CLASS PROJECT 1 – GRAPH ANALYTICS

CSCI 6444: INTRODUCTION TO BIG DATA & ANALYTICS

PROFESSOR: STEPHEN KAISLER

GROUP 12

NAME: RAJITH RAVIKUMAR

GWID: G45692898

NAME: SRAVANI BRAHAMMA ROUTHU

GWID: G28059824

1. Dataset Description

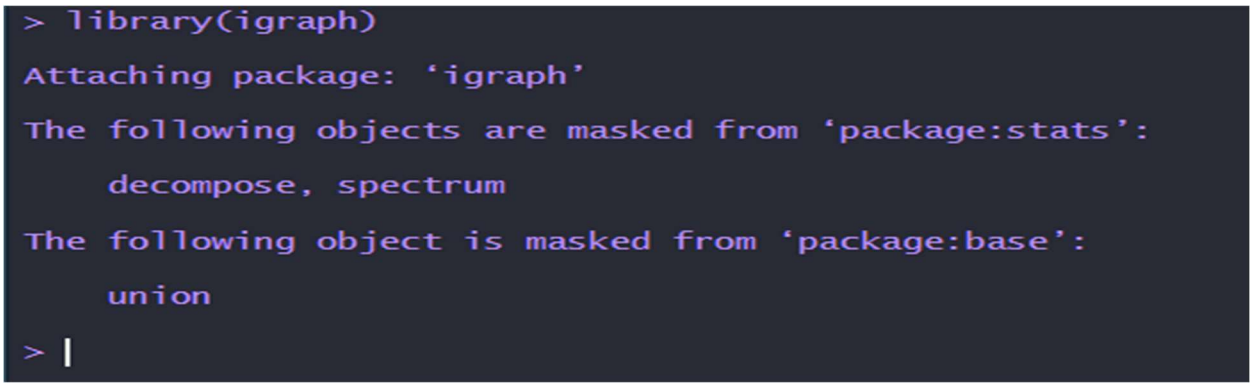
The dataset is a graph of links of opinions from the SOC-E website, also known as a trust network of the Epinions website. Users could express trust in each other, forming a "web of trust" to determine which reviews were shown to each user.

This dataset has anonymized names replaced by numbers. It's medium-sized with 10 million nodes and edges indicating trust relationships.

2. Installing the igraph package from one of the CRAN mirrors and determining how to create a graph and plot. Show the plot in your report.

Based on the problem description, the initial step is to install the igraph package from one of the CRAN servers. To utilize it for our project, type "install.packages(igraph)" in the terminal, followed by "library(igraph)". This is how the outputs will appear.

Declare it as library:



- **Import the specified data set**

```
directory<-"C:/Users/rajit/Downloads"
soc_opinions<- file.path(directory,"soc-Epinions1_adj.tsv")
opinions<-read.table(file = soc_opinions)
optab<-as.matrix(opinions)
n <- 811480
```

➢ Sets the working directory:
  - The first line **directory<-"C:/Users/rajit/Downloads"** sets the working directory to **"C:/Users/rajit/Downloads"**. This tells R where to look for the files it needs.

➢ Defines a file path:
  - The second line **soc_opinions<- file.path(directory, "soc-Epinions1_adj.tsv")** creates a variable called soc_opinions that stores the path to the file "**soc-Epinions1_adj.tsv**". The file is assumed to be in the working directory that was set in the first line.

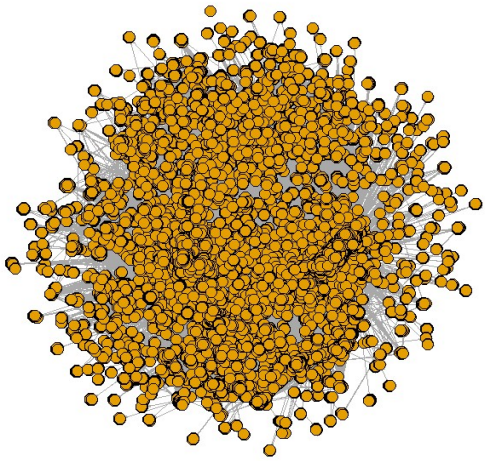
➢ Reads the data:
  - The third line **opinions<-read.table(file = soc_opinions)** reads the data from the file specified by soc_opinions and stores it in a variable called opinions. The read.table function is used to read tabular data from text files.

a. Converts the data to a matrix:
  - The fourth **line optab<-as.matrix(opinions)** converts the data in opinions to a matrix and stores it in a variable called optab. Matrices are a fundamental data structure in R that can be used to store and manipulate numerical data.

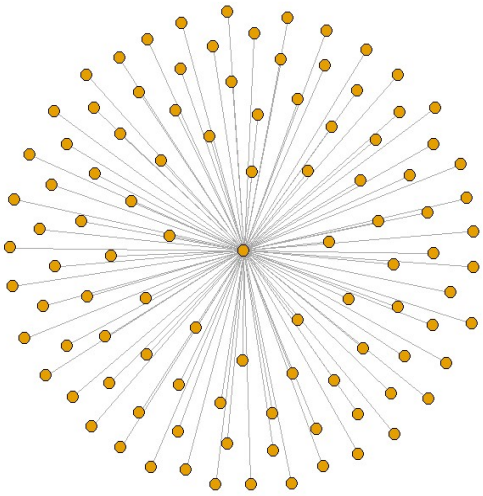
- **Determine how to create a graph and plot. Show the plot in your report.**
- The code defines a function called **directed_graph** that takes two arguments, a and b. These arguments specify the **starting and ending row numbers** of a sub-matrix to be extracted from a larger matrix called **optab**.
- Creates a data frame called **relations** with two columns: "**from**" and "**to**".
- These **columns represent the edges** in the graph, where each row represents a directed edge from the node in the "from" column to the node in the "to" column.
- Uses the **graph.data.frame** function from the **igraph** package **to create a directed graph** object from the relations data frame.
- We use the directed_graph function to create several directed graphs from different sub-matrices of optab. It then uses the plot function from the igraph package to plot these graphs.

```
#Function to create a graph from row 'a' to row 'b'
directed_graph<-function(a,b)
  {
  relations<-data.frame(from = optab[a:b, 1], to = optab[a:b, 2])
  g<-graph.data.frame(relations, directed = TRUE)
  return(g)
  }

#Plotting the first 100 rows
g_first100 <- directed_graph(1,100)
plot(g_first100,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)

#Plotting the first 1000 rows
g_first1000 <- directed_graph(1,1000)
plot(g_first1000,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)

#Plotting the first 10000 rows
g_first10000 <- directed_graph(1,10000)
plot(g_first10000,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)

#Plotting the last 100 rows
g_last100 <- directed_graph(n-100,n)
plot(g_last100,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)

#Plotting the last 1000 rows
g_last1000 <- directed_graph(n-1000,n)
plot(g_last1000,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)

#Plotting the last 10000 rows
g_last10000 <- directed_graph(n-10000,n)
plot(g_last10000,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)

#Plotting all the rows
g <- directed_graph(1,n)
plot(g,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)
```

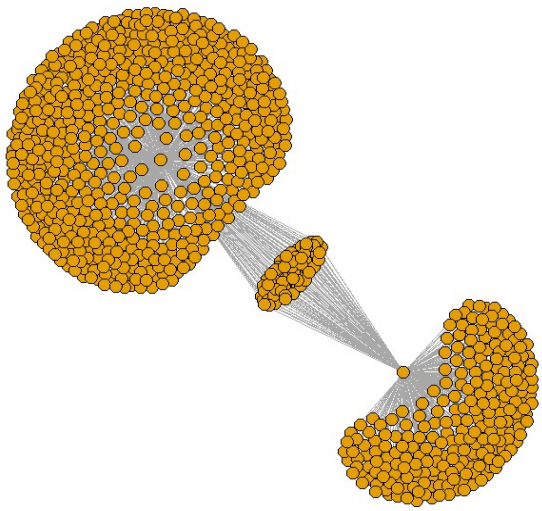
The initial plot generated by running plot(g) was excessively complicated and took a long time to execute. This graph has no vertex labels.



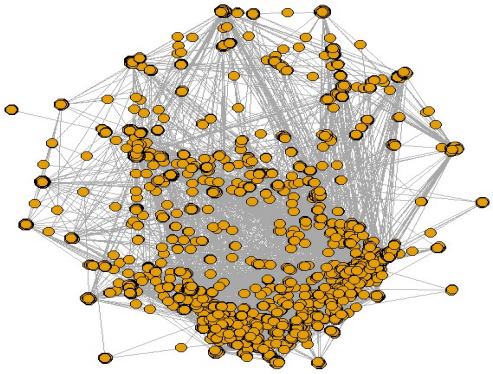
For enhanced clarity in graph representation and to facilitate comprehension, we have amalgamated the plots of the initial 100, 1000, and 10,000 nodes with those of the final 100, 1000, and 10,000 nodes. These plots are represented as follows:



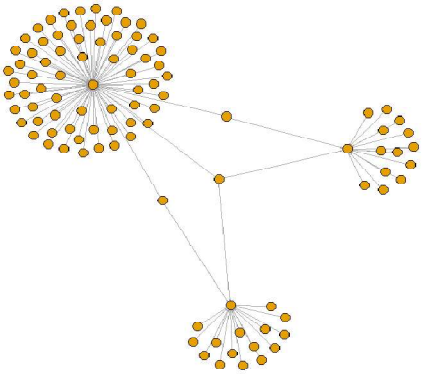
Plotting the first 100 rows



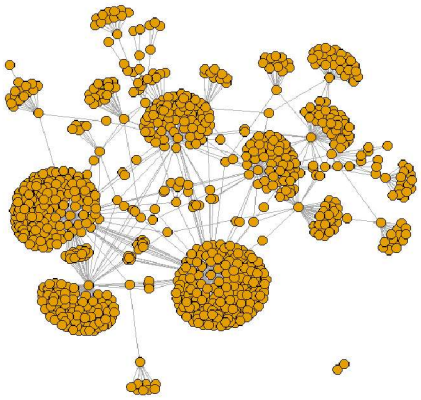
Plotting the first 1000 rows



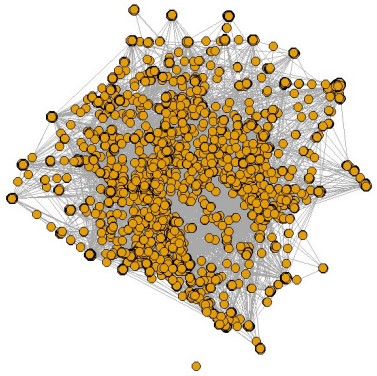
Plotting the first 10000 rows



Plotting the last 100 rows



Plotting the last 1000 rows



Plotting the last 10000 rows

3. Applying the functions in the Introduction to Graph Analytics document on Blackboard on the graph generated from the data set.

- **V(g) - Vertices of a Graph**

```
> #Vertices of the graph
> V(g)
+ 75879/75879 vertices, named, from 0090265:
   [1] 3    4    115  150  182  226  282  337  371  448  559  670  780  826  875  891  897  925  1002 1111 1112 1122 1223 1334 1445 1556
  [27] 1667 1778 1834 1889 2000 2001 2080 2111 2149 2222 2223 2242 2334 2346 2434 2445 2501 2534 2556 2601 2623 2667 2727 2778 2811 2889
  [53] 3000 3041 3089 3110 3111 3145 3222 3305 3333 3334 3379 3410 3445 3534 3556 3574 3667 3778 3889 3989 4000 4111 4158 4222 4333 4341
  [79] 4444 4445 4556 4563 4667 4778 4889 4978 5000 5111 5222 5289 5333 5367 5385 5444 5456 5489 5554 5555 5556 5633 5666 5667 5744 5766
 [105] 5777 5778 5804 5888 5911 5922 5999 6000 6110 6155 6221 6244 6266 6332 6346 6355 6443 6554 6665 6666 6711 6777 6789 6855 6888 6922
 [131] 6999 7110 7221 7266 7332 7443 7444 7554 7665 7776 7777 7800 7888 7965 7998 8109 8220 8331 8442 8553 8664 8691 8705 8775 8789 8886
 [157] 8887 8998 9010 9072 9109 9220 9331 9442 9553 9646 9664 9775 9831 9885 9886 9985 9997 9998 10109 10175 10220 10236 10331 10442 10553 10664
 [183] 10720 10775 10886 10997 11109 11110 11221 11332 11443 11479 11488 11554 11555 11578 11588 11665 11688 11776 11777 11866 11887 11943 11998 12109 12116 12187
 [209] 12220 12221 12321 12332 12388 12443 12552 12554 12625 12665 12776 12788 12799 12887 12998 13065 13109 13143 13220 13276 13309 13331 13332 13443 13448 13554
 [235] 13643 13665 13688 13776 13887 13954 13998 14109 14110 14154 14176 14187 14198 14209 14220 14276 14298 14331 14442 14443 14554 14665 14710 14776 14887 14998
+ ... omitted several vertices
>
```

The function V(g) retrieves the vertices of a graph, indicating that in graph g, we have a total of 75879 vertices. Notably, due to anonymization measures, names have been substituted with numerical identifiers within this dataset, hence the labels for the nodes reflect these numerical replacements.

- **E(g) - Edges of a Graph**

```
> #Edges of the graph
> E(g)
+ 811480/811480 edges from 0090265 (vertex names):
  [1] 3    ->1 4    ->1 115  ->1 150  ->1 182  ->1 226  ->1 282  ->1 337  ->1 371  ->1 448  ->1 559  ->1 670  ->1 780  ->1 826  ->1 875  ->1 891  ->1 897  ->1
 [18] 925  ->1 1002 ->1 1111 ->1 1112 ->1 1122 ->1 1223 ->1 1334 ->1 1445 ->1 1556 ->1 1667 ->1 1778 ->1 1834 ->1 1889 ->1 2000 ->1 2001 ->1 2080 ->1 2111 ->1
 [35] 2149 ->1 2222 ->1 2223 ->1 2242 ->1 2334 ->1 2346 ->1 2434 ->1 2445 ->1 2501 ->1 2534 ->1 2556 ->1 2601 ->1 2623 ->1 2667 ->1 2727 ->1 2778 ->1 2811 ->1
 [52] 2889 ->1 3000 ->1 3041 ->1 3089 ->1 3110 ->1 3111 ->1 3145 ->1 3222 ->1 3305 ->1 3333 ->1 3334 ->1 3379 ->1 3410 ->1 3445 ->1 3534 ->1 3556 ->1 3574 ->1
 [69] 3667 ->1 3778 ->1 3889 ->1 3989 ->1 4000 ->1 4111 ->1 4158 ->1 4222 ->1 4333 ->1 4341 ->1 4444 ->1 4445 ->1 4556 ->1 4563 ->1 4667 ->1 4778 ->1 4889 ->1
 [86] 4978 ->1 5000 ->1 5111 ->1 5222 ->1 5289 ->1 5333 ->1 5367 ->1 5385 ->1 5444 ->1 5456 ->1 5489 ->1 5554 ->1 5555 ->1 5556 ->1 5633 ->1 5666 ->1 5667 ->1
[103] 5744 ->1 5766 ->1 5777 ->1 5778 ->1 5804 ->1 5888 ->1 5911 ->1 5922 ->1 5999 ->1 6000 ->1 6110 ->1 6155 ->1 6221 ->1 6244 ->1 6266 ->1 6332 ->1 6346 ->1
[120] 6355 ->1 6443 ->1 6554 ->1 6665 ->1 6666 ->1 6711 ->1 6777 ->1 6789 ->1 6855 ->1 6888 ->1 6922 ->1 6999 ->1 7110 ->1 7221 ->1 7266 ->1 7332 ->1 7443 ->1
[137] 7444 ->1 7554 ->1 7665 ->1 7776 ->1 7777 ->1 7800 ->1 7888 ->1 7965 ->1 7998 ->1 8109 ->1 8220 ->1 8331 ->1 8442 ->1 8553 ->1 8664 ->1 8691 ->1 8705 ->1
[154] 8775 ->1 8789 ->1 8886 ->1 8887 ->1 8998 ->1 9010 ->1 9072 ->1 9109 ->1 9220 ->1 9331 ->1 9442 ->1 9553 ->1 9646 ->1 9664 ->1 9775 ->1 9831 ->1 9885 ->1
+ ... omitted several edges
>
```

Understanding the number of edges in a graph is crucial to obtaining valuable insights from the data. That's where the E(g) function comes in handy. It provides information on the edges of a graph, just like V(g) does with vertices. In our case, the graph g has 811480 edges, which is an essential piece of information for analyzing the dataset.

- **get.adjacency() - Adjacency Matrix**

```
> #Adjacency matrix
> igraph::get.adjacency(g)
75879 x 75879 sparse Matrix of class "dgCMatrix"
  [[ suppressing 73 column names ‘3’, ‘4’, ‘115’ ... ]]
  [[ suppressing 73 column names ‘3’, ‘4’, ‘115’ ... ]]

3   . 1 . . 1 1 . 1 . 1 . . . . . . . . . 1 1 . . . . . . . . 1 . 1 . 1 . 1 . . . 1 1 . 1 1 1 1 1 1 . . . . . . . . . . . . . . . . . . . . . . . . . ......
4   1 . . . . 1 . 1 . 1 . 1 1 . . 1 . . 1 1 . . . . . . . 1 . 1 . . . 1 . 1 1 . . 1 1 1 . 1 1 . 1 1 . . . . . . . . . . . . . . . . . 1 . . . . . . . ......
115 . . . . . . . 1 . 1 1 . . . . 1 . . . 1 . . . . 1 . . . . 1 . . . 1 . 1 . . . . . 1 . . . . . . . . . . 1 . . . . . . 1 . . . . . . . . . . . . 1 ......
150 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
182 1 . . . . . . 1 1 1 . . . . . 1 . . . . . . 1 1 1 . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . . . . 1 . . . . . . . . . 1 . . . . . . ......
226 1 1 . . . . . 1 1 1 . . 1 . . 1 . . 1 1 . . . . . . . 1 . 1 . 1 . 1 . 1 1 . . . . 1 . . 1 1 . 1 . 1 . . 1 . . . . . . . . . . . . . . . . . . . . ......
282 . . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 . 1 . . ......

 ..............................
 ........suppressing 75806 columns and 75866 rows in show(); maybe adjust 'options(max.print= *, width = *)'
 ..............................
  [[ suppressing 73 column names ‘3’, ‘4’, ‘115’ ... ]]

41784 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
41785 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
41786 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
41789 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
41790 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
41792 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
>
```

The function get.adjacency() is utilized to retrieve the adjacency matrix for a given graph. In this instance, we obtain the adjacency matrix for graph g, which is of dimensions 758789 x 758789. This matrix provides insights into the density of the graph and aids in identifying any isolated vertices.

For instance, upon examination, it becomes apparent that node 282 exhibits relatively fewer connections to other nodes compared to node 4. Such observations enable us to leverage this information for subsequent graph manipulation and analysis endeavors.

- **gden() - Graph Density**

```
> #ANALYTIC FUNCTIONS
> #Graph Density
> gden(data.frame(from = optab[,1], to = optab[,2]))
[1] 0.1569413
>
```

To calculate the density of a graph, we use the gden() function that considers the type of graph and makes adjustments accordingly. Our graph's density is 0.1569413, which means that it has fewer edges compared to all the possible edges. Therefore, we can conclude that our graph is sparse.

- **edge_density() – Edge Density**

```
> #Edge Density
> edge_density(g)
[1] 0.000140942
>
```

The density of a graph is determined as the ratio of the number of edges present in the graph to the maximum potential number of edges it could contain, a calculation facilitated by the edge_density() function.

Notably, even when permitting loops within the graph, the density remains consistent, suggesting a probable absence of loops within the graph structure, and thereby indicating it as a simple graph.

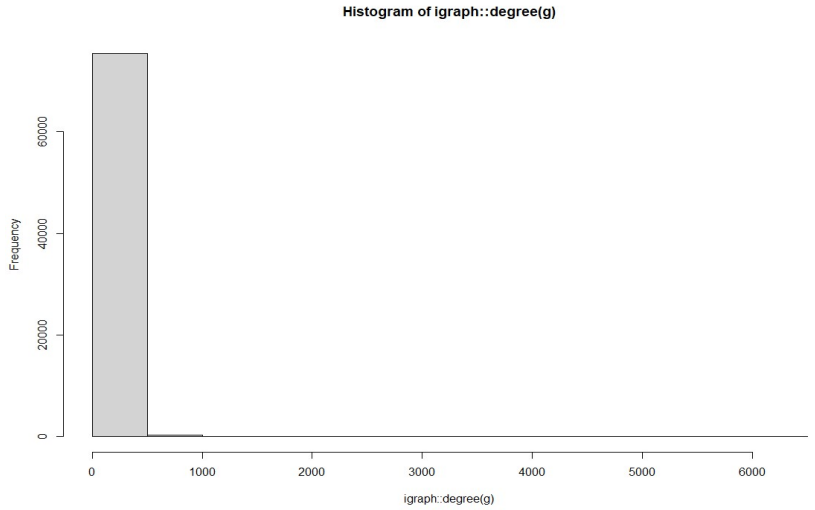
- **degree() - Degree of a Graph**

```
> #degree
> igraph::degree(g)
    3     4   115   150   182   226   282   337   371   448   559   670   780   826   875   891   897   925  1002  1111  1112  1122  1223  1334  1445  1556  1667
  572   690   686    48   636   880   266   924   342   826   618   226   828    38    14   324    40   372   332   448   498    22   362   830   508    82   366
 1778  1834  1889  2000  2001  2080  2111  2149  2222  2223  2242  2334  2346  2434  2445  2501  2534  2556  2601  2623  2667  2727  2778  2811  2889  3000  3041
  432   216   524  2014   210    30   832    26   834   542    30    24   176   252   422  1046   224   602   352   170   510    38  1366    22   396   730    36
 3089  3110  3111  3145  3222  3305  3333  3334  3379  3410  3445  3534  3556  3574  3667  3778  3889  3989  4000  4111  4158  4222  4333  4341  4444  4445  4556
  314    36  1736   814   104    20   732   820   666    12   144   206   362    28   390   262  1296    96  2410   236     8   664   364    20   412   640   980
 4563  4667  4778  4889  4978  5000  5111  5222  5289  5333  5367  5385  5444  5456  5489  5554  5555  5556  5633  5666  5667  5744  5766  5777  5778  5804  5888
   22   332  3198    90   362  1328    86    80   760    36   602    66    52    60   588   112    46   128   310    10   398   352   370     8   432    24    56
 5911  5922  5999  6000  6110  6155  6221  6244  6266  6332  6346  6355  6443  6554  6665  6666  6711  6777  6789  6855  6888  6922  6999  7110  7221  7266  7332
  522   764   612   650   288   232   186   240   226    16    20   650    96   208   222    10    96   140   110   962    16  1134   116    92   684   174   108
 7443  7444  7554  7665  7776  7777  7800  7888  7965  7998  8109  8220  8331  8442  8553  8664  8691  8705  8775  8789  8886  8887  8998  9010  9072  9109  9220
  340   212   142   890   268     6   350     8   192   460    10    22   150    24    24   170    12    40     2   120  6088   132    28   158    32     6    74
 9331  9442  9553  9646  9664  9775  9831  9885  9886  9985  9997  9998 10109 10175 10220 10236 10331 10442 10553 10664 10720 10775 10886 10997 11109 11110 11221
   14   234    84    24   714   126   488    18    60   130   976    42   302    54   154     8   272    86   754     8    36   108   444    90   466   104   382
11332 11443 11479 11488 11554 11555 11578 11588 11665 11688 11776 11777 11866 11887 11943 11998 12109 12116 12187 12220 12221 12321 12332 12388 12443 12552 12554
   48   364    18   428     2   390    10   360    34   512    70   362    64    66   326   112    14    24   172   530   810   254    78   666   494    12    88
12625 12665 12776 12788 12799 12887 12998 13065 13109 13143 13220 13276 13309 13331 13332 13443 13448 13554 13643 13665 13688 13776 13887 13954 13998 14109 14110
   42    60   444   570   348   224   606   386     4   432    40   300   292   682     8    32    12   364   710   608   420    28   560   150    72     6   184
14154 14176 14187 14198 14209 14220 14276 14298 14331 14442 14443 14554 14665 14710 14776 14887 14998 15109 15220 15331 15442 15553 15554 15665 15765 15776 15810
  192   212   518    44   694   190   416   318    72   158    14    30   130   158   552   236   588   396   320   504   268   514    34   156   192   134   156
15832 15887 15921 15998 16109 16220 16331 16420 16442 16553 16631 16664 16665 16666 16765 16776 16821 16887 16998 17010 17109 17220 17331 17365 17442 17466 17532
  170   104   240    76   412   112    30   388   168   290   528   568   148   114   426   122   264    84     6    40   220    58    12   420   158     8    12
17553 17560 17664 17775 17776 17887 17950 17998 18109 18110 18220 18287 18331 18343 18442 18553 18609 18664 18665 18775 18831 18867 18886 18887 18998 19109 19166
    2    20    12   618    76   138    10    78     6   340    28   506    10   396    24    36   122    14   332   140   232     4  2032   120     2    30    12
19198 19220 19287 19331 19409 19442 19443 19454 19460 19542 19553 19664 19754 19775 19886 19997 19998 20109 20220 20227 20331 20442 20553 20664 20742 20775 20886
```

The function degree() can be found in various packages, but in this case, we are using the one from the igraph package. This function returns the number of edges connected to each vertex. By analyzing the degrees of different vertices, we can gather insights about the overall structure and density of the graph from the data.

To visualize the distribution of the degrees of nodes in the graph, we'll create a histogram.

```
> #histogram
> hist(igraph::degree(g))
>
```



This histogram indicates that the majority of nodes exhibit degrees falling within the range of 0 to 500, while the remaining nodes have degrees distributed between 500 and 1000.

- **centr_betw() - Betweenness Centrality**

```
> #Betweenness Centrality
> g.between  = igraph::centr_betw(g)
> g.between
$res
   [1] 3.151719e+06 3.981881e+06 3.266552e+06 1.206334e+05 7.363342e+06 9.469315e+06 1.701157e+06 8.122346e+06 2.725038e+06 5.273894e+06 3.744818e+06 4.063249e+05
  [13] 8.692060e+06 2.335483e+06 8.783339e+02 1.474652e+06 3.244700e+05 3.461340e+06 2.436035e+06 1.039556e+06 7.101634e+06 9.004451e+05 2.007443e+06 1.553106e+07
  [25] 2.858003e+06 1.403503e+05 4.735297e+06 1.712402e+06 2.053627e+06 2.216830e+06 5.637427e+07 2.877706e+06 1.921803e+05 1.090843e+07 2.662356e+03 5.445164e+06
  [37] 4.715068e+06 4.811867e+05 1.129239e+04 5.629457e+05 9.515164e+05 2.272398e+06 2.380167e+07 1.479723e+06 2.850635e+06 4.453057e+06 9.638072e+05 3.046881e+06
  [49] 1.930011e+05 2.534605e+07 3.859493e+04 2.311553e+06 6.605852e+06 7.532751e+05 6.104356e+06 5.109320e+05 4.907386e+07 1.259179e+07 3.950718e+05 2.408225e+03
  [61] 1.068249e+07 1.672638e+07 7.387049e+06 1.527451e+05 1.285051e+06 1.412434e+06 2.400280e+06 1.313003e+04 3.802958e+06 4.471660e+06 2.173360e+07 3.344439e+05
  [73] 8.976368e+07 1.215126e+06 1.577523e+04 1.594366e+07 2.546772e+06 1.589372e+05 4.050165e+06 5.511125e+06 1.211066e+07 1.410818e+04 3.086957e+06 1.609299e+08
  [85] 8.555053e+05 2.563614e+06 2.308962e+07 4.421859e+05 1.343681e+06 8.883725e+06 7.132626e+03 9.118853e+06 9.727240e+05 3.628466e+03 2.363156e+06 8.622036e+06
  [97] 6.774917e+05 1.973869e+05 9.096726e+05 4.077454e+06 4.890626e+02 5.032214e+06 1.153062e+07 2.657635e+06 3.215569e+01 3.119489e+06 2.223556e+05 2.428348e+04
 [109] 3.576354e+06 2.992492e+07 6.872374e+06 5.773731e+06 6.494667e+05 1.308536e+06 4.799707e+05 2.080087e+06 1.582126e+06 1.539632e+05 1.296378e+03 7.188281e+06
 [821] 1.249907e+05 3.921050e+04 2.453724e+05 1.001238e+06 2.199000e+06 2.418541e+06 2.493040e+06 4.073257e+05 2.260590e+06 6.034509e+05 9.106994e+05 6.197303e+05
 [829] 8.730475e+04 2.264560e+06 6.659704e+05 5.451647e+05 1.231348e+07 1.748093e+05 1.498658e+06 8.421703e+05 1.194805e+07 3.380690e+06 8.838111e+04 1.200730e+06
 [841] 5.288801e+06 0.000000e+00 1.704048e+05 2.118121e+06 2.088829e+06 1.102056e+07 8.162086e+05 4.783764e+05 1.518087e+05 9.405069e+06 6.446007e+05 2.281680e+04
 [853] 6.866653e+05 3.129073e+05 5.996400e+04 9.705466e+05 1.264224e+06 2.356403e+05 6.503066e+03 1.098209e+06 2.656243e+06 1.595371e+03 2.903592e+06 2.552012e+03
 [865] 7.967920e+06 6.020077e+06 9.446279e+06 9.601434e+03 1.318922e+06 6.190208e+06 4.458336e+05 4.195658e+07 1.436287e+06 2.257285e+06 2.249448e+05 2.232670e+05
 [877] 2.164062e+06 9.352729e+05 7.113332e+05 2.624993e+06 1.577630e+06 7.643601e+05 8.334756e+06 5.934927e+05 1.935572e+06 1.626048e+05 9.303607e+06 8.324182e+05
 [889] 1.524336e+06 3.524911e+05 8.777645e+05 1.426345e+06 1.782164e+05 3.382947e+05 3.287366e+06 6.184535e+04 5.083898e+05 1.545046e+06 2.122417e+05 5.045005e+06
 [901] 4.798505e+06 6.281231e+06 5.267056e+06 3.408369e+05 3.742723e+06 7.395733e+05 3.075464e+05 7.739140e+05 4.502238e+04 7.028148e+05 3.699075e+05 4.319918e+05
 [913] 4.918855e+05 5.855767e+05 9.524471e+05 3.419097e+04 6.812148e+06 1.102528e+06 1.713223e+06 3.308208e+01 1.360030e+06 6.556471e+05 1.753463e+06 8.204644e+05
 [925] 3.164228e+06 4.854558e+05 3.767764e+05 4.591628e+05 5.510034e+05 3.323746e+03 1.365638e+04 1.585927e+05 1.733604e+06 1.978938e+06 2.914033e+05 7.865149e+06
 [937] 3.969551e+05 9.768958e+05 1.208389e+06 1.634546e+06 0.000000e+00 1.015647e+02 1.600146e+06 5.689131e+05 3.056968e+06 1.128253e+07 9.022492e+02 1.764749e+06
 [949] 5.131798e+06 5.613639e+05 1.042252e+07 3.413214e+05 1.180260e+06 2.864004e+06 1.831078e+06 1.770580e+06 1.046662e+06 1.033573e+06 1.179021e+05 9.361194e+05
 [961] 1.518796e+05 3.266932e+05 2.991570e+05 2.559759e+06 1.350596e+06 2.981400e+05 6.275796e+06 2.968178e+06 2.268620e+06 1.542616e+04 6.550544e+05 1.684846e+06
 [973] 1.027345e+07 1.358577e+06 3.269994e+06 4.239067e+05 2.237957e+06 5.055153e+05 2.921046e+05 3.301058e+06 4.078889e+06 2.041223e+03 8.165402e+05 2.235724e+06
 [985] 6.009564e+05 3.427375e+07 1.131710e+06 8.442873e+06 1.852482e+06 2.690190e+05 4.862595e+05 1.667332e+06 4.150323e+05 8.909589e+06 1.117529e+06 1.650705e+07
 [997] 1.983223e+05 1.918185e+05 6.836492e+07 1.724489e+05
 [ reached getOption("max.print") -- omitted 74879 entries ]

$centralization
[1] 0.0721838

$theoretical_max
[1] 4.368596e+14
```

To determine the number of shortest paths that pass through each vertex in a graph, we can analyze its betweenness centrality. Usually, graphs have fewer nodes with high betweenness, i.e., nodes that significantly impact the network's connection. Instead, the betweenness of the nodes is generally smaller. While this is good in many cases, it could also imply that the nodes in the graph are farther apart from each other, which may not be ideal for certain scenarios.

- **centr_clo() - Closeness Centrality**

```
> #Closenesss Centrality
> g.closeness  = igraph::centr_clo(g)
> g.closeness
$res
   [1] 0.3235830 0.3182144 0.3204927 0.2823251 0.3207840 0.3293043 0.3087729 0.3229494 0.3021660 0.3246296 0.3207826 0.2975331 0.3315679 0.2943417 0.2642594
  [16] 0.3140524 0.2716610 0.3160723 0.3123369 0.3177786 0.3239823 0.2667126 0.3209699 0.3424100 0.3161158 0.2923683 0.3077684 0.3178638 0.3058456 0.3158092
  [31] 0.3518479 0.3082924 0.2727371 0.3334139 0.2699349 0.3197984 0.3230250 0.2749230 0.2786445 0.3031511 0.3050439 0.3111585 0.3431999 0.3161987 0.3207338
  [46] 0.3198981 0.3099168 0.3268967 0.2949711 0.3353028 0.2754350 0.3199953 0.3284334 0.2871730 0.3058234 0.2739839 0.3554810 0.3386626 0.2808288 0.2706291
  [61] 0.3264200 0.3364954 0.3282544 0.2685268 0.3009913 0.3027798 0.3189461 0.2903622 0.3216570 0.3056891 0.3388955 0.3063370 0.3585687 0.3120068 0.2611470
  [76] 0.3222253 0.3155636 0.2713161 0.3162277 0.3303021 0.3366984 0.2834219 0.3070088 0.3623253 0.2915404 0.3218726 0.3302819 0.3036437 0.3008827 0.3308768
  [91] 0.2756131 0.3384390 0.2847589 0.2789292 0.2934538 0.3306302 0.3060861 0.2779269 0.3065214 0.3219778 0.2691621 0.3270278 0.3197081 0.3177414 0.2569664
 [106] 0.3195344 0.2753540 0.2850606 0.3292143 0.3159775 0.3212213 0.3202370 0.3009663 0.3116672 0.3051727 0.3185003 0.3169278 0.2670825 0.2597710 0.3247977
 [121] 0.2874810 0.3021756 0.3127321 0.2596661 0.2942664 0.3012375 0.3016686 0.3420195 0.2708116 0.3397499 0.2922264 0.2856067 0.3222595 0.3131374 0.2940862
 [136] 0.3136331 0.3109749 0.2923367 0.3370020 0.3063927 0.2561172 0.3188201 0.2633769 0.3013260 0.3235098 0.2732803 0.2679890 0.3040379 0.2842011 0.2759810
 [151] 0.2950433 0.2595489 0.2664092 0.2532991 0.3036072 0.3821987 0.2968301 0.2832959 0.3070821 0.2857745 0.2619955 0.2830032 0.2578413 0.3019556 0.3018607
 [166] 0.2685401 0.3273367 0.3101955 0.3298053 0.2701232 0.2839161 0.2977760 0.3348337 0.2848904 0.3188014 0.2930718 0.3036255 0.2754860 0.3055610 0.2885963
```

```
$centralization
[1] 0.7633521

$theoretical_max
[1] 75877
```

The function centr_clo() returns the closest centrality of each node in the graph. This value indicates how close a node is to other nodes in the graph. The nodes in the graph g are well connected and reachable from most other nodes, as indicated by the average proximity centrality of 0.3.

- **shortest.paths - Shortest Path Between Two Nodes**

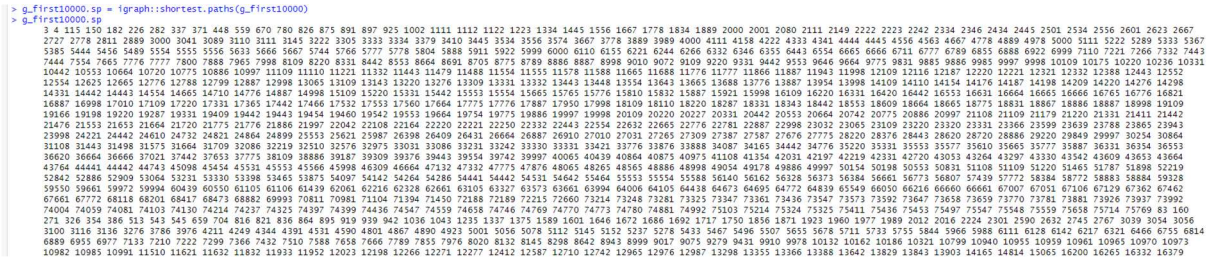
Shortest Path of first 100 Nodes

```
> #Plotting the first 100 rows
> g_first100 <- directed_graph(1,100)
> plot(g_first100,vertex.size=5,edge.arrow.size=0.1,vertex.label = NA)
> g_first100.sp = igraph::shortest.paths(g_first100)
> g_first100.sp
     3 4 115 150 182 226 282 337 371 448 559 670 780 826 875 891 897 925 1002 1111 1112 1122 1223 1334 1445 1556 1667 1778 1834 1889 2000 2001 2080 2111 2149 2222
3    0 2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
4    2 0   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
115  2 2   0   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
150  2 2   2   0   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
182  2 2   2   2   0   2   2   2   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
226  2 2   2   2   2   0   2   2   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
282  2 2   2   2   2   2   0   2   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
337  2 2   2   2   2   2   2   0   2   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
371  2 2   2   2   2   2   2   2   0   2   2   2   2   2   2   2   2   2   2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
     2223 2242 2334 2346 2434 2445 2501 2534 2556 2601 2623 2667 2727 2778 2811 2889 3000 3041 3089 3110 3111 3145 3222 3305 3333 3334 3379 3410 3445 3534 3556
3       2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
4       2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
115     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
150     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
182     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
226     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
282     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
337     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
371     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
     3574 3667 3778 3889 3989 4000 4111 4158 4222 4333 4341 4444 4445 4556 4563 4667 4778 4889 4978 5000 5111 5222 5289 5333 5367 5385 5444 5456 5489 5554 5555
3       2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
4       2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
115     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
150     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
182     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
226     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
282     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
337     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
371     2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2    2
     5556 5633 1
3       2    2 1
4       2    2 1
115     2    2 1
150     2    2 1
182     2    2 1
226     2    2 1
282     2    2 1
337     2    2 1
371     2    2 1
 [ reached getOption("max.print") -- omitted 92 rows ]
>
```

The function paths(g) in igraph can be used to find the shortest path between any two nodes in a graph g. The output of this function is a matrix which shows the shortest route between nodes x and y.

By analyzing the matrix, we can identify nodes or vertices that are more central or isolated than others. The above example shows that most of the shortest paths are of value 2 for the first 100 nodes of the graph.

Shortest Path of first 1000 Nodes



- **get.shortest.path() - Get the shortest paths between vertices in a graph**

```
$vpath[[756]]
+ 4/75879 vertices, named, from 9d43d07:
[1] 1568 5000 3889 5433

$vpath[[757]]
+ 4/75879 vertices, named, from 9d43d07:
[1] 1568 5000 4778 5467

$vpath[[758]]
+ 4/75879 vertices, named, from 9d43d07:
[1] 1568 5000 2    5496
```

We can use the shortest paths function to obtain both the length and the true shortest paths from node 5 in our graph g. This is very useful as it helps determine how well-connected the node is to other vertices. In most cases, node 2675 has a shortest path length value of 4, indicating that it is reasonably well-connected to other vertices in the graph.

- **max_cliques()**

```
> #max cliques
> node <- c(30)
> g.adj_graph <- igraph::graph_from_adjacency_matrix(g.adj)
Warning message:
In (function (edges, n = max(edges), directed = TRUE)  :
  At vendor/cigraph/src/cliques/maximal_cliques_template.h:219 : Edge directions are ignored for maximal clique calculation.
> g.30clique = igraph::max_cliques(g.adj_graph,min = NULL,max = NULL,subset = node)
> g.30clique
[[1]]
+ 2/75879 vertices, named, from c13651c:
[1] 43506 1317
```

To determine the size of the largest clique in a graph, we can use the max_cliques() function from the igraph package. For instance, node 30's largest clique consists of only two nodes, which are 43506 and 1317. The presence of two cliques suggests that these nodes are closely linked to each other, although not necessarily to the entire graph. Moreover, a higher clique would indicate a greater density of interconnected nodes.

- **Clique_num() - Largest clique in the graph**

```
> #Largest clique
> g.lgcliques = igraph::clique_num(g.adj_graph)
Warning message:
In igraph::clique_num(g.adj_graph) :
  At vendor/cigraph/src/cliques/maximal_cliques_template.h:219 : Edge directions are ignored for maximal clique calculation.
> g.lgcliques
[1] 23
```

The function clique_num() determines the size of the largest clique in the graph, which is 23 in our case.

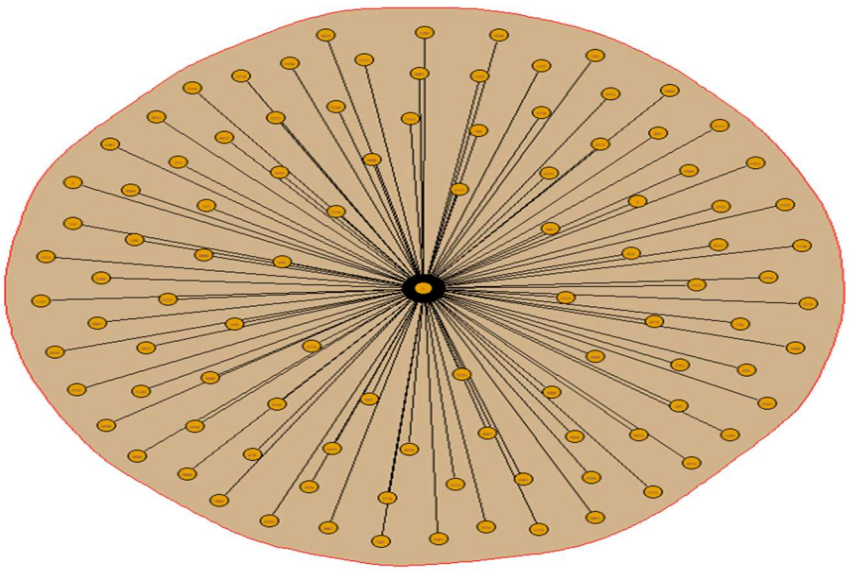
- **Simplify() and is.simple()**

```
> #Simplify function
> sg<-simplify(g)
> is.simple(sg)
[1] TRUE
Warning message:
`is.simple()` was deprecated in igraph 2.0.0.
i Please use `is_simple()` instead.
This warning is displayed once every 8 hours.
Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
>
```

The function is.simple() helps us to determine if a graph is simple, which means it has no loops and only one edge between any two vertices. After analyzing our graph, we found that it was already simple without any additional simplification required. We confirmed this by running the graph through the simplify() function, and the is.simple() function continued to return true.

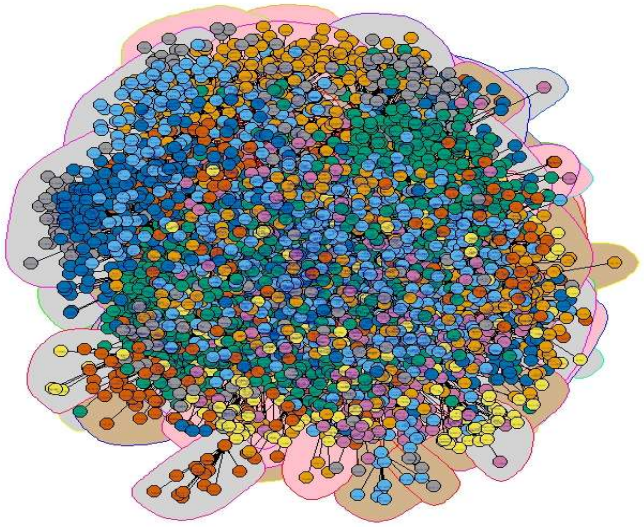
- **Detecting Structures using walktrap.community()**

```
#Detecting Structures using walktrap.community()
wc<-walktrap.community(g_first100)
plot(wc,g_first100,vertex.size = 5,vertex.label.cex=0.2, edge.arrow.size = 0.1, layout=layout.fruchterman.reingold,mark.col=c("tan", "pink", "lightgray"))
```



The walktrap.community() function searches a graph for highly connected subgraphs or communities. Typically, brief random walks stay within the same community. As shown in the graphic above, several communities are highlighted. Although there are a few outliers outside the blob, the majority of the subgraphs are compact and located close to one another near the center.

```
> #Detecting Structures using walktrap.community()
> wc<-walktrap.community(g_first10000)
> plot(wc,g_first10000,vertex.size = 5,vertex.label.cex=0.2, edge.arrow.size = 0.1, layout=layout.fruchterman.reingold,mark.col=c("tan", "pink", "lightgray"))
>
```



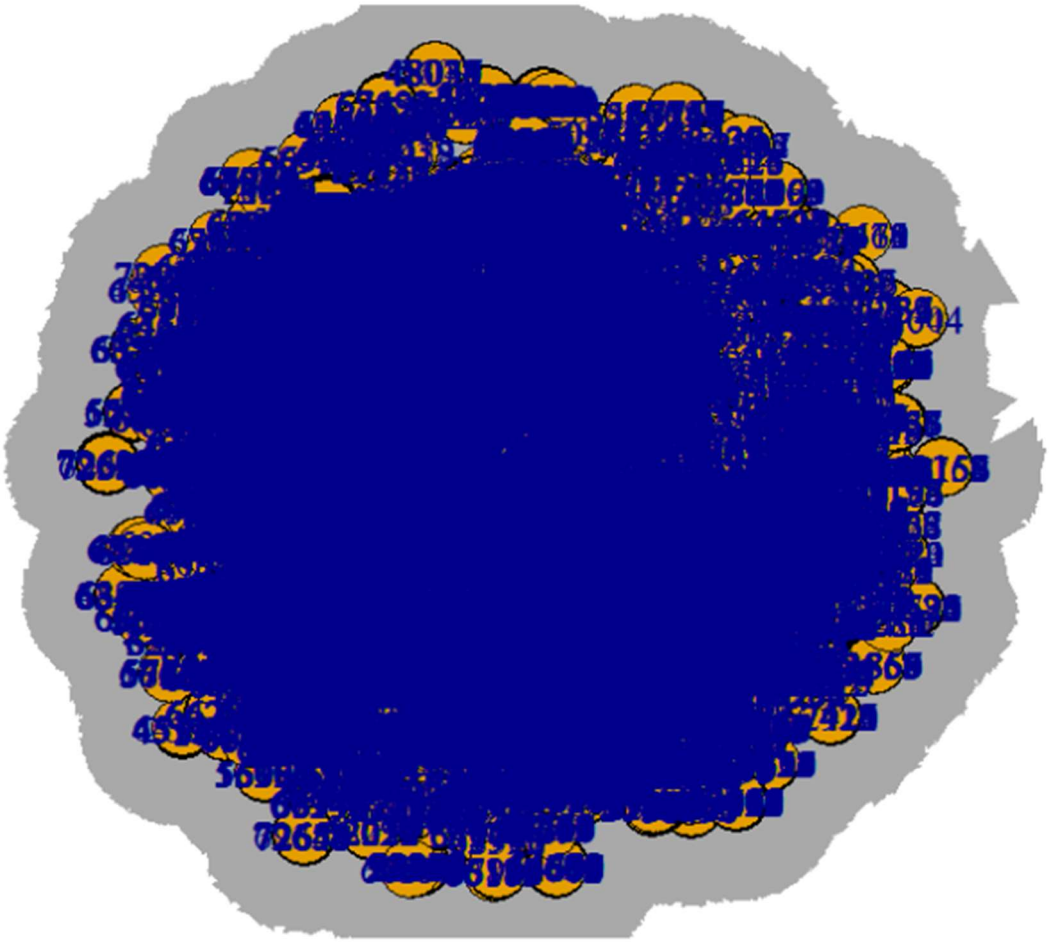
- **Alpha Centrality:**

  Based on our analysis, we have found that node 75758 has the highest alpha centrality value of 46.1810604 in the network, indicating it holds significant influence. Following this node, the nodes 4401, 9202, and 35542 have the next highest alpha centralities, in that order. The remaining alpha centralities are displayed in descending order above. Specifically, 11.1791318, 9.1115509, and 8.8253954 have alpha centrality values lower than the aforementioned nodes.

```
> #Alpha Centrality
> acg <- alpha.centrality(g_first1000)
> sort(acg,decreasing = TRUE)
    1     2     3     4   115   150   182   226   282   337   371   448   559
  683   319     1     1     1     1     1     1     1     1     1     1     1
  670   780   826   875   891   897   925  1002  1111  1112  1122  1223  1334
    1     1     1     1     1     1     1     1     1     1     1     1     1
 1445  1556  1667  1778  1834  1889  2000  2001  2080  2111  2149  2222  2223
    1     1     1     1     1     1     1     1     1     1     1     1     1
 2242  2334  2346  2434  2445  2501  2534  2556  2601  2623  2667  2727  2778
    1     1     1     1     1     1     1     1     1     1     1     1     1
 2811  2889  3000  3041  3089  3110  3111  3145  3222  3305  3333  3334  3379
    1     1     1     1     1     1     1     1     1     1     1     1     1
 3410  3445  3534  3556  3574  3667  3778  3889  3989  4000  4111  4158  4222
    1     1     1     1     1     1     1     1     1     1     1     1     1
 4333  4341  4444  4445  4556  4563  4667  4778  4889  4978  5000  5111  5222
    1     1     1     1     1     1     1     1     1     1     1     1     1
 5289  5333  5367  5385  5444  5456  5489  5554  5555  5556  5633  5666  5667
    1     1     1     1     1     1     1     1     1     1     1     1     1
 5744  5766  5777  5778  5804  5888  5911  5922  5999  6000  6110  6155  6221
    1     1     1     1     1     1     1     1     1     1     1     1     1
```

4. Graph Simplification

Multiple strategies exist for graph simplification. Upon plotting the original graph derived from the dataset, a distinctive cluster, denoted as the "Blue Blob," becomes apparent. It's represented below:



So, to simplify the graph the first step involves the removal of nodes with a limited number of edges. Initially, an assessment is conducted to identify nodes with a degree of 1, signifying nodes connected to only one other node. This preliminary analysis serves as a foundational step in the process of graph simplification, aiming to enhance clarity and reduce complexity in subsequent visualizations and analyses.

First, let’s check how many nodes are present with a degree less than 1.

```
> #SIMPLIFICATION
> #================
> #Nodes with degree 1
> igraph::V(g)[igraph::degree(g)<1]
+ 0/75879 vertices, named, from 4c7ec46:
```

It appears that there are no nodes or vertices within the graph exhibiting a degree less than 1. So, let's identify the nodes with a degree less than 2. Subsequently, nodes characterized by a degree less than 2 will be targeted for removal, thereby streamlining the graph by eliminating these relatively less interconnected nodes.

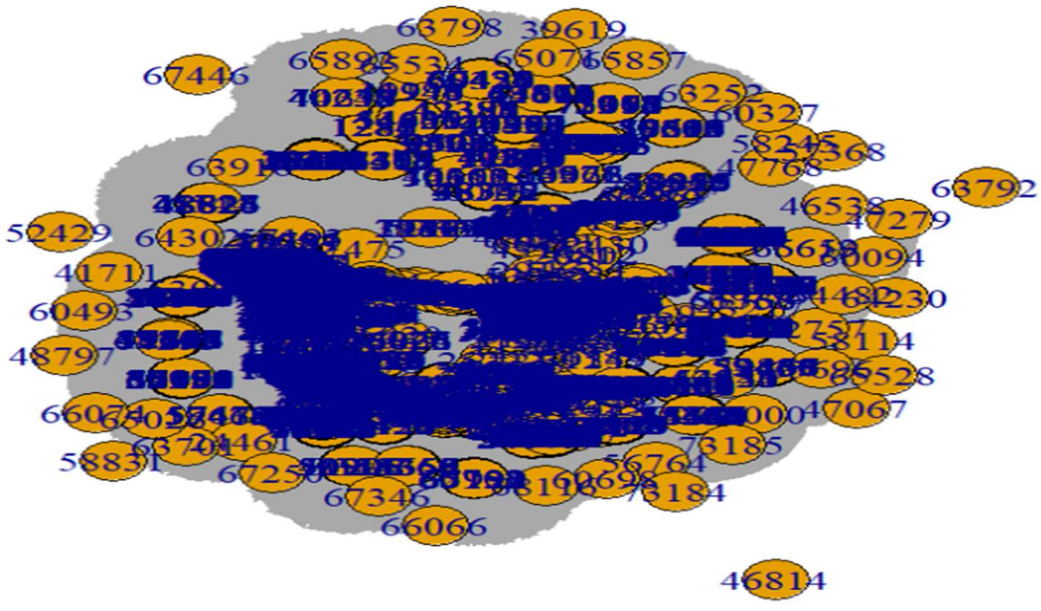
```
> #Nodes with degree 2
> igraph::V(g)[igraph::degree(g)<2]
+ 0/75879 vertices, named, from 4c7ec46:
```

Well, it seems this data set does not have any vertices with degrees less than 1 or 2. So, let's try something larger, such as 9.

```
> #Nodes with degree 9
> igraph::V(g)[igraph::degree(g)<9]
+ 56412/75879 vertices, named, from 596233b:
   [1] 4158  5777  7777  7888  8775  9109  10236 10664 11554 13109 13332 14109
  [13] 16998 17466 17553 18109 18867 18998 19442 19460 19754 19775 23639 24864
  [25] 27031 31575 32510 40439 49178 51108 53875 70811 70981 72188 72189 73781
  [37] 6128  8145  10961 10965 13642 16200 18246 25696 27442 27533 31352 40220
  [49] 40442 41331 41553 41775 41886 42108 42442 42555 43219 43442 43997 44554
  [61] 44776 44887 45109 45442 46665 46776 46887 47442 48664 49109 49853 53657
  [73] 65485 68311 72159 72161 72162 72163 75028 75263 2145  2156  2257  2312
  [85] 2368  2379  2446  2512  70210 70211 70212 4435  49531 51820 51831 51875
  [97] 69192 70051 70052 7415  7417  17170 41796 41797 41799 41801 75002 41802
 [109] 41803 13945 7419  7423  7425  7428  9212  41804 41805 1262  25495 26186
+ ... omitted several vertices
```

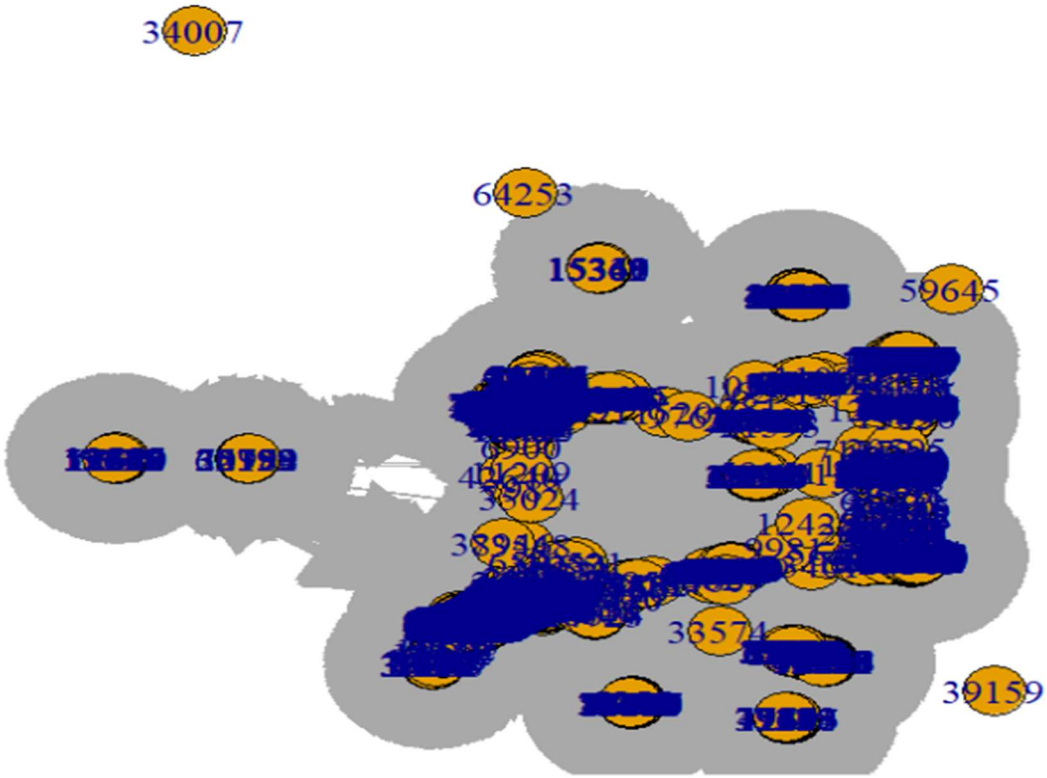
There are 56412 vertices with degree less than 9. Let's try to remove all these vertices with less than 9 edges and plot the graph.

```
#Nodes with degree 9
igraph::V(g)[igraph::degree(g)<9]
#Simplifying the graph by deleting the nodes with degree<9
v9<-igraph::V(g)[igraph::degree(g)<9]
g9<-igraph::delete.vertices(g,v9)
plot(g9)
```



The graph still seems complex, so let's remove all the nodes with degree less than 25 and plot the graph for the same.

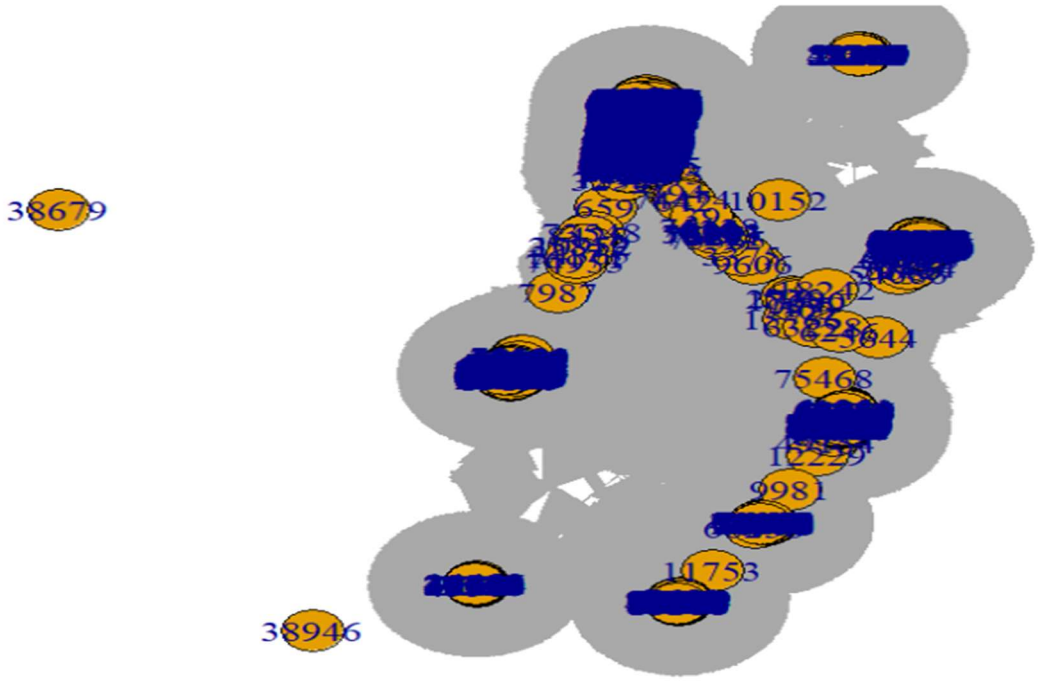
```
#Simplifying the graph by deleting the nodes with degree<25
v25<-igraph::V(g)[igraph::degree(g)<25]
g25<-igraph::delete.vertices(g,v25)
plot(g25)
```



Well, didn't help much! Still, the clusters are not clearly differentiated. So, remove all the nodes with a degree less than 60 and plot the graph for the same.

```
#Simplifying the graph by deleting the nodes with degree<60
v60<-igraph::V(g)[igraph::degree(g)<60]
g60<-igraph::delete.vertices(g,v60)
plot(g60)
```

We can now observe from the following graph the clusters are visible and the graph is less complex.



5. Determining the (a) central nodes(s) in the graph, (b) longest path(s), (c) largest clique(s), (d) ego(s), and (e) power centrality.

- **CENTRAL NODES(S) IN THE GRAPH**

```
> #Central Nodes in the Graph
> most_central <- which.max(degree(g, mode="all"))
> most_central
8886
 156
```

- **LONGEST PATH(S)**

To find the longest path between two nodes in a graph, we use the components() function to extract the longest path, which must be in the greatest connected component. After that, we create a subgraph that contains only the vertices inside that largest connected component. We also assign each vertex a degree attribute.

Once we have the subgraph, we can calculate the longest distance between any two vertices using DFS. Our analysis shows that the longest path between nodes 61710 and 67561 is 9955.

```
> #Longest Paths
> sg = induced.subgraph(g,which(components(g) $membership == 1 ))
> V(sg)$degree = degree(sg)
> result = dfs(sg,root=1,dist = TRUE)$dist
> sort(result,decreasing = TRUE)
61710 67561 36935 61709 61711 63401 63893 38748 69949 57193 61679 61680 61681
 9955  9955  9954  9954  9954  9954  9954  9953  9953  9953  9953  9953  9953
61682 41214 41212 41213 32077 75287 24951 55548 57192 59268 59269 59582 59583
 9953  9953  9953  9953  9952  9952  9952  9952  9952  9952  9952  9952  9952
60676 60677 68358 68359 61180 61687 61688 67355  4166  9044 32087 49257 24360
 9952  9952  9952  9952  9952  9952  9952  9952  9951  9951  9951  9951  9951
30164 55547 32139 39283 57788 57789 28171 59581 61099 28673 62459 67773 30931
 9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9951  9950  9950
13882 25170 46825 32078 32079 25244 15469 54098 18327 55830 55831  6962 20397
 9950  9950  9950  9950  9950  9950  9950  9950  9950  9950  9950  9950  9950
34901 61686 66399 66400 67158 67159  4168 75388 32085 25235 49256 14037 20036
 9950  9950  9950  9950  9950  9950  9949  9949  9949  9949  9949  9949  9949
55546 57804 57805 57806 57808 57809 57810 57811 57812 58535 59411 60955 32084
 9949  9949  9949  9949  9949  9949  9949  9949  9949  9949  9949  9949  9949
61393 66333 66334 66335 15875 75727 32074 20439 49237 30721 50441 54651 24757
 9949  9949  9949  9949  9948  9948  9948  9948  9948  9948  9948  9948  9948
29103 30819 55076 55077 55078 55079 55080 55081 56359 56360 56361 56363 56364
 9948  9948  9948  9948  9948  9948  9948  9948  9948  9948  9948  9948  9948
56365 56366 56367 56368 56369 56370 56371 56372 56374 56375 21621 22526 61100
```

- **LARGEST CLIQUE(S)**

```
> #print largest clique
> largest.cliques(g)
[[1]]
+ 23/75879 vertices, named, from bf5d7bf:
 [1] 26109 45553 75103 38109 226   44441 68882 15553 337   22220 49998 56661 2111  2556  74770 31442 2222  1111  50109 38775 75547 17775 21108

[[2]]
+ 23/75879 vertices, named, from bf5d7bf:
 [1] 26109 45553 75103 38109 226   44441 68882 15553 337   22220 14776 56661 75436 1889  2556  17775 38775 75547 21108 50109 59883 74770 62661

[[3]]
+ 23/75879 vertices, named, from bf5d7bf:
 [1] 26109 45553 75103 38109 226   44441 68882 15553 337   22220 14776 56661 75436 1889  2556  17775 38775 75547 21108 50109 59883 74770 2222

[[4]]
+ 23/75879 vertices, named, from bf5d7bf:
 [1] 26109 45553 33443 68882 38109 226   44441 2222  337   22220 15553 56661 2111  31442 1111  17775 49998 50109 21108 74770 75547 2556  38775

[[5]]
+ 23/75879 vertices, named, from bf5d7bf:
 [1] 26109 45553 33443 68882 38109 226   44441 2222  337   22220 15553 56661 1889  14776 75436 38775 21108 17775 50109 2556  75547 74770 59883
```

In our analysis, we found that the minimum degree required for each vertex in the largest possible clique is 23. We were also able to directly determine the largest cliques in our graph 'g' using the 'largest_cliques()' method, as shown in the above findings.

- **EGO**

```
> #EGOS for All Nodes
> ego.graph <- igraph::ego(g)
> 
```

```
[[997]]
+ 53/75879 vertices, named, from bf5d7bf:
 [1] 32331 4222  5000  8886  13554 19886 20886 23331 33330 33776 44441 63327
[13] 73658 326   24220 30442 34554 37998 45331 2     75647 55552 4457  93
[25] 70993 1734  66549 16043 1434  73392 73614 75571 1401  74118 48431 24265
[37] 66516 9887  75059 60    24466 20809 28498 72771 1512  73213 74314 73642
[49] 28409 75634 74355 40243 75862

[[998]]
+ 35/75879 vertices, named, from bf5d7bf:
 [1] 32442 7800  8886  12187 12788 15331 15921 20220 58883 16265 16332 25442
[13] 70549 2     2112  35409 47365 25398 11743 35132 12444 74636 9953  7599
[25] 31409 36221 43942 62806 75176 74615 54620 75864 26987 45599 75863

[[999]]
+ 978/75879 vertices, named, from bf5d7bf:
  [1] 32487 1334  2778  2889  3111  3145  3334  4778  5778  6855  7665  12388
 [13] 13065 14276 19997 21109 21553 22332 31664 32576 36553 37775 42031 43330
 [25] 47876 48886 49886 54441 59994 61106 67051 67106 73281 74103 83    326
 [37] 354   939   1337  1375  1589  1750  2301  4211  4531  4801  4890  5001
 [49] 5056  5145  5278  5655  5678  6889  6955  9431  10132 10965 12987 18376
 [61] 18932 24154 24443 27331 27799 31920 35554 36887 36998 39409 44197 44330
 [73] 47121 49553 54498 55264 55331 57473 70039 70549 70882 71305 72550 73249
 [85] 74212 74443 74747 74867 74925 2     2056  2112  74870 3501  9054  15209
 [97] 17854 65883 75581 2232  11888 14643 55020 61784 74970 75847 1952  28032
```

For each vertex, we computed the egos, but omitted the majority of the result values.

- **POWER CENTRALITY**

```
> #Power Centrality
> pc <- power_centrality(g_first10000,exponent = 0.8)
> sort(pc,decreasing = TRUE)
    32775      5733     36998     20709     12987     34220      4801     34699
9.0379518 6.1103289 5.5699171 5.2227874 5.0298207 5.0047170 4.8001820 4.6967734
    55643      1989      5567     34331     55610     33776     70771      4211
4.5573774 4.5488463 4.4231494 4.3427504 4.2994229 4.2994019 4.2724810 4.0129967
    55743     34998     32887      4845      1750      2123     11101     18332
3.9099435 3.8297526 3.7300323 3.7300323 3.6873839 3.6855203 3.6742179 3.5035495
     4978     37720     65883      1386     41776     62550     73248     33443
3.4599842 3.4162633 3.4091843 3.4091843 3.3633073 3.3600701 3.3286644 3.3167308
    47021     18987     15799     33064     55020     49886     37954     18398
3.2741055 3.1866247 3.1866247 3.1866247 3.1719744 3.1627688 3.1030120 3.0825984
    36810     33020     33108     73255     53676     69549     43842      7976
3.0825984 3.0717419 3.0662294 3.0630200 3.0586468 3.0586468 2.9604082 2.9422431
    60661     60994      3145      2301     50775     69438     25442     73325
2.9365413 2.9294623 2.9270569 2.8908266 2.8762778 2.8547183 2.8500749 2.8270281
    14309     47221      6911     15910      5445      6955     32998     45943
2.8221680 2.8221680 2.8152427 2.8114779 2.7973157 2.7688294 2.7688294 2.7688294
    54498     49143      5911     52886     57062     74636     11688     61106
```

As operating on the subgraph is less computationally demanding, we use the power_centrality() function to obtain the following outcome for the first 10000 nodes:

6) Discussion

We completed a project that helped us understand how to work with datasets and create graphs from them, especially those that have a medium to large scale. During this project, we gained practical experience in developing our own functions, using the igraph and sna packages, and working with R. We also acquired expertise in making graphs more understandable by charting and visualising them, simplifying them, and modifying their parameters. This project covered a range of graph analytics tools that provided us with valuable insights into the structure and interactions within the graph.

We have learned about important metrics that help us understand the issue space, such as power centrality, longest paths, greatest cliques, egos, and alpha centrality. We have also studied how random walks can help us identify communities within a graph. By understanding these metrics, we can identify patterns, structures, and connections within data, and develop efficient methods for analyzing and visualizing complex networks.

After completing this project, we now feel confident in our ability to utilize R for handling large datasets, generating and visualizing graphs, applying diverse functions and metrics to the graphs to extract different insights about the problem domain, and simplifying the graphs for improved analysis.