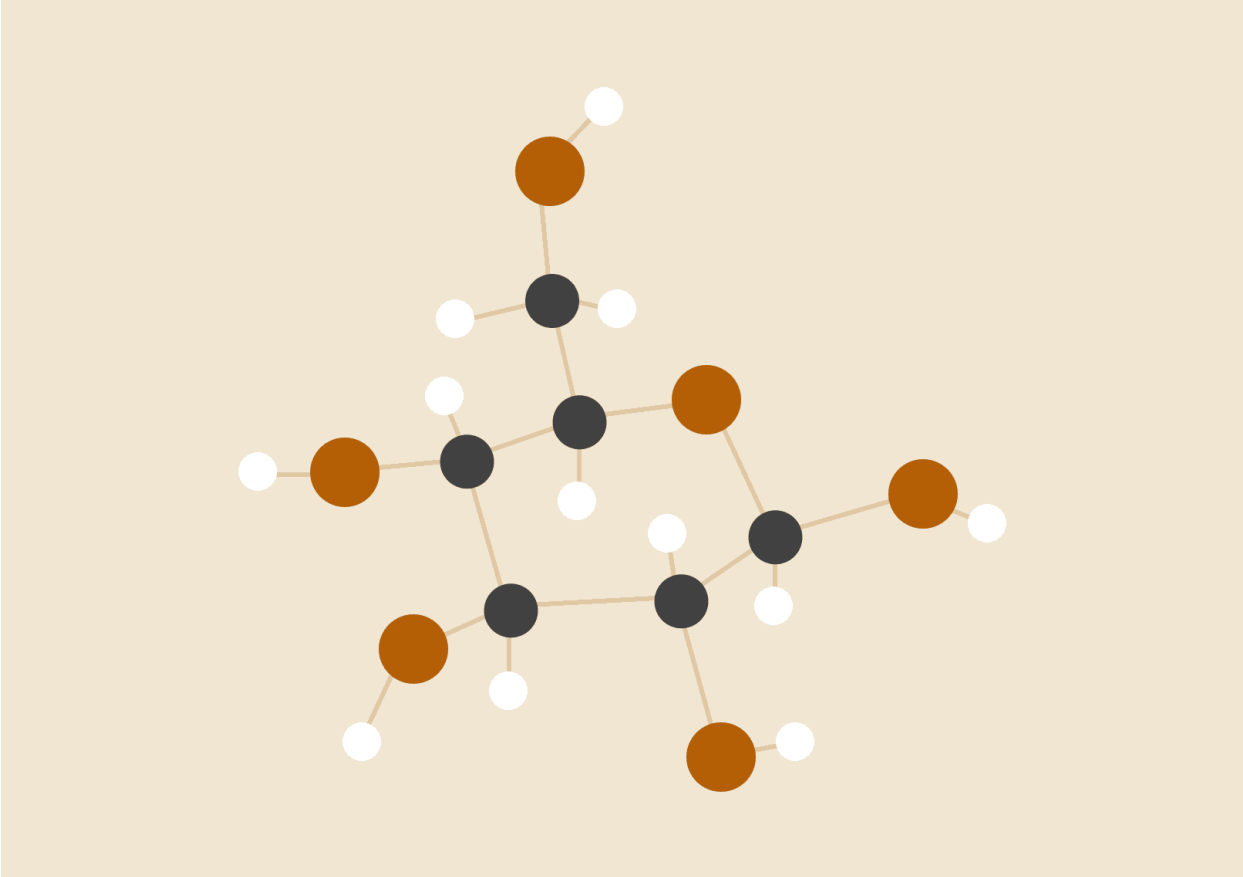
Project 1 - LAB REPORT

Spring 2025



STUDY OF THE “EMAIL - EU” DATASET USING R

Ekaterina Galkina and Yagnamithra Vallabhaneni

Big Data & Analytics

INTRODUCTION

The initial dataset “email - EU” is a network visualising emails transmitted between EU entities. It is a set of pairs of nodes representing a set of edges of the graph. Each node is identified by its name, which is here its id. The graph has near 32,4 thousands of vertices and 54,4 thousands of edges, and is therefore hard to study directly and even just visualising.

This Lab report presents several ways to simplify the initial graph using R studio and some important information revealed during the closer look at the graph’s represented data after its simplification.

METHODS

To be able to study the data, first, we are going to show different methods of graph simplification. Then, we are going to apply functions described in “An introduction to Graph Analytics” on simplified graphs. And for question #4 When it was possible, we studied directly the original graph.

RESULTS

This section describes the code written to study the initial graph. You can find the source code in the file named **sourcecode.r** stored in the initial submitted zip folder.

Demonstrations for question #2

First, we have to upload and store the graph from the dataset file. Below you can find the code to execute to accomplish this task.

First, you should select lines 1 to 22 (Figure 2.1) and click on the run button, the program will ask you to select the file containing the data stored in your computer and then it will read the document and transform the information into a matrix *relations* where the first line would be the initial node and the second - the destination node. Next, this matrix is going to be transformed into an igraph object *g* which will contain the graph itself.

```r
# ---------------------------------------------------
#                 Given graph Email-EU
# ---------------------------------------------------
# Part a - Results for 2a and 2b
# ---------------------------------------------------

# To load the package
library(igraph)

path <- file.choose()
opinions <- read.table(path)

optab <-as.matrix(opinions)
n <- nrow(optab)
v1 <- optab[1:n,1]
v2 <- optab[1:n,2]

relations<- data.frame(from=v1,to=v2)
relations

g <- graph_from_data_frame(relations,directed=TRUE)

# Attempts to plot the whole graph :
plot(g) # not visible at all
# plot.igraph(g) # not visible at all also
# str(mygraph) # too much information, not very useful

plot(g, edge.arrow.size=0.3, edge.color="gray", vertex.size=5, vertex.label=NA) # slightly better
```

Figure 2.1

The next screenshot (Figure 2.2) depicts some of the edges of the graph stored in the file in the form of a matrix *relations*:

```
> relations<- data.frame(from=v1,to=v2)
> relations
     from to
1      26  1
2      29  1
3      51  1
4      52  1
5      56  1
6      82  1
7      83  1
8      85  1
9      87  1
10     94  1
11     95  1
12     98  1
```

Figure 2.2

Then, we can try to plot the entire graph as it is given, (for example by running plot(g) command), however, we can clearly see that the plotted graph is incomprehensive and almost not useful at all.

Below you can find the screenshot of this graph (Figure 2.3), we can see that nothing.

We obtain this by running plot(g).

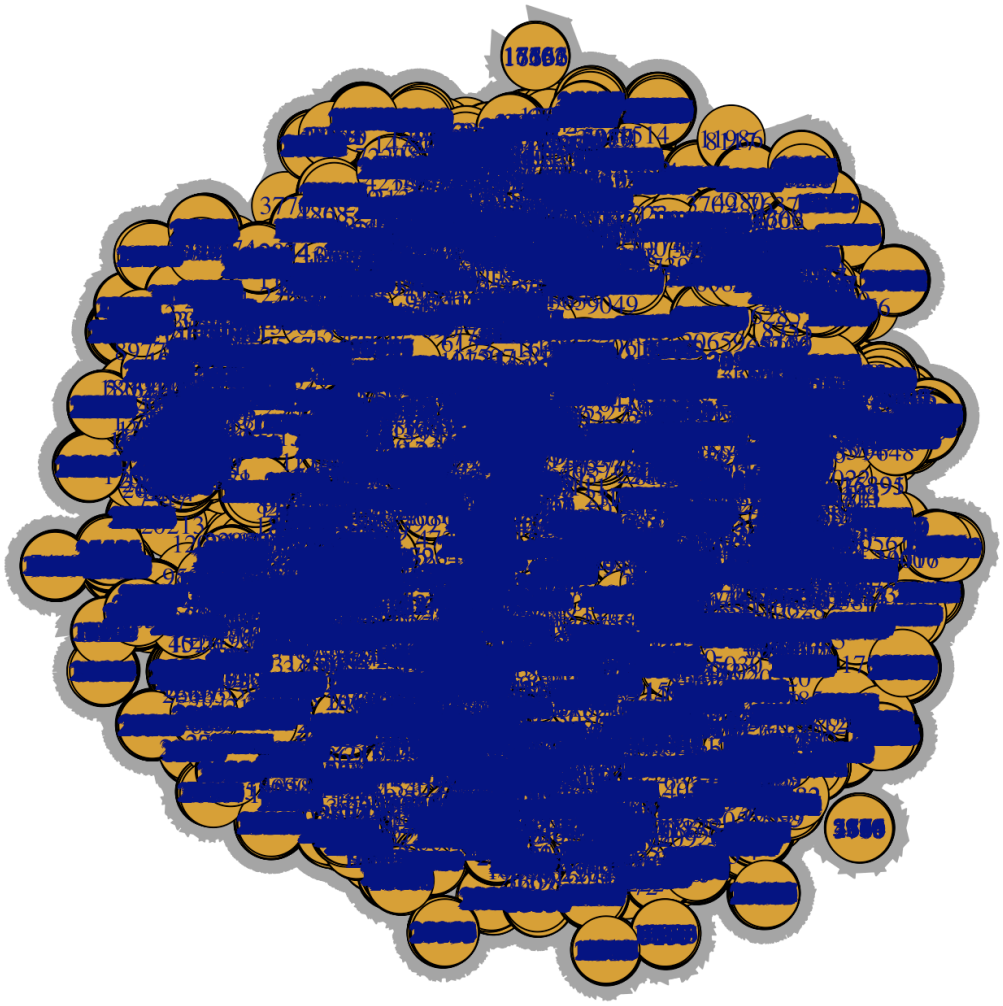


Figure 2.3

We can slightly improve the entire graph representation by lowering the nodes and edges' size, as well as by not displaying the vertices' names.

As we can see on the next screenshot (Figure 2.4), it became better and more visible, nonetheless the graph still contains too much information and it is hard to perceive possible patterns in the dataset. And of course, as the graph is without the nodes' labels it probably became even less useful.

The line of code for this representation is : plot(g, edge.arrow.size=0.3, edge.color="gray", vertex.size=5, vertex.label=NA)

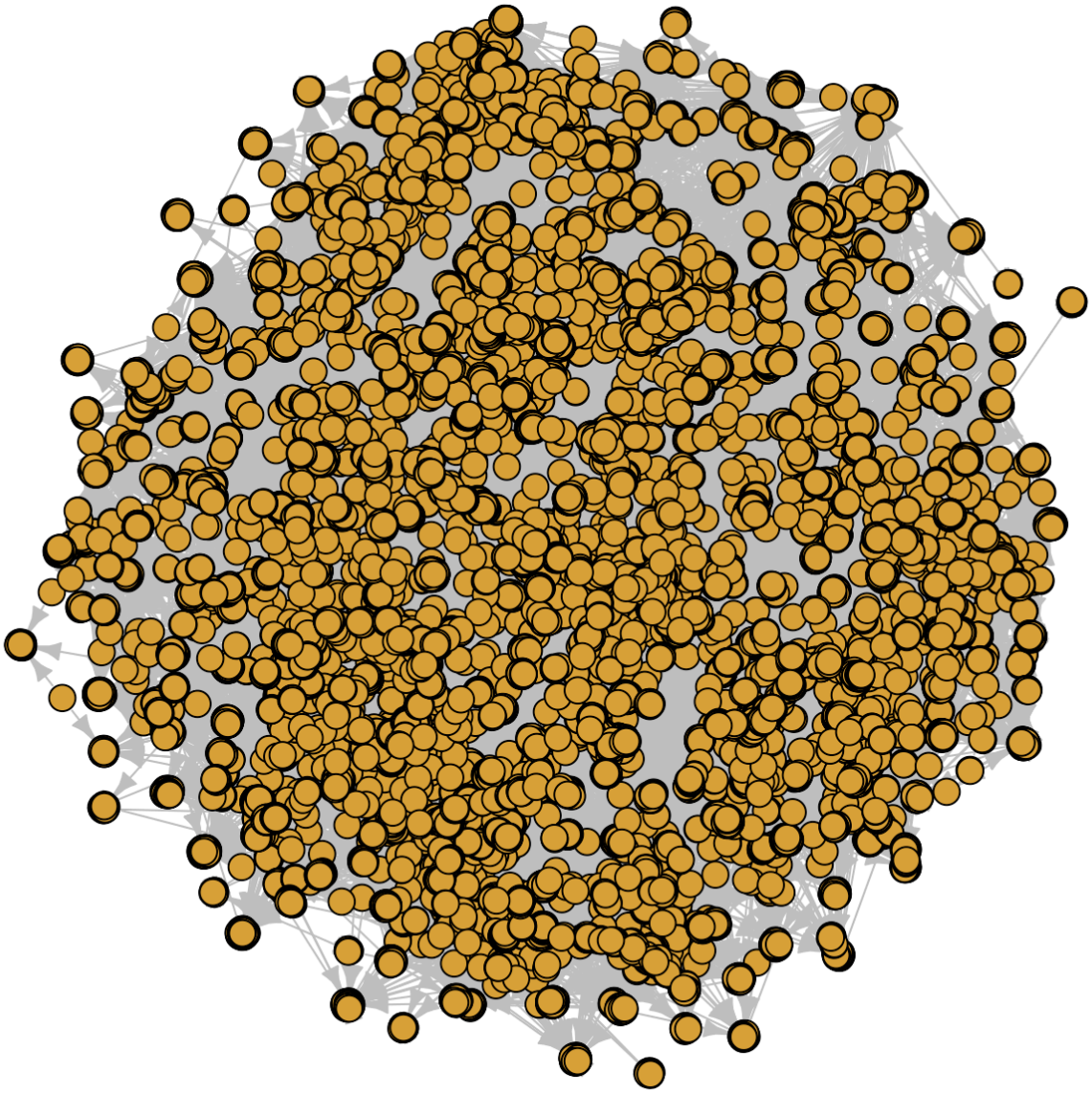


Figure 2.4

Now, let’s begin to simplify the given graph.

We are going to demonstrate several ways to simplify the initial graph to be able to plot and later to analyze it.

First way consists of elimination of some vertices and edges based on a random variable following standard normal distribution.

In the following code, we are first associating to each edge some weight, by generating samples of the random variable and then doing the same action for vertices.

Thereafter, we are computing a subgraph, containing only those vertices whose weight is higher than 1.55. Then, we are removing edges with negative weights to simplify the graph even more and avoid errors due to non positive weights. Finally, we are also removing isolated vertices and the first simplification is ready.

Below (Figure 2.5) you can find the code to execute to obtain the first graph's simplification. By running the following code, the programme will plot two representations of the same graph : the first one is the most simple one, whereas the second one is also showing the graph's communities found by executing a pre-built function which identifies them by random walk method.

```
# First way to simplify the graph, based on random normal variable to eliminate some vertices
E(g)$weight <- rnorm(ecount(g))
V(g)$weight <- rnorm(vcount(g))

sg <- igraph::induced_subgraph(g, which(igraph::V(g)$weight > 1.55))

# Remove edges with weight <= 0
sg <- delete_edges(sg, E(sg)[weight <= 0])

# Remove isolated vertices after edge deletion
sg <- delete_vertices(sg, igraph::degree(sg) == 0)
# plotting the simplified graph
plot(sg, vertex.size=12, edge.arrow.size=0.5)

# Identifying the communities in the simplified graph
wc2 <- cluster_walktrap(sg)
plot(wc2, sg, vertex.size=12, layout = layout.fruchterman.reingold, edge.arrow.size = 0.1)
```

Figure 2.5

Now, by plotting the graph we can perceive a clear and beautiful illustration.

The next screenshot (Figure 2.6) illustrates the second plot with communities separated by different zone colors.

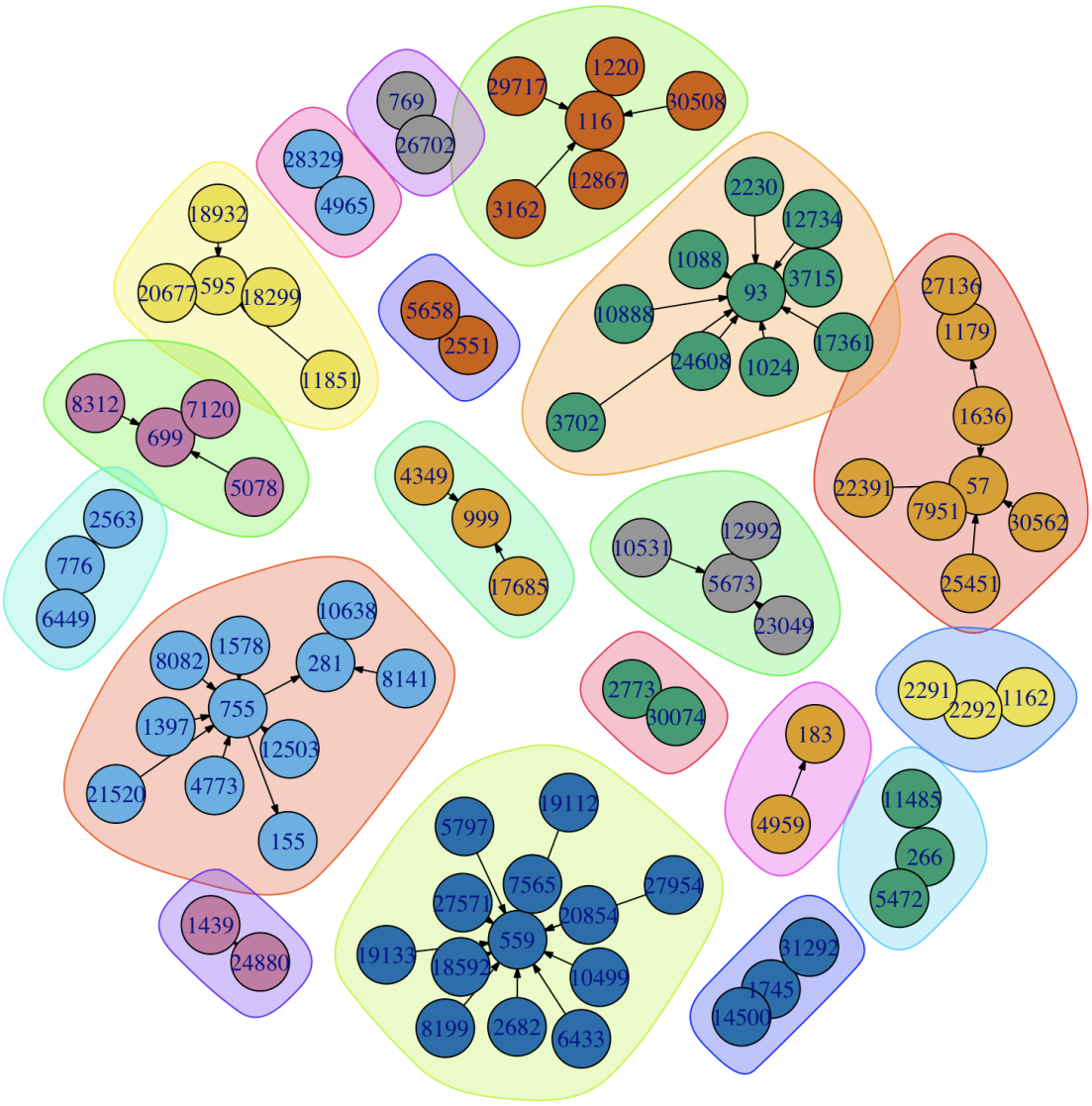


Figure 2.6

Although this simplification may be useful to identify main communities in the graph, this method of graph simplification is too random. Here, we are eliminating a considerable amount of useful information contained in an initial graph and as we can see the graph is not well connected: we can perceive too many connected components, which are not linked between each other.

Therefore, let us present the next less random simplification method. The following

method consists of selecting only nodes which have a considerable amount of neighbors. By consecutively trying several values, we saw that by selecting only vertices with at least 250 neighbors in the initial graph we obtain a comprehensive and yet informative enough graph to be able to perceive interesting patterns.

Here is the code to execute to obtain the new graph and plot it (Figure 2.7):

```
# Second way to simplify : based on each node's number of neighbors
g_simplified <- delete_vertices(g, igraph::degree(g)<=250)
E(g_simplified)$weight <- sample(1, ecount(g_simplified), replace = T)
V(g_simplified)$weight <- sample(1, vcount(g_simplified), replace = T)
plot(g_simplified, vertex.color = "pink", vertex.label.color = "black",
     edge.color = "darkslateblue", vertex.size = 14, edge.arrow.size=0.1)
```

Figure 2.7

The next screenshot illustrates the result produced by the code (Figure 2.8):

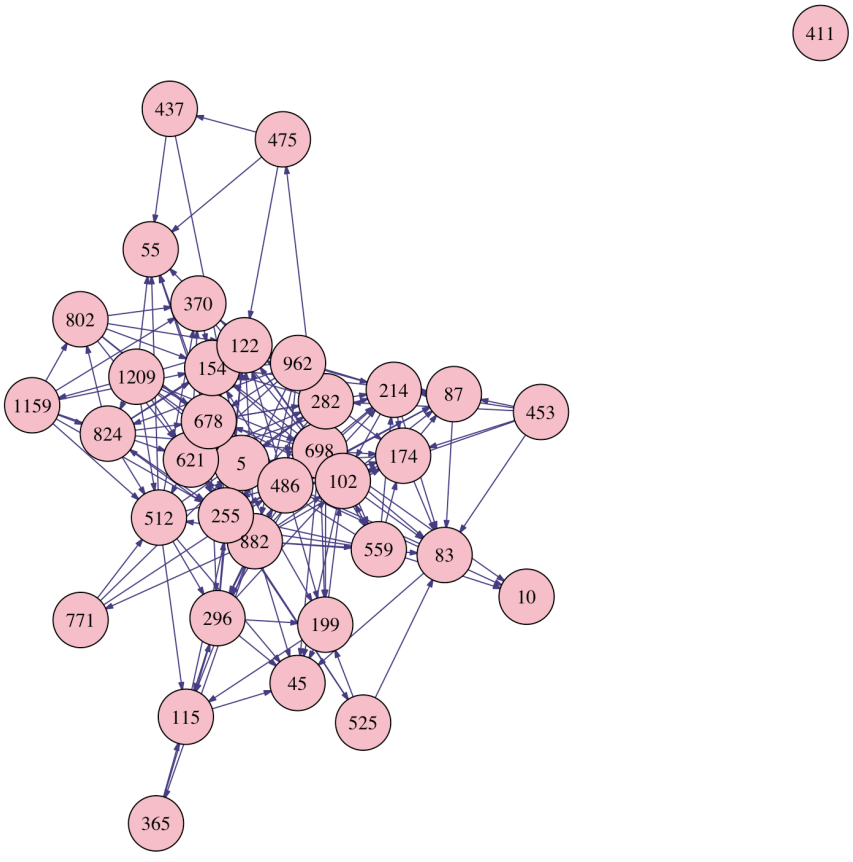


Figure 2.8

Third way of simplifying is by performing community detection on the graph using cluster_walktrap algorithm provided by the igraph library. It detects communities in a graph based on random walks. After identifying the communities within the network, we merged all the vertices belonging to the same community into a single vertex. This merging process was carried out using the contract() function available in the igraph library. For each new vertex, we set its attributes to represent the total sum of the weights of all the vertices that were combined within that community.

Figure 2.9 shows the code to execute to obtain the new graph:

```
# Third way to simplify
# compute clusters ----
cl <- walktrap.community(g, steps = 5)

#contract vertices----
E(g)$weight <- 1
V(g)$weight <- 1

mygraph_simple <- contract(g, cl$membership, vertex.attr.comb = list(weight = "sum",
                  name = function(x)x[1], "ignore"))
```

Figure 2.9

We extract an induced subgraph from the original graph based on the weight of vertices. We select vertices with weights more than 30. It is done by using the induced.subgraph function provided by the igraph library. We simplified the edges of the graph by removing multiple edges between the same vertices. This is done by using simplify() function from the igraph library. Figure 2.10 shows the next part of the code to obtain the final simplified graph, to plot it and to print a histogram of the graph nodes' degree.

```
mygraph_simplified <- induced_subgraph(mygraph_simple, V(mygraph_simple)$weight > 30)
V(mygraph_simplified)$degree <- unname(igraph::degree(mygraph_simplified))
mygraph_simplified <- simplify(mygraph_simplified)
plot(mygraph_simplified, vertex.size = 14, edge.arrow.size=0.1)

hist(igraph::degree(mygraph_simplified))
```

Figure 2.10

Figure 2.11 shows the plotted graph.

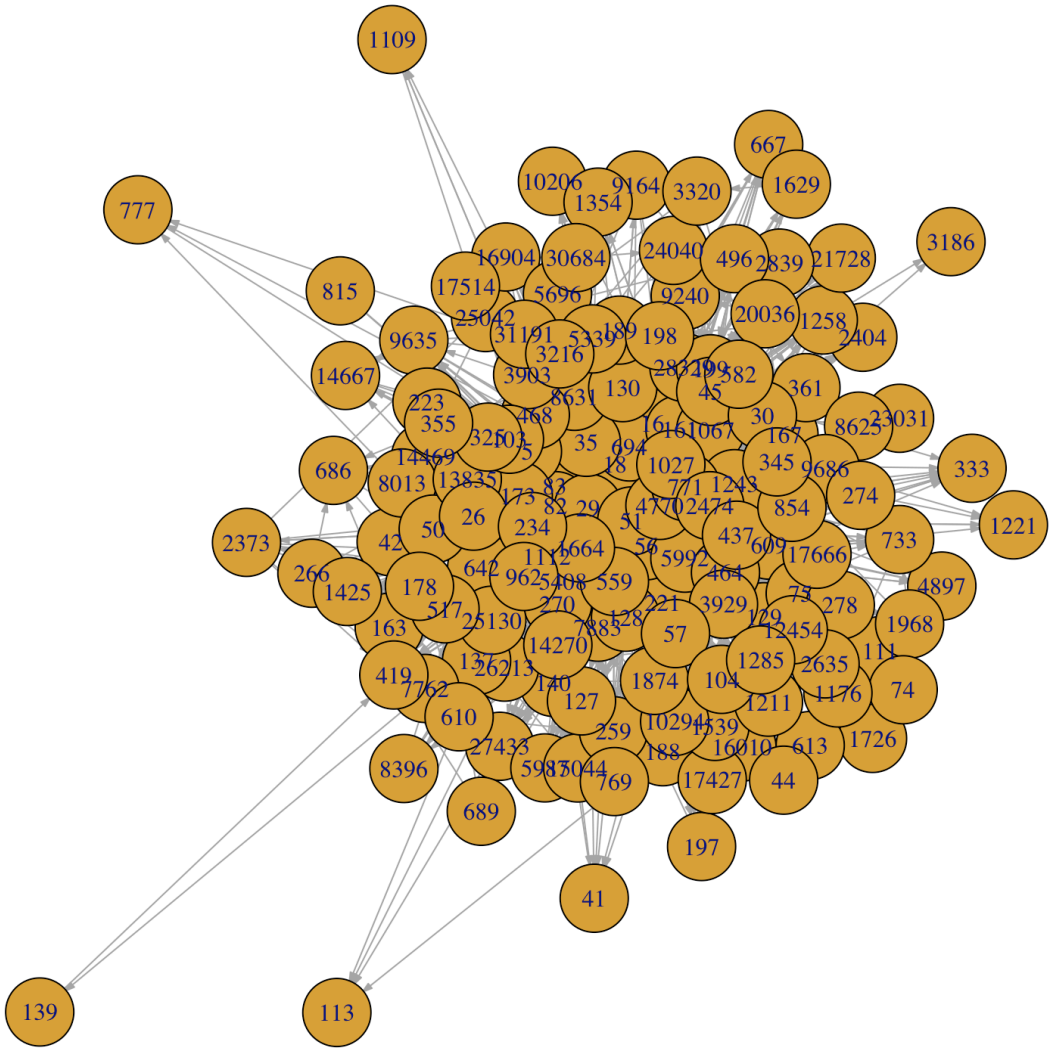


Figure 2.11

The degree distribution, shown by hist() function below, gives frequency of the number of nodes with specific degree. Figure 2.12 shows the histogram of the simplified graph nodes' degrees.

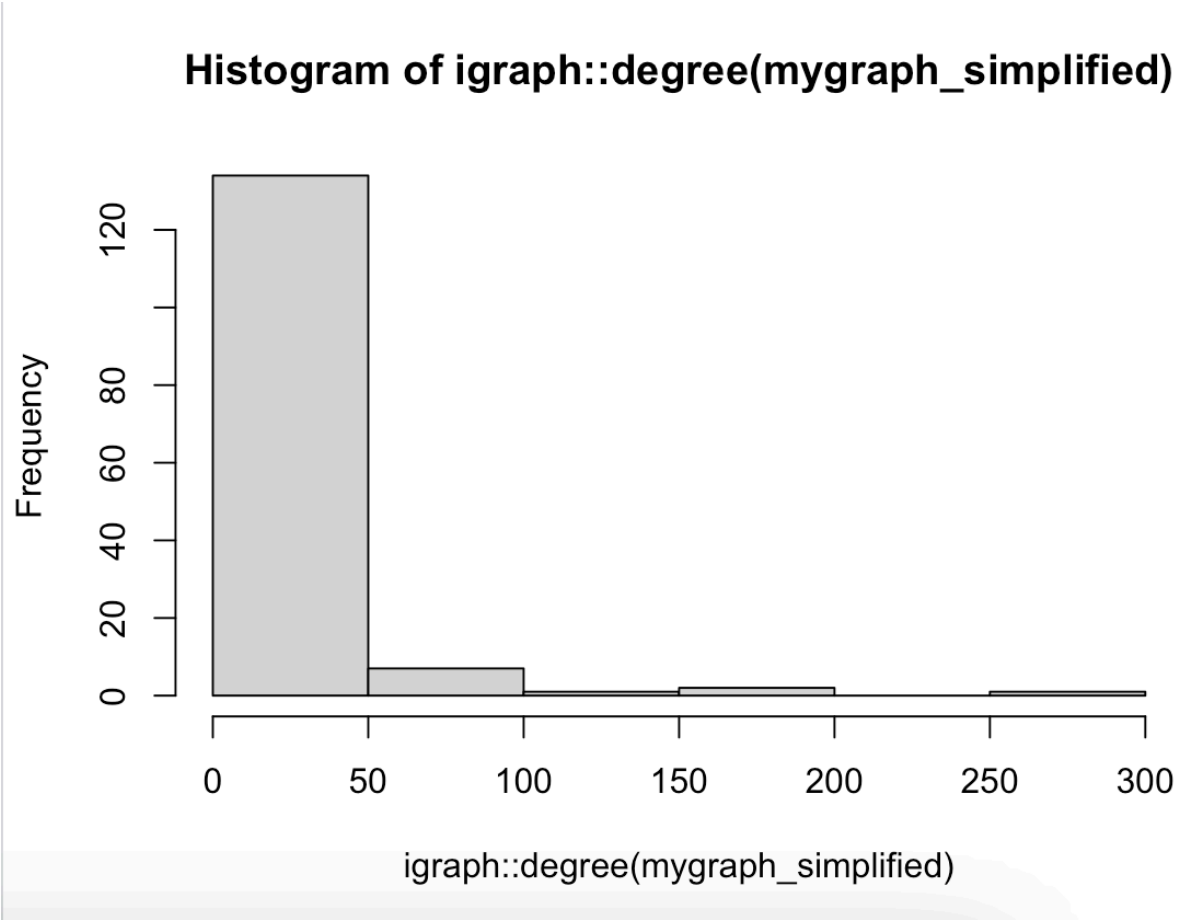


Figure 2.12

Demonstrations for question #3

Vertices of the graph: V()

This function calculates the total number of nodes present in the graph. We tested this function with the 3rd version of the graph. As illustrated in Figure 3.1, once the graph has been simplified, it contains a total of 145 vertices.

```
> V(mygraph_simplified)
+ 145/145 vertices, named, from c56ef45:
  [1] 83    1112  82    16    221   18    29    609   694   56    51    35    8631  5992  140
 [16] 130   161   1067  129   7883  75    25042 464   4770  1243  16904 128   771   167   1027
 [31] 28329 5408  9635  2474  1726  361   5     1221  9240  189   139   41    437   197   686
 [46] 199   777   270   642   333   188   26213 27433 45    42    2404  173   689   559   9164
 [61] 2373  23031 10206 667   111   1109  4897  24040 733   815   613   16010 3320  8625  17514
 [76] 17427 5987  1258  14667 21728 1968  5696  266   74    2839  1176  1354  5339  15044 163
 [91] 1539  278   1211  1629  137   44    1664  259   3929  9686  30684 14469 17666 10294 1425
[106] 2635  7762  1874  25130 468   127   8013  274   50    496   30    8396  198   610   854
[121] 57    517   104   3903  31191 12454 178   223   20036 3216  13835 103   769   419   345
[136] 26    325   1285  355   582   14270 234   962   3186  113
>
```

Figure 3.1

Edges of the graph: E()

This function prints all edges of the graph and calculates the total number of edges connecting the nodes. As illustrated in Figure 3.2, our simplified graph contains 1756 edges.

```
> E(mygraph_simplified)
+ 1756/1756 edges from 3fb6864 (vertex names):
 [1] 83  ->1112  83  ->82    83  ->221   83  ->18    83  ->29    83  ->609   83  ->56    83  ->51    83  ->35
[10] 83  ->5992  83  ->161   83  ->1067  83  ->7883  83  ->464   83  ->4770  83  ->1243  83  ->128   83  ->1027
[19] 83  ->28329 83  ->9635  83  ->5     83  ->270   83  ->642   83  ->45    83  ->42    83  ->173   83  ->559
[28] 83  ->14667 83  ->278   83  ->14469 83  ->1425  83  ->25130 83  ->50    83  ->57    83  ->517   83  ->12454
[37] 83  ->223   83  ->20036 83  ->13835 83  ->103   83  ->26    83  ->325   83  ->1285  83  ->355   83  ->234
[46] 83  ->3186  83  ->113   1112->83    1112->82    1112->16    1112->18    1112->29    1112->694   1112->56
[55] 1112->51    1112->35    1112->5992  1112->161   1112->129   1112->7883  1112->4770  1112->5     1112->270
[64] 1112->642   1112->333   1112->26213 1112->27433 1112->2404  1112->173   1112->559   1112->16010 1112->163
[73] 1112->137   1112->9686  1112->1425  1112->7762  1112->25130 1112->127   1112->8013  1112->610   1112->517
[82] 1112->178   1112->20036 1112->103   1112->419   1112->325   1112->1285  1112->14270 1112->234   1112->962
+ ... omitted several edges
>
```

Figure 3.2

Adjacency matrix: as_adjacency_matrix()

This function is designed to generate the adjacency matrix of a graph. In the matrix, a "1" indicates that there is an edge connecting two nodes. On the other hand, a "." (dot) signifies that there is no edge between the nodes, which is equivalent to a "0". This representation helps visualize the connections within the graph. Figure 3.3 shows the adjacency of our 3rd simplified graph.

```
> mygraph_simplified.adj = igraph::as_adjacency_matrix(mygraph_simplified)
> mygraph_simplified.adj
145 x 145 sparse Matrix of class "dgCMatrix"
  [[ suppressing 49 column names ‘83’, ‘1112’, ‘82’ ... ]]
  [[ suppressing 49 column names ‘83’, ‘1112’, ‘82’ ... ]]

83   . 1 1 . 1 1 1 1 . 1 1 1 . 1 . . 1 1 . 1 . . 1 1 1 . 1 . . 1 1 . 1 . . . 1 . . . . . . . . . . 1 1 ......
1112 1 . 1 1 . 1 1 . 1 1 1 1 . 1 . . 1 . 1 1 . . . 1 . . . . . . . . . . . . 1 . . . . . . . . . . 1 1 ......
82   1 1 . 1 . 1 1 . . 1 1 1 . 1 . . 1 . . 1 . . 1 . 1 . 1 . . 1 . . . . . . 1 . . . . . . . . . . 1 1 ......
16   . . 1 . . 1 1 . . . 1 1 . 1 . . 1 . . . . . . . . . . . . . 1 1 . . . . 1 . . . . . . . . 1 . . . ......
221  . . . . . . . . 1 . 1 . . 1 . . . . . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . . . . . ......
18   1 1 1 1 1 . 1 1 1 1 1 1 1 1 . 1 1 1 . . 1 1 . 1 1 . . 1 . 1 1 1 . 1 . . 1 . . . . 1 1 . . 1 1 1 1 ......
29   1 1 1 1 . 1 . . . 1 1 1 . 1 1 1 1 . . 1 . . 1 . 1 . 1 . . . 1 . . 1 . 1 1 . 1 1 . 1 1 . . . . 1 1 ......
609  1 . . . . 1 1 . . . 1 1 . 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
694  . 1 . . . 1 1 . . 1 1 1 . 1 . . 1 1 . . . 1 . 1 . . . . . 1 . . . . . . 1 1 . . . . . . 1 . . . 1 ......
56   1 1 . . . 1 1 . 1 . 1 1 . 1 . . 1 . . . . . 1 1 . . 1 . . . . 1 . . . . 1 . . . . . 1 . . . . . 1 ......

 ..............................
 ........suppressing 96 columns and 125 rows in show(); maybe adjust 'options(max.print= *, width = *)'
 ..............................
  [[ suppressing 49 column names ‘83’, ‘1112’, ‘82’ ... ]]

26    1 . 1 . . . 1 . . . 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 . . . . . . ......
325   1 . 1 . . 1 1 . . . 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . . ......
1285  . . . . . 1 . . . . 1 . . 1 . . . . 1 . 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
355   1 . 1 . . . 1 . . 1 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . . ......
582   . . . 1 . . 1 . . . 1 . . 1 . . 1 . . . . . . 1 . . . . . . 1 . . . . . 1 . . . . . . . . 1 . . . ......
14270 . 1 . . . . . . . . . 1 1 . 1 . . . . . . . . . 1 . . . . . . . . . . . . . . . . . . . . . . 1 . ......
234   . 1 1 1 . . 1 . . . 1 . . 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1 ......
962   1 1 1 . . 1 1 . 1 . 1 . . 1 . . . . . . . . . . . . . 1 . . . 1 . . . . . . . . . . . . . . . 1 . 1 1 ......
3186  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
113   . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ......
>
```

Figure 3.3

Edge density: edge_density()

By analyzing the edge density, we can understand how tightly the graph is connected. A lower edge density indicates that the graph is not very strongly connected, while a higher density suggests a more tightly bound structure. As illustrated in Figure 3.4, the graph's edge densities with and without loops are relatively low, which means the connections within the graph are not particularly strong or dense.

```
> igraph::edge_density(mygraph_simplified)
[1] 0.08409962
> igraph::edge_density(mygraph_simplified, loops = T)
[1] 0.08351962
>
```

Figure 3.4

Graph's density: gden()

Graph's density is a way to measure how connected a given graph is. It is calculated by dividing the number of edges in the graph by the maximum number of edges that could possibly exist in that graph. The density value always falls between 0 and 1. A density of 0 means the graph has no edges at all - it's completely disconnected. On the other hand, a density of 1 means the graph is fully connected, with every possible edge present. Figure 3.5 depicts that our graph's density is only 0.084, which is quite low - our graph is more disconnected.

```
> mygraph_simplified_matrix <- as.matrix(mygraph_simplified.adj)
> sna::gden(mygraph_simplified_matrix)
[1] 0.08409962
>
```

Figure 3.5

Degree of the graph's nodes: degree()

Determines the number of edges connected to a specific node. Below, on figure 3.6, we can see that nodes are in the form of a table and below each node's id lies its associative degree. For example, the node 83's degree is 93.

```
> # Degree of the graph's nodes
> degree <- igraph::degree(mygraph_simplified)
> degree
   83  1112    82    16   221    18    29   609   694    56    51    35  8631  5992   140   130   161  1067
   93    82    64    42    15   150   154    14    47    59   262    41    18   193    15    17    96    26
  129  7883    75 25042   464  4770  1243 16904   128   771   167  1027 28329  5408  9635  2474  1726   361
   20    25    16    12    33    59    38     9    29    24    12    35    29    18    18    31    10    14
    5  1221  9240   189   139    41   437   197   686   199   777   270   642   333   188 26213 27433    45
   52     5    27    22     2     7    23    10    10    39     4    40    30    10    16    23    14    26
   42  2404   173   689   559  9164  2373 23031 10206   667   111  1109  4897 24040   733   815   613 16010
   19     6    43     7    34     8    10     4     5    10    12     4     6    13    12     2    12    13
 3320  8625 17514 17427  5987  1258 14667 21728  1968  5696   266    74  2839  1176  1354  5339 15044   163
   14     7     6    13    10    15    12    10    10    11    12    10    12     9     9    15    20    16
 1539   278  1211  1629   137    44  1664   259  3929  9686 30684 14469 17666 10294  1425  2635  7762  1874
   10    18    11     7    20    15    25    20    19    14     5    21    11    11    13    10    16     8
25130   468   127  8013   274    50   496    30  8396   198   610   854    57   517   104  3903 31191 12454
   20    16    19    12     7    17    11    12     8    20    15    16    25    24    13    10    11    12
  178   223 20036  3216 13835   103   769   419   345    26   325  1285   355   582 14270   234   962  3186
   17    17    21    26    15    27    11    23    16    19    28    15    19    22    27    27    43     2
  113
    4
>
```

Figure 3.6

Histogram of the nodes' degrees: hist(igraph::degree())

The histogram shows how the degrees of the vertices are distributed in a graph. On the x-axis, we can see different degree values (here this number goes up to 300), while the y-axis represents the approximate number of nodes with corresponding degree value. From the histogram, we can see that very few nodes have a degree of more than 50 - more than 120 nodes (of total 145, as we saw by running V() function) have degree values between 0 and 50. Which confirms the result of the low graph's density, previously obtained.

Figure 3.7 shows the command used and 3.8 - the histogram.

```
> hist(igraph::degree(mygraph_simplified))
>
```

Figure 3.7

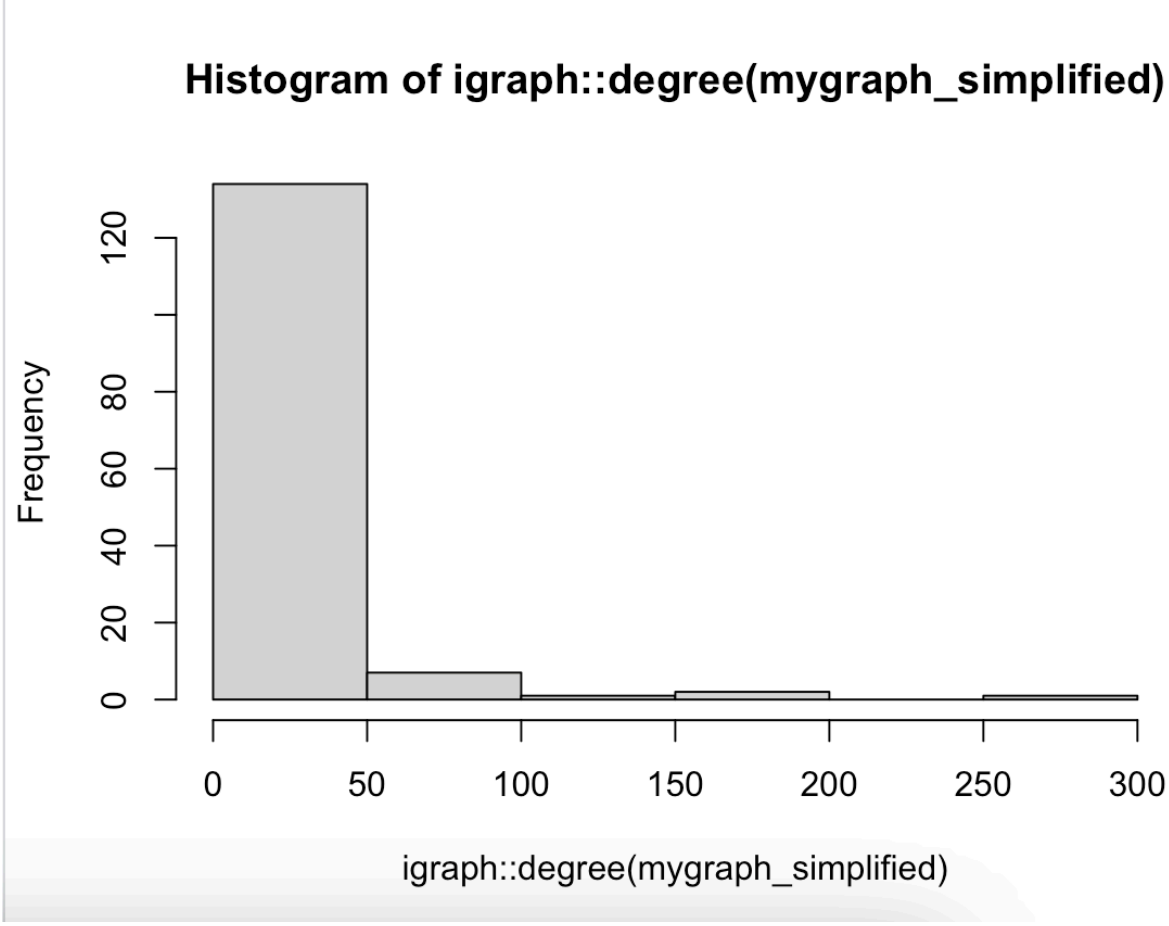


Figure 3.8

Betweenness Centrality: centr_betw()

The function calculates the betweenness centrality for nodes in a graph. Betweenness centrality is a way to measure how important a node is based on how many shortest paths in the network pass through it.

The $centralization value gives the network centralization index based on betweenness centrality. This index shows how much the most central node stands out compared to the rest of the nodes in the network. In this case, the value is 0.4125784, which is close to the half, as the centralization index can range from 0 (meaning no centralization, where all nodes are equally central) to 1 (meaning maximum centralization, where one node dominates the network).

The $theoretical_max value provides the highest possible betweenness centrality score that any node could have in a graph of the same size. Here this value is equal to 2965248.

The figure 3.9 below shows this function applied with our 3rd version of graph simplification.

```
> hist(igraph::degree(mygraph_simplified))
> centr_betw(mygraph_simplified)
$res
  [1]  496.4194620  378.8661642  194.1079503  110.6731815    4.0103535 2018.3723942 1881.4871095    6.0782107
  [9]  140.0470043  157.0141549 8576.3820483   59.9293816    6.3115079 3323.2965575    3.5662707    9.4791934
 [17]  705.5349916   20.1693888   13.8992063   22.3536454   15.1634920    1.1964646   40.9124104  181.3159397
 [25]  182.3866840    0.4000000   17.8603590   27.2294661    1.8210458   54.0506858   15.0993275    7.1366162
 [33]    4.8563492   37.6624111    3.4305556    3.9788233  208.5270612    0.0000000   25.0842795   10.5518562
 [41]    0.0000000    0.2500000   11.3240654    2.2539683    3.9261472   70.7897869    0.0000000   60.6079326
 [49]   48.5172601    0.0000000    6.4488418   23.2551948    1.2587662   18.8215193    7.5746454    0.4916667
 [57]   56.9793873    0.3928571   24.4532981    3.5609320   11.0827292    0.0000000    0.0000000    1.4063492
 [65]    3.4313242    0.0000000    0.2148148    5.4833084    3.6345238    0.0000000    3.2357143    4.8690476
 [73]    9.7030265    0.9000000    0.0000000    0.6602564    1.2954013    2.8044012    0.4384615    0.7873016
 [81]    8.0970423    2.7365440    1.9050208    1.5666667    0.6464646    1.0366013    1.4982864  145.0563242
 [89]   10.6787369    5.0250111    1.1666667    9.9762102    1.3428571    0.3771737   13.2437946   18.2394483
 [97]   19.0460746   10.5618717    7.2862193    3.8196673    0.0000000    7.6008731    0.4500000    2.7781746
[105]    1.4956349    0.0000000    8.6603609    0.4761905    3.2797980    4.5178571    6.5808177    1.6013029
[113]    0.0000000   14.2348278    0.8107504    5.4532828    0.1114286   24.8862890    3.8596972   18.4147267
[121]   14.1275465   22.9492036    5.8434676    0.0000000    2.3944444    1.8281441    4.0548341    2.9541490
[129]   10.5877345   27.8299783    0.3333333   20.5178769    2.4861111  151.6685967    5.2381063    3.4456044
[137]   33.2362222   19.5810840    3.7510678    6.9807974   30.7210040    9.9440948   71.4929717    0.0000000
[145]    0.0000000

$centralization
[1] 0.4125784

$theoretical_max
[1] 2965248
```

Figure 3.9

Closeness Centrality: centr_clo()

The function calculates the closeness centrality of all nodes in a graph. This metric measures how close a node is, on average, to all other nodes in the graph. It is calculated based on the average length of the shortest paths from that node to every other node. On figure 3.10 you can see the application of this function.

The closeness centralization ($centralization) of the network is 0.9269637 (out of 1). This value reflects how unevenly closeness centrality is distributed across the network. A higher centralization score indicates that the network has a more uneven distribution of closeness centrality, meaning some nodes are much closer to others compared to the rest.

The theoretical maximum closeness centrality ($theoretical_max) for a graph of this size and structure is 66.74906. This represents the highest possible closeness centrality score that any node could achieve in a graph with the same number of nodes and connections.

```
> centr_clo(graph_new)
$res
  [1] 0.6473430 0.6380952 0.5929204 0.5630252 0.5214008 0.7701149
  [7] 0.7882353 0.5173745 0.5702128 0.5955556 1.0000000 0.5630252
 [13] 0.5296443 0.8645161 0.5214008 0.5275591 0.6536585 0.5381526
 [19] 0.5338645 0.5425101 0.5254902 0.5193798 0.5491803 0.5955556
 [25] 0.5583333 0.5153846 0.5403226 0.5360000 0.5173745 0.5560166
 [31] 0.5469388 0.5296443 0.5296443 0.5514403 0.5173745 0.5254902
 [37] 0.6008969 0.5403226 0.5360000 0.5134100 0.5360000 0.5153846
 [43] 0.5173745 0.5677966 0.5583333 0.5403226 0.5173745 0.5254902
 [49] 0.5338645 0.5214008 0.5447154 0.5275591 0.5095057 0.5606695
 [55] 0.5095057 0.5537190 0.5134100 0.5173745 0.5153846 0.5173745
 [61] 0.5095057 0.5173745 0.5193798 0.5173745 0.5214008 0.5214008
 [67] 0.5095057 0.5214008 0.5173745 0.5214008 0.5173745 0.5134100
 [73] 0.5153846 0.5173745 0.5193798 0.5153846 0.5173745 0.5153846
 [79] 0.5134100 0.5254902 0.5275591 0.5214008 0.5134100 0.5275591
 [85] 0.5134100 0.5114504 0.5317460 0.5214008 0.5360000 0.5275591
 [91] 0.5317460 0.5214008 0.5338645 0.5153846 0.5134100 0.5214008
 [97] 0.5153846 0.5254902 0.5114504 0.5275591 0.5234375 0.5254902
[103] 0.5193798 0.5095057 0.5296443 0.5153846 0.5173745 0.5134100
[109] 0.5317460 0.5234375 0.5254902 0.5360000 0.5360000 0.5193798
[115] 0.5173745 0.5173745 0.5173745 0.5234375 0.5214008 0.5317460
[121] 0.5381526 0.5214008 0.5403226 0.5193798 0.5360000 0.5254902
[127] 0.5296443 0.5403226 0.5254902 0.5275591 0.5296443 0.5425101
[133] 0.5403226 0.5677966 0.5075758

$centralization
[1] 0.9269637

$theoretical_max
[1] 66.74906
```

Figure 3.10

Graph simplification: is_sample()

This function is designed to check whether the graph contains any multiple edges (more than one edge connecting the same pair of vertices) or loops (an edge that connects a vertex to itself). Here is the result applied on the 3rd simplified graph (figure 3.11).

```
> is_simple(mygraph_simplified)
[1] TRUE
>
```

Figure 3.11

If this function yields FALSE, we can run the simplify() function to render the graph simple (as we did at the beginning when we simplified the 3rd graph).

Geodesic: geodist()

A geodesic is the shortest path connecting any two nodes in a network in the number of edges. In other words, it is the path with the least number of edges connecting them. The figure 3.12 shows the result of application of this function on our graph.

```
> mygraph_simplified.edge_list <- as_edgelist(mygraph_simplified)
> mygraph_simplified.geo <- geodist(mygraph_simplified.edge_list)
> mygraph_simplified.geo
$counts
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14] [,15] [,16] [,17] [,18]
[,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35]
[,36] [,37] [,38] [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46] [,47] [,48] [,49] [,50] [,51] [,52]
[,53] [,54] [,55] [,56] [,57] [,58] [,59] [,60] [,61] [,62] [,63] [,64] [,65] [,66] [,67] [,68] [,69]
[,70] [,71] [,72] [,73] [,74] [,75] [,76] [,77] [,78] [,79] [,80] [,81] [,82] [,83] [,84] [,85] [,86]
[,87] [,88] [,89] [,90] [,91] [,92] [,93] [,94] [,95] [,96] [,97] [,98] [,99] [,100] [,101] [,102]
[,103] [,104] [,105] [,106] [,107] [,108] [,109] [,110] [,111] [,112] [,113] [,114] [,115] [,116]
[,117] [,118] [,119] [,120] [,121] [,122] [,123] [,124] [,125] [,126] [,127] [,128] [,129] [,130]
[,131] [,132] [,133] [,134] [,135] [,136] [,137] [,138] [,139] [,140] [,141] [,142] [,143] [,144]
[,145] [,146] [,147] [,148] [,149] [,150] [,151] [,152] [,153] [,154] [,155] [,156] [,157] [,158]
[,159] [,160] [,161] [,162] [,163] [,164] [,165] [,166] [,167] [,168] [,169] [,170] [,171] [,172]
[,173] [,174] [,175] [,176] [,177] [,178] [,179] [,180] [,181] [,182] [,183] [,184] [,185] [,186]
[,187] [,188] [,189] [,190] [,191] [,192] [,193] [,194] [,195] [,196] [,197] [,198] [,199] [,200]
[,201] [,202] [,203] [,204] [,205] [,206] [,207] [,208] [,209] [,210] [,211] [,212] [,213] [,214]
[,215] [,216] [,217] [,218] [,219] [,220] [,221] [,222] [,223] [,224] [,225] [,226] [,227] [,228]
[,229] [,230] [,231] [,232] [,233] [,234] [,235] [,236] [,237] [,238] [,239] [,240] [,241] [,242]
[,243] [,244] [,245] [,246] [,247] [,248] [,249] [,250] [,251] [,252] [,253] [,254] [,255] [,256]
[,257] [,258] [,259] [,260] [,261] [,262] [,263] [,264] [,265] [,266] [,267] [,268] [,269] [,270]
[,271] [,272] [,273] [,274] [,275] [,276] [,277] [,278] [,279] [,280] [,281] [,282] [,283] [,284]
[,285] [,286] [,287] [,288] [,289] [,290] [,291] [,292] [,293] [,294] [,295] [,296] [,297] [,298]
[,299] [,300] [,301] [,302] [,303] [,304] [,305] [,306] [,307] [,308] [,309] [,310] [,311] [,312]
[,313] [,314] [,315] [,316] [,317] [,318] [,319] [,320] [,321] [,322] [,323] [,324] [,325] [,326]
[,327] [,328] [,329] [,330] [,331] [,332] [,333] [,334] [,335] [,336] [,337] [,338] [,339] [,340]
[,341] [,342] [,343] [,344] [,345] [,346] [,347] [,348] [,349] [,350] [,351] [,352] [,353] [,354]
[,355] [,356] [,357] [,358] [,359] [,360] [,361] [,362] [,363] [,364] [,365] [,366] [,367] [,368]
[,369] [,370] [,371] [,372] [,373] [,374] [,375] [,376] [,377] [,378] [,379] [,380] [,381] [,382]
[,383] [,384] [,385] [,386] [,387] [,388] [,389] [,390] [,391] [,392] [,393] [,394] [,395] [,396]
[,397] [,398] [,399] [,400] [,401] [,402] [,403] [,404] [,405] [,406] [,407] [,408] [,409] [,410]
[,411] [,412] [,413] [,414] [,415] [,416] [,417] [,418] [,419] [,420] [,421] [,422] [,423] [,424]
[,425] [,426] [,427] [,428] [,429] [,430] [,431] [,432] [,433] [,434] [,435] [,436] [,437] [,438]
[,439] [,440] [,441] [,442] [,443] [,444] [,445] [,446] [,447] [,448] [,449] [,450] [,451] [,452]
[,453] [,454] [,455] [,456] [,457] [,458] [,459] [,460] [,461] [,462] [,463] [,464] [,465] [,466]
[,467] [,468] [,469] [,470] [,471] [,472] [,473] [,474] [,475] [,476] [,477] [,478] [,479] [,480]
```

Figure 3.12

Demonstrations for question #4

a) In lecture 2, we have seen four different centrality metrics for nodes. Let us present all of them and compute the central node in the graph according to which centrality measurement is chosen.

These four centrality metrics are the following:

- Degree centrality. In this case, the node with the highest number of connections is considered as the most central. For this metric we studied two cases : when all neighbors of a given node are counted (successors and predecessors) and when only successors are taken into account. In fact, according to lecture 2 these two ways are possible, of course the first option is more fore undirected graph, but for better analysis we have decided to do both ways;

- Eigenvector centrality. Here a node in a graph is more central if it is connected to other important nodes. In other words, this centrality is computed based on the node's connection to other nodes and their centrality. This measure can help to measure indirect high influence within a given graph.

- Betweenness centrality. This measure is based on the number of shortest paths that go through a given node. This provides information on how often a node lies on the shortest paths between other nodes.

- Closeness centrality. This centrality metric of a node is calculated as the inverse of the sum of the shortest paths from this node to all others. Therefore, it measures how close a node is to all other nodes in a graph. The closer a node is to all other nodes, the higher its closeness centrality is going to be.

To find each the most central node according to each of these centrality measurements we have used pre-implemented functions of the igraph package. On the next screenshot (figure 4.1) you can see the results of these functions.

```
> # ------------------------------
> # a - Central node in the graph
> # ------------------------------
>
> # Degree centrality counting all neighbors
> degree_number = igraph::degree(g, mode = "total")
> sort(degree_number)[length(degree_number)]
102
623
>
> # Degree centrality counting only successors
> degree_number = igraph::degree(g, mode = "out")
> sort(degree_number)[length(degree_number)]
14648
  145
>
> # Eigenvector centrality
> eigen_centrality <- igraph::eigen_centrality(g)
> highest_node <- which.max(eigen_centrality$vector)
> highest_node
486
500
>
> # Betweenneess centrality
> betweenness_scores <- igraph::betweenness(g)
> highest_betweenness_node <- which.max(betweenness_scores)
> highest_betweenness_node
622
194
>
> # Closeness centrality
> closeness_scores <- igraph::closeness(g)
> highest_closeness_node <- which.max(closeness_scores)
> highest_closeness_node
51
 3
```

Figure 4.1

For the first metric, in the case of high total degree, we can see that the most central node is 102 and has 623 neighbors. Considering that the graph represents email letter circulation, this could mean that this node represents a highly connected individual which is often involved in different conversations.

In the case of out - total degree, the node with id number 14648 is the most central with a total of 145 successors. Here, we can interpret it as a key broadcaster in the considered network.

Next, according to the second centrality metric, the node number 486 is the most central with the score equal to 500. This shows that this node has hidden high influence within a given graph.

Furthermore, for the third measure, we observe that the most central node is number 622 with the score equal to 194. As this measures how often a node lies on the shortest path between other nodes, this shows that the absence of this individual could result in disruption of the communication flow.

Finally, the last metric shows that the node number 51 with score 3 is the most central. In the email conversation context, this node quickly spreads the information through the given network.

b) To compute the longest path(s) in our graph, first we have to compute the diameter of our graph. The diameter, by definition in graph theory, is the longest shortest path between two distinct nodes in the graph. Thus, after having calculated the diameter, we just have to compute the values of all shortest paths between each pair of nodes and then find those with the value of diameter. Finally, we are going to store all these pairs of modes and then get the entire paths.

In this case we could not directly work with the initial graph, because of its complexity. Thus, we have decided to take the simplified graph number 2, this simplification was made based on the number of node's neighbors.

To compute the diameter we used a specific function of the igraph package : diameter(). This function takes into parameters the graph, whose diameter we need to compute. Then, we have computed all shortest distances between each pair of nodes thanks to another function provided by this package called distances() which as well takes into parameters an igraph object and returns all shortest distances of the graph in the form of a matrix.

After that, we have to check all matrice's values and find the longest distances. This can simply be done with a for loop, and at the same time we are saving each pair of nodes' ids in the list longest_distances_nodes.

Finally, we can get the actual paths by running a pre-built function get.shortest.paths(). This function takes into parameters the graph and two vertices : the starting and the ending.

Below (Figure 4.2), you can find the screenshot of the entire code to find the longest path.

```
# ----------------------
# b - Longest path(s)
# ----------------------

# Calculating the graph's diameter
g_simplified.d <- igraph::diameter(g_simplified)

# Calculating the matrix of shortest paths
g_simplified.sp <- igraph::distances(g_simplified)

# Finding each pair of nodes between which the shortest distance is the diameter
longest_distances_nodes <- list()
for (i in 1:nrow(g_simplified.sp)) {
  for (j in 1:ncol(g_simplified.sp)) {
    if (g_simplified.sp[i, j] == g_simplified.d - 1) {
      longest_distances_nodes <- append(longest_distances_nodes,
                    list(c(V(g_simplified)[i]$name, V(g_simplified)[j]$name)))
    }
  }
}

# The actual path between some pairs of vertices found
for (i in c(1:2)){
  pair = longest_distances_nodes[[i]]
  end_node = V(g_simplified)[name == pair[1]]
  start_node = V(g_simplified)[name == pair[2]]
  g_simplified.spvalues <- igraph::get.shortest.paths(graph = g_simplified,
                    from = start_node, to = end_node)
  g_simplified.spvalues <- g_simplified.spvalues$vpath[[1]]
  cat("Path number", i, ":\n")
  print(g_simplified.spvalues)
  cat("\n")
}
```

Figure 4.2

We can find the number of longest paths by executing the following command : length(longest_distances_nodes) which will print the length of the list (where each element is a pair of nodes whose shortest distance is the diameter of the graph) that we have found by the algorithm.

Below (figure 4.3), you can find examples of such paths, and as you can see the longest paths are composed of only 4 nodes.

```
> # The actual path between some pairs of vertices found
> for (i in c(1:2)){
+   pair = longest_distances_nodes[[i]]
+   end_node = V(g_simplified)[name == pair[1]]
+   start_node = V(g_simplified)[name == pair[2]]
+   g_simplified.spvalues <- igraph::get.shortest.paths(graph = g_simplified,
+                     from = start_node, to = end_node)
+   g_simplified.spvalues <- g_simplified.spvalues$vpath[[1]]
+   cat("Path number", i, ":\n")
+   print(g_simplified.spvalues)
+   cat("\n")
+ }
Path number 1 :
+ 4/35 vertices, named, from 6fb1b38:
[1] 802 486 87  83

Path number 2 :
+ 4/35 vertices, named, from 6fb1b38:
[1] 1159 255  87   83
```

Figure 4.3

Thus, in the context of email conversations, in the simplified network the information is quickly transferred as its the longest is going only through 3 intermediate individuals in the worst case.

c) To be able to find the largest clique(s), the igraph package provides the function called largest_cliques(), which takes as parameters the graph in question. This function returns all largest cliques of a given graph.

We can also obtain information about the size of the largest clique(s) thanks to another pre-built function : clique_num(), which takes into parameters the variable containing the igraph object.

The next screenshot (figure 4.4) captures the entire code for this part.

```
# ------------------------
# c - Largest clique(s)
# ------------------------

# Finding the size of largest clique(s)
g.largest_clique_size = igraph::clique_num(g)
g.largest_clique_size # maximum clique's size is 12

# Finding all largest cliques
g.largest_cliques <- igraph::largest_cliques(g)

length(g.largest_cliques) # 24 largest cliques
g.largest_cliques # their list

# plotting some of them
largest_clique_nodes <- g.largest_cliques[[1]]
largest_clique_subgraph <- igraph::induced_subgraph(g, largest_clique_nodes)
plot(largest_clique_subgraph, vertex.color = "lightgreen", edge.color = "black",
     vertex.size = 30, edge.arrow.size = 0.5)

largest_clique_nodes <- g.largest_cliques[[5]]
largest_clique_subgraph <- igraph::induced_subgraph(g, largest_clique_nodes)
plot(largest_clique_subgraph, vertex.color = "pink", edge.color = "black",
     vertex.size = 30, edge.arrow.size = 0.5)

largest_clique_nodes <- g.largest_cliques[[10]]
largest_clique_subgraph <- igraph::induced_subgraph(g, largest_clique_nodes)
plot(largest_clique_subgraph, vertex.color = "darkslategray1", edge.color = "black",
     vertex.size = 30, edge.arrow.size = 0.5)
```

Figure 4.4

Below (figure 4.5) you can find the result of this code execution.

After having executed these commands we found out that the size of the graph's largest cliques is only 12 nodes. We also discovered that there are 24 different largest cliques in the entire graph.

There is a warning indicating that the edge directions are ignored as the term clique is only related to an undirected graph, which is not the case in our graph. But here, we assume that we had to find a tournament - a directed graph where any two vertices are joined by exactly one edge (we do not take care of the edges' direction). This is one of the equivalents of a clique in a directed graph.

```
> # -----------------------
> # c - Largest clique(s)
> # -----------------------
>
> # Finding the size of largest clique(s)
> g.largest_clique_size = igraph::clique_num(g)
Warning message:
In igraph::clique_num(g) :
  At vendor/cigraph/src/cliques/maximal_cliques_template.h:220 : Edge directions are ignored for maximal clique
calculation.
> g.largest_clique_size # maximum clique's size is 12
[1] 12
>
> # Finding all largest cliques
> g.largest_cliques <- igraph::largest_cliques(g)
Warning message:
In igraph::largest_cliques(g) :
  At vendor/cigraph/src/cliques/maximal_cliques_template.h:220 : Edge directions are ignored for maximal clique
calculation.
>
> length(g.largest_cliques) # 24 largest cliques
[1] 24
> g.largest_cliques # their list
[[1]]
+ 12/32430 vertices, named, from 115a0d0:
 [1] 6712  29    12887 83    102   1023  174   502   325   223   681   173

[[2]]
+ 12/32430 vertices, named, from 115a0d0:
 [1] 223   174   325   502   83    82    102   12887 1023  1022  29    8013

[[3]]
+ 12/32430 vertices, named, from 115a0d0:
 [1] 223   174   325   502   83    82    102   12887 1023  1022  29    173
```

Figure 4.5

Furthermore, to have a better representation of the found cliques, we plotted some of them.

Below (figure 4.6, 4.7 and 4.8) are three random largest cliques (first, fifth and tenth of the list of 24 cliques) are represented.

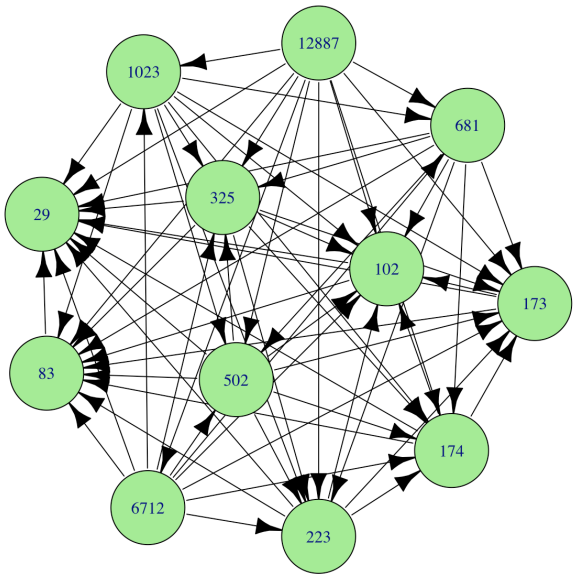


Figure 4.6

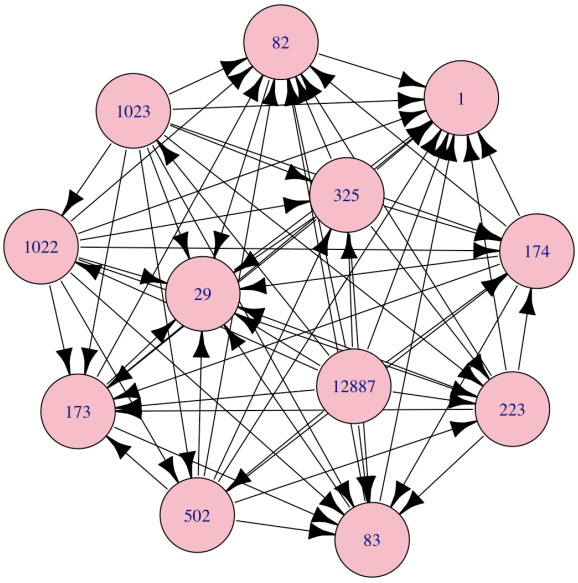


Figure 4.6

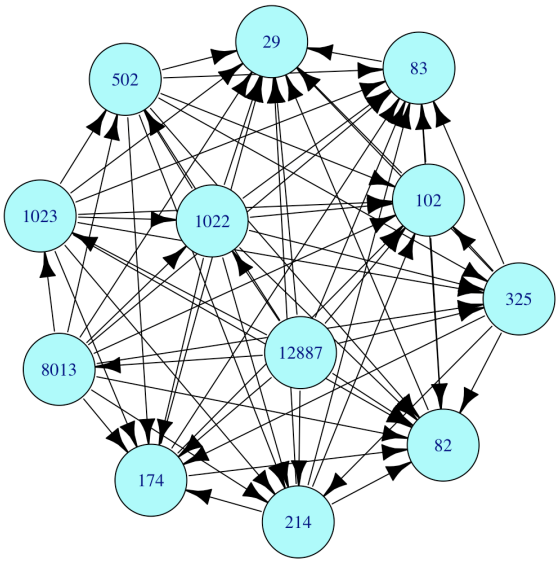


Figure 4.7

This reveals groups of individuals that intensively communicate with each other via email. This could reveal members of a group work where between each couple of members was an interaction.

d) The "egocentric network" of a node is the subset of the graph including the “root” node and all the nodes directly connected to it via outgoing edges.

Hence, to find the egocentric network(s), we used the function ego() of the igraph package. This function takes into parameters the graph whose egos to calculate, as well as the variable mode which indicates which type of neighbors to consider. In our case, the variable containing the initial graph is g and we are concentrating only on successors, thus, mode = “out”.

After that, we used sapply() function to find the size of the largest ego(s), then to find these/ this ego(s) and store the result in the list called ego_nodes, which will contain the “root” node(s) of the largest ego(s) of the graph.

Finally, we plotted the first largest ego found. To make this ego representation more clear, we plotted only the egocentric network of the node. In this graph, we colored the source node in magenta and its out-going edges in blue, whereas other nodes are in green and edges - gray.

Here is the complete code (figure 4.8):

```
# ------------
# d - Ego(s)
# ------------

# Finding the ego(s) of the given graph
egos <- igraph::ego(g, mode = "out")

# Finding the largest ego(s)' length
max_len <- max(sapply(egos, length))

# Identifying "root" nodes of the largest ego(s)
ego_nodes <- which(sapply(egos, length) == max_len)

# Getting their name(s) in the graph
ego_nodes_names <- V(g)$name[ego_nodes]
ego_nodes_names # In the given graph it is only the node "14648"

# Extracting the first (and here unique) largest ego
largest_ego_subgraph <- induced_subgraph(g, egos[[ego_nodes[1]]])

V(largest_ego_subgraph)$color <- "lightgreen"
E(largest_ego_subgraph)$color <- "gray"

# Emphasizing the egocentric node (the "root" of the subgraph) and its ongoing edges
V(largest_ego_subgraph)[name == ego_nodes_names[1]]$color <- "magenta"
E(largest_ego_subgraph)[.from(ego_nodes_names[1])]$color <- "blue"

plot(largest_ego_subgraph, vertex.size = 6, vertex.label = NA, edge.arrow.size = 0.1)

# How many nodes and edges are in this subgraph
ecount(largest_ego_subgraph)
vcount(largest_ego_subgraph)
```

Figure 4.8

Here is the code's result (figure 4.9):

```
> # ------------
> # d - Ego(s)
> # ------------
>
> # Finding the ego(s) of the given graph
> egos <- igraph::ego(g, mode = "out")
>
> # Finding the largest ego(s)' length
> max_len <- max(sapply(egos, length))
>
> # Identifying "root" nodes of the largest ego(s)
> ego_nodes <- which(sapply(egos, length) == max_len)
>
> # Getting their name(s) in the graph
> ego_nodes_names <- V(g)$name[ego_nodes]
> ego_nodes_names # In the given graph it is only the node "14648"
[1] "14648"
>
> # Extracting the first (and here unique) largest ego
> largest_ego_subgraph <- induced_subgraph(g, egos[[ego_nodes[1]]])
>
> V(largest_ego_subgraph)$color <- "lightgreen"
> E(largest_ego_subgraph)$color <- "gray"
>
> # Emphasizing the egocentric node (the "root" of the subgraph) and its ongoing edges
> V(largest_ego_subgraph)[name == ego_nodes_names[1]]$color <- "magenta"
> E(largest_ego_subgraph)[.from(ego_nodes_names[1])]$color <- "blue"
>
> plot(largest_ego_subgraph, vertex.size = 6, vertex.label = NA, edge.arrow.size = 0.1)
>
> # How many nodes and edges are in this subgraph
> ecount(largest_ego_subgraph)
[1] 1092
> vcount(largest_ego_subgraph)
[1] 146
```

Figure 4.9

Here, we can observe that the largest and unique egocentric node is 14648. This id we also found in question a. for the most central node according to the degree centrality. This confirms our found result.

The following screenshot illustrates the the largest egocentric network of the graph (figure 4.10):

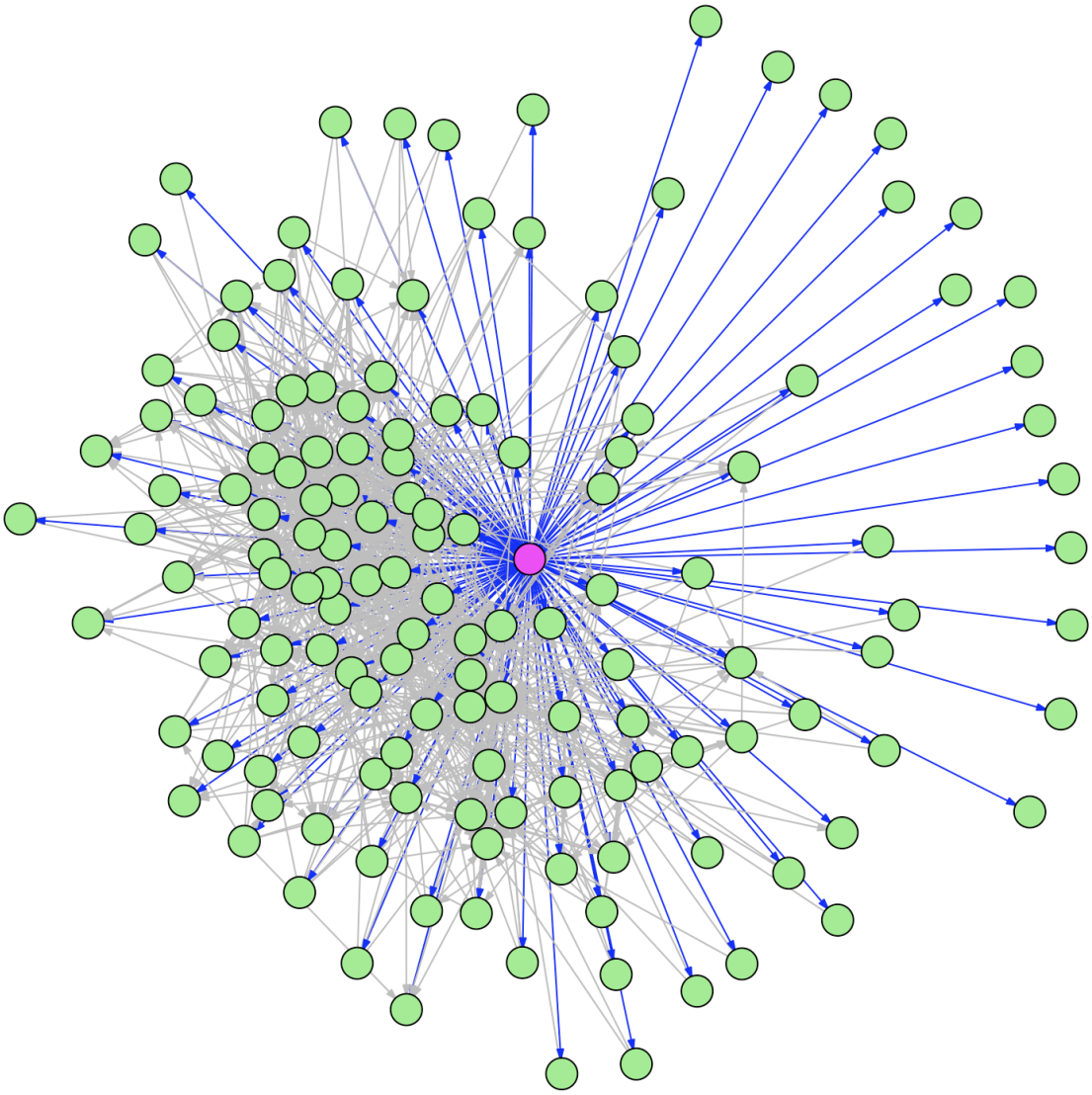


Figure 4.10

In our context of emails, as we have already said in the question a., reveals that the individual with id 14648 has sent the most emails and here we can perceive the list of all entities he sent letter(s) to.

e) To be able to calculate the power centrality of the graph we have used a pre-built function of an igraph called power_centrality().

This function is used to calculate the Bonacich power centrality scores for each node in a

graph given in the parameters. This measures the influence of each node based on its connections to other influential nodes within the network. Higher power centrality scores indicate that the node is more influential within the network due to its connections to other influential nodes.

However, this time we were not able to execute the function on the initial graph due to its complexity, therefore we used one of our simplified graphs to have a better image. Nonetheless, we used the simplified graph number 2, which is the subgraph of the initial graph where only the most connected nodes are included (each node of the graph has at least 250 neighbors in the initial graph). Thus, the found result should be valid in the initial graph, according to the definition of the Bonacich power centrality.

Here is the result (figure 4.11) containing the name of the node with highest power centrality and the source code for this part.

```
> # ----------------------
> # e - Power centrality
> # ----------------------
>
> power_centrality_scores <- power_centrality(g_simplified)
> power_centrality_scores <- sort(power_centrality_scores)
> power_centrality_scores[length(power_centrality_scores)]
     962
4.016421
```

Figure 4.11

This reveals that node number 962 is the most influential based on the connections of its connections, with the total approximative score of 4.016.

ENCOUNTERED PROBLEMS

- First we were trying to test function on a real dataset, but of course it didn't worked because we shortly were run out of memory, but then by simplifying the graph everything worked
- There were also some warnings because some functions were deprecated in our version of R studio, but each time the console's messages helped us to correct the function's name

CONCLUSION - DISCUSSION

By doing this project, we have learned more about how to manipulate large datasets. We have also discovered different ways to simplify graphs, as well as the advantages and disadvantages of each simplification method. We also became acquainted with the R environment and different functions pre-implemented for easier data manipulation and analysis.