DOMINATION NUMBER WITHIN ON-LINE SOCIAL NETWORKS

by

Marc Lozier

Bachelor of Engineering in Computer Engineering,

Ryerson University, 2013

A thesis presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Applied Mathematics

Toronto, Ontario, Canada, 2015

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Domination Number within On-line Social Networks

Master of Science, 2015

Marc Lozier

Applied Mathematics, Ryerson University

Abstract

There is particular interest in on-line social networks (OSNs) and capturing their properties. The memoryless geometric protean (MGEO-P) model provably simulated many OSN properties. We investigated dominating sets in OSNs and their models. The domination numbers were computed using two algorithms, DS-DC and DS-RAI, for MGEO-P samples and Facebook data, known as the Facebook 100 graphs. We establish sub-linear bounds on the domination numbers for the Facebook 100 graphs, and show that these bounds correlate well with bounds in graphs simulated by MGEO-P.

A new model is introduced known as the Distance MGEO-P (DMGEO-P) model. This model incorporates geometric distance to influence the probability that two nodes are adjacent. Domination number upper bounds were found to be well-correlated with the Facebook 100 graphs.

Acknowledgements

I would first like to take this opportunity to extend my thanks and obligation to those people whose help and active guidance were critical in the completion of the thesis.

I am foremost indebted to my supervisor Dr. Anthony Bonato for the wealth of knowledge he has provided. His support and guidance has been invaluable to both my research and future endeavors.

I would like to offer my profound thanks to Dr. Dejan Delic and Dr. Laleh Samarbaksh for their participation as my defense committee. I would also like to thank Dr. Sebastian Ferrando for encouraging me to pursuit my Masters in Applied Mathematics. Collectively, I would like to thank the entire Department of Mathematics at Ryerson University for the great experience as a Masters student and Graduate Assistant.

Lastly, I would like to thank my family — my mother and father Judy and Michel, and my brothers Matthew and Michael — and my friends for their support, encouragement and understanding.

Contents

List of Figures

CHAPTER 1

Introduction

1.1. Motivation

Over the last decade, research on on-line social networks (or OSNs), such as Facebook and Twitter, has been increasing within the network science community. There is particular interest in these networks and capturing their properties. A few of the commonly observed properties of OSNs and other complex networks are power-law degree distributions [**5**] and the small world property [**44**] (constant or shrinking diameter with network size and high local clustering) [**25**]. These properties will be defined in Section 1.3. Graph theorists have the challenge to come up with a model that can mimic these properties asymptotically with high probability, and which correlate well with real-world data sets. Modelling allows mathematicians to uncover the underlying mechanisms behind the network, predict its future, and reveal any hidden reality of the network; see [**8**].

Graphs capture relationships between objects. A graph is called a *network* when it is represented by real-world interactions, such as the web graph, biological networks, or social networks. The friendship graph, collaboration graph and actor graphs are examples of social networks. These graphs are structured differently when compared to the web graph: the web graph is organized by link structure, while social networks are governed by social

interactions between people. On-line networks have their data more readily accessible and measurable than offline social networks (where the data is noisier and harder to gather). Hence, there is an increasing interest for a rigorous model to capture OSN's evolutionary properties. However, some models are more difficult to analyze than others. While some may fit naturally and realistically with real world data, they are often challenging to analyze.

1.2. Graph Theory

Some knowledge of graph theory is necessary before discussing OSNs and their models. A *graph* G consists of a non-empty set of *vertices*, denoted $E(G)$, and an *edge set*, denoted $E(G)$, which consists of pairs of vertices. A graph is *directed* if each edge has an orientation. Otherwise, it is known as an *undirected* graph. When two nodes are joined by an edge, they are *adjacent* to each other or called *neighbours*. For simplicity, we will only consider *simple* graphs (see Figure 1.1): graphs that do not contain loops or multiple edges. The *order* of a graph is its number of vertices (that is, $|V(G)|$), while its *size* is the number of edges (that is, $|E(G)|$). In OSNs, vertices correspond to people, and edges correspond to their friendship relationships.

A *subgraph* of graph G is a graph H where $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. An *induced subgraph* of graph G is a graph S, where $V(S) \subseteq V(G)$ and each vertex is adjacent to each other only if they are adjacent in G. A *path* is said to exist between two distinct vertices when there is a sequence of edges that connect a sequence of non-repeated vertices to them. A graph is

connected when for each pair of distinct vertices, there exist a path between them.

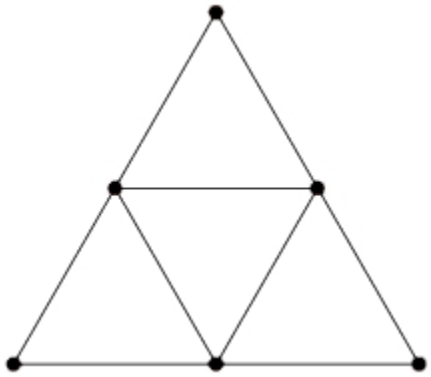


FIGURE 1.1. An example of a simple graph.

The *neighbour set* of a given vertex u, denoted by $N(u)$, consists of the set of vertices adjacent to u. The degree of a vertex, written $\deg(u)$, is the cardinality of its neighbour set $|N(u)|$. The minimum degree of a graph G is denoted as $\delta(G)$. A familiar global measure for distances in a graph is its *diameter*. The diameter of a graph G, written $\mathrm{diam}(G)$, is the length of the longest shortest path. The *distance* between two vertices is the length of the shortest path between them, known as $d(u, v)$.

1.2.1. Dominating Sets. Dominating sets in graphs were first introduced by Ore in 1962 [**37**]. A *dominating set* $S \subseteq V(G)$ has the property that each vertex not in S is adjacent to some vertex in S. The *domination number*, written $\gamma(G)$, is the minimal cardinality of a dominating set. It is difficult to compute the exact value of $\gamma(G)$ when given a large order graph G. In particular, computing the domination number is known to be a **NP**-complete problem [**19**]. In order to obtain a value for $\gamma(G)$, heuristic algorithms are

used. One proof for the upper bounds of the domination number of any given graph was given using the probabilistic method by Alon and Spencer in [**3**], stated as Theorem 1.1. Figure 1.2 shows an example of a dominating set.

THEOREM 1.1. [**3**] *If given any graph G with a minimum degree $\delta > 1$, then*

$$\gamma(G) \leq n\left(\frac{1+\log(\delta+1)}{\delta+1}\right). \tag{1.1}$$

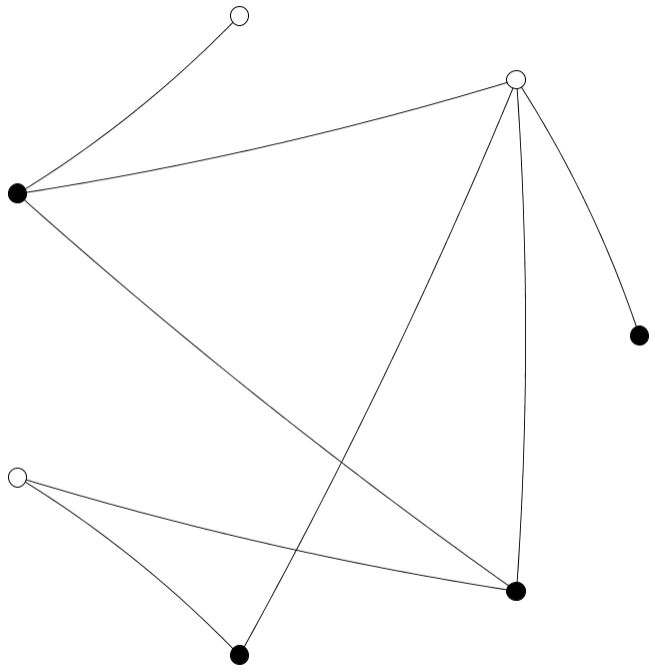
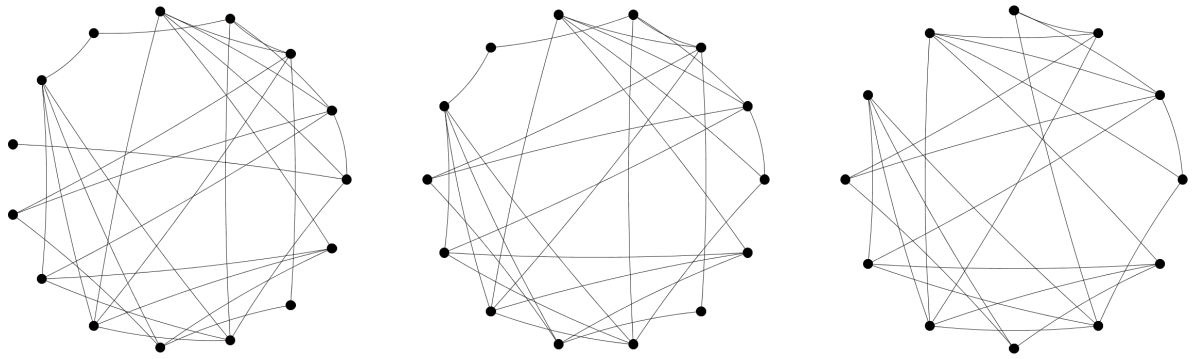


FIGURE 1.2. A graph and one of its dominating sets (represented by the white vertices).

The domination set is quite sensitive to vertices with low degrees. During simulations, the *k-cores* (where $1 \leq k \leq 5$) for the graphs were used to test

how sensitive their domination sets were. The *k-core* of a graph is the largest induced subgraph in which its minimum degree is at least k. Figure 1.3 shows an example of the 1-, 2-, and 3-core of a graph.



1-core 2-core 3-core

FIGURE 1.3. The 1-core, 2-core and 3-core of a random graph.

There were two main algorithms used to provide results. Additional algorithms were used but not described in detailed in the thesis (as they were outperformed by other algorithms). Some of these include DS-RANDOM (adding random vertices to the dominating set), DS-GREEDY (adding the highest degree vertices to the dominating set recursively until entire graph is covered), and DS-RANK (adding vertices with the highest rank recursively, MGEO-P model only).

The first main algorithm used in the thesis is known as DS-RAI. It is the fastest performing algorithm based on heuristic evidence from our experiments and provides a good upper bound (but not the best in our experiments). The algorithm works as described in Algorithm 1 (where {} denotes an empty set). The second main algorithm used is known as DS-DC. Despite

```
INPUT: Graph data - set of nodes V with degrees;
OUTPUT: Dominating set array - DS;
initialization: DS = {}, coveredSet = {}, sort nodes V in descending order by degree;
while not at end of V do
    v ∈ V;
    if v is not in coveredSet then
        add v to DS;
        add v to coveredSet;
        add neighbours of v to coveredSet;
    end
end
```

Algorithm 1: Dominating set algorithm DS-RAI from [**39**].

it performing slower than DS-RAI, it provides a better upper bound. The algorithm is stated in Algorithm 2.

```
INPUT: Graph data - set of nodes V with degrees;
OUTPUT: Dominating set array - DS;
initialization: DS = V, sort nodes V in ascending order by degree;
while not at end of V do
    v ∈ V;
    remove v from DS;
    if DS is not dominating then
        add v to DS;
    end
end
```

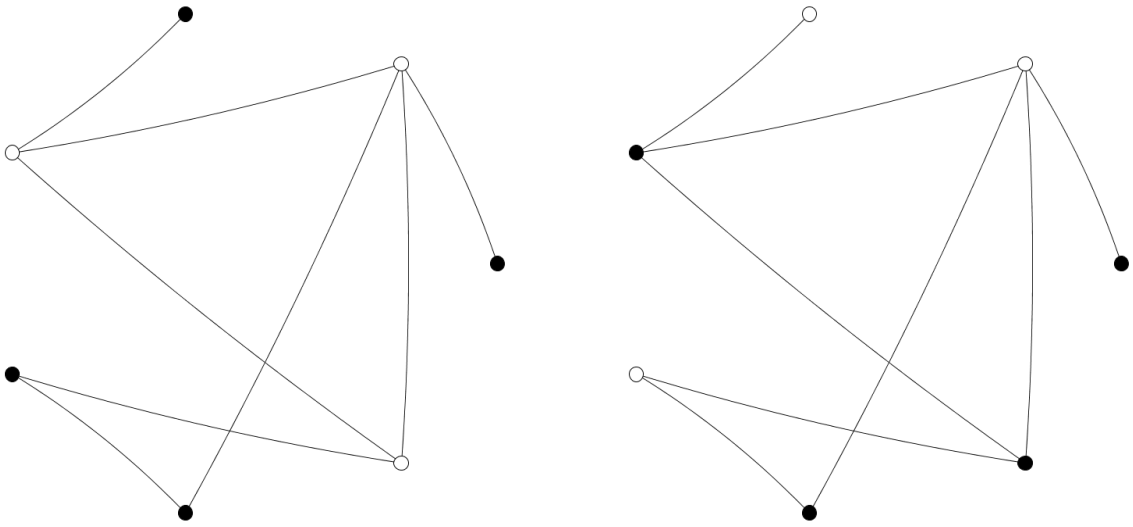
Algorithm 2: Dominating set algorithm DS-DC from [**33**].

1.3. Properties of On-line Social Networks

Emergent topological properties have been observed with the available real world data from large-scale on-line social networks. Some observed properties of OSNs are the following.

(1) *Large scale and on-line.*

OSNs are massive networks that have their order (number of users) and size (number of relationships) in the millions, even billions. There



DS-DC Algorithm DS-RAI Algorithm

FIGURE 1.4. Dominating sets generated by the algorithms DS-DC and DS-RAI.

also exist some users with abnormally high degrees. The order of these networks is also dynamic: users are being removed and added all the time (this is known as the *on-line property*).

(2) *Small world property and shrinking distance.*

A famous chain letter experiment carried out by Stanley Milgram in 1967 [**34**] examined the average path length for the social networks of people in the United States. The experiment consisted of giving several letters to random people in different states addressed to a Boston stockbroker. The person would then be asked to send the letter onto other people whom they think who might know the stockbroker. Although only 20% of the letters were able to reached the stockbroker, on average, they took 6 steps through the social network. This was the beginning of the concept of the small world structure within networks.

The result of this experiment is known as the *six degrees of separation*. It is the theory that it is possible for a connection to be made between highly disparate social populations in a small number of steps. After many years, the small-world property is well-studied, and appears in the literature of many disciplines such as sociology, biology, physics, and information technology; see Chapter 2 of [**8**].

The study of small-world property within complex networks was first introduced by Watts and Strogatz in 1998 [**44**]. Networks with this property were noted to have a low diameter of $O(\log n)$ (or average distance of $O(\log \log n)$) and a high clustering coefficient (higher than a random binomial distributed graph with the same number of nodes and average degree). A more mathematical definition of clustering coefficient and random graphs will be presented in the next section. In the papers [**1, 2, 23, 26, 35**], the small-world property have been observed (in Cyworld, Flickr, LiveJournal, MySpace, Orkut, Twitter and YouTube). Twitter has been observed to have a high clustering coefficient and a low diameter of 6 in [**23, 26**]. In addition, it has been found in Cyworld [**2**], Flickr and Yahoo!360 [**25**] that their diameter shrinks over time.

(3) *Power-law degree distributions.*

Power laws were first studied in 1896 in Pareto's work [**38**] on income distribution. The power law degree distributions represents an *undemocratic* nature. There will be many people with low number of

friends and a few people with a significantly higher number of friends. It has been reported that the power law in- and out-degree distribution exist for Flickr, LiveJournal, Orkut, YouTube [**35**] and Twitter [**23**]. Golder et al. [**20**] analyzed 4.2 million users from Facebook and found that it follows a power law. In addition, the degree distributions amongst the social networks studied in [**35**] were found to be similar. A definition of power law degree distributions will be given in Subsection 1.3.1.

(4) *Densification power law.*

This occurs when the average degree of the network grows to infinity as the order of the network increases. A graph G follows a *densification power law* if there exist a constant $a \in (1, 2)$ such that $|E(G)|$ is proportional to $|V(G)|^a$. In [**2**], it was documented that Cyworld (a South Korean social network) exhibit this densification power law.

(5) *Bad spectral expansion.*

OSNs are often divided into clusters. A common trait about them is that they contain communities, which is a characteristic of social organization [**36**]. In these tightly knit clusters, the number of links within them are significantly higher than the links between other groups or clusters. Because of this structure, these networks have bad spectral expansion properties due to small gaps between the first and second eigenvalues of their adjacency matrices [**18**].

(6) *Component structure.*

In [**25**], Kumar et al. classified users of Flickr and Yahoo!360 into three categories: singletons (isolated nodes), the giant component (dense core of low (shrinking) diameter) and the middle region (isolated communities with star-like structure).

Two main properties that are particularly interesting are the *power-law degree distributions* and *small world-property.* A detailed description of each follows.

1.3.1. Power Law Degree Distribution. Let G be a graph and k a positive integer. The *number of vertices of degree k in G* is defined as:

$$N_{k,G} = |\{x \in V(G) : \deg_G(x) = k\}|.$$

Since real-world complex networks are dynamic, for simplicity, we let $|V(G)| = t$ (this allows us to remember that the number of vertices within a system changes with time). A graph follows a *power-law degree distribution* if each degree k, $\frac{N_{k,G}}{t} \sim k^{-\beta}$, for a fixed real constant $\beta > 1$. Here, β is called the *power law exponent.* Since the function $f(k) = k^{-\beta}$ decays to 0 as k tends to ∞ polynomially (not exponentially), the power law degree distribution is also known as a *heavy-tailed distributions.* See Figure 1.5 for a visualization of the general shape of a power law distribution. A graph for which the power law holds is known as a *power law graph.*

By taking the logarithms of the degree distribution, we will obtain an approximate straight line with slope $-\beta$. The power law usually exists for a

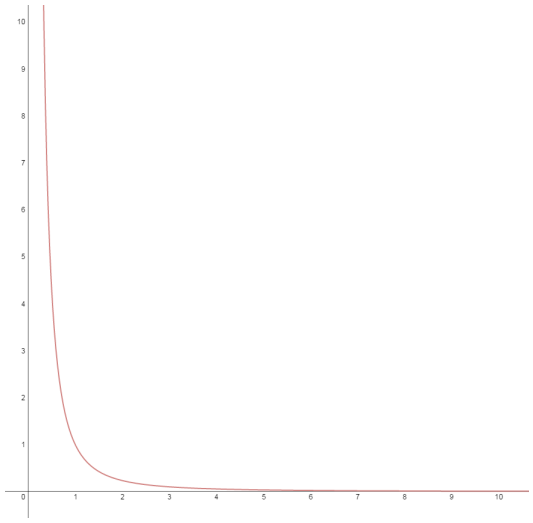


FIGURE 1.5. General shape of a power-law distribution.

specific interval of degrees with some deviation for nodes with small or large degrees. See Figure 1.6 and 1.7 for real-world examples of log-log plots.

$$\log(N_{k,G}) \sim \log(t) - \beta \log(k).$$

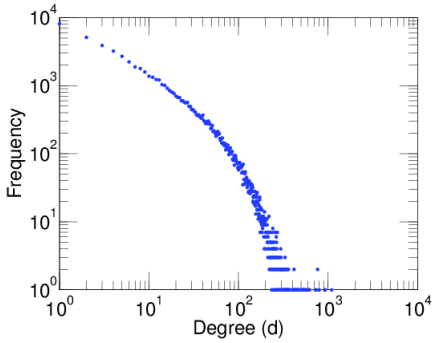


FIGURE 1.6. A log-log plot for Facebook friendships [**43**].

1.3.2. Small World Property. *Small world graphs* were first introduced in [**44**] where Watts and Strogatz defined the average distance (characteristic path length) that measured global distances in a graph and the measure for "cliquishness" of neighbourhoods. Small world graphs with order t satisfy the following bound: $\text{diam}(G) = \Theta(\log(t))$. However, when there

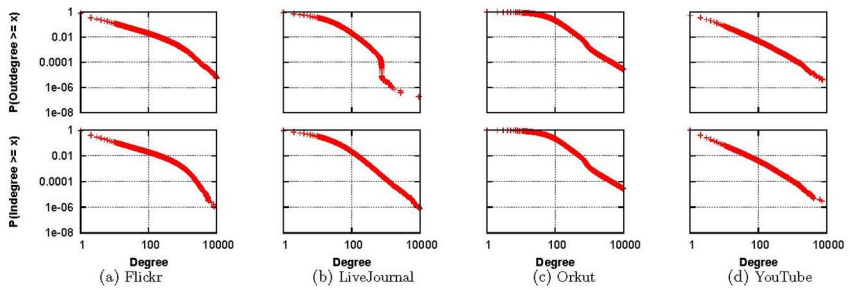


FIGURE 1.7. Log-log plots for Flickr, LiveJournal, Orkut and Youtube demonstrating the existence of power law degree distribution [**35**].

is an isolated vertex in a graph, this causes its diameter to be infinite. Thus, another measurement for global distance is necessary.

The *average distance* for a connected graph with order n, G, is defined as:

$$L(G) = \frac{\sum_{u,v \in V(G)} d(u,v)}{\binom{n}{2}} = \frac{2}{n^2 - n} \sum_{u,v \in V(G)} d(u,v).$$

Watts and Strogatz [**44**] reported the small world property requires $L(G)$ to be much smaller than the graph's order. It is demanded that

$$L(G) = \Theta(\log \log n).$$

A measurement that measures local density is the *clustering coefficient*. This coefficient should be larger relative to a random graph with the same order and size. Given a graph G with order n and $x \in V(G)$, define:

$$C(x) = \frac{\left|E\big(N(x)\big)\right|}{\binom{\deg(x)}{2}} = \frac{2\left|E\big(N(x)\big)\right|}{\deg(x)(\deg(x)-1)}.$$

The *clustering coefficient* of a graph G, $C(G)$, is the average of all the clustering coefficients over all the vertices, which equates to:

$$C(G) = \frac{1}{n}\sum_{x\in V(G)} C(x) = \frac{2}{n}\sum_{x\in V(G)} \frac{\left|E\big(N(x)\big)\right|}{\deg(x)(\deg(x)-1)}.$$

The value of the clustering coefficient is a rational number within $[0, 1]$. Complete graphs (that is, ones that contain all possible edges) will have a clustering coefficient of 1, while graphs without triangles have a clustering coefficient of 0.

1.3.3. Modelling On-line Social Networks. There have been attempts to design a mathematically rigorous model that can simulate observed properties of OSNs. A model is said to *simulate* a property when a graph generated by the model satisfies the property, with high probability.

We provide here a brief list of some of these models and their properties. The preferential attachment (PA) model [**5, 7**] (satisfying properties 1 to 3), the geo-protean (GEO-P) model [**11**] (satisfying properties 1 to 4), the iterated local transitivity (ILT) model [**10**] (satisfying properties 1, 2, 4,

5 (but not 3 and 6)), a model based on affiliation graphs [**29**] (satisfying properties 1 to 4), component structure model [**25**] (satisfying only property 6), the Forest Fire model, and Kronecker multiplication models [**27, 28**] (satisfying properties 1 to 4).

1.4. Probability

A few concepts of probability are needed in the thesis such as expectation and concentration. A discrete *probability space* $\mathcal{S}$ contains the following triplet, $(S, \mathcal{F}, \mathbb{P})$. The *sample space*, denoted S, is a non-empty and finite countable set. Set $\mathcal{F}$ is a collection of all possible subsets of S, these are called *events*. Finally, the function $\mathbb{P} : \mathcal{F} \to \mathbb{R}$ is known as the *probability measure*. The probability measure has the following properties:

(1) For all events $A, \mathbb{P}(A) \in [0, 1]$, and $\mathbb{P}(S) = 1$.

(2) If $\mathcal{F}$ is countable $(A_i : i \in I)$ and the events are pairwise disjoint, then

$$P\left(\bigcup_{i \in I} A_i\right) = \sum_{i \in I} \mathbb{P}(A_i).$$

Since $\mathcal{F}$ is considered to be the power set of S, it does not need to be specified. An example of a probability space is the *uniform probability space* on S with $\mathbb{P}(A) = \frac{|A|}{n}$. The set S is defined as being a finite set with $n > 0$ elements. An element is chosen *uniformly at random* (or *u.a.r.*) when all elements have the same probability of being chosen. Like in this case, every element has probability of $\frac{1}{n}$ to be chosen from S.

A *discrete random variable*, denoted X, is a function $X : S \to \mathbb{R}$ whose inputs are a sample space S on the probability space $\mathcal{S}$. Some information about binomial distribution is required for this thesis. The *probability mass function* (*p.m.f.*), $f : \mathbb{R} \to [0, 1]$, of X is defined by $f(x) = \mathbb{P}(X = x)$. Given a $x \in \mathbb{R}$, we compute

$$\mathbb{P}(X = x) = \sum_{\substack{s \in S, \\ X(s)=x}} \mathbb{P}(\{s\}).$$

The *first moment* of a random variable or its *expectation*, written $\mathbb{E}(X)$, is defined as

$$\mathbb{E}(X) = \sum_{s \in S} X(s)\mathbb{P}(\{s\}).$$

If S is finite, then $\mathbb{E}(X)$ will also be finite.

When taking the expectation of multiple random variables, one property that will provide significant help is stated in Theorem 1.2.

THEOREM 1.2. [**21**] *If the random variables* $X, Y, and X_i$ *where* $1 \leq i \leq n$, *are defined on a probability space, then the following holds.*
Linearity of Expectation. *If* c_i *is any real number constant where* $1 \leq i \leq n$, *then*

$$\mathbb{E}\left(\sum_{i=1}^{n} c_i X_i\right) = \sum_{i=1}^{n} c_i \mathbb{E}(X_i).$$

A *binomial random variable* X has parameters n and p and has the *p.m.f.* $\mathbb{P}(X = i) = \binom{n}{i}p^i(1-p)^{n-i}$ while its expectation is np and its variance $np(1-p)$. This simply counts the number of successes in n independent trails with a probability of p success. We state the following concentration inequality.

THEOREM 1.3. [**21, 22**] *(Chernoff bounds) If X is a binomial random variable, the following holds.*

$$\mathbb{P}(X \geq \mathbb{E}(X) + t) \leq \exp(-\mathbb{E}(X))\phi(t/\mathbb{E}(X)).$$

$$\mathbb{P}(X \leq \mathbb{E}(X) - t) \leq \exp(-\mathbb{E}(X))\phi(-t/\mathbb{E}(X)).$$

where $\phi(x) = (1+x)\log(1+x) - x$ for $x \geq -1$, and $\phi(x) = \infty$ for $x < -1$. If $\varepsilon \leq 3/2, then

$$\mathbb{P}(|X - \mathbb{E}(X)| \geq \varepsilon\mathbb{E}(X)) \leq 2\exp\Big(-\frac{1}{3}\varepsilon^2\mathbb{E}(X)\Big).$$

We say an event A_n holds *asymptotically almost surely (a.a.s.)* if it holds with probability tending to 1 as n tends to infinity.

1.4.1. Random Graphs. Let n be a positive integer, and let p be a real number in $[0, 1]$. The space of random graphs of order n with edge probability p is denoted as $G(n, p)$. Random graphs were first studied by Paul Erdős and Alfred Rényi in 1959 [**17**]. They are graphs that have a binomial degree distribution which follows $\mathbb{P}(G) = p^{|E(G)|}(1-p)^{\binom{n}{2}-|E(G)|}$.

These are graphs with a fixed vertex set V and varying number of edges (two vertices are adjacent independently with probability p). The number of edges follows a binomial distribution and has expectation $\binom{n}{2}p$, as there are $\binom{n}{2}$ many possible edges each occurring with probability p. Although $G(n,p)$ represents a probability space of graphs, we abuse language and refer to it as the *random graph with order n and edge probability p*. See Figure 1.8 for an example of a randomly drawn graph.

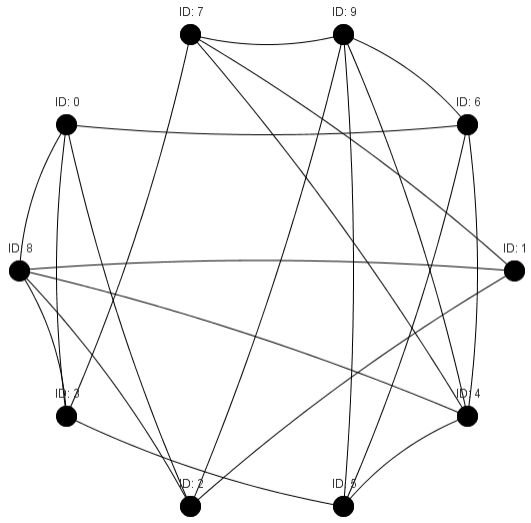


FIGURE 1.8. Sample graph from $G(n,p)$ where $n = 10$ and $p = 0.5$.

If we let the random variable X to be the degree of any particular vertex, then it follows a binomial distribution. A binomial distribution histogram approximates a bell curve shape with a concentration around its expected value when p is fixed and $n \to \infty$. The random variable X is defined as the sum of random variables that follow a Bernoulli distribution (that is, it counts the number of vertices adjacent to the particular node): $X = \sum_{i=1}^{n} Y_i$. Note that a *Bernoulli* distribution is a simple distribution that allows a random

variable to take the value of 1 with probability q (in other terms, 0 with probability $q = 1 - p$). Thus, the expected value of a Bernoulli distribution is simply p:

$$\begin{aligned} E(X) &= E(\sum_{i=1}^{n} Y_i) \\ &= \sum_{i=1}^{n} E(Y_i) \\ &= \sum_{i=1}^{n} p \\ &= np, \end{aligned}$$

where the seconds equality follows by the Linearity of Expectation (Theorem 1.2). Concentration on the expected degree follows by use of the Chernoff bounds; see Theorem 1.3.

Figure 1.9 is the expected shape of a binomial distribution.

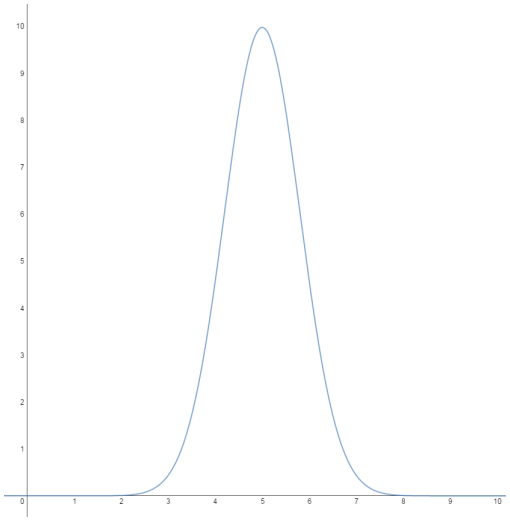


FIGURE 1.9. General shape of a binomial distribution.

1.5. Dominating On-line Social Networks

One key property that is noticed in social networks is that people tend to have characteristics or attributes similar to their friends. The reason for this is because of *social influence* (adopting behaviours through word-of-mouth or imitation) and *selection* (forming relationships with people similar to you) [**16**]. The structure of these networks will vary due to these two forces. However, in the end, this causes people to have relationships with similar people. Social influence produces a network-wide uniformity while selection drives the network to smaller clusters [**16**]. An example of social influence was presented by Kramer et al. in 2014 [**24**] where they provided evidence of emotional contagion by manipulating the news feed for nearly 700,000 Facebook users.

With over a billion users and 70 billion friendships, Facebook is the world's largest on-line social network. In 2012, it was computed that Facebook had an average of 4 degrees of separation amongst their users [**4**]. This brings up an important popular subject of on-line social networks: information diffusion or the spread of (social) influence. Researchers worldwide have been drawn to try and model these processes along with their dynamic relationships. The spread of influence can take in many forms: a computer virus, a marketing ad, a disease, or even social influences (such as emotional states [**24**]). A good indicator of how susceptible a social network is to the spread of influence is the number of users required such that for every user not in this set is adjacent to at least one user in the set (that is, a dominating set).

In Section 1.2.1, we defined the minimum cardinality of the domination set as the domination number of the network. These are the agents that potentially have the most influence in the network. By sending a message, a virus, an emotion, through these users, that information gets rapidly spread out to the entire network. Other various of applications that uses this domination concept are: efficient data routing [**40**], detecting highly significant optimized subsets of proteins in protein-protein interaction networks (PPIs) [**33**], and developing a dynamic backbone for Mobile Ad-Hoc Networks (MANETs) for routing and broadcasting [**45**]. Domination also plays a role in distributed computing and network controllability [**15**].

In addition, other concepts have been studied to determine how sensitive a graph is to influence like social elites [**14**]. Using the domination number to analyze a network only allows us to potentially capture such agents in the network. A more sophisticated measurement would be required to analyze graph sensitivity to influence. The paper published by Kramer et al. [**24**] in 2014 provided empirical evidence of emotional contagion by manipulating the Facebook news feed of over 600,000 Facebook users. The researchers did not change the content of what the users were posting but only restricted what they saw based on keyword analysis. They found that users who were subject to more positive posts tend to post more positive content. Similarly the users subject to negative content were more likely to post something negative. This brings up an important fact that we as humans do not required human-human interactions in order to be influenced. A good way to see the significance of

a dominating set in this example is to imagine such a set existed with a size of say, 1,000. That means if these 1,000 users were to post positive content, then the entire networks (potentially consisting of millions of users) would be influenced.

1.6. Outline of Thesis

In Chapter 2, the MGEO-P model will be introduced with a brief background of its development from the GEO-P model, and how it relates to the concept of Blau space and its properties. Some simulations are presented. Within Chapter 3, upper bound domination numbers are computed for the Facebook 100 (also known as FB100) graphs and corresponding MGEO-P samples. In Chapter 4, a new variation of the MGEO-P model is introduced and analyzed: the Distance MGEO-P model. And finally in Chapter 5, we summarize our findings and present open problems and directions for future work.

CHAPTER 2

Memoryless Geometric Protean Model

2.1. Background

The *memoryless geometric protean model* (MGEO-P) was first introduced by Bonato et al. in 2014 [**9**]. This is a variation on the geometric protean model (GEO-P) that allows us to approximate the GEO-P model without costly sampling. The MGEO-P model has its edges arrive based on a given distance metric in m-dimensional Euclidean space $\mathbb{R}^m$. This space closely mirrors *Blau space* [13], a construction in the social sciences. It is a multidimensional coordinate system where the dimensions are defined by social demographic variables. The relative position of the vertices within Blau space are guided by the principle of *homophily* [14]. That is, vertices that are closer to each other tend to have similar socio-demographic characteristics. Examples of Blau space dimension variables are age, sex, location, education and income. The sociologist Peter Blau introduced this notion which bears his name. He was the first to take individuals in a social network and abstract them to nodes in a multi-dimensional space. He had the idea that "social forces" are the overlaps of social demographic characteristics and that population structure can influence human behaviour [**6**]. It was later on more fully developed by Miller McPherson [**30, 31, 32**].

2.2. The Model

The MGEO-P model consists of five parameters MGEO-P(n, m, α, β, p): the number of nodes n, the dimension m, the attachment strength parameter $0 < \alpha < 1$, the density parameter $0 < \beta < 1 - \alpha$ and the probability of connection $0 < p \leq 1$. Here, the parameters n and m are positive integers that are greater than 0 and α, β, p are real numbers. The network starts out empty. During each iteration of the process, a new vertex is added to the network. There will be a total of n steps resulting in a graph of order n. For each of the n steps, a vertex v is created and given a random rank r_v and random position $q_v \in \mathbb{R}^m$ within the unit-hypercube $[0, 1]^m$. The random rank is chosen from the unused remaining ranks from 1 to n. A unit-hypercube side has length one unit with 2^m vertices in $\mathbb{R}^m$ with coordinates $\{0, 1\}^m$. The volume of space that a node influences is $r_v^{-\alpha} n^{-\beta}$. The radius of a node can be obtained by taking the mth-root of the volume of the node and dividing it by two. Thus, each vertex has a radius of influence given by:

$$I(r_v) = \frac{1}{2}(r_v^{-\alpha} n^{-\beta})^{1/m}.$$

An undirected edge is created between v and any preexisting vertex u with probability p, if $D(v, u) \leq I(r_u)$. The distance is computed using the infinity-norm. The infinity-norm is defined as $||x||_\infty = \max\{|x_1|, |x_2|, ..., |x_m|\}$. The distance used in the model is computed as follows:

$$D(u, v) = \min\{||q_v - q_u - z||_\infty : z \in \{-1, 0, 1\}^m\}.$$

The z term within the distance formula allows the geometric space to be symmetric and "wrap" around like a torus. This process is repeated until all the vertices have been placed within the unit-hypercube.

2.3. Simulations

Figure 2.1 demonstrates the process of simulating MGEO-P(3,2,0.17,0.27,1) which we explain here. For simplicity, we assume that the probability of connection p for the MGEO-P model is 1. In reality, we know this is not the case. A more realistic value would be less than 0.5. The network initially starts out empty. In the first step, a node is placed with the coordinates (0.1894, 0.1284) and a radius of 0.431081. There are no pre-existing nodes to check for any edges. During the second step, another node is generated at (0.7439, 0.7232) and a computed radius of 0.406416. The procedure continues with checking if the distance between the current generated node and any pre-existing node is less than the pre-existing node's radius. In this case, $D(u, v) = D(1, 2) = 0.445500 \nleq 0.431081$. Therefore, an edge is not placed between these two nodes. Finally, the third node is placed in the space at (0.4200, 0.9213) with radius 0.392648. Repeating the process, computing the distance of nodes one and three $D(1, 3) = 0.2306 \leq 0.431081$ tells us an edge is placed. Likewise, for nodes two and three $D(2, 3) = 0.3239 \leq 0.406416$, thus, an edge is placed.

Figure 2.2 shows visual representations of graphs generated by the MGEO-P model using JUNG 2.0.1 (Java Universal Network/Graph) framework which was released in early 2010. In the original GEO-P model, the process does

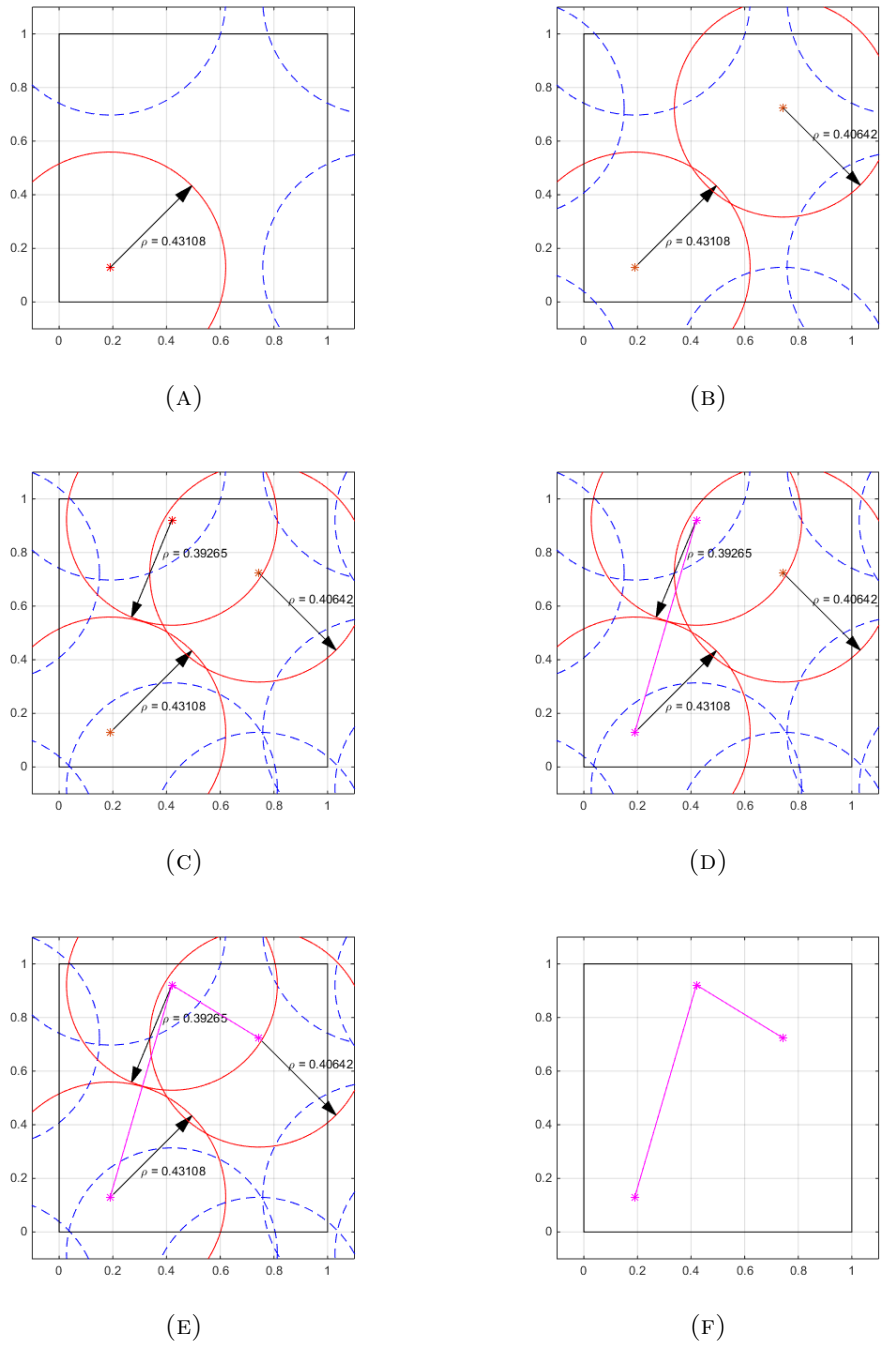
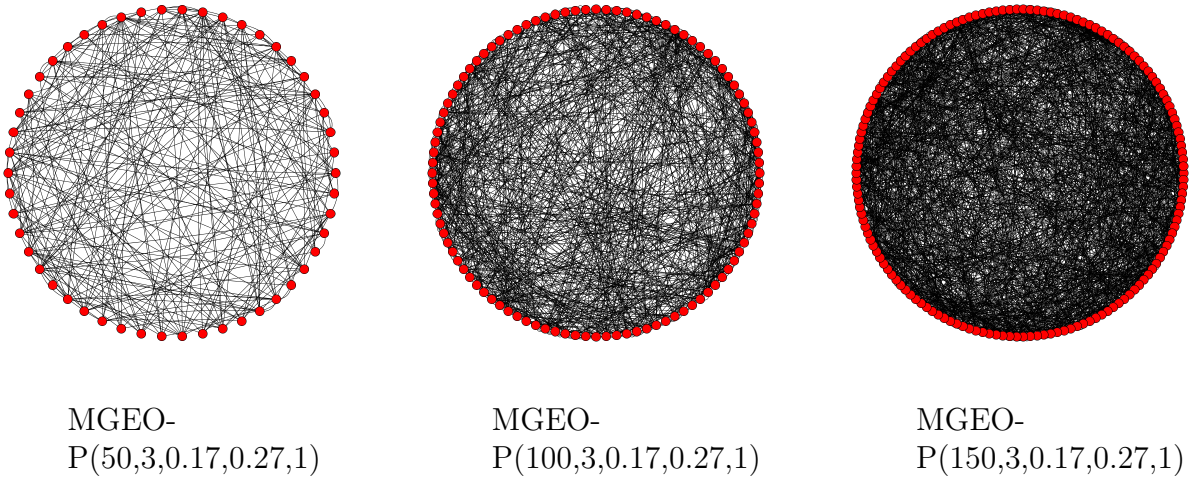
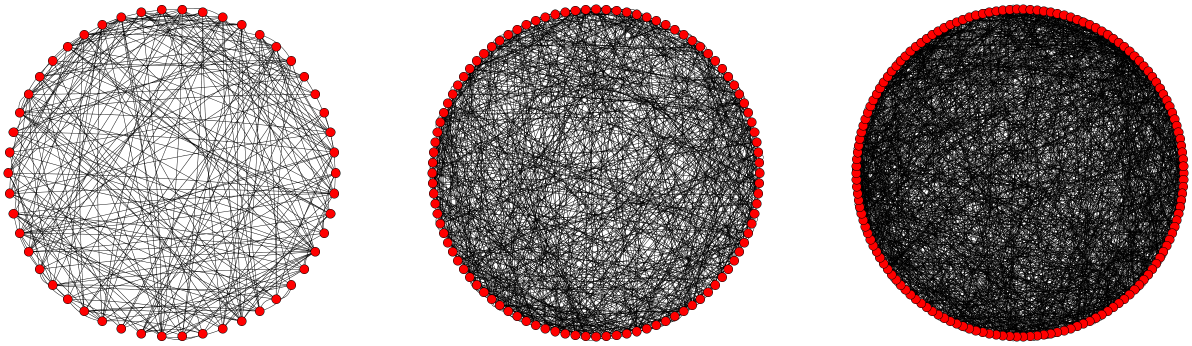


FIGURE 2.1. Time steps of a MGEO-P$(3, 2, 0.17, 0.27)$ sample. The unit square is given by the black interior rectangle from 0 to 1. The broken blue circles are copies of nodes illustrating the torus metric. *(A)* Time step 1: First node is added to the network. *(B)* Time step 2: Second node is added to the network. No nodes are overlapping, thus no edges. *(C)* Time step 3: Third node is added to the network. Overlaps with node one and two. *(D)* An edge is placed in between nodes one and three. *(E)* An edge is placed in between nodes two and three. *(F)* Final graph without node radii.

not end there. The procedure is repeated until a random stop time, where in each step the least-recently added vertex will be removed and a new one inserted.



MGEO-P(50,3,0.17,0.27,1) | MGEO-P(100,3,0.17,0.27,1) | MGEO-P(150,3,0.17,0.27,1)



MGEO-P(50,4,0.17,0.27,1) | MGEO-P(100,4,0.17,0.27,1) | MGEO-P(150,4,0.17,0.27,1)

FIGURE 2.2. Samples of the MGEO-P(n, m, α, β, p) model, with various choices of parameters.

As mentioned in Chapter 3 (Domination Number in Facebook 100 and the MGEO-P Model), the MGEO-P parameters were computed for Facebook 100 data [**12**]. Tables A.1, A.2, A.3 show a preliminary analysis of the MGEO-P samples using the Facebook 100 parameters. The tables provide graph properties such as number of nodes, number of edges, average degree, maximum

degree and density. Density is simply a measurement of how many edges there are relative to the maximum number of edges. It is defined as:

$$D = \frac{2E}{V(V-1)}.$$

Therefore, complete graphs have a density of 1 and graphs with no edges a density of 0. A complete graph for the MGEO-P model would have a attachment strength parameter and density parameter of 0 to keep each nodes radius of influence uniform. These results will be used to make comparisons between the MGEO-P model, the modified MGEO-P model, and the Facebook data to draw any conclusions.

The following theorems about the MGEO-P model appear in [**9**] along with their proofs. We omit the proofs here as they are beyond the scope of this thesis. A useful result that follows from Theorem 2.1(1) is that a.a.s. the minimum degree δ is at least $(1 + o(1))pn^{1-\alpha-\beta}$.

THEOREM 2.1. [**9**] *Asymptotically almost surely, the following properties hold for graphs generated by MGEO-P.*

(1) *If v is a node of a MGEO-P(n, m, α, β, p) sample with rank R and age i, then*

$$\deg(v) = \Big(\frac{i-1}{n-1}\frac{p}{1-\alpha}n^{1-\alpha-\beta} + (n-i)pR^{-\alpha}n^{-\beta}\Big)\left(1 + \mathcal{O}\left(\sqrt{\frac{\log^2(n)}{n^{1-\alpha-\beta}}}\right)\right).$$

(2) *The average degree of a vertex of MGEO-P(n, m, α, β, p) is*

$$d = \frac{p}{1-\alpha} n^{1-\alpha-\beta} \left(1 + \mathcal{O}\left(\sqrt{\frac{\log^2(n)}{n^{1-\alpha-\beta}}}\right)\right).$$

(3) *The diameter of MGEO-P(n, m, α, β, p) is* $n^{\Theta(\frac{1}{m})}$.

(4) *If k is chosen such that*

$$n^{1-\alpha-\beta} \log^{\frac{1}{2}}(n) \leq k \leq n^{1-\frac{\alpha}{2}-\beta} \log^{-2\alpha-1}(n),$$

then MGEO-P(n, m, α, β, p) satisfies

$$N_{\geq k} = \left(1 + \mathcal{O}\left(\log^{-\frac{1}{3}}(n)\right)\right) \frac{\alpha}{1+\alpha} p^{\frac{1}{\alpha}} n^{\frac{(1-\beta)}{\alpha}} k^{-\frac{1}{\alpha}},$$

where the number of vertices of degree at least k is denoted by $N_{\geq k}$.

These theorems demonstrate that the model a.a.s. exhibits a power-law degree distribution. A few samples of the model were simulated with their histogram and log-log plot computed to provide evidence that a power-law distribution appears in graphs generated by the model. See Figure 2.3 and Figure 2.4. As expected, the MGEO-P model follows a power-law degree distribution and has a linear slope in its log-log plot, while G(n, p) follows more of a binomial degree distribution with a noisy log-log plot.

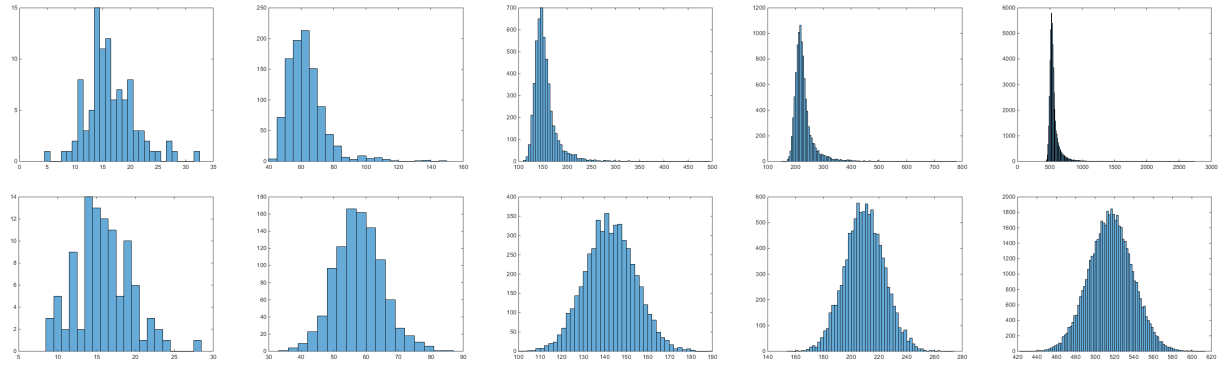


FIGURE 2.3. Degree distribution of MGEO-P$(n, 4, 0.17, 0.27, 1)$ samples and G(n, p) model for $n = 100, 1000, 5000, 10,000$ and $50,000$. Note that p was chosen in G(n, p) so that the graphs sampled have the approximate same average degree as the corresponding MGEO-P samples. The size of each bin were automatically computed using MATLAB's automatic binning algorithm.

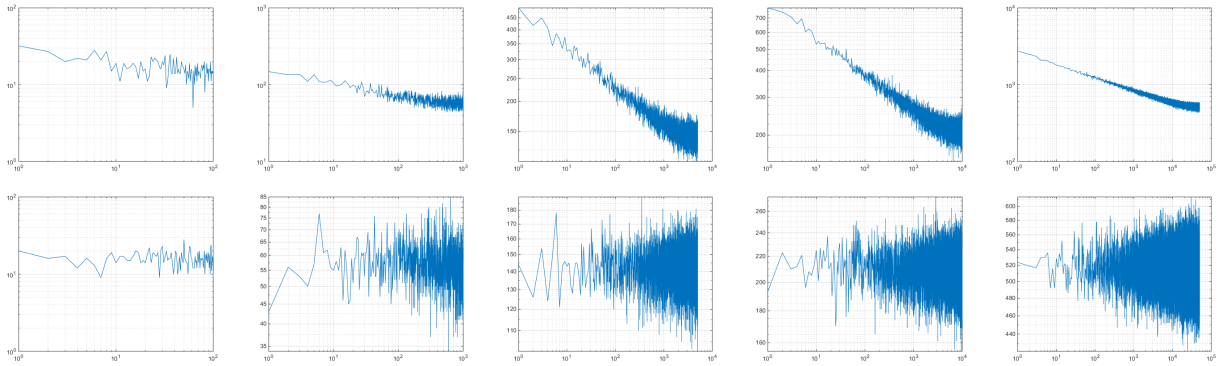


FIGURE 2.4. Log-log plots of MGEO-P$(n,\ 4,\ 0.17,\ 0.27,\ 1)$ samples and G(n, p) model for $n = 100, 1000, 5000, 10,000$ and $50,000$. Note that p was chosen in G(n, p) so that the graphs sampled have the approximate same average degree as the corresponding MGEO-P samples.

CHAPTER 3

Domination Number in Facebook 100 and the MGEO-P Model

3.1. Facebook 100

To investigate the domination number and its values on social networks, we need a data set that resembles a natural on-line social network. We randomly generate MGEO-P samples since they follow most OSN properties. However in [**9**], the parameters of a Facebook dataset were computed for the MGEO-P model. Using these parameters, model samples can be simulated and tested. The social network data used are known as the *Facebook 100* (or FB100) graphs. They were distributed and anonymized by Mason Porter who worked at Facebook. It contains 100 Facebook network samples from 100 universities from the United States in September 2005 [**41**] as separate networks. Their order and size vary from 700 users to 42,000 users and 16,000 friendships to 1,590,655 friendships. For this thesis, the data that was used were the friendship networks of each institution as separate graphs. These graphs provide a single snapshot of subgraphs of the Facebook network.

The Facebook 100 data represents an historical snapshot of agents joining the network over time that can describe different stages of institutions within the network. For example, some users might have joined the network earlier or later. The original released data that has been analyzed in [**41**] contains

additional information such as gender, class year, major, high-school and residence.

Before investigating the domination sets within the FB100, some preliminary analysis should be done to indicate any significant properties about these social networks. It should be noted that the minimum degree for each dataset in the Facebook 100 network is 1 and is excluded from the tables. Tables A.16, A.17, A.18 provides the preliminary analysis of FB100.

3.2. Domination Number within Facebook 100

Given the FB100 graphs, the dominating set algorithms DS-RAI and DS-DC (as described in Chapter 1) were executed multiple times to provide an idea of what the upper bounds of the domination number of on-line social networks could be. Tables A.19, A.20, A.21, A.22 and A.23 in the Appendix, show the results of the DS-RAI algorithm for the k-cores, where $1 \leq k \leq 5$. The 0-core (that is, the original graph) was not accounted for because the Facebook 100 graphs do not have any isolated vertices thus, making the 0-core graphs equivalent to the 1-core graphs.

The DS-RAI algorithm performs the fastest and provides a good indication of what the upper bounds are for the Facebook 100 graphs. The DS-DC algorithm improves on these upper bounds at the cost of a longer execution time. The following Tables A.24, A.25, A.26, A.27 and A.28 in the Appendix, provides the dominating set size achieved by the DS-DC algorithm for the k-cores, where $1 \leq k \leq 5$. The DS-DC algorithm provides upper bounds

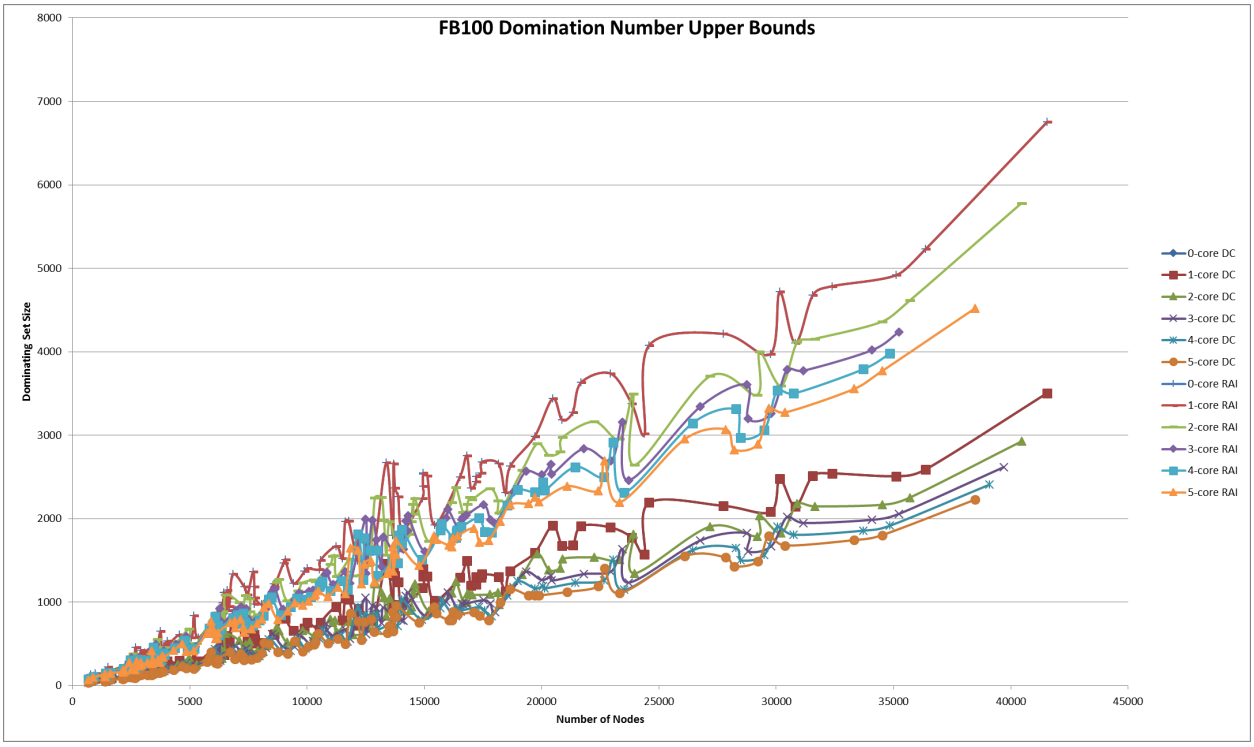


FIGURE 3.1. Domination number upper bounds of the Facebook 100 graphs.

that are approximately 75% of the order of the dominating sets obtained via DS-RAI.

The data represented in the plot Figure 3.1 are the dominating set sizes for the Facebook 100 graphs with DS-DC and DS-RAI relative to the graph order. Since our graphs are quite small when considering asymptotic bounds, the data progression can be consider either linear or sub-linear. The general upper bound mentioned in Theorem 1.1 suggests that the upper bound is potentially linear. Comparing the theoretical and computed bounds shows that the upper bound mentioned by Alon and Spencer in [**3**] is quite significantly higher than the computed bounds for the FB100 graphs. The comparison plot is shown in Figure 3.2, where the minimum degree δ used was considered to be 5 since we took up to the 5-core.

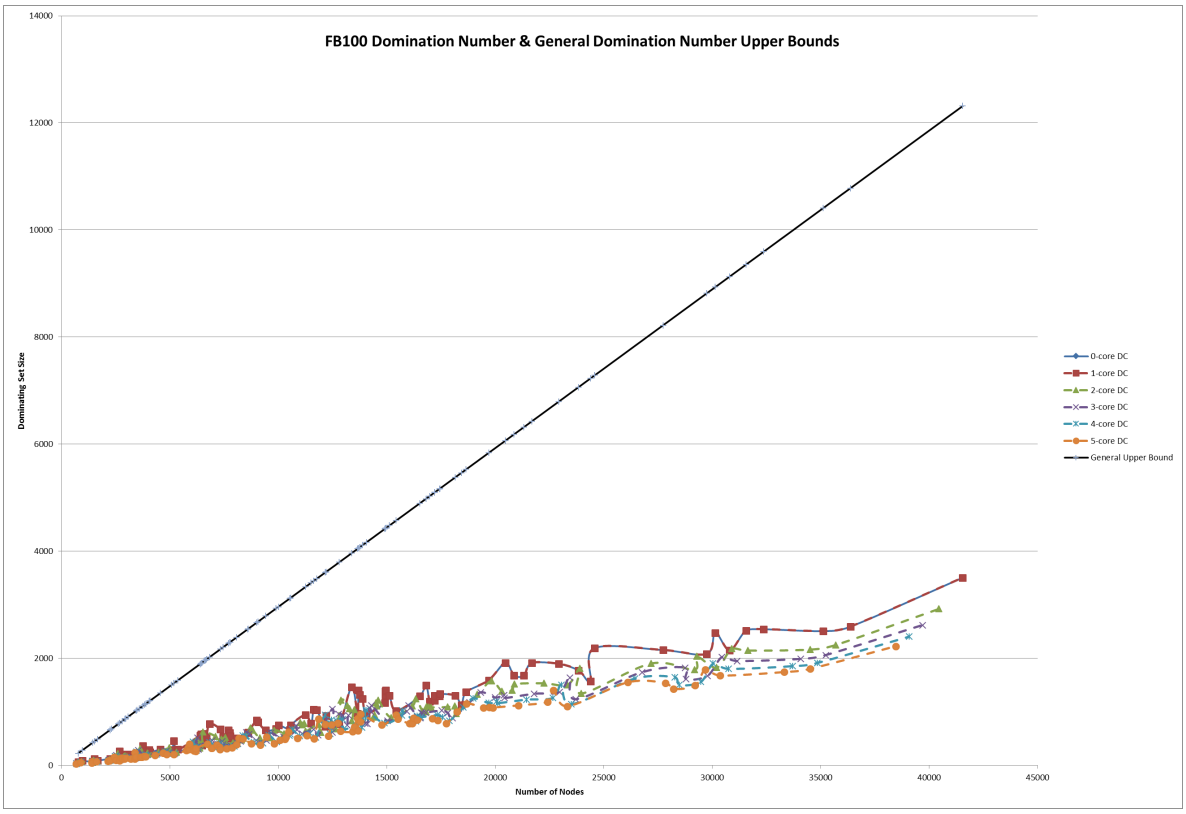


FIGURE 3.2. Domination number upper bounds of the Facebook 100 graphs with Theorem 1.1 upper bound ($\delta = 5$), presenting a significant overestimation.

3.3. Domination Number within MGEO-P

MGEO-P samples generated from the Facebook 100 parameters in [**9**] were used to provide an upper bound of the domination number for the model. For simplicity, when the samples were generated the probability parameter p was considered to be 1. To obtain an appropriate MGEO-P sample for the FB100, *percolation* must be done. This is the process of randomly removing edges until the graph has a set number of edges. For graphs with less edges than those in the data set, they are left intact. Just like in [**9**], our samples were percolated to match the number of edges of the Facebook data. Intuitively, one would expected that the graphs generated using the process of

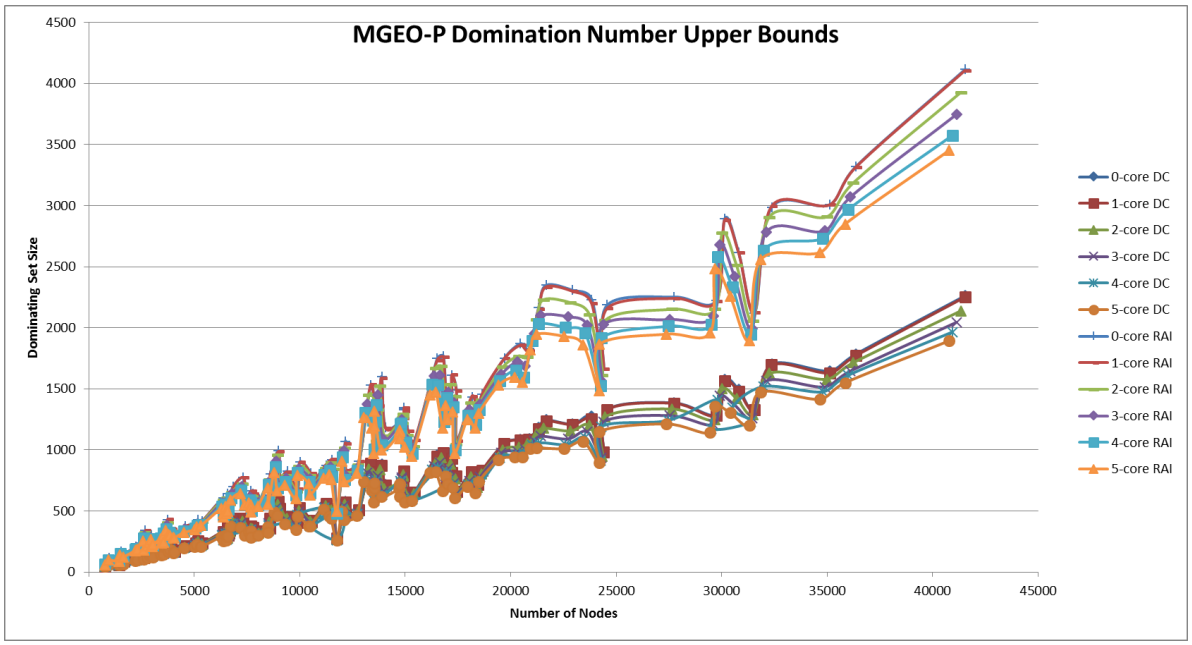


FIGURE 3.3. Domination number upper bounds of MGEO-P samples corresponding to the Facebook 100 graphs.

percolating to be quite similar than the original data. This is indeed true and is seen when comparing the preliminary analysis of Table A.1, A.2, A.3 and Table A.16, A.17 and A.18. A noticeable difference is the maximum degree of the samples. One should expect a larger dominating set for the MGEO-P samples because their maximum degrees are lower than the original data. The upper bound of the domination number is shown in Figure 3.3.

A recent finding in [**12**], provides a theorem regarding the domination number of the MGEO-P(n, m, α, β, p) model. The theorem poses that a sub-linear bound exist on the domination number for OSNs.

THEOREM 3.1. [**12**] *A.a.s the MGEO-P(n, m, α, β, p) model domination number satisfies*

$$\gamma(G) = \Omega(C^{-m/(1-\alpha)} n^{\alpha+\beta}) \text{ and } \gamma(G) = O(n^{\alpha+\beta} \log n),$$

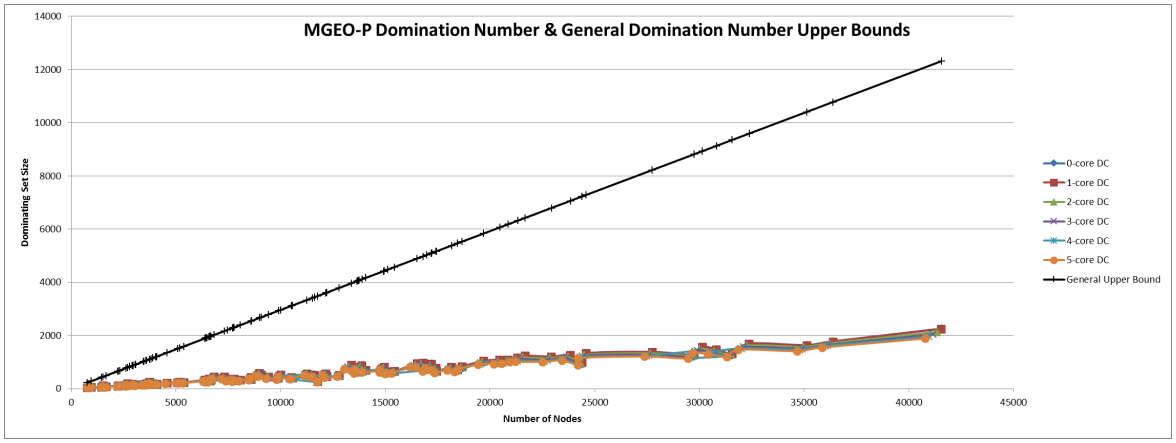


FIGURE 3.4. Domination number upper bounds of MGEO-P samples corresponding to the Facebook 100 graphs with Theorem 1.1 upper bound ($\delta = 5$).

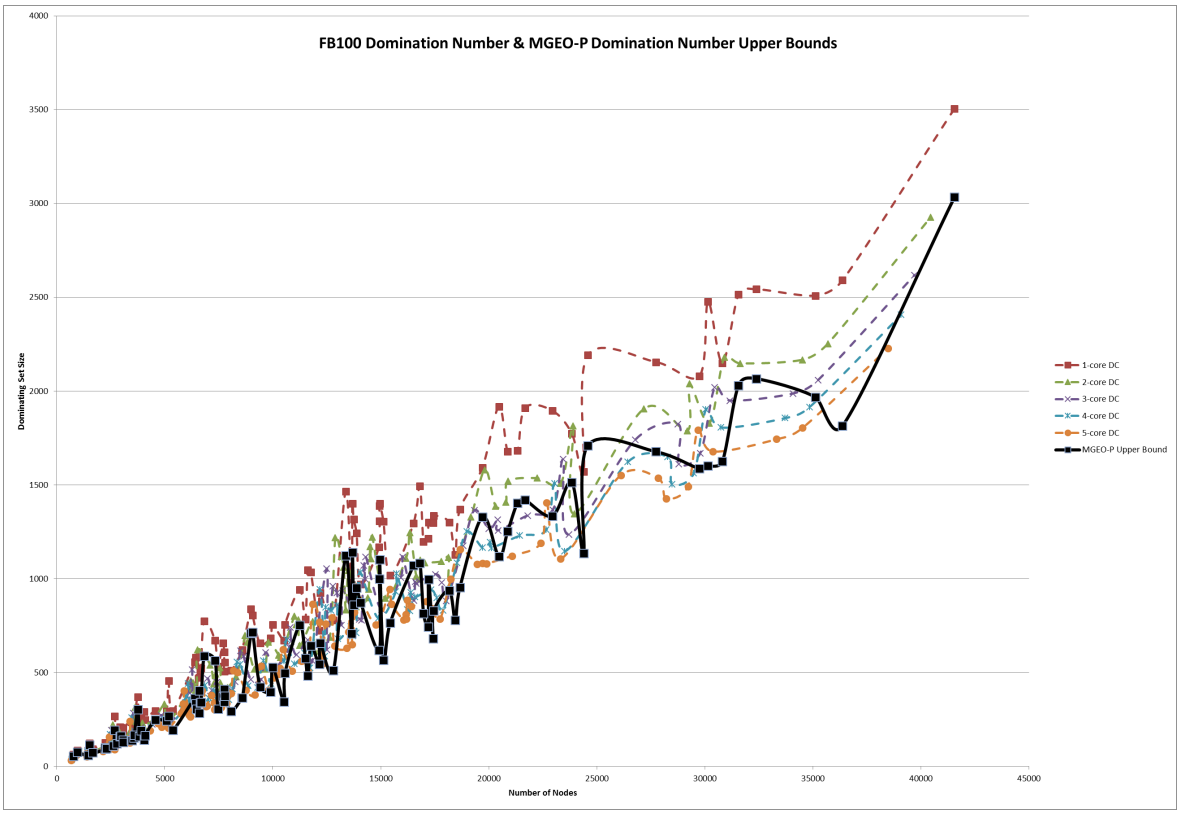


FIGURE 3.5. Domination number upper bounds of the Facebook 100 graphs with Theorem 3.1 upper bound, showing a very close correlation.

where C is any constant greater than 6 and $m = o(\log n)$. Particularly, a.a.s. $\gamma(G) = n^{\alpha+\beta+o(1)}$.

Based on a comparison of the plots, there is a high correlation between the theoretical MGEO-P upper bound and the domination number of the FB100 graphs. For the 100 samples, there is a noticeable pattern emerging. This provides empirical evidence that on-line social networks domination numbers follow sub-linear growth; more precisely:

$$\gamma(G) = O(n^{\alpha+\beta} \log n).$$

By incorporating a sub-linear line of best fits using MATLAB, we will be able to see how well the data fits this type of growth. For simplicity, we ignore the constant in front of the big Oh term. From these trend lines, we can determine how correlated the sub-linear fit is with the data. Table 3.1 provides the coefficient of determination; that is, how close the data fits with the regression line.

There are high correlations even when taken the 5-core. The theoretical upper bound occurs asymptotically almost surely. As our data sets are relatively small order (that is, orders in the thousands rather than millions or billions), the asymptotic results predicted in Theorem 3.1 may not be fully evident in the MGEO-P samples. As more experiments are done with larger orders, we hope that their domination number demonstrates a more clear sub-linear growth and that the data fits more tightly.

k	x	R^2
1	0.509	0.8472
2	0.492	0.8292
3	0.4818	0.8179
4	0.4741	0.8093
5	0.4677	0.803

TABLE 3.1. Fitting $y = n^x \log n$ to the domination number of the k-cores of FB100, where $1 \leq k \leq 5$.

CHAPTER 4

Distance Memoryless Geometric Protean Model

4.1. Motivation

The MGEO-P model has a probability parameter p which was considered 1 during our simulations of the Facebook 100 graphs. Recall that Blau space predicts that agents with similar attributes are closer in the space. We abstract this to a new paradigm in the setting of MGEO-P, where vertices are adjacent in a region of influence with probability given by the function $f(D(u, v))$, with u and v distinct vertices. That is, f is a function which depends on the metric distance between u and v. The distance $D(u, v)$ is defined below. In particular, vertices have a higher probability of being adjacent to other vertices in their radius of influence which are closer. An example of this is friendships within a community. A person will more likely be friends with someone who lives geographically closer to them. A larger scale example would be Facebook friendships between two people from different countries, where one may suspect that such people have a lower probability of being friends. This allows distance to play the role of the probability parameter. The model that uses this as the function of p is known as the *distance memoryless geometric protean model*, written DMGEO-P. This is new model will be the focus of this chapter. We define the model below in Section 4.2, and then give the results of simulations of the model in Section 4.3.

4.2. The Model

The DMGEO-P model is defined in a similar way to the MGEO-P model, except instead of five parameters there are four parameters DMGEO-P(n, m, α, β). The parameters are exactly like the MGEO-P model: the number of nodes n, the dimension m, the attachment strength parameter $0 < \alpha < 1$ and the density parameter $0 < \beta < 1 - \alpha$. The process is the exact same as the original MGEO-P model, however edges are added based on the following rule.

An undirected edge is created between v and any pre-existing vertex u with probability $f(D(u, v))$, if $D(v, u) \leq I(r_u)$. The distance and probability of adjacency, respectively are computed as follows:

$$D(u, v) = \min \{||q_v - q_u - z||_\infty : z \in \{-1, 0, 1\}^m\},$$

$$f(D(u, v)) = 1 - \frac{D(u, v)}{I(r_u)},$$

where $|| \cdot ||_\infty$ is the infinity-norm.

Figure 4.1 demonstrates the process of simulating DMGEO-P(3,2,0.17,0.27) which we explain here. The network initially starts out empty. In the first step, a node is placed with the coordinates $(0.1894, 0.1284)$ and a radius of 0.431081. There are no pre-existing nodes to check for any edges. During the second step, another node is generated at $(0.7439, 0.7232)$ and a computed radius of 0.406416. The procedure continues with checking if the

distance between the current generated node and any pre-existing node is less than the pre-existing node's radius. In this case, $D(u, v) = D(1, 2) = 0.445500 \nleq 0.431081$. Therefore, an edge is not placed between these two nodes. Finally, the third node is placed in the space at $(0.4200, 0.9213)$ with radius 0.392648. Repeating the process, computing the distance of nodes one and three $D(1, 3) = 0.2306 \leq 0.431081$ tells us an edge is placed with a probability of $f(D(1, 3)) = 0.203034$. Likewise, for nodes two and three $D(2, 3) = 0.3239 \leq 0.406416$, thus, an edge is placed with a probability of $f(D(2, 3)) = 0.465065$. Figure 4.2 shows visual representations of graph samples generated by the DMGEO-P model.

Like the MGEO-P samples, simulations were generated and a preliminary analysis was done. Tables A.29 A.30, A.31 provides us with the minimum degree, average degree, maximum degree and density of DMGEO-P samples analogous to the Facebook 100 graphs. When comparing these samples with the real data, the model tends to generate more sparse graphs with much less density and maximum degree. One might expect a domination number that is higher than the Facebook data. More nodes would be required to completely cover the network because the maximum degree is low. However, there are less graph samples with a minimum degree of one as in the Facebook data. And as in mentioned in Chapter 1, the domination set is sensitive to low degrees. We hope to see less "shifts" (that is, changes) in the domination set upper bound when taking the k-core of the DMGEO-P model. There is a higher probability of the low degree nodes being adjacent to at least one

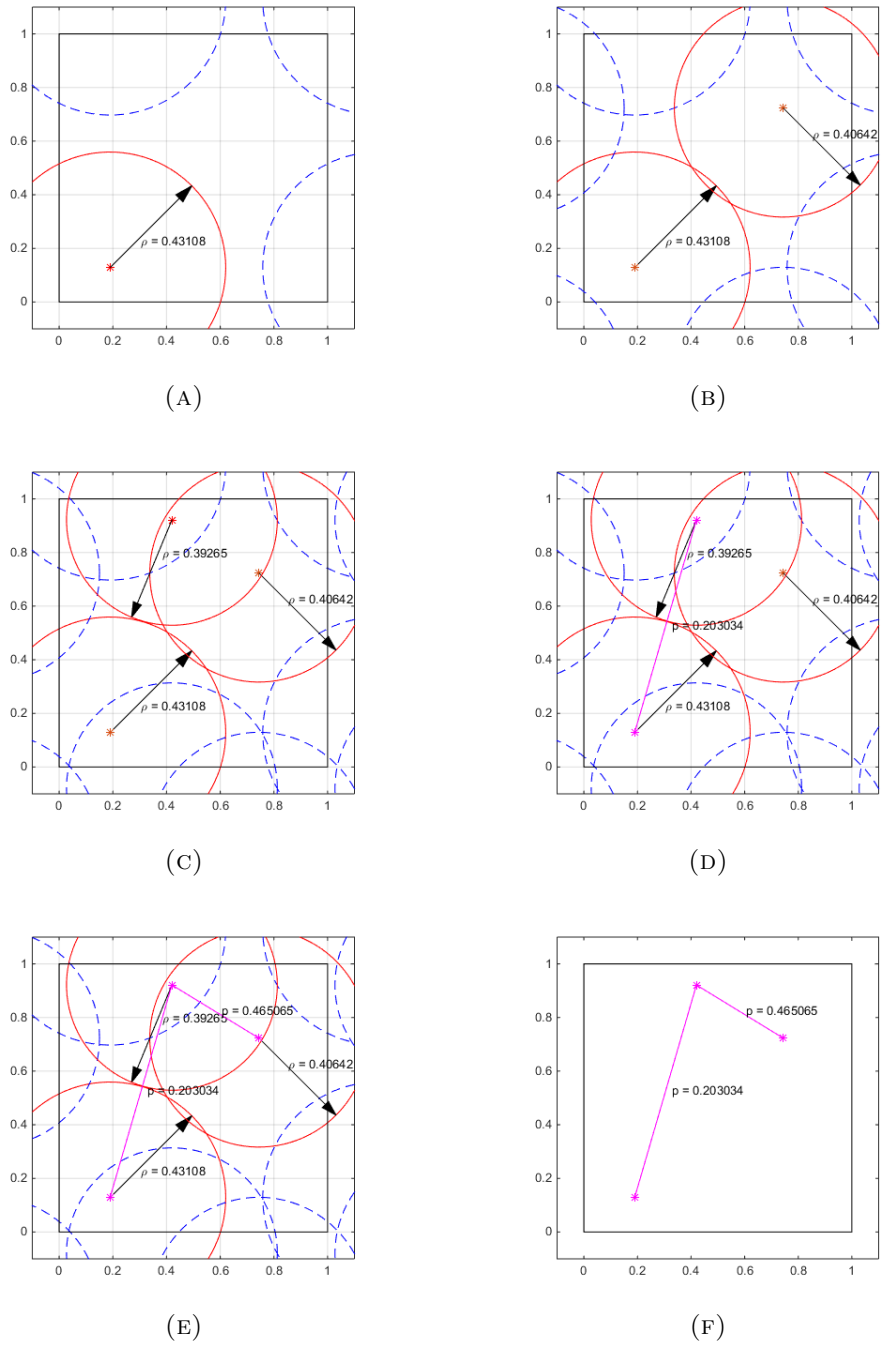


FIGURE 4.1. Time steps of a DMGEO-P$(3, 2, 0.17, 0.27)$ sample. The unit square is given by the black interior rectangle from 0 to 1. The broken blue circles are copies of nodes illustrating the torus metric. The images demonstrate the concept that the closer a node is to another in geometric space, there is a higher probability of adjacency. This is seen in the difference of probability between the two edges shown. *(A)* Time step 1: First node is added to the network. *(B)* Time step 2: Second node is added to the network. No nodes are overlapping, thus no edges. *(C)* Time step 3: Third node is added to the network. Overlaps with node one and two. *(D)* An edge is placed in between nodes one and three with probability of 0.203034. *(E)* An edge is placed in between nodes two and three with probability of 0.465065. *(F)* Final graph without node radii.

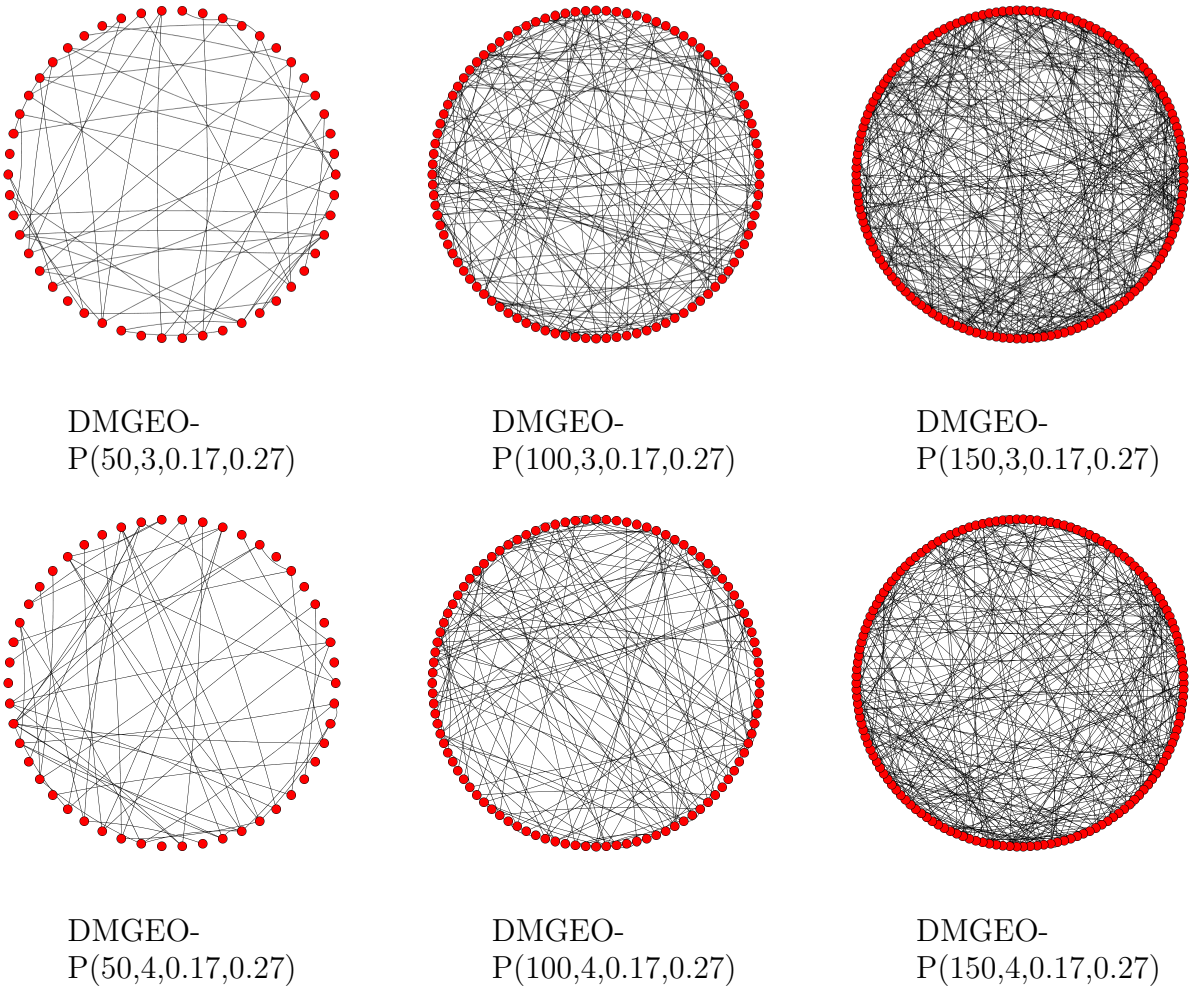


FIGURE 4.2. Samples of the DMGEO-P(n, m, α, β) model, with various choices of parameters.

agent in the dominating set since the maximum degree of the samples are also quite low relative to the order of the graphs.

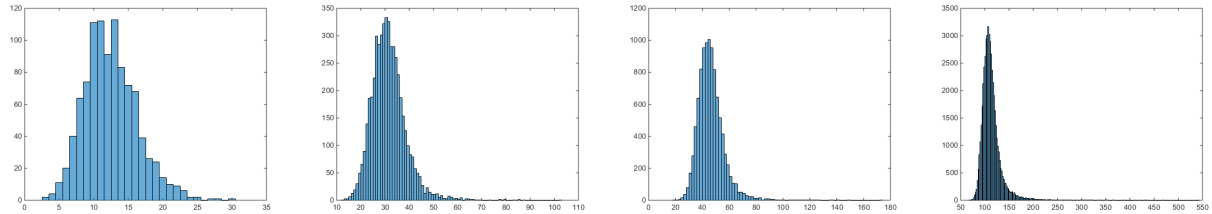


FIGURE 4.3. Degree distribution of DMGEO-P$(n, 4, 0.17, 0.27)$ samples for $n = 1,000, 5,000, 10,000$ and $50,000$.

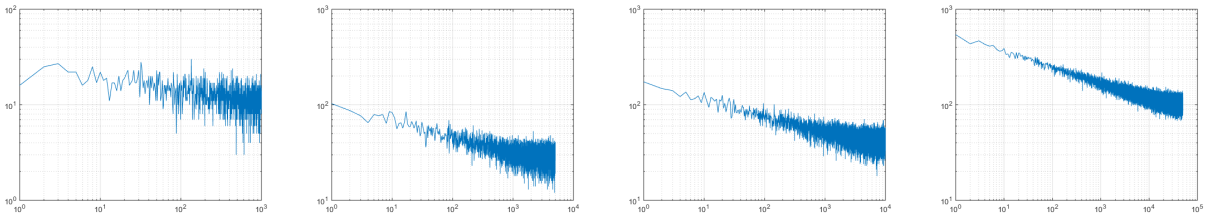


FIGURE 4.4. Log-log plots of DMGEO-P(n, 4, 0.17, 0.27) samples for $n = 1,000, 5,000, 10,000$ and $50,000$.

4.3. Domination Number of the DMGEO-P Model

Tables A.32, A.33, A.34, A.35, A.36, A.37 present the results of the DS-RAI algorithm for DMGEO-P samples that correspond to the FB100 network for their k-cores, where $1 \leq k \leq 5$. See Figure 4.5 for the plot of the dominating set size relative to the graph's order. Notice, as expected, the k-cores are all very similar; there is hardly any noticeable changes in domination set size. The DMGEO-P model appears to generate samples that are not as sensitive to *domination noise* (that is, the effect of having a higher domination number due to nodes with low degrees) than the MGEO-P model.

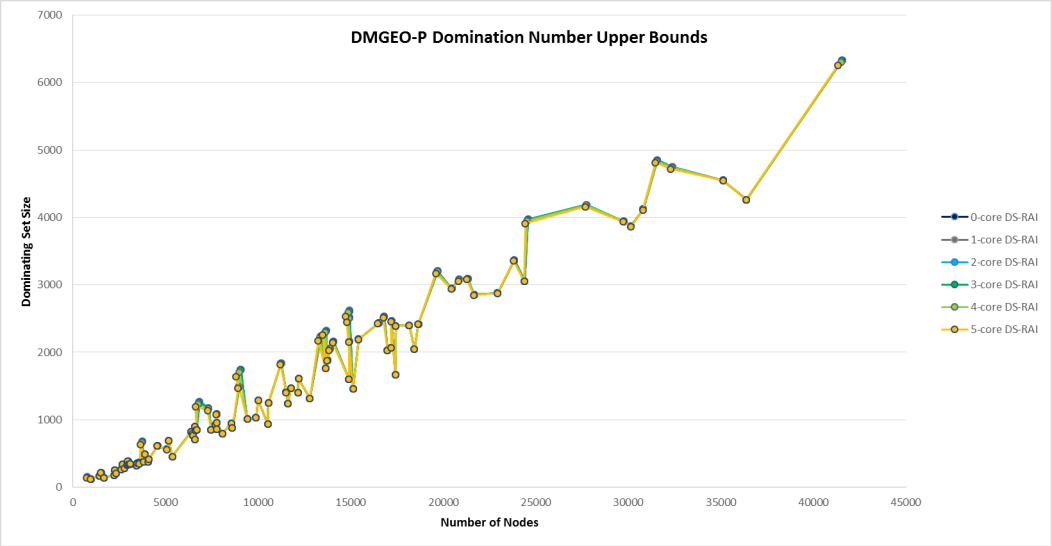


FIGURE 4.5. Domination number upper bounds of DMGEO-P samples corresponding to the Facebook 100 graphs using the DS-RAI algorithm.

Figure 4.6 compares the domination size of the DS-RAI algorithm for both DMGEO-P samples and the FB100 data. They appear to be relative close in size. It is interesting to observe the Facebook 100 graphs are more dense and should require less nodes in its domination set. As opposed to the DMGEO-P

model where graphs are less dense and should require more nodes to cover the graph completely. However, both contains domination sets around the same size. Due to the higher minimum degree in the DMGEO-P model, there are a lower number of nodes with low degrees (when comparing the FB100 data). It would be advantageous to calculate the clustering coefficient of the Facebook 100, MGEO-P and DMGEO-P domination set nodes to see if any pattern or observations can be made about such nodes. We leave this for future work but hope to expect that these agents have a higher than average clustering coefficient.

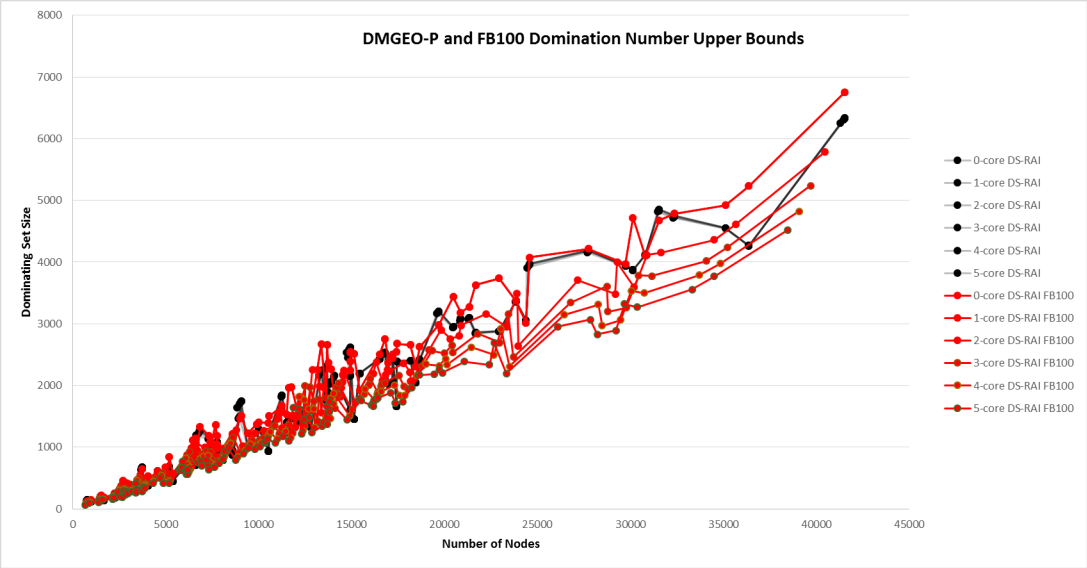


FIGURE 4.6. Domination number upper bounds of DMGEO-P samples corresponding to the Facebook 100 graphs and FB100.

Using the same Theorem 1.1 with the minimum degree of 5, Figure 4.7 shows its large overestimation on the domination size. All of our results are the same for Theorem 1.1. It shows a linear growth in domination number with order. Let us compare the MGEO-P theoretical domination number upper bound mentioned within [**12**] and the DMGEO-P domination size in

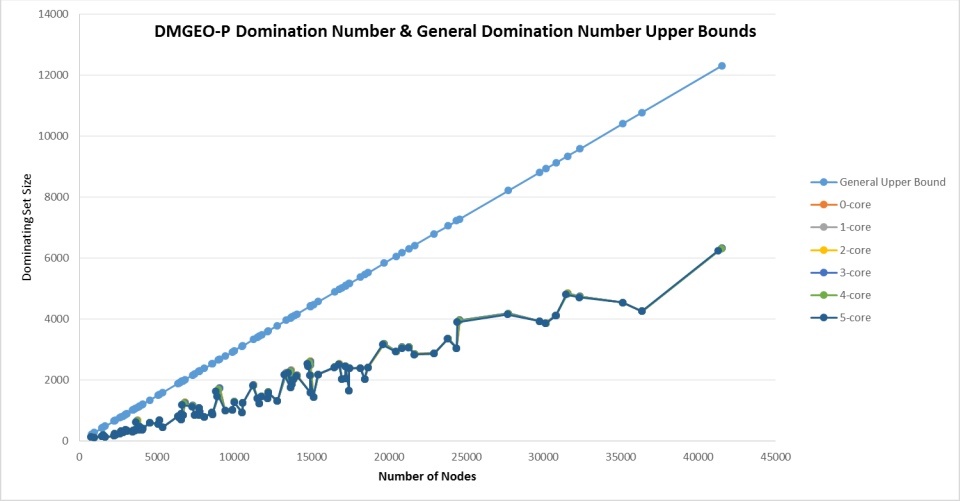


FIGURE 4.7. Domination number upper bounds of DMGEO-P samples corresponding to the Facebook 100 graphs with Theorem 1.1 upper bound ($\delta = 5$).

Figure 4.8. The resulting plots are similar in shape but not order. One should notice however that we only have the results of the DS-RAI algorithm. The DS-DC algorithm will result in a smaller set. An open question now is, does the DMGEO-P model follow Theorem 3.1? In particular, does the DMGEO-P model follow a sub-linear domination number upper bound?

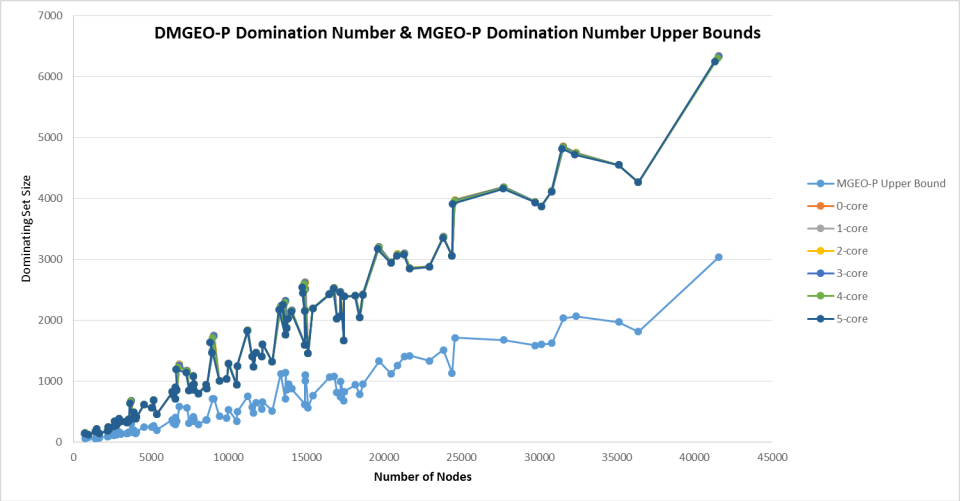


FIGURE 4.8. Domination number upper bounds of DMGEO-P samples corresponding to the Facebook 100 graphs with Theorem 1.1 upper bound ($\delta = 5$).

CHAPTER 5

Conclusions and Future Directions

5.1. Summary

In Chapter 2, we investigated the MGEO-P model for properties found in OSNs. Theorems were stated for the model's average degree, diameter, and power-law degree distribution. We simulated graphs for the model, and confirmed experimentally that these satisfy power-law degree distributions.

In Chapter 3, we began with some brief background and analysis of the Facebook 100 dataset, and then went on to computing domination sets for the k-core, where $1 \leq k \leq 5$, using the DS-DC and DS-RAI algorithms. In [**9**], they investigated the FB100 graphs and computed matching MGEO-P parameters for each institution within the set. With these parameters, MGEO-P samples were generated with their domination set computed using the DS-RAI and DS-DC algorithms. Recent theoretical results have suggested a sub-linear upper bound on the domination number in MGEO-P samples. We plotted the theoretical bound for the MGEO-P model using the parameters in [**9**], and the best estimation we have of the domination number for FB100 (from the DS-DC results). There is a correlation of the domination number of real Facebook datasets and graphs simulated from MGEO-P.

In our simulations of MGEO-P, we noticed that our samples did follow the theoretical bounds on the domination number. As mentioned in Chapter 3,

the constant term of the big Oh bound was not considered in our plots. Also, we used the parameters for the MGEO-P matching the datasets found in [**9**], and then percolated our graphs to match the number of edges of the dataset. This possibly explains why our generated samples are more dense and why our domination sets are smaller than the plotted function $f(n) = n^{\alpha+\beta}$.

In Chapter 4, the MGEO-P model was modified to incorporate a more geometric view, better approximating Blau space. Here we found that these sample's domination number follow closely along with the actual Facebook data and MGEO-P theoretical domination number upper bound. Some investigation should be done to explain why the DMGEO-P model is generating less dense graphs yet have a domination number that more closely resembles dense OSNs such as FB100.

5.2. Open Problems

There is much more work to be done in this area of research. Completing the DS-DC algorithm for the DMGEO-P samples will help us deduce if the model follows OSNs domination numbers. One can perform additional domination set algorithms to find the actual domination number of the Facebook 100 dataset. The ultimate goal would be to apply an algorithm for computing the exact domination number of OSNs (such algorithm is mentioned in [**42**]). One can look into *elite members* (that is, influential members in the social network) of networks and see what role they play in dominating sets. In addition, dominating sets can be widely used in "directors network" as a niche area in Corporate Finance. The PageRank of each network should

be computed, and the agents of the institution's domination set should be crossed referenced to see if there is any correlation with the agents of high PageRank and agents within the domination set. As mentioned in Chapter 4, the clustering coefficients for each graph should be computed and analyzed to see any patterns between the dominating agents and their clustering coefficient.

Another, possibly more complex problem that should be studied, is to consider models where the sphere of influence is not a ball in the corresponding metric space. The idea behind this is agents do not have a uniform radius of influence in m-dimensions, and the radius would depend on what dimension is being considered (for example, socio-demographic characteristics) for that agent. For example, if one dimension was income and another community impact, there can be many users with low level of income and high community impact, and vice versa.

Another important consideration to consider is the domination number in other OSNs datasets. Data from other on-line social networks (such as LinkedIn and Twitter) should all be studied to determine if they follow the theoretical MGEO-P domination number upper bound. If this is true, then the MGEO-P model will be a sound model to simulate both OSN properties and the domination number.

Another question is to determine why agents belong to dominating sets. There might be unique properties about these agents which can play an important role in OSNs. Learning and understanding how agents join or

leave dominating sets may allow us to control future network behaviour. These agents are informally the glue of the network, they have the most spread and value in network information diffusion.

In summary, the MGEO-P follows properties of OSNs and follows their domination number upper bound. Work in the thesis provides experimental evidence that MGEO-P samples do follow theoretical predicted values of dominating sets, and that OSNs follow the MGEO-P sub-linear domination number bound. In addition to introducing a modified model of the MGEO-P, samples of the DMGEO-P model possesses domination number with order similar to that found in OSNs.

APPENDIX A

Data Tables

A.1. MGEO-P Data Tables

Dataset Name	Nodes	Edges	Min Degree	Avg Degree	Max Degree	Density
Caltech36	769	16656	1	43	120	0.056
Reed98	962	18812	0	39	228	0.041
Haverford76	1446	59589	0	82	205	0.057
Simmons81	1518	32988	0	43	195	0.029
Swarthmore42	1659	61050	1	74	298	0.044
Amherst41	2235	90954	1	81	333	0.036
Bowdoin47	2252	84387	1	75	282	0.033
Hamilton46	2314	96394	1	83	429	0.036
Trinity100	2613	111996	0	86	311	0.033
USFCA72	2682	65252	0	49	378	0.018
Williams40	2790	112986	1	81	368	0.029
Oberlin44	2920	89912	0	62	339	0.021
Smith60	2970	97133	1	65	234	0.022
Wellesley22	2970	94899	0	64	340	0.022
Vassar85	3068	119161	1	78	355	0.025
Middlebury45	3075	124610	0	81	291	0.026
Pepperdine86	3445	152007	0	88	386	0.026
Colgate88	3482	155043	1	89	409	0.026
Santa74	3578	151747	0	85	375	0.024
Wesleyan43	3593	138035	0	77	427	0.021
Mich67	3748	81903	1	44	228	0.012
Bucknell39	3826	158864	0	83	354	0.022
Brandeis99	3898	137567	0	71	368	0.018
Howard90	4047	204850	0	101	382	0.025
Rice31	4087	184828	0	90	400	0.022
Rochester38	4563	161404	0	71	275	0.016
Lehigh96	5075	198347	1	78	351	0.015
Johns-Hopkins55	5180	186586	0	72	363	0.014
Wake73	5372	279191	1	104	447	0.019
American75	6386	217662	0	68	561	0.011
MIT8	6440	251252	0	78	379	0.012

TABLE A.1. General graph properties computed for MGEO-P samples of Facebook 100.

Dataset Name	Nodes	Edges	Min Degree	Avg Degree	Max Degree	Density
William77	6472	266378	0	82	490	0.013
UChicago30	6591	208103	0	63	508	0.010
Princeton12	6596	293320	0	89	487	0.013
Carnegie49	6637	249967	0	75	454	0.011
Tufts18	6682	249728	0	75	355	0.011
UC64	6833	155332	0	45	205	0.007
Vermont70	7324	191221	0	52	374	0.007
Emory27	7460	330014	0	88	448	0.012
Dartmouth6	7694	304076	1	79	436	0.010
Tulane29	7752	283918	1	73	291	0.009
WashU32	7755	367541	0	95	476	0.012
Villanova62	7772	314989	0	81	444	0.010
Vanderbilt48	8069	427832	0	106	531	0.013
Yale4	8578	405450	1	95	547	0.011
Brown11	8600	384526	0	89	672	0.010
UCSC68	8991	224584	0	50	285	0.006
Maine59	9069	243247	0	54	429	0.006
Georgetown15	9414	425638	0	90	619	0.010
Duke14	9895	506442	0	102	521	0.010
Bingham82	10004	362894	0	73	419	0.007
Mississippi66	10521	610911	0	116	744	0.011
Northwestern25	10567	488337	1	92	435	0.009
Cal65	11247	351358	1	62	219	0.006
BC17	11509	486967	0	85	505	0.007
Stanford3	11621	568330	1	98	516	0.008
Columbia2	11770	444333	1	76	804	0.006
Notre-Dame57	12155	541339	0	89	628	0.007
GWU54	12193	469528	1	77	761	0.006
Baylor93	12803	679817	1	106	471	0.008
USF51	13377	321214	0	48	351	0.004
Syracuse56	13653	543982	0	80	497	0.006
Temple83	13686	360795	0	53	676	0.004
UC61	13746	442174	0	64	402	0.005
Northeastern19	13882	381934	0	55	482	0.004
JMU79	14070	485564	0	69	548	0.005
UPenn7	14916	686501	0	92	536	0.006
UCSB37	14935	482224	0	65	386	0.004
UCF52	14940	428989	0	57	318	0.004
UCSD34	14948	443221	0	59	392	0.004
Harvard1	15126	824617	0	109	608	0.007

TABLE A.2. General graph properties computed for MGEO-P samples of Facebook 100.

Dataset Name	Nodes	Edges	Min Degree	Avg Degree	Max Degree	Density
MU78	15436	649449	0	84	456	0.005
UMass92	16516	519385	0	63	521	0.004
UC33	16808	522147	0	62	381	0.004
Tennessee95	16979	770659	0	91	779	0.005
UVA16	17196	789321	1	92	549	0.005
UConn91	17212	604870	0	70	469	0.004
Oklahoma97	17425	892528	0	102	656	0.006
USC35	17444	801853	0	92	467	0.005
UNC28	18163	766800	0	84	901	0.005
Auburn71	18448	973918	0	106	581	0.006
Cornell5	18660	790777	0	85	730	0.005
BU10	19700	637528	0	65	360	0.003
UCLA26	20467	747613	0	73	448	0.004
Maryland58	20871	744862	0	71	628	0.003
Virginia63	21325	698178	0	65	501	0.003
NYU9	21679	715715	0	66	1173	0.003
Berkeley13	22937	852444	0	74	916	0.003
Wisconsin87	23842	835952	0	70	841	0.003
UGA50	24389	1174057	1	96	572	0.004
Rutgers89	24580	784602	0	64	459	0.003
FSU53	27737	1034802	0	75	526	0.003
Indiana69	29747	1305765	0	88	620	0.003
Michigan23	30147	1176516	0	78	633	0.003
UIllinois20	30809	1264428	0	82	459	0.003
Texas80	31560	1219650	0	77	603	0.002
MSU24	32375	1118774	0	69	616	0.002
UF21	35123	1465660	0	83	680	0.002
Texas84	36371	1590655	0	87	802	0.002
Penn94	41554	1362229	0	66	986	0.002

TABLE A.3. General graph properties computed for MGEO-P samples of Facebook 100.

0-core		0-core		0-core		0-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
769	74	4563	376	10004	903	17196	1420
962	117	5075	394	10521	809	17212	1613
1446	98	5180	427	10567	717	17425	1493
1518	171	5372	419	11247	874	17444	1079
1659	149	6386	609	11509	919	18163	1435
2235	208	6440	516	11621	885	18448	1292
2252	206	6472	559	11770	515	18660	1457
2314	206	6591	644	12155	1068	19700	1751
2613	200	6596	547	12193	835	20467	1870
2682	343	6637	582	12803	908	20871	1832
2790	248	6682	576	13377	1531	21325	2165
2920	304	6833	697	13653	1111	21679	2348
2970	302	7324	774	13686	1423	22937	2307
2970	253	7460	620	13746	1371	23842	2231
3068	254	7694	663	13882	1599	24389	1665
3075	257	7752	614	14070	1182	24580	2187
3445	311	7755	559	14916	1267	27737	2252
3482	276	7772	641	14935	1327	29747	2225
3578	284	8069	598	14940	1338	30147	2894
3593	359	8578	630	14948	1347	30809	2613
3748	429	8600	804	15126	1160	31560	2119
3826	363	8991	995	15436	1066	32375	2987
3898	359	9069	777	16516	1749	35123	3007
4047	325	9414	824	16808	1769	36371	3320
4087	308	9895	686	16979	1348	41554	4116

TABLE A.4. Dominating set sizes from the DS-RAI algorithm for the 0-core of MGEO-P samples of the Facebook 100 network.

1-core		1-core		1-core		1-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
769	74	4562	375	10001	900	17196	1410
961	119	5075	398	10520	808	17201	1615
1445	97	5179	421	10567	730	17422	1484
1517	170	5372	419	11247	882	17443	1076
1659	149	6384	603	11508	922	18160	1441
2235	210	6439	507	11621	884	18446	1292
2252	210	6471	552	11770	499	18658	1465
2314	206	6590	641	12151	1051	19696	1748
2612	199	6595	550	12193	835	20459	1857
2679	339	6635	580	12803	908	20866	1815
2790	257	6680	571	13374	1538	21316	2152
2919	305	6828	695	13650	1110	21674	2335
2969	301	7323	775	13682	1418	22930	2299
2970	253	7456	618	13738	1366	23828	2202
3068	256	7694	663	13874	1586	24389	1666
3074	258	7752	603	14064	1177	24577	2160
3444	299	7754	547	14908	1257	27733	2241
3482	276	7771	642	14930	1316	29740	2215
3577	281	8066	603	14938	1337	30135	2880
3592	356	8578	630	14944	1347	30799	2616
3748	434	8595	799	15121	1157	31558	2127
3825	359	8988	986	15435	1078	32369	2995
3894	355	9065	770	16508	1734	35115	3006
4045	323	9411	819	16801	1760	36361	3315
4085	314	9893	684	16977	1345	41538	4107

TABLE A.5. Dominating set sizes from the DS-RAI algorithm for the 1-core of MGEO-P samples of the Facebook 100 network.

2-core		2-core		2-core		2-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
766	72	4547	361	9967	868	17126	1536
957	115	5063	385	10504	793	17152	1361
1443	95	5162	409	10544	693	17375	1437
1509	162	5358	410	11218	850	17415	1046
1653	145	6366	589	11479	898	18102	1384
2227	200	6426	495	11584	845	18418	1274
2248	203	6442	525	11759	487	18607	1409
2306	198	6555	607	12117	1019	19620	1678
2607	194	6575	524	12170	812	20376	1770
2662	323	6618	565	12778	878	20792	1763
2783	244	6661	553	13280	1450	21223	2070
2902	289	6793	653	13608	1356	21551	2230
2960	293	7276	724	13617	1073	22839	2205
2961	244	7433	596	13697	1327	23723	2110
3054	242	7667	626	13799	1522	24334	1610
3069	249	7735	591	14008	1117	24476	2061
3433	288	7738	531	14870	1264	27648	2152
3472	266	7745	619	14872	1220	29678	2152
3563	267	8047	577	14885	1281	30020	2775
3578	344	8554	759	14886	1292	30707	2510
3719	405	8559	615	15086	1122	31493	2056
3813	348	8947	961	15394	1029	32256	2906
3877	339	9035	750	16428	1668	34989	2908
4032	311	9385	793	16709	1685	36233	3185
4080	302	9868	659	16938	1312	41344	3928

TABLE A.6. Dominating set sizes from the DS-RAI algorithm for the 2-core of MGEO-P samples of the Facebook 100 network.

3-core		3-core		3-core		3-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
761	67	4537	351	9938	844	17045	1470
947	105	5047	371	10472	764	17101	1318
1442	94	5148	394	10512	679	17323	1386
1494	150	5345	394	11184	818	17387	1015
1649	142	6337	568	11445	859	18043	1331
2222	195	6412	488	11555	820	18377	1225
2239	197	6424	511	11756	479	18561	1374
2298	193	6518	573	12080	995	19558	1619
2605	192	6552	508	12151	793	20314	1722
2632	296	6594	543	12752	858	20700	1684
2771	229	6634	533	13197	1376	21109	1953
2887	275	6764	640	13527	1278	21424	2100
2945	284	7249	700	13575	1039	22717	2090
2948	231	7420	586	13639	1280	23624	2022
3039	226	7641	614	13709	1448	24298	1568
3057	237	7715	577	13965	1085	24400	2027
3419	288	7722	522	14811	1219	27555	2066
3462	257	7724	595	14820	1175	29600	2094
3552	257	8030	566	14826	1234	29913	2680
3563	332	8527	735	14840	1249	30601	2422
3695	381	8545	599	15049	1093	31438	1995
3806	342	8891	906	15356	1006	32119	2784
3856	320	8999	713	16356	1605	34878	2792
4021	300	9358	770	16635	1611	36107	3070
4071	303	9850	643	16897	1278	41137	3747

TABLE A.7. Dominating set sizes from the DS-RAI algorithm for the 3-core of MGEO-P samples of the Facebook 100 network.

4-core		4-core		4-core		4-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
756	66	4518	336	9901	812	16981	1428
935	99	5033	361	10446	742	17056	1279
1442	94	5132	381	10491	647	17276	1353
1484	149	5335	386	11164	797	17359	999
1638	132	6316	540	11410	832	17994	1282
2210	183	6392	471	11522	783	18336	1205
2233	194	6406	499	11748	478	18517	1330
2292	185	6493	554	12034	941	19486	1567
2598	186	6537	489	12117	759	20238	1650
2607	276	6574	525	12721	831	20620	1595
2761	229	6611	513	13103	1304	21025	1892
2864	258	6733	605	13465	1240	21317	2031
2935	273	7204	668	13541	1006	22602	2002
2939	222	7401	566	13586	1237	23528	1956
3029	219	7620	590	13618	1370	24247	1527
3043	230	7693	570	13907	1038	24310	1916
3409	281	7694	559	14756	1165	27465	2013
3455	254	7702	506	14768	1134	29519	2027
3547	253	8013	549	14774	1211	29798	2582
3550	320	8501	714	14793	1216	30504	2331
3669	354	8523	579	15020	1065	31371	1947
3789	327	8842	863	15322	969	31985	2634
3847	313	8966	686	16276	1535	34788	2731
4014	293	9325	738	16529	1530	35994	2967
4060	294	9832	630	16836	1228	40947	3576

TABLE A.8. Dominating set sizes from the DS-RAI algorithm for the 4-core of MGEO-P samples of the Facebook 100 network.

5-core		5-core		5-core		5-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
751	61	4503	324	9872	794	16907	1364
931	98	5015	346	10426	722	17006	1250
1437	89	5117	373	10475	631	17230	1313
1476	140	5323	383	11124	766	17337	973
1629	125	6290	519	11378	799	17942	1249
2203	177	6375	451	11487	760	18298	1176
2222	181	6382	482	11746	495	18472	1297
2289	185	6466	539	11992	906	19431	1530
2581	257	6520	479	12092	742	20174	1596
2589	179	6552	507	12703	810	20551	1552
2755	215	6590	497	13029	1267	20931	1818
2850	246	6698	587	13384	1180	21207	1947
2923	210	7163	643	13496	973	22509	1930
2926	267	7380	552	13524	1195	23437	1861
3015	207	7595	569	13539	1316	24196	1484
3032	225	7668	549	13866	999	24200	1864
3399	270	7670	534	14704	1131	27381	1948
3448	237	7686	492	14719	1161	29439	1955
3530	302	7999	537	14721	1152	29680	2480
3537	245	8474	688	14727	1090	30405	2259
3649	342	8497	553	14984	1022	31303	1895
3772	306	8787	817	15294	950	31845	2557
3827	299	8934	666	16180	1451	34668	2617
4005	286	9293	713	16443	1474	35851	2849
4048	281	9805	600	16785	1178	40769	3450

TABLE A.9. Dominating set sizes from the DS-RAI algorithm for the 5-core of MGEO-P samples of the Facebook 100 network.

0-core		0-core		0-core		0-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
769	49	4563	216	10004	524	17196	814
962	69	5075	241	10521	423	17212	942
1446	62	5180	254	10567	411	17425	788
1518	115	5372	234	11247	561	17444	664
1659	85	6386	333	11509	525	18163	820
2235	104	6440	291	11621	505	18448	717
2252	118	6472	313	11770	276	18660	832
2314	112	6591	373	12155	576	19700	1054
2613	119	6596	297	12193	470	20467	1084
2682	190	6637	320	12803	511	20871	1092
2790	137	6682	343	13377	894	21325	1185
2920	175	6833	441	13653	647	21679	1245
2970	151	7324	436	13686	823	22937	1211
2970	170	7460	341	13746	783	23842	1276
3068	151	7694	374	13882	885	24389	981
3075	150	7752	360	14070	719	24580	1331
3445	165	7755	318	14916	714	27737	1385
3482	160	7772	362	14935	791	29747	1287
3578	171	8069	337	14940	821	30147	1577
3593	199	8578	360	14948	820	30809	1494
3748	257	8600	445	15126	647	31560	1324
3826	186	8991	585	15436	656	32375	1701
3898	209	9069	520	16516	959	35123	1646
4047	181	9414	457	16808	979	36371	1785
4087	176	9895	404	16979	751	41554	2263

TABLE A.10. Dominating set sizes from the DS-DC algorithm for the 0-core of MGEO-P samples of the Facebook 100 network.

1-core		1-core		1-core		1-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
769	48	4562	215	10001	525	17196	812
961	68	5075	240	10520	424	17201	932
1445	60	5179	256	10567	413	17422	785
1517	115	5372	235	11247	559	17443	660
1659	85	6384	332	11508	521	18160	818
2235	104	6439	290	11621	505	18446	715
2252	118	6471	313	11770	276	18658	835
2314	112	6590	372	12151	571	19696	1050
2612	118	6595	300	12193	472	20459	1082
2679	188	6635	319	12803	509	20866	1088
2790	137	6680	341	13374	890	21316	1173
2919	174	6828	442	13650	647	21674	1238
2969	168	7323	439	13682	819	22930	1208
2970	153	7456	336	13738	773	23828	1253
3068	151	7694	376	13874	875	24389	980
3074	152	7752	361	14064	709	24577	1328
3444	164	7754	319	14908	705	27733	1382
3482	161	7771	362	14930	787	29740	1280
3577	170	8066	335	14938	824	30135	1568
3592	195	8578	358	14944	813	30799	1480
3748	257	8595	438	15121	639	31558	1327
3825	184	8988	580	15435	653	32369	1695
3894	205	9065	519	16508	950	35115	1630
4045	178	9411	454	16801	974	36361	1772
4085	171	9893	404	16977	748	41538	2252

TABLE A.11. Dominating set sizes from the DS-DC algorithm for the 1-core of MGEO-P samples of the Facebook 100 network.

2-core		2-core		2-core		2-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
766	49	4547	203	9967	501	17126	882
957	66	5063	232	10504	414	17152	791
1443	61	5162	247	10544	398	17375	763
1509	106	5358	228	11218	546	17415	646
1653	81	6366	319	11479	510	18102	783
2227	100	6426	283	11584	489	18418	696
2248	117	6442	297	11759	269	18607	802
2306	106	6555	347	12117	554	19620	1001
2607	114	6575	280	12170	466	20376	1024
2662	179	6618	312	12778	497	20792	1051
2783	133	6661	328	13280	842	21223	1134
2902	163	6793	420	13608	782	21551	1178
2960	164	7276	414	13617	622	22839	1155
2961	150	7433	326	13697	750	23723	1202
3054	147	7667	359	13799	838	24334	947
3069	147	7735	353	14008	684	24476	1270
3433	160	7738	310	14870	748	27648	1335
3472	149	7745	349	14872	689	29678	1246
3563	162	8047	321	14885	797	30020	1501
3578	188	8554	411	14886	785	30707	1421
3719	241	8559	353	15086	615	31493	1288
3813	180	8947	555	15394	640	32256	1626
3877	197	9035	507	16428	897	34989	1579
4032	171	9385	441	16709	936	36233	1714
4080	172	9868	388	16938	727	41344	2139

TABLE A.12. Dominating set sizes from the DS-DC algorithm for the 2-core of MGEO-P samples of the Facebook 100 network.

3-core		3-core		3-core		3-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
761	45	4537	203	9938	493	17045	849
947	65	5047	225	10472	402	17101	768
1442	60	5148	235	10512	384	17323	730
1494	95	5345	219	11184	531	17387	633
1649	80	6337	306	11445	495	18043	745
2222	97	6412	276	11555	472	18377	676
2239	111	6424	283	11756	269	18561	781
2298	103	6518	325	12080	541	19558	975
2605	113	6552	274	12151	453	20314	988
2632	158	6594	300	12752	481	20700	1005
2771	126	6634	312	13197	812	21109	1074
2887	155	6764	402	13527	740	21424	1113
2945	159	7249	405	13575	610	22717	1090
2948	144	7420	325	13639	726	23624	1151
3039	140	7641	345	13709	786	24298	938
3057	139	7715	346	13965	660	24400	1231
3419	155	7722	301	14811	728	27555	1281
3462	145	7724	334	14820	668	29600	1208
3552	154	8030	314	14826	772	29913	1442
3563	180	8527	397	14840	763	30601	1369
3695	227	8545	345	15049	596	31438	1260
3806	178	8891	528	15356	616	32119	1566
3856	191	8999	486	16356	864	34878	1516
4021	161	9358	423	16635	896	36107	1646
4071	170	9850	377	16897	717	41137	2045

TABLE A.13. Dominating set sizes from the DS-DC algorithm for the 3-core of MGEO-P samples of the Facebook 100 network.

4-core		4-core		4-core		4-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
756	50	4518	199	9901	471	17056	743
935	58	5033	215	11164	525	16981	819
1484	90	5132	231	10491	379	17276	712
1442	60	5335	216	11410	471	17359	621
1638	74	6493	311	10446	388	17994	718
2607	147	6316	300	11748	263	18517	751
2233	106	6392	262	12117	443	18336	662
2210	94	6574	294	11522	462	19486	937
2292	102	6406	277	12034	517	20238	959
2598	109	6537	266	12721	472	21025	1038
2864	144	6733	391	13103	762	20620	968
2939	137	7204	381	13465	708	21317	1056
2935	152	6611	306	13541	586	22602	1044
2761	121	7694	350	13618	751	23528	1103
3669	220	7620	334	13586	703	24247	912
3029	135	7401	311	13907	629	24310	1192
3043	130	7693	321	14774	745	27465	1242
3550	168	7702	290	14756	702	29798	1411
3409	152	8013	310	14768	647	29519	1175
3547	153	8523	337	14793	740	31371	1232
3455	143	8842	510	15020	592	30504	1343
3847	188	8966	473	16529	853	31985	1519
3789	171	8501	381	16276	837	34788	1476
4060	166	9325	404	15322	595	35994	1611
4014	159	9832	365	16836	693	40947	1966

TABLE A.14. Dominating set sizes from the DS-DC algorithm for the 4-core of MGEO-P samples of the Facebook 100 network.

5-core		5-core		5-core		5-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
751	47	4503	196	9872	457	16907	786
931	57	5015	209	10426	378	17006	721
1437	56	5117	224	10475	374	17230	694
1476	89	5323	208	11124	516	17337	605
1629	67	6290	289	11378	456	17942	698
2203	91	6375	256	11487	439	18298	646
2222	105	6382	265	11746	262	18472	741
2289	99	6466	304	11992	499	19431	918
2581	137	6520	258	12092	428	20174	941
2589	106	6552	290	12703	464	20551	944
2755	116	6590	298	13029	741	20931	1013
2850	140	6698	379	13384	673	21207	1019
2923	133	7163	365	13496	575	22509	1012
2926	150	7380	302	13524	690	23437	1068
3015	128	7595	331	13539	726	24196	897
3032	124	7668	310	13866	617	24200	1148
3399	146	7670	337	14704	684	27381	1213
3448	140	7686	284	14719	721	29439	1141
3530	157	7999	301	14721	717	29680	1357
3537	144	8474	373	14727	621	30405	1304
3649	205	8497	324	14984	572	31303	1199
3772	168	8787	491	15294	584	31845	1474
3827	185	8934	461	16180	812	34668	1415
4005	158	9293	394	16443	820	35851	1550
4048	160	9805	347	16785	665	40769	1892

TABLE A.15. Dominating set sizes from the DS-DC algorithm for the 5-core of MGEO-P samples of the Facebook 100 network.

Dataset Name	Nodes	Edges	Average Degree	Maximum Degree	Density
Caltech36	769	16656	43	248	0.056
Reed98	962	18812	39	313	0.041
Haverford76	1446	59589	82	375	0.057
Simmons81	1518	32988	43	300	0.029
Swarthmore42	1659	61050	74	577	0.044
Amherst41	2235	90954	81	467	0.036
Bowdoin47	2252	84387	75	670	0.033
Hamilton46	2314	96394	83	602	0.036
Trinity100	2613	111996	86	404	0.033
USFCA72	2682	65252	49	405	0.018
Williams40	2790	112986	81	610	0.029
Oberlin44	2920	89912	62	478	0.021
Smith60	2970	97133	65	349	0.022
Wellesley22	2970	94899	64	746	0.022
Vassar85	3068	119161	78	482	0.025
Middlebury45	3075	124610	81	473	0.026
Pepperdine86	3445	152007	88	674	0.026
Colgate88	3482	155043	89	773	0.026
Santa74	3578	151747	85	1129	0.024
Wesleyan43	3593	138035	77	549	0.021
Mich67	3748	81903	44	419	0.012
Bucknell39	3826	158864	83	506	0.022
Brandeis99	3898	137567	71	1972	0.018
Howard90	4047	204850	101	1215	0.025
Rice31	4087	184828	90	581	0.022
Rochester38	4563	161404	71	1224	0.016
Lehigh96	5075	198347	78	973	0.015
Johns-Hopkins55	5180	186586	72	886	0.014
Wake73	5372	279191	104	1341	0.019
American75	6386	217662	68	930	0.011
MIT8	6440	251252	78	708	0.012
William77	6472	266378	82	1124	0.013
UChicago30	6591	208103	63	1624	0.010
Princeton12	6596	293320	89	628	0.013
Carnegie49	6637	249967	75	840	0.011
Tufts18	6682	249728	75	827	0.011
UC64	6833	155332	45	660	0.007
Vermont70	7324	191221	52	864	0.007
Emory27	7460	330014	88	1095	0.012

TABLE A.16. General graph properties computed for Facebook 100.

A.2. Facebook 100 Data Tables

Dataset Name	Nodes	Edges	Average Degree	Maximum Degree	Density
Dartmouth6	7694	304076	79	948	0.010
Tulane29	7752	283918	73	1188	0.009
WashU32	7755	367541	95	1794	0.012
Villanova62	7772	314989	81	1183	0.010
Vanderbilt48	8069	427832	106	2041	0.013
Yale4	8578	405450	95	2517	0.011
Brown11	8600	384526	89	1075	0.010
UCSC68	8991	224584	50	454	0.006
Maine59	9069	243247	54	1011	0.006
Georgetown15	9414	425638	90	1235	0.009
Duke14	9895	506442	102	1887	0.010
Bingham82	10004	362894	73	553	0.007
Mississippi66	10521	610911	116	1691	0.011
Northwestern25	10567	488337	92	2105	0.009
Cal65	11247	351358	62	415	0.006
BC17	11509	486967	85	1377	0.007
Stanford3	11621	568330	98	1172	0.008
Columbia2	11770	444333	76	3375	0.006
Notre-Dame57	12155	541339	89	1344	0.007
GWU54	12193	469528	77	2002	0.006
Baylor93	12803	679817	106	2109	0.008
USF51	13377	321214	48	897	0.004
Syracuse56	13653	543982	80	1340	0.006
Temple83	13686	360795	53	1394	0.004
UC61	13746	442174	64	687	0.005
Northeastern19	13882	381934	55	968	0.004
JMU79	14070	485564	69	3274	0.005
UPenn7	14916	686501	92	1602	0.006
UCSB37	14935	482224	65	810	0.004
UCF52	14940	428989	57	4765	0.004
UCSD34	14948	443221	59	2165	0.004
Harvard1	15126	824617	109	1183	0.007
MU78	15436	649449	84	653	0.005
UMass92	16516	519385	63	3684	0.004
UC33	16808	522147	62	1415	0.004
Tennessee95	16979	770659	91	4943	0.005
UVA16	17196	789321	92	3182	0.005
UConn91	17212	604870	70	1709	0.004
Oklahoma97	17425	892528	102	2568	0.006
USC35	17444	801853	92	4459	0.005

TABLE A.17. General graph properties computed for Facebook 100 (continued).

Dataset Name	Nodes	Edges	Average Degree	Maximum Degree	Density
UNC28	18163	766800	84	3795	0.005
Auburn71	18448	973918	106	5160	0.006
Cornell5	18660	790777	85	3156	0.005
BU10	19700	637528	65	1819	0.003
UCLA26	20467	747613	73	1180	0.004
Maryland58	20871	744862	71	3784	0.003
Virginia63	21325	698178	65	7206	0.003
NYU9	21679	715715	66	2315	0.003
Berkeley13	22937	852444	74	3434	0.003
Wisconsin87	23842	835952	70	3484	0.003
UGA50	24389	1174057	96	2926	0.004
Rutgers89	24580	784602	64	1642	0.003
FSU53	27737	1034802	75	2555	0.003
Indiana69	29747	1305765	88	1358	0.003
Michigan23	30147	1176516	78	2031	0.003
UIllinois20	30809	1264428	82	4632	0.003
Texas80	31560	1219650	77	1796	0.002
MSU24	32375	1118774	69	5267	0.002
UF21	35123	1465660	83	8246	0.002
Texas84	36371	1590655	87	6312	0.002
Penn94	41554	1362229	66	4410	0.002

TABLE A.18. General graph properties computed for Facebook 100 (continued).

1-core		1-core		1-core		1-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
769	131	4563	610	10004	1405	17196	2442
962	150	5075	579	10521	1392	17212	2505
1446	143	5180	842	10567	1503	17425	2549
1518	221	5372	573	11247	1672	17444	2679
1659	187	6386	989	11509	1524	18163	2660
2235	240	6440	1118	11621	1967	18448	2310
2252	254	6472	774	11770	1972	18660	2628
2314	242	6591	1144	12155	1495	19700	2984
2613	267	6596	904	12193	1816	20467	3437
2682	458	6637	1056	12803	1598	20871	3183
2790	285	6682	951	13377	2670	21325	3275
2920	422	6833	1334	13653	1801	21679	3631
2970	378	7324	1182	13686	2658	22937	3739
2970	348	7460	1086	13746	2367	23842	3376
3068	321	7694	1364	13882	2263	24389	3017
3075	394	7752	1186	14070	1861	24580	4077
3445	466	7755	974	14916	2246	27737	4215
3482	348	7772	1051	14935	2388	29747	3972
3578	427	8069	979	14940	2534	30147	4719
3593	386	8578	1218	14948	2545	30809	4103
3748	650	8600	1186	15126	2513	31560	4678
3826	368	8991	1468	15436	1926	32375	4785
3898	475	9069	1508	16516	2500	35123	4919
4047	534	9414	1227	16808	2757	36371	5231
4087	494	9895	1369	16979	2370	41554	6755

TABLE A.19. Dominating set sizes from the DS-RAI algorithm for the 1-core of Facebook 100 graphs.

2-core		2-core		2-core		2-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
734	102	4484	535	9800	1231	16801	2084
927	127	4979	678	10270	1265	16832	2174
1428	126	5001	506	10356	1245	16942	2259
1468	185	5291	504	10996	1456	17032	2232
1644	176	6129	873	11087	1554	17799	2364
2199	210	6194	840	11185	1561	18138	2216
2214	227	6318	938	11253	1311	18187	2068
2281	216	6371	686	11842	1520	19176	2581
2575	232	6392	737	11951	1318	19818	2900
2589	379	6426	884	12636	1444	20311	2759
2743	245	6490	796	12892	2252	20789	2804
2857	349	6514	1087	13171	2255	20892	2974
2910	294	7104	996	13277	1983	22243	3164
2915	334	7260	912	13376	1564	23326	2955
3013	345	7370	1083	13511	1968	23902	3494
3034	298	7562	1041	13868	1665	23966	2644
3342	381	7570	889	14398	1816	27175	3707
3439	310	7578	826	14443	1965	29184	3483
3522	381	7903	854	14503	2177	29288	3998
3526	333	8292	996	14533	2054	30203	3592
3629	552	8380	1010	14600	2243	30898	4116
3794	398	8716	1240	15199	1730	31640	4155
3795	337	8813	1275	16168	2197	34514	4361
3956	458	9141	1019	16342	2375	35694	4614
4006	427	9621	1139	16638	2080	40458	5782

TABLE A.20. Dominating set sizes from the DS-RAI algorithm for the 2-core of Facebook 100 graphs.

3-core		3-core		3-core		3-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
699	83	4415	490	9665	1111	39703	5236
906	114	4840	587	10082	1128	16540	1910
1414	120	4945	475	10243	1144	16587	1986
1438	165	5242	467	10767	1332	16637	2013
1630	163	5953	761	10816	1345	16785	2040
2176	188	6061	746	10852	1355	17536	2164
2192	211	6153	824	11108	1215	17811	1988
2264	205	6266	924	11624	1362	18022	1939
2526	336	6275	665	11833	1224	18808	2317
2548	211	6281	784	12495	1995	19342	2566
2719	221	6308	646	12502	1345	19998	2525
2817	337	6379	725	12777	1980	20405	2652
2863	306	6964	898	12962	1743	20432	2532
2880	266	7121	811	13174	1424	21800	2841
2971	308	7180	941	13242	1777	22964	2693
3005	276	7403	922	13734	1572	23440	3158
3276	339	7444	724	14002	1649	23718	2461
3424	298	7457	808	14105	1635	26771	3344
3484	304	7802	778	14209	1974	28750	3607
3491	355	8114	871	14272	1855	28796	3197
3544	496	8250	918	14299	2036	29781	3259
3741	364	8517	1122	15031	1609	30457	3785
3774	323	8642	1157	15925	2005	31157	3773
3902	414	8984	918	16008	2110	34080	4019
3954	395	9459	1027	16408	1897	35240	4239

TABLE A.21. Dominating set sizes from the DS-RAI algorithm for the 3-core of Facebook 100 graphs.

4-core		4-core		4-core		4-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
684	75	4364	454	9555	1049	16337	1775
891	103	4765	539	9925	1034	16365	1842
1408	113	4895	439	10145	1067	16392	1866
1411	150	5203	444	10518	1186	16570	1901
1614	155	5842	689	10555	1206	17318	2008
2153	179	5954	677	10639	1247	17583	1840
2170	199	5994	740	10993	1134	17878	1837
2241	187	6070	823	11465	1255	18530	2162
2470	304	6181	604	11727	1152	18985	2345
2530	200	6189	718	12170	1815	19712	2322
2701	214	6235	603	12397	1271	20064	2436
2772	312	6291	674	12442	1769	20123	2338
2831	296	6840	812	12697	1622	21430	2620
2851	243	7020	742	12975	1619	22661	2499
2936	285	7039	849	13006	1317	23047	2916
2990	267	7286	864	13608	1487	23511	2310
3224	302	7367	754	13689	1471	26435	3145
3404	287	7372	685	13862	1468	28272	3313
3455	334	7701	719	13912	1802	28486	2971
3460	291	7970	783	14025	1704	29483	3061
3462	459	8131	840	14047	1872	30046	3537
3692	336	8351	1032	14886	1511	30750	3500
3744	300	8496	1060	15692	1863	33712	3791
3858	386	8869	848	15723	1941	34844	3977
3905	366	9291	944	16224	1769	39081	4825

TABLE A.22. Dominating set sizes from the DS-RAI algorithm for the 4-core of Facebook 100 graphs.

5-core		5-core		5-core		5-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
674	68	4319	423	9459	991	16159	1731
879	97	4687	503	9803	965	16174	1664
1392	140	4856	413	10065	1013	16228	1765
1399	106	5168	416	10297	1076	16408	1795
1603	150	5746	633	10342	1092	17108	1886
2138	162	5844	623	10470	1137	17356	1717
2154	186	5877	678	10883	1067	17746	1741
2225	176	5898	767	11318	1178	18238	1968
2424	277	6089	634	11635	1105	18662	2169
2515	195	6111	559	11848	1645	19449	2184
2676	189	6182	566	12162	1618	19716	2257
2736	291	6194	633	12308	1217	19890	2205
2792	261	6726	753	12443	1457	21077	2389
2829	234	6909	776	12733	1492	22413	2336
2905	281	6942	694	12867	1240	22679	2690
2968	248	7165	794	13424	1346	23326	2196
3193	281	7278	702	13499	1402	26107	2954
3380	267	7300	638	13654	1661	27848	3070
3387	420	7625	681	13676	1375	28221	2828
3426	316	7870	744	13782	1575	29219	2893
3427	275	8063	797	13802	1737	29683	3321
3653	314	8174	937	14769	1441	30372	3273
3715	280	8369	985	15417	1792	33323	3555
3816	356	8754	792	15502	1753	34518	3770
3867	338	9162	893	16052	1682	38475	4521

TABLE A.23. Dominating set sizes from the DS-RAI algorithm for the 5-core of Facebook 100 graphs.

1-core		1-core		1-core		1-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
769	65	4563	297	10004	754	17196	1215
962	86	5075	293	10521	671	17212	1300
1446	71	5180	455	10567	754	17425	1297
1518	123	5372	295	11247	942	17444	1336
1659	93	6386	556	11509	783	18163	1300
2235	126	6440	581	11621	1047	18448	1127
2252	125	6472	371	11770	1034	18660	1369
2314	126	6591	596	12155	724	19700	1591
2613	146	6596	474	12193	924	20467	1918
2682	267	6637	514	12803	779	20871	1679
2790	142	6682	524	13377	1465	21325	1682
2920	210	6833	775	13653	967	21679	1911
2970	187	7324	672	13686	1400	22937	1896
2970	207	7460	533	13746	1316	23842	1774
3068	176	7694	657	13882	1243	24389	1571
3075	203	7752	610	14070	970	24580	2192
3445	236	7755	507	14916	1168	27737	2155
3482	177	7772	554	14935	1308	29747	2081
3578	217	8069	511	14940	1388	30147	2476
3593	220	8578	611	14948	1401	30809	2150
3748	370	8600	621	15126	1306	31560	2515
3826	196	8991	839	15436	1017	32375	2543
3898	264	9069	806	16516	1295	35123	2509
4047	290	9414	656	16808	1495	36371	2591
4087	247	9895	684	16979	1198	41554	3506

TABLE A.24. Dominating set sizes from the DS-DC algorithm for the 1-core of Facebook 100 graphs.

2-core		2-core		2-core		2-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
734	51	4484	251	9800	661	16801	998
927	70	4979	331	10270	593	16832	1099
1428	59	5001	258	10356	580	16942	1091
1468	94	5291	251	10996	800	17032	1086
1644	82	6129	411	11087	756	17799	1092
2199	104	6194	457	11185	782	18138	1113
2214	107	6318	447	11253	644	18187	969
2281	110	6371	324	11842	771	19176	1328
2575	123	6392	362	11951	613	19818	1581
2589	221	6426	407	12636	688	20311	1387
2743	120	6490	413	12892	1218	20789	1408
2857	169	6514	622	13171	1118	20892	1520
2910	151	7104	541	13277	1062	22243	1536
2915	184	7260	440	13376	835	23326	1508
3013	163	7370	492	13511	1044	23902	1815
3034	159	7562	525	13868	850	23966	1346
3342	176	7570	443	14398	898	27175	1906
3439	151	7578	411	14443	943	29184	1789
3522	178	7903	421	14503	1170	29288	2038
3526	186	8292	477	14533	1107	30203	1829
3629	315	8380	505	14600	1222	30898	2180
3794	206	8716	698	15199	896	31640	2148
3795	176	8813	661	16168	1110	34514	2167
3956	231	9141	518	16342	1246	35694	2253
4006	218	9621	519	16638	1013	40458	2927

TABLE A.25. Dominating set sizes from the DS-DC algorithm for the 2-core of Facebook 100 graphs.

3-core		3-core		3-core		3-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
699	43	4415	225	9665	607	16540	884
906	64	4840	273	10082	508	16587	985
1414	56	4945	237	10243	536	16637	975
1438	85	5242	231	10767	635	16785	985
1630	73	5953	338	10816	741	17536	1024
2176	94	6061	401	10852	667	17811	981
2192	97	6153	392	11108	594	18022	882
2264	105	6266	515	11624	673	18808	1175
2526	193	6275	319	11833	563	19342	1365
2548	107	6281	356	12495	1054	19998	1266
2719	110	6308	298	12502	620	20405	1314
2817	156	6379	365	12777	960	20432	1258
2863	155	6964	469	12962	926	21800	1337
2880	145	7121	376	13174	752	22964	1368
2971	142	7180	407	13242	936	23440	1637
3005	146	7403	448	13734	804	23718	1235
3276	143	7444	352	14002	783	26771	1740
3424	143	7457	392	14105	779	28750	1825
3484	165	7802	374	14209	1071	28796	1610
3491	169	8114	404	14272	998	29781	1669
3544	285	8250	452	14299	1117	30457	2020
3741	183	8517	616	15031	836	31157	1949
3774	171	8642	587	15925	1007	34080	1987
3902	208	8984	463	16008	1117	35240	2059
3954	198	9459	463	16408	911	39703	2618

TABLE A.26. Dominating set sizes from the DS-DC algorithm for the 3-core of Facebook 100 graphs.

4-core		4-core		4-core		4-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
684	35	4364	208	9555	563	16337	829
891	61	4765	254	9925	442	16365	883
1408	56	4895	224	10145	495	16392	928
1411	78	5203	215	10518	562	16570	903
1614	71	5842	304	10555	580	17318	941
2153	85	5954	366	10639	675	17583	898
2170	92	5994	340	10993	548	17878	832
2241	92	6070	441	11465	604	18530	1084
2470	171	6181	294	11727	526	18985	1253
2530	103	6189	333	12170	945	19712	1166
2701	105	6235	282	12397	588	20064	1194
2772	150	6291	339	12442	848	20123	1165
2831	146	6840	432	12697	831	21430	1230
2851	132	7020	340	12975	861	22661	1262
2936	133	7039	365	13006	680	23047	1508
2990	148	7286	404	13608	745	23511	1147
3224	134	7367	362	13689	696	26435	1624
3404	136	7372	328	13862	712	28272	1650
3455	157	7701	338	13912	967	28486	1503
3460	158	7970	356	14025	886	29483	1563
3462	260	8131	412	14047	1036	30046	1902
3692	169	8351	556	14886	791	30750	1807
3744	162	8496	544	15692	931	33712	1857
3858	190	8869	430	15723	1027	34844	1916
3905	178	9291	420	16224	836	39081	2407

TABLE A.27. Dominating set sizes from the DS-DC algorithm for the 4-core of Facebook 100 graphs.

5-core		5-core		5-core		5-core	
Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC	Nodes	DS-DC
674	33	4319	191	9459	534	16159	810
879	56	4687	230	9803	407	16174	787
1392	69	4856	211	10065	472	16228	886
1399	51	5168	204	10297	491	16408	853
1603	69	5746	282	10342	523	17108	880
2138	80	5844	334	10470	624	17356	842
2154	90	5877	303	10883	509	17746	786
2225	90	5898	403	11318	562	18238	1000
2424	158	6089	307	11635	503	18662	1156
2515	96	6111	275	11848	866	19449	1078
2676	91	6182	264	12162	770	19716	1082
2736	137	6194	312	12308	547	19890	1081
2792	138	6726	403	12443	760	21077	1122
2829	125	6909	320	12733	794	22413	1190
2905	124	6942	323	12867	642	22679	1406
2968	137	7165	380	13424	630	23326	1106
3193	126	7278	338	13499	717	26107	1552
3380	126	7300	303	13654	878	27848	1536
3387	242	7625	316	13676	651	28221	1428
3426	150	7870	336	13782	822	29219	1492
3427	154	8063	390	13802	960	29683	1792
3653	155	8174	512	14769	756	30372	1679
3715	154	8369	502	15417	944	33323	1744
3816	173	8754	406	15502	865	34518	1804
3867	166	9162	381	16052	782	38475	2228

TABLE A.28. Dominating set sizes from the DS-DC algorithm for the 5-core of Facebook 100 graphs.

A.3. DMGEO-P Data Tables

Dataset Name	Nodes	Edges	Min Degree	Avg Degree	Max Degree	Density
Caltech36	769	4142	1	11	28	0.014
Reed98	962	7564	4	16	68	0.016
Haverford76	1446	14927	8	21	52	0.014
Simmons81	1518	10300	2	14	61	0.009
Swarthmore42	1659	21837	9	26	78	0.016
Amherst41	2235	32673	9	29	130	0.013
Bowdoin47	2252	23851	8	21	86	0.009
Hamilton46	2314	30769	9	27	134	0.011
Trinity100	2613	30343	8	23	95	0.009
USFCA72	2682	22240	4	17	114	0.006
Williams40	2790	32770	8	23	112	0.008
Oberlin44	2920	28940	4	20	93	0.007
Smith60	2970	26589	3	18	68	0.006
Wellesley22	2970	29984	5	20	105	0.007
Vassar85	3068	32678	7	21	94	0.007
Middlebury45	3075	34063	6	22	71	0.007
Pepperdine86	3445	44192	8	26	112	0.007
Colgate88	3482	42672	10	25	92	0.007
Santa74	3578	42130	8	24	79	0.007
Wesleyan43	3593	43480	7	24	122	0.007
Mich67	3748	21174	1	11	52	0.003
Bucknell39	3826	46528	9	24	93	0.006
Brandeis99	3898	36284	4	19	91	0.005
Howard90	4047	56430	11	28	103	0.007
Rice31	4087	49878	9	24	84	0.006
Rochester38	4563	37545	3	16	86	0.004
Lehigh96	5075	53745	4	21	73	0.004
Johns-Hopkins55	5180	44719	4	17	102	0.003
Wake73	5372	78506	10	29	134	0.005
American75	6386	55080	2	17	115	0.003
MIT8	6440	58792	4	18	98	0.003
William77	6472	64734	5	20	91	0.003
UChicago30	6591	53290	3	16	108	0.002

TABLE A.29. General graph properties computed for DMGEO-P Facebook 100 samples.

Dataset Name	Nodes	Edges	Min Degree	Avg Degree	Max Degree	Density
Princeton12	6596	71730	5	22	116	0.003
Carnegie49	6637	62763	5	19	131	0.003
Tufts18	6682	59879	5	18	80	0.003
UC64	6833	36166	1	11	41	0.002
Vermont70	7324	48241	1	13	93	0.002
Emory27	7460	80004	4	21	81	0.003
Dartmouth6	7694	74083	5	19	104	0.003
Tulane29	7752	63654	2	16	70	0.002
WashU32	7755	84762	6	22	139	0.003
Villanova62	7772	74730	4	19	102	0.002
Vanderbilt48	8069	104241	4	26	155	0.003
Yale4	8578	93824	6	22	102	0.003
Brown11	8600	102424	5	24	170	0.003
UCSC68	8991	55744	1	12	79	0.001
Maine59	9069	48694	1	11	61	0.001
Georgetown15	9414	107699	5	23	159	0.002
Duke14	9895	117759	6	24	108	0.002
Bingham82	10004	88035	3	18	75	0.002
Mississippi66	10521	155807	9	30	152	0.003
Northwestern25	10567	108453	4	21	123	0.002
Cal65	11247	74532	2	13	52	0.001
BC17	11509	113743	4	20	119	0.002
Stanford3	11621	134172	7	23	121	0.002
Columbia2	11770	105214	4	18	205	0.002
Notre-Dame57	12155	123570	4	20	164	0.002
GWU54	12193	104158	3	17	141	0.001
Baylor93	12803	155582	6	24	132	0.002
USF51	13377	82776	1	12	104	0.001
Syracuse56	13653	126100	4	18	88	0.001
Temple83	13686	81345	1	12	135	0.001
UC61	13746	111133	1	16	111	0.001
Northeastern19	13882	98728	2	14	134	0.001

TABLE A.30. General graph properties computed for DMGEO-P Facebook 100 samples (continued).

Dataset Name	Nodes	Edges	Min Degree	Avg Degree	Max Degree	Density
JMU79	14070	100381	2	14	102	0.001
UPenn7	14916	173402	6	23	119	0.002
UCSB37	14935	111503	3	15	95	0.001
UCF52	14940	87574	0	12	64	0.001
UCSD34	14948	92593	0	12	89	0.001
Harvard1	15126	198767	5	26	122	0.002
MU78	15436	126959	2	16	99	0.001
UMass92	16516	123744	2	15	150	0.001
UC33	16808	122404	1	15	109	0.001
Tennessee95	16979	166551	4	20	161	0.001
UVA16	17196	173907	3	20	128	0.001
UConn91	17212	133615	2	16	107	0.001
Oklahoma97	17425	230563	7	26	169	0.002
USC35	17444	151482	3	17	82	0.001
UNC28	18163	154789	3	17	178	0.001
Auburn71	18448	202971	4	22	188	0.001
Cornell5	18660	166569	2	18	122	0.001
BU10	19700	131217	1	13	82	0.001
UCLA26	20467	163294	2	16	79	0.001
Maryland58	20871	157377	2	15	99	0.001
Virginia63	21325	161292	2	15	159	0.001
NYU9	21679	179477	2	17	243	0.001
Berkeley13	22937	206236	3	18	229	0.001
Wisconsin87	23842	186132	2	16	159	0.001
UGA50	24389	232397	2	19	108	0.001
Rutgers89	24580	162784	2	13	106	0.001
FSU53	27737	206406	1	15	83	0.001
Indiana69	29747	261927	2	18	120	0.001
Michigan23	30147	270614	3	18	139	0.001
UIllinois20	30809	269799	3	18	108	0.001
Texas80	31560	227070	1	14	95	0.000
MSU24	32375	244658	1	15	139	0.000
UF21	35123	316021	2	18	205	0.001
Texas84	36371	368676	4	20	198	0.001
Penn94	41554	288212	1	14	245	0.000

TABLE A.31. General graph properties computed for DMGEO-P Facebook 100 samples (continued).

0-core		0-core		0-core		0-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
769	153	4563	612	10004	1292	17196	2063
962	119	5075	561	10521	937	17212	2467
1446	171	5180	687	10567	1249	17425	1665
1518	214	5372	452	11247	1839	17444	2392
1659	142	6386	823	11509	1407	18163	2403
2235	180	6440	796	11621	1239	18448	2047
2252	248	6472	769	11770	1466	18660	2420
2314	209	6591	901	12155	1402	19700	3204
2613	257	6596	711	12193	1607	20467	2948
2682	337	6637	842	12803	1319	20871	3083
2790	281	6682	854	13377	2240	21325	3096
2920	325	6833	1273	13653	1760	21679	2859
2970	384	7324	1172	13686	2323	22937	2880
2970	337	7460	850	13746	1883	23842	3371
3068	337	7694	928	13882	2060	24389	3057
3075	345	7752	1081	14070	2161	24580	3973
3445	320	7755	859	14916	1598	27737	4188
3482	353	7772	955	14935	2164	29747	3943
3578	370	8069	795	14940	2622	30147	3872
3593	344	8578	946	14948	2520	30809	4123
3748	682	8600	875	15126	1458	31560	4854
3826	378	8991	1502	15436	2195	32375	4750
3898	487	9069	1748	16516	2434	35123	4551
4047	378	9414	1008	16808	2533	36371	4265
4087	413	9895	1035	16979	2030	41554	6336

TABLE A.32. Dominating set sizes from the DS-RAI algorithm for the 0-core of DMGEO-P Facebook 100 graphs.

1-core		1-core		1-core		1-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
769	153	4563	612	10004	1292	17196	2063
962	119	5075	561	10521	937	17212	2467
1446	171	5180	687	10567	1249	17425	1665
1518	214	5372	452	11247	1839	17444	2392
1659	142	6386	823	11509	1407	18163	2403
2235	180	6440	796	11621	1239	18448	2047
2252	248	6472	769	11770	1466	18660	2420
2314	209	6591	901	12155	1402	19700	3204
2613	257	6596	711	12193	1607	20467	2948
2682	337	6637	842	12803	1319	20871	3083
2790	281	6682	854	13377	2240	21325	3096
2920	325	6833	1273	13653	1760	21679	2859
2970	384	7324	1172	13686	2323	22937	2880
2970	337	7460	850	13746	1883	23842	3371
3068	337	7694	928	13882	2060	24389	3057
3075	345	7752	1081	14070	2161	24580	3973
3445	320	7755	859	14916	1598	27737	4188
3482	353	7772	955	14935	2164	29747	3943
3578	370	8069	795	14939	2621	30147	3872
3593	344	8578	946	14947	2519	30809	4123
3748	682	8600	875	15126	1458	31560	4854
3826	378	8991	1502	15436	2195	32375	4750
3898	487	9069	1748	16516	2434	35123	4551
4047	378	9414	1008	16808	2533	36371	4265
4087	413	9895	1035	16979	2030	41554	6336

TABLE A.33. Dominating set sizes from the DS-RAI algorithm for the 1-core of DMGEO-P Facebook 100 graphs.

2-core		2-core		2-core		2-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
768	152	4563	612	10004	1292	17196	2063
962	119	5075	561	10521	937	17212	2467
1446	171	5180	687	10567	1249	17425	1665
1518	214	5372	452	11247	1839	17444	2392
1659	142	6386	823	11509	1407	18163	2403
2235	180	6440	796	11621	1239	18448	2047
2252	248	6472	769	11770	1466	18660	2420
2314	209	6591	901	12155	1402	19699	3203
2613	257	6596	711	12193	1607	20467	2948
2682	337	6637	842	12803	1319	20871	3083
2790	281	6682	854	13372	2237	21325	3096
2920	325	6832	1272	13653	1760	21679	2859
2970	384	7322	1170	13681	2320	22937	2880
2970	337	7460	850	13745	1882	23842	3371
3068	337	7694	928	13882	2060	24389	3057
3075	345	7752	1081	14070	2161	24580	3973
3445	320	7755	859	14916	1598	27736	4187
3482	353	7772	955	14935	2164	29747	3943
3578	370	8069	795	14936	2619	30147	3872
3593	344	8578	946	14941	2515	30809	4123
3745	680	8600	875	15126	1458	31559	4853
3826	378	8989	1500	15436	2195	32373	4748
3898	487	9064	1745	16516	2434	35123	4551
4047	378	9414	1008	16807	2533	36371	4265
4087	413	9895	1035	16979	2030	41552	6334

TABLE A.34. Dominating set sizes from the DS-RAI algorithm for the 2-core of DMGEO-P Facebook 100 graphs.

3-core		3-core		3-core		3-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
764	149	4563	612	10004	1292	17196	2063
962	119	5075	561	10521	937	17211	2466
1446	171	5180	687	10567	1249	17425	1665
1517	214	5372	452	11245	1839	17444	2392
1659	142	6385	822	11509	1407	18163	2403
2235	180	6440	796	11621	1239	18448	2047
2252	248	6472	769	11770	1466	18659	2420
2314	209	6591	901	12155	1402	19693	3202
2613	257	6596	711	12193	1607	20466	2947
2682	337	6637	842	12803	1319	20869	3081
2790	281	6682	854	13364	2234	21324	3095
2920	325	6812	1258	13653	1760	21676	2856
2970	384	7321	1170	13670	2317	22937	2880
2970	337	7460	850	13745	1882	23838	3369
3068	337	7694	928	13878	2059	24388	3056
3075	345	7751	1084	14066	2159	24569	3966
3445	320	7755	859	14916	1598	27735	4186
3482	353	7772	955	14935	2164	29746	3942
3578	370	8069	795	14923	2611	30147	3872
3593	344	8578	946	14928	2509	30809	4123
3735	675	8600	875	15126	1458	31552	4846
3826	378	8983	1497	15435	2194	32368	4743
3898	487	9051	1739	16515	2434	35122	4551
4047	378	9414	1008	16804	2532	36371	4265
4087	413	9895	1035	16979	2030	41541	6327

TABLE A.35. Dominating set sizes from the DS-RAI algorithm for the 3-core of DMGEO-P Facebook 100 graphs.

4-core		4-core		4-core		4-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
760	147	4562	611	10003	1292	17195	2063
962	119	5075	561	10521	937	17204	2460
1446	171	5180	687	10567	1249	17425	1665
1515	213	5372	452	11233	1832	17443	2392
1659	142	6385	822	11509	1407	18159	2400
2235	180	6440	796	11621	1239	18448	2047
2252	248	6472	769	11770	1466	18655	2419
2314	209	6590	900	12155	1402	19667	3193
2613	257	6596	711	12192	1607	20462	2945
2682	337	6637	842	12803	1319	20859	3075
2790	281	6682	854	13330	2214	21315	3090
2920	325	6752	1228	13653	1760	21673	2855
2969	383	7310	1163	13623	2293	22935	2878
2970	337	7460	850	13741	1879	23828	3361
3068	337	7694	928	13863	2049	24386	3055
3075	345	7750	1084	14051	2152	24532	3951
3445	320	7755	859	14916	1598	27721	4181
3482	353	7772	955	14929	2161	29744	3940
3578	370	8069	795	14873	2589	30146	3871
3593	344	8578	946	14895	2502	30804	4119
3707	664	8600	875	15126	1458	31532	4835
3826	378	8965	1489	15434	2194	32355	4737
3898	487	8984	1709	16505	2430	35121	4550
4047	378	9414	1008	16792	2526	36371	4265
4087	413	9895	1035	16979	2030	41485	6303

TABLE A.36. Dominating set sizes from the DS-RAI algorithm for the 4-core of DMGEO-P Facebook 100 graphs.

5-core		5-core		5-core		5-core	
Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI	Nodes	DS-RAI
743	142	4559	610	10001	1290	17194	2063
960	119	5073	560	10521	937	17179	2454
1446	171	5179	687	10565	1249	17425	1665
1508	212	5372	452	11202	1820	17439	2390
1659	142	6384	821	11508	1406	18156	2399
2235	180	6439	795	11621	1239	18447	2046
2252	248	6472	769	11766	1464	18653	2419
2314	209	6587	900	12154	1401	19600	3170
2613	257	6596	711	12190	1606	20451	2936
2678	336	6637	842	12803	1319	20834	3058
2790	281	6682	854	13244	2175	21286	3079
2919	324	6609	1194	13650	1758	21652	2846
2969	383	7265	1137	13488	2253	22929	2875
2970	337	7459	851	13731	1872	23807	3355
3068	337	7694	928	13819	2027	24382	3053
3075	345	7747	1082	14029	2144	24432	3910
3445	320	7755	859	14916	1598	27677	4160
3482	353	7771	955	14904	2155	29736	3935
3578	370	8068	794	14732	2536	30140	3866
3593	344	8578	946	14782	2450	30793	4112
3647	635	8600	875	15126	1458	31460	4815
3826	378	8910	1467	15426	2192	32296	4718
3897	487	8815	1640	16475	2424	35118	4549
4047	378	9414	1008	16756	2516	36369	4263
4087	413	9895	1035	16978	2030	41305	6251

TABLE A.37. Dominating set sizes from the DS-RAI algorithm for the 5-core of DMGEO-P Facebook 100 graphs.

APPENDIX B

Program Code

The following two sections describe the code used during the research of this thesis. The first section provides Java code that was used along with the JUNG 2.0.1 framework to generate visual MGEO-P and DMGEO-P samples. The second section is the C++ code used injunction with the Stanford Network Analysis Platform (SNAP).

The MGEO-P simulations were generated with a Quad Core i7-4700MQ 2.40 GHz 16 GB laptop. The computation of the domination number of the data sets were done using RAMLab equipment. The hardware used during DMGEO-P experiments, simulations and domination, was a 12-core (2x 6-core) Xeon X5690 3.46 GHz with 96 GB RAM workstation.

An online simulator for the G(n, p), G(n, r) and MGEO-P(n, m, α, β, p) models were created for presentation purposes.

Simulator is located at http://www.math.ryerson.ca/people/mlozier/.

B.1. JUNG Code (Java)

createGUI.java

```
1 /**
2  * @author Marc
3  */
4 package samplegraph;
5
```

```
6 import edu.uci.ics.jung.algorithms.layout.*;
7 import edu.uci.ics.jung.graph.*;
8 import edu.uci.ics.jung.visualization.*;
9 import edu.uci.ics.jung.visualization.control.CrossoverScalingControl;
10 import edu.uci.ics.jung.visualization.control.ScalingControl;
11 import edu.uci.ics.jung.visualization.decorators.ToStringLabeller;
12 import edu.uci.ics.jung.visualization.renderers.Renderer.VertexLabel.Position;
13 import graphpackage.*;
14 import java.awt.BorderLayout;
15 import java.awt.Color;
16 import java.awt.Container;
17 import java.awt.Dimension;
18 import java.awt.GridLayout;
19 import java.awt.Paint;
20 import java.awt.event.*;
21 import java.awt.geom.Point2D;
22 import java.awt.image.BufferedImage;
23 import java.io.BufferedWriter;
24 import java.io.File;
25 import java.io.FileWriter;
26 import java.io.IOException;
27 import java.text.DateFormat;
28 import java.text.DecimalFormat;
29 import java.text.SimpleDateFormat;
30 import java.util.ArrayList;
31 import java.util.Arrays;
32 import java.util.Comparator;
33 import java.util.Date;
34 import java.util.List;
35 import javax.imageio.ImageIO;
36 import javax.swing.*;
```

```
37 import org.apache.commons.collections15.Transformer;
38
39 public class createGUI extends JApplet implements ActionListener,
       MouseWheelListener {
40     //constants for action commands
41     protected final static int GNP_GRAPH = 1;
42     protected final static int GNR_GRAPH = 2;
43     protected final static String CREATE_GRAPH = "new_graph";
44     protected final static String CREATE_GRAPH2 = "new_graph2";
45     protected final static String SAVE_GRAPH_IMAGE = "save_graph_img";
46     protected final static String EXPORT_GRAPH_DATA = "export_graph_data";
47     protected final static String GRAPH_DEGREE_DIST = "DEGREE_DIST";
48     protected final static String GRAPH_DIAMETER = "GRAPH_DIAMETER";
49     protected final static String GRAPH_DSRAI_PHASE1 = "GRAPH_DSRAI1";
50     protected final static String GRAPH_DSRAI_PHASE2 = "GRAPH_DSRAI2";
51     protected final static String GRAPH_DSRAI_PHASE3 = "GRAPH_DSRAI3";
52     protected final static String GRAPH_DSDC = "GRAPH_DSDC";
53     protected final static String MGEOP_GRAPH = "MGEOP_GRAPH";
54     protected final static String GRAPH_1CORE = "GRAPH_1CORE";
55     protected final static String GRAPH_2CORE = "GRAPH_2CORE";
56     protected final static String GRAPH_3CORE = "GRAPH_3CORE";
57     protected final static String GRAPH_KCORE = "GRAPH_KCORE";
58     // controlling class
59
60     private static final JFrame frame = new JFrame("");
61     private static final JFrame graphFrame = new JFrame("");
62     private static JComponent panel1, panel2, panel3, panel4, panel5, panel6,
       panel7;
63     private static final JTabbedPane tabbedPane = new JTabbedPane();
64     private static final createGUI gui = new createGUI();
65     private static final JTextField txtNodes = new JTextField("10",3);
```

```
66      private static final JTextField txtNodes2 = new JTextField("10",3);
67      private static final JTextField txtP = new JTextField("0.5",3);
68      private static final JTextField txtR = new JTextField("0.1",3);
69      private static final JTextField txtK = new JTextField("5",3);
70      private static final JTextField txtN = new JTextField("769",3); // mgeop
71      private static final JTextField txtM = new JTextField("4",3); // mgeop
72      private static final JTextField txtAlpha = new JTextField("0.17",3); //
        mgeop
73      private static final JTextField txtBeta = new JTextField("0.27",3); //
        mgeop
74      private static final JTextField txtP_MGEOP = new JTextField("1",3); //
        mgeop
75      private static final JTextField txtE_MGEOP = new JTextField("16656",8); //
         mgeop
76      public static int graphType = GNP_GRAPH;
77      private static JComponent graph;
78      private static BufferedImage image;
79      private static boolean blnLabel = true;
80      private static boolean blnShowVisual = true;
81      private static boolean blnNewWindow = true;
82      private static UndirectedSparseGraph<Node, String> g = null;
83      private static List<Node> dominatingSet = new ArrayList<>();
84
85      public createGUI() {
86          //constructor
87      }
88
89      public void createGraphWithDS() {
90          generateVisualGraph(true);
91          graphFrame.pack();
92      }
```

```
93
94      public static void createGraph() {
95          //int maxEdgeCount = (n*(n-1))>>1;
96          int nodes = 10;
97          double p = 0.5;
98          try {
99              if (graphType == GNR_GRAPH) {
100                 nodes = Integer.parseInt(txtNodes2.getText());
101                 p = Double.parseDouble(txtR.getText());
102             } else {
103                 nodes = Integer.parseInt(txtNodes.getText());
104                 p = Double.parseDouble(txtP.getText());
105             }
106
107         } catch (NumberFormatException e) {
108             // do nothing
109         }
110
111         RandomGraphGenerator rgg = new RandomGraphGenerator(nodes, p);
112         g = null;
113         if (graphType == GNP_GRAPH) {
114             g = rgg.generateGNPRandomGraph();
115         } else if (graphType == GNR_GRAPH) {
116             g = rgg.generateGNRRandomGraph();
117         }
118         System.out.println("done");
119
120     }
121
122     public static void createAndShowGUI() {
123         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
124         createGUI maingui = new createGUI();
125         maingui.createTabs();
126 
127         //Display the window.
128         frame.pack();
129         frame.setLocationRelativeTo(null); //center it
130         frame.setVisible(true);
131     }
132 
133     public static void generateVisualGraph(boolean showDS) {
134         Container contentPane = graphFrame.getContentPane();
135         if (graph != null) {
136             contentPane.remove(graph);
137         }
138 
139         Layout<Node, String> layout = null;
140         try {
141             layout = new CircleLayout<>(g);
142         } catch (Exception e) {
143             // do nothing
144             return;
145         }
146 
147         layout.setSize(new Dimension(800, 800));
148 
149         BasicVisualizationServer<Node, String> vv = new
    BasicVisualizationServer<>(layout);
150 
151         vv.setPreferredSize(new Dimension(800, 800));
152         ScalingControl scaler = new CrossoverScalingControl();
153         scaler.scale(vv, 1 / 1.1f, vv.getCenter());
```

```
154
155         // Transformer maps the vertex number to a vertex property
156         Transformer<Node,Paint> vertexColor;
157         vertexColor = (Node i) -> {
158             if (showDS) {
159                 if (dominatingSet.contains(i)) {
160                     return Color.WHITE;
161                 } else {
162                     return Color.GRAY;
163                 }
164             } else {
165                 return Color.RED;
166             }
167         };
168         vv.getRenderContext().setVertexFillPaintTransformer(vertexColor);
169         vv.getRenderContext().setVertexFillPaintTransformer(vertexColor);
170
171         if (blnLabel) {
172            vv.getRenderContext().setVertexLabelTransformer(new
    ToStringLabeller<>());
173            vv.getRenderer().getVertexLabelRenderer().setPosition(Position.N);
174         }
175
176         Box box = Box.createVerticalBox();
177         box.add(vv);
178         box.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10)); //Add
    some breathing room.
179
180         // Create the VisualizationImageServer
181         // vv is the VisualizationViewer containing my graph
182
```

```
183         VisualizationImageServer<Node, String> vis = new
    VisualizationImageServer<>(vv.getGraphLayout(), vv.getGraphLayout().
    getSize());
184
185         vis.setPreferredSize(new Dimension(800, 800));
186         ScalingControl scaler2 = new CrossoverScalingControl();
187         scaler2.scale(vis, 1 / 1.1f, vis.getCenter());
188         vis.setBackground(new java.awt.Color(255,255,255));
189         vis.getRenderContext().setVertexFillPaintTransformer(vertexColor);
190
191         if (blnLabel) {
192           vis.getRenderContext().setVertexLabelTransformer(new
    ToStringLabeller<>());
193           vis.getRenderer().getVertexLabelRenderer().setPosition(Position.N);
194         }
195
196         // Create the buffered image
197         image = (BufferedImage) vis.getImage(new Point2D.Double(vv.
    getGraphLayout().getSize().getWidth() / 2,
198         vv.getGraphLayout().getSize().getHeight() / 2),
199         new Dimension(vv.getGraphLayout().getSize()));
200
201         graphFrame.setVisible(true);
202         if ((blnNewWindow) & (!blnShowVisual)) {
203             graph = null;
204             box = Box.createVerticalBox();
205             box.setBorder(BorderFactory.createEmptyBorder(20, 700, 20, 20));
206             contentPane.add(box, BorderLayout.CENTER);
207         } else if ((!blnNewWindow) & (!blnShowVisual)) {
208             // nothing
209             graphFrame.setVisible(false);
```

```
210         } else {
211             graph = box;
212             contentPane.add(box, BorderLayout.CENTER);
213         }
214     }
215
216     // TABS
217     private void createTabs() {
218         panel1 = makeGnpPanel();
219         panel1.setPreferredSize(new Dimension(500,100));
220         tabbedPane.addTab("G(n,p)", null, panel1,"Generate G(n,p) graphs");
221         tabbedPane.setMnemonicAt(0, KeyEvent.VK_1);
222
223         panel2 = makeGnrPanel();
224         tabbedPane.addTab("G(n,r)", null, panel2, "Generate G(n,r) graphs");
225         tabbedPane.setMnemonicAt(1, KeyEvent.VK_2);
226
227         panel3 = makeMGEOPPanel();
228         tabbedPane.addTab("MGEO-P", null, panel3, "Generate MGEO-P graphs");
229         tabbedPane.setMnemonicAt(2, KeyEvent.VK_3);
230
231         panel4 = makeDominationPanel();
232         tabbedPane.addTab("Domination", null, panel4,"Run domination
    algorithms");
233         tabbedPane.setMnemonicAt(3, KeyEvent.VK_4);
234
235         panel5 = makeKCoresPanel();
236         tabbedPane.addTab("K-Cores", null, panel5,"Run k-core algorithms");
237         tabbedPane.setMnemonicAt(4, KeyEvent.VK_5);
238
239
```

```
240         panel6 = makeExportPanel();
241         tabbedPane.addTab("Export", null, panel6, "Export graph data");
242         tabbedPane.setMnemonicAt(5, KeyEvent.VK_6);
243 
244         panel7 = makeOptionsPanel();
245         tabbedPane.addTab("Options", null, panel7, "Options");
246         tabbedPane.setMnemonicAt(6, KeyEvent.VK_7);
247 
248         frame.add(tabbedPane);
249     }
250 
251     // G(n,p) Tab
252     protected JComponent makeGnpPanel() {
253         JPanel panel = new JPanel(false);
254         panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
255         JPanel line1 = new JPanel(false);
256         JPanel line2 = new JPanel(false);
257         JPanel line3 = new JPanel(false);
258         JLabel lblNodes = new JLabel("Number of nodes (n) : ");
259         JLabel lblP = new JLabel("Probability of edges (p): ");
260         JButton btnGenerate = new JButton("Generate New G(n, p) Graph");
261         btnGenerate.setActionCommand(CREATE_GRAPH);
262         btnGenerate.addActionListener(this);
263         line1.add(lblNodes);
264         line1.add(txtNodes);
265         line2.add(lblP);
266         line2.add(txtP);
267         line3.add(btnGenerate);
268         panel.add(line1);
269         panel.add(line2);
270         panel.add(line3);
```

```
271         txtNodes.addMouseWheelListener(this);
272         txtP.addMouseWheelListener(this);
273         return panel;
274     }
275
276     // G(n,r) Tab
277     protected JComponent makeGnrPanel() {
278         JPanel panel = new JPanel(false);
279         panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
280         JPanel line1 = new JPanel(false);
281         JPanel line2 = new JPanel(false);
282         JPanel line3 = new JPanel(false);
283         JLabel lblNodes = new JLabel("Number of nodes (n) : ");
284         JLabel lblR = new JLabel("Radius of influence (r): ");
285         JButton btnGenerate = new JButton("Generate New G(n, r) Graph");
286         btnGenerate.setActionCommand(CREATE_GRAPH2);
287         btnGenerate.addActionListener(this);
288         line1.add(lblNodes);
289         line1.add(txtNodes2);
290         line2.add(lblR);
291         line2.add(txtR);
292         line3.add(btnGenerate);
293         panel.add(line1);
294         panel.add(line2);
295         panel.add(line3);
296         txtNodes2.addMouseWheelListener(this);
297         txtR.addMouseWheelListener(this);
298         return panel;
299     }
300
301     // MGEO-P Tab
```

```
302     protected JComponent makeMGEOPPanel() {
303         JPanel panel = new JPanel(false);
304         panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
305         JPanel line1 = new JPanel(false);
306         JPanel line2 = new JPanel(false);
307         JPanel line3 = new JPanel(false);
308         JPanel line4 = new JPanel(false);
309         JPanel line5 = new JPanel(false);
310         JPanel line6 = new JPanel(false);
311         JPanel line7 = new JPanel(false);
312         JPanel line8 = new JPanel(false);
313         JLabel lblN = new JLabel("Number of vertices (n) : ");
314         JLabel lblM = new JLabel("Dimension of metric space (m) : ");
315         JLabel lblAlpha = new JLabel("Attachment strength parameter (alpha) :
    ");
316         JLabel lblBeta = new JLabel("Density parameter (beta) : ");
317         JLabel lblP = new JLabel("Connection probability (p) : ");
318         JLabel lblE = new JLabel("Set Edges (e) : ");
319         JButton btnMGEOP = new JButton("Generate New MGEO-P(n, m, alpha, beta,
     p) Graph");
320         JButton btnMGEOPINV = new JButton("INV-MGEO-P");
321
322         btnMGEOP.setActionCommand(MGEOP_GRAPH);
323         btnMGEOP.addActionListener(this);
324         btnMGEOPINV.setActionCommand("INV-MGEO-P");
325         btnMGEOPINV.addActionListener(this);
326
327         line1.add(lblN);
328         line1.add(txtN);
329         line2.add(lblM);
330         line2.add(txtM);
```

```
331         line3.add(lblAlpha);
332         line3.add(txtAlpha);
333         line4.add(lblBeta);
334         line4.add(txtBeta);
335         line5.add(lblP);
336         line5.add(txtP_MGEOP);
337         line6.add(lblE);
338         line6.add(txtE_MGEOP);
339         line7.add(btnMGEOP);
340         line8.add(btnMGEOPINV);
341         panel.add(line1);
342         panel.add(line2);
343         panel.add(line3);
344         panel.add(line4);
345         panel.add(line5);
346         panel.add(line6);
347         panel.add(line7);
348         panel.add(line8);
349         txtN.addMouseWheelListener(this);
350         txtM.addMouseWheelListener(this);
351         txtAlpha.addMouseWheelListener(this);
352         txtBeta.addMouseWheelListener(this);
353         txtP_MGEOP.addMouseWheelListener(this);
354         return panel;
355     }
356
357     // Domination Tab
358     protected JComponent makeDominationPanel() {
359         JPanel panel = new JPanel(false);
360         panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
```

```
361         final JButton btnDegreeDistribution = new JButton("Degree Distribution
    (Histogram)");
362         btnDegreeDistribution.setActionCommand(GRAPH_DEGREE_DIST);
363         btnDegreeDistribution.addActionListener(this);
364 
365         final JButton btnDiameter = new JButton("Diameter");
366         btnDiameter.setActionCommand(GRAPH_DIAMETER);
367         btnDiameter.addActionListener(this);
368 
369         final JButton btnDSRAI1 = new JButton("DS-RAI (PHASE 1)");
370         btnDSRAI1.setActionCommand(GRAPH_DSRAI_PHASE1);
371         btnDSRAI1.addActionListener(this);
372 
373         final JButton btnDSRAI2 = new JButton("DS-RAI (PHASE 2)");
374         btnDSRAI2.setActionCommand(GRAPH_DSRAI_PHASE1);
375         btnDSRAI2.addActionListener(this);
376 
377         final JButton btnDSRAI3 = new JButton("DS-RAI (PHASE 3)");
378         btnDSRAI3.setActionCommand(GRAPH_DSRAI_PHASE3);
379         btnDSRAI3.addActionListener(this);
380 
381         final JButton btnDSDC = new JButton("DS-DC");
382         btnDSDC.setActionCommand(GRAPH_DSDC);
383         btnDSDC.addActionListener(this);
384 
385         final JButton btnBurning = new JButton("Burning");
386         btnBurning.setActionCommand("BURNING");
387         btnBurning.addActionListener(this);
388 
389         panel.add(btnDegreeDistribution);
390         panel.add(btnDiameter);
```

```
391         panel.add(btnDSRAI1);
392         panel.add(btnDSRAI2);
393         panel.add(btnDSRAI3);
394         panel.add(btnDSDC);
395         panel.add(btnBurning);
396 
397         return panel;
398     }
399 
400     // KCores Tab
401     protected JComponent makeKCoresPanel() {
402         JPanel panel = new JPanel(false);
403         panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
404 
405         final JButton btnOneCore = new JButton("1-Core");
406         btnOneCore.setActionCommand(GRAPH_1CORE);
407         btnOneCore.addActionListener(this);
408 
409         final JButton btnTwoCore = new JButton("2-Core");
410         btnTwoCore.setActionCommand(GRAPH_2CORE);
411         btnTwoCore.addActionListener(this);
412 
413         final JButton btnThreeCore = new JButton("3-Core");
414         btnThreeCore.setActionCommand(GRAPH_3CORE);
415         btnThreeCore.addActionListener(this);
416 
417         final JButton btnKCore = new JButton("k-Core");
418         btnKCore.setActionCommand(GRAPH_KCORE);
419         btnKCore.addActionListener(this);
420 
421         JLabel lblK = new JLabel("K: ");
```

```
422
423             panel.add(btnOneCore);
424             panel.add(btnTwoCore);
425             panel.add(btnThreeCore);
426             panel.add(lblK);
427             panel.add(txtK);
428             panel.add(btnKCore);
429
430             return panel;
431         }
432
433         // Export Tab
434         protected JComponent makeExportPanel() {
435             JPanel panel = new JPanel(false);
436             panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
437             JButton btnSaveImage = new JButton("Save Image");
438             JButton btnExport = new JButton("Export Graph Data");
439
440             btnSaveImage.setActionCommand(SAVE_GRAPH_IMAGE);
441             btnSaveImage.addActionListener(this);
442
443             btnExport.setActionCommand(EXPORT_GRAPH_DATA);
444             btnExport.addActionListener(this);
445             panel.add(btnSaveImage);
446             panel.add(btnExport);
447
448             return panel;
449         }
450
451         // Options Tab
452         protected JComponent makeOptionsPanel() {
```

```
453         JPanel panel = new JPanel(false);
454         panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
455         final JCheckBox chkLabels = new JCheckBox("Labels", true);
456         chkLabels.addItemListener((ItemEvent e) -> {
457             blnLabel = (e.getStateChange() == 1);
458         });
459 
460         final JCheckBox chkShowVisual = new JCheckBox("Visual", true);
461         chkShowVisual.addItemListener((ItemEvent e) -> {
462             blnShowVisual = (e.getStateChange() == 1);
463             if (blnShowVisual == false) {
464                 Container contentPane = graphFrame.getContentPane();
465                 if (graph !=null) {
466                     contentPane.remove(graph);
467                 }
468                 if ((blnNewWindow) || (blnShowVisual)) {
469                     generateVisualGraph(false);
470                     graphFrame.pack();
471                     graphFrame.setVisible(true);
472                 } else {
473                     graphFrame.setVisible(false);
474                 }
475             } else {
476                 if (g.getVertexCount() >= 500) {
477                     Object[] options = {"Yes, please",
478                         "No, thanks"};
479                     int value = JOptionPane.showOptionDialog(frame, "There is
    over 500 nodes in your current graph.\n"
480                         + "Nodes: " + g.getVertexCount(), "Visual Graph",
481                         JOptionPane.YES_NO_OPTION,
482                         JOptionPane.QUESTION_MESSAGE,
```

```
483                         null,
484                         options,
485                         options[1]);
486                 System.out.println(value);
487                 if (value == 0) {
488                     if ((blnNewWindow) || (blnShowVisual)) {
489                         generateVisualGraph(false);
490                         graphFrame.pack();
491                         graphFrame.setVisible(true);
492                     } else {
493                         graphFrame.setVisible(false);
494                     }
495                 } else if (value == 1) {
496                     // do nothing
497                 }
498             } else {
499                 if ((blnNewWindow) || (blnShowVisual)) {
500                     generateVisualGraph(false);
501                     graphFrame.pack();
502                     graphFrame.setVisible(true);
503                     } else {
504                         graphFrame.setVisible(false);
505                     }
506             }
507         }
508     });
509
510     final JCheckBox chkNewWindow = new JCheckBox("New Window", true);
511     chkNewWindow.addItemListener((ItemEvent e) -> {
512         blnNewWindow = (e.getStateChange() == 1);
513     });
```

```
514         panel.add(chkLabels);
515         panel.add(chkShowVisual);
516         panel.add(chkNewWindow);
517         return panel;
518     }
519
520     protected static JComponent makeTextPanel(String text) {
521         JPanel panel = new JPanel(false);
522         JLabel filler = new JLabel(text);
523         filler.setHorizontalAlignment(JLabel.CENTER);
524         panel.setLayout(new GridLayout(1, 1));
525         panel.add(filler);
526         return panel;
527     }
528
529     protected JComponent createButtonPane() {
530         JPanel btmPanel = new JPanel();
531
532         JButton button = new JButton("Generate New G(n, p) Graph");
533         JButton button2 = new JButton("Generate New G(n, r) Graph");
534         JButton btnMGEOP = new JButton("MGEO-P");
535         JButton btnSaveImage = new JButton("Save Image");
536         JButton btnExport = new JButton("Export Graph Data");
537
538         JLabel lblNodes = new JLabel("n: ");
539         JLabel lblP = new JLabel("P/R: ");
540
541
542         button.setActionCommand(CREATE_GRAPH);
543         button.addActionListener(this);
544
```

```
545         button2.setActionCommand(CREATE_GRAPH2);
546         button2.addActionListener(this);
547
548         btnMGEOP.setActionCommand(MGEOP_GRAPH);
549         btnMGEOP.addActionListener(this);
550
551         btnSaveImage.setActionCommand(SAVE_GRAPH_IMAGE);
552         btnSaveImage.addActionListener(this);
553
554         btnExport.setActionCommand(EXPORT_GRAPH_DATA);
555         btnExport.addActionListener(this);
556
557         txtNodes.addMouseWheelListener(this);
558         txtP.addMouseWheelListener(this);
559
560         JPanel pane = new JPanel();
561         JPanel pane2 = new JPanel();
562         JPanel pane3 = new JPanel();
563         JPanel pane4 = new JPanel();
564         JPanel pane5 = new JPanel();
565
566         pane.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
567         pane.add(button);
568         pane.add(button2);
569         pane.add(btnMGEOP);
570
571         pane2.add(lblNodes);
572         pane2.add(txtNodes);
573         pane2.add(lblP);
574         pane2.add(txtP);
575
```

```
576         btmPanel.setLayout(new BoxLayout(btmPanel, BoxLayout.Y_AXIS));
577         btmPanel.add(pane);
578         btmPanel.add(pane2);
579         btmPanel.add(pane3);
580         btmPanel.add(pane4);
581         btmPanel.add(pane5);
582 
583         return btmPanel;
584     }
585 
586     @Override
587     public void actionPerformed(ActionEvent e) {
588         String command = e.getActionCommand();
589         int nodes, m;
590         double p, alpha, beta;
591         if (null != command) //Handle the New window button.
592         //Handle the New window button.
593         switch (command) {
594             case CREATE_GRAPH:
595                 graphType = GNP_GRAPH;
596                 createGraph();
597                   if (blnShowVisual) {
598                     generateVisualGraph(false);
599                     graphFrame.pack();
600                     graphFrame.setVisible(true);
601                 }
602                 graphFrame.setTitle("G(n="+txtNodes.getText()+",p="+txtP.
    getText()+") - " + g.getVertexCount() + " nodes - " + g.getEdgeCount() + "
     edges");
603 
604                 break;
```

```
605             case SAVE_GRAPH_IMAGE:
606                 saveGraphImage();
607                 break;
608             case CREATE_GRAPH2:
609                 graphType = GNR_GRAPH;
610                 createGraph();
611                 graphFrame.setTitle("G(n="+txtNodes.getText()+",r="+txtR.
    getText()+") - " + g.getVertexCount() + " nodes - " + g.getEdgeCount() + "
     edges");
612                 break;
613             case EXPORT_GRAPH_DATA:
614                 //exportGraphData();
615                 exportSMATData();
616                 break;
617             case GRAPH_DEGREE_DIST:
618                 GraphProperties.degreeDistribution(g);
619                 break;
620             case GRAPH_DIAMETER:
621                 System.out.println(GraphProperties.diameter(g));
622                 break;
623             case GRAPH_DSRAI_PHASE1:
624                 dominatingSet = DominatingSet.dsrai(g,1);
625                 System.out.println(dominatingSet.size());
626                 if (blnShowVisual) {
627                     createGraphWithDS();
628                 }
629                 break;
630             case GRAPH_DSRAI_PHASE2:
631                 // dominatingSet = DominatingSet.dsrai(g,2);
632                 break;
633             case GRAPH_DSRAI_PHASE3:
```

```
634                 // dominatingSet = DominatingSet.dsrai(g,3);
635                 break;
636             case GRAPH_DSDC:
637                 dominatingSet = DominatingSet.dsdc(g);
638                 System.out.println(dominatingSet.size());
639                 if (blnShowVisual) {
640                     createGraphWithDS();
641                 }
642                 break;
643             case "INV-MGEO-P":
644                 nodes = Integer.parseInt(txtN.getText());
645                 p = Double.parseDouble(txtP_MGEOP.getText());
646                 alpha = Double.parseDouble(txtAlpha.getText());
647                 beta = Double.parseDouble(txtBeta.getText());
648                 m = Integer.parseInt(txtM.getText());
649                 g = new MGEOP(nodes,m,alpha,beta,p).generateInvGraph();
650                 if (blnShowVisual) {
651                     generateVisualGraph(false);
652                     graphFrame.pack();
653                     graphFrame.setVisible(true);
654                 }
655                 graphFrame.setTitle("DMGEO-P(n="+nodes+",alpha="+alpha+",beta=
    "+beta+",m="+m+") - " + g.getVertexCount() + " nodes - " + g.getEdgeCount
    () + " edges");
656                 break;
657             case MGEOP_GRAPH:
658                 nodes = Integer.parseInt(txtN.getText());
659                 p = Double.parseDouble(txtP_MGEOP.getText());
660                 alpha = Double.parseDouble(txtAlpha.getText());
661                 beta = Double.parseDouble(txtBeta.getText());
662                 m = Integer.parseInt(txtM.getText());
```

```
663                 g = new MGEOP(nodes,m,alpha,beta,p).generateGraph();
664                 if (blnShowVisual) {
665                     generateVisualGraph(false);
666                     graphFrame.pack();
667                     graphFrame.setVisible(true);
668                 }
669                 graphFrame.setTitle("MGEO-P(n="+nodes+",alpha="+alpha+",beta="
    +beta+",m="+m+",p="+p+") - " + g.getVertexCount() + " nodes - " + g.
    getEdgeCount() + " edges");
670                 break;
671             case GRAPH_1CORE:
672                 g = GraphManipulation.oneCore(g);
673                 graphFrame.setTitle(graphFrame.getTitle()+" (changed: "+ g.
    getEdgeCount() +")");
674                 if (blnShowVisual) {
675                     generateVisualGraph(false);
676                 }
677                 graphFrame.pack();
678                 break;
679             case GRAPH_2CORE:
680                 g = GraphManipulation.twoCore(g);
681                 graphFrame.setTitle(graphFrame.getTitle()+" (changed: "+ g.
    getEdgeCount() +")");
682                 if (blnShowVisual) {
683                     generateVisualGraph(false);
684                 }
685                 graphFrame.pack();
686                 break;
687             case GRAPH_3CORE:
688                 g = GraphManipulation.threeCore(g);
```

```
689                 graphFrame.setTitle(graphFrame.getTitle()+" (changed: "+ g.
    getEdgeCount() +")");
690                 if (blnShowVisual) {
691                     generateVisualGraph(false);
692                 }
693                 graphFrame.pack();
694                 break;
695             case GRAPH_KCORE:
696                 g = GraphManipulation.kCore(g, Integer.parseInt(txtK.getText()
    ));
697                 graphFrame.setTitle(graphFrame.getTitle()+" (changed: "+ g.
    getEdgeCount() +")");
698                 if (blnShowVisual) {
699                     generateVisualGraph(false);
700                 }
701                 graphFrame.pack();
702                 break;
703             case "BURNING":
704                 dominatingSet = DominatingSet.burning(g);
705                 System.out.println((dominatingSet.size()+1)+" rounds");
706                 System.out.println(dominatingSet.toString());
707                 if (blnShowVisual) {
708                     createGraphWithDS();
709                 }
710                 break;
711         }
712     }
713
714     public void saveGraphImage() {
715         DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd-HHmmss");
716         Date date = new Date();
```

```
717         String imageDate = dateFormat.format(date);
718         if (graphType == GNR_GRAPH) {
719             imageDate += "-gnr";
720         } else if (graphType == GNP_GRAPH) {
721             imageDate += "-gnp";
722         }
723         File outputfile = new File(imageDate+"-graph.png");
724 
725         try {
726             ImageIO.write(image, "png", outputfile);
727         } catch (IOException e) {
728             // Exception handling
729         }
730     }
731 
732 
733     public static void exportSMATData() {
734          BufferedWriter writer = null;
735         try {
736             //create a temporary file
737             DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd-HHmmss");
738             Date date = new Date();
739             String fileDate = dateFormat.format(date);
740             if (graphType == GNR_GRAPH) {
741                 fileDate += "-gnr";
742             } else if (graphType == GNP_GRAPH) {
743                 fileDate += "-gnp";
744             }
745             File file = new File(fileDate+"-data.txt");
746             writer = new BufferedWriter(new FileWriter(file));
```

```
747             Comparator<Node> comp = (Node arg0, Node arg1) -> arg0.compareTo(
    arg1);
748             Node[] vtxs = new Node[g.getVertexCount()];
749             g.getVertices().toArray(vtxs);
750             Arrays.sort(vtxs,comp);
751             for (Node vtx : vtxs) {
752                 for (Node adj : vtx.getAdjacent()) {
753                     writer.write(adj.getID() + " " + vtx.getID());
754                     writer.newLine();
755                     adj.deleteEdge(vtx);
756                 }
757             }
758         } catch (IOException e) {
759             //e.printStackTrace();
760         } finally {
761             try {
762                 writer.close();
763             } catch (IOException e) {
764                 //e.printStackTrace();
765             }
766         }
767 
768     }
769 
770     public static void exportGraphData() {
771         BufferedWriter writer = null;
772         try {
773             //create a temporary file
774             DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd-HHmmss");
775             Date date = new Date();
776             String fileDate = dateFormat.format(date);
```

```
777             if (graphType == GNR_GRAPH) {
778                 fileDate += "-gnr";
779             } else if (graphType == GNP_GRAPH) {
780                 fileDate += "-gnp";
781             }
782             File file = new File(fileDate+"-data.txt");
783             writer = new BufferedWriter(new FileWriter(file));
784             Comparator<Node> comp = (Node arg0, Node arg1) -> arg0.compareTo(
    arg1);
785             Node[] vtxs = new Node[g.getVertexCount()];
786             g.getVertices().toArray(vtxs);
787             Arrays.sort(vtxs,comp);
788             for (Node vtx : vtxs) {
789                 writer.write(vtx.adjacentString());
790                 writer.newLine();
791             }
792         } catch (IOException e) {
793             //e.printStackTrace();
794         } finally {
795             try {
796                 writer.close();
797             } catch (IOException e) {
798                 //e.printStackTrace();
799             }
800         }
801     }
802
803     @Override
804     public void mouseWheelMoved(MouseWheelEvent e) {
805         JTextField source = (JTextField) e.getSource();
806         int notches = e.getWheelRotation();
```

```
807         DecimalFormat df = new DecimalFormat("#.###");
808         if (source.equals(txtNodes)) {
809             if (notches < 0) {
810                 txtNodes.setText(Integer.parseInt(txtNodes.getText()) + 1 + ""
    );
811             } else {
812                 txtNodes.setText(Integer.parseInt(txtNodes.getText()) - 1 + ""
    );
813             }
814         } else if (source.equals(txtP)) {
815             if (notches < 0) {
816                 txtP.setText(df.format(Double.parseDouble(txtP.getText()) +
    0.05));
817             } else {
818                 txtP.setText(df.format(Double.parseDouble(txtP.getText()) -
    0.05));
819             }
820         } else if (source.equals(txtR)) {
821             if (notches < 0) {
822                 txtR.setText(df.format(Double.parseDouble(txtR.getText()) +
    0.05));
823             } else {
824                 txtR.setText(df.format(Double.parseDouble(txtR.getText()) -
    0.05));
825             }
826         } else if (source.equals(txtNodes2)) {
827             if (notches < 0) {
828                 txtNodes2.setText(Integer.parseInt(txtNodes2.getText()) + 1 +
    "");
829             } else {
```

```
830                 txtNodes2.setText(Integer.parseInt(txtNodes2.getText()) - 1 +
    "");
831             }
832         } else if (source.equals(txtN)) {
833             if (notches < 0) {
834                 txtN.setText(Integer.parseInt(txtN.getText()) + 1 + "");
835             } else {
836                 txtN.setText(Integer.parseInt(txtN.getText()) - 1 + "");
837             }
838         } else if (source.equals(txtM)) {
839             if (notches < 0) {
840                 txtM.setText(Integer.parseInt(txtM.getText()) + 1 + "");
841             } else {
842                 txtM.setText(Integer.parseInt(txtM.getText()) - 1 + "");
843             }
844         } else if (source.equals(txtAlpha)) {
845             if (notches < 0) {
846                 txtAlpha.setText(df.format(Double.parseDouble(txtAlpha.getText
    ()) + 0.05));
847             } else {
848                 txtAlpha.setText(df.format(Double.parseDouble(txtAlpha.getText
    ()) - 0.05));
849             }
850         } else if (source.equals(txtBeta)) {
851             if (notches < 0) {
852                 txtBeta.setText(df.format(Double.parseDouble(txtBeta.getText()
    ) + 0.05));
853             } else {
854                 txtBeta.setText(df.format(Double.parseDouble(txtBeta.getText()
    ) - 0.05));
855             }
```

```
856         } else if (source.equals(txtP_MGEOP)) {
857             if (notches < 0) {
858                 txtP_MGEOP.setText(df.format(Double.parseDouble(txtP_MGEOP.
    getText()) + 0.05));
859             } else {
860                 txtP_MGEOP.setText(df.format(Double.parseDouble(txtP_MGEOP.
    getText()) - 0.05));
861             }
862         }
863     }
864
865     public static void main(String[] args) {
866         JApplet applet = new createGUI();
867         frame.add(applet);
868         applet.init();
869         applet.start();
870     }
871
872     public void init() {
873         javax.swing.SwingUtilities.invokeLater(() -> {
874             createAndShowGUI();
875         });
876     }
877
878 }
```

GraphProperties.java

```
1 /*
2  *  @author Marc Lozier
3  */
4
```

```
5  package graphpackage;
6
7  import edu.uci.ics.jung.graph.Graph;
8  import java.util.Arrays;
9  import java.util.Comparator;
10 import matlabcontrol.MatlabConnectionException;
11 import matlabcontrol.MatlabInvocationException;
12 import matlabcontrol.MatlabProxy;
13 import matlabcontrol.MatlabProxyFactory;
14 import matlabcontrol.MatlabProxyFactoryOptions;
15
16 /**
17  *
18  * @author Marc
19  */
20 public class GraphProperties {
21
22     public static double diameter(Graph g) {
23         double best = -1;
24         Node arg0, arg1;
25         Comparator<Node> comp = new Comparator<Node>() {
26             @Override
27             public int compare(final Node arg0, final Node arg1) {
28                 return arg0.compareTo(arg1);
29             }
30         };
31         Node[] vtxs = new Node[g.getVertexCount()];
32         g.getVertices().toArray(vtxs);
33         Arrays.sort(vtxs,comp);
34         for (Node s : vtxs) {
35             PathFinder finder = new PathFinder(g, s);
```

```
36              for (Node v : vtxs) {
37                  if (finder.isReachable(v) && finder.distanceTo(v) > best) {
38                      best = finder.distanceTo(v);
39                  }
40                  if (v.getAdjacent().isEmpty()) {
41                      best = Double.POSITIVE_INFINITY;
42                  }
43              }
44          }
45          return best;
46      }
47
48      public static void degreeDistribution(Graph g) {
49          int size = g.getVertexCount();
50          Node arg0, arg1;
51          Comparator<Node> comp = new Comparator<Node>() {
52              @Override
53              public int compare(final Node arg0, final Node arg1) {
54                  return arg0.compareTo(arg1);
55              }
56          };
57
58          Node[] vtxs = new Node[size];
59          g.getVertices().toArray(vtxs);
60          Arrays.sort(vtxs,comp);
61          String command = "hist([";
62          for (Node vtx : vtxs) {
63              command += vtx.getDegree()+",";
64          }
65          command = command.substring(0,command.length()-1);
66          command += "],"+ size+")";
```

```
67
68          //Create a proxy, which we will use to control MATLAB
69          try {
70              MatlabProxyFactoryOptions options = new MatlabProxyFactoryOptions.
    Builder().setUsePreviouslyControlledSession(true).build();
71              MatlabProxyFactory factory = new MatlabProxyFactory(options);
72              MatlabProxy proxy = factory.getProxy();
73              proxy.eval(command);
74              proxy.disconnect();
75          } catch ( MatlabInvocationException e) {
76          } catch ( MatlabConnectionException e) {
77
78          } finally {
79
80          }
81      }
82
83
84 }
```

GraphManipulation.java

```
1 /*
2  *  @author Marc Lozier
3  */
4
5 package graphpackage;
6
7 import cern.colt.Arrays;
8 import edu.uci.ics.jung.graph.Graph;
9 import edu.uci.ics.jung.graph.UndirectedSparseGraph;
10 import java.util.ArrayList;
```

```
11 import java.util.List;
12 import java.util.Random;
13
14 /**
15  *
16  * @author Marc
17  */
18 public class GraphManipulation {
19     public static UndirectedSparseGraph<Node, String> oneCore(
    UndirectedSparseGraph<Node,String> oldGraph) {
20         UndirectedSparseGraph<Node, String> newGraph = oldGraph;
21         Node[] nodes = DominatingSet.graphToArray(newGraph);
22         for(Node v : nodes) {
23             if (newGraph.degree(v) == 0) {
24                 newGraph.removeVertex(v);
25             }
26         }
27         return newGraph;
28     }
29
30     public static UndirectedSparseGraph<Node, String> twoCore(
    UndirectedSparseGraph oldGraph) {
31         UndirectedSparseGraph<Node, String> newGraph = oldGraph;
32         Node[] nodes;
33         List<Node> toRemove = new ArrayList<Node>();
34         boolean repeat = true;
35         while (repeat) {
36             toRemove.clear();
37             repeat = false;
38             nodes = DominatingSet.graphToArray(newGraph);
39             for(Node v : nodes) {
```

```
40                 if (Integer.parseInt(v.getDegree()) < 2) {
41                     for (Node x : v.getAdjacent()) {
42                      x.deleteEdge(v);
43                      toRemove.add(x);
44                      newGraph.removeEdge(newGraph.findEdge(x, v));
45                      repeat = true;
46                     }
47                     for (Node w : toRemove) {
48                         v.deleteEdge(w);
49                     }
50                     newGraph.removeVertex(v);
51                 }
52             }
53         }
54         return newGraph;
55     }
56 
57     public static UndirectedSparseGraph<Node, String> threeCore(
    UndirectedSparseGraph oldGraph) {
58         UndirectedSparseGraph<Node, String> newGraph = oldGraph;
59         Node[] nodes;
60         List<Node> toRemove = new ArrayList<Node>();
61         boolean repeat = true;
62         while (repeat) {
63             toRemove.clear();
64             repeat = false;
65             nodes = DominatingSet.graphToArray(newGraph);
66             for(Node v : nodes) {
67                 if (newGraph.degree(v) < 3) {
68                     for (Node x : v.getAdjacent()) {
69                         x.deleteEdge(v);
```

```
70                          toRemove.add(x);
71                          newGraph.removeEdge(newGraph.findEdge(x, v));
72                          repeat = true;
73                      }
74                      for (Node w : toRemove) {
75                          v.deleteEdge(w);
76                      }
77                      newGraph.removeVertex(v);
78                  }
79              }
80          }
81          return newGraph;
82      }
83
84      public static UndirectedSparseGraph<Node, String> kCore(
    UndirectedSparseGraph oldGraph, int k) {
85          UndirectedSparseGraph<Node, String> newGraph = oldGraph;
86          Node[] nodes;
87          List<Node> toRemove = new ArrayList<Node>();
88          boolean repeat = true;
89          while (repeat) {
90              repeat = false;
91              nodes = DominatingSet.graphToArray(newGraph);
92              for(Node v : nodes) {
93                  if (newGraph.degree(v) < k) {
94                      for (Node x : v.getAdjacent()) {
95                       x.deleteEdge(v);
96                          toRemove.add(x);
97                          newGraph.removeEdge(newGraph.findEdge(x, v));
98                          repeat = true;
99                      }
```

```
100                     for (Node w : toRemove) {
101                         v.deleteEdge(w);
102                     }
103                     newGraph.removeVertex(v);
104                 }
105             }
106         }
107         return newGraph;
108     }
109
110     public static UndirectedSparseGraph<Node, String> percolateEdges(
    UndirectedSparseGraph<Node,String> oldGraph, int edges) {
111         UndirectedSparseGraph<Node, String> newGraph = oldGraph;
112         Node[] nodes = DominatingSet.graphToArray(newGraph);
113         shuffleArray(nodes);
114         Random genRandom = new Random(System.nanoTime());
115
116         int toRemove = newGraph.getEdgeCount() - edges;
117         while (toRemove > 0) {
118             for(Node v : nodes) {
119                 int currentDegree = Integer.parseInt(v.getDegree());
120                 if (currentDegree == 0) continue;
121                 int amountDelete = genRandom.nextInt(currentDegree);
122                 if (amountDelete == 0) continue;
123                 List<Node> adjacent = v.getAdjacent();
124                 List<Node> deletedNodes = new ArrayList<Node>();
125                 while(amountDelete > 0) {
126                     for (Node w : adjacent) {
127                         int toDelete = genRandom.nextInt(2);
128                         if ((toDelete == 1) & (amountDelete > 0) ) {
129                             if (newGraph.findEdge(w, v) == null) continue;
```

```
130                             newGraph.removeEdge(newGraph.findEdge(w, v));
131                             amountDelete--;
132                             toRemove--;
133                             deletedNodes.add(w);
134                             if (toRemove == 0) {
135                                 if (deletedNodes.size() > 0) {
136                                     for (Node x : deletedNodes) {
137                                         v.deleteEdge(x);
138                                         x.deleteEdge(v);
139                                     }
140                                 }
141                                 return newGraph;
142                             }
143                         } else if (amountDelete == 0) {
144                             break;
145                         }
146                     }
147                 }
148 
149                 if (deletedNodes.size() > 0) {
150                     for (Node x : deletedNodes) {
151                         v.deleteEdge(x);
152                         x.deleteEdge(v);
153                     }
154                 }
155 
156                 if (toRemove == 0) {
157                     return newGraph;
158                 }
159             }
160         }
```

```
161
162         while (toRemove < 0) {
163             Node nodeOne = nodes[genRandom.nextInt(nodes.length)];
164             Node nodeTwo = nodes[genRandom.nextInt(nodes.length)];
165
166             if (nodeOne != nodeTwo) {
167                 if (newGraph.findEdge(nodeOne, nodeTwo) == null) {
168                     newGraph.addEdge(nodeOne.getID() + "-to-" + nodeTwo.getID()
    , nodeOne, nodeTwo);
169                     toRemove++;
170                     nodeOne.addEdge(nodeTwo);
171                     nodeTwo.addEdge(nodeOne);
172                 }
173                 }
174             }
175         return newGraph;
176     }
177
178     // Implementing FisherYates shuffle
179     static void shuffleArray(Node[] ar) {
180         Random rnd = new Random(System.nanoTime());
181         for (int i = ar.length - 1; i > 0; i--) {
182             int index = rnd.nextInt(i + 1);
183             // Simple swap
184             Node a = ar[index];
185             ar[index] = ar[i];
186             ar[i] = a;
187         }
188     }
189 }
```

FileReader.java

```
1  /*
2   *  @author Marc Lozier
3   */
4
5  package graphpackage;
6
7  import edu.uci.ics.jung.graph.Graph;
8  import edu.uci.ics.jung.graph.SparseMultigraph;
9  import edu.uci.ics.jung.graph.UndirectedSparseGraph;
10 import java.io.BufferedReader;
11 import java.io.BufferedWriter;
12 import java.io.File;
13 import java.io.FileInputStream;
14 import java.io.FileNotFoundException;
15 import java.io.FileWriter;
16 import java.io.IOException;
17 import java.io.InputStreamReader;
18 import java.util.ArrayList;
19 import java.util.Arrays;
20 import java.util.Comparator;
21 import java.util.List;
22 import java.util.logging.Level;
23 import java.util.logging.Logger;
24
25 /**
26  *
27  * @author Marc
28  */
29 public class FileReader {
30     private String fileName = "";
```

```
31      private BufferedReader reader = null;
32      private String tempLine = "";
33      private FileInputStream file = null;
34      private int k = 0;
35      private UndirectedSparseGraph<Node, String> graph = null;
36      private final List<Node> nodeList = new ArrayList<Node>();
37      private boolean mgeop = false;
38      private boolean percolate = false;
39
40      public FileReader(String fileName) {
41          this.fileName = fileName;
42          this.k = 0;
43      }
44
45      public FileReader(String fileName, int core) {
46          this.fileName = fileName;
47          this.k = core;
48      }
49
50      public void setMGEOP(boolean s) {
51          this.mgeop = s;
52      }
53
54      public void setPercolate(boolean p) {
55          this.percolate = p;
56      }
57
58      public void setCore(int core) {
59          this.k = core;
60      }
61
```

```
62      private void readFile() {
63          try {
64              if (mgeop) {
65                  if (percolate) {
66                      File tmp = new File(fileName.replace(".smat", "_p_"+k+"
    _compressed.smat"));
67                      if (tmp.exists() && !tmp.isDirectory()) { // if compressed
     file exist - use it
68                          fileName = fileName.replace(".smat", "_p_"+k+"
    _compressed.smat");
69                      } else {
70                          fileName = fileName.replace(".smat", "_0_compressed.
    smat");
71                      }
72                  } else {
73                      File tmp = new File(fileName.replace(".smat", "_" + k+"
    _compressed.smat"));
74                      if (tmp.exists() && !tmp.isDirectory()) { // if compressed
     file exist - use it
75                          fileName = fileName.replace(".smat", "_" + k+"
    _compressed.smat");
76                      } else {
77                          fileName = fileName.replace(".smat", "_0_compressed.
    smat");
78                      }
79                  }
80              } else {
81                  File tmp = new File(fileName.replace(".smat", "_" + k+"
    _compressed.smat"));
82                  if (tmp.exists() && !tmp.isDirectory()) { // if compressed
    file exist - use it
```

```
83                      fileName = fileName.replace(".smat", "_" + k+"_compressed.
    smat");
84                  }
85              }
86
87              file = new FileInputStream(fileName);
88              reader = new BufferedReader(new InputStreamReader(file));
89          } catch (FileNotFoundException ex) {
90              Logger.getLogger(FileReader.class.getName()).log(Level.SEVERE,
    null, ex);
91          }
92      }
93
94      private void closeFile() {
95          try {
96              reader.close();
97          } catch (IOException ex) {
98              Logger.getLogger(FileReader.class.getName()).log(Level.SEVERE,
    null, ex);
99          }
100     }
101
102     public void compressFileNoRead() {
103         try {
104             UndirectedSparseGraph<Node, String> g = new UndirectedSparseGraph<
    Node, String>();
105             file = new FileInputStream(fileName);
106             reader = new BufferedReader(new InputStreamReader(file));
107             try {
108                 nodeList.clear();
109                 String first = reader.readLine();
```

```
110                 String[] tmpSplit = first.split("\\s+");
111                 int intNodes = Integer.parseInt(tmpSplit[0]);
112                 Node tmp;
113                 for(int i = 0; i < intNodes; i++) {
114                     tmp = new Node(i);
115                     nodeList.add(tmp);
116                     g.addVertex(tmp);
117                 }
118                 while ((tempLine = reader.readLine()) != null) {
119                     tmpSplit = tempLine.split("\\s+");
120                     int toNode = Integer.parseInt(tmpSplit[0]);
121                     int fromNode = Integer.parseInt(tmpSplit[1]);
122                     //int numEdges = Integer.parseInt(tmpSplit[2]);
123                     Node a = nodeList.get(fromNode);
124                     Node b = nodeList.get(toNode);
125                     if ((!g.containsEdge(a.getID() + "-to-" + b.getID())) &&
    (!g.containsEdge(b.getID() + "-to-" + a.getID()))) {
126                         g.addEdge(a.getID() + "-to-" + b.getID(), a, b);
127                         a.addEdge(b);
128                         // NOTE: this line below is not needed (compresses
    file if omitted)
129                         b.addEdge(a);
130                     }
131                 }
132                 exportSMAT(g);
133             }   catch (IOException ex) {
134                 ex.printStackTrace();
135             } finally {
136                 closeFile();
137             }
138         }   catch (FileNotFoundException ex) {
```

```
139             Logger.getLogger(FileReader.class.getName()).log(Level.SEVERE,
    null, ex);
140         }
141     }
142
143
144     public void compressFile() {
145         UndirectedSparseGraph<Node, String> g = new UndirectedSparseGraph<Node
    , String>();
146         readFile();
147         try {
148             nodeList.clear();
149             String first = reader.readLine();
150             String[] tmpSplit = first.split("\\s+");
151             int intNodes = Integer.parseInt(tmpSplit[0]);
152             Node tmp;
153             for(int i = 0; i < intNodes; i++) {
154                 tmp = new Node(i);
155                 nodeList.add(tmp);
156                 g.addVertex(tmp);
157             }
158             while ((tempLine = reader.readLine()) != null) {
159                 tmpSplit = tempLine.split("\\s+");
160                 int toNode = Integer.parseInt(tmpSplit[0]);
161                 int fromNode = Integer.parseInt(tmpSplit[1]);
162                 //int numEdges = Integer.parseInt(tmpSplit[2]);
163                 Node a = nodeList.get(fromNode);
164                 Node b = nodeList.get(toNode);
165                 if ((!g.containsEdge(a.getID() + "-to-" + b.getID())) && (!g.
    containsEdge(b.getID() + "-to-" + a.getID()))) {
166                     g.addEdge(a.getID() + "-to-" + b.getID(), a, b);
```

```
167                     a.addEdge(b);
168                     // NOTE: this line below is not needed (compresses file if
    omitted)
169                     b.addEdge(a);
170                 }
171             }
172             exportSMAT(g);
173         }   catch (IOException ex) {
174            ex.printStackTrace();
175         } finally {
176             closeFile();
177         }
178     }
179
180
181        // EXPORT SMAT FILE TYPE (COMPRESSED)
182     public void exportSMAT(UndirectedSparseGraph g) {
183         try {
184             switch (k) {
185                 case 0:
186                     // nothing
187                     break;
188                 case 1:
189                     g = GraphManipulation.oneCore(g);
190                     break;
191                 case 2:
192                     g = GraphManipulation.twoCore(g);
193                     break;
194                 case 3:
195                     g = GraphManipulation.threeCore(g);
196                     break;
```

```
197                 default:
198                     g = GraphManipulation.kCore(g, k);
199                     break;
200             }
201             String prepend = g.getVertexCount() + " " + g.getVertexCount() + "
    " + g.getEdgeCount();
202             Node[] vtxs = new Node[g.getVertexCount()];
203             g.getVertices().toArray(vtxs);
204             Arrays.sort(vtxs, new Comparator() {
205                 @Override
206                 public int compare(Object o1, Object o2) {
207                     return ((Node) o1).compareTo((Node) o2);
208                 }
209             });
210             File filew;
211             if (percolate) {
212                 fileName = fileName.replace("_0_compressed.smat", "_" + k+"
    _compressed.smat");
213                 fileName = fileName.replace("_" + k+"_compressed.smat", ".smat
    ");
214                 filew = new File(fileName.replace(".smat", "_p_" + k+"
    _compressed.smat"));
215             } else {
216                 filew = new File(fileName.replace("_0_compressed.smat", "_"+k+
    "_compressed.smat"));
217             }
218 
219             BufferedWriter writer = new BufferedWriter(new FileWriter(filew));
220             writer.write(prepend);
221             writer.newLine();
222             for (Node vtx : vtxs) {
```

```
223                 for (Node adj : vtx.getAdjacent()) {
224                     writer.write(adj.getID() + " " + vtx.getID() + " 1");
225                     writer.newLine();
226                     adj.deleteEdge(vtx);
227                 }
228             }
229             writer.close();
230         } catch (IOException ex) {
231             ex.printStackTrace();
232         }
233     }
234 
235 
236     // EXPORT SMAT FILE TYPE (COMPRESSED)
237     public static void exportSMAT(UndirectedSparseGraph g, int k, boolean
    percolate, String name) {
238         try {
239             switch (k) {
240                 case 0:
241                     // nothing
242                     break;
243                 case 1:
244                     g = GraphManipulation.oneCore(g);
245                     break;
246                 case 2:
247                     g = GraphManipulation.twoCore(g);
248                     break;
249                 case 3:
250                     g = GraphManipulation.threeCore(g);
251                     break;
252                 default:
```

```
253                     g = GraphManipulation.kCore(g, k);
254                     break;
255             }
256             String prepend = g.getVertexCount() + " " + g.getVertexCount() + "
    " + g.getEdgeCount();
257             Node[] vtxs = new Node[g.getVertexCount()];
258             g.getVertices().toArray(vtxs);
259             Arrays.sort(vtxs, new Comparator() {
260                 @Override
261                 public int compare(Object o1, Object o2) {
262                     return ((Node) o1).compareTo((Node) o2);
263                 }
264             });
265             File filew;
266             String fileName = name;
267             if (percolate) {
268                 fileName = fileName.replace("_0_compressed.smat", "_" + k+"
    _compressed.smat");
269                 fileName = fileName.replace("_" + k+"_compressed.smat", ".smat
    ");
270                 filew = new File(fileName.replace(".smat", "_p_" + k+"
    _compressed.smat"));
271             } else {
272                 filew = new File(fileName.replace("_0_compressed.smat", "_"+k+
    "_compressed.smat"));
273             }
274
275             BufferedWriter writer = new BufferedWriter(new FileWriter(filew));
276             writer.write(prepend);
277             writer.newLine();
278             for (Node vtx : vtxs) {
```

```
279                 for (Node adj : vtx.getAdjacent()) {
280                     writer.write(adj.getID() + " " + vtx.getID() + " 1");
281                     writer.newLine();
282                     adj.deleteEdge(vtx);
283                 }
284             }
285             writer.close();
286         } catch (IOException ex) {
287             ex.printStackTrace();
288         }
289     }
290 
291     public UndirectedSparseGraph<Node, String> generateGraph() {
292         graph = new UndirectedSparseGraph<Node, String>();
293         readFile();
294         try {
295             tempLine = reader.readLine();
296             String[] tmpSplit = tempLine.split("\\s+");
297             int intNodes = Integer.parseInt(tmpSplit[0]);
298             createNodes(intNodes);
299             while ((tempLine = reader.readLine()) != null) {
300                 tmpSplit = tempLine.split("\\s+");
301                 int toNode = Integer.parseInt(tmpSplit[0]);
302                 int fromNode = Integer.parseInt(tmpSplit[1]);
303                 //int numEdges = Integer.parseInt(tmpSplit[2]);
304                 Node a = nodeList.get(fromNode);
305                 Node b = nodeList.get(toNode);
306                 if (!a.isAdjacent(b) && !b.isAdjacent(a)) {
307                     graph.addEdge(a.getID() + "-to-" + b.getID(), a, b);
308                     a.addEdge(b);
```

```
309                     // NOTE: this line below is not needed (compresses file if
    omitted)
310                     b.addEdge(a);
311                 }
312             }
313         } catch (IOException ex) {
314             Logger.getLogger(FileReader.class.getName()).log(Level.SEVERE,
    null, ex);
315         } finally {
316             closeFile();
317         }
318         return graph;
319     }
320
321     private void createNodes(int numOfNodes) {
322         Node tmp;
323         for(int i = 0; i < numOfNodes; i++) {
324             tmp = new Node(i);
325             nodeList.add(tmp);
326             graph.addVertex(tmp);
327         }
328     }
329 }
```

DominatingSet.java

```
1 /*
2  *  @author Marc Lozier
3  */
4
5 package graphpackage;
6
```

```
7  import edu.uci.ics.jung.graph.Graph;
8  import edu.uci.ics.jung.graph.UndirectedSparseGraph;
9  import java.io.BufferedReader;
10 import java.io.FileInputStream;
11 import java.io.IOException;
12 import java.io.InputStreamReader;
13 import java.util.ArrayList;
14 import java.util.Arrays;
15 import java.util.Comparator;
16 import java.util.HashSet;
17 import java.util.List;
18 import java.util.Set;
19 import java.util.logging.Level;
20 import java.util.logging.Logger;
21
22
23 /**
24  *
25  * @author Marc
26  */
27 public class DominatingSet {
28
29     private static boolean isDominating(UndirectedSparseGraph g, List<Node> ds
    ) {
30         List<Node> coverSet = new ArrayList<Node>();
31         for (Node v : ds) {
32             coverSet.addAll(v.getAdjacent());
33             coverSet.add(v);
34         };
35         Set<Node> uniqueCoverSet = new HashSet<Node>(coverSet);
36         return uniqueCoverSet.size() == g.getVertexCount();
```

```
37      }
38
39      public static Node[] graphToArray(UndirectedSparseGraph g) {
40          Node[] vtxs = new Node[g.getVertexCount()];
41          g.getVertices().toArray(vtxs);
42          return vtxs;
43      }
44      // Algorithm: RANDOM
45      public static List<Node> dsrandom(UndirectedSparseGraph g) {
46          List<Node> ds = new ArrayList<Node>();
47          Node[] vtx = graphToArray(g);
48          GraphManipulation.shuffleArray(vtx);
49          for (Node v : vtx) {
50              ds.add(v);
51              if (isDominating(g,ds)) {
52                  break;
53              }
54          }
55          return ds;
56      }
57
58       // Algorithm: DS-Rank Improved
59      public static List<Node> dsrankImproved(UndirectedSparseGraph g, String
    rankingFile) {
60          List<Node> ds = new ArrayList<Node>();
61          List<Node> coverSet = new ArrayList<Node>();
62          try {
63              BufferedReader br = new BufferedReader(new InputStreamReader(new
    FileInputStream(rankingFile)));
64              String line ;
65              Node[] vtxs = graphToArray(g);
```

```
66              while ((line = br.readLine()) != null) {
67                  for (Node v : vtxs) {
68                      if (String.valueOf(v.getID()).equalsIgnoreCase(line)) {
69                          if (!coverSet.contains(v)) {
70                              ds.add(v);
71                              coverSet.add(v);
72                              coverSet.addAll(v.getAdjacent());
73                              break; // break out of for loop - only 1 node with
    that ID
74                          } else {
75                              continue;
76                          }
77                      } else {
78                          // skip node
79                      }
80                  }
81                  if (isDominating(g,ds)) {
82                      break; // leave file, found DS!
83                  }
84              }
85              br.close();
86          } catch (IOException ex) {
87              Logger.getLogger(DominatingSet.class.getName()).log(Level.SEVERE,
    null, ex);
88          }
89          return ds;
90      }
91
92      // Algorithm: DS-Rank
93      public static List<Node> dsrank(UndirectedSparseGraph g, String
    rankingFile) {
```

```
94          List<Node> ds = new ArrayList<Node>();
95          try {
96              BufferedReader br = new BufferedReader(new InputStreamReader(new
    FileInputStream(rankingFile)));
97              String line ;
98              Node[] vtxs = graphToArray(g);
99              while ((line = br.readLine()) != null) {
100                 for (Node v : vtxs) {
101                     if (String.valueOf(v.getID()).equalsIgnoreCase(line)) {
102                         ds.add(v);
103                         break; // break out of for loop - only 1 node with
    that ID
104                     } else {
105                         // skip node
106                     }
107                 }
108                 if (isDominating(g,ds)) {
109                     break; // leave file, found DS!
110                 }
111             }
112             br.close();
113         } catch (IOException ex) {
114             Logger.getLogger(DominatingSet.class.getName()).log(Level.SEVERE,
    null, ex);
115         }
116         return ds;
117     }
118
119     // Algorithm: DS-DC
120     public static List<Node> dsdc(UndirectedSparseGraph g) {
121         List<Node> ds = new ArrayList<Node>(g.getVertices()); // DS = V
```

```
122         Node arg0, arg1;
123         Comparator<Node> comp = new Comparator<Node>() {
124             @Override
125             public int compare(final Node arg0, final Node arg1) {
126                 return arg0.compareDegreeToAsc(arg1);
127             }
128         };
129 
130         Node[] vtxs = graphToArray(g);
131         Arrays.sort(vtxs,comp);
132         for (Node v : vtxs) {
133             // v is the minimum node
134             ds.remove(v);
135             if (!isDominating(g, ds)) {
136                 ds.add(v);
137             }
138         }
139         return ds;
140     }
141 
142     // Algorithm: DS-RAI
143     public static List<Node> dsrai(UndirectedSparseGraph g, int phase) {
144         List<Node> ds;
145         List<Node> black = new ArrayList<Node>();
146         List<Node> gray = new ArrayList<Node>();
147         if (phase == 1) {
148             // phase 1
149             Node arg0, arg1;
150             Comparator<Node> comp = new Comparator<Node>() {
151             @Override
152             public int compare(final Node arg0, final Node arg1) {
```

```
153                 return arg0.compareDegreeTo(arg1);
154             }
155         };
156 
157             Node[] vtxs = graphToArray(g);
158             Arrays.sort(vtxs,comp);
159             for (Node v : vtxs) {
160                 if (!gray.contains(v)) {
161                     black.add(v);
162                     for (Node dominatee : v.getAdjacent()) {
163                         gray.add(dominatee);
164                     }
165                 }
166             }
167         }
168         ds = black;
169         return ds;
170     }
171 
172     public static List<Node> burning(UndirectedSparseGraph<Node, String>
    gCored) {
173         List<Node> burningSequence = new ArrayList<Node>();
174         List<Node> burnt = new ArrayList<Node>();
175 
176         Node arg0, arg1;
177             Comparator<Node> comp = new Comparator<Node>() {
178             @Override
179             public int compare(final Node arg0, final Node arg1) {
180                 return arg0.compareDegreeTo(arg1);
181             }
182         };
```

```
183
184             Node[] vtxs = graphToArray(gCored);
185             Arrays.sort(vtxs,comp);
186             // loop highest to lowest degree
187
188             // Extend burning set
189
190             for (int i = 0; i < vtxs.length; i++) {
191                 if (!burnt.contains(vtxs[i])) {
192                     List<Node> temp = new ArrayList<Node>();
193                     for (Node burned : burnt) {
194                         temp.addAll(burned.getAdjacent());
195                     }
196                     burnt.addAll(temp);
197
198                     Set<Node> uniqueCoverSet = new HashSet<Node>(burnt);
199                     burnt = new ArrayList<Node>(uniqueCoverSet);
200
201                     boolean flag = true;
202                     while (flag) {
203                         if (!burnt.contains(vtxs[i])) {
204                             burnt.add(vtxs[i]);
205                             burningSequence.add(vtxs[i]);
206                             flag = false;
207                         } else {
208                             // need next higher vertex
209                             i++;
210
211                         }
212                         if (i == vtxs.length) {
213                             break;
```

```
214                             }
215                         }
216                     }
217                     if (burnt.size() == vtxs.length) {
218                         break;
219                     }
220                 }
221                 return burningSequence;
222     }
223 }
```

MGEOP.java

```
1 /*
2  *  @author Marc Lozier
3  */
4
5 package graphpackage;
6
7 import edu.uci.ics.jung.graph.Graph;
8 import edu.uci.ics.jung.graph.SparseMultigraph;
9 import edu.uci.ics.jung.graph.UndirectedSparseGraph;
10 import java.io.BufferedWriter;
11 import java.io.File;
12 import java.io.FileWriter;
13 import java.io.IOException;
14 import java.text.DecimalFormat;
15 import java.util.ArrayList;
16 import java.util.Arrays;
17 import java.util.Collections;
18 import java.util.List;
19 import java.util.Random;
```

```
20 import java.util.logging.Level;
21 import java.util.logging.Logger;
22
23 /**
24  *
25  * @author Marc
26  */
27 public class MGEOP {
28     private int n;
29     private int m = 2;
30     private double alpha = 0.6;
31     private double beta = 0.2;
32     private double p = 0.5;
33     public static List<Node> nodeList = new ArrayList<Node>();
34     public boolean saveRank = false;
35     public String dataSet = "";
36     public int kcore = 0;
37     public static final int MAX_THREADS = 3;
38     public static int threads_counter = 0;
39     private static UndirectedSparseGraph<Node, String> gGlobal = new
       UndirectedSparseGraph<Node, String>();
40
41
42     public MGEOP(int n, int m, double alpha, double beta, double p) {
43         this.n = n;
44         this.m = m;
45         this.alpha = alpha;
46         this.beta = beta;
47         this.p = p;
48     }
49
```

```
50      // INV-MGEO-P
51      public UndirectedSparseGraph<Node, String> generateInvGraph() {
52          gGlobal = new UndirectedSparseGraph<Node, String>();
53          createNodes();
54          generateRank();
55          generateRadius();
56          createEdgesInv();
57          return gGlobal;
58      }
59
60      // MGEO-P
61      public UndirectedSparseGraph<Node, String> generateGraph() {
62          gGlobal = new UndirectedSparseGraph<Node, String>();
63          createNodes();
64          generateRank();
65          generateRadius();
66          createEdges();
67          return gGlobal;
68      }
69
70      public Graph<Node, String> generateGraphSpeedTest() {
71          System.out.println(" > Creating Nodes");
72          createNodes();
73          System.out.println(" > Generating Rank");
74          generateRank();
75          System.out.println(" > Generating Radius");
76          generateRadius();
77          System.out.println(" > Creating Edges");
78          createEdgesThread();
79          System.out.println(" > Complete");
80          return gGlobal;
```

```
81      }
82
83      // Make MGEOP with at most e edges
84      public Graph<Node, String> generateGraph(int e) {
85           boolean invalid = true;
86           Graph<Node, String> graph = new SparseMultigraph<Node,String>();
87           while(invalid) {
88               createNodes();
89               generateRank();
90               generateRadius();
91               createEdges();
92               if (graph.getEdgeCount() > e) {
93                    invalid = false;
94               }
95           }
96          return graph;
97      }
98
99      public static void returnNodeThread(Node a) {
100          nodeList.add(a);
101          gGlobal.addVertex(a);
102     }
103
104
105     private void createNodesThreaded() {
106          long start = System.nanoTime();
107          try {
108               nodeList = new ArrayList<Node>();
109
110               for(int i = 0; i < n; i++) {
111                    NodeThread nt = new NodeThread(i, m);
```

```
112                 Thread t = new Thread(nt);
113                 t.start();
114                 if (i == (n/4)) {
115                     System.out.println("25% done Nodes");
116                     t.join();
117                 } else if (i == (n/2)) {
118                     System.out.println("50% done Nodes");
119                     t.join();
120                 } else if (i == (3*(n/4))) {
121                     System.out.println("75% done Nodes");
122                     t.join();
123                 }
124 
125             }
126         } catch (InterruptedException ex) {
127             Logger.getLogger(MGEOP.class.getName()).log(Level.SEVERE, null, ex
    );
128         }
129         long dur = System.nanoTime() - start;
130         System.out.println(dur);
131     }
132 
133     private void createNodes() {
134         long start = System.nanoTime();
135         nodeList = new ArrayList<Node>(n);
136         Node tmp;
137 
138         for(int i = 0; i < n; i++) {
139             tmp = new Node(i, m);
140             for(int temp = 0; temp < m; temp++) {
141                 tmp.setCoordinates(Math.random(), temp);
```

```
142             }
143
144             nodeList.add(tmp);
145             gGlobal.addVertex(tmp);
146             if (i == (n/4)) {
147                 System.out.println("25% done Nodes");
148             } else if (i == (n/2)) {
149                 System.out.println("50% done Nodes");
150             } else if (i == (3*(n/4))) {
151                 System.out.println("75% done Nodes");
152             }
153
154         }
155         long dur = System.nanoTime() - start;
156     }
157
158     private void generateRank() {
159         Collections.shuffle(nodeList, new Random(System.nanoTime()));
160         File fileOut = new File("/home/mlozier/Work/FB100-DATA/mgeop-models/
    data-" + dataSet + "-" +kcore + "-rank.txt");
161         try {
162             BufferedWriter writer = new BufferedWriter(new FileWriter(fileOut)
    );
163             String dataOut = "";
164             for (Node nodeList1 : nodeList) {
165                 dataOut = String.valueOf(nodeList1.getID());
166                 writer.write(dataOut);
167                 writer.newLine();
168             }
169             writer.close();
170         } catch (IOException ex) {
```

```
171             //Logger.getLogger(MGEOP.class.getName()).log(Level.SEVERE, null,
    ex);
172         }
173     }
174
175     private void generateRadius() {
176         int rank = 1;
177         for (Node v: nodeList) {
178             v.setR(0.5*Math.pow(Math.pow(n, -1*beta)*Math.pow(rank, -1*alpha),
    1.0/m));
179             rank++;
180         }
181     }
182
183     private void createEdgesInv() {
184         for(int a = 0; a < nodeList.size(); a++) {
185             Node x = nodeList.get(a);
186             for (int b = a + 1; b < nodeList.size(); b++) {
187                 Node y = nodeList.get(b);
188                 if (x.inInfluence(y)) {
189                         p = 1 - (x.returnInfinityDistance(y)/x.getR());
190                         //System.out.println(p);
191                         if (Math.random() < p) {
192                             gGlobal.addEdge(x.getID() + "-to-" + y.getID(), x,
    y);
193                             x.addEdge(y);
194                             y.addEdge(x);
195                         }
196                     }
197             }
198         }
```

```
199     }
200
201     private void createEdgesTest() {
202         long start;
203         long end = 0, dur;
204         for(int a = 0; a < nodeList.size(); a++) {
205             Node x = nodeList.get(a);
206             start = System.nanoTime();
207             for (int b = a + 1; b < nodeList.size(); b++) {
208                 Node y = nodeList.get(b);
209
210                     if ((x.inInfluence(y)) && (Math.random() < p)) {
211                         x.addEdge(y);
212                         y.addEdge(x);
213                     }
214             }
215             dur = System.nanoTime() - start;
216             end = end + dur;
217         }
218         System.out.println("Total time: " + end);
219     }
220
221     private void createEdgesThread() {
222         int s = nodeList.size();
223         for(int a = 0; a < nodeList.size(); a++) {
224             EdgeThread et = new EdgeThread(a, p);
225             Thread t = new Thread(et);
226             t.start();
227             try {
228                 if (a == (s/4)) {
229           System.out.println("25% done Edges");
```

```
230                 } else if (a == (s/2)) {
231             System.out.println("50% done Edges");
232                 } else if (a == (3*(s/4))) {
233             System.out.println("75% done Edges");
234                 }
235                 if ((a % 3) == 0) {
236                     t.join();
237                 }
238             } catch (Exception e) {
239 
240             }
241         }
242     }
243 
244     private void createEdges() {
245         int s = nodeList.size();
246         for(int a = 0; a < s; a++) {
247             double start = System.nanoTime();
248             for (int b = a; b < s; b++) {
249                 if ((Math.random() <= p) && (a != b) && (!gGlobal.isNeighbor(
    nodeList.get(a), nodeList.get(b)))) {
250                     if (nodeList.get(a).inInfluence(nodeList.get(b))) {
251                         gGlobal.addEdge(nodeList.get(a).getID() + "-" +
    nodeList.get(b).getID(), nodeList.get(a), nodeList.get(b));
252                         nodeList.get(a).addEdge(nodeList.get(b));
253                         nodeList.get(b).addEdge(nodeList.get(a));
254                     }
255                 }
256             }
257             if (a == (s/4)) {
258                 System.out.println("25% done Edges");
```

```
259             } else if (a == (s/2)) {
260                 System.out.println("50% done Edges");
261             } else if (a == (3*(s/4))) {
262                 System.out.println("75% done Edges");
263             }
264             double nElapsed = System.nanoTime() - start; //In seconds
265             double nTotalTime = (1.0 / ((double) a / (double) s)) * nElapsed;
266             double nRemaining = nTotalTime - nElapsed;
267             System.out.println(((double) a / (double) s  * 100.0) + "%
    complete, Estimated "+nRemaining+" seconds remaining");
268         }
269     }
270
271 }
```

RandomGraphGenerator.java

```
1  package graphpackage;
2
3  import edu.uci.ics.jung.graph.Graph;
4  import edu.uci.ics.jung.graph.SparseMultigraph;
5  import edu.uci.ics.jung.graph.UndirectedSparseGraph;
6  import java.text.DecimalFormat;
7  import java.util.ArrayList;
8  import java.util.Arrays;
9  import java.util.Comparator;
10 import java.util.List;
11 import java.util.concurrent.TimeUnit;
12 import javafx.scene.shape.Box;
13 import matlabcontrol.*;
14
15 public class RandomGraphGenerator {
```

```
16      private int n;
17      private double p;
18
19      public static Box box;
20
21      public RandomGraphGenerator(int n, double p) {
22          this.n = n;
23          this.p = p;
24      }
25
26      public UndirectedSparseGraph<Node, String> generateGNPRandomGraph() {
27          UndirectedSparseGraph<Node, String> g = new UndirectedSparseGraph<Node
        , String>();
28          List<Node> nodeList = new ArrayList<Node>();
29          System.out.println("Creating G(n,p)=G(" + n+","+p+")");
30          for (int i = 0; i < n; i++) {
31              Node tmp = new Node(i);
32              g.addVertex(tmp);
33              nodeList.add(tmp);
34          }
35
36          for (int i = 0; i < n; i++) {
37              Node a = nodeList.get(i);
38              for (int j = i + 1; j < n; j++) {
39                  Node b = nodeList.get(j);
40                  if (Math.random() < p) {
41                      g.addEdge(a.getID() + "-to-" + b.getID(), a, b);
42                      a.addEdge(b);
43                      // NOTE: this line below is not needed (compresses file if
        omitted)
44                      b.addEdge(a);
```

```
45                  }
46              }
47
48
49              System.out.println(((double) i / (double) n * 100.0) + "%
    complete.");
50          }
51          return g;
52      }
53      private static String timeConversion(long totalSeconds) {
54
55      final int MINUTES_IN_AN_HOUR = 60;
56      final int SECONDS_IN_A_MINUTE = 60;
57
58      int seconds = (int) (totalSeconds % SECONDS_IN_A_MINUTE);
59      int totalMinutes = (int) (totalSeconds / SECONDS_IN_A_MINUTE);
60      int minutes = totalMinutes % MINUTES_IN_AN_HOUR;
61      int hours = totalMinutes / MINUTES_IN_AN_HOUR;
62
63      return hours + " hours " + minutes + " minutes " + seconds + " seconds";
64  }
65      // Given: n = number of vertices, r = radius of influence
66      public UndirectedSparseGraph<Node, String> generateGNRRandomGraph() {
67          UndirectedSparseGraph<Node, String> g = new UndirectedSparseGraph<Node
    , String>();
68          List<Node> nodeList = new ArrayList<Node>();
69          double r = p;
70
71          double[] xCoordinates = generateNdoubles(n);
72          double[] yCoordinates = generateNdoubles(n);
73
```

```
74          for(int x = 0; x < n; x++) {
75              Node tmp = new Node(x, xCoordinates[x], yCoordinates[x], r);
76              g.addVertex(tmp);
77              nodeList.add(tmp);
78          }
79
80          for(int x = 0; x < n; x++) {
81              Node a = nodeList.get(x);
82              for(int y = x + 1; y < n; y++) {
83                  Node b = nodeList.get(y);
84                  if (a.inInfluence(b)) {
85                      g.addEdge(a.getID() + "-to-" + b.getID(), a, b);
86                      a.addEdge(b);
87                      // NOTE: this line below is not needed (compresses file
    if omitted)
88                      b.addEdge(a);
89                  }
90              }
91          }
92
93          return g;
94      }
95
96      // Generate list of n number between 0 and 1
97      public static double[] generateNdoubles(int n) {
98          double[] list = new double[n];
99          DecimalFormat df = new DecimalFormat("#.###");
100         for(int i = 0; i  < n; i++) {
101             list[i] = Double.parseDouble(df.format(Math.random()));
102         }
103
```

```
104             return list;
105         }
106 
107 }
```

B.2. SNAP Code (C++)

graphgenerator.cpp

```
#include "stdafx.h"
#include <string>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <bd.h>
#include <Windows.h>
#include <algorithm>
#include <omp.h>
#include <cstdio>
#include <ctime>
#include "matlabFunctions.h"
#include <vector>
#include <sstream>
#include "loadFunctions.h"
#include "graphFunctions.h"

int threads = 18;
int dynamic_threads = 18;



void getMGEOPTheoreticals(long n, int m, double alpha, double beta, double p,
    TFltV& results) {
  results[0] = 1 + (1 / alpha); // 0-power law
  results[1] = (p / (1 - alpha))*TMath::Power(n, 1 - alpha - beta); // 1-
    average degree
  results[2] = TMath::Power(n, 1 / m); // 2-diameter
```

```
27    results[3] = TMath::Log(TMath::Log(n)); //3-average distance
28 }
29
30 void estimatePowerLaw(const PUNGraph &Graph) {
31   // x - degrees | y - number of vertices
32   TVec<TIntPr> degreeCount;
33   for (PUNGraph::TObj::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++)
      {
34     bool exist = false;
35     for (int tmp = 0; tmp < degreeCount.Len(); tmp++) {
36       if (degreeCount[tmp].Val1 == NI.GetDeg()) {
37         degreeCount[tmp].Val2++;
38         exist = true;
39       }
40     }
41     if (!exist) {
42       degreeCount.Add(TIntPr(NI.GetDeg(), 1));
43     }
44   }
45
46
47   int n = degreeCount.Len();
48   double b_max = 0;
49   double s_max = 0;
50   double e_max = n;
51   for (double incre = 0; incre < 0.5; incre = incre + 0.001) {
52     double sum = 0.0;
53     double sum2 = 0.0;
54     double sum3 = 0.0;
55     double sum4 = 0.0;
56
```

```
57      double s = incre*n;
58      double e = n;
59      for (int tmp = int(s); tmp < int(e); tmp++) {
60        int x = degreeCount[tmp].Val1;
61        int y = degreeCount[tmp].Val2;
62        sum += TMath::Log(x) * TMath::Log(y);
63        sum2 += TMath::Log(x);
64        sum3 += TMath::Log(y);
65        sum4 += TMath::Log(x)*TMath::Log(x);
66      }
67      int n2 = int(e) - int(s);
68      double b = ((n2 * sum) - (sum2*sum3))/((n2*sum4)-(sum2*sum2));
69      if (b_max > b) {
70        b_max = b;
71        s_max = s;
72      }
73    }
74
75    for (double incre = 0; incre < 0.5; incre = incre + 0.001) {
76      double sum = 0.0;
77      double sum2 = 0.0;
78      double sum3 = 0.0;
79      double sum4 = 0.0;
80
81      double s = s_max;
82      double e = (1 - incre)*n;
83      for (int tmp = int(s); tmp < int(e); tmp++) {
84        int x = degreeCount[tmp].Val1;
85        int y = degreeCount[tmp].Val2;
86        sum += TMath::Log(x) * TMath::Log(y);
87        sum2 += TMath::Log(x);
```

```
88          sum3 += TMath::Log(y);
89          sum4 += TMath::Log(x)*TMath::Log(x);
90        }
91        int n2 = int(e) - int(s);
92        double b = ((n2 * sum) - (sum2*sum3)) / ((n2*sum4) - (sum2*sum2));
93        if (b_max > b) {
94          b_max = b;
95          e_max = e;
96        }
97      }
98
99      printf("Estimmated Power Law Exponennt (Least Squares Fitting): %lf\n",
          b_max);
100     printf("In range: (%d, %d)...(%lf, %lf)\n", int(s_max), int(e_max), s_max,
          e_max);
101
102 }
103
104 double calAverageDistance(const PUNGraph& Graph, bool isConnected) {
105   double avgDistance = 0.0;
106   double sum = 0.0;
107   if (!isConnected) return avgDistance;
108   int n = Graph->GetNodes();
109   int counter = 1;
110   LARGE_INTEGER start;
111   TimerStart(&start);
112   omp_set_nested(1);
113   omp_set_num_threads(6);
114   omp_set_dynamic(6);
115 #pragma omp parallel for schedule(dynamic) shared(sum)
116   for (int x = 0; x < n; x++) {
```

```
117     for (int y = 0; y < n; y++) {
118       if (x <= y) continue;
119       sum += TSnap::GetShortPath(Graph, x, y);
120     }
121     Try{
122       double nPercentage = (double)counter / (double)n; //Ranges from 0-1
123       double nElapsed = TimerQuery(&start); //In seconds
124       double nTotalTime = (1.0 / nPercentage) * nElapsed;
125       double nRemaining = nTotalTime - nElapsed;
126       printf("%2.1f%% complete, Estimated %f seconds remaining\r", (
      nPercentage * 100.0), nRemaining);
127       counter = counter + 1;
128     }
129       Catch{}
130   }
131   printf("\r\n");
132   double na = (double)Graph->GetNodes();
133   avgDistance = (sum * 2.0)/((na*na) - na);
134   return avgDistance;
135 }
136
137 int GetMnDegNId(const PUNGraph& Graph) {
138   TIntV MnDegV;
139   int MnDeg = Graph->GetNodes();
140   for (PUNGraph::TObj::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++)
      {
141     if (MnDeg > NI.GetDeg()) { MnDegV.Clr(); MnDeg = NI.GetDeg(); }
142     if (MnDeg == NI.GetDeg()) { MnDegV.Add(NI.GetId()); }
143   }
144   EAssertR(!MnDegV.Empty(), "Input graph is empty!");
145   return MnDeg;
```

```
146 }
147
148 int GetMaxDegNId(const PUNGraph& Graph) {
149   TIntV MaxDegV;
150   int MaxDeg = -1;
151   for (PUNGraph::TObj::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++)
      {
152     if (MaxDeg < NI.GetDeg()) { MaxDegV.Clr(); MaxDeg = NI.GetDeg(); }
153     if (MaxDeg == NI.GetDeg()) { MaxDegV.Add(NI.GetId()); }
154   }
155   EAssertR(!MaxDegV.Empty(), "Input graph is empty!");
156   return MaxDeg;
157 }
158
159 void printGraphInfoObj(const PUNGraph &Graph) {
160   printf("nodes:%d  edges:%d\n", Graph->GetNodes(), Graph->GetEdges());
161   TInt maxDegree = GetMaxDegNId(Graph);
162   printf("Max Degree: %d\n", maxDegree);
163   TFlt avgDegree = 2.0*(double)Graph->GetEdges() / (double)Graph->GetNodes();
164   printf("Avg Degree: %f\n", avgDegree);
165   TInt minDegree = GetMnDegNId(Graph);
166   printf("Min Degree: %d\n", minDegree);
167
168   TFlt diameter = TSnap::GetBfsFullDiam(Graph, (int)(Graph->GetNodes()*0.10));
169   printf("Diameter: %f\n", diameter);
170   TFlt effDiameter = TSnap::GetBfsEffDiam(Graph, (int)(Graph->GetNodes()*0.10)
      );
171   printf("Eff. Diameter: %f\n", effDiameter);
172   std::string isConnected;
173   bool con;
174   if (minDegree == 0) {
```

```
175       isConnected = "Yes.";
176       con = false;
177     }
178     else {
179       isConnected = "No.";
180       con = true;
181     }
182     printf("Does 0-degree exist?: %s\n", isConnected.c_str());
183     TFlt avgDistance = calAverageDistance(Graph, con);
184     printf("avgDistance: %f\n", avgDistance);
185     TFlt clustingCf = TSnap::GetClustCf(Graph);
186     printf("Clustering Coeff: %f\n", clustingCf);
187 }
188
189 bool isDominating(const TIntV DS, const PUNGraph &Graph) {
190     TIntSet set;
191     TIntV covered;
192     int a = 0;
193
194     omp_set_nested(1);
195     omp_set_num_threads(20);
196     omp_set_dynamic(20);
197
198 #pragma omp parallel for schedule(dynamic) shared(a,DS,set)
199     for (a = 0; a < DS.Len(); a++) {
200 #pragma omp critical
201       covered.Add(DS[a]);
202
203       TUNGraph::TNodeI nbrs = Graph->GetNI(DS[a]);
204       for (int temp = 0; temp < nbrs.GetOutDeg(); temp++) {
205 #pragma omp critical
```

```
206         covered.Add(nbrs.GetOutNId(temp));
207       }
208     }
209
210     set.AddKeyV(covered);
211     if (set.Len() == Graph->GetNodes()) {
212       return true;
213     }
214     else {
215       return false;
216     }
217 }
218
219 TIntV dominatingsetDC(const PUNGraph &Graph) {
220     TIntV DS;
221     TIntPrV nodes;
222     LARGE_INTEGER start;
223     TimerStart(&start);
224     int counter = 1;
225
226     printf("--starting DC--");
227
228     for (TUNGraph::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++) {
229       nodes.Add(TIntPr(NI.GetOutDeg(), NI.GetId()));
230       DS.Add(NI.GetId());
231     }
232
233     printf("initalized.\n");
234
235     nodes.Sort(); // lowest to highest sort
236     DS.Sort();
```

```
237   printf("running...\n");
238   for (int c = 0; c < nodes.Len(); c++) {
239     TIntV DSTemp = DS;
240     DSTemp.DelIfIn(nodes[c].Val2);
241     if (isDominating(DSTemp, Graph)) {
242       DS = DSTemp;
243     }
244     Try{
245       double nPercentage = (double)counter / (double)nodes.Len(); //Ranges
      from 0-1
246       double nElapsed = TimerQuery(&start); //In seconds
247       double nTotalTime = (1.0 / nPercentage) * nElapsed;
248       double nRemaining = nTotalTime - nElapsed;
249       printf("%2.1f%% complete, Estimated %f seconds remaining\r", (
      nPercentage * 100.0), nRemaining);
250       counter = counter + 1;
251     }
252       Catch{}
253   }
254
255   printf("\nDS-DC Size: %d\n", DS.Len());
256   return DS;
257 }
258
259
260 TIntV dominatingsetRAI(const PUNGraph &Graph) {
261   // RAI algoirthm
262   TIntV coverSet, DS;
263   TIntPrV nodes;
264   TVec<TIntV> neighbours;
265   LARGE_INTEGER start;
```

```
266  TimerStart(&start);
267  int counter = 1;
268
269  for (TUNGraph::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++) {
270    nodes.Add(TIntPr(NI.GetOutDeg(), NI.GetId()));
271    TIntV nodeNbrs;
272    for (int deg = 0; deg < NI.GetOutDeg(); deg++) {
273      nodeNbrs.Add(NI.GetNbrNId(deg));
274    }
275    neighbours.Add(nodeNbrs);
276  }
277  nodes.Sort(false); // highest to lowest sorted1
278
279  for (int c = 0; c < nodes.Len(); c++) {
280    if (coverSet.Count(nodes[c].Val2) == 0) {
281
282      coverSet.Add(nodes[c].Val2);
283      DS.Add(nodes[c].Val2);
284
285      TUNGraph::TNodeI nbrs = Graph->GetNI(nodes[c].Val2);
286      for (int temp = 0; temp < nbrs.GetOutDeg(); temp++) {
287        coverSet.AddUnique(nbrs.GetOutNId(temp));
288      }
289    }
290
291    Try{
292      double nPercentage = (double)counter / (double)nodes.Len(); //Ranges
     from 0-1
293      double nElapsed = TimerQuery(&start); //In seconds
294      double nTotalTime = (1.0 / nPercentage) * nElapsed;
295      double nRemaining = nTotalTime - nElapsed;
```

```
296         printf("%2.1f%% complete, Estimated %f seconds remaining\r", (
      nPercentage * 100.0), nRemaining);
297         counter = counter + 1;
298       }
299       Catch{}
300     }
301     printf("\n%d,%d = DS-RAI Size: %d\n", Graph->GetNodes(), Graph->GetEdges(),
       DS.Len());
302     return DS;
303 }
304
305 void printStatsMenu() {
306     printf("Graph Statistics (Basic)\n");
307     printf("\t 1 -- Min/Avg/Max Degrees \n");
308     printf("\t 2 -- LogLog Plot with Bestfit\n");
309     printf("\t 3 -- \n");
310     printf("\t 4 -- \n");
311     printf("\t 5 -- \n");
312 }
313
314 void printMenu() {
315     TVec<TStr> options;
316     options.Add("gnp     - Generate G(n,p) Graph Sample");
317     options.Add("mgeop   - Generate MGEOP(n,m,a,b,p) Graph Sample");
318     options.Add("dmgeop - Generate Distance-MGEOP(n,m,a,b,p) Graph Sample");
319     options.Add("fb      - Facebook 100");
320     options.Add("q       - Quit");
321     printf("-------------------------------------\n");
322     for (int counter = 0; counter < options.Len(); counter++) {
323       printf("|%s\n", options[counter].CStr());
324     }
```

```
325   printf("-----------------------------------\n");
326   printf("> ");
327 }
328
329 int main(int argc, char* argv[]) {
330   PUNGraph GraphPt;
331   Env = TEnv(argc, argv, TNotify::StdNotify);
332   printf("Graph Generator By Marc Lozier\nbuild: %s, %s. Time: %s\n", __TIME__
        , __DATE__, TExeTm::GetCurTm());
333   TExeTm ExeTm;
334   __time64_t long_time;
335   _time64(&long_time);
336   Try{
337
338     if (Env.IsEndOfRun()) { return 0; }
339     TInt::Rnd.PutSeed(ExeTm.GetSecInt()); // initialize random seed
340     TFlt::Rnd.PutSeed(ExeTm.GetSecInt());
341
342     std::string menu_option;
343     char input[256];
344     omp_set_nested(1);
345     omp_set_num_threads(threads);
346     omp_set_dynamic(dynamic_threads);
347
348     printf("Starting MATLAB Engine...");
349     startMatlab();
350     printf("[OK]\n");
351     printMenu();
352
353     while (menu_option != "q") {
354
```

```
355       std::getline(std::cin, menu_option);
356
357       if (menu_option == "gnp") {
358         printf("0 - Generate sample (n,p)\n");
359         printf("1 - General Chart [+]\n");
360         printf("2 - Dominating Set\n");
361         printf("\n > ");
362
363         std::cin.getline(input, 256);
364         int input_num = atoi(input);
365
366         TStr filename;
367         long n_gnp;
368         double p_gnp;
369         switch (input_num) {
370           case 0:
371             printf("n = ");
372             std::cin.getline(input, 256);
373             n_gnp = atoi(input);
374             printf("p = ");
375             std::cin.getline(input, 256);
376             p_gnp = atof(input);
377
378             filename = TStr::Fmt("gnp_%ld_%4.2f.bin", n_gnp, p_gnp);
379             GraphPt = generateGNP(n_gnp, p_gnp, filename);
380             break;
381           case 1:
382             break;
383           case 2:
384             break;
385         }
```

```
386
387       }
388       else if (menu_option == "8" || menu_option == "mgeop") {
389         printf("1 - Generate from FB100 parameters\n");
390         printf("2 - General Chart [+]\n");
391         printf("3 - Dominating Set (DS-RAI)");
392         printf("\n > ");
393         std::cin.getline(input, 256);
394
395         int input_num = atoi(input);
396
397         printf("FB set (0-99) = ");
398         std::cin.getline(input, 256);
399         std::vector<std::string> range = split(input, '-');
400         int start = atoi(range[0].c_str());
401         int end = atoi(range[1].c_str());
402         printf("Iterations = ");
403         std::cin.getline(input, 256);
404         int iterations = atoi(input);
405
406         std::ifstream file("mgeop_parameters.csv");
407         std::string line_input;
408
409         if (input_num == 1) {
410           if (!file.good()) {
411             printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
412           }
413           else {
414             getline(file, line_input); // read header of columns
415             for (int i = 0; i < 100; i++) {
```

```
416             getline(file, line_input);
417             if (i < start) continue;
418             if (i > end) continue;
419             std::vector<std::string> data = split(line_input, ',');
420
421             long n_mgeop = atol(data[1].c_str());
422             long e_mgeop = atol(data[2].c_str());
423             int m_mgeop = atoi(data[5].c_str());
424             double a_mgeop = atof(data[3].c_str());
425             double b_mgeop = atof(data[4].c_str());
426             double p_mgeop = 1.00;
427             for (int current_iteration = 0; current_iteration < iterations;
    current_iteration++) {
428               TStr filename;
429               filename = TStr::Fmt("mgeop_%s.bin", data[0].c_str());
430               GraphPt = generateMGEOP(n_mgeop, m_mgeop, a_mgeop, b_mgeop,
    p_mgeop, filename);
431               // default: cluster coefficient, distance, diameter, power law
432             }
433           }
434         }
435         file.close();
436       }
437       else if (input_num == 2) {
438         std::ofstream fileout("mgeop_p_results.txt");
439         if (!file.good()) {
440           printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
441         }
442         else {
443           getline(file, line_input); // read header of columns
```

```
444             for (int i = 0; i < 100; i++) {
445               getline(file, line_input);
446               if (i < start) continue;
447               if (i > end) continue;
448               std::vector<std::string> data = split(line_input, ',');
449               TStr filename = TStr::Fmt("mgeop_%s_p_0_compressed.smat", data
    [0].c_str());
450
451               GraphPt = loadFile(filename);
452
453               TFlt e = (double)GraphPt->GetEdges();
454               TFlt v = (double)GraphPt->GetNodes();
455               TInt maxDegree = GetMaxDegNId(GraphPt);
456               int avgDegree = (int)((2.0*(double)GraphPt->GetEdges() / (double
    )GraphPt->GetNodes()) + 0.5);
457               TInt minDegree = GetMnDegNId(GraphPt);
458               TFlt density = (2 * e) / (v*(v - 1));
459               TStr output = TStr::Fmt("%s\t%d\t%d\t%d\t%d\t%d\t%.3f\n", data
    [0].c_str(), GraphPt->GetNodes(), GraphPt->GetEdges(), minDegree,
    avgDegree, maxDegree, density);
460               fileout << output.CStr();
461             }
462           }
463           fileout.close();
464         }
465         else if (input_num == 3) {
466           std::ofstream fileout_ds_rai0("0_rai_mgeop_domintaing_results.txt");
467           std::ofstream fileout_ds_rai1("1_rai_mgeop_domintaing_results.txt");
468           std::ofstream fileout_ds_rai2("2_rai_mgeop_domintaing_results.txt");
469           std::ofstream fileout_ds_rai3("3_rai_mgeop_domintaing_results.txt");
470           std::ofstream fileout_ds_rai4("4_rai_mgeop_domintaing_results.txt");
```

```
471         std::ofstream fileout_ds_rai5("5_rai_mgeop_domintaing_results.txt");
472
473         if (!file.good()) {
474           printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
475         }
476         else {
477           TStrV datasets;
478           getline(file, line_input); // read header of columns
479           for (int i = 0; i < 100; i++) {
480             getline(file, line_input);
481             if (i < start) continue;
482             if (i > end) continue;
483             std::vector<std::string> data = split(line_input, ',');
484             TStr filename = TStr::Fmt("mgeop_%s.bin", data[0].c_str());
485             datasets.Add(filename);
486           }
487
488           for (int z = 0; z < datasets.Len(); z++) {
489             TStr filename = datasets[z];
490             GraphPt = loadFile(filename);
491             for (int k = 0; k <= 5; k++)  {
492               TIntV del;
493               TIntPrV edges_del;
494               PUNGraph grp;
495               grp.New();
496
497               grp = TSnap::GetKCore(GraphPt, k);
498               while (GetMnDegNId(grp) < k) {
499
```

```
500                 for (TUNGraph::TNodeI NI = grp->BegNI(); NI < grp->EndNI();
    NI++) {
501                   if (NI.GetOutDeg() < k) {
502                     del.Add(NI.GetId());
503                     for (TUNGraph::TNodeI NIA = grp->BegNI(); NIA < grp->
    EndNI(); NIA++) {
504                       for (int u = 0; u < NIA.GetDeg(); u++) {
505                         if (NIA.GetNbrNId(u) == NI.GetId()) {
506                           edges_del.Add(TIntPr(NIA.GetNbrNId(u), NI.GetId())
    );
507                         }
508                       }
509                     }
510                   }
511                 }
512
513                 for (int del_len = 0; del_len < del.Len(); del_len++) {
514                   grp->DelNode(del[del_len]);
515                 }
516                 for (int del_len = 0; del_len < del.Len(); del_len++) {
517                   grp->DelEdge(edges_del[del_len].Val1, edges_del[del_len].
    Val2);
518                 }
519               }
520
521               grp->Defrag();
522               GraphPt = grp;
523               printf("%d,%d\n", grp->GetNodes(), grp->GetEdges());
524               TIntV ds_rai_k = dominatingsetRAI(grp);
525               TStr core = TStr::Fmt("%d&%d\\\\ \\hline\n", grp->GetNodes(),
    ds_rai_k.Len());
```

```
526                 switch (k) {
527                 case 0:
528                   fileout_ds_rai0 << core.CStr();
529                   break;
530                 case 1:
531                   fileout_ds_rai1 << core.CStr();
532                   break;
533                 case 2:
534                   fileout_ds_rai2 << core.CStr();
535                   break;
536                 case 3:
537                   fileout_ds_rai3 << core.CStr();
538                   break;
539                 case 4:
540                   fileout_ds_rai4 << core.CStr();
541                   break;
542                 case 5:
543                   fileout_ds_rai5 << core.CStr();
544                   break;
545                 }
546               }
547             }
548             fileout_ds_rai0.close(); fileout_ds_rai1.close(); fileout_ds_rai2.
    close(); fileout_ds_rai3.close(); fileout_ds_rai4.close(); fileout_ds_rai5
    .close();
549           }
550           file.close();
551         }
552       }
553       else if (menu_option == "9" || menu_option == "dmgeop") {
554         printf("0 - Generate sample\n");
```

```
555         printf("1 - Generate from FB100 parameters\n");
556         printf("2 - General Chart [+]\n");
557         printf("3 - Dominating Set (RAI)\n");
558         printf("4 - Power Law Graphs\n >");
559         std::cin.getline(input, 256);
560 
561         int input_num = atoi(input);
562 
563         std::string line_input;
564 
565         if (input_num == 0) {
566           printf("n = ");
567           std::cin.getline(input, 256);
568           int n_temp = atoi(input);
569           printf("m = ");
570           std::cin.getline(input, 256);
571           int m_temp = atoi(input);
572           printf("a = ");
573           std::cin.getline(input, 256);
574           double a_temp = atof(input);
575           printf("b = ");
576           std::cin.getline(input, 256);
577           double b_temp = atof(input);
578           printf("Iterations = ");
579           std::cin.getline(input, 256);
580           int iterations = atoi(input);
581           TStr filename = TStr::Fmt("dmgeop_%d_%d.bin", n_temp, m_temp);
582           GraphPt = generateMGEOPDistance(n_temp, m_temp, a_temp, b_temp,
    filename);
583         }
584         else if (input_num == 1) {
```

```
585
586           std::ifstream file("mgeop_parameters.csv");
587
588           printf("FB set (0-99) = ");
589           std::cin.getline(input, 256);
590           std::vector<std::string> range = split(input, '-');
591           int start = atoi(range[0].c_str());
592           int end = atoi(range[1].c_str());
593
594           printf("Iterations = ");
595           std::cin.getline(input, 256);
596           int iterations = atoi(input);
597
598           if (!file.good()) {
599             printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
600           }
601           else {
602             getline(file, line_input); // read header of columns
603             for (int i = 0; i < 100; i++) {
604               getline(file, line_input);
605               if (i < start) continue;
606               if (i > end) continue;
607               std::vector<std::string> data = split(line_input, ',');
608
609               long n_mgeop = atol(data[1].c_str());
610               long e_mgeop = atol(data[2].c_str());
611               int m_mgeop = atoi(data[5].c_str());
612               double a_mgeop = atof(data[3].c_str());
613               double b_mgeop = atof(data[4].c_str());
614               double p_mgeop = 1.00;
```

```
615
616             for (int current_iteration = 0; current_iteration < iterations;
    current_iteration++) {
617               TStr filename;
618               filename = TStr::Fmt("dmgeop_%s_%d.bin", data[0].c_str(),
    current_iteration);
619               GraphPt = generateMGEOPDistance(n_mgeop, m_mgeop, a_mgeop,
    b_mgeop, filename);
620             }
621           }
622         }
623         file.close();
624       }
625       else if (input_num == 2) {
626         printf("FB set (0-99) = ");
627         std::cin.getline(input, 256);
628         std::vector<std::string> range = split(input, '-');
629         int start = atoi(range[0].c_str());
630         int end = atoi(range[1].c_str());
631
632         std::ifstream file("mgeop_parameters.csv");
633         std::ofstream fileout("results.txt");
634         if (!file.good()) {
635           printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
636         }
637         else {
638           getline(file, line_input); // read header of columns
639           for (int i = 0; i < 100; i++) {
640             getline(file, line_input);
641             if (i < start) continue;
```

```
642             if (i > end) continue;
643             std::vector<std::string> data = split(line_input, ',');
644             TStr filename = TStr::Fmt("dmgeop_%s_0.bin", data[0].c_str());
645             GraphPt = loadFile(filename);
646
647             TFlt e = (double)GraphPt->GetEdges();
648             TFlt v = (double)GraphPt->GetNodes();
649             TInt maxDegree = GetMaxDegNId(GraphPt);
650             int avgDegree = (int)((2.0*(double)GraphPt->GetEdges() / (double
    )GraphPt->GetNodes()) + 0.5);
651             TInt minDegree = GetMnDegNId(GraphPt);
652             TFlt density = (2 * e) / (v*(v - 1));
653             TStr output = TStr::Fmt("%s\t%d\t%d\t%d\t%d\t%d\t%.3f\n", data
    [0].c_str(), GraphPt->GetNodes(), GraphPt->GetEdges(), minDegree,
    avgDegree, maxDegree, density);
654             fileout << output.CStr();
655           }
656         }
657         fileout.close();
658       }
659       else if (input_num == 3) {
660         // Dominating Set
661         std::ofstream fileout_ds_dc0("0_dc_dmgeop_domintaing_results.txt");
662         std::ofstream fileout_ds_dc1("1_dc_dmgeop_domintaing_results.txt");
663         std::ofstream fileout_ds_dc2("2_dc_dmgeop_domintaing_results.txt");
664         std::ofstream fileout_ds_dc3("3_dc_dmgeop_domintaing_results.txt");
665         std::ofstream fileout_ds_dc4("4_dc_dmgeop_domintaing_results.txt");
666         std::ofstream fileout_ds_dc5("5_dc_dmgeop_domintaing_results.txt");
667         std::ofstream fileout_ds_rai0("0_rai_dmgeop_domintaing_results.txt")
    ;
```

```
668         std::ofstream fileout_ds_rai1("1_rai_dmgeop_domintaing_results.txt")
    ;
669         std::ofstream fileout_ds_rai2("2_rai_dmgeop_domintaing_results.txt")
    ;
670         std::ofstream fileout_ds_rai3("3_rai_dmgeop_domintaing_results.txt")
    ;
671         std::ofstream fileout_ds_rai4("4_rai_dmgeop_domintaing_results.txt")
    ;
672         std::ofstream fileout_ds_rai5("5_rai_dmgeop_domintaing_results.txt")
    ;
673         printf("FB set (0-99) = ");
674         std::cin.getline(input, 256);
675         std::vector<std::string> range = split(input, '-');
676         int start = atoi(range[0].c_str());
677         int end = atoi(range[1].c_str());
678
679         std::ifstream file("mgeop_parameters.csv");
680
681         if (!file.good()) {
682           printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
683         }
684         else {
685           TStrV datasets;
686           getline(file, line_input); // read header of columns
687           for (int i = 0; i < 100; i++) {
688             getline(file, line_input);
689             if (i < start) continue;
690             if (i > end) continue;
691             std::vector<std::string> data = split(line_input, ',');
692             TStr filename = TStr::Fmt("dmgeop_%s_0.bin", data[0].c_str());
```

```
693             datasets.Add(filename);
694           }
695
696           for (int z = 0; z < datasets.Len(); z++) {
697             TStr filename = datasets[z];
698             GraphPt = loadFile(filename);
699             for (int k = 0; k <= 5; k++)  {
700               TIntV del;
701               TIntPrV edges_del;
702               PUNGraph grp;
703               grp.New();
704
705               grp = TSnap::GetKCore(GraphPt, k);
706               while (GetMnDegNId(grp) < k) {
707
708                 for (TUNGraph::TNodeI NI = grp->BegNI(); NI < grp->EndNI();
    NI++) {
709                   if (NI.GetOutDeg() < k) {
710                     del.Add(NI.GetId());
711                     for (TUNGraph::TNodeI NIA = grp->BegNI(); NIA < grp->
    EndNI(); NIA++) {
712                       for (int u = 0; u < NIA.GetDeg(); u++) {
713                         if (NIA.GetNbrNId(u) == NI.GetId()) {
714                           edges_del.Add(TIntPr(NIA.GetNbrNId(u), NI.GetId())
    );
715                         }
716                       }
717                     }
718                   }
719                 }
720
```

```
721                   for (int del_len = 0; del_len < del.Len(); del_len++) {
722                     grp->DelNode(del[del_len]);
723                   }
724                   for (int del_len = 0; del_len < del.Len(); del_len++) {
725                     grp->DelEdge(edges_del[del_len].Val1, edges_del[del_len].
    Val2);
726                   }
727                 }
728
729                 grp->Defrag();
730                 GraphPt = grp;
731                 printf("%d,%d\n", grp->GetNodes(), grp->GetEdges());
732                 //TStr core;
733
734                 if (input_num == 3) {
735                   TIntV ds_rai_k = dominatingsetRAI(grp);
736                   TStr core = TStr::Fmt("%d&%d\\\\ \\hline\n", grp->GetNodes()
    , ds_rai_k.Len());
737                   switch (k) {
738                   case 0:
739                     fileout_ds_rai0 << core.CStr();
740                     break;
741                   case 1:
742                     fileout_ds_rai1 << core.CStr();
743                     break;
744                   case 2:
745                     fileout_ds_rai2 << core.CStr();
746                     break;
747                   case 3:
748                     fileout_ds_rai3 << core.CStr();
749                     break;
```

```
750                     case 4:
751                       fileout_ds_rai4 << core.CStr();
752                       break;
753                     case 5:
754                       fileout_ds_rai5 << core.CStr();
755                       break;
756                     }
757                   }
758                   else if (input_num == 4) {
759                     TIntV ds_dc_k = dominatingsetDC(grp);
760                     TStr core = TStr::Fmt("%d&%d\\\\ \\hline\n", grp->GetNodes()
    , ds_dc_k.Len());
761                     switch (k) {
762                     case 0:
763                       fileout_ds_dc0 << core.CStr();
764                       break;
765                     case 1:
766                       fileout_ds_dc1 << core.CStr();
767                       break;
768                     case 2:
769                       fileout_ds_dc2 << core.CStr();
770                       break;
771                     case 3:
772                       fileout_ds_dc3 << core.CStr();
773                       break;
774                     case 4:
775                       fileout_ds_dc4 << core.CStr();
776                       break;
777                     case 5:
778                       fileout_ds_dc5 << core.CStr();
779                       break;
```

```
780                     }
781                   }
782
783
784                 }
785               }
786             }
787             fileout_ds_rai0.close(); fileout_ds_rai1.close(); fileout_ds_rai2.
    close(); fileout_ds_rai3.close(); fileout_ds_rai4.close(); fileout_ds_rai5
    .close();
788             fileout_ds_dc0.close(); fileout_ds_dc1.close(); fileout_ds_dc2.close
    (); fileout_ds_dc3.close(); fileout_ds_dc4.close(); fileout_ds_dc5.close()
    ;
789         }
790         else if (input_num == 4) {
791           // power Law
792           printf("0 - Facebook dataset equivalent\n");
793           printf("1 - Filename\n");
794           printf("> ");
795           std::cin.getline(input, 256);
796           int input_num = atoi(input);
797
798           switch (input_num) {
799             case 0:
800               break;
801             case 1:
802               printf("Filename = ");
803               std::cin.getline(input, 256);
804               GraphPt = loadFile(TStr(input));
805               getHistogram(GraphPt);
806               break;
```

```
807             }
808           }
809         }
810         else if (menu_option == "10" || menu_option == "stats") {
811           printStatsMenu();
812           std::cin.getline(input, 256);
813           if (atoi(input) == 2) {
814             getLogLogPlotBestfit(GraphPt);
815           }
816         }
817         else if (menu_option == "11" || menu_option == "rai") {
818
819         }
820         else if (menu_option == "12" || menu_option == "dc") {
821
822         }
823         else if (menu_option == "13" || menu_option == "fb") {
824           printf("1 - Generate from parameters\n");
825           printf("2 - Generete from file\n");
826           printf("3 - Dominating Set (RAI)\n");
827           printf("4 - Dominating Set (DC)\n");
828           printf("5 - Dominating Set (RAI & DC)\n");
829           printf("> ");
830           std::cin.getline(input, 256);
831
832           if (atoi(input) == 1) {
833             printf("FB set (0-99) = ");
834             std::cin.getline(input, 256);
835             std::vector<std::string> range = split(input, '-');
836             int start = atoi(range[0].c_str());
837             int end = atoi(range[1].c_str());
```

```
838         printf("number of iterations = ");
839         std::cin.getline(input, 256);
840         int iterations = atoi(input);
841 
842         printf("1-p=1\n2-varying p\n> ");
843         std::cin.getline(input, 256);
844         int p_type = atoi(input);
845         std::ifstream file("mgeop_parameters.csv");
846         std::string line_input;
847         if (!file.good()) {
848           printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
849         }
850         else {
851           getline(file, line_input); // read header of columns
852           for (int i = 0; i < 100; i++) {
853             getline(file, line_input);
854             if (i < start) continue;
855             if (i > end) continue;
856             std::vector<std::string> data = split(line_input, ',');
857 
858             long n_mgeop = atol(data[1].c_str());
859             long e_mgeop = atol(data[2].c_str());
860             int m_mgeop = atoi(data[5].c_str());
861             double a_mgeop = atof(data[3].c_str());
862             double b_mgeop = atof(data[4].c_str());
863             double p_mgeop = 1.00;
864             if (p_type == 2) {
865               p_mgeop = (((2.0*e_mgeop) / n_mgeop)*(1.0 - a_mgeop)*TMath::
    Power(n_mgeop, a_mgeop + b_mgeop))/n_mgeop;
866             }
```

```
867             for (int current_iteration = 0; current_iteration < iterations;
    current_iteration++) {
868               TStr filename;
869
870               if (p_type == 2) {
871                 filename = TStr::Fmt("p_%s_%d.bin", data[0].c_str(),
    current_iteration);
872               }
873               else {
874                 filename = TStr::Fmt("%s_%d.bin", data[0].c_str(),
    current_iteration);
875               }
876
877               GraphPt = generateMGEOP(n_mgeop, m_mgeop, a_mgeop, b_mgeop,
    p_mgeop, filename);
878             }
879           }
880         }
881         file.close();
882
883       }
884       else if (atoi(input) == 2) {
885         printf("FB set (0-99) = ");
886         std::cin.getline(input, 256);
887         std::vector<std::string> range = split(input, '-');
888         int start = atoi(range[0].c_str());
889         int end = atoi(range[1].c_str());
890         std::ifstream file("mgeop_parameters.csv");
891         std::string line_input;
892
893         std::ofstream fileout("results.txt");
```

```
894
895         if (!file.good()) {
896           printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
897         }
898         else {
899           printf("Dataset Name\tNodes\tEdges\Min\Avg\Max\Density\n");
900           getline(file, line_input); // read header of columns
901           for (int i = 0; i < 100; i++) {
902             getline(file, line_input);
903             if (i < start) continue;
904             if (i > end) continue;
905             std::vector<std::string> data = split(line_input, ',');
906             TStr filename = TStr::Fmt("%s.smat", data[0].c_str());
907             GraphPt = loadFile(filename);
908
909             TFlt e = (double)GraphPt->GetEdges();
910             TFlt v = (double)GraphPt->GetNodes();
911             TInt maxDegree = GetMaxDegNId(GraphPt);
912             TFlt avgDegree = 2.0*(double)GraphPt->GetEdges() / (double)
    GraphPt->GetNodes();
913             TInt minDegree = GetMnDegNId(GraphPt);
914             TFlt density = (2 * e) / (v*(v - 1));
915             TStr output = TStr::Fmt("%s\t%d\t%d\t%d\t%f\t%d\t%f\n", data[0].
    c_str(), GraphPt->GetNodes(), GraphPt->GetEdges(), minDegree, avgDegree,
    maxDegree, density);
916             fileout << output.CStr();
917           }
918         }
919         fileout.close();
920         file.close();
```

```
921         }
922         else if (atoi(input) == 5) {
923           printf("FB set (0-99) = ");
924           std::cin.getline(input, 256);
925           std::vector<std::string> range = split(input, '-');
926           int start = atoi(range[0].c_str());
927           int end = atoi(range[1].c_str());
928           std::ifstream file("mgeop_parameters.csv");
929           std::string line_input;
930
931           //std::ofstream fileout("results.txt");
932           std::ofstream fileout_ds_dc0("0_dc_domintaing_results.txt");
933           /*std::ofstream fileout_ds_dc1("1_dc_domintaing_results.txt");
934           std::ofstream fileout_ds_dc2("2_dc_domintaing_results.txt");
935           std::ofstream fileout_ds_dc3("3_dc_domintaing_results.txt");
936           std::ofstream fileout_ds_dc4("4_dc_domintaing_results.txt");
937           std::ofstream fileout_ds_dc5("5_dc_domintaing_results.txt");*/
938           //std::ofstream fileout_ds_rai0("0_rai_domintaing_results.txt");
939           /*std::ofstream fileout_ds_rai1("1_rai_domintaing_results.txt");
940           std::ofstream fileout_ds_rai2("2_rai_domintaing_results.txt");
941           std::ofstream fileout_ds_rai3("3_rai_domintaing_results.txt");
942           std::ofstream fileout_ds_rai4("4_rai_domintaing_results.txt");
943           std::ofstream fileout_ds_rai5("5_rai_domintaing_results.txt");*/
944
945           if (!file.good()) {
946             printf("Error opening FB100 parameter file (mgeop_parameters.csv)\
    n.");
947           }
948           else {
949             //printf("Dataset Name\tNodes\tEdges\Min\Avg\Max\Density\n");
950             getline(file, line_input); // read header of columns
```

```
951             for (int i = 0; i < 100; i++) {
952               getline(file, line_input);
953               if (i < start) continue;
954               if (i > end) continue;
955               std::vector<std::string> data = split(line_input, ',');
956               TStr filename = TStr::Fmt("p_%s_0.bin", data[0].c_str());
957               GraphPt = loadFile(filename);
958
959               TFlt e = (double)GraphPt->GetEdges();
960               TFlt v = (double)GraphPt->GetNodes();
961               TInt maxDegree = GetMaxDegNId(GraphPt);
962               int avgDegree = (int)((2.0*(double)GraphPt->GetEdges() / (double
    )GraphPt->GetNodes()) + 0.5);
963               TInt minDegree = GetMnDegNId(GraphPt);
964               TFlt density = (2 * e) / (v*(v - 1));
965               TStr output = TStr::Fmt("%s\t%d\t%d\t%d\t%d\t%d\t%.3f\n", data
    [0].c_str(), GraphPt->GetNodes(), GraphPt->GetEdges(), minDegree,
    avgDegree, maxDegree, density);
966               //fileout << output.CStr();
967
968               TIntV ds_dc = dominatingsetDC(GraphPt);
969               TStr core = TStr::Fmt("%d\t%d\n", GraphPt->GetNodes(), ds_dc.Len
    ());
970               fileout_ds_dc0 << core.CStr();
971
972
973
974             }
975           }
976           fileout_ds_dc0.close();
977           file.close();
```

```
978           }
979         }
980         else if (menu_option == "q") {
981           return 1;
982         }
983         else if (menu_option == "cpu_threads") {
984           printf("threads = ");
985           std::cin.getline(input, 256);
986           threads = atoi(input);
987           omp_set_num_threads(threads);
988         }
989         else if (menu_option == "cpu_dynamic") {
990           printf("threads_dynamic = ");
991           std::cin.getline(input, 256);
992           dynamic_threads = atoi(input);
993           omp_set_dynamic(dynamic_threads);
994         }
995         printf("> ");
996       } //end of menu_options
997       stopMatlab();
998     } Catch {
999       printf("\nrun time: %s (%s)\n", ExeTm.GetTmStr(), TSecTm::GetCurTm().
      GetTmStr().CStr());
1000    }
1001
1002    printf("\nPress enter to exit.\n");
1003    std::string nodepause;
1004    std::getline(std::cin, nodepause);
1005    return 0;
1006  }
```

loadFunctions.h

```
#include "stdafx.h"
#include <vector>

std::vector<std::string> &split(const std::string &, char, std::vector<std::
    string> &);
std::vector<std::string> split(const std::string &, char);

PUNGraph readTXT(TStr);
PUNGraph readSMAT(TStr);
PUNGraph loadFile(TStr);
```

loadFunctions.cpp

```
#include "stdafx.h"
#include <fstream>
#include <vector>
#include <stdio.h>
#include <sstream>
//Split string functions
std::vector<std::string> &split(const std::string &s, char delim, std::vector<
    std::string> &elems) {
  std::stringstream ss(s);
  std::string item;
  while (std::getline(ss, item, delim)) {
    elems.push_back(item);
  }
  return elems;
}
std::vector<std::string> split(const std::string &s, char delim) {
  std::vector<std::string> elems;
  split(s, delim, elems);
```

```
18   return elems;
19 }
20
21 PUNGraph readTXT(TStr filename) {
22   PUNGraph GraphPt = TUNGraph::New();
23   int nodes1, nodes2;
24   std::ifstream fin(filename.CStr());
25
26   if (!fin.good()) {
27     printf("Error file does not exist.\n");
28     return GraphPt;
29   }
30   while (!fin.eof())  {
31     fin >> nodes1 >> nodes2;
32     if (!GraphPt->IsNode(nodes1)) GraphPt->AddNode(nodes1);
33     if (!GraphPt->IsNode(nodes2)) GraphPt->AddNode(nodes2);
34     GraphPt->AddEdge(nodes2, nodes1);
35   }
36   fin.close();
37
38   GraphPt->Defrag();
39   return GraphPt;
40 }
41 PUNGraph readSMAT(TStr filename) {
42   PUNGraph GraphPt = TUNGraph::New();
43   int nodes1, nodes2, edgesA, edges;
44   std::ifstream fin(filename.CStr());
45
46   if (!fin.good()) {
47     printf("Error file does not exist.\n");
48     return GraphPt;
```

```
49    }
50
51    fin >> nodes1 >> nodes2 >> edgesA;
52
53    for (int i = 0; i < nodes1; i++) {
54      GraphPt->AddNode(i);
55    }
56    for (int i = 0; i < edgesA; i++) {
57      fin >> nodes1 >> nodes2 >> edges;
58      GraphPt->AddEdge(nodes2, nodes1);
59    }
60    fin.close();
61
62    GraphPt->Defrag();
63    return GraphPt;
64  }
65
66  PUNGraph loadFile(TStr file_name) {
67    PUNGraph GraphPt = TUNGraph::New();
68
69    printf("Loading file: %s...", file_name.CStr());
70
71    // load bin for faster loading
72    std::vector<std::string> names = split(std::string(file_name.CStr()).c_str()
        , '.');
73    std::ifstream test_bin(names[0] + ".bin");
74    if (test_bin.good()) {
75      test_bin.close();
76      TStr file_name = TStr::Fmt("%s.bin", names[0].c_str());
77      TFIn FIn(file_name);
78      GraphPt = TUNGraph::Load(FIn);
```

```
79      GraphPt->Defrag();
80      printf("Completed.\n");
81      return GraphPt;
82    }
83
84    // check extension
85    if (file_name.IsStrIn(TStr::Fmt(".smat"))) { // .smat file
86      GraphPt = readSMAT(file_name);
87    }
88    else if (file_name.IsStrIn(TStr::Fmt(".txt"))) {
89      GraphPt = TSnap::LoadEdgeList<PUNGraph>(file_name, 0, 1);
90    }
91    else {
92      printf("Sorry, file type (.%s) not recognized.\n", names[1].c_str());
93    }
94    GraphPt->Defrag();
95    printf("Completed.\n");
96
97    // create BIN file
98    printf("\nCreating BIN file for optimization...");
99    TStr filename = TStr::Fmt("%s.bin", names[0].c_str());
100   TFOut FOut(filename);
101   GraphPt->Save(FOut);
102   printf("Completed.\n");
103
104   return GraphPt;
105 }
```

graphFunctions.h

```
1 #include "stdafx.h"
2
```

```
3 void TimerStart(LARGE_INTEGER *);
4 double TimerQuery(LARGE_INTEGER *);
5 PUNGraph generateGNP(const long, const double, TStr);
6 PUNGraph generateMGEOP(const long, const long, const double, const double,
      const double, TStr);
7 PUNGraph generateMGEOPDistance(const long, const long, const double, const
      double, TStr);
8 void GetHypercubeCoordinates(const int &, TRnd &, TFltV &);
9 double getMaxElement(const TFltV &, const TFltV &, const TIntVV &, int, int);
10 double GetInfinityDistance(const TFltV &, const TFltV &, const int, const
      TIntVV &);
11 TIntVV generateCombinations(int, TIntV &);
```

graphFunctions.cpp

```
1 #include "stdafx.h"
2 #include <fstream>
3 #include <vector>
4 #include <stdio.h>
5 #include <sstream>
6 #include <omp.h>
7
8 LARGE_INTEGER liFreq;
9
10 //High Precision Event Timers.
11 void TimerStart(LARGE_INTEGER *pSpec) {
12   QueryPerformanceCounter(pSpec);
13   QueryPerformanceFrequency(&liFreq);
14 }
15 double TimerQuery(LARGE_INTEGER *pSpec) {
16   LARGE_INTEGER li;
17   QueryPerformanceCounter(&li);
```

```
18   return (li.QuadPart - pSpec->QuadPart) / (double)liFreq.QuadPart;
19 }
20
21 void GetHypercubeCoordinates(const int& Dim, TRnd& Rnd, TFltV& ValV) {
22   if (ValV.Len() != Dim) { ValV.Gen(Dim); }
23   for (int i = 0; i < Dim; i++) {
24     ValV[i] = Rnd.GetUniDev();
25   }
26 }
27
28 double getMaxElement(const TFltV& u, const TFltV& v, const TIntVV& sets, int
     ii, int m) {
29   TFltV results;
30   results.Gen(m);
31   for (int i = 0; i < m; i++) {
32     results[i] = abs(u[i] - v[i] - ((double)sets.At(ii, i)));
33   }
34   results.Sort();
35   return (double)results.Last();
36 }
37
38 double GetInfinityDistance(const TFltV& u, const TFltV& v, const int& dim,
     const TIntVV& sets) {
39   TFltV result;
40   int size = (int)pow(3, dim);
41   result.Gen(size);
42   for (int i = 0; i < size; i++) {
43     result[i] = getMaxElement(u, v, sets, i, dim);
44   }
45   result.Sort(false);
46   return (double)result.Last();
```

```
47 }
48
49 TIntVV generateCombinations(int arraySize, TIntV& possibleValues) {
50   int carry;
51   TIntV indices(arraySize);
52   int comb = (int)pow(3, arraySize);
53   TIntVV sets(comb, arraySize);
54   int count = 0;
55   int b = 0;
56
57   for (int q = 0; q < arraySize; q++) {
58     indices[q] = 0;
59   }
60
61   do {
62     for (int i = 0; i < arraySize; i++) {
63       sets.PutXY(count, b, possibleValues[indices[i]]);
64       b++;
65     }
66
67     b = 0;
68     count++;
69
70     carry = 1;
71     for (int i = arraySize - 1; i >= 0; i--) {
72       if (carry == 0)
73         break;
74
75       indices[i] += carry;
76       carry = 0;
77
```

```
78        if (indices[i] == 3)
79        {
80          carry = 1;
81          indices[i] = 0;
82        }
83      }
84    } while (carry != 1); // Call this method iteratively until a carry is left
      over;
85
86    return sets;
87  }
88
89  PUNGraph generateGNP(const long n, const double p, TStr filename) {
90    PUNGraph GraphPt = TUNGraph::New();
91    int counter = 1;
92    for (long i = 0; i < n; i++) {
93      GraphPt->AddNode(i);
94    }
95
96    LARGE_INTEGER start;
97    TimerStart(&start);
98  #pragma omp parallel for schedule(dynamic) shared(GraphPt)
99    for (long a = 0; a < n; a++) {
100     for (long b = 0; b < n; b++) {
101       if (b <= a) continue;
102       if (TFlt::Rnd.GetUniDev() <= p) {
103 #pragma omp critical
104         GraphPt->AddEdge(a, b);
105       }
106     }
107     Try{
```

```
108         double nPercentage = (double)counter / (double)n; //Ranges from 0-1
109         double nElapsed = TimerQuery(&start); //In seconds
110         double nTotalTime = (1.0 / nPercentage) * nElapsed;
111         double nRemaining = nTotalTime - nElapsed;
112         printf("%2.1f%% complete, Estimated %f seconds remaining\r", (
        nPercentage * 100.0), nRemaining);
113         counter = counter + 1;
114       }
115         Catch{}
116         // nothing
117     }
118     GraphPt->Defrag();
119     TFOut FOut(filename);
120     GraphPt->Save(FOut);
121     return GraphPt;
122 }
123
124 PUNGraph generateMGEOP(const long n, const long m, const double alpha, const
        double beta, const double p, TStr filename) {
125     PUNGraph GraphPt = TUNGraph::New();
126     TFltV ValRadius;
127     TFltV ValV;
128     printf("Creating graph:\nAllocating nodes...\n");
129     ValRadius.Gen(n);
130     TVec<TFltV> coord;
131
132     for (long i = 0; i < n; i++) {
133       GetHypercubeCoordinates(m, TFlt::Rnd, ValV);
134
135       /* For THESIS Sample
136       if (i == 0) {
```

```
137         ValV.Gen(2);
138         ValV[0] = 0.1894; ValV[1] = 0.1284;
139       }
140       else if (i == 1) {
141         ValV.Gen(2);
142         ValV[0] = 0.7439; ValV[1] = 0.7232;
143       }
144       else if (i == 2) {
145         ValV.Gen(2);
146         ValV[0] = 0.4200; ValV[1] = 0.9213;
147       }*/
148       coord.Add(ValV);
149       ValRadius[i] = 0.5*TMath::Power(TMath::Power((double)(i + 1), -1.0 * alpha
        )*TMath::Power((double)n, -1.0 * beta), 1.0 / m);
150       GraphPt->AddNode(i);
151     }
152
153     printf("Nodes generated.\n");
154     TIntV itms; itms.Add(-1); itms.Add(0); itms.Add(1);
155     TIntVV sets = generateCombinations(m, itms);
156
157     LARGE_INTEGER start;
158     TimerStart(&start);
159     // create edges
160     int counter = 1;
161 #pragma omp parallel shared(counter, n, coord, m, sets, ValRadius, start)
162     {
163       long x = 0;
164 #pragma omp for schedule(static, n/20)
165       for (x = 0; x < n; x++) {
166         for (long y = 0; y < n; y++) {
```

```
167           if (y <= x) continue;
168           if (TFlt::Rnd.GetUniDev() <= p) {
169             if (GetInfinityDistance(coord[x], coord[y], m, sets) <= ValRadius[x
      ]) {
170               #pragma omp critical
171               GraphPt->AddEdge(x, y);
172             }
173           }
174         }
175         Try{
176           double nPercentage = (double)counter / (double)n; //Ranges from 0-1
177           double nElapsed = TimerQuery(&start); //In seconds
178           double nTotalTime = (1.0 / nPercentage) * nElapsed;
179           double nRemaining = nTotalTime - nElapsed;
180           printf("%d Nodes left. Estimated %f seconds remaining\r", (int)(n-
      counter), nRemaining);
181           counter = counter + 1;
182         }
183           Catch{ //nothing
184         }
185       }
186     }
187
188
189     GraphPt->Defrag();
190     TFOut FOut(filename);
191     GraphPt->Save(FOut);
192     printf("Graph complete (%d, %d) - %s\n", GraphPt->GetNodes(), GraphPt->
        GetEdges(), filename);
193     return GraphPt;
194   }
```

```
195
196 PUNGraph generateMGEOPDistance(const long n, const long m, const double alpha,
        const double beta, TStr filename) {
197   PUNGraph GraphPt = TUNGraph::New();
198   TFltV ValRadius;
199   TFltV ValV;
200   TVec<TFltV> coord;
201
202   printf("Creating graph:\nAllocating nodes...\n");
203   ValRadius.Gen(n);
204
205   // Generate random m-coordinates, radius for all nodes (rank is nodes id,
       since arbitrary)
206   for (long i = 0; i < n; i++) {
207     GetHypercubeCoordinates(m, TFlt::Rnd, ValV);
208     coord.Add(ValV);
209     ValRadius[i] = 0.5*TMath::Power(TMath::Power((double)(i + 1), -1.0 * alpha
      )*TMath::Power((double)n, -1.0 * beta), 1.0 / m);
210     GraphPt->AddNode(i);
211   }
212
213   printf("Nodes generated.\n");
214   TIntV itms; itms.Add(-1); itms.Add(0); itms.Add(1);
215   TIntVV sets = generateCombinations(m, itms);
216
217   LARGE_INTEGER start;
218   TimerStart(&start);
219   // create edges
220   int counter = 1;
221
222 #pragma omp parallel shared(counter, n, coord, m, sets, ValRadius, start)
```

```
223   {
224     long x = 0;
225 #pragma omp for schedule(dynamic)
226     for (x = 0; x < n; x++) {
227       for (long y = 0; y < n; y++) {
228         if (y <= x) continue;
229         if (GetInfinityDistance(coord[x], coord[y], m, sets) <= ValRadius[x])
      {
230           if (TFlt::Rnd.GetUniDev() <= (1 - (GetInfinityDistance(coord[x],
      coord[y], m, sets) / ValRadius[x]))) {
231 #pragma omp critical
232             GraphPt->AddEdge(x, y);
233           }
234
235         }
236       }
237       Try{
238         double nPercentage = (double)counter / (double)n; //Ranges from 0-1
239         double nElapsed = TimerQuery(&start); //In seconds
240         double nTotalTime = (1.0 / nPercentage) * nElapsed;
241         double nRemaining = nTotalTime - nElapsed;
242         printf("%2.1f%% complete, Estimated %f seconds remaining\r", (
      nPercentage * 100.0), nRemaining);
243         counter = counter + 1;
244       }
245         Catch{ //nothing
246       }
247     }
248   }
249
250   printf("\nend edges\n");
```

```
251     GraphPt->Defrag();
252     TFOut FOut(filename);
253     GraphPt->Save(FOut);
254     return GraphPt;
255 
256 }
```

matlabFunctions.h

```
1 #include "stdafx.h"
2 
3 void startMatlab();
4 void getHistogram(const PUNGraph &);
5 void printMatlabOutput();
6 TFltPr bestPowerLawLocation(const PUNGraph &);
7 void getLogLogPlotBestfit(const PUNGraph &);
8 void getLogLogPlot(const PUNGraph &);
9 void stopMatlab();
```

matlabFunctions.cpp

```
1 #include "stdafx.h"
2 #include <fstream>
3 #include <vector>
4 #include <stdio.h>
5 #include <sstream>
6 #include "engine.h"
7 
8 // Global Variables
9 Engine *ep;
10 char *buff;
11 int nbuff = 4096;
12 
```

```
13 // Start MATLAB Engine
14 void startMatlab() {
15   if (!(ep = engOpen(NULL))) {
16     MessageBox((HWND)NULL, (LPSTR)"Can't start MATLAB engine",
17       (LPSTR) "Engwindemo.c", MB_OK);
18     exit(-1);
19   }
20   if ((buff = (char *)malloc(nbuff)) == NULL)
21     printf("Unable to allocate output buffer\n");
22   engOutputBuffer(ep, buff, nbuff);
23 }
24
25 // Histogram
26 void getHistogram(const PUNGraph &Graph) {
27   std::string evalaa("figure;histogram([");
28   for (PUNGraph::TObj::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++)
      {
29     evalaa += std::to_string(NI.GetDeg()) + " ";
30   }
31   evalaa += "])";
32   engEvalString(ep, evalaa.c_str());
33
34 }
35
36 void printMatlabOutput() {
37   if (buff != NULL)
38     printf(buff);
39 }
40
41 TFltPr bestPowerLawLocation(const PUNGraph &Graph) {
42   // x - degrees | y - number of vertices
```

```
43  TVec<TIntPr> degreeCount;
44  for (PUNGraph::TObj::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++)
     {
45    bool exist = false;
46    for (int tmp = 0; tmp < degreeCount.Len(); tmp++) {
47      if (degreeCount[tmp].Val1 == NI.GetDeg()) {
48        degreeCount[tmp].Val2++;
49        exist = true;
50      }
51    }
52    if (!exist) {
53      degreeCount.Add(TIntPr(NI.GetDeg(), 1));
54    }
55  }
56
57
58  int n = degreeCount.Len();
59  double b_max = 0;
60  double s_max = 0;
61  double e_max = n;
62  for (double incre = 0; incre < 0.5; incre = incre + 0.001) {
63    double sum = 0.0;
64    double sum2 = 0.0;
65    double sum3 = 0.0;
66    double sum4 = 0.0;
67
68    double s = incre*n;
69    double e = n;
70    for (int tmp = int(s); tmp < int(e); tmp++) {
71      int x = degreeCount[tmp].Val1;
72      int y = degreeCount[tmp].Val2;
```

```
73        sum += TMath::Log(x) * TMath::Log(y);
74        sum2 += TMath::Log(x);
75        sum3 += TMath::Log(y);
76        sum4 += TMath::Log(x)*TMath::Log(x);
77      }
78      int n2 = int(e) - int(s); if (n2 < 0.1*n) continue;
79      double b = ((n2 * sum) - (sum2*sum3)) / ((n2*sum4) - (sum2*sum2));
80      if (b_max > b) {
81        b_max = b;
82        s_max = s;
83      }
84    }
85
86    for (double incre = 0; incre < 0.5; incre = incre + 0.001) {
87      double sum = 0.0;
88      double sum2 = 0.0;
89      double sum3 = 0.0;
90      double sum4 = 0.0;
91
92      double s = s_max;
93      double e = (1 - incre)*n;
94      for (int tmp = int(s); tmp < int(e); tmp++) {
95        int x = degreeCount[tmp].Val1;
96        int y = degreeCount[tmp].Val2;
97        sum += TMath::Log(x) * TMath::Log(y);
98        sum2 += TMath::Log(x);
99        sum3 += TMath::Log(y);
100       sum4 += TMath::Log(x)*TMath::Log(x);
101     }
102     int n2 = int(e) - int(s); if (n2 < 0.1*n) continue;
103     double b = ((n2 * sum) - (sum2*sum3)) / ((n2*sum4) - (sum2*sum2));
```

```
104     if (b_max > b) {
105       b_max = b;
106       e_max = e;
107     }
108   }
109
110   return TFltPr(s_max, e_max);
111
112 }
113 // LogLog Plot with Line of Best-fit
114 void getLogLogPlotBestfit(const PUNGraph &Graph) {
115   char *command = "";
116   TVec<TIntPr> degreeCount;
117   for (PUNGraph::TObj::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++)
      {
118     bool exist = false;
119     for (int tmp = 0; tmp < degreeCount.Len(); tmp++) {
120       if (degreeCount[tmp].Val1 == NI.GetDeg()) {
121         degreeCount[tmp].Val2++;
122         exist = true;
123       }
124     }
125     if (!exist) {
126       degreeCount.Add(TIntPr(NI.GetDeg(), 1));
127     }
128   }
129
130   std::string cmd("x = [");
131   std::string cmd2(" y = [");
132
133   TFltPr range = bestPowerLawLocation(Graph);
```

```
134
135   for (int tmp = int(range.Val1); tmp < int(range.Val2); tmp++) {
136     cmd += std::to_string(degreeCount[tmp].Val2) + " ";
137     cmd2 += std::to_string(degreeCount[tmp].Val1) + " ";
138   }
139
140   cmd += "];";
141   cmd2 += "];";
142   cmd += cmd2;
143   cmd += "[x, index] = sort(x); y = y(index); y2 = x; x = y; y = y2;";
144   cmd += "figure;hold on;";
145   cmd += "loglog(x, y, '.');";
146   cmd += "p = polyfit(log(x),log(y),1);";
147   cmd += "y_hat = exp(p(1) * log(x) + p(2));";
148   cmd += "loglog(x,y_hat);";
149   cmd += "label = ['log(y) = ' num2str(p(1)) 'log(x) + ' num2str(p(2))]; ";
150   cmd += "legend('data',label);set(gca,'xscale','log');set(gca,'yscale','log')
        ;";
151
152   engEvalString(ep, cmd.c_str());
153   printMatlabOutput();
154 }
155
156
157 // LogLog Plot
158 void getLogLogPlot(const PUNGraph &Graph) {
159   std::string evalaa("figure;loglog([");
160   for (PUNGraph::TObj::TNodeI NI = Graph->BegNI(); NI < Graph->EndNI(); NI++)
        {
161     evalaa += std::to_string(NI.GetDeg()) + " ";
162   }
```

```
163     evalaa += "]);grid;";
164     engEvalString(ep, evalaa.c_str());
165 }
166
167
168 // Stop MATLAB Engine
169 void stopMatlab() {
170     engClose(ep);
171     delete[] buff;
172 }
```

Bibliography

[1] L.A. Adamic, O. Buyukkokten, E. Adar, A social network caught in the web, *First Monday* **8** (2003).

[2] Y. Ahn, S. Han, H. Kwak, S. Moon, H. Jeong, Analysis of topological characteristics of huge on-line social networking services, In: *Proceedings of the 16th International Conference on World Wide Web*, 2007.

[3] N. Alon, J. Spencer, *The Probabilistic Method*, Wiley, New York, 2000.

[4] L. Backstorm, P. Boldi, M. Rosa, J. Ugander, S. Vigna, Four degrees of separation, In: *Proceedings of the 4th Annual ACM Web Science Conference*, 2012.

[5] A.L. Barabási, R. Albert, Emergence of scaling in random networks, *Science* **286** (1999) 509-512.

[6] P. M. Blau, Population structure and exchange process, *Ethik Und Sozialwissenschaften* **6** (1995) 20-22.

[7] B. Bollobás, O. Riordan, J. Spencer, G. Tusnády, The degree sequence of a scale-free random graph process, *Random Structures and Algorithms* **18** (2001) 279-290.

[8] A. Bonato, *A Course on the Web Graph*, American Mathematical Society, Providence, Rhode Island, 2008.

[9] A. Bonato, D.F. Gleich, M. Kim, D. Mitsche, P. Pralat, A. Tian, S.J. Young, Dimensionality matching of social networks using motifs and eigenvalues, *PLOS ONE* **9** (2014) e106052.

[10] A. Bonato, N. Hadi, P. Horn, P. Prałat, C. Wang, Models of On-Line Social Networks, *Internet Mathematics* **6** (2011) 285-313.

[11] A. Bonato, J. Janssen, P. Prałat, Geometric protean graphs, *Internet Mathematics* **8** (2012) 2–28.

[12] A. Bonato, M. Lozier, D. Mitsche, X. Pérez-Giménez, P. Prałat, The domination number of on-line social networks and random geometric graphs, In: *Proceedings of the 12th Conference on Theory and Applications of Models of Computation (TAMC'15)*, 2015.

[13] T.F. Coleman, J.J. Moré, Estimation of sparse Jacobian matrices and graph coloring problems, *SIAM Journal on Numerical Analysis* **20** (1983) 187-209.

[14] B. Corominas-Murtra, B. Fuches, S. Thurner, Detection of the elite structure in a virtual multiplex social system by means of a generalized K-core, Preprint 2015.

[15] N.J. Cowan, E.J. Chastain, D.A. Vilhena, J.S. Freudenberg, C.T. Bergstrom, Nodal dynamics, not degree distributions, determine the structural controllability of complex networks, *PLOS ONE* **7** (2012) e38398.

[16] D. Crandall, D. Cosley, D. Huttenlocher, J. Kleinberg, S. Suri, Feedback effects between similarity and social influence in on-line communities, In: *Proceedings of the 14th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2008.

[17] P. Erdős, A. Rényi, On random graphs I, *Publicationes Mathematicae Debrecen* **6** (1959) 290-297.

[18] E. Estrada, Spectral scaling and good expansion properties in complex networks, *Europhysics Letters* **73** (2006), 649–655.

[19] M.R. Garey, D.S. Johnson, *Computers and intractability: a guide to theory of NP-completeness*, W.H. Freeman & Company, New York, 1979.

[20] S. Golder, D. Wilkinson, B. Huberman, Rhythms of social interaction: messaging within a massive on-line network, In: *Proceedings of 3rd International Conference on Communities and Technologies*, 2007.

[21] G.R. Grimmett, D.R. Stirzaker, *Probability and Random Processes*, Oxford University Press, 2001.

[22] S. Janson, T. Luczak, A. Rucinski, *Random Graphs*, Wiley-Interscience Series in Discrete Mathematics and Optimization, 2000.

[23] A. Java, X. Song, T. Finin, B. Tseng, Why we twitter: understanding microblogging usage and communities, In: *Proceedings of the Joint 9th WEBKDD and 1st SNA-KDD Workshop 2007*, 2007.

[24] A.D.I. Kramer, J.E. Guillory, J.T. Hancock, Experimental evidence of massive-scale emotional contagion through social networks, In: *Proceedings of the National Academy of Sciences*, 2014.

[25] R. Kumar, J. Novak, A. Tomkins, Structure and evolution of on-line social networks, In: *In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006.

[26] H. Kwak, C. Lee, H. Park, S. Moon, What is twitter, a social network or a news media?, In: *Proceedings of the 19th International World Wide Web Conference*, 2010.

[27] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication, In: *Proceedings of European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2005.

[28] J. Leskovec, J. Kleinberg, C. Faloutsos, Graphs over time: densification laws, shrinking diameters and possible explanations, In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2005.

[29] S. Lattanzi, D. Sivakumar, Affiliation Networks, In: *In Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, 2009.

[30] M. McPherson, A Blau space primer: prolegomenon to an ecology of affiliation, In: *Industrial and Corporate Change*, Oxford University Press, 2004.

[31] J.M. McPherson, J.R. Ranger-Moore, Evolution on a dancing landscape: organizations and networks in dynamic Blau space, *Social Forces* (1991) **70** 19-42.

[32] M. McPherson, L. Smith-John, J.M. Cook, Birds of a feather: Homophily in social networks, *Annual Review of Sociology* **27** (2001) 415-444.

[33] T. Milenković, V. Memišević, A. Bonato, N. Pržulj, Dominating biological networks, *PLOS ONE* **6** (2013) (8), e23016.

[34] S. Milgram, The small world problem, *Psychology Today* **2** (1967) 60-67.

[35] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, B. Bhattacharjee, Measurement and analysis of on-line social networks, In: *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, 2007.

[36] M.E.J. Newman, J. Park, Why social networks are different from other types of networks, *Physical Review E* **68**(3) 036122 (2003).

[37] Ø. Ore, *Theory of Graphs*, American Mathematical Society Colloquium Publications, Providence, Rhode Island, 1962.

[38] V. Pareto, *Cours d'Economie Politique*, Geneva: Droz, 1896.

[39] M. Rai, S. Verma, S. Tapaswi, A power aware minimum connected dominating set for wireless sensor networks, *Journal of networks* **4** (2009) 511-519.

[40] I. Stojmenovic, M. Seddigh, J. Zunic, Dominating sets and neighbor elimination-based broadcasting algorithms in wireless networks, *IEEE Transactions on Parallel and Distributed Systems* **13** (2002) 14-25.

[41] A.L. Traud, P.J. Mucha, M.A. Porter, Social structure of Facebook networks, *Physica A: Statistical Mechanics and its Applications* **391** (2012) 4165-4180.

[42] J.M.M. van Rooij, J. Nederlof, T.C. van Dijk, Inclusion/Exclusion Meets Measure and Conquer, *Algorithms - ESA 2009* **5757** (2009) 554-565.

[43] B. Viswanath, A. Mislove, M. Cha, K.P. Gummadi, On the evolution of user interaction in Facebook, In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, 2009.

[44] D.J. Watts, S. H. , CollectiStrogatzve dynamics of 'small-world' networks, *Nature* **393** (1998) 440–442.

[45] J. Wu, M. Cardei, F. Dai, S. Yang, Extended Dominating Set and its Applications in Ad-Hoc Networks using Cooperative Communication, In: *IEEE Transactions on Parallel and Distributed Systems*, 2006.