COMPUTER SCIENCE AND DATA ANALYTICS
Course: CSCI 6444 Intro to Big Data Analytics

Class project #1

R and Graph Analytics

Student(s): Group 6
Anar Shikhaliyev
Eljan Mammadli

Instructor: Dr. Abzatdin Adamov

Baku 2023

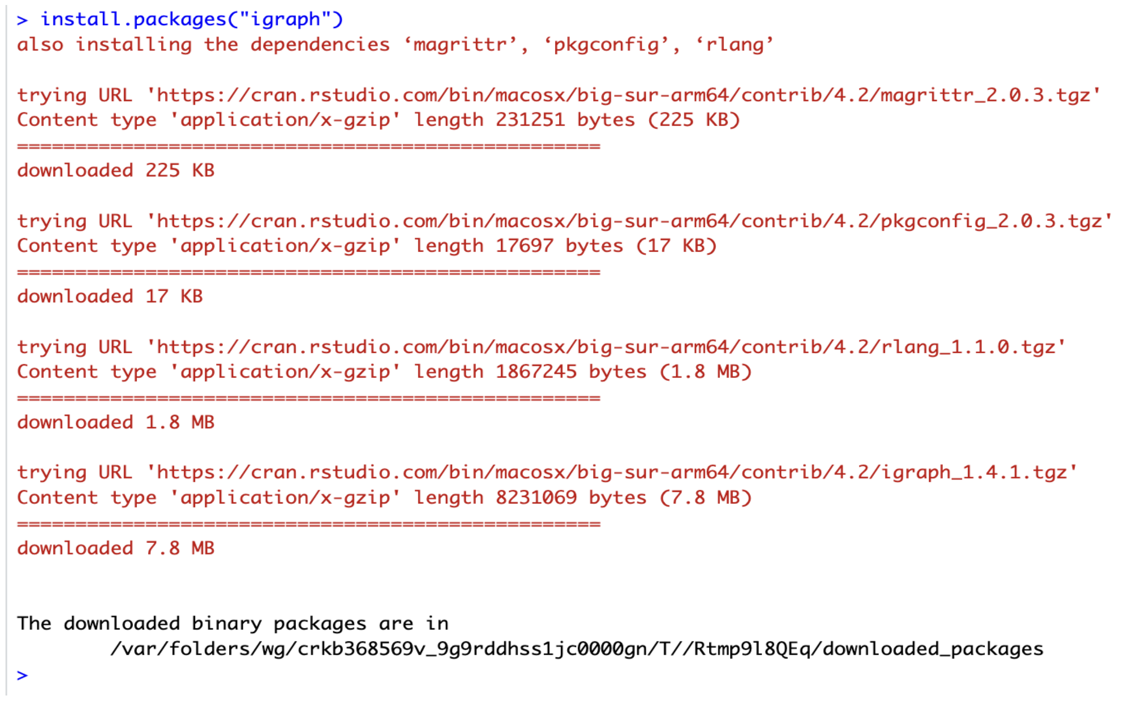
1. Data Set

In this assignment, we have a network of all the incoming and outgoing email of the research institution communication between Europian countries. We have 3,038,531 emails in total, distributed among 287,755 different email accounts. We only have a complete email graph for 1,258 of the research institution's email addresses. Also, within the dataset's time frame, 34,203 email addresses sent and received email. All other email addresses are either invalid, typographically incorrect, or spam.

2. Install the igraph package from one of the CRAN mirrors

In order to install the igraph, we need to follow these steps:

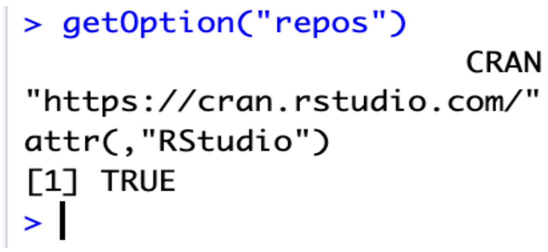
1. Open RStudio
2. Type following command in the R console: install.packages("igraph")



```
> install.packages("igraph")
also installing the dependencies ‘magrittr’, ‘pkgconfig’, ‘rlang’

trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-arm64/contrib/4.2/magrittr_2.0.3.tgz'
Content type 'application/x-gzip' length 231251 bytes (225 KB)
==================================================
downloaded 225 KB

trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-arm64/contrib/4.2/pkgconfig_2.0.3.tgz'
Content type 'application/x-gzip' length 17697 bytes (17 KB)
==================================================
downloaded 17 KB

trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-arm64/contrib/4.2/rlang_1.1.0.tgz'
Content type 'application/x-gzip' length 1867245 bytes (1.8 MB)
==================================================
downloaded 1.8 MB

trying URL 'https://cran.rstudio.com/bin/macosx/big-sur-arm64/contrib/4.2/igraph_1.4.1.tgz'
Content type 'application/x-gzip' length 8231069 bytes (7.8 MB)
==================================================
downloaded 7.8 MB


The downloaded binary packages are in
	/var/folders/wg/crkb368569v_9g9rddhss1jc0000gn/T//Rtmp9l8QEq/downloaded_packages
>
```

Picture 1. Installing igraph

3. R did not prompt any CRAN mirror selection because, it installed packages from default CRAN mirror in R configuration which we can access it by:

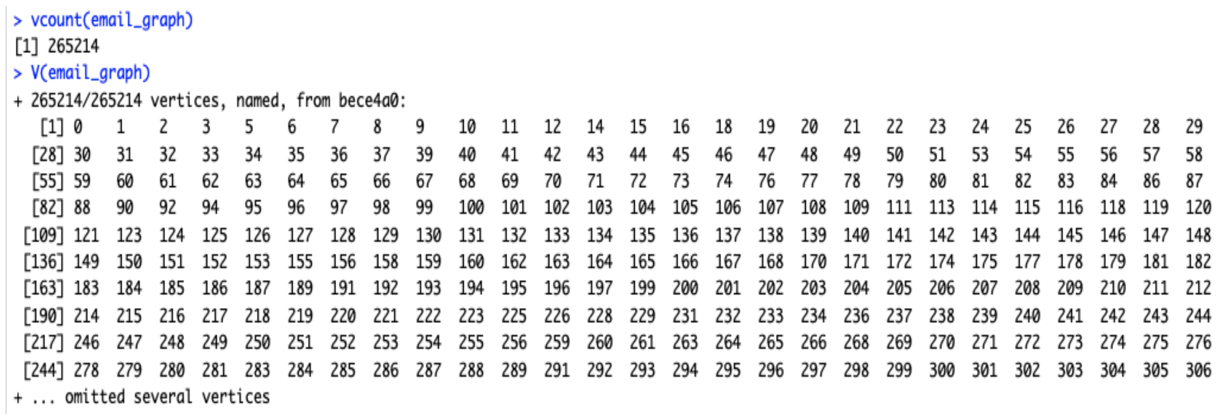


```
> getOption("repos")
                           CRAN
"https://cran.rstudio.com/"
attr(,"RStudio")
[1] TRUE
> 
```

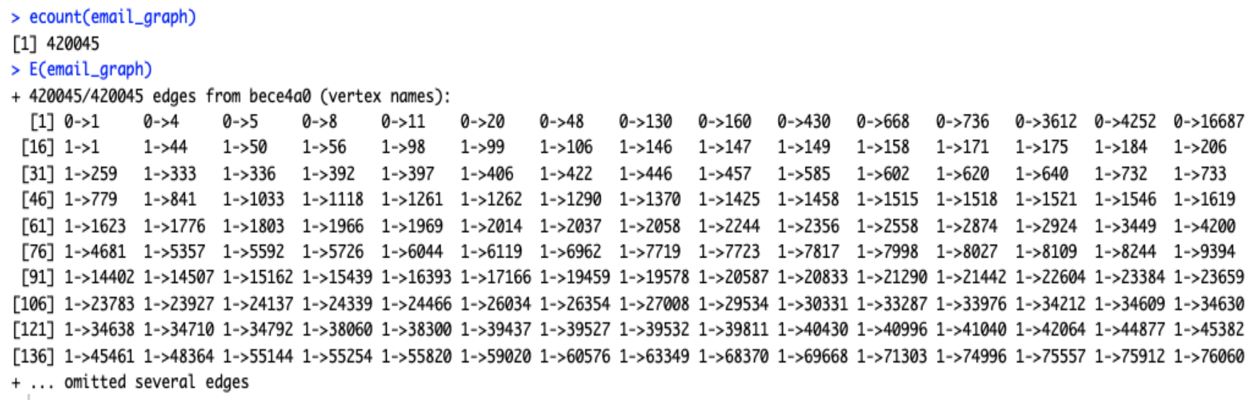
3. Experiment with some of the functions that shown in the Introduction to Graph Analytics document on Blackboard on the graph generated from the data set. Present the results in your write-up.

We have experimented this R functions on the given dataset:

- Vcount(), V() and ecount(), E() - These functions help to gain information about the vectors and edges of the graph respectively.

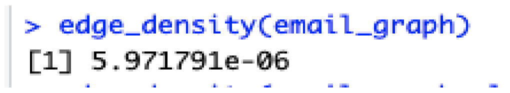


```
> vcount(email_graph)
[1] 265214
> V(email_graph)
+ 265214/265214 vertices, named, from bece4a0:
   [1] 0   1   2   3   5   6   7   8   9   10  11  12  14  15  16  18  19  20  21  22  23  24  25  26  27  28  29 
  [28] 30  31  32  33  34  35  36  37  39  40  41  42  43  44  45  46  47  48  49  50  51  53  54  55  56  57  58 
  [55] 59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  76  77  78  79  80  81  82  83  84  86  87 
  [82] 88  90  92  94  95  96  97  98  99  100 101 102 103 104 105 106 107 108 109 111 113 114 115 116 118 119 120
 [109] 121 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
 [136] 149 150 151 152 153 155 156 158 159 160 162 163 164 165 166 167 168 170 171 172 174 175 177 178 179 181 182
 [163] 183 184 185 186 187 189 191 192 193 194 195 196 197 199 200 201 202 203 204 205 206 207 208 209 210 211 212
 [190] 214 215 216 217 218 219 220 221 222 223 225 226 228 229 231 232 233 234 236 237 238 239 240 241 242 243 244
 [217] 246 247 248 249 250 251 252 253 254 255 256 259 260 261 263 264 265 266 268 269 270 271 272 273 274 275 276
 [244] 278 279 280 281 283 284 285 286 287 288 289 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306
+ ... omitted several vertices
```



```
> ecount(email_graph)
[1] 420045
> E(email_graph)
+ 420045/420045 edges from bece4a0 (vertex names):
  [1] 0->1     0->4     0->5     0->8     0->11    0->20    0->48    0->130   0->160   0->430   0->668   0->736   0->3612  0->4252  0->16687
 [16] 1->1     1->44    1->50    1->56    1->98    1->99    1->106   1->146   1->147   1->149   1->158   1->171   1->175   1->184   1->206
 [31] 1->259   1->333   1->336   1->392   1->397   1->406   1->422   1->446   1->457   1->585   1->602   1->620   1->640   1->732   1->733
 [46] 1->779   1->841   1->1033  1->1118  1->1261  1->1262  1->1290  1->1370  1->1425  1->1458  1->1515  1->1518  1->1521  1->1546  1->1619
 [61] 1->1623  1->1776  1->1803  1->1966  1->1969  1->2014  1->2037  1->2058  1->2244  1->2356  1->2558  1->2874  1->2924  1->3449  1->4200
 [76] 1->4681  1->5357  1->5592  1->5726  1->6044  1->6119  1->6962  1->7719  1->7723  1->7817  1->7998  1->8027  1->8109  1->8244  1->9394
 [91] 1->14402 1->14507 1->15162 1->15439 1->16393 1->17166 1->19459 1->19578 1->20587 1->20833 1->21290 1->21442 1->22604 1->23384 1->23659
[106] 1->23783 1->23927 1->24137 1->24339 1->24466 1->26034 1->26354 1->27008 1->29534 1->30331 1->33287 1->33976 1->34212 1->34609 1->34630
[121] 1->34638 1->34710 1->34792 1->38060 1->38300 1->39437 1->39527 1->39532 1->39811 1->40430 1->40996 1->41040 1->42064 1->44877 1->45382
[136] 1->45461 1->48364 1->55144 1->55254 1->55820 1->59020 1->60576 1->63349 1->68370 1->69668 1->71303 1->74996 1->75557 1->75912 1->76060
+ ... omitted several edges
```

- Density - proportion between the number of edges and the number of potential edges is known as a graph's density.

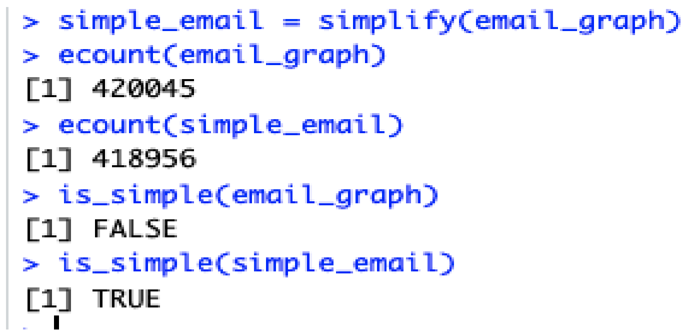


```
> edge_density(email_graph)
[1] 5.971791e-06
```

- Degree - shows the degree of every node inside the graph. Since our data is too big we will not show the actual output.

Since our data is too large some of the commands were taking too much time and resource to execute. For that reason, it might be helpful to simplify the graph when working with big networks like ours to minimize the number of nodes and edges, making it easier to manipulate. So, we need to simplify the graph in order to execute those commands.

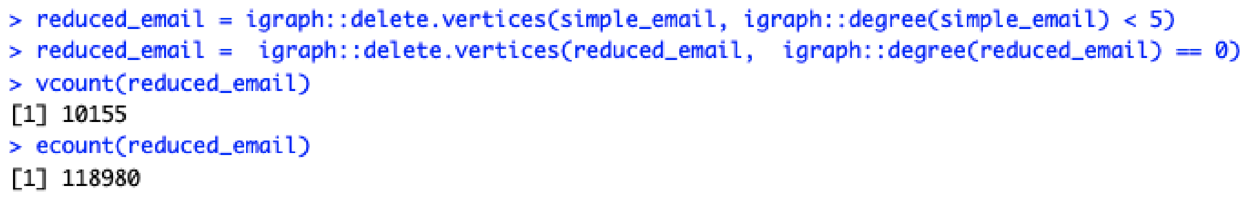
- Simplify and is_simple - eliminates the loop as well as several edges from a graph. As we can see from the output below, there is a slight decrease in number of edges after removing loops and it is verified by the is_simple function.



```
> simple_email = simplify(email_graph)
> ecount(email_graph)
[1] 420045
> ecount(simple_email)
[1] 418956
> is_simple(email_graph)
[1] FALSE
> is_simple(simple_email)
[1] TRUE
```

But doing only this was not enough to simplify the graph, we needed to implement other simplification tactics.

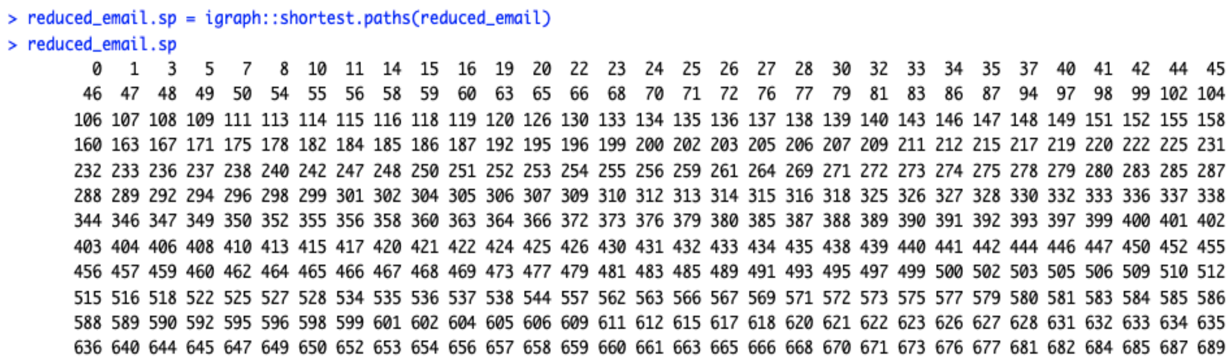
First technic we used is *Node degree thresholding*. With this method we eliminated nodes with a low degree, considering they have less connections and therefore not crucial to the network's overall structure. The minimal number of edges a node must possess in order to be included in the condensed graph can be specified as a threshold (which is **5** in our case). After deleting nodes with less than five it is likely to have some nodes with zero degree, thus we eliminated them as well.



```
> reduced_email = igraph::delete.vertices(simple_email, igraph::degree(simple_email) < 5)
> reduced_email =  igraph::delete.vertices(reduced_email,  igraph::degree(reduced_email) == 0)
> vcount(reduced_email)
[1] 10155
> ecount(reduced_email)
[1] 118980
```

- shortest.paths()

  The shortest pathways between each pair of vertices in a graph are determined using the igraph::shortest.paths() method. The shortest path between a source vertex and every other vertex in the graph is determined using Dijkstra's method.

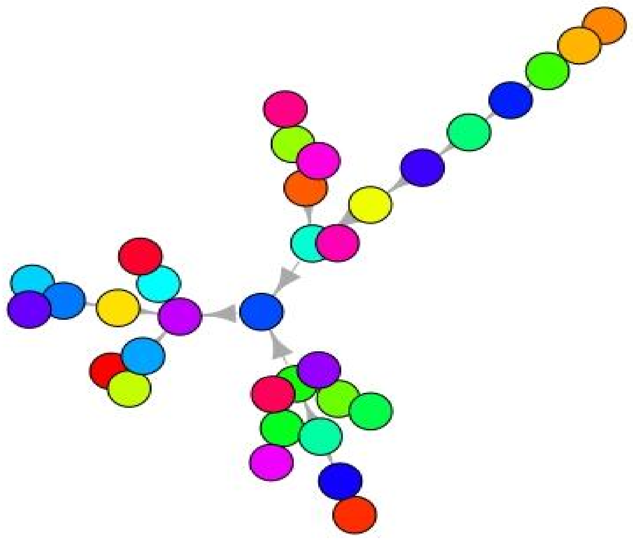


```
> reduced_email.sp = igraph::shortest.paths(reduced_email)
> reduced_email.sp
          0   1   3   5   7   8  10  11  14  15  16  19  20  22  23  24  25  26  27  28  30  32  33  34  35  37  40  41  42  44  45
         46  47  48  49  50  54  55  56  58  59  60  63  65  66  68  70  71  72  76  77  79  81  83  86  87  94  97  98  99 102 104
        106 107 108 109 111 113 114 115 116 118 119 120 126 130 133 134 135 136 137 138 139 140 143 146 147 148 149 151 152 155 158
        160 163 167 171 175 178 182 184 185 186 187 192 195 196 199 200 202 203 205 206 207 209 211 212 215 217 219 220 222 225 231
        232 233 236 237 238 240 242 247 248 250 251 252 253 254 255 256 259 261 264 269 271 272 273 274 275 278 279 280 283 285 287
        288 289 292 294 296 298 299 301 302 304 305 306 307 309 310 312 313 314 315 316 318 325 326 327 328 330 332 333 336 337 338
        344 346 347 349 350 352 355 356 358 360 363 364 366 372 373 376 379 380 385 387 388 389 390 391 392 393 397 399 400 401 402
        403 404 406 408 410 413 415 417 420 421 422 424 425 426 430 431 432 433 434 435 438 439 440 441 442 444 446 447 450 452 455
        456 457 459 460 462 464 465 466 467 468 469 473 477 479 481 483 485 489 491 493 495 497 499 500 502 503 505 506 509 510 512
        515 516 518 522 525 527 528 534 535 536 537 538 544 557 562 563 566 567 569 571 572 573 575 577 579 580 581 583 584 585 586
        588 589 590 592 595 596 598 599 601 602 604 605 606 609 611 612 615 617 618 620 621 622 623 626 627 628 631 632 633 634 635
        636 640 644 645 647 649 650 652 653 654 656 657 658 659 660 661 663 665 666 668 670 671 673 676 677 681 682 684 685 687 689
```

4. Explore other functions in the igraph package

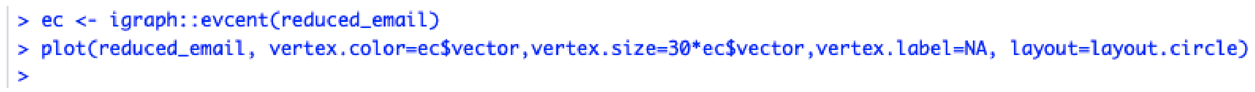
- mst()

  Minimal Spanning Tree connects all the vertices of the graph with the minimum possible total edge weight. Interpretation can vary depending on the context such as identifying most important (central nodes), efficient paths between pairs of vertices, and even identfying communities.

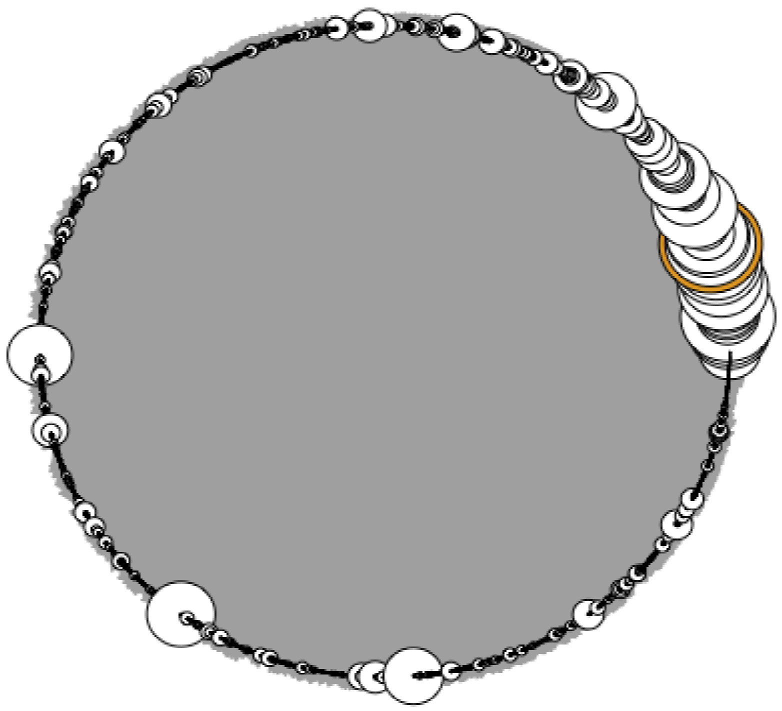


- eigenvector_centrality()

  A measure of a vertex's relevance in a network called eigenvector centrality and considers both the vertex's connectivity to other nodes and the significance of the nodes to which those connections are made.

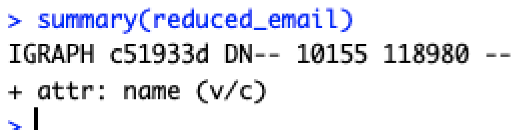


```
> ec <- igraph::evcent(reduced_email)
> plot(reduced_email, vertex.color=ec$vector,vertex.size=30*ec$vector,vertex.label=NA, layout=layout.circle)
>
```



- summary()

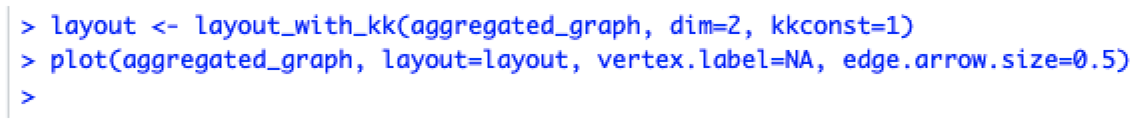
  This function offers a summary of the graph, including its type, size, number of vertices and edges, and any properties.



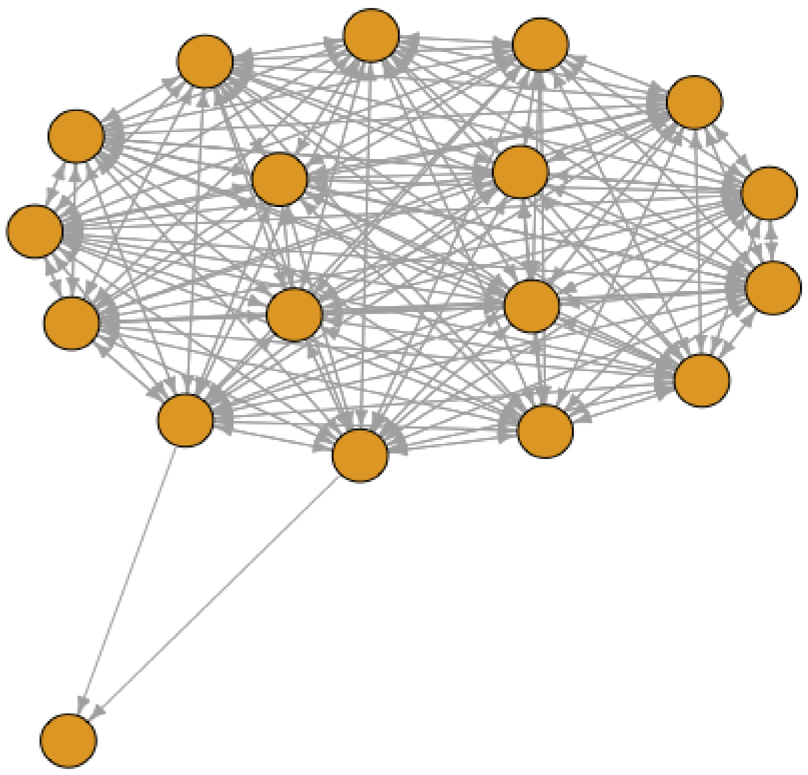
```
> summary(reduced_email)
IGRAPH c51933d DN-- 10155 118980 --
+ attr: name (v/c)
>
```

- Cluster_louvain (explained in more detail at 5.f)
- Layout_with_kk

  This function from the igraph package uses the Kamada-Kawai technique to determine a graph's layout. By placing vertices in places that minimize the sum of the spring and electrical forces connecting them, the Kamada-Kawai algorithm, a force-directed graph layout technique, seeks to reduce the overall energy of the network.

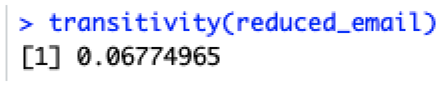


```
> layout <- layout_with_kk(aggregated_graph, dim=2, kkconst=1)
> plot(aggregated_graph, layout=layout, vertex.label=NA, edge.arrow.size=0.5)
>
```



- Transitivity

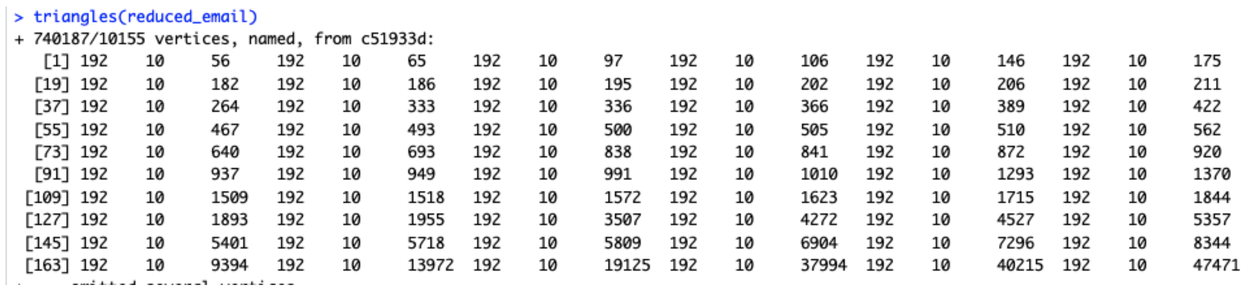
  This function determines a graph's transitivity, which is the proportion of triangles to linked triples of vertices in the graph.



```
> transitivity(reduced_email)
[1] 0.06774965
```

- Triangles

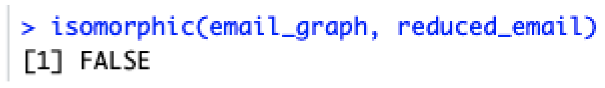
This function determines a graph's triangle count, which is a gauge of the graph's clustering coefficient.



```
> triangles(reduced_email)
+ 740187/10155 vertices, named, from c51933d:
   [1] 192   10    56    192   10    65    192   10    97    192   10    106   192   10    146   192   10    175
  [19] 192   10    182   192   10    186   192   10    195   192   10    202   192   10    206   192   10    211
  [37] 192   10    264   192   10    333   192   10    336   192   10    366   192   10    389   192   10    422
  [55] 192   10    467   192   10    493   192   10    500   192   10    505   192   10    510   192   10    562
  [73] 192   10    640   192   10    693   192   10    838   192   10    841   192   10    872   192   10    920
  [91] 192   10    937   192   10    949   192   10    991   192   10    1010  192   10    1293  192   10    1370
 [109] 192   10    1509  192   10    1518  192   10    1572  192   10    1623  192   10    1715  192   10    1844
 [127] 192   10    1893  192   10    1955  192   10    3507  192   10    4272  192   10    4527  192   10    5357
 [145] 192   10    5401  192   10    5718  192   10    5809  192   10    6904  192   10    7296  192   10    8344
 [163] 192   10    9394  192   10    13972 192   10    19125 192   10    37994 192   10    40215 192   10    47471
```

- Isomorphic

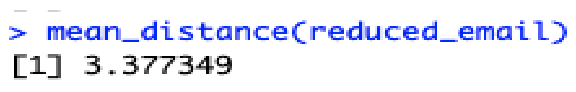
This function determines if two graphs are isomorphic, which implies they share the same structure but may have distinct vertex and edge names. Output below is clearly false, because we eliminated some connections, resulting changing the structure.



```
> isomorphic(email_graph, reduced_email)
[1] FALSE
```

- Mean_distance

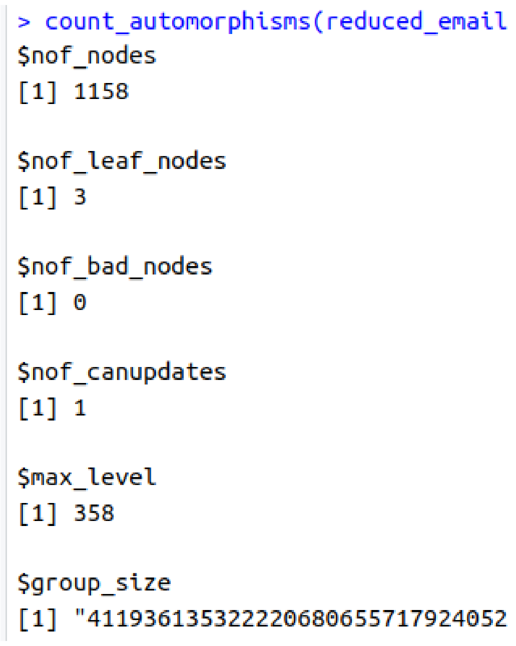
The term "mean distance" describes the shortest path's average length between every pair of vertices in a network. The smallest number of edges that must be crossed to get from one vertex to another is known as the shortest path length.



```
> mean_distance(reduced_email)
[1] 3.377349
```

- count_automorphisms()

This is a method to find the number of automorphisms in a graph, which are isomorphisms from the graph to itself. An automorphism of a graph is a permutation of its vertices that preserves its edges. Isomorphisms means, two graphs being structurally equivalent. It seems that there is a high degree of symmetry in the network.



```
> count_automorphisms(reduced_email
$nof_nodes
[1] 1158

$nof_leaf_nodes
[1] 3

$nof_bad_nodes
[1] 0

$nof_canupdates
[1] 1

$max_level
[1] 358

$group_size
[1] "41193613532220680655717924052
```

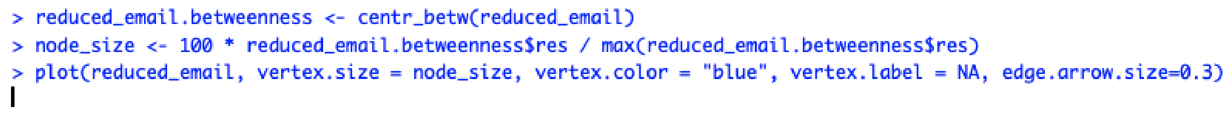
5. Determine the (a) central nodes(s) in the graph, (b) longest path(s), (c) largest clique(s), (d) ego(s), (e) power centrality, (f) find communities.

- central nodes(s) in the graph

  There are several methods for calculating centrality in a network, and various centrality metrics can draw attention to various kinds of significant nodes.

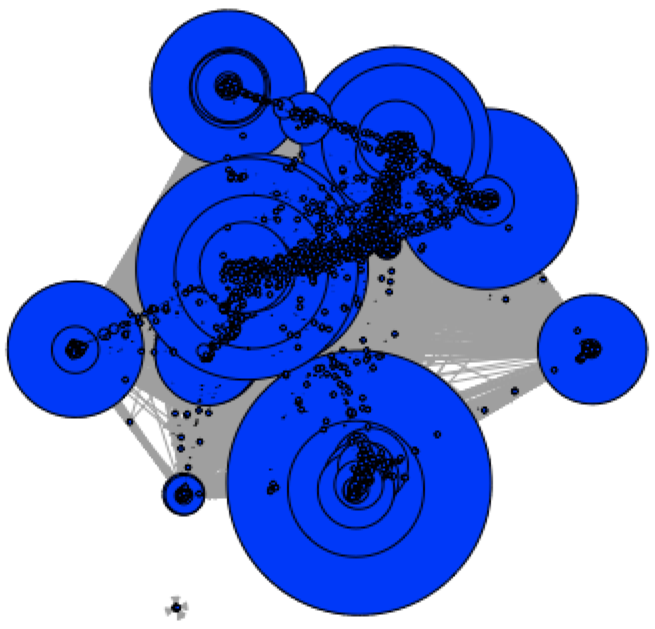
  a. Betweenness centrality

     This calculates how far a node is located along the network's shortest pathways to other nodes. Bridges connecting various regions of the network are built by nodes with high betweenness centrality.

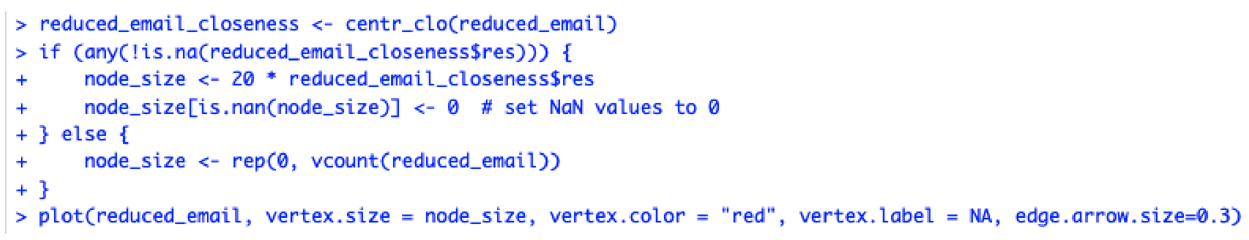


```
> reduced_email.betweenness <- centr_betw(reduced_email)
> node_size <- 100 * reduced_email.betweenness$res / max(reduced_email.betweenness$res)
> plot(reduced_email, vertex.size = node_size, vertex.color = "blue", vertex.label = NA, edge.arrow.size=0.3)
```

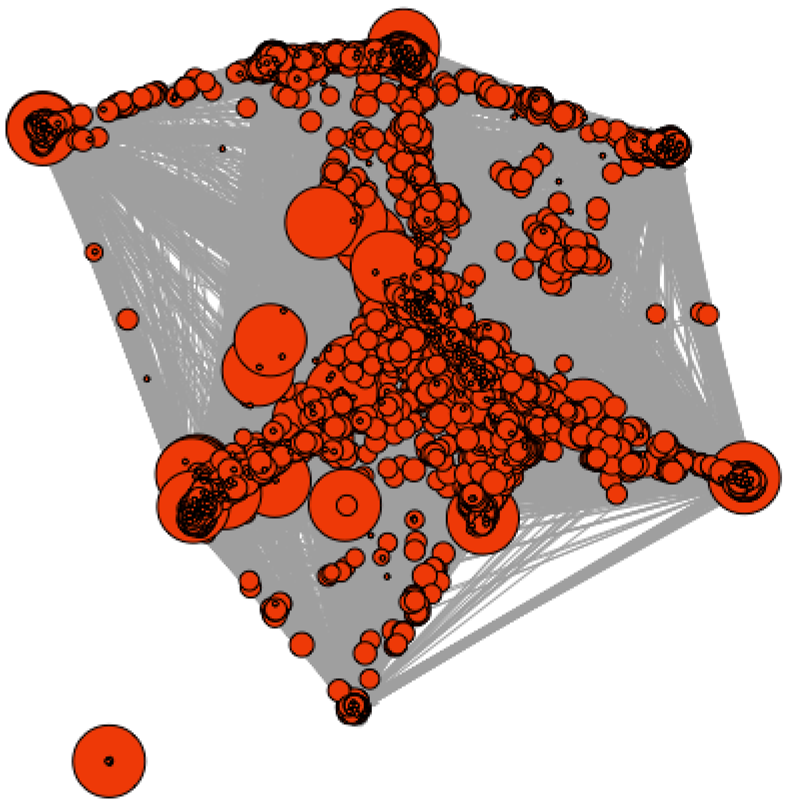
In order to visualize betweenness, we can filter nodes where size is proportional to the corresponding centrality score:



b. Closeness centrality determined by taking the reciprocal of the lengths of the shortest routes that connect the node to every other node in the graph. As in betweenness centrality, we filter nodes by size as a measure of the importance of each node based on its closeness centrality, where larger nodes indicate greater importance.

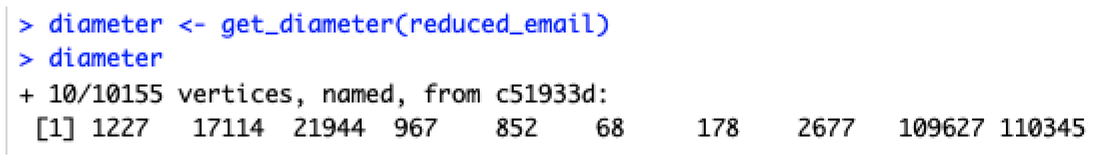


```
> reduced_email_closeness <- centr_clo(reduced_email)
> if (any(!is.na(reduced_email_closeness$res))) {
+     node_size <- 20 * reduced_email_closeness$res
+     node_size[is.nan(node_size)] <- 0  # set NaN values to 0
+ } else {
+     node_size <- rep(0, vcount(reduced_email))
+ }
> plot(reduced_email, vertex.size = node_size, vertex.color = "red", vertex.label = NA, edge.arrow.size=0.3)
```

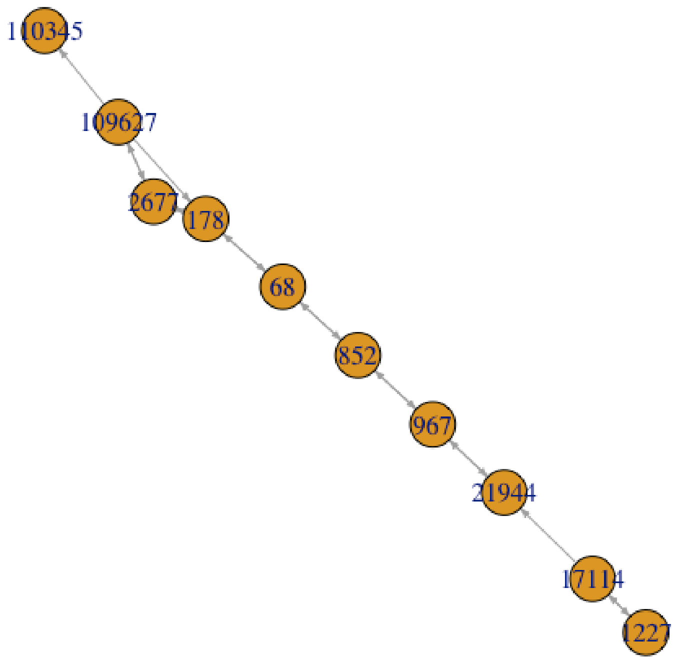


- longest path

  In order to find the longest path of the graph, we can calculates the diameter of the graph, which is the length of the longest shortest path in the graph.

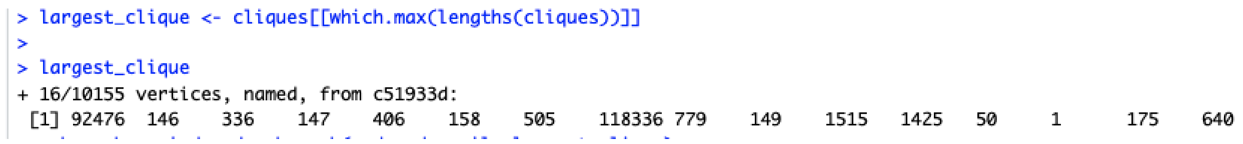


```
> diameter <- get_diameter(reduced_email)
> diameter
+ 10/10155 vertices, named, from c51933d:
 [1] 1227   17114  21944  967    852    68     178    2677   109627 110345
```

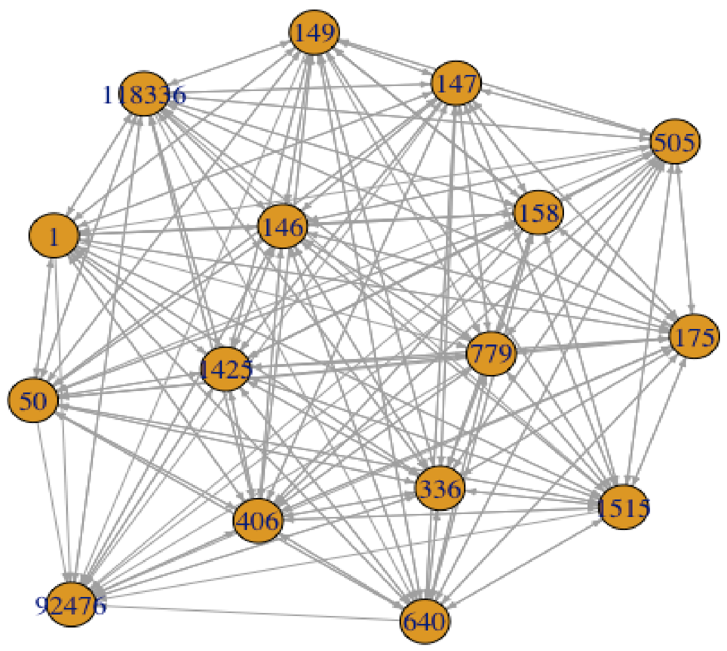


- largest clique

This function returns all the maximal cliques in the graph, and we can select the largest one using the which.max function. After finding the largest clique, we can visualize it as subgraph.

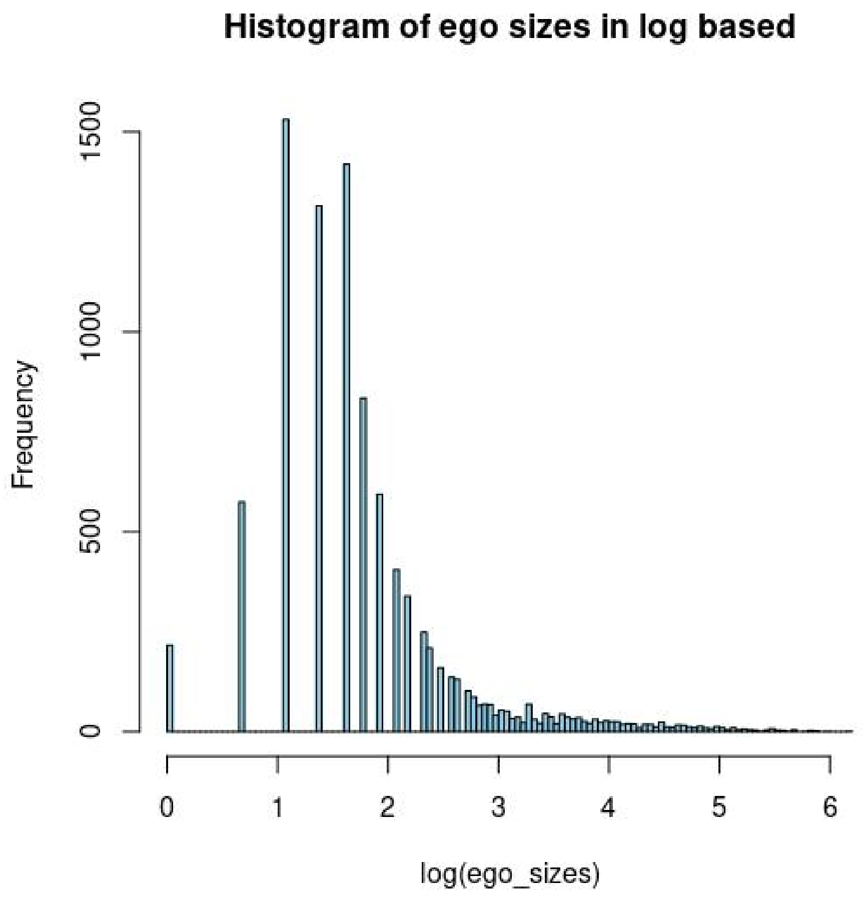


```
> largest_clique <- cliques[[which.max(lengths(cliques))]]
> 
> largest_clique
+ 16/10155 vertices, named, from c51933d:
 [1] 92476  146    336    147    406    158    505    118336 779    149    1515   1425   50     1      175    640
```



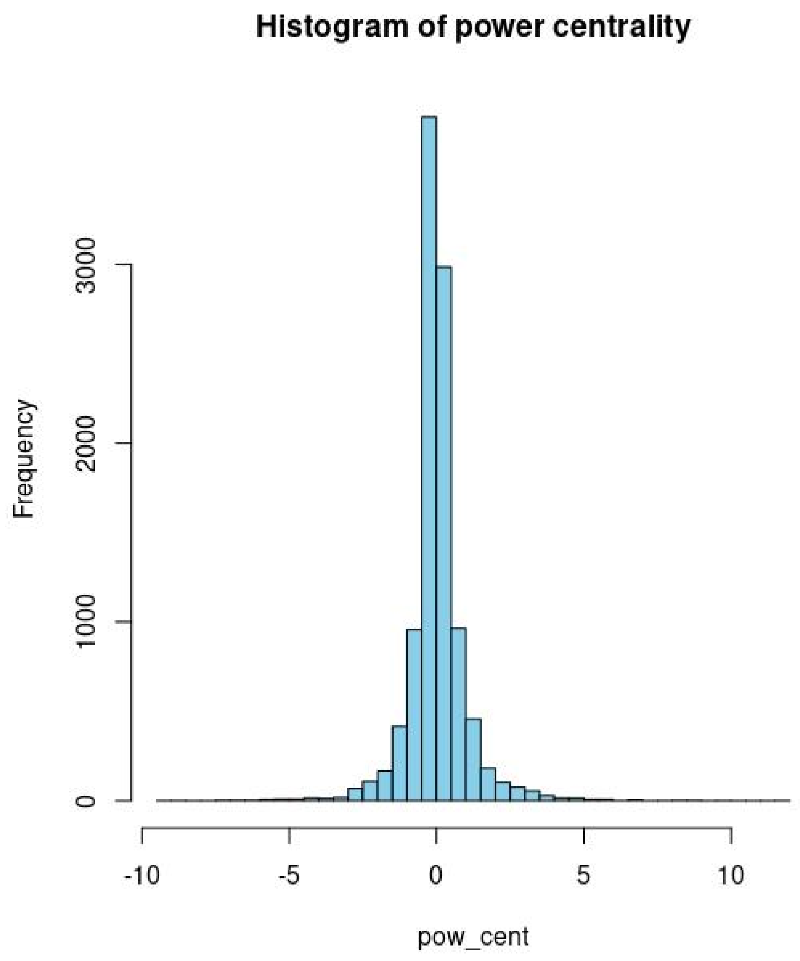
- (d) ego(s),

  From the histogram below it is visible that it is right skewed which means generality of the nodes in the network have ego sizes and only among 10536 nodes 150 of them has ego sizes more than 100. The mean, median, maximum values are 11.2, 5, 479 respectively.



- (e) power centrality,

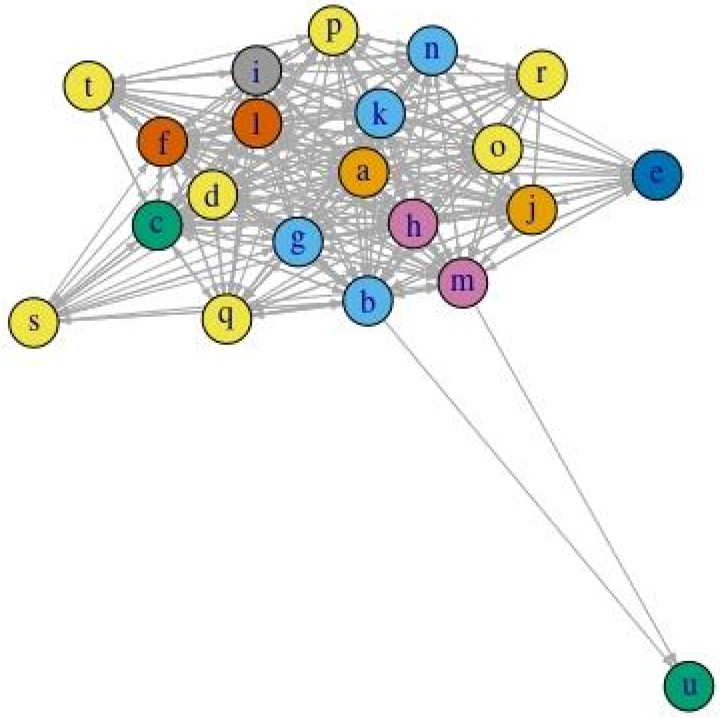
  It is measure of the influence of nodes in the network using the connections of them. In this network we can clearly see from distribution that majority of the the values in the power centrality lies around zero which are very small number. This can indicate that few of the nodes in the network play crucial influence in the organization while lots of them do not have reasonable effect. The maximum value is 11.9 while median is 0.



- (f) find communities

  We performed both clustering and community detection on the network. Community discovery is process of clustering nodes of the network which are densely connected but sparsely to the rest of the network. In order to understand the shared network characteristics among the nodes we used *Clustering and Node aggregation technic.* With this method, we find closely linked groups of nodes and consider them as a single node in the condensed graph. By doing so, it enables us to decrease the quantity of nodes and edges while maintaining the network's general structure.

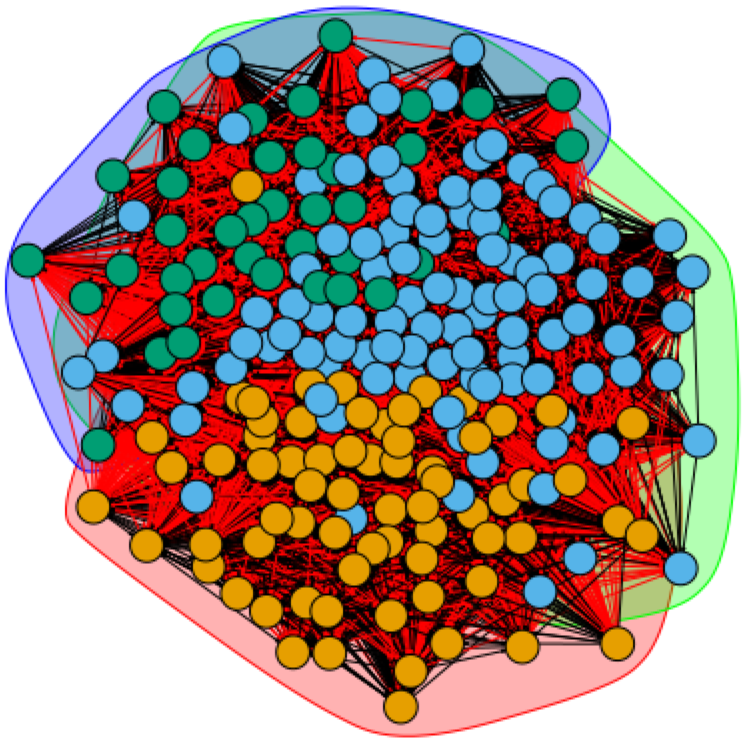
We used Louvian method which is a well-known technique for community discovery in graphs. It seeks to divide a graph's nodes into distinct communities or clusters depending on the graph's topology. It is a two-stage procedure: the first phase involves assigning each node to a distinct community, and the second step involves iteratively merging neighboring communities to maximize a quality function that gauges the modularity of the final partition. The modularity gauges how closely related nodes within a community are to those connecting nodes between communities.
After aggregating the vertices which belong to the same community into a vertex, we removed the isolated vertices from the graph.



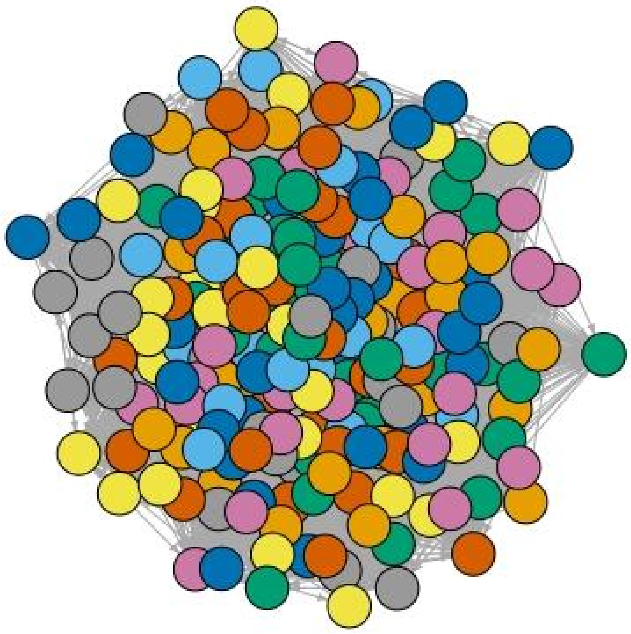
In the picture above we can clearly see that there are departments or teams in this network which have mostly close relationship with each other. One of the insight we can get from graph is that clusters “a”, “k”, “l”, “o” seems to be important having dense relationship with almost other groups. Another striking pattern is that cluster “u” never emailed to any of the groups which may indicate that this group does not actively play role in the organization but only gets some updates from “b” and “m” teams. Next outstanding point can be that there are some groups such as “t”, “s”, “e”, “q” which are located at the edges of the network, seems to have close relationship with only specific departments.

6. Resulting graph with too many vertices and edges will look very messy in the plot. Try to filter vertices and their edges in some way having in resulting plot (visualization) 30 – 100 vertexes. Differentiate vertices (by color, size, shape) and edges (color, type) of graph. Think about opportunity to assign weights to edges differentiating them accordingly.

We used walktrap community detection on the network. It outputs the plot below:



We also cluster the reduced graph using louvain algorithm using resolution parameter 10 which produces 221 clusters.



We then assign weights to the edges and nodes to prune some of them. After selecting nodes with weight more than two using induced subgraph we end up with 34 nodes. We talked about interpretation in the last lection of the previous task so, we do not go to much detail here.