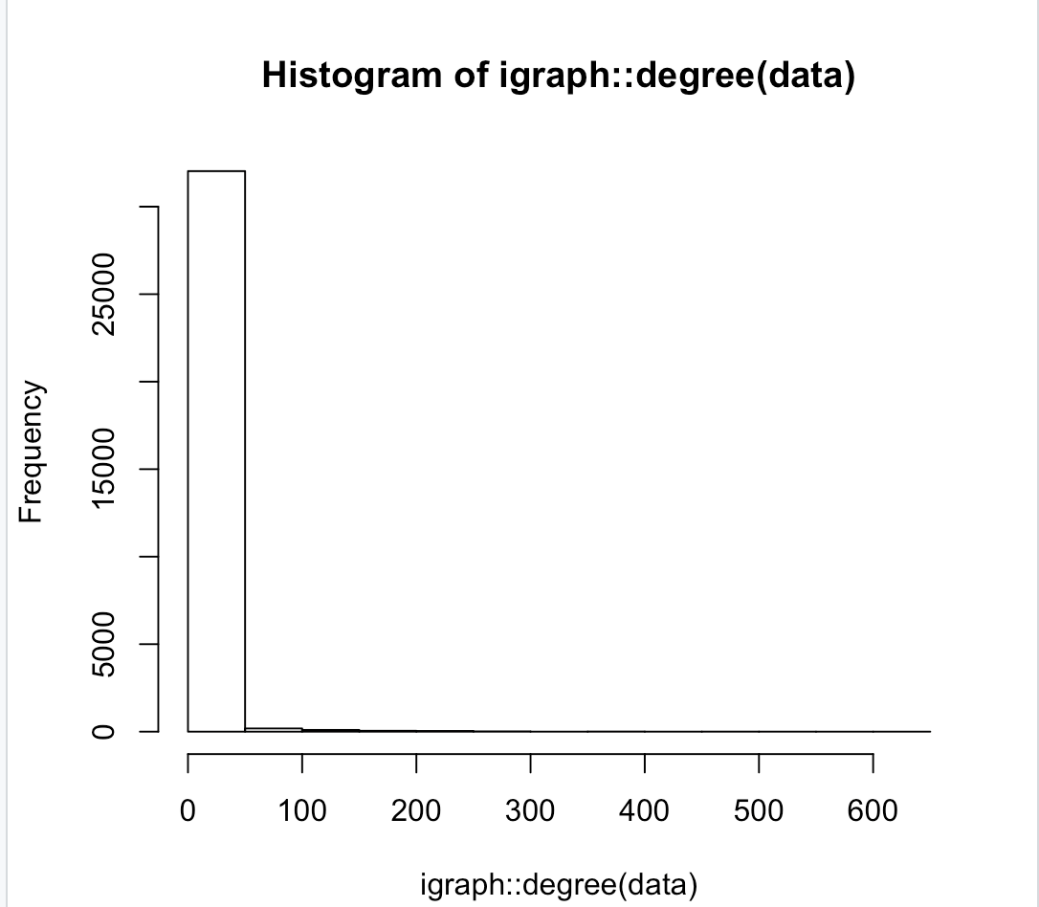
**CS3907: Big Data and Analytics - Project #1 Write Up**

As a team, we began the project by first creating a repository on Github and uploading the data from blackboard. We placed all these files (the data, our own R file, that interacts with the data, and write-up file) into our shared repository. Our next step was to read the graph through the R function “read.graph” and begin our analysis. We then looked at 10 different R functions in order to better grasp what the data was showing us. From looking at the graphs connectivity, its diameter, and neighboring nodes, we were able to figure out different characteristics of the data. Finally, through our research, we were then able to determine who the central person(s) is, the longest path, the largest clique, ego, and power centrality.

**# 3**

After simplification, we saved the simplified graph as an Rdata file. Now in working with the graph and exploring methods, all we have to do is refer to data.Rdata and the simplified version is ready to go. As per number 3 of the project specification, we explored some of the methods that were demoed in the lecture slides. This included methods like centr\_betw, edge\_density, shortest\_paths, and plotting a histogram of the degrees. These methods provided us with insight into the data, and we also discovered a potential error in the lecture slides: when using shortest\_paths like the slides shows there is an error, the definition of the method may have changed. Here is an image of the histogram generated of node degrees

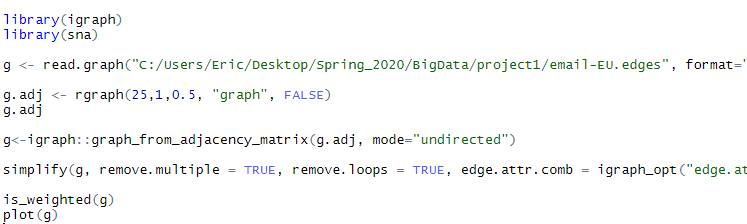


Below is the set of 10 functions we utilized in order to figure out how each of the 25 inputs is connected and how they interacted with one another.

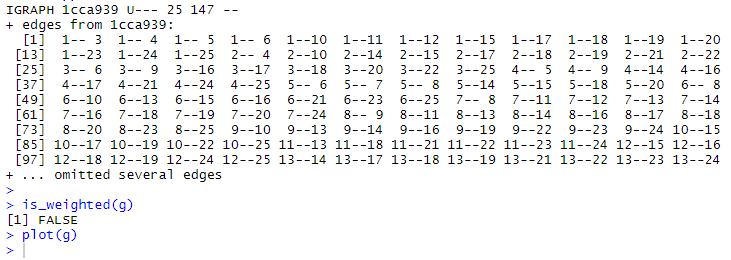
**Function 1:** is\_weighted

This function allows users to determine if an inputted graph, either directed or undirected, has an edge attribute, which identifies if it is weighted or not. We first simplified the graph through the simplify function. This function removes multiples and loops. The outputted results are pictured below, showing that our graph was not weighted.

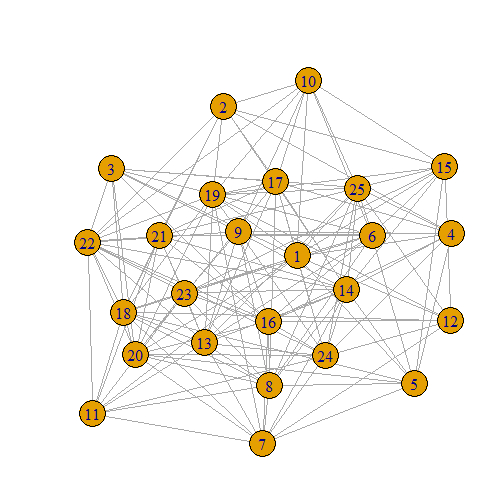
Code:



Result:



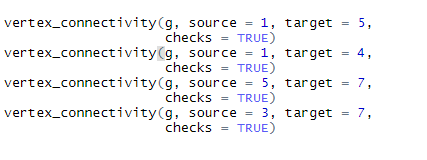
Graph:



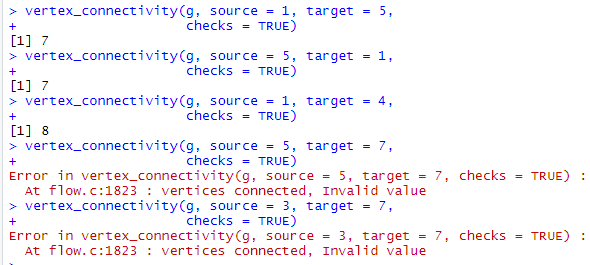
**Function 2:** vertex\_connectivity

This function analyzes the connection between a source and target node in a directed graph, to identify how many nodes need to be removed in order for there to be no direct paths between the source and target node. We tested this function with several nodes. The output showed that the connectivity of nodes one and five is seven. This means that seven nodes would need to be removed in order for there to be no direct paths from one to five.

Code:



Result:



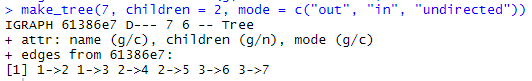
**Function 3:** make\_tree

Given a number of vertexes, this function will produce a tree and print relevant associations.

Code:



Result:



**Function 4:** Isomorphic

This function is also self-explanatory. It determines whether two groups correspond to each other in a manner that is one-to-one (whether they are isomorphic). The output was True.

**Code:**



**Result:**



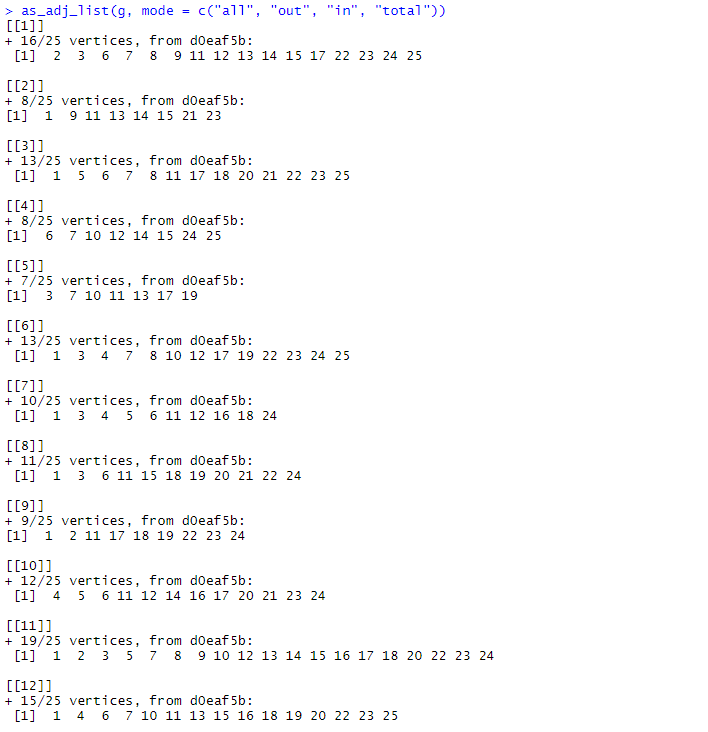
**Function 5:** as\_adj\_list

This function creates an adjacency list from the inputted graph. The result returns all of the neighbors of each node in the graph. This accounts for outgoing edges and incoming edges since this is a directed graph.

Code:



Result:



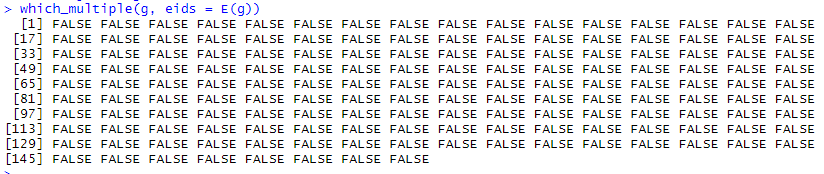
**Function 6:** which\_multiple

This function returns if there are any duplicate connections within the graph. The result was false for every connection, meaning that there were no connections, connected more than once.

Code:



Result:



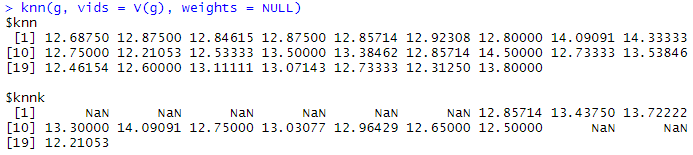
**Function 7:** knn

The function, knn, returns the average nearest neighbor degree.

Code:



Result:



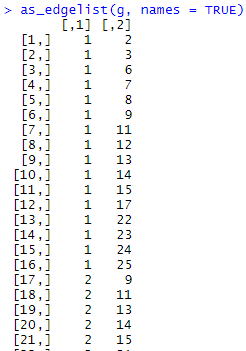
**Function 8:** as\_edgelist

This function outputs the neighbors of each node in a list format. This is similar to function five.

Code:



Result:



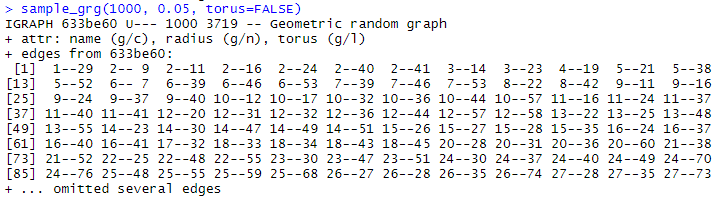
**Function 9:** sample\_grg

This function generates a random graph using points of a unit square.

Code:



Result:



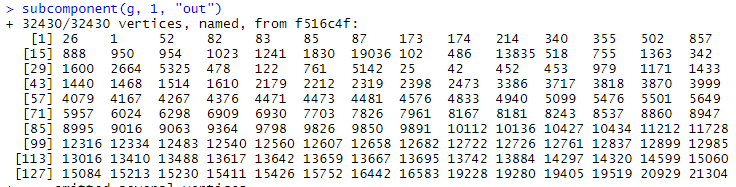
**Function 10:** subcomponent

This function takes in a vertex and returns an adjacency list of all vertexes that are reachable from that point. It can also be reversed. This is less flashy for a connected graph.

Code:



Result:



For getting the answers to who the central person(s) is, the longest path, the largest clique, ego, and power centrality, we looked to five different R functions that will be described below, with an explanation, and answer. These functions were included in the packages that were installed.

**Central Persons:**

**Longest Path:**

The function used to extract the longest path from the graph by using the path that has the largest diameter. The result outputted is 9, highlighting that the longest path goes through 9 nodes.

Code:



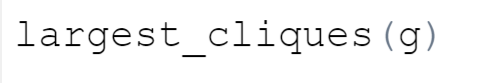
Result:



**Largest Clique:**

A clique is a completely connected subgraph. In this, every node is connected to every other node. These cliques have a strong connection, as every node is connected.

Code:



Result:

[[1]]

+ 12/32431 vertices, from 9387f47:

[1] 951 215 175 84 83 88 356 503 454 280 96 889

[[2]]

+ 12/32431 vertices, from 9387f47:

[1] 951 215 175 84 83 88 356 503 454 280 96 104

[[3]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 6713 326 30 503 84 224 1024 12888

[11] 174 682

[[4]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 224 326 84 503 30 1024 682 174

[11] 1023 12888

[[5]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 224 326 84 503 30 1024 83 1023

[11] 12888 8014

[[6]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 224 326 84 503 30 1024 83 1023

[11] 12888 174

[[7]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 560 682 12888 30 174 326 503

[11] 1023 1024

[[8]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 682 174 326 1023 1024

[11] 30 12888

[[9]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 326 1024 1023 30

[11] 12888 8014

[[10]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 326 1024 1023 30

[11] 12888 174

[[11]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 1024 1023 8014

[11] 30 12888

[[12]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 1024 1023 174 356 280

[[13]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 1024 1023 174

[11] 30 12888

[[14]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 1024 1023 104 280 356

[[15]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 1024 96 104 280 356

[[16]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 454 356 280 104 96

[[17]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 454 356 280 53 174

[[18]]

+ 12/32431 vertices, from 9387f47:

[1] 103 175 215 84 503 83 88 454 356 27 53 174

[[19]]

+ 12/32431 vertices, from 9387f47:

[1] 2 175 84 83 224 326 503 30 1024 1023

[11] 12888 8014

[[20]]

+ 12/32431 vertices, from 9387f47:

[1] 2 175 84 83 224 326 503 30 1024 1023

[11] 12888 174

[[21]]

+ 12/32431 vertices, from 9387f47:

[1] 2 175 84 83 215 503 326 30 1024 1023

[11] 12888 8014

[[22]]

+ 12/32431 vertices, from 9387f47:

[1] 2 175 84 83 215 503 326 30 1024 1023

[11] 12888 174

[[23]]

+ 12/32431 vertices, from 9387f47:

[1] 2 175 84 83 215 503 88 1024 30 1023

[11] 12888 8014

[[24]]

+ 12/32431 vertices, from 9387f47:

[1] 2 175 84 83 215 503 88 1024 30 1023

[11] 12888 174

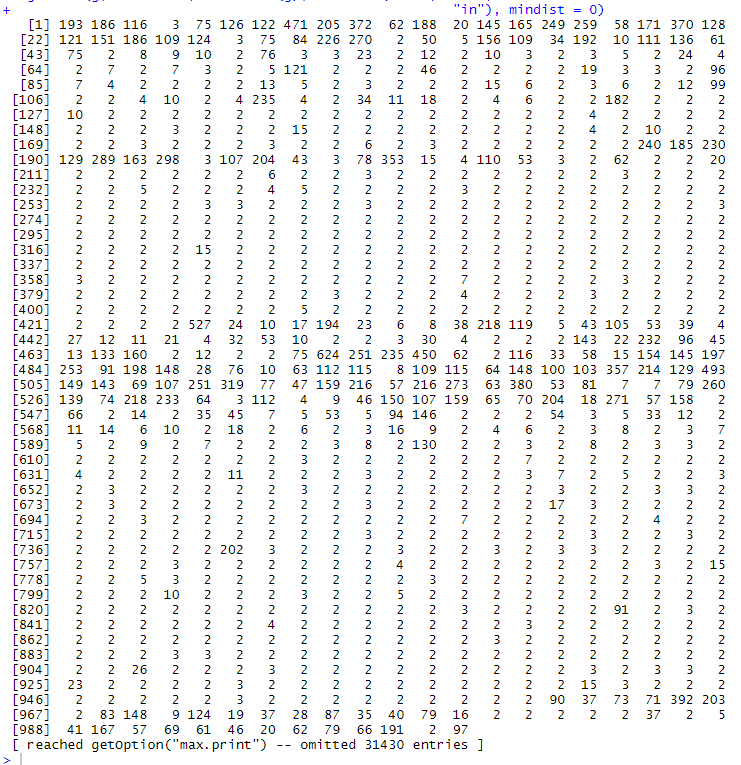
**Ego:**

Ego focuses upon seeing which nodes are directly and indirectly connected to one another. For example, if a node has a direct link to one node or if a node goes through another node to get to the target node. Below is the result of all the connections of nodes to one another.

Code:



Result:



**Power Centrality:**

Code:

Result: