Decision Support and Business Intelligence

Big Data Research Project

Kunal Arora
Amritansh Sharma

Graph Partitioning for efficient Data analytics

CentraleSupélec

Supervisors: Nacéra Seghouani Bennacer, Adnan EL MOUSSAWI

Contents

Chapter 1

Introduction

In this section, we will explain what does it mean by a graph, what are different types of graphs and what are the different properties of graph.

We will also see what are the different traditional graph algorithms and why do we even need graphs.

1.1 What is Graph?

A set of items connected by edges. Each item is called a vertex or node. Formally, a graph is a set of vertices and a binary relation between vertices, adjacency. Figure 1.1 provides an example of a graph with three vertices and three edges.

A graph G can be defined as a pair (V,E), where V is a set of vertices, and E is a set of edges between the vertices $E \subseteq \{(u, v)|u, v\varepsilon V\}$. If the graph is undirected, the adjacency relation defined by the edges is symmetric, or $E \subseteq \{(u, v)|u, v\varepsilon V\}$ (sets of vertices rather than ordered pairs). If the graph does not allow self-loops, adjacency is irreflexive.

A simple undirected graph is shown below. This graph has three vertices and three edges.

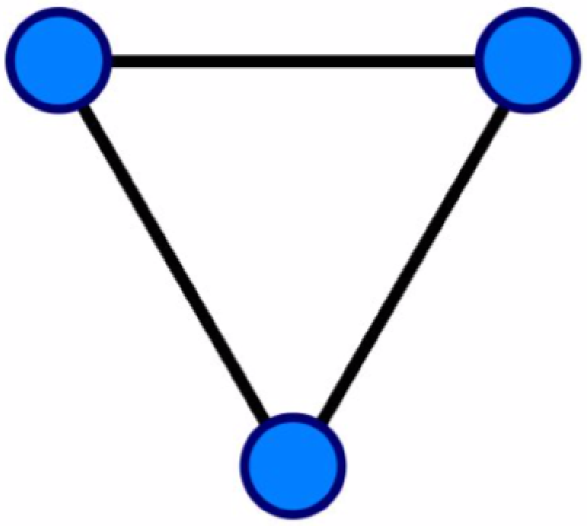


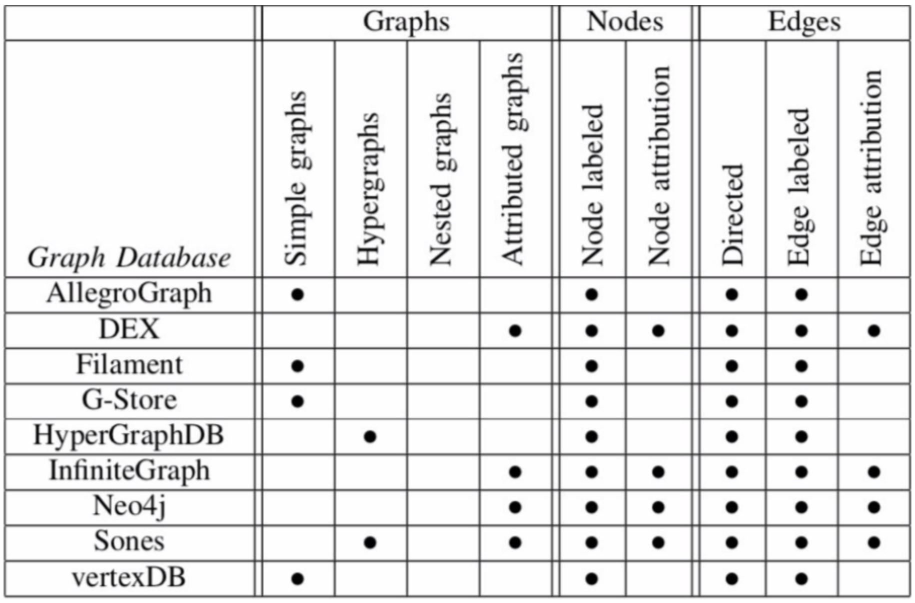
Figure 1.1: A simple graph with three vertices and three edges

There are different kinds of Graphs. Two of the most common ones are:

- **Directed Graphs:** A **directed graph** or **digraph** is a graph in which edges have orientations.

- **Weighted Graph:** A **weighted graph** or a **network** is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand.

Some very famous graph databases[Angles 2012] are mentioned in Figure 1.2. The comparison is made based on the Graph Data Structures. These databases are all after year 2003.



	Graphs				Nodes		Edges		
Graph Database	Simple graphs	Hypergraphs	Nested graphs	Attributed graphs	Node labeled	Node attribution	Directed	Edge labeled	Edge attribution
AllegroGraph	●				●		●	●	
DEX				●	●	●	●	●	●
Filament	●				●		●	●	
G-Store	●				●		●	●	
HyperGraphDB		●			●		●	●	
InfiniteGraph				●	●	●	●	●	●
Neo4j				●	●	●	●	●	●
Sones		●		●	●	●	●	●	●
vertexDB	●				●		●	●	

Figure 1.2: Different Graph Data Structures

1.2 Why do we need Graphs?

There is a huge influx of data in the world with no relevant systems that can perform fast analytics on a scale. Data is increasing at an exponential rate and is more connected that cannot be processed by the classical storage and processing systems that we have at our disposal.

The need of graph arises after the concept of Big Data became popular where we are talking about million to trillion GB of information.

According to the ex-CEO of Google, Eric Schmidt, **"There were 5 billion GB of information created between the dawn of civilization through 2003, but that much information is now created every 2 days"**

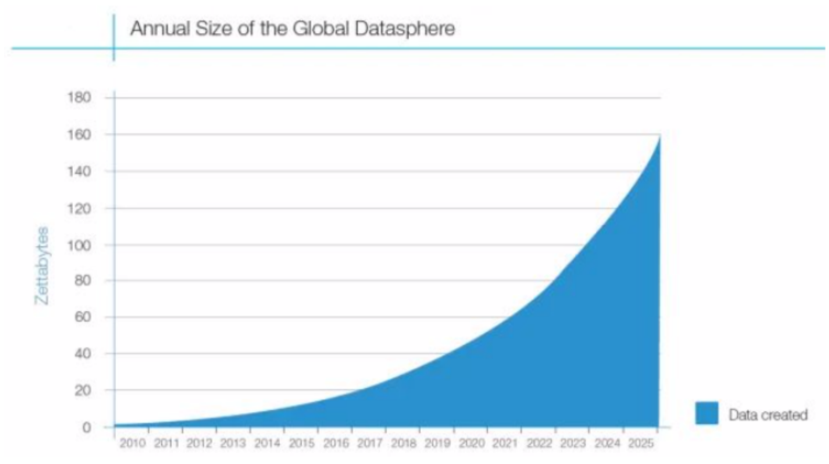


Figure 1.3: Increase in the amount of data creation by 2025

Looking at Figure 1.3, we can state that the data is increasing at a very rapid exponential rate with,160 trillion GB expected by 2025. The classical storage systems like, relational, NoSQL DBs are not scalable and efficient to process such data and hence, the data processing time is quite high. We need systems and storage which can give us analysis in fraction of seconds, and can operate in distributed parallel fashion.

Graphs are present everywhere now-a-days, including social-media like, Facebook, Twitter, Science like, acm, bio-medical field, Advertising on Amazon, Netflix, Web, Wikipedia, google search etc.

The advantages of using graphs for such systems are:

- Faster Machine Learning and Data analytics.
- Helps to identify communities or Influential people to target for specific ads, products or services.

Chapter 2

Related Work

In this section, we will explain about different distributed and graph processing algorithms that already exist for performing graph computations. Some of the most widely studied graph distributing models/algorithms are, Pregel, GraphLab and PowerGraph. For graph partitioning are, Label Propagation, Spinner and Multi-level coarsening.

We will describe briefly these algorithms in this section and get deeper understanding of a few in a different section.

2.1 Distributed Graphs

In this section, we will explain Pregel, GraphLab and PowerGraph algorithms. We will also see the main differences between the above 3 algorithms and also study Natural graphs and distributed graph systems.

2.1.1 Pregel

The pregel[Malewicz 2010] programming model is based on the high-level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel model. Pregel computations consist of a sequence of iterations, called supersteps. During a superstep the framework invokes a user-defined function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex V and a single superstep S. It can read messages sent to V in superstep S 1, send messages to other vertices that will be received at superstep S + 1, and modify the state of V and its outgoing edges. Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifier is known.

Pregel is designed to run on Google's cluster architecture. The Pregel library divides a graph into partitions, each consisting of a set of vertices and all of those vertices' outgoing edges. Assignment of a vertex to a partition depends solely on the vertex ID, which implies it is possible to know which partition a given vertex belongs to even if the vertex is owned by a different machine, or even if the vertex does not yet exist. The default partitioning function is just hash(ID) mod N, where N is the number of partitions, but users can replace it.

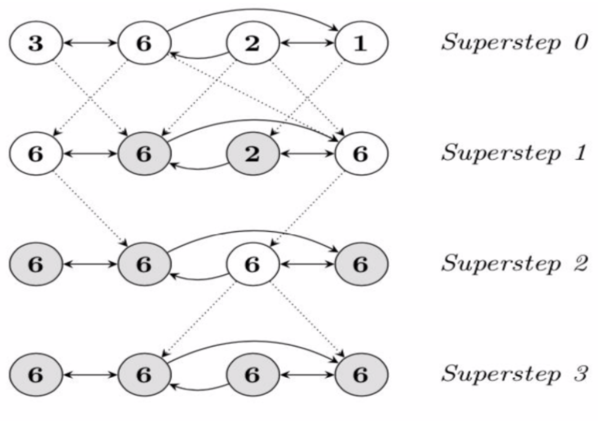


Figure 2.1: Maximum Value Example using Think-Like-A-Vertex paradigm in Pregel

2.1.2 GraphLab

The GraphLab framework[Low 2012] is a parallel programming abstraction targeted for sparse iterative graph algorithms. GraphLab provides a high level programming interface, allowing a rapid deployment of distributed machine learning algorithms. The main design considerations behind the design of GraphLab are:

- Sparse data with local dependencies.
- Iterative algorithms
- Potentially asynchronous execution

Main features of GraphLab are:

- A unified multicore and distributed API: write once run efficiently in both shared and distributed memory systems
- Tuned for performance: optimized C++ execution engine leverages extensive multi-threading and asynchronous IO
- Scalable: GraphLab intelligently places data and computation using sophisticated new algorithms
- HDFS Integration
- Powerful Machine Learning Toolkits

Graph databases typically focus on efficient storage and retrieval of graph structured data with support for basic graph computation. In contrast, GraphLab focuses on iterative graph structured computation.

Some additional features of GraphLab are:

- A distributed data graph format built around a two-stage partitioning scheme which allows for efficient load balancing and distributed ingress on variable-sized cluster deployment.

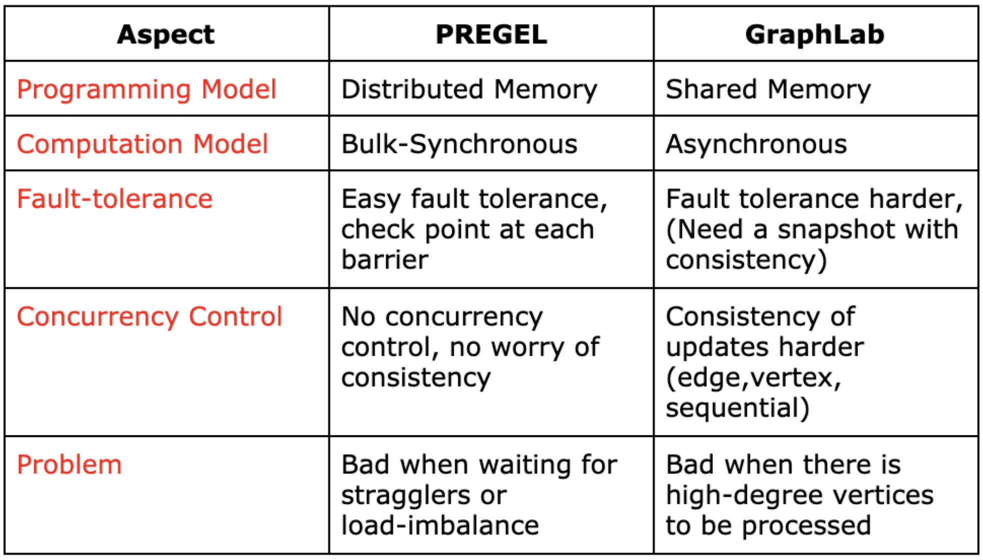
- GraphLab works on **two engines**: a chromatic engine that is partially synchronous and assumes the existence of a graph coloring, and a locking engine that is fully asynchronous, supports general graph structures, and relies upon a novel graph-based pipelined locking system to hide network latency.

- It also introduced **two fault tolerance mechanisms**: a synchronous snapshot algorithm and a fully asynchronous snapshot algorithm based on Chandy-Lamport snapshots that can be expressed using regular GraphLab primitive.

2.1.3 Pregel vs GraphLab

The key difference between Pregel and GraphLab is that Pregel has a barrier at the end of every iteration, whereas GraphLab is completely asynchronous. Asynchrony in GraphLab allows it to prioritize more complex vertices over others, but it also calls for consistency models to maintain sanity of results. GraphLab proposes three consistency models: full, edge, and vertex consistency, to allow different levels of parallelism. Another difference is that Pregel allows dynamic modifications to the graph structure, whereas GraphLab does not.

The key differences and a major problem with PREGEL and GraphLab are provided in Figure 2.2.



Aspect	PREGEL	GraphLab
Programming Model	Distributed Memory	Shared Memory
Computation Model	Bulk-Synchronous	Asynchronous
Fault-tolerance	Easy fault tolerance, check point at each barrier	Fault tolerance harder, (Need a snapshot with consistency)
Concurrency Control	No concurrency control, no worry of consistency	Consistency of updates harder (edge,vertex, sequential)
Problem	Bad when waiting for stragglers or load-imbalance	Bad when there is high-degree vertices to be processed

Figure 2.2: Pregel vs GraphLab key differences

2.1.4 PowerGraph

PowerGraph is a graph-based, high performance, distributed computation framework written in C++. GraphLab PowerGraph[Gonzalez 2012] is the culmination of 4-years of research and development into graph computation, distributed computing, and machine learning. GraphLab PowerGraph scales to graphs with billions of vertices and edges easily, performing orders of magnitude faster than competing systems. GraphLab PowerGraph combines advances in machine learning algorithms, asynchronous distributed graph computation, prioritized scheduling, and graph placement with optimized low-level system design and efficient data-structures to achieve unmatched performance and scalability in challenging machine learning tasks.

Main features of PowerGraph are:

- It is tailored to handle the distributed storage and processing of **Natural or complex graphs**
- Unified multicore/distributed API: write once run anywhere
- Tuned for performance: optimized C++ execution engine leverages extensive multithreading and asynchronous IO
- Scalable: Run on large cluster deployments by intelligently placing data and computation
- HDFS Integration: Access your data directly from HDFS
- Powerful Machine Learning Toolkits: Tackle challenging machine learning problems with ease

2.1.5 Natural Graphs

Natural graphs[Kim 2008] are a special kind of graphs which are specific to social-media, biomedical, defence field. These are the graphs which are widely present now-a-days.

Natural graphs have a defining characteristics, which is strong power-law distribution. It means some vertices have just one or 2 neighbors while 1% of the vertices in the graph have more than 50% of the total vertices as it's neighbors. An example of natural graphs is the AltaVista graph whose degree distribution is shown in Figure 2.3. It has a large number fo vertices with only one neighbour as well as some vertices having more than one million neighbours.

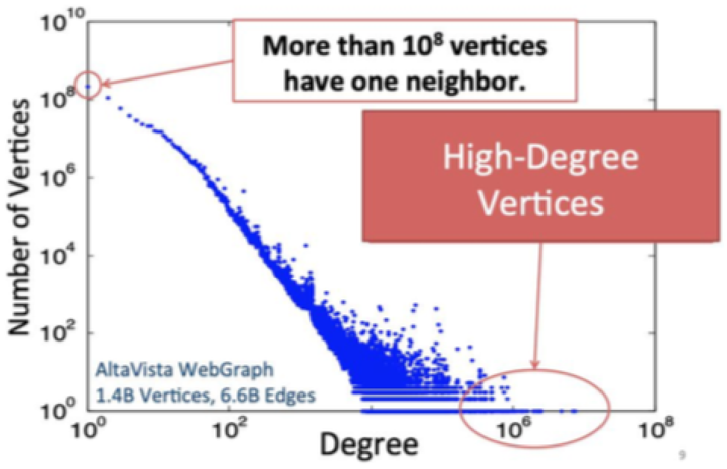


Figure 2.3: Power-law degree distribution for AltaVista graph

This leads to a star like schema in the graph structure. The very basic example of Natural graphs is, Twitter Follower Graph. In that, imagine Barack Obama will be followed by many millions (Figure 2.4). This is the case of high-degree vertices.

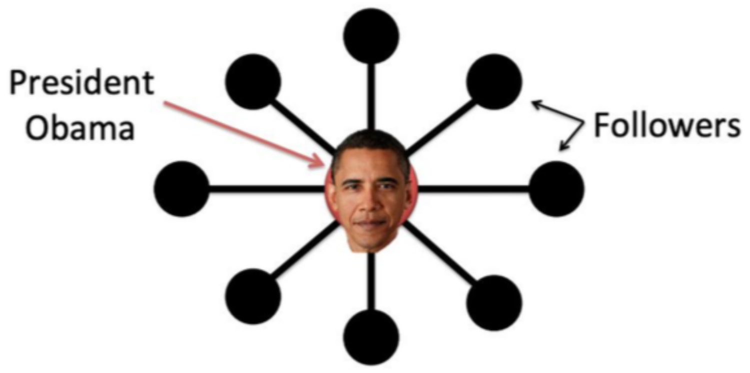


Figure 2.4: High-Degree vertices: Obama followed by many people on twitter

PowerGraph identified common patterns from the vertex-program of Pregel and GraphLab and decomposed them into 3 steps which is **G**ather, **A**pply and **S**catter (GAS).

Also, PowerGraph performs a vertex-cut on these high degree vertices in order to execute the computation on different machines in a fast and distributed manner.

2.1.6 Distributed Graph Systems

There are many distributed graph systems[Ammar 2018] in the market which are targeted to specific needs. Some of them are mentioned below based on their type:

- Vertex-centric
  - Synchronous: Giraph, Blogel-Vertex (Blogel-V)
  - Asynchronous: GraphLab
- Block-centric: Blogel-Block (Blogel-B)
- MapReduce: Hadoop
- MapReduce extensions: HaLoop , Spark/GraphX
- Relational: Vertica
- Stream: Flink Gelly

2.2 Graph Partitioning

Graph Partitioning is the problem of partitioning a graph so that the vertices in each partition are mutually exclusive and the partitions are exhaustive, that is, their union include all vertices of the original unpartitioned graph.

Edges of the original graph that cross between the groups are called cut-edges in the partitioned graph. If the number of cut-edges is small compared to the original graph, then the partitioned graph is better suited for analysis than the original, when distributed across a cluster.

Typically, graph partitioning falls under the category of NP-hard problems; so solutions to these problems are generally derived using heuristics and approximation algorithms. Also, there does not exist any approximation algorithms with a constant ratio factor for general graphs. In addition to minimising the edge-cut, two further constraints are typically applied in the literature of Graph Partitioning, namely vertex-balance and edge-balance. An ()-Vertex-balanced partitioning of a graph ensures that the number of vertices in any partition is at-most (1+) times the average number of vertices per partition. The Edge-balance constraint is similarly defined.

A low Edge-cut partitioning reduced the Network traffic and Memory usage on a distributed system. A balanced partitioning is necessary to ensure better parallelism by avoiding stragglers leading to idling nodes, when using Synchronous Graph Processing frameworks. We study the most important systems for Graph Partitioning, that is, Label Propagation, Spinner Algorithm and Multi-level graph Partitioning approach.

2.2.1 Label Propagation

Label Propagation[Ugander 2013] is an iterative algorithm which only considers minimising the Edge-cut criteria while partitioning. To partition a graph into K-partitions, the vertices in the graph are initialised with one of K labels at random. At every iteration, each vertex receives calculates a score of each label based on an edge-weighted sum of the labels of its neighbours, and chooses the label with the maximum score. This is repeated until convergence. Since the score computation is vertex-centric, Label Propagation algorithm is scalable using distributed system.

2.2.2 Spinner Algorithm

Spinner[Martella 2017] is a scalable and adaptive graph partitioning algorithm based on label propagation and designed on top of the Pregel model. Spinner scales to massive graphs, produces partitions with low edge-cut and balance. The idea behind Spinner Algorithm is to add a penalty term to the score based on how unbalanced the partitioning becomes on assigning a particular label to a vertex. This is done by maintaining a counter per partition representing its size. Figure 2.5 visualises the improvement in locality over hash partitioning using Spinner.

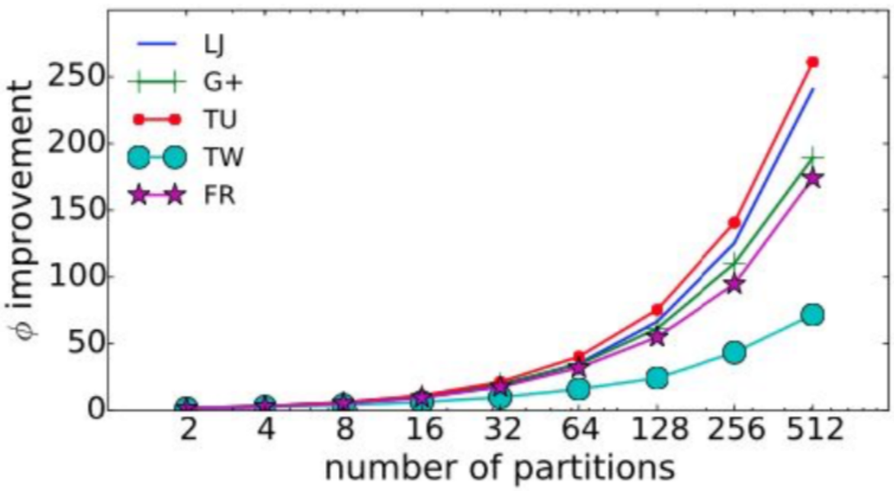


Figure 2.5: Spinner Algorithm: improvement in locality over hash partitioning

2.2.3 Multi-level Graph Partitioning

Multi-level[Meyerhenke 2017] algorithms work in 3 levels:

- Coarsening: parts(subgraphs) of the graph are combined into a single node preserving the edge information.
- Partitioning: the coarsened graph is partitioned.
- Un-coarsening or refinement: the partitioned coarsened graph is un-coarsened to obtain a partitioning of the original graph.

Meyerhenke et al proposed a multi-level algorithm that uses the size-constrained label propagation algorithm to compute a graph clustering in the contraction phase. This is repeated until the graph is small enough. The coarsest graph is then partitioned by using a distributed evolutionary algorithm. During un-coarsening, the size-constraint label propagation algorithm is used as an effective parallel local search algorithm. Figure 2.6 depicts the above algorithm step-by-step.

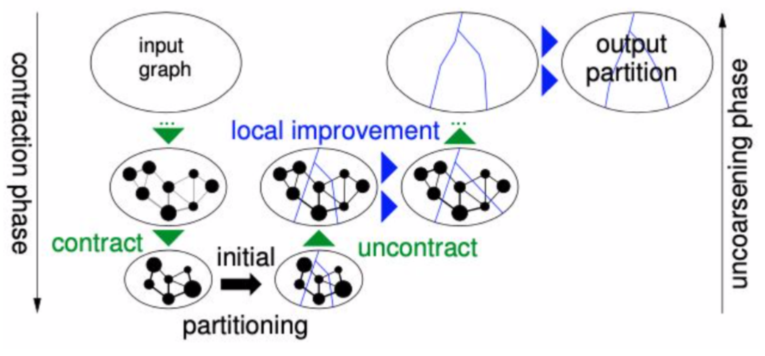


Figure 2.6: Algorithm by Meyerhenke et al: multi-level partitioning based on size-constrained label propagation

Chapter 3

Research

The research goal of this study is to benchmark different real-world and artificially generated Natural Graphs in terms of graph partitioning and the performance of graph processing algorithms on these graphs on two graph algorithms, namely, page-rank and triangle-counting on a state-of-the-art Graph processing framework. We are using NetworkX-Metis[met] to generate graphs with varying power-law degree distribution and GraphX[gra] to run graph processing algorithms.

In Section 3.1, we outline the research methodology followed to achieve our research goal. In Section 3.2, we present the results of the evaluation of the developed methods and the experimental study. Finally, we conclude by discussing the achievements of our study, its limitations and the learning outcomes.

3.1 Methodology

3.1.1 Tools

We used two different tools to generate and process Natural graphs.

- **NetworkX-Metis:** NetworkX-Metis is a Python package for the creation, manipulation and study of the structure, dynamics, and functions of complex graphs. Metis is a C library written for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. NetworkX-Metis uses Cython to wrap the Metis library to make it available in Python.

  NetworkX-Metis github: `https://github.com/networkx/networkx-metis`

- **GraphX:** GraphX run on top of spark and uses spark configuration. We are installing spark in our local machine and running the graph processing algorithms supported by GraphX which are, Page rank and Triangle couting.

  GraphX github: `https://github.com/apache/spark/tree/v2.4.4/graphx`

3.1.2 Datasets

In this section, we will give the statistics and sources of real-world graphs (like facebook, twitter etc) and also, the methodology and statistics of artificially generating Natural graphs.

3.1.2.1 Real-world graphs

Here we will give the statistics and sources of the already existing real-world natural graphs that follow a certain power-law degree distribution. These datasets are specific for working with Power-Law degree graphs based on the Ego networks[Leskovec 2012].

- **Twitter Graph:** Twitter graph is one of the best datasets for understanding and bench-marking Natural Graphs available at the moment. In twitter, there could be many followers of a particular person like famous personalities, Barack Obama, Donald Trump etc. which generates the perfect Power-Law degree distribution[Leskovec 2012].

- **Facebook Graph:** This webgraph is a page-page graph of verified Facebook sites. Nodes represent official Facebook pages while the links are mutual likes between sites. Node features are extracted from the site descriptions that the page owners created to summarize the purpose of the site. This graph was collected through the Facebook Graph API in November 2017 and restricted to pages from 4 categories which are defined by Facebook. These categories are: politicians, governmental organizations, television shows and companies. The task related to this dataset is multi-class node classification for the 4 site categories[Rozemberczki 2019].

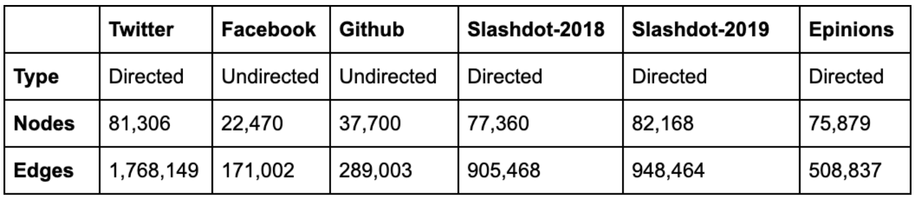
- **Github Graph:** A large social network of GitHub developers which was collected from the public API in June 2019. Nodes are developers who have starred at least 10 repositories and edges are mutual follower relationships between them. The vertex features are extracted based on the location, repositories starred, employer and e-mail address. The task related to the graph is binary node classification - one has to predict whether the GitHub user is a web or a machine learning developer. This target feature was derived from the job title of each user[Rozemberczki 2019].

- **Slashdot-2018 Graph:** Slashdot is a technology-related news website know for its specific user community. The website features user-submitted and editor-evaluated current primarily technology oriented news. In 2002 Slashdot introduced the Slashdot Zoo feature which allows users to tag each other as friends or foes. The network cotains friend/foe links between the users of Slashdot. The network was obtained in November 2008[Leskovec 2009].

- **Slashdot-2019 Graph:** In 2002 Slashdot introduced the Slashdot Zoo feature which allows users to tag each other as friends or foes. The network cotains friend/-foe links between the users of Slashdot. The network was obtained in February 2009[Leskovec 2009].

- **Epinions Graph:** This is a who-trust-whom online social network of a a general consumer review site Epinions.com. Members of the site can decide whether to ”trust” each other. All the trust relationships interact and form the Web of Trust which is then combined with review ratings to determine which reviews are shown to the user[Leskovec 2012].

Figure 3.1 and 3.2 show the statistics of various real-world graphs as a table and plot respectively.



	Twitter	Facebook	Github	Slashdot-2018	Slashdot-2019	Epinions
Type	Directed	Undirected	Undirected	Directed	Directed	Directed
Nodes	81,306	22,470	37,700	77,360	82,168	75,879
Edges	1,768,149	171,002	289,003	905,468	948,464	508,837

Figure 3.1: Dataset statistics of different real-world graphs

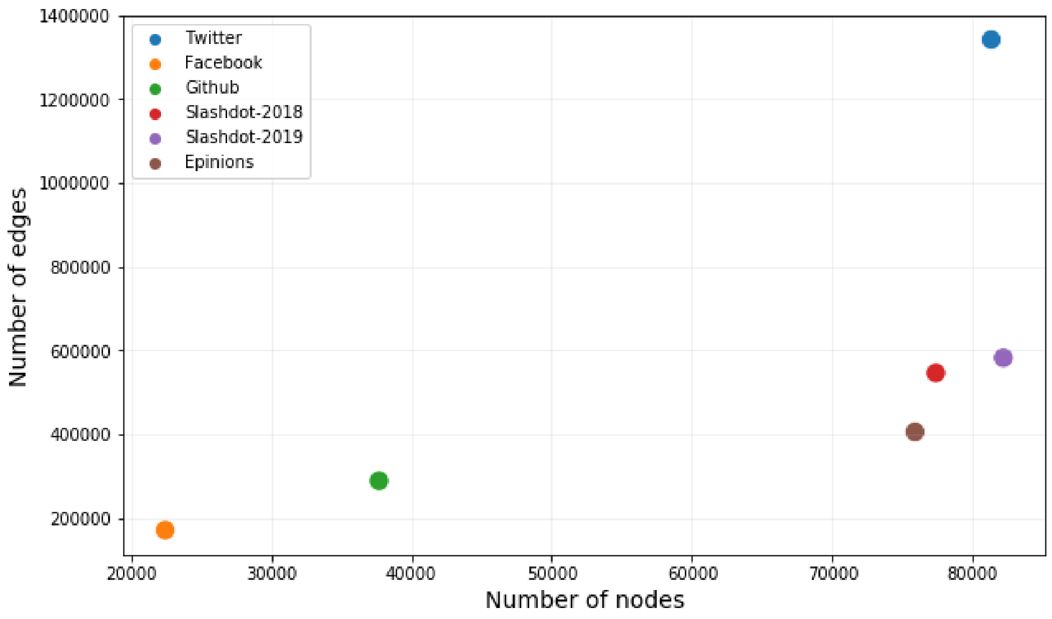


Figure 3.2: Scatter plot of real-world graphs

3.1.2.2 Artificial graphs

In this section, we propose a method for the generation of graphs that follow the Power law degree distribution of vertices. The idea is to extend the implementation of 2 libraries of the available in the networkx Python package, that is:

Powerlaw Sequence[pls] : Generates a power law sequence. Specifically, it returns a sample sequence of length n from a power law distribution with parameter α.

The API usage is: $powerlaw\_sequence(n, alpha)$

Random partition graph[rpg] : A random partition graph is a graph of communities with sizes defined by parameter *size*. Nodes in the same group are connected with probability p_in and nodes of different groups are connected with probability p_out.

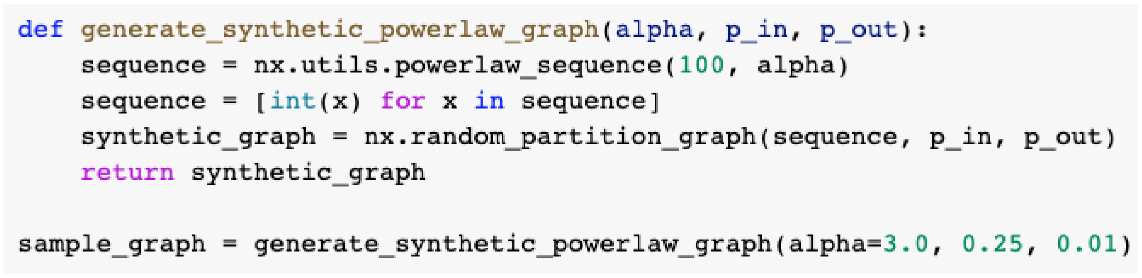
The API usage is: $random\_partition\_graph(sizes, p\_out, p\_in)$

where,

– $sizes$ (list of integers) – sizes of groups

– *p_out* (float) – probability of edges with in groups
– *p_in* (float) – probability of edges between groups

The Python code in Figure 3.3 illustrates the generation of a power law graph by combining Power law sequence generation and Random partition graphs.



```python
def generate_synthetic_powerlaw_graph(alpha, p_in, p_out):
    sequence = nx.utils.powerlaw_sequence(100, alpha)
    sequence = [int(x) for x in sequence]
    synthetic_graph = nx.random_partition_graph(sequence, p_in, p_out)
    return synthetic_graph

sample_graph = generate_synthetic_powerlaw_graph(alpha=3.0, 0.25, 0.01)
```

Figure 3.3: Python code for artificial power law graph generation

This approach for generating artificial power-law graphs should work since the group size follow Power law distribution and hence the overall degree distribution of the resulting Random partition graph should follow the same distribution by way of construction. Further, this method enables the construction of Power law graphs with different extent of Power law skew (alpha) as well as density of edges (through *p_out* and *p_in*).

The plot in Figure 3.4 displays the size (in terms of number of nodes and edges) of different artificial graphs that generated by our system. Note the range of *alpha* values of these different graphs. In Section 3.2.1, we will present the degree distributions of these graphs to ensure that they indeed follow a Power law distribution shape. Figure 3.4 and 3.5 show the statistics of various artificially generated graphs as a plot and table respectively.

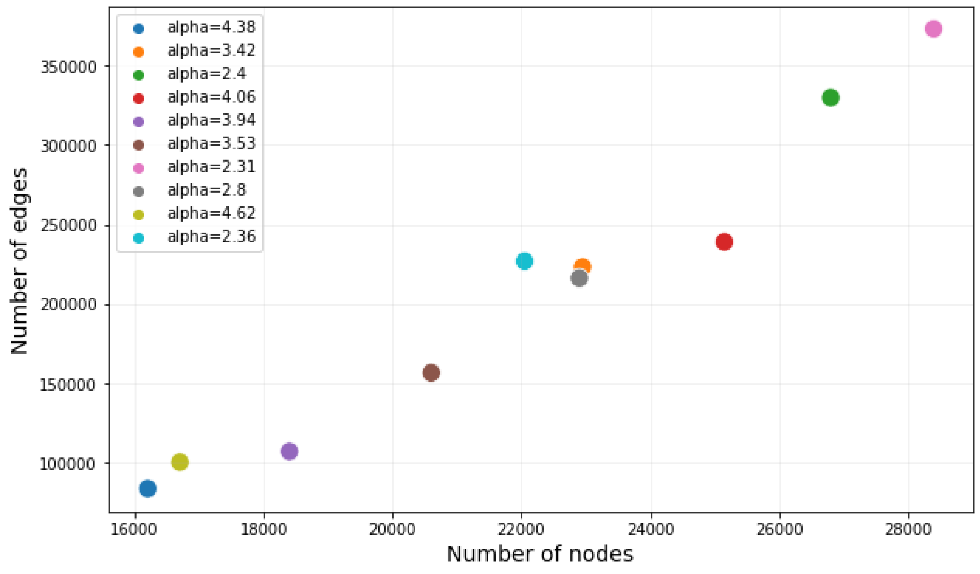
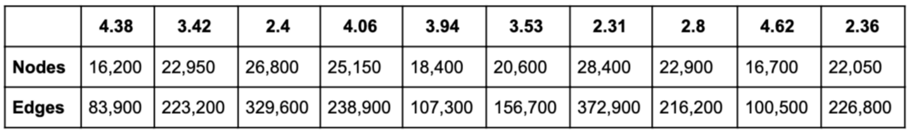


Figure 3.4: Scatter plot of artificially generated graphs

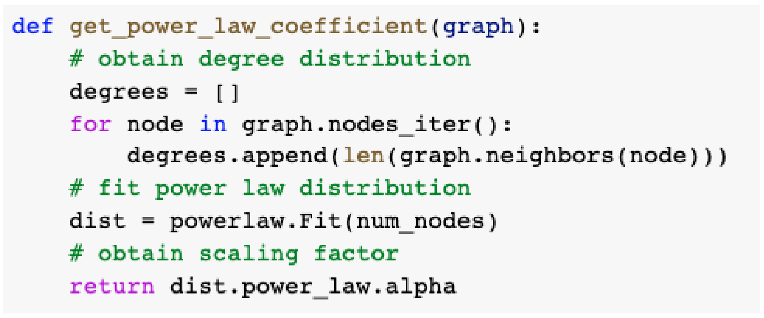


	4.38	3.42	2.4	4.06	3.94	3.53	2.31	2.8	4.62	2.36
Nodes	16,200	22,950	26,800	25,150	18,400	20,600	28,400	22,900	16,700	22,050
Edges	83,900	223,200	329,600	238,900	107,300	156,700	372,900	216,200	100,500	226,800

Figure 3.5: Dataset statistics of different real-world graphs

3.1.3 Power-Law degree distribution

There are many ways of estimating the value of the scaling exponent (*alpha*) for Power-law distribution. The most reliable techniques are often based on the method of maximum likelihood[Hanel 2017]. We utilise the maximum likelihood based approach named *powerlaw* published as a Python package [Alstott 2014]. Our Python implementation to obtain the scaling factor is provided in Figure 3.6.



```python
def get_power_law_coefficient(graph):
    # obtain degree distribution
    degrees = []
    for node in graph.nodes_iter():
        degrees.append(len(graph.neighbors(node)))
    # fit power law distribution
    dist = powerlaw.Fit(num_nodes)
    # obtain scaling factor
    return dist.power_law.alpha
```

Figure 3.6: Python code for fitting a Power law distribution to a given graph.

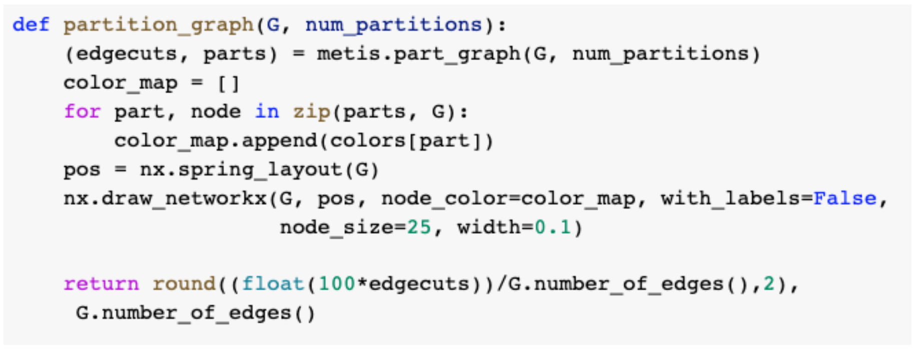
3.1.4 Graph partitioning

In this section, we outline the graph partitioning algorithm and the edge-cut metric used for our analysis on partitioning of Natural Graphs. Further, we present the methodology for visualising a partitioned graph as well as a graph sampling method that we have designed to enable visualisation of large graphs.

3.1.4.1 Partitioning strategy and Edge-cut calculation

We have implemented graph partitioning for both artificial and real-world graphs using the METIS library[Karypis 1995]. The underlying partitioning algorithms used by METIS are based on the multilevel paradigm that has been shown to produce high quality results and scale to very large problems[LaSalle 2016].

The Python code using the METIS Python package[met] is shown in the Figure 3.7. Note that the result of the partitioning is an assignment of vertices to partitions (depending on the *num_partitions* input parameter) as well as the size of the edge-cut (*edgecuts* variable). The latter is an important metric for interpreting the quality of a partitioning since a lower edge-cut minimises the network communication overhead during distributed graph processing.



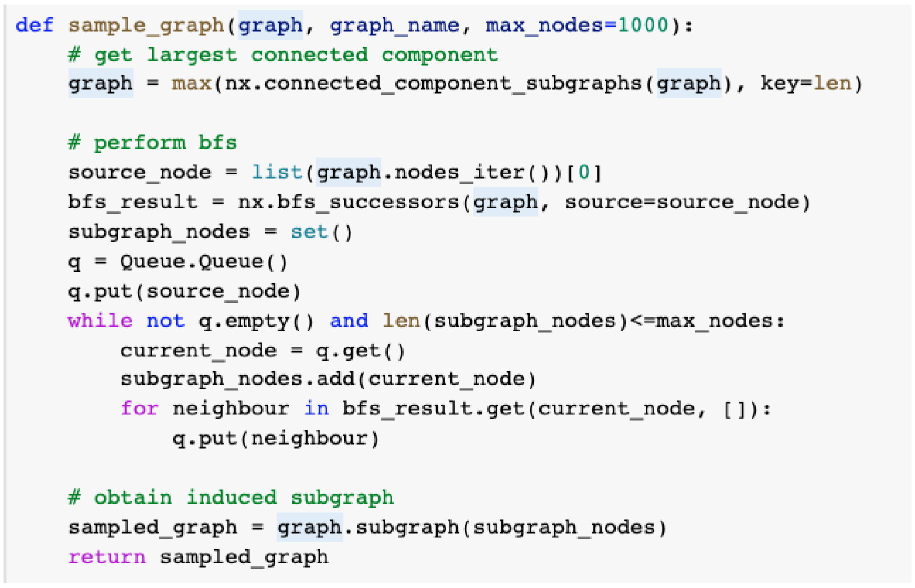
```python
def partition_graph(G, num_partitions):
    (edgecuts, parts) = metis.part_graph(G, num_partitions)
    color_map = []
    for part, node in zip(parts, G):
        color_map.append(colors[part])
    pos = nx.spring_layout(G)
    nx.draw_networkx(G, pos, node_color=color_map, with_labels=False,
                     node_size=25, width=0.1)

    return round((float(100*edgecuts))/G.number_of_edges(),2),
     G.number_of_edges()
```

Figure 3.7: Python code for graph partitioning and visualisation using METIS.

3.1.4.2 Graph sampling for visualisation

The real-world graphs used in this study have a relatively very large number of nodes (refer Figure 3.1). As a consequence, visualising such graphs(post partitioning) would not result in visually interpretable graphics due to vertex overlap. Further, attempting to visualise some of these graphs (using *networkx* libraries[dra]) results in an Out-of-Memory Error on a machine with 16GB memory for some of these big graphs. In order to solve these problems, we propose a graph sampling method specific for Natural graphs based on breadth-first-traversal of graphs. The Python implementation for this method is provided in Figure 3.8. The idea of this sampling algorithm is to capture a neighborhood of the graph starting from a random starting vertex from the largest connected component of the graph and traversing breadth-wise until a pre-defined number of nodes (*max_nodes*) of the graph are visited. The sub-graph of the original graph induced on this set of visited nodes is the resulting sampled graph.



```python
def sample_graph(graph, graph_name, max_nodes=1000):
    # get largest connected component
    graph = max(nx.connected_component_subgraphs(graph), key=len)

    # perform bfs
    source_node = list(graph.nodes_iter())[0]
    bfs_result = nx.bfs_successors(graph, source=source_node)
    subgraph_nodes = set()
    q = Queue.Queue()
    q.put(source_node)
    while not q.empty() and len(subgraph_nodes)<=max_nodes:
        current_node = q.get()
        subgraph_nodes.add(current_node)
        for neighbour in bfs_result.get(current_node, []):
            q.put(neighbour)

    # obtain induced subgraph
    sampled_graph = graph.subgraph(subgraph_nodes)
    return sampled_graph
```

Figure 3.8: Python code for sampling Natural graphs using breadth-first traversal.

The method for visualising the (sampled) partitioned graph in different colors is performed using the *draw_networkx* method available for *networkx* graphs[dra]. Example visualisations of different partitioned (real-world and artificial) power law graphs are provided in Section 3.2.2.2.

3.1.5 Graph processing algorithms

In this section, we will describe the two graph processing algorithms that we implemented in GraphX and ran on the above graphs, both for real-world and artificially generated graphs.

3.1.5.1 PageRank algorithm

PageRank is an algorithm that measures the transitive influence or connectivity of nodes. It can be computed by either iteratively distributing one node's rank (originally based on degree) over its neighbours or by randomly traversing the graph and counting the frequency of hitting each node during these walks.

It counts the number, and quality, of links to a page which determines an estimation of how important the page is. The underlying assumption is that pages of importance are more likely to receive a higher volume of links from other pages.

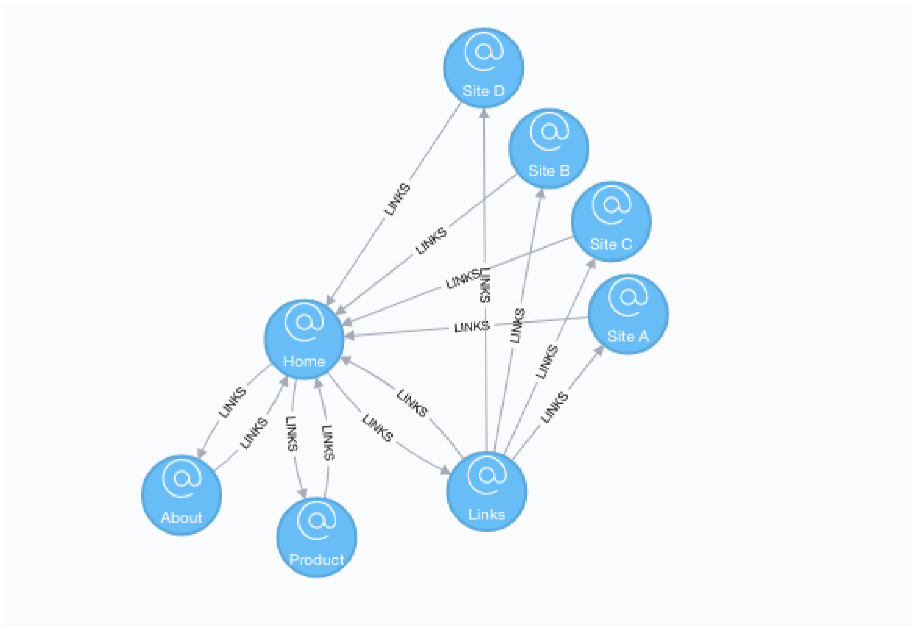


Figure 3.9: Sample graph linking website pages

PageRank is defined in the original Google paper[Page 1999] as follows:

$PR(A) = (1 - d) + d(PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))$

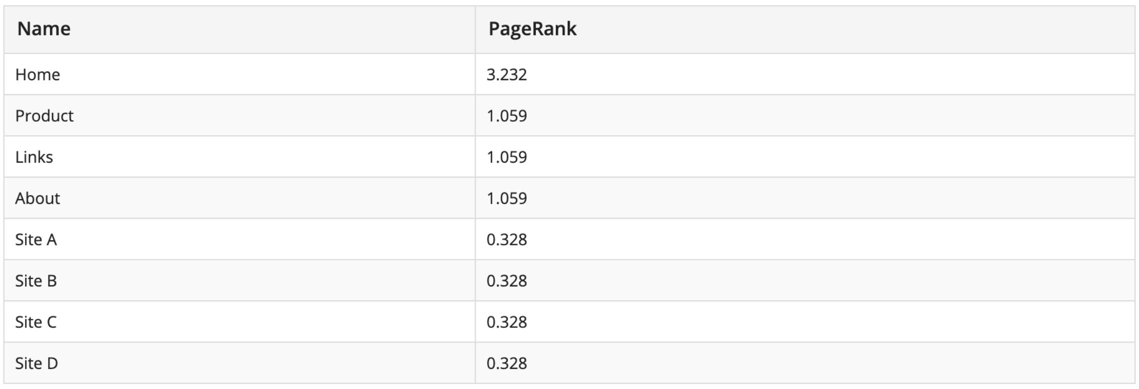
where,

- we assume that a page A has pages $T1$ to Tn which point to it (i.e., are citations).
- d is a damping factor which can be set between 0 and 1. It is usually set to 0.85.
- $C(A)$ is defined as the number of links going out of page A.

We will explain the concept of calculating the PageRank values with the help of an example.

Consider the Figure 3.9 simple graph linking different website pages.

The result after running the page rank algorithm on the above graph is shown below 3.10. As we might expect, the *Home* page has the highest PageRank because it has incoming links from all other pages. We can also see that it's not only the number of incoming links that is important, but also the importance of the pages behind those links.



Name	PageRank
Home	3.232
Product	1.059
Links	1.059
About	1.059
Site A	0.328
Site B	0.328
Site C	0.328
Site D	0.328

Figure 3.10: PageRank algorithm results

3.1.5.2 Triangle Counting

Triangle counting is a community detection graph algorithm that is used to determine the number of triangles passing through each node in the graph. A triangle is a set of three nodes, where each node has a relationship to all other nodes. It can also be used to determine the stability of a graph, and is often used as part of the computation of network indices, such as the clustering coefficient.

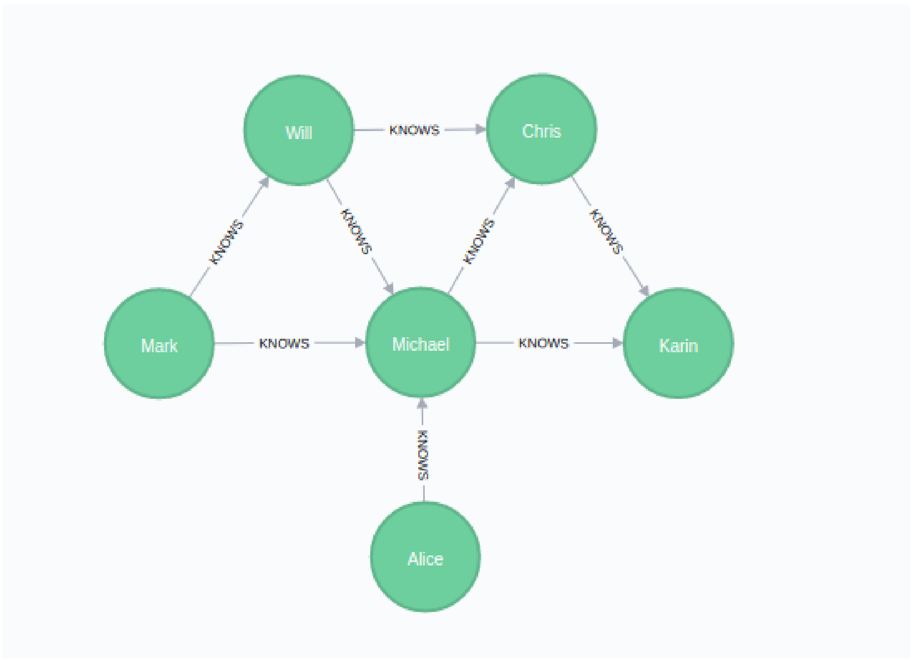
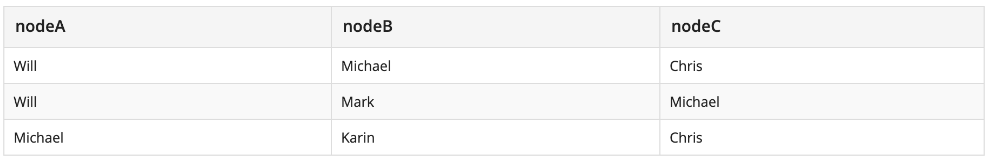


Figure 3.11: Sample graph linking people who knows each other

We will explain the concept of calculating the Triangle counting values with the help of an example. Consider Figure 3.11, a simple graph linking different website pages. The result after running the page rank algorithm on the above graph is shown in Figure 3.12. We can see that there are KNOWS triangles containing "Will, Michael, and Chris", "Will, Mark, and Michael", and "Michael, Karin, and Chris". This means that everybody in the triangle knows each other.



nodeA	nodeB	nodeC
Will	Michael	Chris
Will	Mark	Michael
Michael	Karin	Chris

Figure 3.12: Triangle counting algorithm results

3.2 Results

In this section, we provide the results and graphs obtained from various experiments with methodology outlined in the previous section.

3.2.1 Power-Law degree distribution

In Figure 3.13 and 3.14, we plot the degree distribution of real-world and artificially generated Natural graphs respectively. It is evident that the real-world graphs have the power law degree distribution. Further, the artificially generated Natural graphs also have degree distribution similar to a power-law distribution.

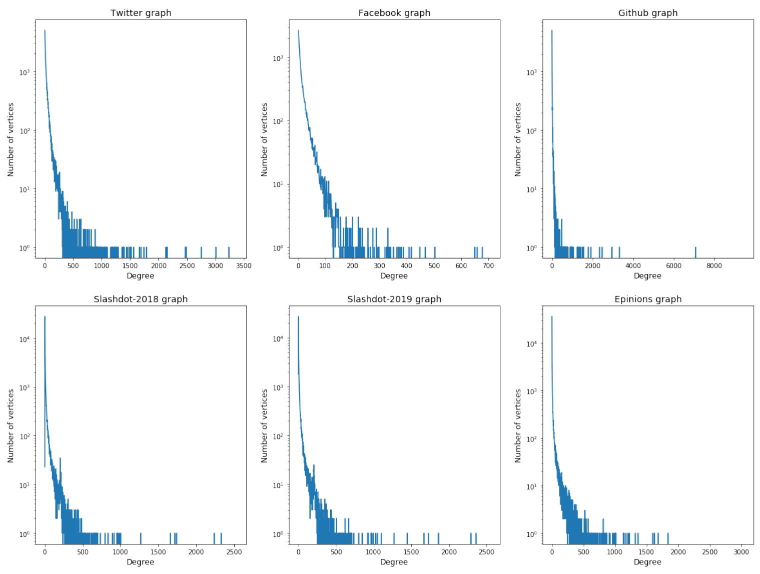


Figure 3.13: Degree distribution for real-world Natural Graphs

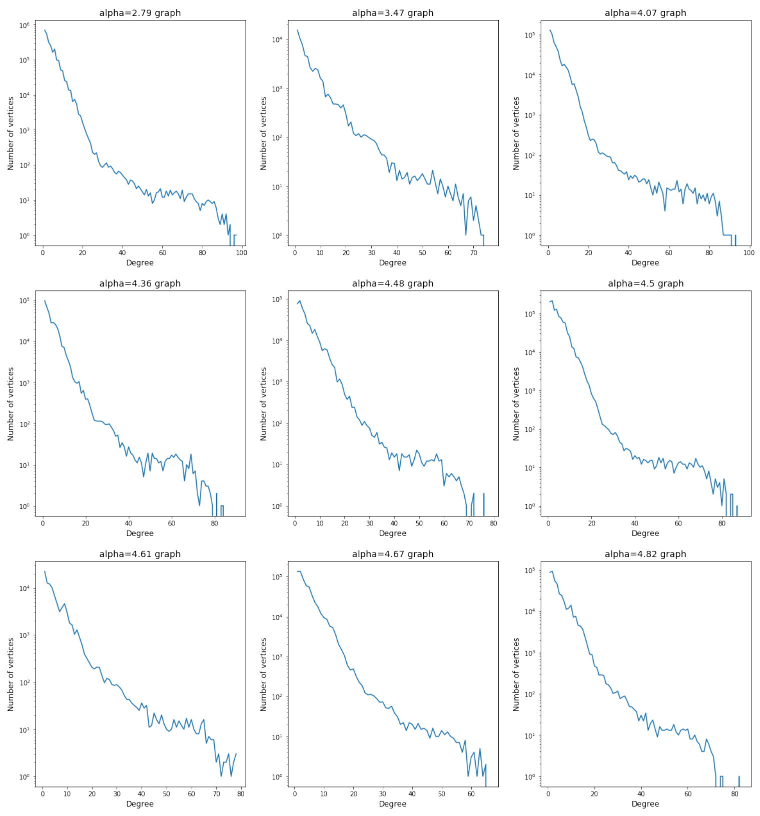


Figure 3.14: Degree distribution for artificially generated Natural Graphs

3.2.2 Graph partitioning

In this section, we evaluate graph partitioning of different graphs using the edge-cut metric. Also, we evaluate our method for sampling Natural graphs to aid visualisation.

3.2.2.1 Edge-cut evaluation

Figure 3.15 and 3.16 display the variation of edge-cut as well as number of edges in a graph with *alpha* for real-world and artificially generated Natural graphs. No strong pattern emerges from these graphs indicating the inherent topological complexity of Natural graphs.

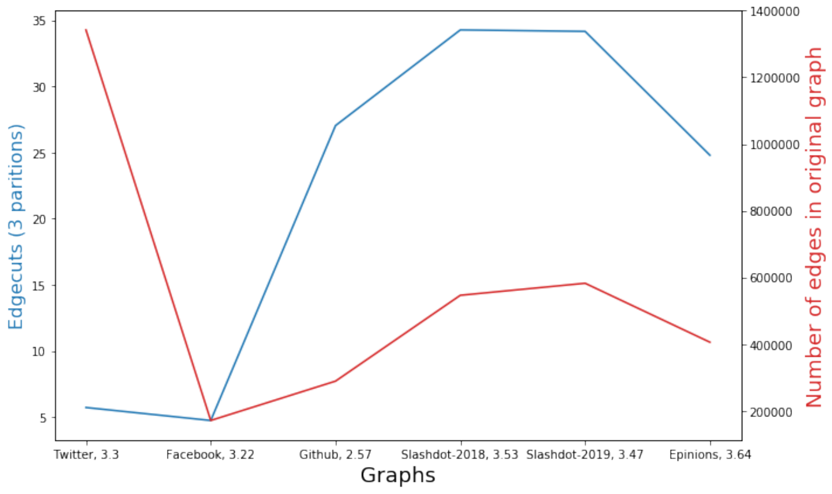


Figure 3.15: Edgecut and Number of Edges - Real-world Graphs

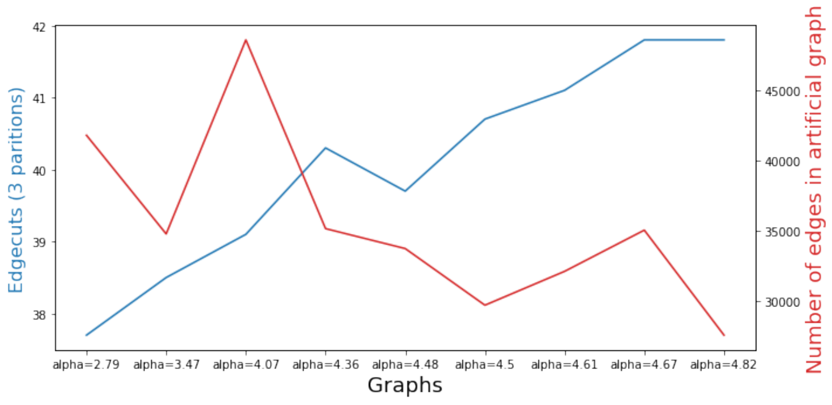
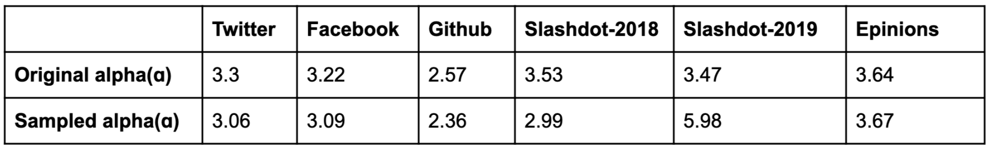


Figure 3.16: Edgecut and Number of Edges - Artificial Graphs

3.2.2.2 Graph sampling for visualisation

The Power law factor, *alpha* for original real-world graphs is compared to the corresponding sampled graphs. Figure 3.17 displays these exact *alpha* values while Figure 3.18 provides a visual comparison of original and sampled Power-law degree distributions.



	Twitter	Facebook	Github	Slashdot-2018	Slashdot-2019	Epinions
Original alpha(α)	3.3	3.22	2.57	3.53	3.47	3.64
Sampled alpha(α)	3.06	3.09	2.36	2.99	5.98	3.67

Figure 3.17: Power law distribution for original and sampled real-world Natural Graphs

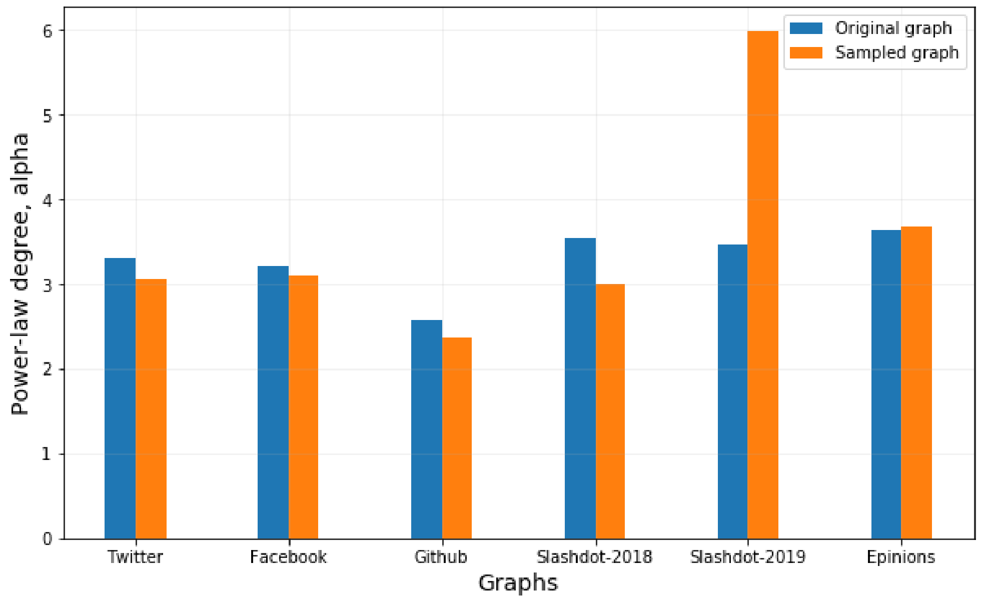


Figure 3.18: Comparing Power law distribution after sampling real-world Natural Graphs

In Figure 3.19, we visualise the sampled real-world Natural graphs. These can be compared to the visualisations of artificially generated natural graphs in Figure 3.20.

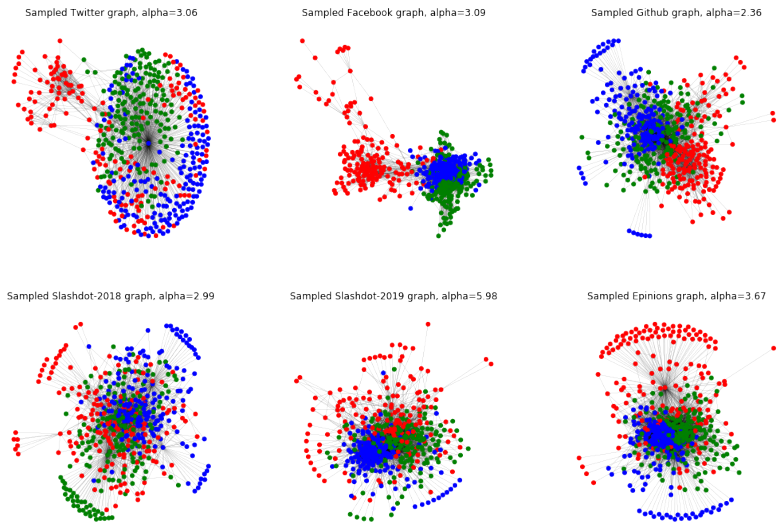

Sampled Twitter graph, alpha=3.06
Sampled Facebook graph, alpha=3.09
Sampled Github graph, alpha=2.36
Sampled Slashdot-2018 graph, alpha=2.99
Sampled Slashdot-2019 graph, alpha=5.98
Sampled Epinions graph, alpha=3.67

Figure 3.19: Visualisations of sampled real-world Natural graphs

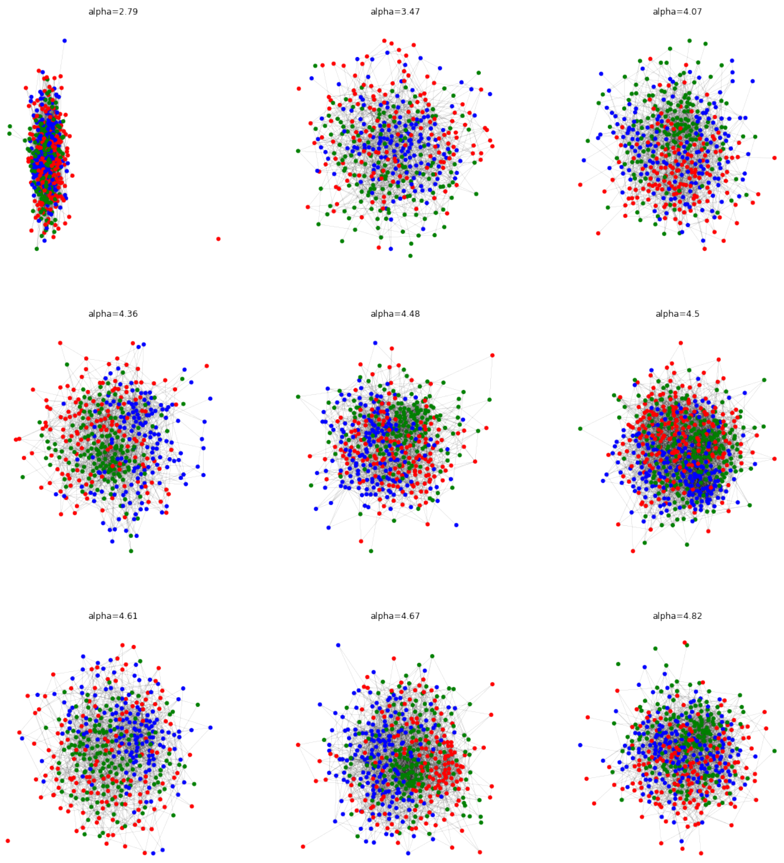


Figure 3.20: Visualisations of artificially generated Natural graphs

3.2.3 Graph processing algorithms

In this section, we present the results obtained after running the Page rank and Triangle counting algorithms for both real-world and artificially generated graphs. The idea is to benchmark, how close does the artificially generated graphs resemble with values and distribution with the similar α values of their counterpart real-world graphs.

3.2.3.1 Page rank distribution

The idea of plotting page rank distribution is to see if there are some vertices that have very high page rank, usually the high degree vertices in a natural graph should have higher page rank values. From Figures 3.21 and 3.22, it is evident that all the graphs either real-world or artificially generated have some or few vertices with very page rank values. Also, the artificially generated graphs have the same page rank distributions as the real-world graphs.

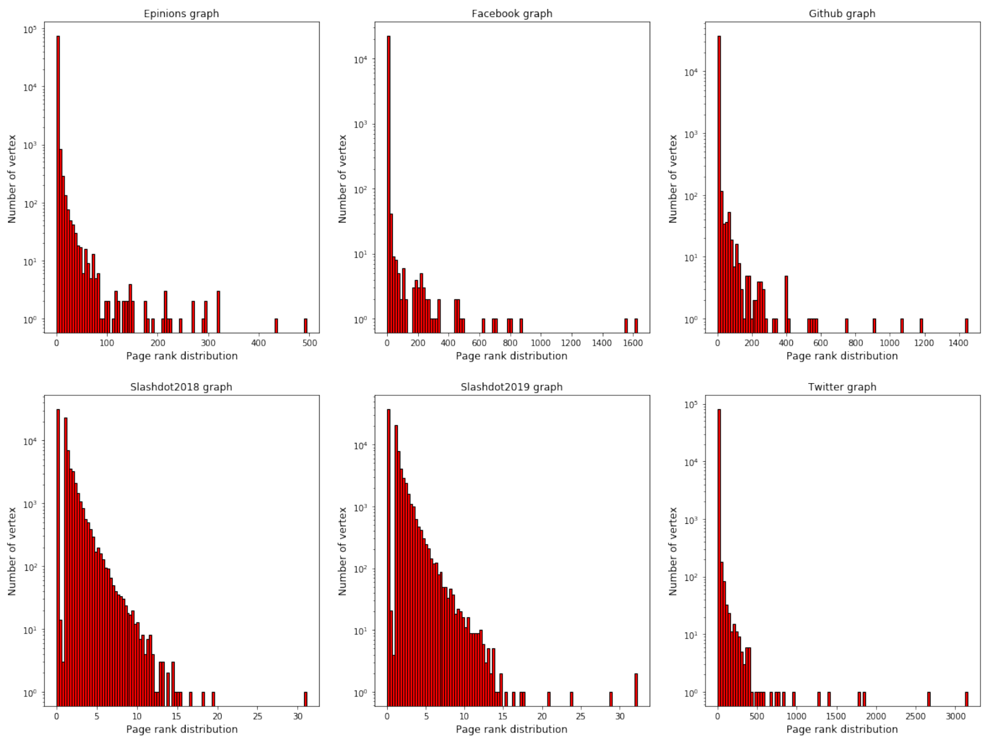


Figure 3.21: Page rank distribution of real-world graphs

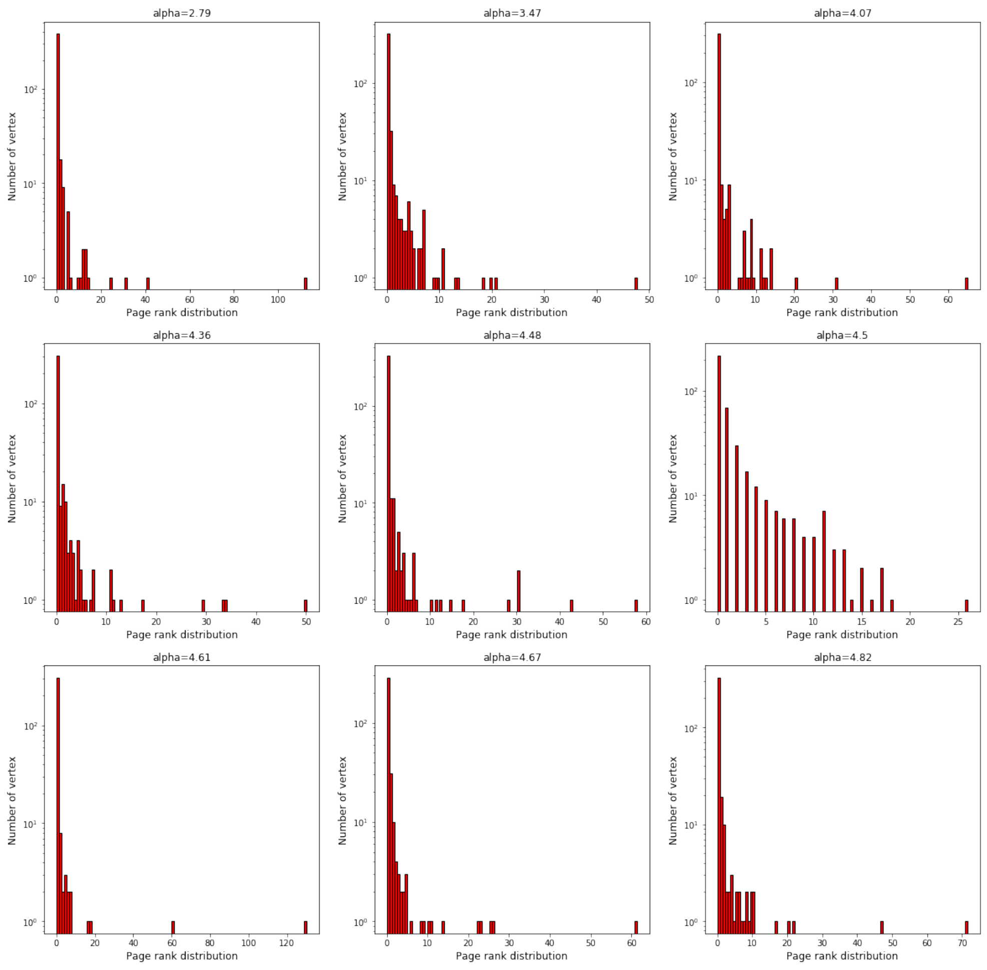


Figure 3.22: Page rank distribution of artificially generated graphs

3.2.3.2 Triangle counting distribution

The idea of plotting triangle count distribution is to see if there are some high degree vertices that are present in almost every triangle count and hence, higher triangle count distribution.

From the Figures 3.23 and 3.24, it is evident that all the graphs either real-world or artificially generated have some or few vertices with very triangle count values. But, the artificially generated graphs are not performing that well for triangle count values. This is because there is not a few high-degree vertices but, rather a lot of vertices who have similar degrees and hence, all the triangle counts are well distributed.

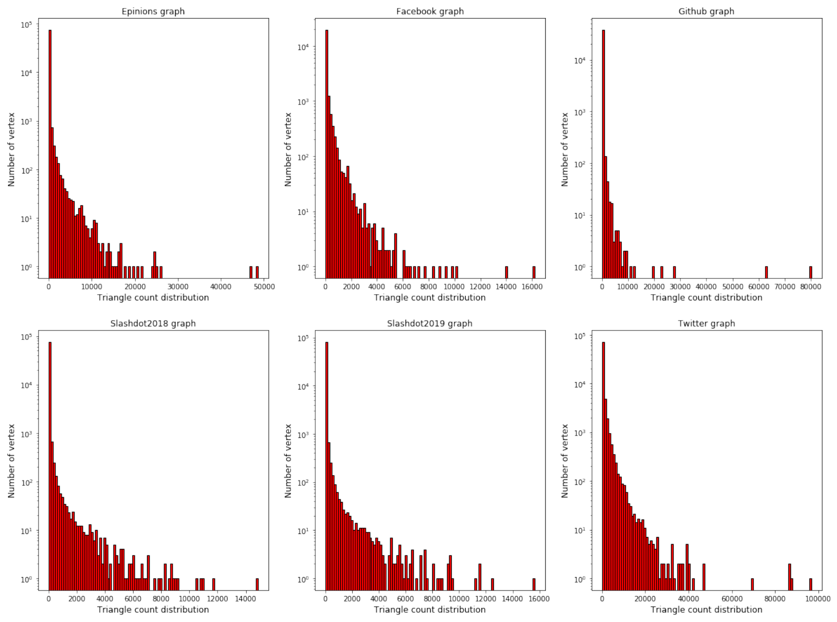


Figure 3.23: Triangle counting distribution of real-world graphs

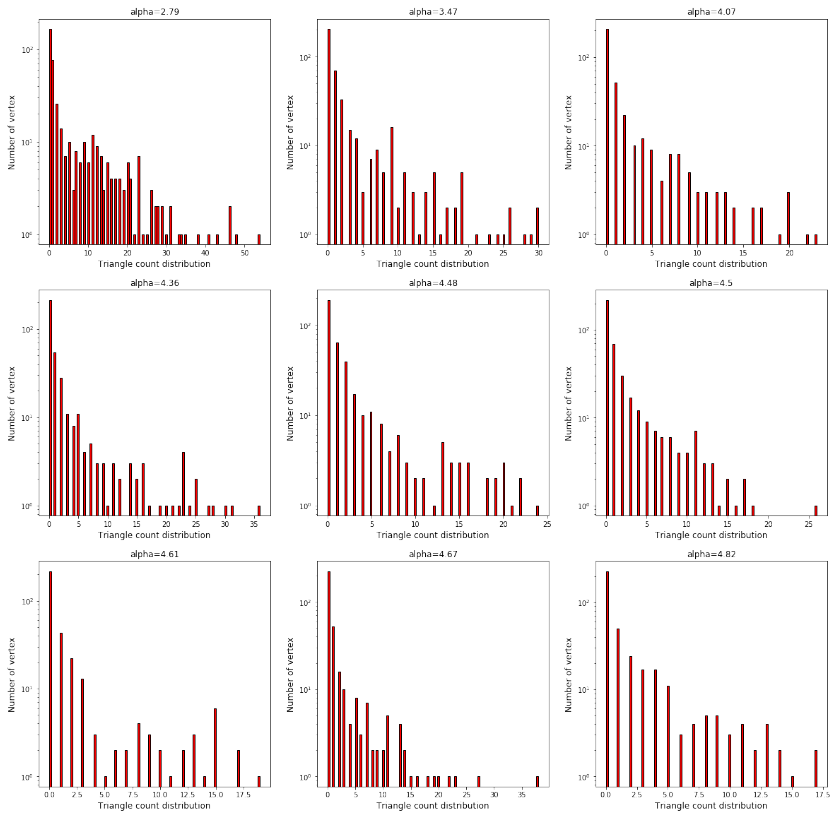


Figure 3.24: Triangle counting distribution of artificially generated graphs

3.3 Conclusions and Future Work

Natural Graphs represent an important class of graph topology given their presence in real-world graph dataset (particularly social network datasets). As a result, it is important to develop a set of methods to study and analyse Natural graphs to uncover their properties experimentally. In this work, we have developed benchmarks involving the pipeline for studying Natural graphs. Precisely, we have have proposed and/or developed systems for:

- testing for Natural graphs through scaling factor calculation and degree distribution plots
- artificially generating Natural graphs of different scaling factors of degree distribution as well as edge-density
- partitioning Natural graphs (we provide implementation with METIS it can be replaced by any available graph partitioning algorithms)
- visualisation of partitioned Natural graphs
- a sampling method for visualising large Natural Graphs
- and, running graph processing algorithms on Natural Graphs on GraphX.

We have not implemented any statistical measure to indicate the confidence that a particular degree distribution fits a Power Law Distribution. This analysis in our case is limited to a visual inspection of the degree distribution plot. In the future, we would like to develop more robust and automatic tests for Power law degree distribution (and hence Natural Graphs). This could be done through a loss function between the observed data and the best fit power law distribution normalised by the graph size. Further, we were unable to benchmark graph processing algorithms in a distributed environment due to lack of the requisite computational infrastructure. In the future, benchmarking of distributed graph processing can be added (independently) at the end of the pipeline to make it more complete.

Bibliography

[Alstott 2014] Jeff Alstott and Dietmar Plenz Bullmore. powerlaw: a Python package for analysis of heavy-tailed distributions. PloS one, vol. 9, no. 1, 2014. (Cited on page 17.)

[Ammar 2018] Khaled Ammar and M Tamer Özsu. Experimental analysis of distributed graph systems. Proceedings of the VLDB Endowment, vol. 11, no. 10, pages 1151–1164, 2018. (Cited on page 9.)

[Angles 2012] Renzo Angles. A comparison of current graph database models. In 2012 IEEE 28th International Conference on Data Engineering Workshops, pages 171–177. IEEE, 2012. (Cited on page 2.)

[dra] drawnetworkX doc. `https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html`. (Cited on pages 18 and 19.)

[Gonzalez 2012] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12), pages 17–30, 2012. (Cited on page 7.)

[gra] GraphX official website. `https://spark.apache.org/graphx/`. (Cited on page 13.)

[Hanel 2017] Rudolf Hanel, Bernat Corominas-Murtra, Bo Liu and Stefan Thurner. Fitting power-laws in empirical data with estimators that work for all exponents. PloS one, vol. 12, no. 2, 2017. (Cited on page 17.)

[Karypis 1995] George Karypis and Vipin Kumar. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995. (Cited on page 17.)

[Kim 2008] Jongkwang Kim and Thomas Wilhelm. What is a complex graph? Physica A: Statistical Mechanics and its Applications, vol. 387, no. 11, pages 2637–2652, 2008. (Cited on page 8.)

[LaSalle 2016] Dominique LaSalle and George Karypis. A parallel hill-climbing refinement algorithm for graph partitioning. In 2016 45th International Conference on Parallel Processing (ICPP), pages 236–241. IEEE, 2016. (Cited on page 17.)

[Leskovec 2009] Jure Leskovec, Kevin J Lang, Anirban Dasgupta and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics, vol. 6, no. 1, pages 29–123, 2009. (Cited on page 14.)

[Leskovec 2012] Jure Leskovec and Julian J Mcauley. Learning to discover social circles in ego networks. In Advances in neural information processing systems, pages 539–547, 2012. (Cited on pages 13 and 14.)

[Low 2012] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment, vol. 5, no. 8, pages 716–727, 2012. (Cited on page 6.)

[Malewicz 2010] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 135–146. ACM, 2010. (Cited on page 5.)

[Martella 2017] Claudio Martella, Dionysios Logothetis, Andreas Loukas and Georgos Siganos. Spinner: Scalable graph partitioning in the cloud. In 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pages 1083–1094. Ieee, 2017. (Cited on page 10.)

[met] NetworkX-Metis docs. `https://metis.readthedocs.io/en/latest/_modules/metis.html`. (Cited on pages 13 and 17.)

[Meyerhenke 2017] Henning Meyerhenke, Peter Sanders and Christian Schulz. Parallel graph partitioning for complex networks. IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 9, pages 2625–2638, 2017. (Cited on page 11.)

[Page 1999] Lawrence Page, Sergey Brin, Rajeev Motwani and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Rapport technique, Stanford InfoLab, 1999. (Cited on page 20.)

[pls] Power law seq doc. `https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.utils.random_sequence.powerlaw_sequence.html`. (Cited on page 15.)

[Rozemberczki 2019] Benedek Rozemberczki, Carl Allen and Rik Sarkar. Multi-scale Attributed Node Embedding, 2019. (Cited on page 14.)

[rpg] Random partition graph doc. `https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.generators.community.random_partition_graph.html#networkx.generators.community.random_partition_graph`. (Cited on page 15.)

[Ugander 2013] Johan Ugander and Lars Backstrom. Balanced label propagation for partitioning massive graphs. In Proceedings of the sixth ACM international conference on Web search and data mining, pages 507–516. ACM, 2013. (Cited on page 10.)