903		2106		24141		113	
	P6	767	54.20	1981	32.71	21569	-31.84	210	-87.95
	P7	926		2274		29727		158	
	P8	761		2099		22780		171	
	P9	688		1900		19883		123	
Kth	P10	850		2148		25674		72	

Table 6.6: Sweden 5K Ten Partition Comparison - Cross-Sectional and Random Shuffle. Column CC presents the transitivity (global clustering coefficient results).

		Node	**#N** %Var	Edge	**#E** %Var	CC	**#CC** %Var	Avg CC	**#CC** %Var	Avg Path
Orig		5000	5000	15297	15297	0.093	0.093	0.268	0.268	4.59
dK-2		4926	-1.48	15060	-1.55	0.044	-51.68	0.010	-96.12	3.98
	P1	3384	**9015**	5957	**24623**	2E-4	**0.009**	0.001	**0.0133**	4.98
	P2	1296		4273		0.003		0.005		3.87
	P3	741		3007		0.007		0.008		3.37
	P4	644		2736		0.010		0.008		3.26
	P5	453		2034		0.017		0.018		3.15
	P6	378	80.30	1595	60.97	0.015	-90.61	0.021	-95.03	3.12
	P7	300		1024		0.016		0.027		3.15
	P8	393		976		0.009		0.008		3.41
	P9	543		1207		0.004		0.007		3.44
C-S	P10	883		1814		0.007		0.031		3.50
	P1	1092	**8123**	2521	**21173**	0.009	**0.018**	0.013	**0.022**	4.50
	P2	857		2216		0.022		0.037		3.97
	P3	593		1874		0.021		0.027		3.54
	P4	655		1834		0.016		0.030		3.53
	P5	737		2049		0.015		0.016		3.77
	P6	837	62.46	1796	38.41	0.023	-80.36	0.022	-91.86	4.23
	P7	825		2196		0.023		0.011		3.98
	P8	672		1799		0.026		0.036		3.79
	P9	1004		2601		0.012		0.012		4.05
R-S	P10	851		2287		0.016		0.014		3.95

6.3 Performance of Borromean-P-Python

The partitioning experiments show that fractional partitioning provides the most even distribution of graph properties across partitions. Table 6.1 shows the statistics for the labeled 4 partition merge on the Sweden 5K subgraph. The data suggests we have come close to the average path length of the original graph, but also that it is computationally intensive to do so in Python even on small graphs. We calculate the APL at each swap step and use the original as a stop condition. We include a 0.25 tolerance in case the APL does not converge.

Table 6.7: Sweden 5K Ten Partition Comparison - Cross-Sectional and Random Shuffle - Triads

		Node	**#N** %Var	Edge	**#E** %Var	Open Triad	**#Open** %Var	Closed Triad	**#Closed** %Var
Orig		5000	5000	15297	15297	338006	338006	11485	11485
dK-2		4926	-1.48	15060	-1.55	3651188	980.21	547	-95.24
	P1	3384	**9015**	5957	**24623**	110593	**465879**	9	**1062**
	P2	1296		4273		39853		41	
	P3	741		3007		31389		77	
	P4	644		2736		27160		152	
	P5	453		2034		26160		152	
	P6	378	80.30	1595	60.97	25827	-37.83	131	-90.75
	P7	300		1024		19370		102	
	P8	393		976		24030		71	
	P9	543		1207		41663		58	
C-S	P10	883		1814		119834		269	
	P1	1092	**8123**	2521	**21173**	36779	**269978**	107	**1608**
	P2	857		2216		29779		220	
	P3	593		1874		23851		173	
	P4	655		1834		30994		164	
	P5	737		2049		22405		114	
	P6	837	62.46	1796	38.41	22004	-20.13	179	-86.00
	P7	825		2196		25168		197	
	P8	672		1799		20436		184	
	P9	1004		2601		29435		117	
R-S	P10	851		2287		28127		153	

Table 6.8: Sweden 50K Four Partition Comparison. Column CC presents the transitivity (global clustering coefficient) results. Row O presents results for the original graph. Row D presents results for the original graph. Row F presents results for fractional partitioning. Row C presents the cross-sectional results. Row R presents the random shuffle results. Row K presents the Kth partitioning results.

<table>
<tr><th></th><th></th><th>Node</th><th>#N
%Var</th><th>Edge</th><th>#E
%Var</th><th>CC</th><th>#CC
%Var</th><th>Avg
CC</th><th>#CC
%Var</th><th>Avg
Path</th></tr>
<tr><td>O</td><td></td><td>50000</td><td>50000</td><td>255750</td><td>255750</td><td>0.074</td><td>0.074</td><td>0.191</td><td>0.191</td><td>4.49</td></tr>
<tr><td>D</td><td></td><td>49546</td><td>-0.91</td><td>253739</td><td>-0.79</td><td>0.002</td><td>-96.77</td><td>0.002</td><td>-98.80</td><td>4.08</td></tr>
<tr><td rowspan="4">C</td><td>P1</td><td>48488</td><td rowspan="2">73323</td><td>223115</td><td rowspan="2">378137</td><td>6E-4</td><td rowspan="2">0.008</td><td>0.001</td><td rowspan="2">0.009</td><td>4.30</td></tr>
<tr><td>P2</td><td>12364</td><td>86065</td><td>0.005</td><td>0.004</td><td>3.67</td></tr>
<tr><td>P3</td><td>7222</td><td rowspan="2">46.65</td><td>43718</td><td rowspan="2">47.85</td><td>0.009</td><td rowspan="2">-89.35</td><td>0.009</td><td rowspan="2">-95.35</td><td>3.58</td></tr>
<tr><td>P4</td><td>5249</td><td>25239</td><td>0.017</td><td>0.021</td><td>3.56</td></tr>
<tr><td rowspan="4">R</td><td>P1</td><td>19980</td><td rowspan="2">85224</td><td>103355</td><td rowspan="2">425627</td><td>0.005</td><td rowspan="2">0.005</td><td>0.006</td><td rowspan="2">0.005</td><td>3.87</td></tr>
<tr><td>P2</td><td>20505</td><td>103568</td><td>0.005</td><td>0.005</td><td>3.90</td></tr>
<tr><td>P3</td><td>22127</td><td rowspan="2">70.45</td><td>110525</td><td rowspan="2">66.42</td><td>0.005</td><td rowspan="2">-93.43</td><td>0.005</td><td rowspan="2">-97.55</td><td>3.94</td></tr>
<tr><td>P4</td><td>22612</td><td>108179</td><td>0.005</td><td>0.004</td><td>3.99</td></tr>
<tr><td rowspan="4">K</td><td>P1</td><td>21367</td><td rowspan="2">86525</td><td>107706</td><td rowspan="2">430824</td><td>0.005</td><td rowspan="2">0.005</td><td>0.005</td><td rowspan="2">0.004</td><td>3.91</td></tr>
<tr><td>P2</td><td>21142</td><td>107335</td><td>0.005</td><td>0.004</td><td>3.91</td></tr>
<tr><td>P3</td><td>22058</td><td rowspan="2">73.05</td><td>107842</td><td rowspan="2">68.46</td><td>0.005</td><td rowspan="2">-93.56</td><td>0.004</td><td rowspan="2">-97.82</td><td>3.96</td></tr>
<tr><td>P4</td><td>21958</td><td>107941</td><td>0.005</td><td>0.004</td><td>3.94</td></tr>
<tr><td rowspan="4">F</td><td>P1</td><td>12474</td><td rowspan="2">49938</td><td>66074</td><td rowspan="2">264297</td><td>0.013</td><td rowspan="2">0.013</td><td>0.0084</td><td rowspan="2">0.009</td><td>3.71</td></tr>
<tr><td>P2</td><td>12494</td><td>66079</td><td>0.013</td><td>0.009</td><td>3.71</td></tr>
<tr><td>P3</td><td>12474</td><td rowspan="2">-0.12</td><td>66034</td><td rowspan="2">3.34</td><td>0.012</td><td rowspan="2">-83.05</td><td>0.008</td><td rowspan="2">-95.52</td><td>3.71</td></tr>
<tr><td>P4</td><td>12496</td><td>66110</td><td>0.013</td><td>0.009</td><td>3.71</td></tr>
</table>

Table 6.9: Sweden 50K Four Partition Comparison - Triads. Row O presents results for the original graph. Row D presents results for the original graph. Row F presents results for fractional partitioning. Row C presents the cross-sectional results. Row R presents the random shuffle results. Row K presents the Kth partitioning results.

		Node	**#N** %Var	Edge	**#E** %Var	Open Triad	**#Open** %Var	Closed Triad	**#C** %Var
O		50000	50000	255750	255750	8.9E6	8.9E6	239683	239683
D		49546	-0.91	253739	-0.79	9590898	7.05	8000	-96.66
	P1	48488	**73323**	223115	**378137**	6882858	**1.5E7**	1597	**28403**
	P2	12364		86065		3335588		5662	
	P3	7222	46.65	43718	47.85	2584291	68.52	8216	-88.15
C	P4	5249		25239		2293403		12928	
	P1	19980	**85224**	103355	**425627**	3641708	**1.4E7**	5824	**23921**
	P2	20505		103568		3564832		6227	
	P3	22127	70.45	110525	66.42	3696939	61.65	5758	-90.02
R	P4	22612		108179		3578671		6082	
	P1	21367	**86525**	107706	**430824**	3712965	**1.4E7**	6215	**23888**
	P2	21142		107335		3652384		5975	
	P3	22058	73.05	107842	68.46	3655230	63.88	5986	-90.03
K	P4	21958		107941		3661825		5712	
	P1	12474	**49938**	66074	**264297**	2382570	**9.5E6**	10157	**40690**
	P2	12494		66079		2385262		10377	
	P3	12474	-0.12	66034	3.34	2381411	6.42	10019	-83.02
F	P4	12496		66110		2384903		10137	

Table 6.10: Sweden 50K Ten Partition Comparison - Cross-Sectional and Random Shuffle. Column CC presents the transitivity (global clustering coefficient results. Row O presents results for the original graph. Row D presents results for the original graph. Row C presents the cross-sectional results. Row R presents the random shuffle results.

		Node	**#N** %Var	Edge	**#E** %Var	CC	**#CC** %Var	Avg CC	**#CC** %Var	Avg Path
O		50000	50000	255750	255750	0.07	0.07	0.191	0.191	4.49
D		49546	-0.91	253739	-0.79	0.002	-96.77	0.002	-98.80	4.08
	P1	43304	**86599**	151489	**431199**	2E-4	**8E-4**	5E-4	**0.009**	4.67
	P2	13833		93534		0.001		0.001		3.79
	P3	7917		58303		0.003		0.002		3.58
	P4	5363		38214		0.006		0.005		3.54
	P5	3718		24755		0.009		0.006		3.50
	P6	3050	73.20	19374	68.60	0.010	-89.70	0.009	-94.98	3.48
	P7	3079		17484		0.011		0.010		3.51
	P8	2304		11152		0.009		0.013		3.50
	P9	1867		8954		0.013		0.020		3.40
C	P10	2164		7940		0.015		0.029		3.46
	P1	10291	**91934**	46349	**445455**	0.009	**0.009**	0.006	**0.008**	3.93
	P2	10041		46974		0.009		0.006		3.85
	P3	9366		44138		0.009		0.008		3.83
	P4	8203		42428		0.010		0.010		3.68
	P5	9791		46360		0.008		0.008		3.79
	P6	8551	83.87	43807	74.18	0.009	-87.60	0.009	-95.84	3.71
	P7	9559		46934		0.009		0.007		3.80
	P8	7732		38696		0.010		0.009		3.75
	P9	9051		45888		0.009		0.008		3.74
R	P10	9349		43881		0.009		0.008		3.81

Table 6.11: Sweden 50K Ten Partition Comparison - Cross-Sectional and Random Shuffle - Triads. Row O presents results for the original graph. Row D presents results for the original graph. Row C presents the cross-sectional results. Row R presents the random shuffle results.

		Node	**#N** %Var	Edge	**#E** %Var	Open Triad	**#Open** %Var	Closed Triad	**#C** %Var
O		50000	50000	255750	255750	8959048	8.9E6	239683	239683
D		49546	-0.91	253739	-0.79	9590898	7.05	8000	-96.66
	P1	43304	**86599**	151489	**431199**	3825159	**1.4E7**	215	**24627**
	P2	13833		93534		2339333		1111	
	P3	7917		58303		1748417		1796	
	P4	5363		38214		1391885		2704	
	P5	3718		24755		1059651		3058	
	P6	3050	73.20	19374	68.60	964982	61.42	3307	-89.73
	P7	3079		17484		1022799		3635	
	P8	2304		11152		728625		2164	
	P9	1867		8954		668090		2945	
C	P10	2164		7940		712642		3692	
	P1	10291	**91934**	46349	**445455**	1271233	**1.3E7**	3889	**41733**
	P2	10041		46974		1370389		4071	
	P3	9366		44138		1300450		4181	
	P4	8203		42428		1343902		4586	
	P5	9791		46360		1388127		3687	
	P6	8551	83.87	43807	74.18	1393252	49.78	4226	-82.59
	P7	9559		46934		1375810		4201	
	P8	7732		38696		1251909		4385	
	P9	9051		45888		1360210		4115	
R	P10	9349		43881		1363835		4392	

Table 6.12: Sweden 50K Ten Partition Comparison - Kth and Fractional. Column CC presents the transitivity (global clustering coefficient) results. Row O presents results for the original graph. Row D presents results for the original graph. Row F presents results for fractional partitioning. Row K presents the Kth partitioning results.

		Node	**#N** %Var	Edge	**#E** %Var	CC	**#CC** %Var	Avg CC	**#CC** %Var	Avg Path
O		50000	50000	255750	255750	0.074	0.074	0.191	0.191	4.49
D		49546	-0.91	253739	-0.79	0.002	-96.77	0.002	-98.80	4.08
	P1	9172	**91554**	44583	**444003**	0.011	**0.009**	0.011	**0.008**	3.78
	P2	8965		44264		0.009		0.007		3.82
	P3	9232		46098		0.008		0.008		3.77
	P4	8571		43658		0.009		0.009		3.73
	P5	8894		42882		0.009		0.007		3.79
	P6	8892	83.11	43597	73.61	0.008	-88.11	0.006	-95.90	3.78
	P7	9833		44238		0.008		0.008		3.89
	P8	9538		44866		0.006		0.005		3.81
	P9	9285		45201		0.009		0.007		3.80
K	P10	9172		44616		0.011		0.011		3.77
	P1	5251	**53487**	30799	**307949**	0.042	**0.042**	0.020	**0.019**	3.47
	P2	5344		30779		0.042		0.021		3.47
	P3	5341		30783		0.043		0.021		3.47
	P4	5342		30809		0.043		0.019		3.47
	P5	5351		30815		0.042		0.021		3.47
	P6	5348	6.97	30798	20.41	0.042	-42.91	0.019	-89.60	3.47
	P7	5355		30816		0.043		0.019		3.47
	P8	5358		30804		0.042		0.019		3.47
	P9	5353		30759		0.042		0.021		3.47
F	P10	5344		30787		0.043		0.019		3.47

Table 6.13: Sweden 50K Ten Partition Comparison - Kth and Fractional - Triads. Row O presents results for the original graph. Row D presents results for the original graph. Row F presents results for fractional partitioning. Row K presents the Kth partitioning results.

		Node	**#N** %Var	Edge	**#E** %Var	Open Triad	**#Open** %Var	Closed Triad	**#C** %Var
O		50000	50000	255750	255750	8959048	8.9E6	239683	239683
D		49546	-0.91	253739	-0.79	9590898	7.05	8000	-96.66
	P1	9172	**91554**	44583	**444003**	1340538	**1.3E7**	4744	**39775**
	P2	8965		44264		1293380		4066	
	P3	9232		46098		1364192		3869	
	P4	8571		43658		1313348		4306	
	P5	8894		42882		1284151		4128	
	P6	8892	83.11	43597	73.61	1307456	48.80	3410	-83.41
	P7	9833		44238		1364498		3668	
	P8	9538		44866		1355560		2851	
	P9	9285		45201		1364111		3933	
K	P10	9172		44616		1344127		4800	
	P1	5251	**53487**	30799	**307949**	1183423	**1.1E7**	17367	**174792**
	P2	5344		30779		1182236		17309	
	P3	5341		30783		1182120		17568	
	P4	5342		30809		1184339		17621	
	P5	5351		30815		1185844		17511	
	P6	5348	6.97	30798	20.41	1184548	32.15	17317	-27.07
	P7	5355		30816		1186294		17659	
	P8	5358		30804		1184229		17449	
	P9	5353		30759		1181981		17307	
F	P10	5344		30787		1183933		17684	

Table 6.14: Properties of Labeled Graphs Generated with Borromean-P Python

Data Set	Parts	Nodes	Edges	Transitivity	APL
Polblogs	4	1452	16903	0.22	3.01
Polblogs	10	3500	40187	0.27	3.40
FB-C	4	1260	22883	0.23	2.76
FB-C	2	758	14879	0.29	2.47
FB-D	2	5062	203550	0.08	3.01
FB-D	4	7192	264909	0.12	3.75
FB-M	4	25700	1025535	0.06	6.46

Table 6.15: Borromean-P Python Node Attribute Percentages

Data Set	Parts	Label A %	Label B %	A-A %	B-B %	A-B %
Polblogs	4	73	27	40	47	13
FB-C	2	74	25	55	7	38
FB-D	4	70	29	41	14	45
FB-M	4	64	36	42	12	45

Table 6.16: Borromean-P Python Performance (Time). Column t presents the total time taken in minutes. Column DK presents the time taken to generate the DK-2 distribution in seconds. Column Part presents the time taken to fractionally partition the distribution. Column S-G presents the time taken to generate a subgraph in seconds. Subgraphs may be generated in parallel. Column 1st M presents the time taken for the first subgraph merge (partitions 1&2) in seconds. The following columns likewise present the time taken for the second merge in seconds and the full merge (partitions 1&2 merged with 3&4) in minutes.

Data	Parts	t (m)	DK (s)	Part (s)	S-G (s)	1st M (s)	2nd M (s)	Full Merge (m)
Pol	4	5	0.2	0.3	0.8	25	19	4
Pol	10	91	0.2	0.5	0.8	24	40	72
FB-C	4	5	40	0.4	1	31	22	4
FB-C	2	1	0.35	1	1.4	NA	NA	1
FB-D	2	44	6	4	83	NA	NA	41
FB-D	4	386	3.4	3.2	36	480	480	367
FB-M	4	2460	10.9	5.8	240	12660	12660	2010

As designed, edge swap candidates are discerned using Trajan's algorithm to find bridge edges and a triad census to distinguish those edges that have not already formed triangles. It takes over 5 hours to complete the full merge on a commodity laptop.

Table 6.14 shows the network properties of our sequentially generated merged graphs. For small graphs such as Polblogs and Facebook Caltech, a larger number of partitions can cause the fractional partitioning method to over-count nodes and edges. In the joint-degree distribution, the fractional method divides the total number of edges of each degree/label pairing by the number of partitions. However, the resulting number of edges is rounded up such that if it is a fraction less than one, it will be rounded to 1. Projecting this across a large number of partitions creates additional edges not present in the original graph. In contrast, the Facebook Dartmouth network shows greater fidelity at 4 partitions rather than 2.

The Facebook Michigan graph stalled while attempting to converge to its original path length. We had to set a high APL target of 6.46 to complete the merge. The 4 partition Dartmouth merge also did not converge within the tolerance, showing a 12% increase in APL.

Table 6.15 shows the labeled attribute node and edge percentages in the sequentially generated merge graphs. FB-Caltech and FB-Dartmouth have close to the same node distribution, while FB-Michigan and Polblogs exhibit a loss in label A nodes and a gain in label B nodes. The general trend in edge distribution is a loss in label A-A and a gain in mixed egdes.

Table 6.16 gives the timing performance of the sequential merges. Time taken grows sharply with FB-Michigan as it searches for shorter path lengths. As the path length of the FB-Michigan graph reaches 6.75, the change in path length caused by each swap becomes minuscule. Sequential performance statistics for the Amazon graph and the full Steam Sweden graph are not listed due to the extended time (days) required.

CHAPTER 7: BORROMEAN-P SPARK: PARALLEL DESIGN AND IMPLEMENTATION

In Chapter 6, we proposed the design of Borromean-P-Python, a partitioned swap-merge node attribute graph generation algorithm and a sequential implementation in Python. However, Python has significant overhead due to its interpreter that makes it computationally unsuitable for very large workloads. Implementation in C/C++ would ostensibly be faster, but the C/C++ language is not optimized for large graph workloads and parallel clusters. In an effort to optimize performance and enable generation of larger graphs, we propose the design and implementation of a parallel algorithm, Borromean-P-Spark, in Apache Spark [14]. This chapter presents Borromean-P-Spark and details our design, implementation, and its performance.

In addition, we introduce the the architecture of Spark. Adaptations for Spark cause Borromean-P-Spark to differ from Borromean-P-Python in the stitching phase. Chiefly, bridge edges are not excluded and the average path length is not used as a stop condition. Section 7.5 details the differences in design.

7.1 Spark

Spark is a distributed data processing and analytics engine framework that most commonly functions as a part of the Hadoop map reduce ecosystem, running on top of an Hadoop cluster with the YARN scheduler and drawing data from the Hadoop distributed file system (HDFS). Spark is similar to Map-Reduce, but features resilient distributed data stores (RDDs) [14]. RDDs can remain in memory between operations and reuse data, thus allowing a more flexible and efficient execution pipeline for iterative algorithms that perform the same task on many different partitions of a data set.

The Spark task scheduler constructs a directed acyclic graph (DAG) of RDD lineage for each job. In the event of an incident, all the RDDs can be reconstructed from the DAG lineage. The lineage also enables new RDDs to be constructed from existing RDD structures. The DAG traces the complete sequence of RDD transformations from input to final output [54].

There are two major types of RDD based data transformations within Spark, narrow and wide. Narrow transformations such as *map* and *filter* have a one-to-one dependency between partitions of the parent RDD in the DAG and the child RDD. All data required to perform the transformation of a partition resides within that partition. Wide transformations such as *join*, *sort*, and *group*, however, have a one-to-many, indeterminate dependency between parent and child partitions.

Sequences of narrow transformations can be completed in a single stage, while wide transformations trigger new stages. The movement of data across partitions and stage boundaries in wide transformations is carried out by shuffle operations. Shuffle operations can cause an excess of communication between worker nodes and between worker nodes and the application driver possibly resulting in processing delays and memory overruns. It is best practice to avoid wide transformations whenever the needs of the application permit it. They are difficult to avoid in our case since we must implement mergers and calculate metrics that traverse entire graphs.

The transformations of an RDD are not actually implemented until an *action* is called on the structure. Actions are output or data transfer operations that return variables or structures other than RDDs. For example, *collect* copies data from the worker nodes to the application driver as an array where it can be sequentially processed by other methods. *Reduce* and *saveAsText* operations are also actions.

Each job is comprised of a number of stages, and each stage has one task for each partition in the stage. The number of stages is equal to the shuffle operations needed to accommodate the job's wide transformations. The partitions in each of a job's stages are distributed across the worker nodes in a cluster to Java virtual machines called executors. Multiple executors reside in the RAM of the worker nodes [55]. Users can specify the number and memory size of the executors in a cluster at compile time as well as the number of cores the executor will use.

The memory an executor will use is somewhat deceptive. An executor's memory is not fully dedicated to data storage and execution. By default, at least 384 MB is used for internal overhead expanding to 10% as executor sizes increase, and more will be set aside if any data has been cached. In addition, memory should be set aside for external Java overhead as detailed in Section 7.3. The additional Java overhead is not considered when Spark configures the executor memory threshold. So, the application driver can

completely fill an executor leaving no room for garbage collection or other maintenance functions if the memory size is too small. The executor consequently fails. These considerations factor into our design decisions, particularly in how jobs and tasks should be executed.

7.2 GraphX

The GraphX [56] graph processing module of the Spark framework represents graphs as abstractions of vertex and edge RDDs. The vertex RDD contains a parallelized data structure of all the vertices in (id, attribute) pairs. The vertex id is typically of type long and the attribute may be any type or even another data structure. For example, the cheater and non-cheater labels in the Steam graph would be set as vertex attributes. The edge RDD contains a parallelized data structure of all the edges. It is typically specified just with its edge attribute type, but in reality it is a triplet with access to source and destination vertex information as well as the edge attribute.

Graphs in GraphX are partitioned by random vertex cuts, distributing the partitions, including copies of the cut vertex, across a set of executors that reside on the worker nodes within the cluster. Each partition is contained in an RDD within the executor. The executor manipulates the RDD as a logical graph construct.

The GraphX API implements several basic graph transformations and actions, including triad census, page rank, connected components, and neighbor collection. For example, average path length is processed in a bulk synchronous parallel manner similar to the Pregel [27] framework. Initial versions of the algorithm calculated the average shortest path length of the partitions at the end of each super step, but given the number of super steps in large graphs, this approach proved computationally prohibitive.

The average path calculation can also be processed incrementally. GraphMod [57, 58] is a incremental version of the Bellman-Ford shortest path algorithm [59, 60, 61, 62] for Spark based on GraphInc [63] and Pregel. After the first pass of the graph, it is designed to only calculate changed edges in a vertex-centric manner sending update messages to the affected vertices. Graphmod can be slower than the traditional Pregel APL since it must process the entire graph on the first pass. The traditional Pregel implementation can be seeded with only a small sample of the graph. Therefore, we chose the traditional Pregel implementation for APL calculation.

7.3 Tuning and Optimization

Spark processes tasks in-memory. Therefore, the number of executors on each node is determined by the amount of RAM and processor cores available. Each executor requires 7% overhead above its assigned memory. Too much or too little memory allotted for an executor can result in excessive garbage collection that will eventually cause the executor to fail, especially with very large datasets requiring many inter-partition shuffle tasks. By default, if Spark loses 4 executors, it will terminate the processing job. Assigning too many cores to an executor can overwhelm the throughput of I/O operations with the datastore.

For example, Ryza et al. [64] recommend a maximum of 64 GB and 5 cores per executor. In practice, the excessive garbage collection issue also occurs at much lower memory thresholds. Partitions of large datasets can grow beyond the assigned memory of an executor, and the garbage collector has difficulty flushing them. This growth is due to large hash maps constructed during inter-partition data shuffle operations. A solution is to increase the task parallelism, thereby increasing the level of partitioning, and consequently decreasing the data size of individual tasks sent to each executor [65]. Parallelism should be set for 2-3 tasks per core in the executor. Note, however, that parallelism is a cluster-wide value, so it is not set for each executor. The number of executors is also set cluster-wide.

7.4 Cluster Configuration

A portion of our Spark experiments were conducted using the XSEDE supercomputing environment [66], specifically the Wrangler data analysis and storage system [67]. Wrangler has 128 GB RAM per worker node and 24 cores. To calculate the optimal configuration parameters for Wrangler Spark jobs, we reserved at least 1 GB and 1 core for system functions. We then reserved 1 executor to act as the *Driver*, and divide the 23 remaining cores by the number of executors per node. We use 4 executors per node resulting in 5.75 cores per executor, but we round this down to 5. The value of 4 executors per node is arbitrary and can be tweaked if need be.

If there are 4 executors per node, then there will be $127/4 = 31.75$ GB memory per executor. We then allot 7% overhead per executor, giving $31.75 * 0.07 = 2.22$ GB. We subtract the overhead from the total such that we have 29 GB per executor working memory.

In Spark, the level of parallelism controls the size of tasks per executor. There should be 2-3 tasks per working core in the cluster. Since we are aiming at a 10 worker node cluster, the parallelism should be $(9*4+3)*5*3 = 585$. We have 9 nodes with 4 executors each, plus an additional node with 3 executors not 4, as one is reserved for the *Driver*. We then multiply by 5 cores per executor and by 3 tasks per core.

It should be noted that this configuration is only a general guideline for most workloads [64]. In our experiments, Wrangler continued to lose executors during runs with several million vertices or more.

We also used a smaller cluster of our own with 10 nodes each with 32 GB RAM. In this configuration, we used 29 executors total with 9 GB of working memory and 1 core per executor. The level of parallelism is set to 87.

7.5 Adapting the Parallel Merge Algorithm for Spark

The parallel version of the Borromean algorithm in Spark is similar to the sequential version; however, several adaptations were made to accommodate the design of Spark. Several transformations of the data are required before the dk-2 partitions can be merged.

First, the labeled partition edge lists cannot be fed directly into the built-in graph loader. The built-in edge list method does not support labeled vertex attributes, so separate vertex and edge RDDs must be constructed from the tuples in the partition.

Also, the vertex *ids* across the two graphs to be merged must be set uniquely. Since the partition graphs are generated independently, they have many of the same *ids*. GraphX will consider the same *id* to be the same vertex and under-count the total number of nodes for the merged graph. To overcome this limitation, we assigned nodes from each graph a unique prefix.

We merge the fractional partitions of the dk-2 graphs by swapping non-triad edges between graphs that obey both the joint degree and labeled attribute distributions. As we match on degree, we must join the degree of each vertex to the vertex RDDs of the graphs.

To identify non-triad edges, an adjacency list is formed by joining the neighbors of each vertex to the graph with an *outer join* operation. Then each neighbor pairing is tested to see if they have a common

neighbor. Each triad edge is labeled. Spark has a built-in function for triangle counts, but it does not list individual edges. We do not distinguish bridge edges here because it is difficult to implement DFS due to the Spark DAG execution planning structure.

The non-triad edge set is filtered into its own RDD. A sample is taken of this RDD to reduce the swap/merge time while still reducing the average path length. The sampled edges are subtracted from the non-triad set via a set operation. The EdgeRDD must be converted to an RDD of tuples before the subtraction because of a bug in the GraphX EdgeRDD implementation of the subtract method [68].

The sampled edges from both graphs must be indexed. The indexed sets are then joined with a cartesian operation that pairs each edge with every other edge. A cartesian join can be computationally intensive for very large graphs, but it allows Spark to ensure a matching swap can be found between edges that share a degree and an attribute and that the matching search is exhaustive.

An edge swap occurs between partitions if both the joint degree and labeled attributes match. Only these swapped edges are added to a new RDD, eliminating their previous counterparts. The newly swapped RDD is added back to the rest of the edges that were not swapped to form the new merged graph.

7.6 Performance of the Parallel Borromean

As described above, we propose a parallel implementation of the Borromean algorithm. However, in initial iterations its edge matching search incurred high memory overhead on the application drive, enough to trigger a garbage collection fault. If the available memory overhead of a Spark job is too small, a garbage collection fault will result from the use of the sequential collect method to increment through the non-triad edges. The memory space taken by the edges collected on the driver node of the Spark cluster exceeds the available overhead on the node.

In addition to increasing the memory overhead itself, the high requirements of the search can be lessened by sampling the non-triad edge set then using the sample in the swapping operation. It is also important to design each phase of the merging algorithm using Spark's parallel mapping API, as Spark also supports additional API functions that are sequential and more resource intensive. The same operation can often be

completed with both sequential and parallel functions. The parallel approach is normally the most indirect. Thus, it can be difficult to discern a flow of logic that is entirely executed in parallel.

Unlike the sequential implementation, the parallel merge algorithm does not swap until the original APL is reached, rather swapping all candidate non-triad edges. RDD operations are designed by default to act on all the data within the structure at once.

The properties of the graphs we generated with the labeled parallel edge-swapping merge algorithm are given in Table 7.1. All graphs experience a loss in nodes and edges, but FB-Caltech is closest to its original graph, seeing only a slight deterioration in all properties. FB-Dartmouth diverges from its original graph in effective diameter, being 49% larger. The Steam Sweden graph loses 16% of its nodes and 16% of its edges. The clustering coefficient is two orders of magnitude lower, effectively zero. The Amazon products graph shows similar losses.

The labeled attribute node and edge percentages for the graphs generated with the labeled parallel merge are shown in Table 7.2. All graphs have a loss of type A-A edges, slightly more than the sequential merge graphs. FB-Caltech is closest to its original values.

The timing for each labeled merge stage is listed in Table 7.3. The full Steam Sweden graph completes a 4 partition merge in under 15 minutes. The labeled merge times reflect runs on our in-house cluster.

Table 7.1: Properties of Borromean-P-Spark Graphs Generated in Parallel

Data Set	Parts	Nodes	Edges	CC	Eff. Diam
Polblogs	4	1452	16904	0.25	2.76
FB-C	2	758	14878	0.20	2.41
FB-D	4	7192	264898	0.12	4.91
FB-M	4	25700	1025497	0.06	4.93
Amazon	4	240869	682921	0.0004	9.28
Steam SE	4	646392	1721444	0.0012	7.83

Table 7.2: Borromean-P-Spark Node Attribute Percentages

Data Set	Parts	Label A %	Label B %	A-A %	B-B %	A-B %
Polblogs	4	69	31	32	47	22
FB-C	2	68	31	45	11	45
FB-D	4	67	33	43	12	45
FB-M	4	64	36	42	12	45
Amazon	4	72	28	47	9	44
Steam SE	4	87	13	50	8	42

Table 7.3: Borromean-P Spark - Parallel Performance (Time)

Dataset	Parts	1st Merge (s)	2nd Merge (s)	Final Merge (s)
Polblogs	4	52	52	52
FB - C	2	NA	NA	51
FB - D	4	61	61	70
FB-M	4	98	96	220
Amazon	4	86	86	130
Steam SE	4	161	153	585

CHAPTER 8: DISCUSSIONS AND FUTURE WORK

We proposed an algorithm, Borromean, to generate labeled graphs with binary node attributes that follow a specific attribute-based joint-degree distribution. We used the dK-2 model because it provides fidelity to the degree-based properties of the original input graph.

We have implemented a sequential implementation, Borromean-S, in C++. The resulting randomly generated graphs retain not only the degree based characteristics preserved by the standard dk-2 series, but also the labeled node attributes useful for research and analysis.

The inclusion of node attributes helps to better capture the structure of social graphs. These factors should facilitate the sharing of scientifically meaningful graph datasets, particularly those of large social networks with an abundance of metadata attributes.

We proposed a parallel version of the algorithm, Borromean-P, to be able to generate large graphs (in the order of hundreds of thousands nodes). We partitioned the problem into smaller graph segments fractionally and merge the segments into a final graph respecting the triads already formed. We used the average path length as a termination condition to more accurately reproduce a desired average path length.

We implemented the parallel algorithm in two versions. For testing the accuracy of the results we implemented the parallel algorithm in Python. We refer to this implementation as Borromean-P-Python.

For testing its performance in terms of execution time and scalability, we also implemented the parallel Borromean-P algorithm using Spark, Borromean-P-Spark. We have demonstrated success on graphs up to 750,000 nodes. Performance improves as more machines and memory are added.

We have provided thorough experimental evaluations on six datasets from real networks with binary node attributes.

In the next section, we present a discussion of results across all algorithm variants for both the percentage of binary node attributes and the transitivity of the respective generated graph. Then we conclude with a discussion of lessons and directions for future work.

8.1 Discussion of Results for All Algorithm Variants

In this section, we compare the node attribute and edge type percentages of each generated graph. Variation from the attribute percentages of the original graph occurs because of several factors. The dk-2 generated graphs include only the greatest connected component (GCC) which drops nodes and edges from the original. In addition, the pseudo-graph matching algorithm drops edges when it finds only saturated nodes. Attribute variation is also affected by the structure of particular graphs.

Table 8.1: Label A Attribute Percentage Comparison. Column B-S presents results for Borromean-S. Column B-P-P presents results for Borromean-P-Python. Column B-P-S presents results for Borromean-P-Spark. Column % Var presents the percentage variation from the original graph.

Dataset	Original %	B-S %	% Var	B-P-P %	% Var	B-P-S %	& Var
Polblogs	48	66	38	73	52	69	44
FB-C	72	79	10	74	3	68	6
FB-D	62	63	2	70	13	67	8
FB-M	78	65	-17	64	-18	64	-18
Amazon	81	81	0	NA	NA	72	-11
Steam SE	97	88	-9	NA	NA	87	-11

Table 8.2: Label B Attribute Percentage Comparison

Dataset	Original %	B-S %	% Var	B-P-P %	% Var	B-P-S %	% Var
Polblogs	52	34	-35	27	-48	31	-40
FB-C	28	20	-28	25	-11	31	11
FB-D	37	36	-3	29	-22	33	-11
FB-M	22	34	55	36	64	36	64
Amazon	18	18	0	NA	NA	28	56
Steam SE	2	12	500	NA	NA	13	550

In Tables 8.1 and 8.2 we compare the relative percentages of the node attributes A and B respectively in each graph. Facebook-Caltech and Dartmouth have the lowest variation of attribute A. FB-Dartmouth

Table 8.3: Edge Type A-A Percentage Comparison

Dataset	Original %	B-S %	% Var	B-P-P %	% Var	B-P-S %	% Var
Polblogs	44	31	-30	40	-9	32	-27
FB-C	69	53	-23	55	-20	45	-35
FB-D	58	39	-33	41	-29	43	-26
FB-M	72	42	-42	42	-42	42	-42
Amazon	83	57	-31	NA	NA	47	-43
Steam SE	84	53	-37	NA	NA	50	-40

Table 8.4: Edge Type B-B Percentage Comparison

Dataset	Original %	B-S %	% Var	B-P-P %	% Var	B-P-S %	% Var
Polblogs	48	48	0	47	-2	47	-2
FB-C	8	7	-13	7	-13	11	38
FB-D	18	13	-28	14	-22	12	-33
FB-M	9	12	33	12	33	12	33
Amazon	2	5	150	NA	NA	9	350
Steam SE	0.9	6	567	NA	NA	8	789

Table 8.5: Edge Type A-B Percentage Comparison

Data Set	Original %	B-S %	% Var	B-P-P %	% Var	B-P-S %	% Var
Polblogs	8	19	138	13	63	22	175
FB-C	23	39	70	38	65	45	96
FB-D	24	46	92	45	88	44	83
FB-M	19	46	142	45	137	45	137
Amazon	16	37	131	NA	NA	44	175
Steam SE	14	40	186	NA	NA	42	200

has a 2% variation from the original graph in Borromean-S, 13% in Borromean-P-Python, and 8% in Borromean-P-Spark. FB-Caltech has a 10% variation in Borromean-S, 3% in Borromean-P-Python and 6% in Borromean-P-Spark. FB-Dartmouth has the closest variation overall for attribute B with a decrease of 3% in Borromean-S, 22% in Borromean-P-Python, and 11% in Borromean-P-Spark. The Amazon products graph matches attributes A and B exactly in Borromean-S, but has a 56% variation with regard to attribute B in Borromean-P-Spark.

The Polblogs graph exhibits the highest variation among the smaller graphs on average across both attributes A and B. This may be because the graph is almost equally split between attributes.

The Amazon and Steam Sweden graphs are too large to be computationally feasible with Borromean-P-Python, but the Steam Sweden graph shows high variation in attribute B nodes in Borromean-S and Borromean-P-Spark increasing 500 and 550% respectively. The disparity is possibly due to its low percentage of attribute B. In the Sweden graph, attribute B represents players who have been identified as cheaters. The graph has 2% cheaters, so any loss in generation has a large effect.

In Tables 8.3, 8.4, and 8.5 we compare the percentages of the edge types A-A, B-B, and A-B. FB-Caltech has the lowest overall variation in A-A and A-B edges, but FB-Dartmouth has a lower variation for A-A edges in Borromean-P-Spark. The Polblogs has fewer A-B edges due to the relatively small number of connections between liberal and conservative bloggers. The Amazon and Steam Sweden graphs show a high variation of B-B and A-B edges.

Table 8.6: Clustering Coefficient (Transitivity) Comparison. The B-P-P column is the transitivity for the Borromean-Parallel-Python graph.

Dataset	Orig	Orbis	% Var	B-S	% Var	B-P-P	% Var	B-P-S	% Var
Polblogs	0.22	0.17	-23	0.19	-14	0.22	0	0.25	14
FB-C	0.29	0.16	-45	0.18	-38	0.29	0	0.20	-31
FB-D	0.15	0.03	-80	0.05	-67	0.12	-20	0.12	-20
FB-M	0.13	0.01	-92	0.02	-85	0.06	-54	0.06	-54
Amazon	0.21	1E-4	-99	5E-5	-99	NA	NA	4E-4	-99
Steam SE	0.13	1E-4	-99	0.0002	-99	NA	NA	1E-3	-99

We also compare the transitivity values of each generated graph. Table 8.6 shows the transitivity resulting from each generation method. As discussed in Chapter 5, Borromean improves generated graph transitivity versus Orbis except in the Amazon products data set. The Amazon graph has fewer edges for its node set size, and its average path length is greater. Borromean-P-Python shows the greatest fidelity overall. The transitivity of the Amazon and Steam Sweden graphs in Borromean-P-Spark improves upon the results from Borromean-S. The Facebook Dartmouth and Michigan graphs show the same transitivity with both Borromean-P-Python and Borromean-P-Spark. Borromean-P-Python matches the transitivity of the Polblogs and Facebook Caltech graphs exactly.

A potential drawback of this work is that we have reported only a single graph generation for each implementation variant rather than an average over multiple generations given that the graphs are randomized. However, the fractional partitioning used in Borromean-P creates similar labeled joint-degree distributions for each partition. Thus, the partitions are roughly equivalent to multiple generations within the probabilistic space of the given distribution.

8.2 Future Work

This work can be extended in various ways. First, the exact scale curve of the algorithm needs to be determined, and the best practices for the randomized preservation of the clustering coefficient needs to be investigated further. Second, the performance of the swap/stitch version of our Spark algorithm needs to be improved so as to more precisely influence clustering behavior. Third, the parallel implementation needs improvement to prevent the loss of edges with label B, and both implementations need to improved to alleviate the edge starvation that occurs during the pseudo-graph matching phase of dk2 generation. It may be possible to incorporate an implementation of the NeigbhorSwap function used in 2k_Simple [25] into Borromean-P.

GraphX has become obsolete and its development inactive [54], so it would be worthwhile to attempt to implement the algorithm in its likely successor GraphFrames. GraphFrames is designed for Spark version 2.0, though it is partially compatible with 1.6.

REFERENCES

[1] Z. Ruan, M. Tang, and Z. Liu, "Epidemic spreading with information-driven vaccination," *Physical Review E*, vol. 86, no. 3, p. 036117, 2012.

[2] J. Skvoretz, "Diversity, integration, and social ties: Attraction versus repulsion as drivers of intra- and intergroup relations," *American Journal of Sociology*, vol. 119, pp. 486–517, 2013.

[3] K. Lerman, R. Ghosh, and T. Surachawala, "Social contagion: An empirical study of information spread on digg and twitter follower graphs," *CoRR*, vol. abs/1202.3162, 2012. [Online]. Available: http://arxiv.org/abs/1202.3162

[4] J. Blackburn, R. Simha, N. Kourtellis, X. Zuo, M. Ripeanu, J. Skvoretz, and A. Iamnitchi, "Branded with a scarlet "C": cheaters in a gaming social network," in *Proceedings of the 21st international conference on World Wide Web*. New York, NY, USA: ACM, 2012, pp. 81–90.

[5] J. Blackburn, N. Kourtellis, J. Skvoretz, M. Ripeanu, and A. Iamnitchi, "Cheating in online games: A social network perspective," *ACM Trans. Internet Technol.*, vol. 13, no. 3, pp. 9:1–9:25, May 2014. [Online]. Available: http://doi.acm.org/10.1145/2602570

[6] P. Erdös and A. Rényi, "On random graphs, I," *Publicationes Mathematicae (Debrecen)*, vol. 6, pp. 290–297, 1959.

[7] C. Seshadhri, T. G. Kolda, and A. Pinar, "Community structure and scale-free collections of erdös-rényi graphs," *Phys. Rev. E*, vol. 85, p. 056109, May 2012. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.85.056109

[8] T. G. Kolda, A. Pinar, T. D. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *CoRR*, vol. abs/1302.6636, 2013. [Online]. Available: http://arxiv.org/abs/1302.6636

[9] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[10] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *The Journal of Machine Learning Research*, vol. 11, pp. 985–1042, 2010.

[11] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat, "Systematic topology analysis and generation using degree correlations," in *ACM SIGCOMM Computer Communication Review*, vol. 36. ACM, 2006, pp. 135–146.

[12] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Y. Zhao, "Sharing graphs using differentially private graph models," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011, pp. 81–98.

[13] W. Aiello, F. Chung, and L. Lu, "A random graph model for massive graphs," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. Acm, 2000, pp. 171–180.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[15] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772751

[16] B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research," in *15th European Conference on Machine Learning*. Springer, 2004, pp. 217–226.

[17] J. Shetty and J. Adibi, "Discovering important nodes through graph entropy the case of enron email database," in *Proceedings of the 3rd international workshop on Link discovery*, ser. LinkKDD '05. New York, NY, USA: ACM, 2005, pp. 74–81. [Online]. Available: http://doi.acm.org/10.1145/1134271.1134282

[18] M. Barbaro and T. Zeller, "A face is exposed for aol searcher no. 4417749," *The New York Times*, p. A1, August 9 2006.

[19] A. Narayanan and V. Shmatikov, "De-anonymizing social networks," in *Security and Privacy, 2009 30th IEEE Symposium on*, 2009, pp. 173–187.

[20] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2005, pp. 133–145.

[21] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1217299.1217301

[22] P. W. Holland and S. Leinhardt, "An exponential family of probability distributions for directed graphs," *Journal of the american Statistical association*, vol. 76, no. 373, pp. 33–50, 1981.

[23] M. Morris, M. S. Handcock, and D. R. Hunter, "Specification of exponential-family random graph models: terms and computational aspects," *Journal of statistical software*, vol. 24, no. 4, p. 1548, 2008.

[24] M. Gjoka, M. Kurant, and A. Markopoulou, "2.5-graphs: from sampling to generation," in *INFOCOM*. IEEE, 2013, pp. 1968–1976.

[25] M. Gjoka, B. Tillman, and A. Markopoulou, "Construction of simple graphs with a target joint degree matrix and beyond," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 2015, pp. 1553–1561.

[26] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat, "Systematic topology analysis and generation using degree correlations," in *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '06. New York, NY, USA: ACM, 2006, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1159913.1159930

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[28] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824077

[29] M. Kabiljo, D. Logothetis, S. Edunov, and A. Ching. (2016, October) A comparison of state-of-the-art graph processing systems. [Online]. Available: https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/

[30] S. Edunov, D. Logothetis, C. Wang, A. Ching, and M. Kabiljo, "Darwini: Generating realistic large-scale social graphs," *arXiv preprint arXiv:1610.00664*, 2016.

[31] A. Prat-Pérez, J. Guisado-Gámez, X. F. Salas, P. Koupy, S. Depner, and D. B. Bartolini, "Towards a property graph generator for benchmarking," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. ACM, 2017, p. 6.

[32] L. Backstrom, C. Dwork, and J. Kleinberg, "Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography," *Communications of the ACM*, vol. 54, no. 12, pp. 133–141, 2011.

[33] M. Hay, G. Miklau, D. Jensen, P. Weis, and S. Srivastava, "Anonymizing social networks," *Computer Science Department Faculty Publication Series*, p. 180, 2007.

[34] G. Cormode, D. Srivastava, T. Yu, and Q. Zhang, "Anonymizing bipartite graph data using safe groupings," *The VLDB Journal*, vol. 19, no. 1, pp. 115–139, Sep. 2009.

[35] C. Dwork, "Differential privacy: A survey of results," in *Theory and Applications of Models of Computation*. Springer, 2008, pp. 1–19.

[36] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. IEEE, May 2008, pp. 111–125.

[37] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, 2002.

[38] B. Zhou, J. Pei, and W. Luk, "A brief survey on anonymization techniques for privacy preserving publishing of social network data," *SIGKDD Explor. Newsl.*, vol. 10, no. 2, pp. 12–22, Dec. 2008. [Online]. Available: http://doi.acm.org/10.1145/1540276.1540279

[39] X. Wu, X. Ying, K. Liu, and L. Chen, *A survey of privacy-preservation of graphs and social networks*. Springer, 2010, pp. 421–453.

[40] M. Arrington, "Aol proudly releases massive amounts of private data," August 2006. [Online]. Available: http://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data.

[41] A. Narayanan, E. Shi, and B. I. Rubinstein, "Link prediction by de-anonymization: How we won the kaggle social network challenge," in *Proceedings of the 2011 IEEE IJCNN International Joint Conference on Neural Networks*. IEEE, 2011, pp. 1825–1834.

[42] C. Aggarwal, Y. Li, and P. Yu, "On the hardness of graph anonymization," in *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, Dec 2011, pp. 1002–1007.

[43] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of Cryptography*. Springer, 2006, pp. 265–286.

[44] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," in *Foundations and Trends in Theoretical Computer Science*. Now, 2014, vol. 9, no. 3-4, pp. 211–407.

[45] E. Shen and T. Yu, "Mining frequent graph patterns with differential privacy," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 545–553.

[46] A. L. Traud, P. J. Mucha, and M. A. Porter, "Social structure of facebook networks," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 16, pp. 4165–4180, 2012.

[47] L. A. Adamic and N. Glance, "The political blogosphere and the 2004 us election: divided they blog," in *Proceedings of the 3rd international workshop on Link discovery*. ACM, 2005, pp. 36–43.

[48] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, p. 5, 2007.

[49] J. Blackburn, R. Simha, N. Kourtellis, X. Zuo, M. Ripeanu, J. Skvoretz, and A. Iamnitchi, "Branded with a scarlet c: cheaters in a gaming social network," in *Proceedings of the 21st International Conference on World Wide Web*. ACM, 2012, pp. 81–90.

[50] P. Mahadevan. Orbis topology generator suite. [Online]. Available: http://www.sysnet.ucsd.edu/~pmahadevan/topo_research/topo.html

[51] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, 1st ed. Indianapolis: Pearson Education, 2002.

[52] M. Newman, *Networks: An Introduction*, 1st ed. Oxford: Oxford University Press, May 2010.

[53] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley, 2011.

[54] H. Karau and R. Warren, *High Performance Spark*. O'Reilly, 2017.

[55] S. Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly, 2015.

[56] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of 11th USENIX Symposium Operating Systems Design and Implementation*, 2014, pp. 599–613.

[57] A. Wadia. Computing shortest distances incrementally with spark. [Online]. Available: https://blog.insightdatascience.com/computing-shortest-distances-incrementally-with-spark-1a280064a0b9

[58] Graphmod. [Online]. Available: https://github.com/akshaywadia/graphMod

[59] A. Shimbel, "Structure in communication nets," in *Proceedings of the symposium on information networks*, vol. 4, 1954.

[60] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[61] L. R. Ford Jr, "Network flow theory," RAND CORP SANTA MONICA CA, Tech. Rep., 1956.

[62] E. F. Moore, "The shortest path through a maze," in *Proc. Int. Symp. Switching Theory, 1959*, 1959, pp. 285–292.

[63] Cai, Logothetis, and Siganos, "Facilitating real-time graph mining," in *Cloud Data Management*, 2012.

[64] S. Ryza. How-to: Tune your apache spark jobs. [Online]. Available: http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

[65] Apache. Tuning spark. [Online]. Available: https://spark.apache.org/docs/latest/tuning.html

[66] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "Xsede: Accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, Sept.-Oct. 2014.

[67] Wrangler data portal. [Online]. Available: https://portal.wrangler.tacc.utexas.edu

[68] Apache. EdgeRDD difference throws an exception. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-17265