AFRL-RH-WP-TR-2008-0059

Investigation of Means of Mitigating Congestion in Complex, Distributed Network Systems by Optimization Means and Information Theoretic Proceedures

Frank Mufalli
Rakesh Nagi
Jim Llinas
Sumita Mishra

SUNY at Buffalo—CUBRC
4455 Genessee Street
Buffalo NY 14225

W.F. Lawless

Paine College
1235 15th Street
Augusta GA 30901-3182

February 2008

Final Report for the period August 2006 to November 2007

Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Battlespace Visualization Branch
Wright-Patterson AFB OH 45433

NOTICE AND SIGNATURE PAGE

TECHNICAL REVIEW AND APPROVAL

AFRL-RH-WP-TR-2008-0059

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

FOR THE DIRECTOR

//signed//
Daniel Repperger
Program Manager
Battlespace Visualization Branch

//signed//
DANIEL G. GODDARD
Chief, Warfighter Interface Division
Air Force Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
1 Feb 2008	Final	August 2006 – November 2007

4. TITLE AND SUBTITLE
Investigation of Means of Mitigating Congestion in Complex, Distributed Network Systems by Optimization Means and Information Theoretic Proceedures

5a. CONTRACT NUMBER
FA8650-06-1-6745

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62202F

6. AUTHOR(S)
Frank Mufalli *
Rakesh Nagi *
Jim Llinas *
Sumita Mishra *
W.F. Lawless **

5d. PROJECT NUMBER
7184

5e. TASK NUMBER
08

5f. WORK UNIT NUMBER
NY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SUNY at Buffalo—CUBRC *
4455 Genessee Street
Buffalo NY 14225

Paine College **
1235 15th Street
Augusta GA 30901-3182

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Materiel Command
Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Battlespace Visualization Branch
Wright Patterson AFB OH 45433-7022

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RHCV

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RH-WP-TR-2008-0059

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

88th ABW/PA Cleared 5/16/2008 WPAFB 08-3296.

14. ABSTRACT

This work investigates how the values of network metrics affect congestion in wireless ad-hoc networks. The metrics considered in this work are average path length, average degree, clustering coefficient, and offdiagonal complexity. Based on the levels of these metrics, insight is provided on the clustering algorithm to choose what will minimize congestion. Congestion is evaluated using a node betweenness measure and the candidate clustering algorithms are lowest ID, highest degree, and MOBIC. To obtain data for analysis, a network simulator was developed using Microsoft Visual C++ 2005. The simulator is capable of creating networks of varying complexity, clustering these networks using the aforementioned algorithms, and evaluating each of the five metrics. Analysis of the results confirmed that congestion levels increase with complexity. This was evidenced by evaluation of all five network metrics. Also, networks with relatively low levels of complexity will have minimal congestion, regardless of the clustering method used.

15. SUBJECT TERMS
Wireless ad-hoc network, Lowest ID Clustering, Highest Degree Clustering, MOBIC, Visualization

16. SECURITY CLASSIFICATION OF: Unclassified			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel Repperger
U	U	U	SAR	97	19b. TELEPONE NUMBER *(Include area code)*

This page blank intentionally.

Contents

List of Figures

List of Tables

Author's Acknowledgment

I would like to express sincere gratitude to my advisor Dr. Rakesh Nagi, who not only guided me through my thesis, but also challenged me to put forth my best effort in all aspects of my academic career. Along with Dr. Nagi, Dr. Sumita Mishra, Dr. James Llinas, and Jim Scandale have been a constant source of encouragement and guidance during my graduate studies. This work would not have been possible without their patience and support.

Summary

There are two key findings. First, complexity and congestion have a strong positive correlation. Second, for non-complex networks, the choice of a clustering algorithm is irrelevant. Each of the five metrics showed little or no congestion with low values. The clustering algorithm selected can minimize congestion given the level of a complexity metric.

Much time and effort was spent to develop the network simulation software. It is a versatile tool that can be utilized for other purposes. It was coded in a manner that makes it scalable for other network analyses. Additional network metrics and clustering methods can be integrated with little modification to the existing code But for analysis of networks larger than 150 nodes, the code must be optimized for more efficient run times. In its current state, analyzing networks exceeding 150 nodes is not feasible.

The network simulator is a console application that provides the means for user input and simulation status while results are exported for analysis. This method works well as long as the user is relatively “computer literate”. However, it can cause confusion for large-scale projects. As a fix, a graphical user interface could provide a clean interface. Visualization could then be incorporated into the software to graphically show node movement, congestion levels, and more.

1 Introduction and Research Objectives

1.1 Introduction

Traditionally, networks have operated in an infrastructure mode where designated hardware (routers, hubs, switches, etc.) is responsible for forwarding and maintaining the flow of data. With the advent of advanced warfare requirements, traditional networking methods were not adequate as they do not allow for the rapid deployment of resources (land vehicles, air vehicles, etc). This requirement led to the development of wireless ad-hoc networks.

A wireless ad-hoc network does not rely on fixed infrastructure or predetermined connectivity. It is a self organizing multi-hop wireless network in which all of the nodes can be mobile. Data is exchanged between nodes via wireless communication. Aside from the ability to be rapidly deployed, wireless ad-hoc networks have the ability to exist in highly volatile environments. Unlike traditional networks, if one node is destroyed it will not impact the data exchange between the remaining nodes within the network.

Given the dynamic characteristics of a wireless ad-hoc network, the way in which data is exchanged must be efficient to avoid congestion. This is necessary to ensure data transfers are quick and reliable. The primary factors affecting communication between nodes is the network topology and the routing method. Network topology and its impact on congestion is the focus of this work and an overview is provided in the literature review.

1.2 Research Objectives

There are two main objectives of this work. Primarily, the following network metrics will be analyzed to determine how they impact congestion in wireless ad-hoc networks:

- Average Path Length
- Average Degree
- Network Diameter
- Clustering Coefficient
- Offdiagonal Complexity

The values of these metrics will be studied for various random networks in an effort to link the metric levels to congestion levels. Furthermore, the secondary objective is to select the best clustering algorithm that will minimize congestion

given a particular metric level. For example, if we know the average path length is high, a clustering technique that minimizes congestion when the average path length is high should be selected. Our list of candidate clustering techniques is as follows:

- Lowest ID Clustering
- Highest Degree Clustering
- MOBIC

In order to complete these objectives, a network simulator was developed using Microsoft Visual C++ 2005. The network simulator has the ability to generate networks of various types and size, compute the five metrics as previously discussed, apply each of the clustering techniques, and evaluate congestion. Betweenness was selected to measure congestion. This, along with the network metrics and clustering techniques will be discussed in Chapter 2 and Chapter 3.

1.3 Outline of the Thesis

The rest of this work is outlined as follows. Chapter 2 provides a literature review of network topologies, clustering algorithms, and complexity metrics used in this work. Chapter 3 defines our technical approach and includes a detailed overview of the network simulation software. This leads into Chapter 4 which lays out the

experimental design and provides an analysis of the results obtained from simulation. Finally, Chapter 5 offers conclusions on these results and provides insight into future work.

2 Literature Review

2.1 Network Topology

The network simulation software developed in this work begins with a flat network and uses clustering techniques to convert the network into hierarchical form where congestion is evaluated. The decision to evaluate congestion in hierarchical networks was made as they are used in the vast majority of real world applications (i.e. environmental sensors, intelligence, etc.) Despite their increased overhead costs and vulnerability, exceptional communication efficiency makes hierarchical networks a more desirable choice. Both hierarchical and flat topologies will be discussed in detail.

2.1.1 Hierarchical Topology

The selection of the correct network topology given the network characteristics is extremely important to ensure reliable and efficient communication between nodes. The topology of a network can be either hierarchical or flat.

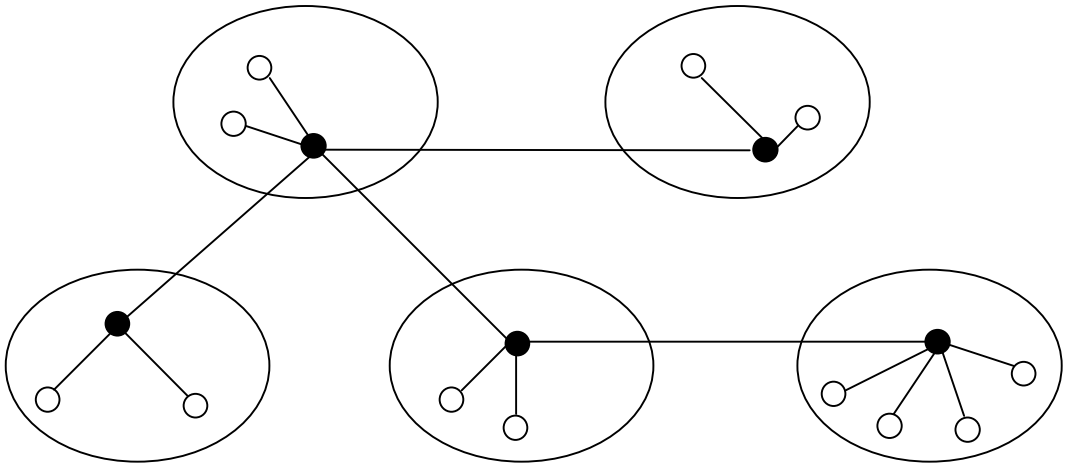


Figure 2.1: Example of a hierarchical topology

In a hierarchical topology nodes are divided into clusters. Within each cluster, a cluster head is selected via a mathematical formulation or heuristic method. Cluster heads are responsible for keeping track of which nodes are maintained in their respective cluster. Furthermore, they are responsible for transmitting data between clusters. The figure above is an example of a hierarchical topology. Each of the five ovals represents a cluster, and the black circles within each cluster represent the cluster heads. The white dots are regular nodes.

Each of the cluster heads maintains a continuously updated routing table. This table contains specific information detailing which cluster each node belongs to. If a node desires to transfer information to another node, the information is sent to the sending node's cluster head. This cluster head scans its routing table to determine which cluster the recipient is in. If the recipient is in the same cluster,

the data is immediately forwarded to the receiving node. If not, the cluster head scans its routing table to determine which cluster the recipient is in and forwards the data to the appropriate cluster head where it is again forwarded to the recipient.

2.1.2 Flat Topology

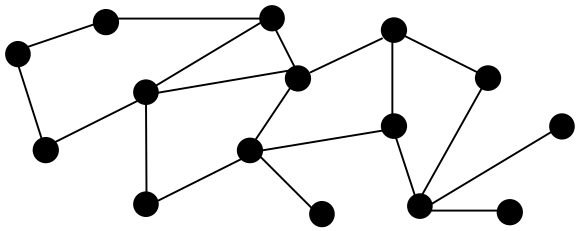


Figure 2.2: Example of a flat topology

Unlike hierarchical networks, flat networks do not contain cluster heads. All nodes are equal in terms of communication capabilities and each maintains its own routing information. The direct one-hop connections between nodes are generally based on proximity. Nodes communicate with each other via an infinitely variable number of hops between other nodes in the network. Haas and Tabrizi [5] favor a flat topology over a hierarchical topology for several reasons:

In hierarchical networks there is often times a single path between a pair of nodes. In a high threat environment, the elimination of a single node in that path would cause a communication failure between the nodes. This is avoided in flat networks as there are often a number of paths between a pair of nodes.

Nodes in a flat environment operate using much less power than nodes in a hierarchical environment. The additional power consumed in a hierarchical topology is due the constant change of clusters and cluster heads. When cluster heads change, the routing information among all cluster heads must update. This process requires a significant amount of energy. Lower power consumption will allow nodes with constrained energy resources to exist longer in the network. More importantly, lower energy use will give nodes a lower probability of being detected. Thus, the overall threat of attack to a network will be reduced.

In summary, hierarchical networks are advantageous to implement as they allow for more efficient communication among nodes. However, these networks have greater overhead costs and are more susceptible to attacks.

2.2 Clustering Algorithms

As stated in Section 2.1.1, clusterheads are a requirement of hierarchical networks. They are the channel of communication among all nodes in the network, making their selection key to successful data transfer throughout the entire network. The clustering algorithms selected for this work are highest degree, lowest id, and MOBIC. Each of these will be discussed in detail.

2.2.1 Highest Degree Clustering

Typically, the degree of a node is defined as the number of direct one-hop connections it has with other nodes. For this heuristic, the degree of a node is defined as the number of nodes within its transmission range. The highest degree clustering method assigns clusterheads using the following procedure [7]:

1. All of the nodes in the network are scanned and the node with the highest degree is selected as a cluster head.

2. All of the nodes in the transmission range of the selected cluster head are assigned to a cluster.

3. The remaining nodes that are not in a cluster are once again scanned and the process repeats until all nodes are assigned to a cluster.

This heuristic provides excellent stability within the clusters but lacks proper load balancing.

2.2.2 Lowest ID Clustering

The lowest ID clustering method [4, 6] is similar to the highest degree heuristic, but the selection method is based on a ID assigned to each node. The network is scanned and the node with the lowest ID is selected. As with the highest degree method, all nodes within the transmission range of the selected node form a cluster head. The process is repeated with the remaining nodes until all nodes are assigned to a cluster head.

2.2.3 MOBIC

Basu et al. [1] proposed this clustering algorithm which elects clusterheads based on relative mobility. This algorithm assumes that the position and speed of each node is unknown and uses transmission and signal strength to determine relative mobility. For the work, the exact position and speed of all nodes in the network is known at all times. Thus, slight modifications to the clusterhead election procedure were made. The procedure is as follows:

1. Calculate the pair wise relative mobility of each node: $M_Y^{rel} = 10 log_{10}\left(\frac{new\ position}{old\ position}\right)$
2. Calculate aggregate relative mobility of each node: $M_Y = E\left[\left(M_Y^{rel}\right)^2\right]$
3. Select node with lowest aggregate relative mobility among its neighbors as a cluster head

At this point, the clusterhead and all of the nodes connected to the clusterhead are considered to be covered. The steps of selecting a clusterhead repeat but only consider uncovered nodes. The algorithm is completed when all nodes are covered.

2.3 Complexity Metrics

The two example networks below will be referred to during the discussion of the complexity metrics.

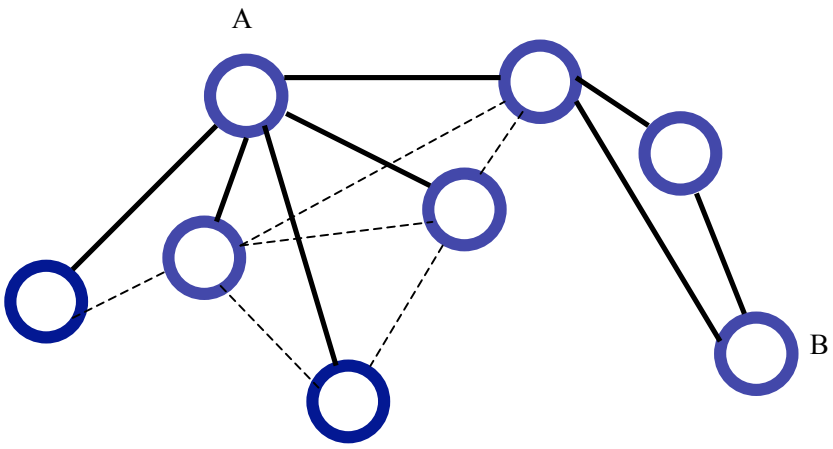


Figure 2.3: Depiction of example network 1

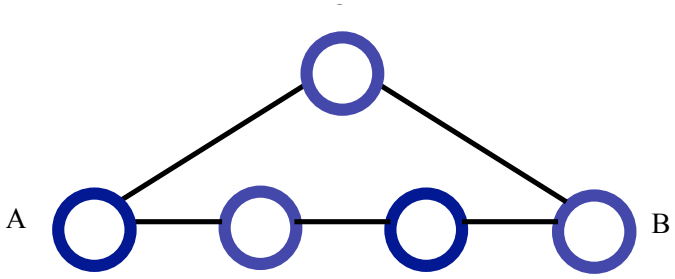


Figure 2.4: Depiction of example network 2

2.3.1 Average Path Length

The path length between two nodes can be defined as the smallest number of edges connecting them. This assumes that all of the edges have an equal length. The path

length between nodes A and B in the Example Network is 2. This can be formally written as follows:

$$l(A,B) = \min l(A,B)$$

The average path length of a network is simply the average of all the path lengths between nodes. This is formally defined as:

$$L =< l(A,B) >= 2/N(N-1)\sum_{A,B} l(A,B)$$

2.3.2 Network Diameter

The network diameter is defined as the maximum path length between any pair of nodes in the network. This is formally written as:

$$D = \max l(A,B)$$

For Example Network 2, the diameter of the network is 4. This is the maximum distance between any two pairs of nodes in the network.

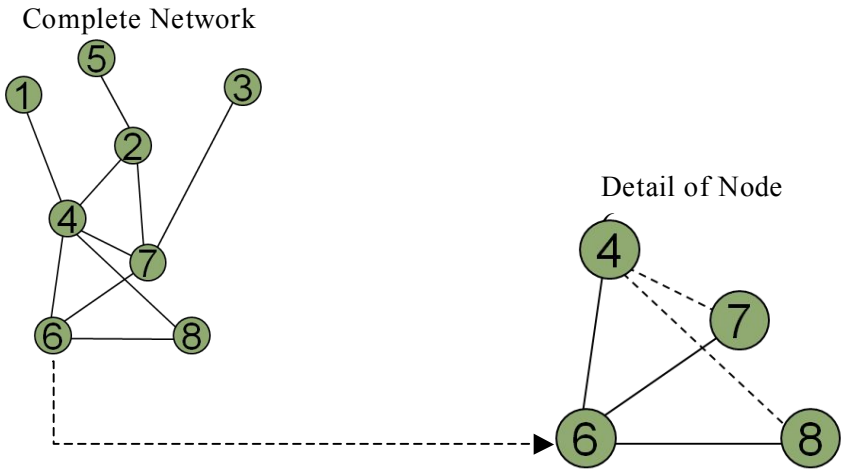
2.3.3 Average Degree

The degree of a node (k) is the sum of the edges it shares with other nodes. The degree of Node A in Example Network 1 has a degree of 5. The average degree of a network is simply the sum of the degrees of each node divided by the total number of nodes. This is formally written as:

$$< k >= 1/N\sum_{A} k_A$$

2.3.4 Network Clustering Coefficient

The clustering coefficient [8] can be explained through an example. Consider the network below:



Clustering Coefficient for Node 6 = A / B
C = Number of One Hop Connections
B = Number of Potential Connections Between Pairs of Connected Nodes
B = C (C-1) / 2 = 3 (3 – 1) / 2 = 3
A = Actual Connections Between These Pairs of Nodes = 2

Clustering Coefficient for Node 6 = 2 / 3

Figure 2.5: Method for determining the network clustering coefficient metric

Thus, the clustering coefficient of a node is the actual one hop connections between the neighbors of the node divided by all possible connections between neighbors of a node. The clustering coefficient of the entire network is simply the average of the clustering coefficients of all the nodes.

2.3.5 Offdiagonal Complexity

The offdiagonal complexity [2] is computed by first calculating the node-node link correlation matrix. The matrix appears as follows:

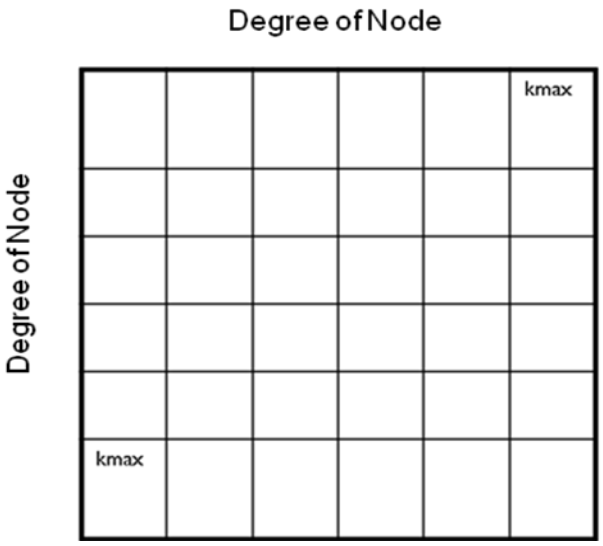


Figure 2.6: Layout of the node-node link correlation matrix

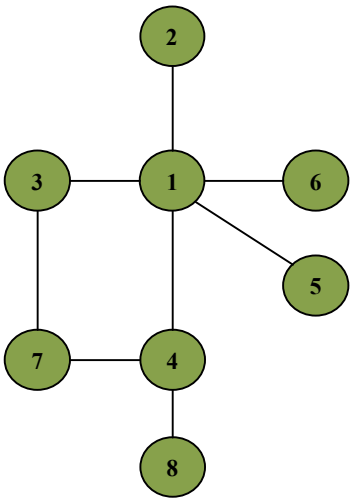
Each box in the square matrix above represents the number of connections between pairs of nodes with degrees corresponding to those indicated on the left and upper portions of the chart. The notation *kmax* simply represents the maximum degree of any node in the network. After this matrix is determined, the following formula is used to determine a_n:

$$a_n = \frac{\sum_{i=1}^{k_{max}-n} c_{i,i+n}}{\sum_{i=1}^{k_{max}} \sum_{j=1}^{k_{max}} c_{i,j}}$$

After all a_n values are computed, the offdiagonal complexity is calculated as follows:

$$\mathrm{OdC} \equiv -\sum_{k=1}^{k_{\max}} a_k \log a_k.$$

The example in Figure 2.7 helps clarify this computational procedure.



Node-Node Link Correlation

	1	2	3	5
1	0	1	1	3
2	1	0	1	1
3	1	1	0	1
5	3	1	1	0

$a_1 \quad \frac{(1) + (1) + (1)}{8}$ $a_2 \quad \frac{(1) +}{8}$ $a_3 \quad \frac{(3)}{8}$

$a_1 = 3/8$ $a_2 = 2/8$ $a_3 = 3/8$

OdC = .46999

Figure 2.7: A numerical example for computing offdiagonal complexity

3 Technical Approach

The first step in analyzing the metrics and clustering techniques was to obtain data via simulation output. Since the data required for this work is relatively unique, existing software capable of outputting the needed information was not commercially available. Thus, a customized network simulator specifically tailored to the objectives of this work was developed.

The simulation environment was designed in Microsoft Visual C++ 2005. The simulator is capable of generating a network based on several user defined characteristics, calculating five metrics to characterize the network, and cluster the network using three different techniques. The figure below represents an overview of the simulator. Each component of the figure will be discussed in detail.

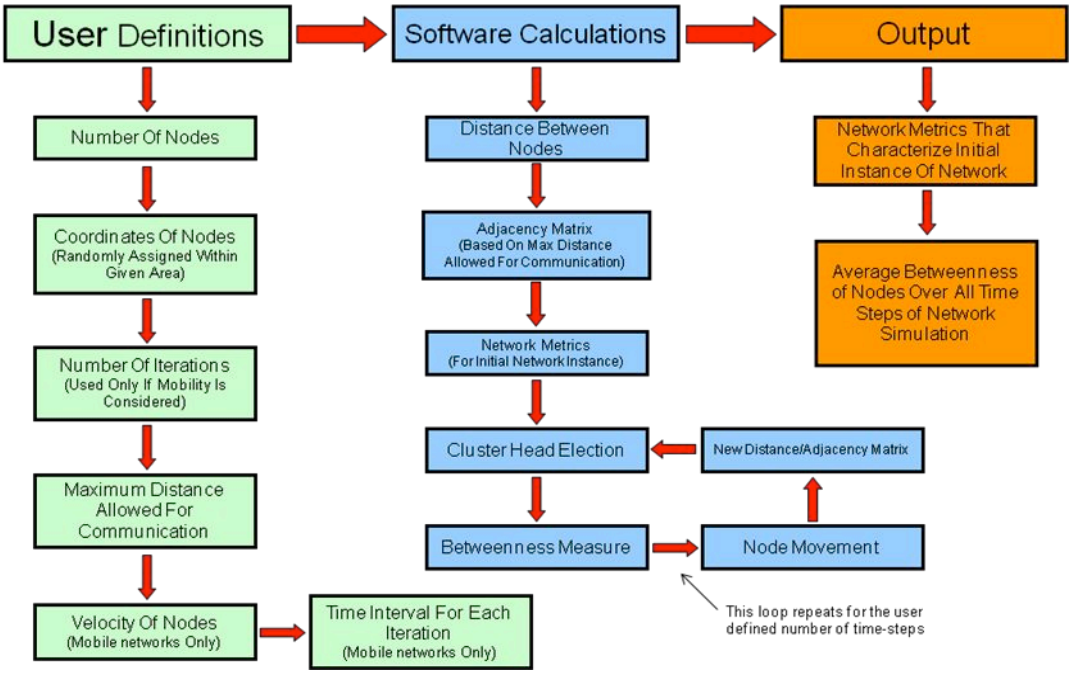


Figure 3.1: An overview of the simulation environment

3.1 User Definitions

The program begins with a user defining several network parameters. These parameters include the number of nodes in the network, the area of the field, the number of mobile iterations, the communication range for each node, the speed of each node, the number of replications, and the time interval for each instance of the network. The number of nodes is input at compile time. Thus, to enter the number of nodes the user must modify the header file named *NetGenDefs.h* and recompile the program. Clearly, this is not efficient programming and is something that could be worked out in a future build. After the user modifies the header file to include the desired number of nodes, the program is compiled and executed.

The user will be prompted to enter the square length of the field and press the enter key. Given that a user enters an integer P, this would indicate that the simulation area would have an area of P^2 meters. Next, the user is prompted to enter the maximum distance for communication and press enter. This value tells the program the farthest distance apart (in meters) two nodes may be to have communication with each other. Following this definition, the user must define the velocity of nodes (this is constant for all nodes in the network) and the time interval by which simulation calculations are made. Also, the user is prompted to specify the number of mobile iterations, or time steps, to include in the simulation run. To illustrate, assume the user enters 100, 20, and 10 for velocity, time interval, and iterations, respectively. This would indicate that all nodes are moving at 100 m/s, the time between each iteration is 20 seconds, and a total of 10 iterations will be considered. Finally, the user must define the number of replications. Each replication will begin with the same set of user defined parameters (i.e. number of nodes, field area, etc.) but the random number seed will be different (thus changing the random placement of nodes in the field). It should be noted that the program also allows these values to be "streamed in" instead of requiring a user to sit down and manually enter values for each simulation trial. This saves a substantial amount of time when running the simulation for multiple combinations of the parameters.

3.2 Software Calculations

Following the user definitions of network parameters, numerous software calculations are performed. First, the distance between each node in the matrix is calculated and stored in a square matrix called *distancematrix*. The elementary equation to determine the distance between two points is used and its implementation in C++ is shown below.

```
//Calculate distance between all nodes and store values in distancematrix

for ( int counter1 = 0; counter1 < NUMBER_OF_NODES; counter1++ )

{

for ( int counter2 = 0; counter2 < NUMBER_OF_NODES; counter2++ )

{

distancematrix[counter1][counter2] = sqrt((pow(nodexCurrent[counter1]-
nodexCurrent[counter2],2)) + (pow(nodeyCurrent[counter1]-
nodeyCurrent[counter2],2)));

}

}
```

Figure 3.2: C++ code for determining the distance between two nodes

Each value in *distancematrix* is compared with the maximum distance allowed for communication as previously discussed. If the *distancematrix* value is less than the maximum communication distance clearly there is a link between the corresponding nodes and thus an edge is created in the network. The square matrix

named *edgematrix* preserves all of these edges. A value of 1 in the edge matrix indicates that a connection exists, while a value of 0 indicates no connection. For clarification, consider the *distancematrix* and *edgematrix* below. Assume that the user defined the maximum distance for communication to be 40.

Distance Matrix			
	Node 1	Node 2	Node 3
Node 1	-	35	67
Node 2	35	-	50
Node 3	67	50	-

Table 3.1: Sample distance matrix

Edge Matrix			
	Node 1	Node 2	Node 3
Node 1	-	1	0
Node 2	1	-	0
Node 3	0	0	-

Table 3.2: Sample edge matrix

Since the distance between Node 1 and Node 2 is less than 40, a 1 has been placed in *edgematrix* to indicate that a connection (or edge) between these two nodes exists. At this stage, all of the preliminary definitions and calculations have been performed enabling the calculation of network metrics to begin.

The five network metrics are recomputed at each mobile instance of the simulation run. At the end of the simulation, the average of each metric is computed and outputted to a text file. The implementation of each metric is briefly discussed below.

3.2.1 Average Path Length

The calculation of this metric begins by determining the shortest path between each pair of nodes in the network. To do this, the Floyd-Warshall Algorithm [3] was implemented. The shortest paths are stored in a matrix named *shortestpath*. Below is the implementation of the Floyd-Warshall shortest path algorithm in C++.

```
for (int k = 0; k < NUMBER_OF_NODES; k++)
{
for (int i = 0; i < NUMBER_OF_NODES; i++)
{
for (int j = 0; j < NUMBER_OF_NODES; j++)
{
 shortestpath[i][j] = min(shortestpath[i][j], shortestpath[i][k] + shortestpath[k][j]) ;
}
}
}
```

Figure 3.3: C++ code for the Floyd-Warshall shortest path algorithm

It should be noted that zero values (indicating a lack of connection between nodes) in the *edgematrix* are temporarily converted to a large integer value (999). Otherwise, the algorithm would not work as zero would incorrectly be assigned as

the shortest path between many pairs of nodes in the network. After the shortest paths are calculated and stored for every pair of nodes in the network, the average value of the numbers in the matrix is calculated. The result of this calculation is the average path length. This algorithm is computed in *AvgPathLength.cpp*.

3.2.2 Network Diameter

The network diameter is defined as the longest shortest path. Thus, the Floyd-Warshall algorithm is again used to determine the shortest path between each pair of nodes in the network and a simple algorithm is used to determine maximum value in the *shortestpath* matrix. This value is returned as *maxpath* and is computed in *NetworkDiameter.cpp*.

3.2.3 Average Degree

The degree of a node is sum of edges connecting it to other nodes. Thus, the maximum possible degree of any node in the network is equal to the total number of nodes minus one. Furthermore, the degree of any node in a network is simply the sum of its corresponding row in *edgematrix*. Thus, summing all of the values in *edgematrix* and dividing by the total number of nodes gives us the average degree. This metric is computed in *AvgDegree.cpp*.

3.2.4 Clustering Coefficient

As discussed earlier, the clustering coefficient of a node is the number of actual connections between the neighbors of a node divided by the potential connections between the neighbors of a node.

Clustering Coefficient of a Node = A / B

A = Actual Connections Between Neighbors of a Node
B = Potential Connections Between Neighbors of a Node

The number of potential connections for each node (the denominator) is relatively simple to calculate. We mentioned earlier that the degree of a node could be computed by taking the sum of the corresponding row in *edgematrix*. The sum of each row (degree of each node) can then be stored in an array named *degreearray*. Thus, the denominator, B, can be computed as:

*coeffdenom[i] = (degreearray[i] * (degreearray[i] - 1)) / 2*

The numerator is a bit more challenging to compute. The details of the code used to compute the numerator can be found in the appendix. After the numerator is computed for each node it is stored in array named *coeffnum*. *Coeffnum* is then divided by *coeffdenom* and the result is stored in an array named *clusteringcoeff*. This array represents the clustering coefficient of each node in the network. The

average of all the values of this array is the network clustering coefficient. This metric is computed in ClusteringCoeff.cpp.

3.2.5 Offdiagonal Complexity

The offdiagonal complexity is the most difficult and demanding metric in the network simulation. Due to the complexity of this metric, the C++ implemenation will not be discussed in detail. The implementation follows the calculation steps as detailed earlier:

1. Develop Node-Node Link Correlation Matrix
2. Determine a_n values using appropriate formula
3. Determine offdiagonal complexity using appropriate formula

If interested in the detailed C++ implementation of offdiagonal complexity, the reader may compare the above steps with the code in the appendix. This metric is computed in *ODComplexity.cpp*.

The preceding metrics are computed for every time step of the network. Thus, if there are ten time steps, each of the metrics will be computed 10 times and only the average of these will be output to the user. After all of the network metrics are calculated at each time step, the network is clustered using three techniques.

3.2.6 Clustering Algorithms

The simulator utilizes the Lowest ID, Highest Degree, and MOBIC clustering algorithms at each time step in the simulation. These are discussed in detail in an earlier section. Similar to offdiagonal complexity, their C++ implementation is relatively complex for discussion. Once again, a reader interested in the detailed implementation is asked to compare the steps outlined in the prior section and compare them with the C++ code attached in the appendix. Lowest ID, Highest Degree, and MOBIC are computed in *LowestIDClustering.cpp*, *HighestDegreeClustering.cpp*, and *MOBICClustering.cpp*, respectively.

3.2.7 Betweenness

Following the clustering of the network using the aforementioned algorithms, the betweenness measure is calculated for all three cases. The betweenness of a node is the number of shortest paths passing through it. It is computed using a modified version of the Floyd-Warshall algorithm. This code is included in the appendix. The betweenness of each node is stored in an array named *betweenness*. Each value of the betweenness array is then compared with the user defined node capacity. Finally, the percentage of nodes in the *betweenness* array that exceed the

node capacity is calculated. This is our output measure. It is stored as a variable named *percentExceedingCapacity*. The betweenness measure is computed in *betweenness.cpp*.

3.2.8 Node Movement

Once the metrics have been calculated, clustering has been performed, and the betweenness is determined, we have successfully performed all of the requirements for a single time step in the network. The next task is to move the nodes based on the user defined velocity and time interval and prepare the next iteration of the network. The nodes move according to the following code.

```
for ( i = 0; i < NUMBER_OF_NODES; i++ )
{
 sincosvalueforiterations = (-15 + rand() % 15);

nodexCurrent[i] = nodexPrevious[i] + (cos(sincosvalueforiterations) * (nodespeed *
60)) ;

if (sincosvalueforiterations > 0)

nodeyCurrent[i] = nodeyPrevious[i] + (sin(sincosvalueforiterations) * (nodespeed *
60)) ;

else

nodeyCurrent[i] = nodeyPrevious[i] - (sin(sincosvalueforiterations) * (nodespeed *
60)) ;
```

Figure 3.4: C++ code for node movement

The array *nodexPrevious* and *nodeyPrevious* store the x-coordinates for every node in network before any movement takes place. The *sincosvalueforiteration*

variable indicates that the nodes in the network may move in a straight line anywhere between -15 and +15 degrees. The *nodespeed* variable is the user defined velocity of each node and the 60 indicates that there is 60 seconds between each time step. Finally, *nodexCurrent* and *nodeyCurrent* represent the new x and y coordinates for the subsequent iteration of the simulation. It should be noted that it is necessary to preserve the prior coordinates of the nodes as the MOBIC clustering algorithm elects clusterheads by examining the relative mobility.

3.3 Simulation Output and Summary

After the node movement has completed, we are ready to start the cycle over by recalculating all of the network metrics, determine clustering using each of the three techniques, determining betweenness, and once again move the nodes. The process continues until all of the time steps have completed. At the end of each time step, numerous pieces of data were sent to a comma separated file. The contents of this file include the values of all the user defined input, the random number seed used to generate the results, the average values of the metrics for all time steps, and the average percentage of nodes that exceeded capacity for each clustering algorithm over all time steps. The comma separated file was imported into excel where the data could be easily manipulated and results could developed.

4 Experimental Design and Results

4.1 Experimental Design

There was a significant limitation to consider when developing the experiments. As a result of the complexity and number of algorithms that must be performed for one time step of a simulation, the program took a substantial amount of time to produce results for simulations having more than 150 nodes. Thus, this was the maximum number of nodes considered for our experimental design. Further optimizations of the code and the algorithms within the code would allow for quicker computational times and a more robust analysis. The table on the following page represents the levels of variables that were tested.

Number Of Nodes	Node Capacity	Communication Range	Square Length Of Field
20	20	10	20
63	60	30	50
106	100	50	100
150	150	75	150
	200	100	200

Table 4.1: Levels of experimental design parameters

Five replications of every combination of metrics were tested. Thus, there were (4 X 5 X 5 X 5) X 5 replications = 2500 data points. Twenty-five of these data points were discarded because they did not have any node-node connection during the entire simulation. This was caused because too few nodes were distributed through too big of an area. Thus, a total of 2475 data points were used in the analysis. As a result of the large number of data points that were considered, the entire data sheet could not be included. However, a sample of the excel table used to determine the results is given in the appendix.

Using the results, comparisons were made between each metric and the impact that it had on congestion for each of the three clustering algorithms. This approach was selected so an individual could select the best clustering algorithm given a particular metric level. To develop each graph, all of the data was sorted for a given metric in ascending order. Furthermore, the data was equally divided into five groups containing 495 data points (2475 / 5 = 495). The range of values for

each group was simply determined by where the data points fell with respect to the value of the metric.

Two graphs were developed for each network metric. The first graph returns the average congestion levels for each type of clustering for different ranges of a metric. Given that a network metric falls within one of these ranges, this graph indicates how much congestion to expect for each type of clustering. The second graph details how many times a clustering algorithm resulted in the least amount of congestion given a specific range of a metric. In this graph, it should be noted that if two or more algorithms provided the least amount of congestion, they were both included in the graph. Thus, the theoretical sum of the values in each block of a graph is (495 X 3 = 1485). This would mean that for all 495 cases, the three clustering algorithms all provided the same average level of congestion.

4.2 Results

As discussed in the previous section, a total of ten graphs were developed (two for each complexity metric). Each of the graphs will be reviewed and conclusions with respect to the impact these metrics have on congestion. Also, insight will be provided on how varying levels of these metrics impact congestion. Analysis will begin with the average path length metric.

4.2.1 Average Path Length Results

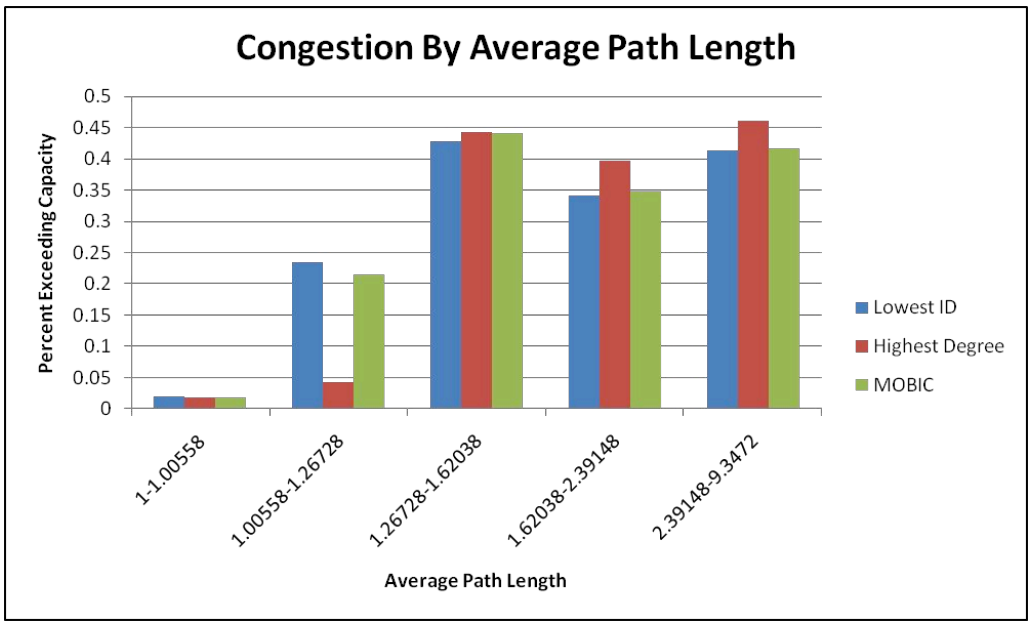


Figure 4.1: Bar graph showing congestion level by average path length

The first thing that can be noted about this graph is the overall increase in congestion as the average path length increases. This makes sense considering that as the average path length increases, the density, size, and connections among nodes also increases. When the average path length is very close to 1, it is evident that each of the clustering algorithms will equally provide minimal congestion. Interestingly, as the average path length increases to values between 1.0058 and 1.27, the highest degree clustering algorithm clearly provide the lowest level of congestion. As the average path length increases once again to values between 1.27 and 1.62 we can see that there is little difference in the clustering algorithms with respect to congestion, with Lowest ID having a slight advantage. When the

average path length is between 1.62 and 9.35 we can see that both the Lowest ID and MOBIC clustering methods perform better than the Highest Degree Method.

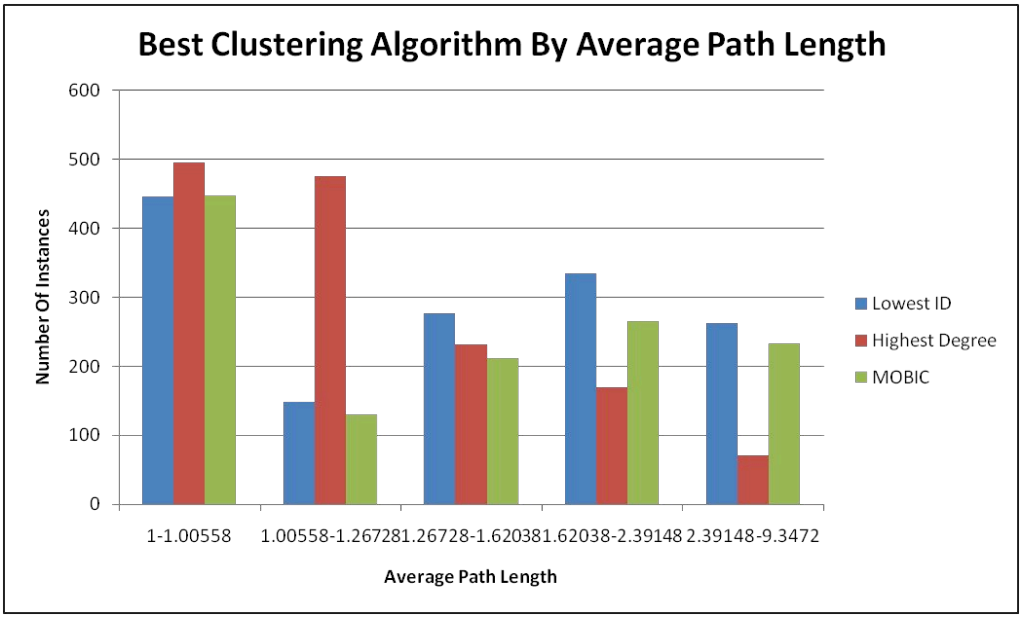


Figure 4.2: Bar graph showing clustering algorithm which minimizes congestion by average path length

The above graph validates what was previously stated. It is evident that when the path length is near 1, all three algorithms equally provide the best level of congestion. When the path length is between 1.0058 and 1.27, we can see that the Highest Degree method provided the lowest level of congestion for significantly more cases than the other two methods combined. When the average path length is greater than 1.27, it is clear that the Lowest ID method provides the least amount of congestion the majority of the time with MOBIC not too far behind. Clearly, as the average path length increases beyond a value of 1.27, the Highest Degree

method provides greater and greater levels of congestion making it a less desirable choice.

4.2.2 Average Degree Results

Next, we will look at the average degree metric.

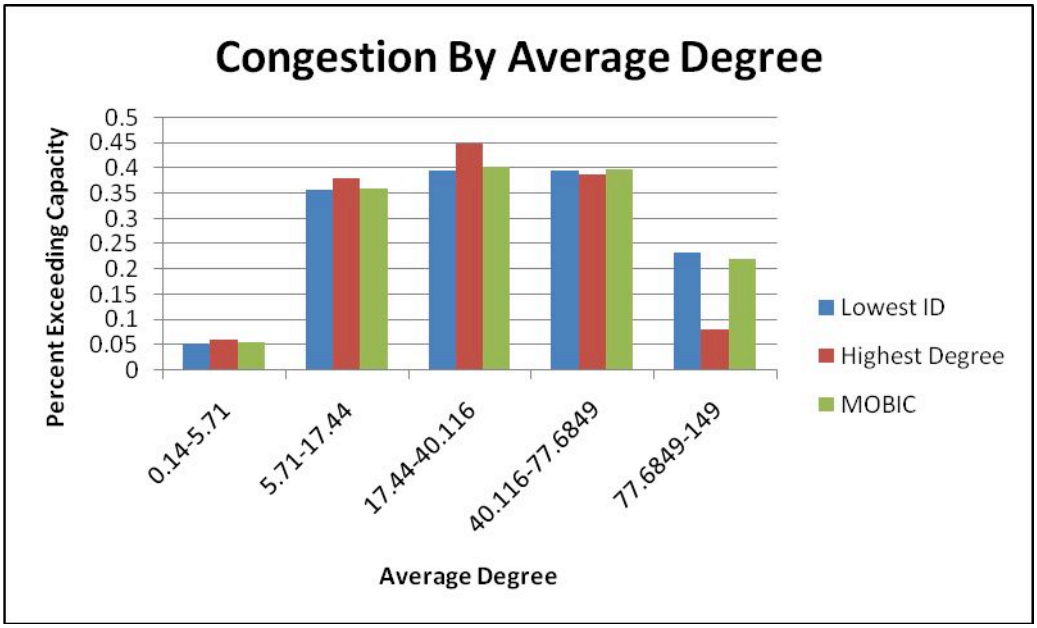


Figure 4.3: Bar graph showing congestion level by average degree

This graph shows that when a network has a low average degree, any clustering algorithm will provide minimal levels of congestion. Similar to average path length, a low average degree represents a small network with few edges between nodes. A low average degree could also be present with a large number of nodes, given they are spread out with little to no connectivity among each other (low density). Thus, it makes sense that congestion will be minimal regardless of the

clustering algorithm selected. When the average degree is between 5.71 and 40.116, congestion is relatively similar, with Lowest ID and MOBIC performing slightly better. However, when the average degree is between 77.69 and 149, the use of Highest Degree clustering provides networks with significantly less congestion. The lack of trend for the last block (77.69 – 149) is most likely a result of an inadequate sample size. The inadequate sample size was a result of the network simulators limitations, as previously stated in Chapter 4. The chart below validates these points.

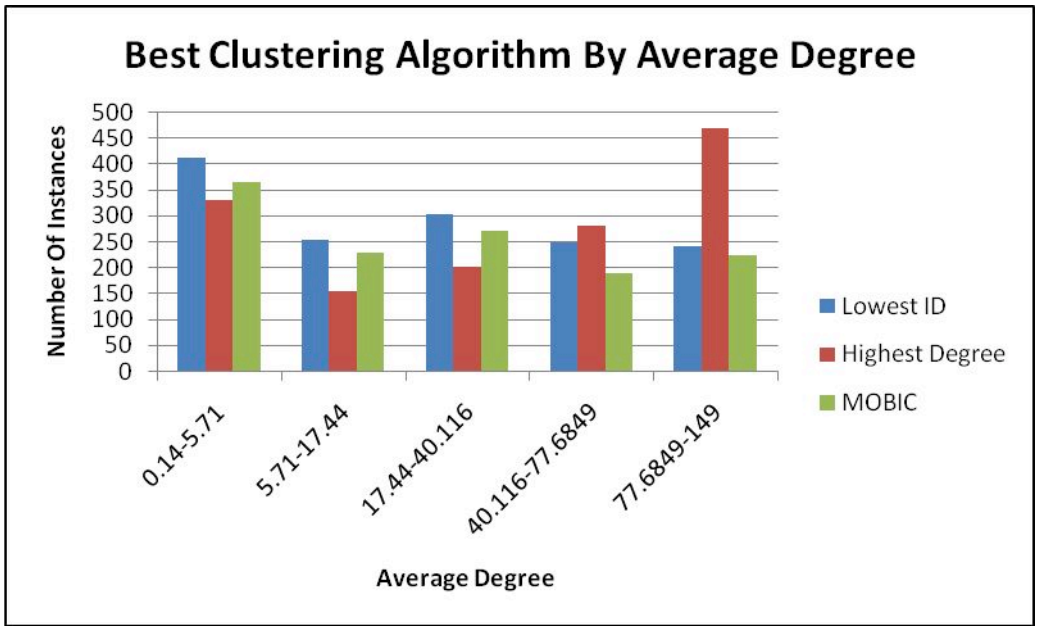


Figure 4.4: Bar graph showing clustering algorithm which minimizes congestion by average degree

4.2.3 Clustering Coefficient Results

Moving on, we will consider the network clustering coefficient.

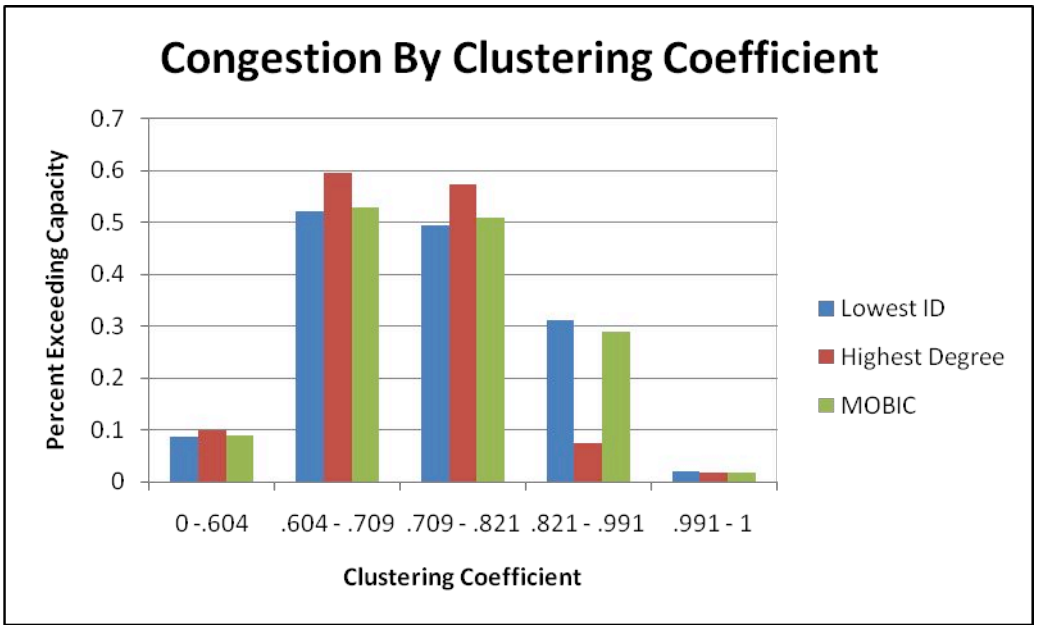


Figure 4.5: Bar graph showing congestion level by clustering coefficient

Once again, we see the same trend as in the previous two metrics. While the network is small and less complex (this corresponds to a relatively low clustering coefficient), the choice of clustering technique is irrelevant as all three produce low congestion levels. When the clustering coefficient is between .604 and .821 Lowest ID and MOBIC appear to be better choices to reduce congestion compared to Highest Degree. When the clustering coefficient is between .821 and .991, Highest Degree produced on average much lower levels of congestion compared to Lowest ID and MOBIC. Interestingly, for networks with high network clustering

coefficients all three of the algorithms produced very low levels of congestion. These results coincide with the results on the graph below.

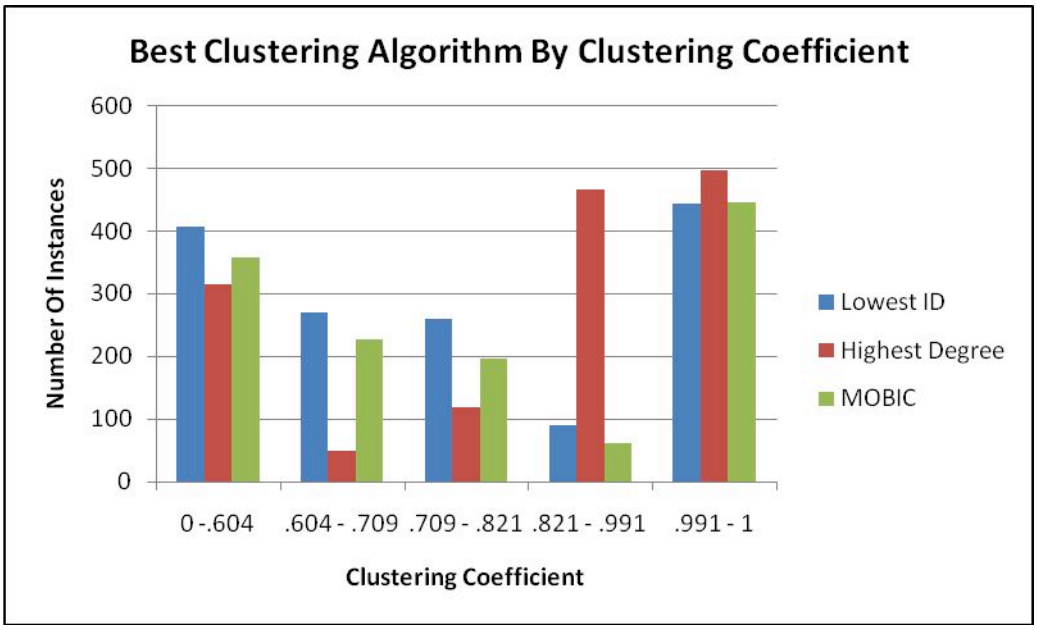


Figure 4.6: Bar graph showing clustering algorithm which minimizes congestion by clustering coefficient

Further analysis was performed to determine the cause of the results for network clustering coefficients between .991 and 1. The value of the clustering coefficient suggests that each node's neighbors are highly connected. This corresponds to a network with many edges. Since there are many edges and connections between nodes, it can be concluded that the average path length is relatively low. Going back to the results from average path length, we saw that networks with a low average path length had minimal congestion regardless of the clustering algorithm

selection. Thus, this explains the results for network clustering coefficients between .991 and 1.

4.2.4 Network Diameter Results

The network diameter metric will be discussed next.

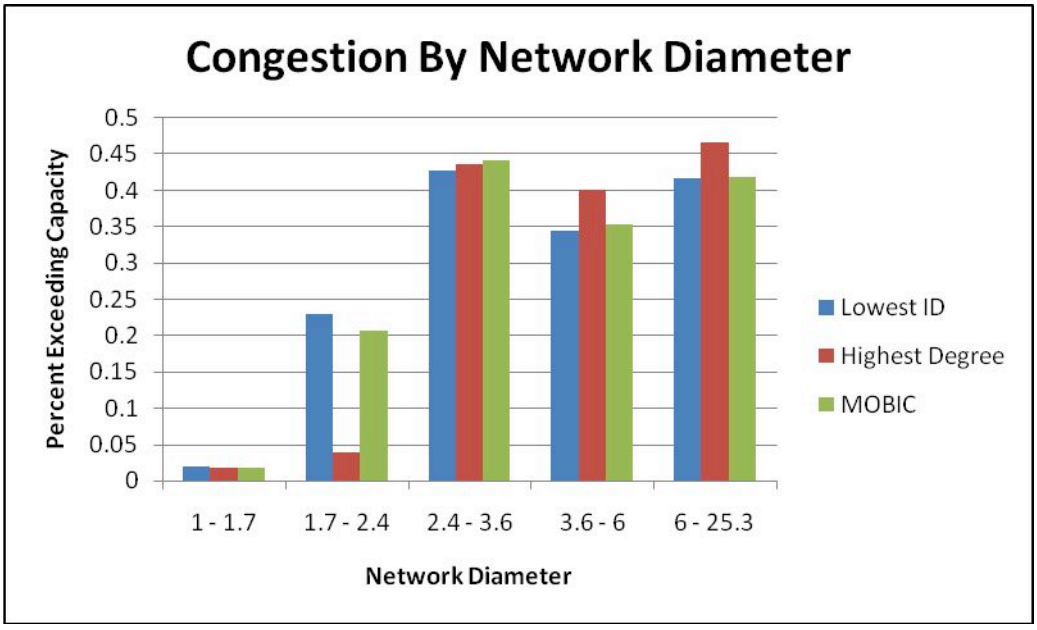


Figure 4.7: Bar graph showing congestion level by network diameter

The first thing that stands out about this graph is that the overall congestion increases as the network diameter increases. This is a validated since a higher network diameter typically corresponds to a more complex network, and thus, a more congested network. Recurring for the fourth time, we see that when the network diameter is low (i.e. network is small/simple) the choice of clustering algorithm is negligible. When the network diameter is between 1.7 and 2.4 the

benefit of selecting the highest degree algorithm is quite significant. Networks with diameters between 2.4 and 3.6 produce similar congestion levels despite the clustering technique used with Lowest ID having a slight advantage. When the network diameter is between 3.6 and 25.3, Highest Degree produces congestion levels that are relatively higher than both Lowest ID and MOBIC. Once again, this information is validated on the subsequent graph below.

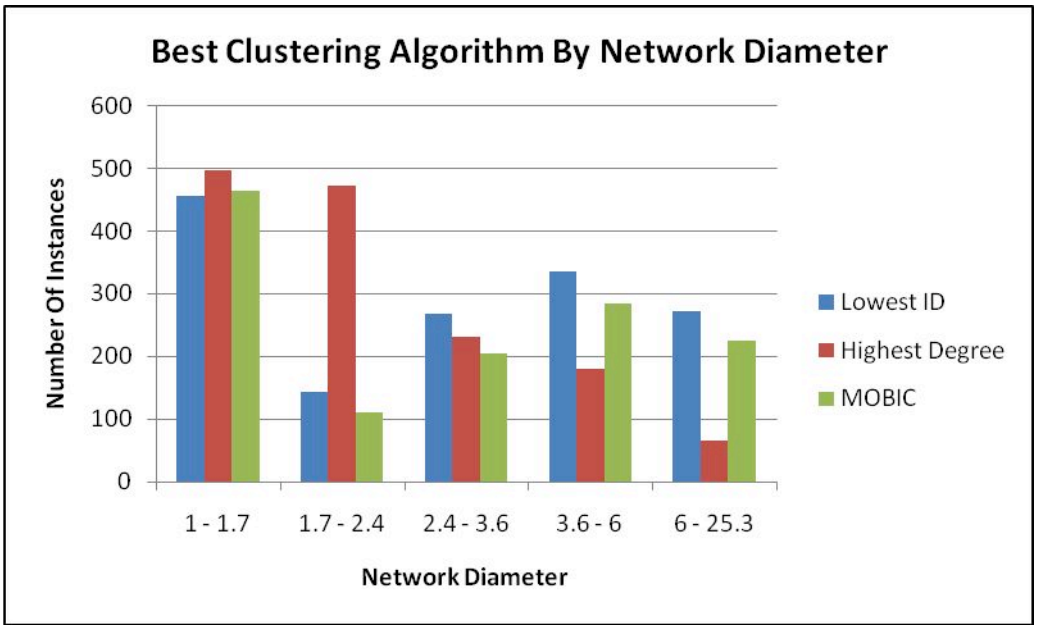


Figure 4.8: Bar graph showing clustering algorithm which minimizes congestion by network diameter

4.2.5 Offdiagonal Complexity Results

Finally, we will look at offdiagonal complexity.

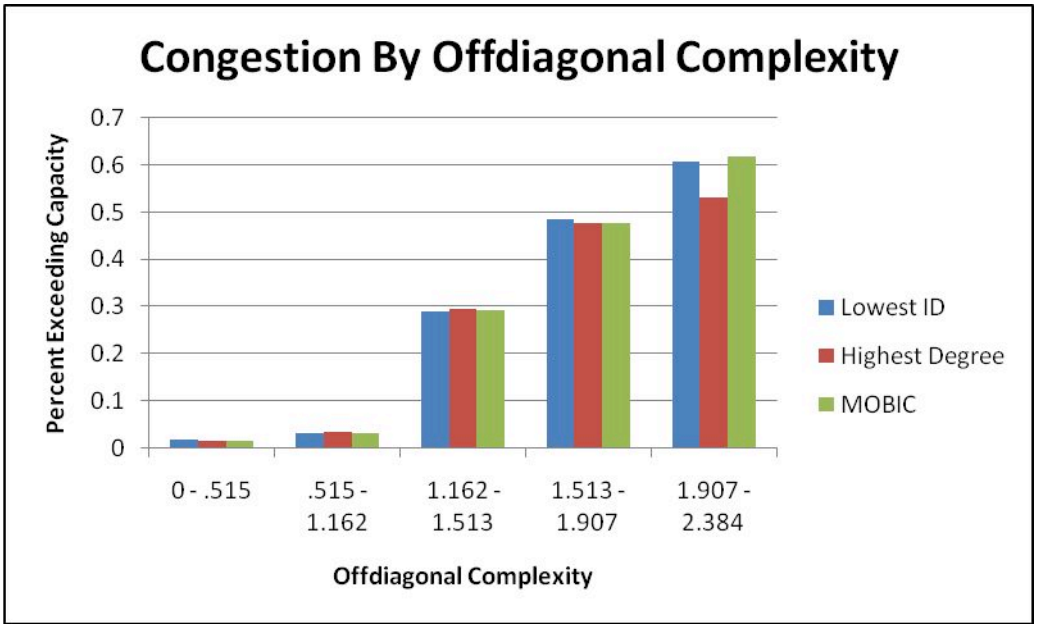


Figure 4.9: Bar graph showing congestion level by offdiagonal complexity

As the offdiagonal complexity increases, there is a steady increase in the overall congestion. Interestingly, for networks with offdiagonal complexities between 0 and 1.97, there does not appear to be a significant difference between clustering algorithms in terms of congestion. This may indicate that offdiagonal complexity is not as important as the other metrics when it is used to determine congestion. However, it should be noted that when the offdiagonal complexity is between 1.907 and 2.384, Highest Degree outperforms both Lowest ID and MOBIC. Once again, this is validated in the figure below.

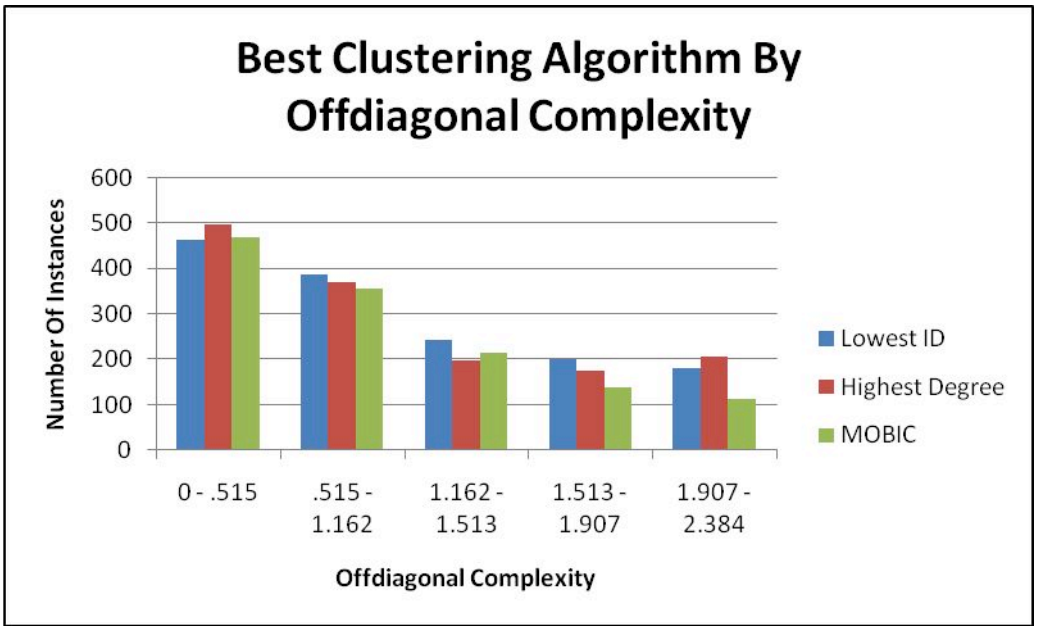


Figure 4.10: Bar graph showing clustering algorithm which minimizes congestion by offdiagonal complexity

5 Conclusions and Future Work

5.1 Conclusions

The results in the previous section conclude two key findings. First, it was shown that complexity and congestion have a strong positive correlation. Hence, congestion levels increase with complexity. This was revealed in all five of the complexity metrics that were analyzed. Secondly, for non-complex networks, the choice of clustering algorithm is irrelevant. Each of the five metrics showed little to no congestion when their values were low. Furthermore, all three clustering techniques provided congestion free networks in all five of these cases. The tables on the following page recap the clustering algorithm that should be selected to minimize congestion given the level of a complexity metric.

Average Path Length		Average Degree		Clustering Coefficient	
Metric Level	**Clustering Algorithm**	**Metric Level**	**Clustering Algorithm**	**Metric Level**	**Clustering Algorithm**
1 – 1.006	ANY	0.14 – 5.71	LID, MOBIC	0.00 – 0.60	LID, MOBIC
1.006 – 1.27	HD	5.71 – 17.44	LID, MOBIC	0.60 – 0.71	LID, MOBIC
1.27 – 1.62	LID	17.44 – 40.17	LID, MOBIC	0.71 – 0.82	LID
1.62 – 2.39	LID	40.17 – 77.69	HD	0.82 – 0.99	HD
2.39 – 9.35	LID, MOBIC	77.69 - 149	HD	0.99 – 1.00	ANY

Table 5.1: Summary of clustering algorithm which minimizes congestion given a specific level of average path length, average degree, or clustering coefficient

Network Diameter		Offdiagonal Complexity	
Metric Level	**Clustering Algorithm**	**Metric Level**	**Clustering Algorithm**
1.00 – 1.70	ANY	0.00 – 0.52	ANY
1.70 – 2.40	HD	0.52 – 1.16	ANY
2.40 – 3.60	LID	1.16 – 1.51	LID, MOBIC
3.60 – 6.00	LID	1.51 – 1.91	MOBIC
6.00 – 25.30	LID, MOBIC	1.91 – 2.38	HD

Table 5.2: Summary of clustering algorithm which minimizes congestion given a specific level of network diameter or offdiagonal complexity

5.2 Future Work

A great deal of time and effort was put forth in developing the network simulation software discussed in Chapter 3. This is a versatile tool that can be utilized for purposes outside of this work. The program was coded in a manner that makes it scalable for other network analyses. Additional network metrics and clustering methods can be integrated with little modification to the existing code. For analysis of networks larger than 150 nodes, the code must be optimized for more efficient run times. In its current state, analysis of networks exceeding 150 nodes is not feasible.

Currently, the network simulator is a console application. The console provides the means for user input and simulation status while results are exported to comma separated file. From here they can be opened in Microsoft Excel for analysis. This method works well as long as the user is relatively "computer literate". For large scale projects with multiple team members involved this may not work as some individuals may get confused. To remedy this, a graphical user interface should run on top of the software to provide a clean interface that will reduce confusion. Finally, for presentation purposes, visualization could be incorporated into the software to graphically show node movement, congestion levels, and more.

Appendix A

C++ Code for Network Simulator

The following is the C++ code for each file of the Network Simulator.

Main.cpp

```
#include <iostream>


#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include <vector>
#include <fstream>
#include "NetGenDefs.h"

using namespace std;

int main()
{
	ofstream outputFileDB;

// constant definitions
	const int iterationCount = 30;

//Variable Def
	int degreeofeachnode[NUMBER_OF_NODES] ;


// function prototypes
	double APL(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES]);
	double AD(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES]);
	double LDC(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES],
double nodeCapacity);
	double ND(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES]);
	int DD(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES], int
degreeofeachnode[NUMBER_OF_NODES], char outputFilename);
	double HDC(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES],
double nodeCapacity ) ;
	double BW(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES],
double nodeCapacity);
```

```
        double Clustering(int nn, int
edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES]) ;
        double ODC(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES],
int degreeofeachnode[NUMBER_OF_NODES]) ;
        double MOBIC(int nn, int
edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES], double
nodexPrevious[NUMBER_OF_NODES], double nodexCurrent[NUMBER_OF_NODES], double
nodeyPrevious[NUMBER_OF_NODES], double nodeyCurrent[NUMBER_OF_NODES], double
nodeCapacity);



        typedef vector<double> VectorMOBIC;
        VectorMOBIC storageTankMOBIC;

        typedef vector<double> VectorHD;
        VectorHD storageTankHD;

        typedef vector<double> VectorLD;
        VectorLD storageTankLD;

        typedef vector<double> VectorAPL;
        VectorLD storageTankAPL;

        typedef vector<double> VectorAD;
        VectorLD storageTankAD;

        typedef vector<double> VectorND;
        VectorLD storageTankND;

        typedef vector<double> VectorClustering;
        VectorLD storageTankClustering;

        typedef vector<double> VectorODC;
        VectorLD storageTankODC;

/*
//Seed The Generator
        int generatorseed ;
        cout << "\n\nPlease enter the random number seed and press enter.\n";
        cin >> generatorseed ;
        srand(generatorseed);

// get user parameters
        int iterationMAX;
        cout << "\n\nPlease enter the desired number of iterations and press enter.\n";
        cin >> iterationMAX ;

//Get Communication Distance
```

```
		int distanceThreshold;
		cout << "\n\nPlease enter the maximum internode distance and press enter.\n";
		cin >> distanceThreshold ;

//Get Square Length Of Field
		int fieldLengthandWidth ;
		cout << "\n\nPlease enter the field length.\n" ;
		cin >> fieldLengthandWidth ;

//Get Node Capacity For Network
		double nodeCapacity;
		cout << "\n\nPlease enter the node capacity and press enter.\n" ;
		cin >> nodeCapacity ;

//Get Filename For Output File
		char outputFilename;
		cout << "\n\nPlease enter the filename for the output data and press enter.\n";
		cin >> outputFilename;

*/

// Program initialization

		int iterationMAX;
		cout << "\n\nPlease enter the desired number of iterations and press enter.\n";
		cin >> iterationMAX ;


		double nodexPrevious[NUMBER_OF_NODES];
		double nodeyPrevious[NUMBER_OF_NODES];
		double nodexCurrent[NUMBER_OF_NODES];
		double nodeyCurrent[NUMBER_OF_NODES];

		double distancematrix[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0};
		double averagepathlength = 0;
		double averagedegree = 0 ;
		double sumofdegrees = 0 ;
		double ODComplexity = 0 ;
		double maxpath = 0 ;
		double percentExceedingCapacity = 0 ;
		double LDpercentExceedingCapacity = 0 ;
		double MOBICpercentExceedingCapacity = 0 ;
		double HDpercentExceedingCapacity = 0 ;
		double networkclusteringcoeff = 0 ;
		int lowestdegreematrix[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;
		vector <int> savedvaluesforlowestdegree ;
		double metricCalculationDecisicion = 0 ;
		double MOBICSum = 0 ;
		double HDSum = 0 ;
```

```
double LDSum = 0 ;
double MOBICOutput = 0 ;
double HDOutput = 0 ;
double LDOutput = 0 ;
double APLSum = 0 ;
double APLOutput = 0 ;
double ADOutput = 0 ;
double ADSum = 0 ;
double ODCOutput = 0 ;
double ODCSum = 0 ;
double NDOutput = 0 ;
double NDSum = 0 ;
double ClusteringSum = 0 ;
double ClusteringOutput = 0 ;

int generatorseed = 0 ;

int distanceThreshold = 0 ;
int fieldLengthandWidth = 0 ;
double nodeCapacity = 0 ;
char outputFilename = 0 ;


int distanceThresholdCounter[5] = {10, 30, 50, 75, 100} ;
double nodeCapacityCounter[5] = {20, 60, 100, 150, 193} ;
int fieldLengthandWidthCounter[5] = {20, 50, 100, 150, 200} ;
int generatorseedCounter[10] = {1, 5, 10, 20, 30, 40, 50, 60, 70, 80} ;


//Enter node speed in km/second
double nodespeed = .332 ;
int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0};
double sincosvalueforiterations ;


// Generate initial network node locations
for (int iterationCounter = 0; iterationCounter < iterationMAX; ++iterationCounter)
{
        for (int j=0; j < 5; j++)
        {
                for (int k=0; k < 5; k++)
                {
                                        for (int l=0; l < 5; l++)
                                        {
        outputFileDB.open ("N:\\ResultsForFinalMeeting\\FinalMeetingResults150.txt", ios::app);
        fieldLengthandWidth = fieldLengthandWidthCounter[j] ;
        nodeCapacity = nodeCapacityCounter[k]  ;
        distanceThreshold = distanceThresholdCounter[l] ;
        generatorseed = generatorseedCounter[iterationCounter] ;
```

```
//Set Initial X and Y Coordinates For All Nodes
for (int i = 0; i < NUMBER_OF_NODES; i++ )
{
        nodexCurrent[i] = rand() % fieldLengthandWidth;
        nodeyCurrent[i] = rand() % fieldLengthandWidth;
}


metricCalculationDecisicion = 0 ;

HDOutput = 0 ;
HDSum = 0 ;

MOBICOutput = 0 ;
MOBICSum = 0 ;

LDOutput = 0 ;
LDSum = 0 ;

APLSum = 0 ;
APLOutput = 0 ;

ADOutput = 0 ;
ADSum = 0 ;

ODCOutput = 0 ;
ODCSum = 0 ;

NDOutput = 0 ;
NDSum = 0 ;

ClusteringSum = 0 ;
ClusteringOutput = 0 ;

storageTankMOBIC.clear() ;
storageTankHD.clear() ;
storageTankLD.clear();
storageTankAPL.clear() ;
storageTankND.clear() ;
storageTankODC.clear();
storageTankClustering.clear() ;
storageTankAD.clear();


for (int i = 0 ; i < 10 ; i++)

{
```

```
//Allows metric calculations to be performed only on the first instance of the network
	metricCalculationDecisicion++ ;



//Calculate Distance Between All Nodes And Store Values In distancematrix
		for ( int i = 0; i < NUMBER_OF_NODES; i++ )
		{
			for ( int j = 0; j < NUMBER_OF_NODES; j++ )
			{
				distancematrix[i][j] = sqrt((pow(nodexCurrent[i]-
nodexCurrent[j],2)) + (pow(nodeyCurrent[i]-nodeyCurrent[j],2)));
			}
		}

/*Determine Edges Between Nodes (If Distance Between Two Points Is Less Than Or Equal To
distanceThreshold) And
Store Values In edgematrix*/
		for ( int i = 0; i < NUMBER_OF_NODES; i++ )
		{
			for ( int j = 0; j < NUMBER_OF_NODES; j++ )
			{
				if (distancematrix[i][j] <= distanceThreshold )

				{
					edgematrix[i][j] = 1;
				}

				if (distancematrix[i][j] > distanceThreshold )
				{
					edgematrix[i][j] = 999 ;
				}

				if ( i == j )
				{
					edgematrix[i][j] = 0 ;
				}
			}

		}

//Calculate Network Characteristics (Metrics) On INITIAL NODE COORDINATES ONLY

//if (metricCalculationDecisicion = 1)
//{
//		averagepathlength = APL(NUMBER_OF_NODES, edgematrix);
//		averagedegree = AD(NUMBER_OF_NODES, edgematrix);
```

```
//                      maxpath = ND(NUMBER_OF_NODES, edgematrix);
//                      percentExceedingCapacity = BW(NUMBER_OF_NODES, edgematrix,
nodeCapacity);
//                      networkclusteringcoeff = Clustering(NUMBER_OF_NODES, edgematrix);
//                      ODComplexity = ODC(NUMBER_OF_NODES, edgematrix,
degreeofeachnode);
//}

//          Prepare to do the next iteration of the network positions
                        for (int i = 0; i < NUMBER_OF_NODES; ++i)
                        {
                                nodexPrevious[i] = nodexCurrent[i];
                                nodeyPrevious[i] = nodeyCurrent[i];
                        }

                        for (int i = 0; i < NUMBER_OF_NODES; i++ )
                        {
                                sincosvalueforiterations = (-15 + rand() % 15);
                                nodexCurrent[i] = nodexPrevious[i] + (cos(sincosvalueforiterations) *
(nodespeed * 15)) ;

                                if (sincosvalueforiterations > 0)

                                        nodeyCurrent[i] = nodeyPrevious[i] +
(sin(sincosvalueforiterations) * (nodespeed * 15)) ;

                                else

                                        nodeyCurrent[i] = nodeyPrevious[i] -
(sin(sincosvalueforiterations) * (nodespeed * 15)) ;
                        }



                        int rval = DD(NUMBER_OF_NODES, edgematrix, degreeofeachnode,
outputFilename);


                        HDpercentExceedingCapacity = HDC(NUMBER_OF_NODES, edgematrix,
nodeCapacity) ;
                        LDpercentExceedingCapacity = LDC(NUMBER_OF_NODES, edgematrix,
nodeCapacity);
                        MOBICpercentExceedingCapacity = MOBIC( NUMBER_OF_NODES,
edgematrix,nodexPrevious,nodexCurrent, nodeyPrevious, nodeyCurrent, nodeCapacity ) ;

                        averagepathlength = APL(NUMBER_OF_NODES, edgematrix);
                        averagedegree = AD(NUMBER_OF_NODES, edgematrix);
                        maxpath = ND(NUMBER_OF_NODES, edgematrix);
                        networkclusteringcoeff = Clustering(NUMBER_OF_NODES, edgematrix);
```

```
                ODComplexity = ODC(NUMBER_OF_NODES, edgematrix,
degreeofeachnode);


                storageTankAPL.push_back(averagepathlength) ;
                storageTankAD.push_back(averagedegree) ;
                storageTankND.push_back(maxpath) ;
                storageTankClustering.push_back(networkclusteringcoeff) ;
                storageTankODC.push_back(ODComplexity) ;
                storageTankMOBIC.push_back(MOBICpercentExceedingCapacity) ;
                storageTankHD.push_back(HDpercentExceedingCapacity) ;
                storageTankLD.push_back(LDpercentExceedingCapacity) ;
}

//Calculate APL Average Over Time Steps
                for (unsigned int i = 0; i < storageTankAPL.size() ; i++)
                {
                        APLSum = APLSum + storageTankAPL[i] ;
                }

                APLOutput = APLSum / storageTankAPL.size() ;

//Calculate AD Average Over Time Steps
                for (unsigned int i = 0; i < storageTankAD.size() ; i++)
                {
                        ADSum = ADSum + storageTankAD[i] ;
                }

                ADOutput = ADSum / storageTankAD.size() ;

//Calculate ND Average Over Time Steps
                for (unsigned int i = 0; i < storageTankND.size() ; i++)
                {
                        NDSum = NDSum + storageTankND[i] ;
                }

                NDOutput = NDSum / storageTankND.size() ;

//Calculate Clustering Coefficient Average Over Time Steps
                for (unsigned int i = 0; i < storageTankClustering.size() ; i++)
                {
                        ClusteringSum = ClusteringSum + storageTankClustering[i] ;
                }

                ClusteringOutput = ClusteringSum / storageTankClustering.size() ;

//Calculate ODC Average Over Time Steps
                for (unsigned int i = 0; i < storageTankODC.size() ; i++)
                {
```

```
                ODCSum = ODCSum + storageTankODC[i] ;
        }

        ODCOutput = ODCSum / storageTankODC.size() ;


//Calculate MOBIC Betweenness Average Over Time Steps
        for (unsigned int i = 0; i < storageTankMOBIC.size() ; i++)
        {
                MOBICSum = MOBICSum + storageTankMOBIC[i] ;
        }

        MOBICOutput = MOBICSum / storageTankMOBIC.size() ;

//Calculate Lowest Degree Betweenness Average Over Time Steps
        for (unsigned int i = 0; i < storageTankLD.size() ; i++)
        {
                LDSum = LDSum + storageTankLD[i] ;
        }

        LDOutput = LDSum / storageTankLD.size() ;

//Calculate Highest Degree Betweenness Average Over Time Steps
        for (unsigned int i = 0; i < storageTankHD.size() ; i++)
        {
                HDSum = HDSum + storageTankHD[i] ;
        }

        HDOutput = HDSum / storageTankHD.size() ;

    outputFileDB << endl << NUMBER_OF_NODES << "," << nodeCapacity << "," <<
distanceThreshold << "," << generatorseed << "," << fieldLengthandWidth << "," <<  APLOutput
<< "," << ADOutput << "," << ClusteringOutput << "," << NDOutput << "," << ODCOutput <<
"," << percentExceedingCapacity << "," << LDOutput << "," << HDOutput << "," <<
MOBICOutput    ;

    outputFileDB.close();

//After Each Iteration Of The Program The Following Will Be Printed To The Screen
    cout << "\n\n\n\n\n\n**********RETURNED VALUES**********\n\n" ;
    cout << "\n\nThe average path length is " << averagepathlength ;
    cout << "\n\nThe clustering coefficient is " << networkclusteringcoeff ;
    cout << "\n\nThe Offdiagonal Complexity is " << ODComplexity ;
    cout << "\n\nThe average degree is " << averagedegree ;
    cout << "\n\nThe network diameter is " << maxpath ;
    cout << "\n\nThe seed used to generate these results was " << generatorseed ;
    cout << "\n\nThe percentage of nodes that exceed capacity for the initial instance is " <<
percentExceedingCapacity ;
```

```
	cout << "\n\nThe average percentage of nodes that exceed capacity using MOBIC for all
instance is " << MOBICOutput ;
	cout << "\n\nThe average percentage of nodes that exceed capacity using Highest Degree
for all instance is " << HDOutput ;
	cout << "\n\nThe average percentage of nodes that exceed capacity using Lowest Degree
for all instance is " << LDOutput ;
	cout << endl << endl ;

	}


		/*cout << "\n\nTHE EDGE MATRIX IS\n\n" ;
		for (int i=0; i < NUMBER_OF_NODES; ++i)
			{
				for (int j = 0; j < NUMBER_OF_NODES; j++)

				{
					if (edgematrix[i][j] == 999)

						edgematrix[i][j] = 0 ;
				}
		}*/


		/*for (int i=0; i < NUMBER_OF_NODES; ++i)
			{
				for (int j = 0; j < NUMBER_OF_NODES; j++)
			{
					printf("%d ",edgematrix[i][j]);
			}
					printf("\n");
			}*/


		//cout << "\n\nITERATION ENDS HERE" ;


		}
		}
		}


		 return 0;
}
```

NetGenDefs.h

```
#ifndef h_DEFINITIONSFORTHENETWORKGENERATOR_0001
  #define h_DEFINITIONSFORTHENETWORKGENERATOR_0001
  #define NUMBER_OF_NODES 150
#endif
```

AvgDegree.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include "NetGenDefs.h"

using namespace std;
double sumofdegrees ;
double averagedegree ;

double AD(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES])
{
        sumofdegrees = 0 ;

        for (int i=0; i < NUMBER_OF_NODES; i++)
                {
                        for (int j=0; j < NUMBER_OF_NODES; j++)
                        {
                                if(edgematrix[i][j] == 1 && i != j && edgematrix[i][j] != 999)
                                {
                                        sumofdegrees++ ;
                                }
                        }
                }

                averagedegree = (sumofdegrees / NUMBER_OF_NODES);
                return averagedegree;
}
```

AvgPathLength.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include "NetGenDefs.h"

using namespace std;
double averagepathlength ;

double APL(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES])
{
                double sumofshortestpaths = 0 ;
          double numberofpaths = 0 ;
                double shortestpath[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;

        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        shortestpath[i][j] = edgematrix[i][j];
                }
        }

                for (int k = 0; k < NUMBER_OF_NODES; k++)
                {
                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        shortestpath[i][j] = min(shortestpath[i][j],
shortestpath[i][k] + shortestpath[k][j]) ;
                                }
                        }
                }


                for (int i=0; i < NUMBER_OF_NODES; i++)
                {
                        for (int j=0; j < NUMBER_OF_NODES; j++)
                        {
                                if(shortestpath[i][j] > 0 && i!=j && shortestpath[i][j] != 999 )
                                {
                                        numberofpaths++;
                                }
                        }
                }
```

```
                    for (int i=0; i < NUMBER_OF_NODES; i++)
                    {
                            for (int j=0; j < NUMBER_OF_NODES; j++)
                            {
                                    if(i != j && shortestpath[i][j] != 999)
                                    {
                                            sumofshortestpaths = sumofshortestpaths +
shortestpath[i][j];
                                    }
                            }
                    }
                    sumofshortestpaths = sumofshortestpaths / 2 ;
                    numberofpaths = numberofpaths / 2 ;
                    averagepathlength = (sumofshortestpaths / numberofpaths) ;

                    return averagepathlength ;

}
```

Betweenness.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include "NetGenDefs.h"


using namespace std;

double BW(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES], double
nodeCapacity)
{
                double btwtemp = 0 ;
                double betweenness[NUMBER_OF_NODES];
                double btwshortestpath[NUMBER_OF_NODES][NUMBER_OF_NODES] ;
                double Number_Of_Nodes_That_Exceed_Capacity = 0 ;
                double percentExceedingCapacity;

                for(int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        betweenness[i] = 0;
                }

        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        btwshortestpath[i][j] = edgematrix[i][j];
                }
        }


                for (int k = 0; k < NUMBER_OF_NODES; k++)
                {
                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        btwshortestpath[i][j] = min(btwshortestpath[i][j],
btwshortestpath[i][k] + btwshortestpath[k][j]) ;
                                }
                        }
                }

        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
```

```
                                        {
                                                if (btwshortestpath[i][j] >= 2)
                                                {
                                                        for (int k = 0; k < NUMBER_OF_NODES;
k++)
                                                        {
                                                                if ( i != j && i != k && j != k &&
((btwshortestpath[i][k] + btwshortestpath[k][j]) == btwshortestpath[i][j]))
                                                                {
                                                                        betweenness[k]++;
                                                                }
                                                        }
                                                }
                        }
                        }


                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                betweenness[i] = (betweenness[i] / 2) ;
                        }

                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                if (nodeCapacity < betweenness[i])
                                {
                                        Number_Of_Nodes_That_Exceed_Capacity++ ;
                                }
                        }

                        percentExceedingCapacity = Number_Of_Nodes_That_Exceed_Capacity /
NUMBER_OF_NODES ;

                        /*cout << "\n\nThe percentage of nodes that exceed capacity is " <<
percentExceedingCapacity ;
                        cout << endl << endl ;*/


                        //cout << endl << endl ;

                        /*cout << "\n\nThe betweenness of each node in the network is...\n\n" ;

                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                cout << "Node " << i + 1 << " " << betweenness[i] ;
                                cout << "\n"  ;
                        }*/
                        return percentExceedingCapacity ;  }
```

ClusteringCoef.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include <vector>
#include "NetGenDefs.h"

using namespace std;


typedef vector<int>    integerArray;
typedef integerArray::iterator arrayPtr;

integerArray savedvaluesfordegree ;

double networkclusteringcoeff ;

double Clustering(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES])
{
double degreearray[NUMBER_OF_NODES] = {0};
double coeffdenom[NUMBER_OF_NODES] = {0} ;
double coeffnum[NUMBER_OF_NODES] = {0} ;
double clusteringcoeff[NUMBER_OF_NODES] = {0} ;
savedvaluesfordegree.clear();
double numsum = 0 ;
networkclusteringcoeff = 0 ;

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        savedvaluesfordegree.clear();

        for (int j = 0; j < NUMBER_OF_NODES; j++)
        {


                if (i != j && edgematrix[i][j] == 1)

                {
                        savedvaluesfordegree.push_back(j) ;
                }

                unsigned int k = 0;
                for (; k < savedvaluesfordegree.size(); k++)

                {
                        if (j == savedvaluesfordegree[k])
                        {
```

```
                                        degreearray[i]++ ;
                                }
                        }
                }
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        coeffdenom[i] = (degreearray[i] * (degreearray[i] - 1)) / 2  ;
}



for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        for (int j=0; j < NUMBER_OF_NODES; j++)
        {
                if (edgematrix[i][j] == 1)

                {
                        for (int k=0; k < NUMBER_OF_NODES; k++)
                        {
                                if ((edgematrix[i][k] == 1) && (edgematrix[j][k] == 1))
                                {
                                        coeffnum[j]++ ;
                                }
                        }
                }
        }
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        coeffnum[i] = coeffnum[i] / 2 ;
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        if (coeffdenom[i] != 0)
        {
                clusteringcoeff[i] = coeffnum[i] / coeffdenom[i] ;
        }
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        numsum = numsum + clusteringcoeff[i] ;
}
```

```
networkclusteringcoeff = (numsum / NUMBER_OF_NODES) ;
//cout << "\n\nThe network clustering coefficient is " << networkclusteringcoeff << endl ;
return   networkclusteringcoeff ;
}
```

HighestDegreeClustering.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include <vector>
#include "NetGenDefs.h"

using namespace std;

typedef vector<int>    integerArray;
typedef integerArray::iterator arrayPtr;

integerArray savedvaluesforhighestdegree ;


double HDC(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES], double
nodeCapacity)
{

savedvaluesforhighestdegree.clear();
int degreeArray[NUMBER_OF_NODES] = {0} ;
int sortedDegreeArray[NUMBER_OF_NODES] = {0}  ;
int originalPositionArray[NUMBER_OF_NODES] = {0} ;
double highestdegreematrix[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;
int temp = 0 ;
int temp1 = 0 ;

for (int i=0; i < NUMBER_OF_NODES; ++i)
                                {
                                        degreeArray[i] = 0;
                                }

//Compute Degree For Each Node
for (int i=0; i < NUMBER_OF_NODES; i++)
{
        for (int j=0; j < NUMBER_OF_NODES; j++)
        {
                if ( i != j && edgematrix[i][j] == 1)
                {
                        degreeArray[i]++ ;
                }
        }
}

//Copy Contents Of degreeArray To sortedDegreeArray
for (int i = 0; i < NUMBER_OF_NODES; i++)
                {
```

```
                    sortedDegreeArray[i] = degreeArray[i] ;
                    originalPositionArray[i] = i ;
                }

//Sort Contents Of sortedDegreeArray In Descending Order

        for (int i = 0; i < NUMBER_OF_NODES ; i++)

        {
                for(int j=0; j < NUMBER_OF_NODES; j++)

                {
                        if(sortedDegreeArray[i] > sortedDegreeArray[j])
                        {
                                temp = sortedDegreeArray[i] ;
                                sortedDegreeArray[i] = sortedDegreeArray[j] ;
                                sortedDegreeArray[j] = temp ;
                                temp1 = originalPositionArray[i] ;
                                originalPositionArray[i] = originalPositionArray[j] ;
                                originalPositionArray[j] = temp1 ;
                        }
                }
        }

        for (int i=0; i < NUMBER_OF_NODES; i++)
                {
                        unsigned int p = 0;
                        for (;p < savedvaluesforhighestdegree.size(); ++p )
                        {
                                if ( (originalPositionArray[i] ==
savedvaluesforhighestdegree[p]) )
                                {
                                        break;
                                }
                        }

                        if (p == savedvaluesforhighestdegree.size())
                        {
                                for (int j=0; j < NUMBER_OF_NODES; j++)
                                {
                                        if (edgematrix[originalPositionArray[i]][j] != 999 &&
originalPositionArray[i] != j)
                                        {
                                        highestdegreematrix[originalPositionArray[i]][j] =
edgematrix[originalPositionArray[i]][j];
                                        savedvaluesforhighestdegree.push_back(j) ;
                                        }
                                }
```

```
                        }
        }


/*cout << "\n\n\n" ;
cout << "Highest Degree Clustering.....\n\n" ;
for (int i=0; i < NUMBER_OF_NODES; ++i)
                        {
                                cout << endl ;
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                        {
                                cout << highestdegreematrix[i][j] << " " ;
                        }
}*/

//Betweenness Test Starts Here
                double HDbtwtemp = 0 ;
                double HDbetweenness[NUMBER_OF_NODES];
                double HDbtwshortestpath[NUMBER_OF_NODES][NUMBER_OF_NODES] ;
                double HDNumber_Of_Nodes_That_Exceed_Capacity = 0 ;
                double HDpercentExceedingCapacity = 0;

                for(int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        HDbetweenness[i] = 0;
                }

        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        HDbtwshortestpath[i][j] = highestdegreematrix[i][j];
                }
        }

        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        if (HDbtwshortestpath[i][j] == 0)
                        HDbtwshortestpath[i][j] = 999;
                }
        }


        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
```

```
                        if (HDbtwshortestpath[i][j] == 1)
                        HDbtwshortestpath[j][i] = 1;
                }
        }


        for (int k = 0; k < NUMBER_OF_NODES; k++)
                {
                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        HDbtwshortestpath[i][j] =
min(HDbtwshortestpath[i][j], HDbtwshortestpath[i][k] + HDbtwshortestpath[k][j]) ;
                                }
                        }
                }


        /*cout << "\n\n\n" ;
        cout << "Shortest Path.....\n\n" ;
        for (int i=0; i < NUMBER_OF_NODES; ++i)
                        {
                                cout << endl ;
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                        {
                                cout << HDbtwshortestpath[i][j] << " " ;
                        }
                        }*/


        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        if (HDbtwshortestpath[i][j] >= 2)
                                        {
                                                for (int k = 0; k < NUMBER_OF_NODES;
k++)
                                                {
                                                        if ( i != j && i != k && j != k &&
((HDbtwshortestpath[i][k] + HDbtwshortestpath[k][j]) == HDbtwshortestpath[i][j]))
                                                        {
                                                                HDbetweenness[k]++;
                                                        }
                                                }
                                        }
                }
                }
```

```
		for (int i = 0; i < NUMBER_OF_NODES; i++)
		{
			HDbetweenness[i] = (HDbetweenness[i] / 2) ;
		}


		for (int i = 0; i < NUMBER_OF_NODES; i++)
		{
			if (nodeCapacity < HDbetweenness[i])
			{
				HDNumber_Of_Nodes_That_Exceed_Capacity++ ;
			}
		}

		HDpercentExceedingCapacity = HDNumber_Of_Nodes_That_Exceed_Capacity
/ NUMBER_OF_NODES ;

		/*cout << "\n\nThe percentage of nodes that exceed capacity is " <<
HDpercentExceedingCapacity ;
		cout << endl << endl ;


		cout << endl << endl ;

		cout << "\n\nThe betweenness of each node in the network is...\n\n" ;

		for (int i = 0; i < NUMBER_OF_NODES; i++)
		{
			cout << "Node " << i + 1 << " " << HDbetweenness[i] ;
			cout << "\n"  ;
		} */

		return HDpercentExceedingCapacity ;

return 0;


}
```

LowestIDClustering.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include <vector>
#include "NetGenDefs.h"

using namespace std;


typedef vector<int>    integerArray;
typedef integerArray::iterator arrayPtr;

integerArray savedvaluesforlowestdegree ;


double LDC(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES], double
nodeCapacity)
{
int lowestdegreematrix[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0};
savedvaluesforlowestdegree.clear();
for (int i=0; i < NUMBER_OF_NODES; i++)
                {
                        unsigned int p = 0;
                        for (;p < savedvaluesforlowestdegree.size(); ++p)
                        {
                                if (i == savedvaluesforlowestdegree[p])
                                {
                                        break;
                                }
                        }
                        if (p == savedvaluesforlowestdegree.size())
                        {
                                for (int j=0; j < NUMBER_OF_NODES; j++)
                                {
                                        if(i != j && edgematrix[i][j] == 1)
                                        {
                                                lowestdegreematrix[i][j] = 1 ;
                                                savedvaluesforlowestdegree.push_back(j) ;
                                        }
                                }
                        }
                }

/*cout << "\n\n\n" ;
cout << "Lowest ID Clustering.....\n\n" ;
for (int i=0; i < NUMBER_OF_NODES; ++i)
```

```
                                {
                                        for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                                printf("%d ",lowestdegreematrix[i][j]);
                                }
                                                printf("\n");
                                }*/

//Betweenness Test Starts Here
double btwtemp = 0 ;
                        double LDbetweenness[NUMBER_OF_NODES];
                        double LDbtwshortestpath[NUMBER_OF_NODES][NUMBER_OF_NODES] ;
                        double LDNumber_Of_Nodes_That_Exceed_Capacity = 0 ;
                        double LDpercentExceedingCapacity;

                        for(int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                LDbetweenness[i] = 0;
                        }

                for(int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        for(int j = 0; j < NUMBER_OF_NODES; j++)
                        {
                                LDbtwshortestpath[i][j] = lowestdegreematrix[i][j];
                        }
                }

                for(int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        for(int j = 0; j < NUMBER_OF_NODES; j++)
                        {
                                if (LDbtwshortestpath[i][j] == 0)
                                LDbtwshortestpath[i][j] = 999;
                        }
                }

                for(int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        for(int j = 0; j < NUMBER_OF_NODES; j++)
                        {
                                if (LDbtwshortestpath[i][j] == 1)
                                LDbtwshortestpath[j][i] = 1;
                        }
                }


                        for (int k = 0; k < NUMBER_OF_NODES; k++)
                        {
```

```
                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        LDbtwshortestpath[i][j] =
min(LDbtwshortestpath[i][j], LDbtwshortestpath[i][k] + LDbtwshortestpath[k][j]) ;
                                }
                        }
                }

        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        if (LDbtwshortestpath[i][j] >= 2)
                                        {
                                                for (int k = 0; k < NUMBER_OF_NODES;
k++)
                                                {
                                                        if ( i != j && i != k && j != k &&
((LDbtwshortestpath[i][k] + LDbtwshortestpath[k][j]) == LDbtwshortestpath[i][j]))
                                                        {
                                                                LDbetweenness[k]++;
                                                        }
                                                }
                                        }
                }
                }


                for (int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        LDbetweenness[i] = (LDbetweenness[i] / 2) ;
                }

                for (int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        if (nodeCapacity < LDbetweenness[i])
                        {
                                LDNumber_Of_Nodes_That_Exceed_Capacity++ ;
                        }
                }

                LDpercentExceedingCapacity = LDNumber_Of_Nodes_That_Exceed_Capacity /
NUMBER_OF_NODES ;

        /*      cout << "\n\nThe percentage of nodes that exceed capacity is " <<
LDpercentExceedingCapacity ;
                cout << endl << endl ;
```

```
                cout << endl << endl ;

                cout << "\n\nThe betweenness of each node in the network is...\n\n" ;

                for (int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        cout << "Node " << i + 1 << " " << LDbetweenness[i] ;
                        cout << "\n"  ;
                }*/

                return LDpercentExceedingCapacity ;

return   0;

}
```

MobicClustering.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include <vector>
#include "NetGenDefs.h"

using namespace std;
double distancematrixcurrent[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;
double distancematrixprevious[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;
double distancematrixMOBIC[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;
double MOBICArray[NUMBER_OF_NODES] = {0} ;
double MOBICMean[NUMBER_OF_NODES] = {0} ;
double MOBICVarianceMatrix[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;
double MOBICMyMatrix[NUMBER_OF_NODES] = {0} ;
double MOBICClusteringMatrix[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;
int originalPositionArrayMOBIC[NUMBER_OF_NODES] = {0} ;
double sortedMOBICArray[NUMBER_OF_NODES] = {0}  ;
double temp = 0 ;
int temp1 = 0 ;

typedef vector<int>    integerArrayMOBIC;
typedef integerArrayMOBIC::iterator arrayPtr;

integerArrayMOBIC savedvaluesforMOBIC ;

double MOBIC(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES], double
nodexPrevious[NUMBER_OF_NODES], double nodexCurrent[NUMBER_OF_NODES], double
nodeyPrevious[NUMBER_OF_NODES], double nodeyCurrent[NUMBER_OF_NODES], double
nodeCapacity )
{

savedvaluesforMOBIC.clear() ;


//Clear contents of all arrays
for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        for (int j = 0; j < NUMBER_OF_NODES; j++)
        {
                distancematrixcurrent[i][j] = 0 ;
        }
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
        for (int j = 0; j < NUMBER_OF_NODES; j++)
```

```
		{
			distancematrixprevious[i][j] = 0 ;
		}
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
		for (int j = 0; j < NUMBER_OF_NODES; j++)
		{
			distancematrixMOBIC[i][j] = 0 ;
		}
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
		MOBICArray[i] = 0 ;
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
		MOBICMean[i] = 0 ;
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
		for (int j = 0; j < NUMBER_OF_NODES; j++)
		{
			MOBICVarianceMatrix[i][j] = 0 ;
		}
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
		MOBICMyMatrix[i] = 0 ;
}

for (int i = 0; i < NUMBER_OF_NODES; i++)
{
		for (int j = 0; j < NUMBER_OF_NODES; j++)
		{
			MOBICClusteringMatrix[i][j] = 0 ;
		}
}
```

```
//Calculate Distance Matrix For Previous Node Locations
```

```
	for ( int i = 0; i < NUMBER_OF_NODES; i++ )
		{
			for ( int j = 0; j < NUMBER_OF_NODES; j++ )
			{
				distancematrixcurrent[i][j] = sqrt((pow(nodexCurrent[i]-
nodexCurrent[j],2)) + (pow(nodeyCurrent[i]-nodeyCurrent[j],2)));
			}
		}



	//Calculate Distance Matrix For Current Node Locations
	for ( int i = 0; i < NUMBER_OF_NODES; i++ )
		{
			for ( int j = 0; j < NUMBER_OF_NODES; j++ )
			{
				distancematrixprevious[i][j] = sqrt((pow(nodexPrevious[i]-
nodexPrevious[j],2)) + (pow(nodeyPrevious[i]-nodeyPrevious[j],2)));
			}
		}


	//Calculate MOBIC Matrix
	for ( int i = 0; i < NUMBER_OF_NODES; i++ )
		{
			for ( int j = 0; j < NUMBER_OF_NODES; j++ )
			{
				if (i == j)
				{
					distancematrixMOBIC[i][j] = 0 ;
				}

				else
				{
				distancematrixMOBIC[i][j] = distancematrixcurrent[i][j] /
distancematrixprevious[i][j] ;
				}
			}
		}

	//***Variance calculation (My) for each row of MOBIC matrix begins here

	//The next two loops compute the mean for each row of the matrix

	for ( int i = 0; i < NUMBER_OF_NODES; i++ )
		{
```

```
                        for ( int j = 0; j < NUMBER_OF_NODES; j++ )
                        {
                                MOBICArray[i] = MOBICArray[i] +
distancematrixMOBIC[i][j] ;
                        }
                }

        //cout << "\n\n\nThis is the mobic section" ;
        for ( int i = 0; i < NUMBER_OF_NODES; i++ )
                {
                        MOBICMean[i] = MOBICArray[i] / (NUMBER_OF_NODES - 1)   ;
                }

        //Calculate (Value - Mean)^2 For each node
        for ( int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for (int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        MOBICVarianceMatrix[i][j] = pow(distancematrixMOBIC[i][j] -
MOBICMean[i],2) ;
                }
        }

        //Final calculations for My of each node
        for ( int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for (int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        MOBICMyMatrix[i] = MOBICMyMatrix[i] +
MOBICVarianceMatrix[i][j] ;
                }
        }

        for ( int i = 0; i < NUMBER_OF_NODES; i++)
        {
                MOBICMyMatrix[i] = MOBICMyMatrix[i] / (NUMBER_OF_NODES - 1);
        }

        /*for ( int i = 0; i < NUMBER_OF_NODES; i++)
        {
                cout << MOBICMyMatrix[i] << endl ;
        }*/


//Clustering Begins Here

for (int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        sortedMOBICArray[i] = MOBICMyMatrix[i] ;
```

```
                        originalPositionArrayMOBIC[i] = i ;
                }

for (int i = 0; i < NUMBER_OF_NODES ; i++)
        {
                for(int j=0; j < NUMBER_OF_NODES; j++)

                {
                        if(sortedMOBICArray[i] > sortedMOBICArray[j])
                        {
                                temp = sortedMOBICArray[i] ;
                                sortedMOBICArray[i] = sortedMOBICArray[j] ;
                                sortedMOBICArray[j] = temp ;
                                temp1 = originalPositionArrayMOBIC[i] ;
                                originalPositionArrayMOBIC[i] =
originalPositionArrayMOBIC[j] ;
                                originalPositionArrayMOBIC[j] = temp1 ;
                        }
                }
        }

        for (int i=0; i < NUMBER_OF_NODES; i++)
                {
                        unsigned int p = 0;
                        for (;p < savedvaluesforMOBIC.size(); ++p )
                        {
                                if ( (originalPositionArrayMOBIC[i] ==
savedvaluesforMOBIC[p]) )
                                {
                                        break;
                                }
                        }

                        if (p == savedvaluesforMOBIC.size())
                        {
                                for (int j=0; j < NUMBER_OF_NODES; j++)
                                {
                                        if (edgematrix[originalPositionArrayMOBIC[i]][j] !=
999 && originalPositionArrayMOBIC[i] != j)
                                        {

        MOBICClusteringMatrix[originalPositionArrayMOBIC[i]][j] =
edgematrix[originalPositionArrayMOBIC[i]][j];
                                        savedvaluesforMOBIC.push_back(j) ;
                                        }
                                }

                        }
                }
```

```
/*cout << "\n\n\n" ;
cout << "MOBIC CLUSTERING.....\n\n" ;
for (int i=0; i < NUMBER_OF_NODES; ++i)
                        {
                                cout << endl ;
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                        {
                                cout << MOBICClusteringMatrix[i][j] << " " ;
                        }
}*/


//Betweenness Test Starts Here
                double MOBICbtwtemp = 0 ;
                double MOBICbetweenness[NUMBER_OF_NODES];
                double
MOBICbtwshortestpath[NUMBER_OF_NODES][NUMBER_OF_NODES] ;
                double MOBICNumber_Of_Nodes_That_Exceed_Capacity = 0 ;
                double MOBICpercentExceedingCapacity = 0;

                for(int i = 0; i < NUMBER_OF_NODES; i++)
                {
                        MOBICbetweenness[i] = 0;
                }

        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        MOBICbtwshortestpath[i][j] = MOBICClusteringMatrix[i][j];
                }
        }

        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        if (MOBICbtwshortestpath[i][j] == 0)
                        MOBICbtwshortestpath[i][j] = 999;
                }
        }


        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        if (MOBICbtwshortestpath[i][j] == 1)
```

```
                        MOBICbtwshortestpath[j][i] = 1;
                }
        }


        for (int k = 0; k < NUMBER_OF_NODES; k++)
                {
                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        MOBICbtwshortestpath[i][j] =
min(MOBICbtwshortestpath[i][j], MOBICbtwshortestpath[i][k] + MOBICbtwshortestpath[k][j]) ;
                                }
                        }
                }


        /*cout << "\n\n\n" ;
        cout << "Shortest Path.....\n\n" ;
        for (int i=0; i < NUMBER_OF_NODES; ++i)
                        {
                                cout << endl ;
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                        {
                                cout << MOBICbtwshortestpath[i][j] << " " ;
                        }
                        }*/


        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        if (MOBICbtwshortestpath[i][j] >= 2)
                                        {
                                                for (int k = 0; k < NUMBER_OF_NODES;
k++)
                                                {
                                                        if ( i != j && i != k && j != k &&
((MOBICbtwshortestpath[i][k] + MOBICbtwshortestpath[k][j]) == MOBICbtwshortestpath[i][j]))
                                                        {
                                                                MOBICbetweenness[k]++;
                                                        }
                                                }
                                        }
                }
                }
```

```
		for (int i = 0; i < NUMBER_OF_NODES; i++)
		{
			MOBICbetweenness[i] = (MOBICbetweenness[i] / 2) ;
		}


		for (int i = 0; i < NUMBER_OF_NODES; i++)
		{
			if (nodeCapacity < MOBICbetweenness[i])
			{
				MOBICNumber_Of_Nodes_That_Exceed_Capacity++ ;
			}
		}

		MOBICpercentExceedingCapacity =
MOBICNumber_Of_Nodes_That_Exceed_Capacity / NUMBER_OF_NODES ;

		/*cout << "\n\nThe percentage of nodes that exceed capacity is " <<
MOBICpercentExceedingCapacity ;
		cout << endl << endl ;


		cout << endl << endl ;

		cout << "\n\nThe betweenness of each node in the network is...\n\n" ;

		for (int i = 0; i < NUMBER_OF_NODES; i++)
		{
			cout << "Node " << i + 1 << " " << MOBICbetweenness[i] ;
			cout << "\n"  ;
		} */

		return MOBICpercentExceedingCapacity ;



	return   0;

}
```

NetworkDiameter.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include "NetGenDefs.h"
using namespace std;

double ND(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES])
{

double maxpath = 0 ;
maxpath = 0;

double shortestpath[NUMBER_OF_NODES][NUMBER_OF_NODES] = {0} ;

        for(int i = 0; i < NUMBER_OF_NODES; i++)
        {
                for(int j = 0; j < NUMBER_OF_NODES; j++)
                {
                        shortestpath[i][j] = edgematrix[i][j];
                }
        }

                for (int k = 0; k < NUMBER_OF_NODES; k++)
                {
                        for (int i = 0; i < NUMBER_OF_NODES; i++)
                        {
                                for (int j = 0; j < NUMBER_OF_NODES; j++)
                                {
                                        shortestpath[i][j] = min(shortestpath[i][j],
shortestpath[i][k] + shortestpath[k][j]) ;
                                }
                        }
                }

                for (int i=0; i < nn; i++)
                {
                        for (int j=0; j < nn; j++)
                        {
                                if(shortestpath[i][j] < 999 && shortestpath[i][j] > 0 && i!=j
&& shortestpath[i][j] > maxpath )
                                {
                                        maxpath = shortestpath[i][j];
                                }
                        }
                }
                return maxpath ; } }
```

ODComplexity.cpp

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>
#include <ctime>
#include <vector>
#include <algorithm>
#include "NetGenDefs.h"

using namespace std;


typedef vector<double>    integerArray;
typedef integerArray::iterator arrayPtr;
typedef vector<double>    integerArray2;


typedef vector<double> multiDimensionalVector1;
typedef vector<multiDimensionalVector1> multiDimensionalVector2;

integerArray ODCVector ;
integerArray2 ODCNumeratorVector ;
multiDimensionalVector2 ODCArray ;
double ODCDenom = 0 ;
int ODCIncrementor ;
double ODCNumElement ;
double insideincrementor ;
double ODCNumeratorTotal ;


double ODComplexity ;

double ODC(int nn, int edgematrix[NUMBER_OF_NODES][NUMBER_OF_NODES], int
degreeofeachnode[NUMBER_OF_NODES])
{

        //This section clears all the variables within the function
        ODCDenom = 0 ;
        ODCIncrementor = 0 ;
        ODCNumElement = 0 ;
        insideincrementor = 0 ;
        ODCNumeratorTotal = 0 ;
        ODComplexity = 0 ;



        //This loop sets the degree for each node equal to zero
```

```
for (int i=0; i < NUMBER_OF_NODES; i++)
	{
		for (int j=0; j < NUMBER_OF_NODES; j++)
		{
			if(edgematrix[i][j] == 1 && i != j && edgematrix[i][j] != 999)
			{
				degreeofeachnode[i] = 0 ;
			}
		}
	}

//This loop determines the degree for each node
for (int i=0; i < NUMBER_OF_NODES; i++)
	{
		for (int j=0; j < NUMBER_OF_NODES; j++)
		{
			if(edgematrix[i][j] == 1 && i != j && edgematrix[i][j] != 999)
			{
				degreeofeachnode[i]++ ;
			}
		}
	}

ODCVector.clear() ;
ODCArray.clear() ;
ODCNumeratorVector.clear();

for (int i = 0; i < NUMBER_OF_NODES; i++)
	{
		unsigned int p = 0;
		for (; p < ODCVector.size(); p++)
		{
			if ( degreeofeachnode[i] == ODCVector[p] )
			{
				break ;
			}
		}

		if (p == ODCVector.size())
		{
			ODCVector.push_back(degreeofeachnode[i]) ;
		}
	}
size_t ArraySize = ODCVector.size();
multiDimensionalVector1 initializer;
for (unsigned int i = 0; i < ArraySize; ++i)
{
	initializer.push_back(0.0L);
}
```

```
	for (unsigned int i = 0; i < ArraySize; ++i)
	{
		ODCArray.push_back(initializer);
	}

	for (unsigned int i = 0; i < ArraySize; i++)
	{
		for (unsigned int j = 0; j < ArraySize; j++)
		{
			ODCArray[i][j] = 0;
		}
	}

	sort (ODCVector.begin(), ODCVector.end()) ;

	for(unsigned int i = 0; i < ODCVector.size(); i++)
	{
		for(int j = 0; j < NUMBER_OF_NODES; j++)
		{
			if (degreeofeachnode[j] == ODCVector[i])

			for(unsigned int k = 0; k < ODCVector.size(); k++)
			{
				for(int l = 0; l < NUMBER_OF_NODES; l++)
				{
					if(l != j && degreeofeachnode[j] == ODCVector[i]
&& (degreeofeachnode[l] == ODCVector[k]) && (edgematrix[l][j] == 1) && (edgematrix[j][l] ==
1))
					{
						ODCArray[i][k] = ODCArray[i][k] + 1  ;
					}
				}
			}
		}
	}

	for (unsigned int i = 0; i < ODCVector.size(); i++)
	{
		for (unsigned int j = 0; j < ODCVector.size(); j++)
		{
			if(i != j)
			{
				ODCArray[i][j] = ODCArray[i][j] + ODCArray[j][i] ;
			}
		}
	}
```

```
for (unsigned int i = 0; i < ODCVector.size(); i++)
{
	for (unsigned int j = 0; j < ODCVector.size(); j++)
	{
		if(i != j)
		{
			ODCArray[j][i] = ODCArray[i][j] ;
		}
	}
}

for (unsigned int i = 0; i < ODCVector.size(); i++)
{
	for (unsigned int j = 0; j < ODCVector.size(); j++)
	{
		ODCArray[i][j] = ODCArray[i][j] / 2 ;
	}
}

ODCDenom = 0 ;

// This calculates the demoninator for the ODC metric
for (unsigned int i = 0; i < ODCVector.size(); i++)
{
	for (unsigned int j = 0; j < ODCVector.size(); j++)
	{
		ODCDenom = ODCDenom + ODCArray[i][j] ;
	}
}

ODCIncrementor = -1 ;

while (ODCIncrementor != ODCVector.size() )
{
	ODCIncrementor++ ;

	if (ODCIncrementor != 0)
	{
		ODCNumeratorVector.push_back(ODCNumElement) ;
	}

	ODCNumElement = 0 ;

	for (unsigned int i = 0; i < ODCVector.size(); i++)
	{
		if ((i + ODCIncrementor) < ODCVector.size())
		{
			ODCNumElement = ODCNumElement + ODCArray[i][i +
ODCIncrementor] ;
```

```
                        }
                }
        }

        /*cout << "\n\nODCNUM Vector 1 Printed Below\n\n" ;

        for (unsigned int i = 0; i < ODCVector.size(); i++)
        {
                cout << ODCNumeratorVector[i] << endl ;
        }*/



        for (unsigned int i = 0; i < ODCVector.size(); i++)
        {
                ODCNumeratorVector[i] = ODCNumeratorVector[i] / ODCDenom ;
                if (ODCNumeratorVector[i] != 0)
                {
                ODCNumeratorVector[i] = (ODCNumeratorVector[i] *
log(ODCNumeratorVector[i])) ;
                }
        }

        ODCNumeratorTotal = 0 ;

        for (unsigned int i = 0; i < ODCVector.size(); i++)
        {
                ODCNumeratorTotal = ODCNumeratorTotal + ODCNumeratorVector[i] ;
        }

        ODComplexity = - (ODCNumeratorTotal) ;


        //cout << "\n\nThe offdiagonal complexity is " << ODComplexity ;


        /*cout << "\n\nODCNUM Vector Printed Below\n\n" ;

        for (unsigned int i = 0; i < ODCVector.size(); i++)
        {
                cout << ODCNumeratorVector[i] << endl ;
        }

        cout << "\n\nThis is the ODC Part\n\n" ;


                for (unsigned int q = 0; q < ODCVector.size() ; q++)
                {
                        cout << ODCVector[q] << endl ;
```

```
                }


        cout << "\n\nThis is the next part of ODC\n\n" ;

        for (unsigned int i = 0; i < ODCVector.size(); i++)
                        {
                                for (unsigned int j = 0; j < ODCVector.size(); j++)
                                {
                                        cout << " " << ODCArray[i][j] ;
                                }
                                        cout << "\n"  ;
                        }
*/


return   ODComplexity;

}
```

Appendix B

Sample Data File Output

Number Of Nodes	Node Capacity	Generator Seed	Square Length Of Field	Network Diameter	Offdiagonal Complexity	Highest Degree Congestion
150	193	5	100	25.3	1.20754	0.274
150	193	20	100	24.8	1.14537	0.253333
150	193	1	100	22.5	1.17753	0.175333
150	150	10	100	23.5	1.16442	0.201333
150	100	1	100	21.5	1.17494	0.3
150	60	20	100	22.5	1.15866	0.357333
150	60	30	100	23.1	1.18808	0.301333
150	60	5	100	22.4	1.15074	0.292
150	100	20	100	21.4	1.16806	0.332
150	100	30	100	22.4	1.16603	0.228

It should be noted that this represents a very small subset of the actual data that was collected from the simulation runs. A selected number of columns and rows were selected for formatting purposes.

Bibliography

[1] P. Basu, N. Khan, and T.D.C. Little. A Mobility Based Metric for Clustering in Mobile Ad Hoc Networks. In *Proceedings of IEEE International Conference on Distributed Computing Systems Workshops*, Mesa, Arizona, 2001.

[2] J.C. Claussen. Characterization of networks by the Offdiagonal Complexity. *Physica A: Statistical Mechanics and its Applications*, 375(1): 365-373, 2007.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. (1990). *Introduction to Algorithms*, first edition, MIT Press and McGraw-Hill.

[4] M. Gerla and J.T.C. Tsai. Multicluster, mobile, multimedia radio network. *Wireless Networks*, 1(3) : 255-265, 1995.

[5] Z.J. Hass and S. Tabrizi. On some challenges and design choice in ad hoc communication. In *Proceeding of IEEE MILCOM'98, 1998*.

[6] C.R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 15(7): 1265-1275, 1997.

[7] A.K. Parekh. Selecting routers in ad hoc wireless networks. *Proceeding of the IEEE International Telecommunication Symposium*, 1994.

[8] D.J. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393: 440-442, 1998.