project2

May 5, 2022

1 Project2: Social Network Mining

- 505851728 Yang-Shan Chen
- 005627440 Chih-En Lin
- 505297814 Rikako Hatoya

[ ]:
```
install.packages("igraph")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

[ ]:
```
library('igraph')
```

```
Attaching package: ‘igraph’


The following objects are masked from ‘package:stats’:

    decompose, spectrum


The following object is masked from ‘package:base’:

    union
```

2 Part1 - Facebook Network

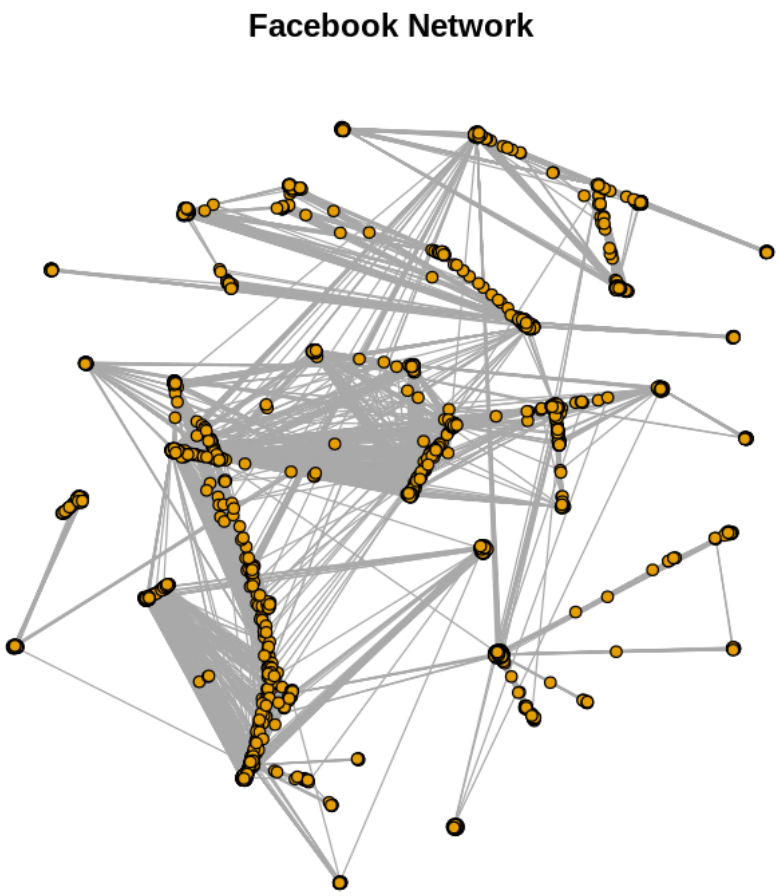
3 Q1.1

[ ]:
```
table <- drive_get("~/facebook_combined.txt")
drive_download(table)
```

```
  The input `path` resolved to exactly 1 file.

File downloaded:

• facebook_combined.txt <id:

1Q4IG7Mc5JBSqQhCZC2eO3svNJathW-3R>

Saved locally as:

• facebook_combined.txt
```

[ ]:
```r
data <- read.table("./facebook_combined.txt")
g <- graph.data.frame(data, directed=FALSE)
```

[ ]:
```r
plot(g, vertex.label="", vertex.size=3, main="Facebook Network")
```



[ ]:
```
print(sprintf("The Number of Nodes: %d", gorder(g)))
print(sprintf("The Number of Edges: %d", gsize(g)))
```

```
[1] "The Number of Nodes: 4039"
[1] "The Number of Edges: 88234"
```

4 Q1.2

[ ]:
```
is.connected(g)
```

TRUE

Yes. The Facebook Network is connected.

5 Q2

[ ]:
```
print(diameter(g))
```

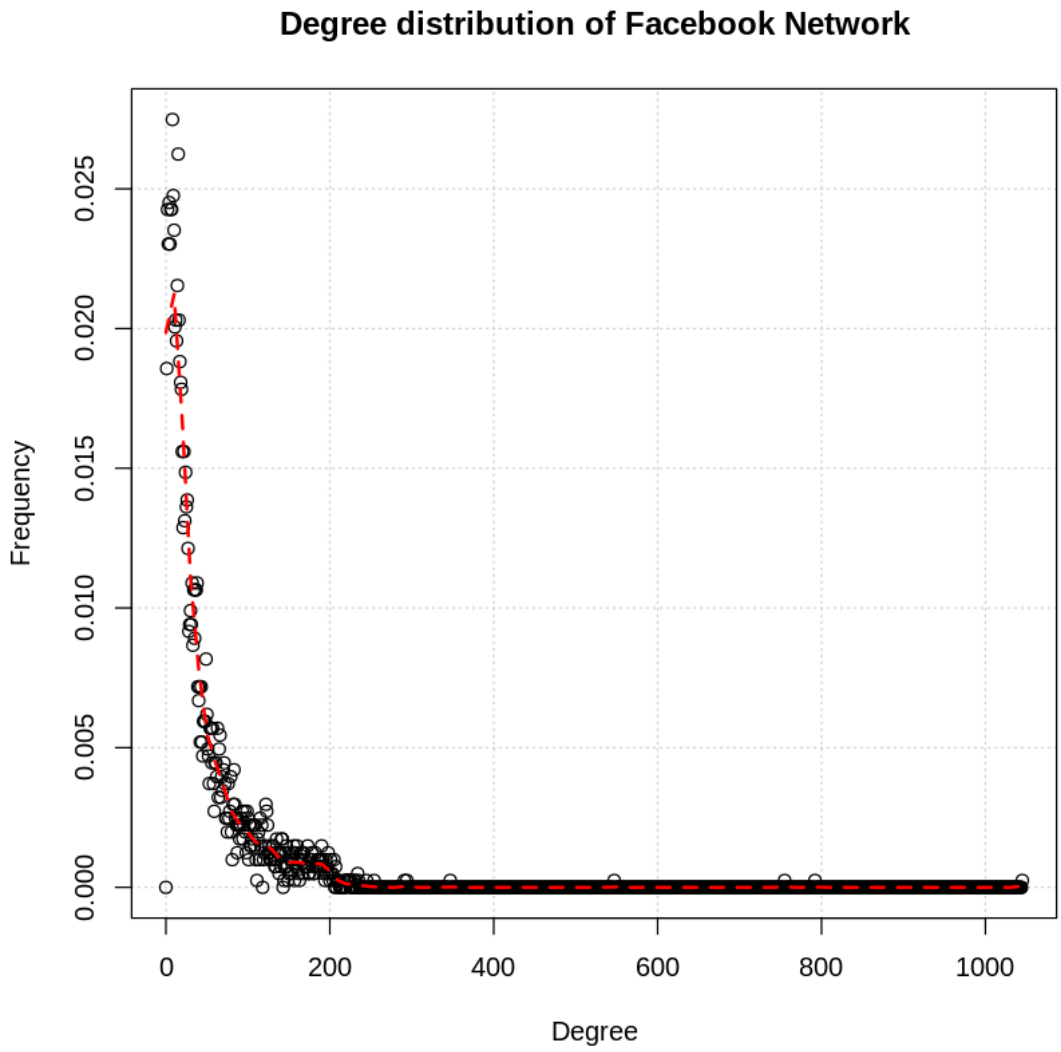
```
[1] 8
```

The diameter of the network is 8

6 Q3

[ ]:
```
deg_distri = degree.distribution(g)
```

[ ]:
```
plot(seq_along(deg_distri)-1, deg_distri, main="Degree distribution of Facebook␣
↪Network", xlab="Degree", ylab="Frequency", grid())
lines(lowess(seq_along(deg_distri) - 1, deg_distri, f = 0.027), col="red",␣
↪lwd=2, lty=2)
```



[ ]:
```
print(sprintf("Average Degree: %f", mean(degree(g))))
```

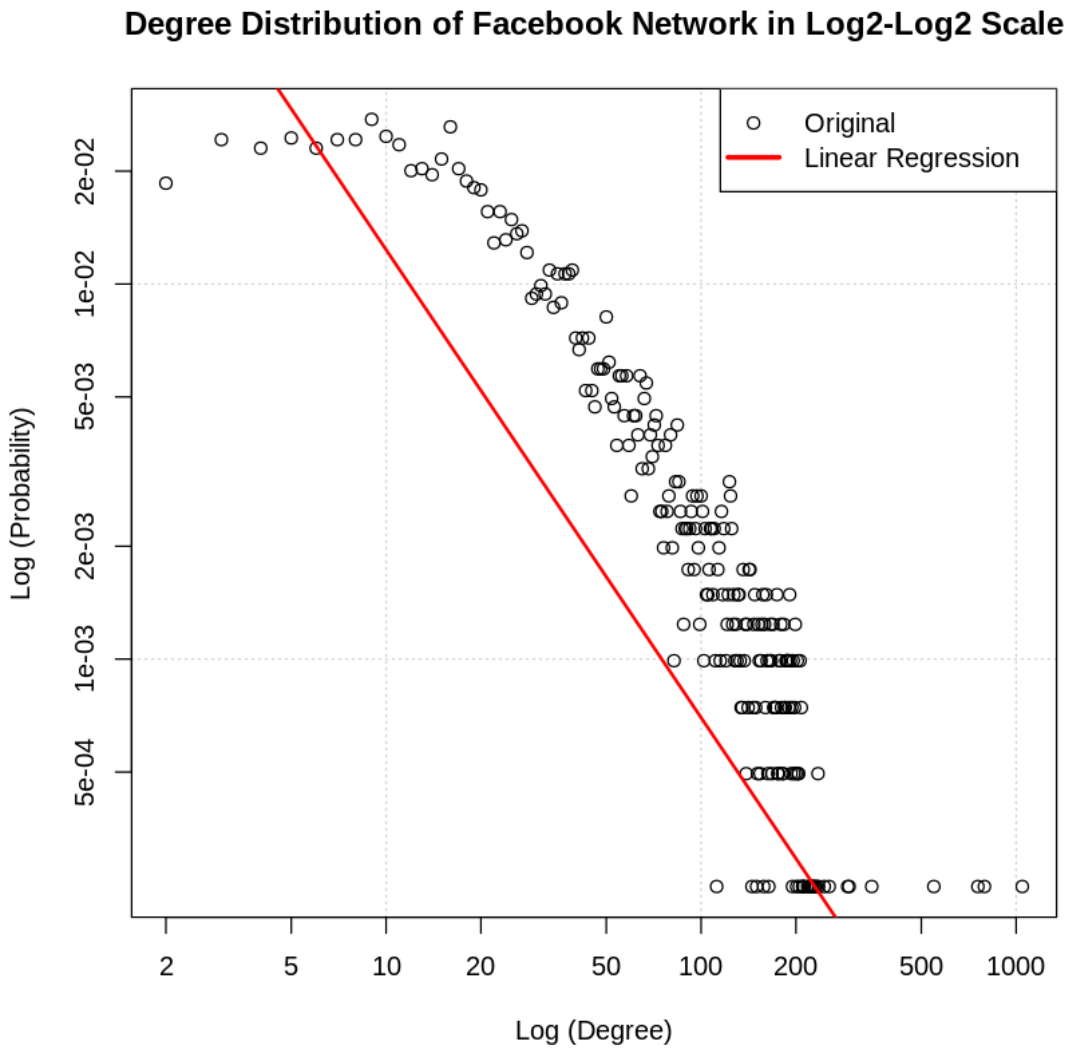
```
[1] "Average Degree: 43.691013"
```

7 Q4

[ ]:
```
deg <- c(1:length(deg_distri))[which(deg_distri !=0, arr.ind = TRUE)]
distri <- deg_distri[which(deg_distri !=0, arr.ind = TRUE)]
```

[ ]:
```
plot(deg, distri, main="Degree Distribution of Facebook Network in Log2-Log2␣
  ↪Scale", xlab="Log (Degree)", ylab="Log (Probability)", grid(), log="xy")
abline(lm(log(distri)~log(deg)), col="red", lwd=2)
```

```
legend('topright', legend=c("Original", "Linear Regression"), lty=c(0, 1),␣
↪lwd=c(1, 3), pch=c(1, NA), col=c("black", "red"))
```



[ ]:
```
print("Slope and Intercept of the Line: ")
print(lm(log(distri)~log(deg)))
```

```
[1] "Slope and Intercept of the Line: "

Call:
lm(formula = log(distri) ~ log(deg))

Coefficients:
(Intercept)     log(deg)
```

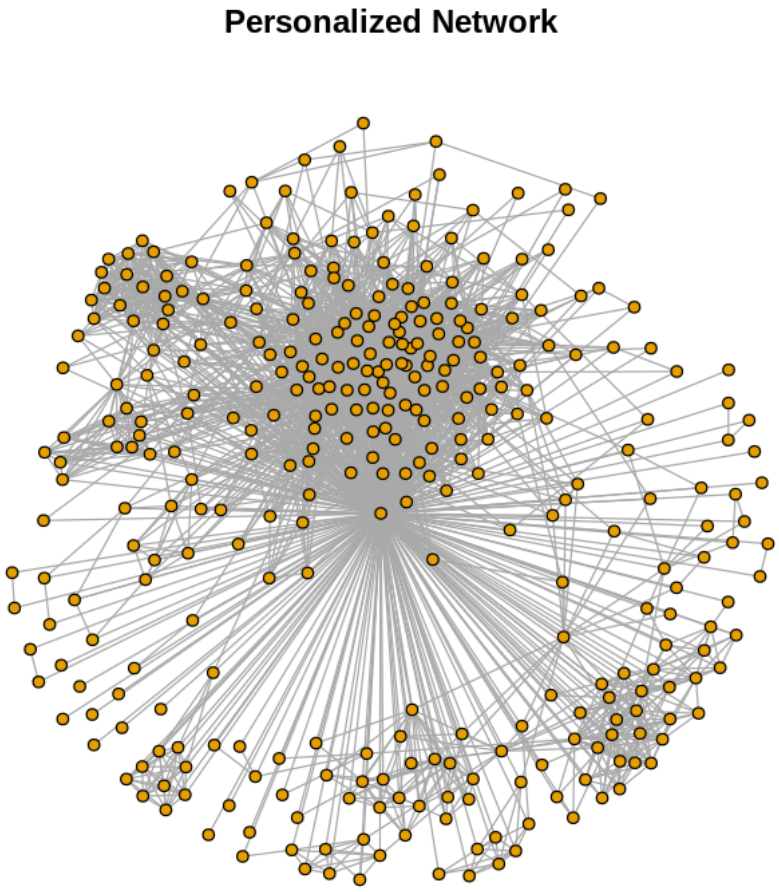
```
-0.6611       -1.2475
```

The slope of the line is -1.2475

8 Q5

[ ]:
```
pg = make_ego_graph(g, order=1, nodes=V(g), mindist=0)[[1]]
```

[ ]:
```
plot(pg, vertex.label="", vertex.size=3, main = "Personalized Network")
```



[ ]:
```
print(sprintf("The Number of Nodes: %d", gorder(pg)))
print(sprintf("The Number of Edges: %d", gsize(pg)))
```

```
[1] "The Number of Nodes: 348"
[1] "The Number of Edges: 2866"
```

9 Q6

[ ]:
```
print(diameter(pg))
```

```
[1] 2
```

- The diameter of the personalized network of the user whose ID is 1 is 2
- The trivial lower bound for the diameter of this network is 1, and the trivial upper bound is 2

10 Q7

The diameter of a graph means the largest distance between any pairs of nodes (greatest shortest path between any two nodes). - If the diameter of the personalized network is 1, then it means that all users in the network know each other directly. Thus, there are direct edges between all vertices in the network, which is fully-connected. - If the diameter of the personalized network is 2, then it means that some users are not connected directly, but they are connected through a mutual user. Thus, the network is not fully-connected. Not all of the users in the network know each other directly.

11 Q8

[ ]:
```
data <- read.table("facebook_combined.txt")
g <- graph.data.frame(data, directed=FALSE)
```

[ ]:
```
core <- which(neighborhood.size(g,1,nodes=V(g))>201)
print(length(core))
degCore <- mean(degree(g, v=V(g)[core]))
print(degCore)
```

```
[1] 40
[1] 279.375
```

There are 40 core nodes and the average degree is 279.375.

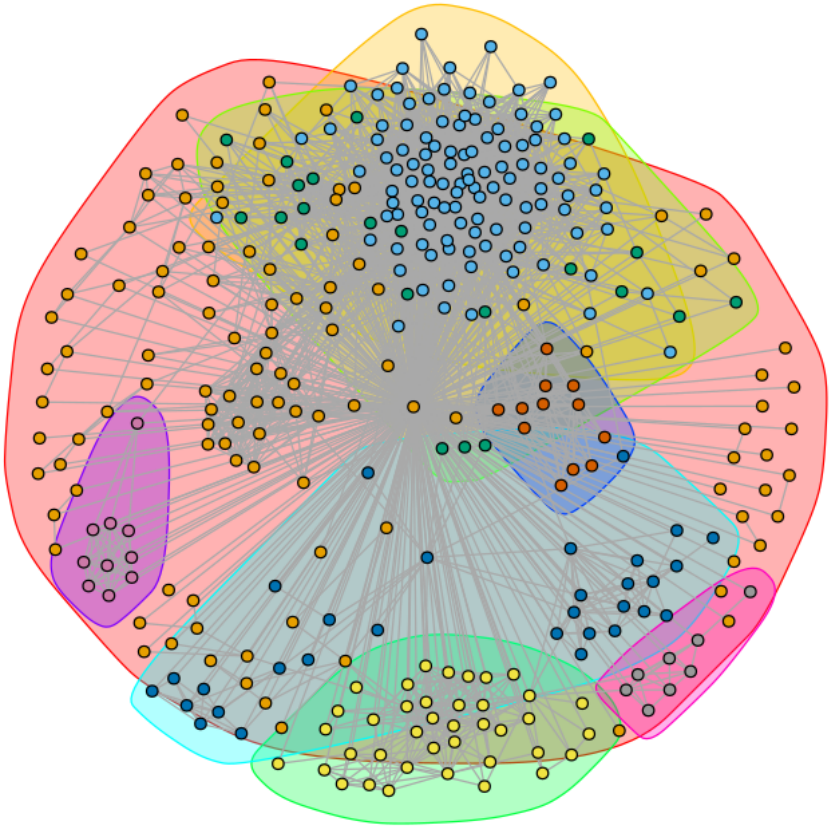
12 Q9

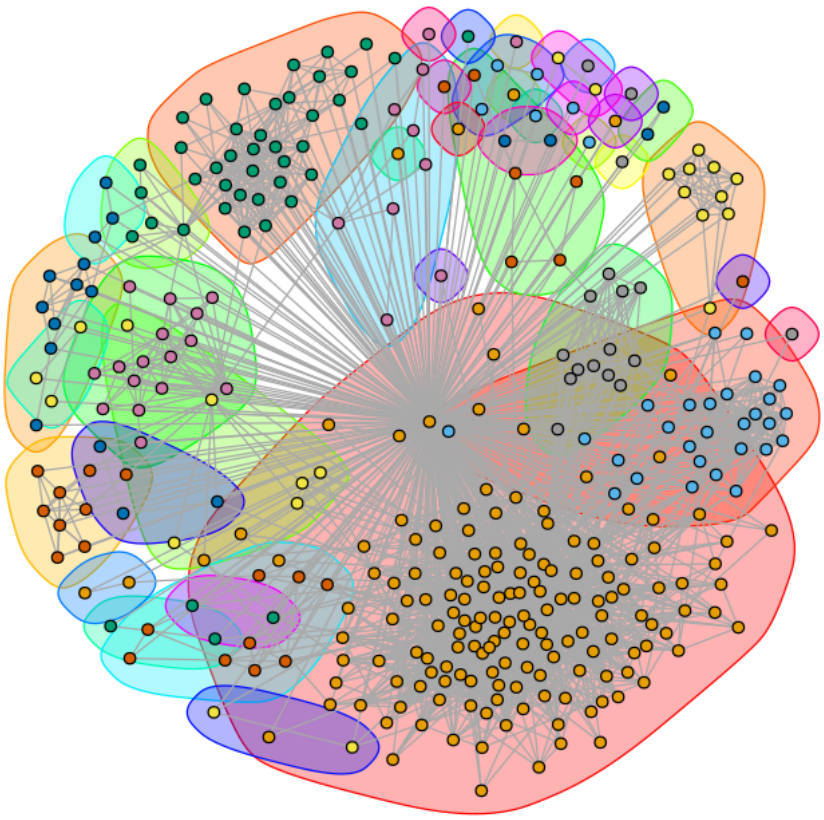
[ ]:
```
node_list <- c(1,108,349,484,1087)
g <- read.graph("facebook_combined.txt", format="edgelist", directed=FALSE)
ego <- make_ego_graph(g, order = 1, nodes = V(g)[node_list])
```

[ ]:
```
for(i in c(1:5)) {
  net = ego[[i]]
```

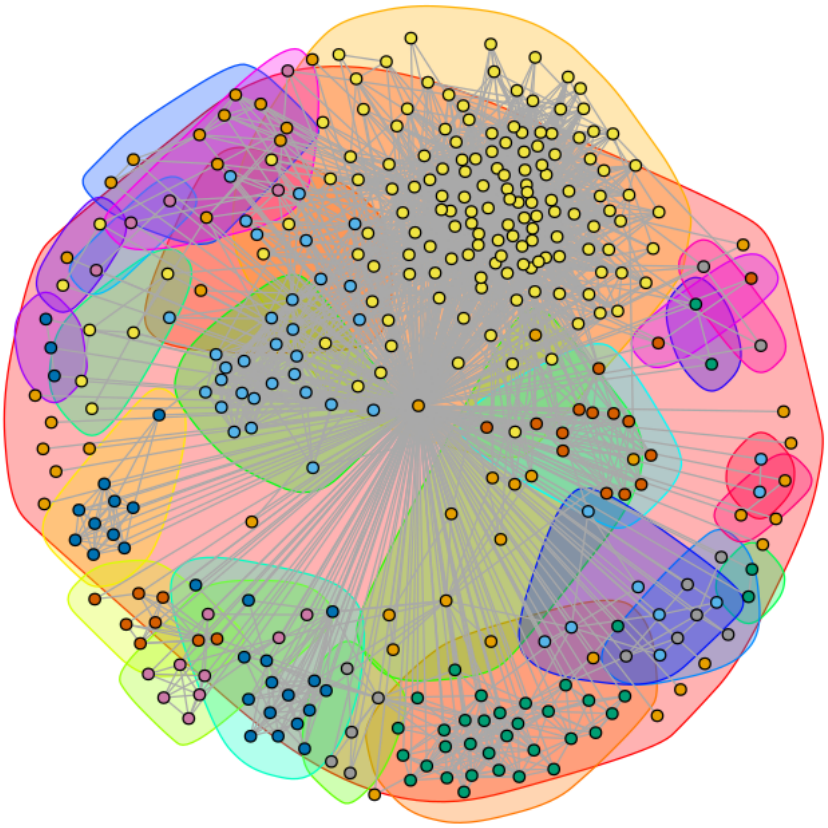
```
  fg <- cluster_fast_greedy(net)
  eb <- cluster_edge_betweenness(net)
  im <- cluster_infomap(net)
  print(sprintf("Modularity, Fast Greedy, Node ID: %d: %f",node_list[i],␣
 ↪modularity(fg)))
  print(sprintf("Modularity, Edge-Betweenness, Node ID: %d: %f",node_list[i],␣
 ↪modularity(eb)))
  print(sprintf("Modularity, Infomap, Node ID: %d: %f",node_list[i],␣
 ↪modularity(im)))
  plot(ego[[i]],mark.groups = fg,vertex.size=3,vertex.
 ↪color=fg$membership,vertex.label="",main = sprintf("Community Structure,␣
 ↪Fast Greedy, Node ID: %d", node_list[i]))
  plot(ego[[i]],mark.groups = eb,vertex.size=3,vertex.
 ↪color=eb$membership,vertex.label="",main = sprintf("Community Structure,␣
 ↪Edge-Betweenness, Node ID: %d", node_list[i]))
  plot(ego[[i]],mark.groups = im,vertex.size=3,vertex.
 ↪color=im$membership,vertex.label="",main = sprintf("Community Structure,␣
 ↪Infomap, Node ID: %d", node_list[i]))
}
```

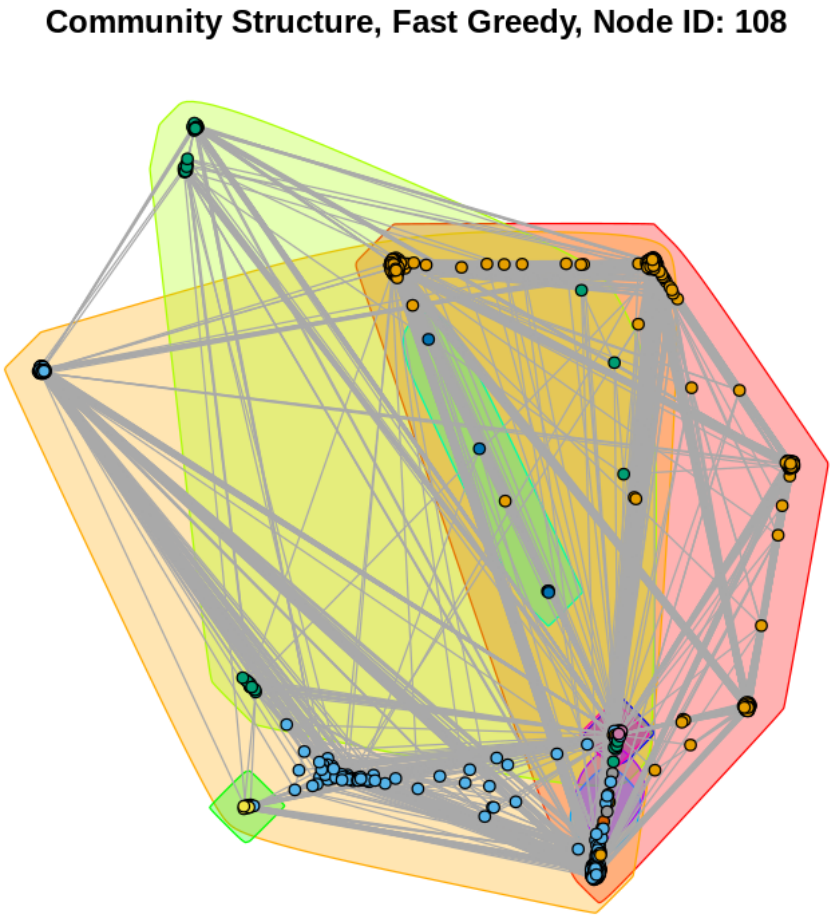
```
[1] "Modularity, Fast Greedy, Node ID: 1: 0.413101"
[1] "Modularity, Edge-Betweenness, Node ID: 1: 0.353302"
[1] "Modularity, Infomap, Node ID: 1: 0.389118"
```

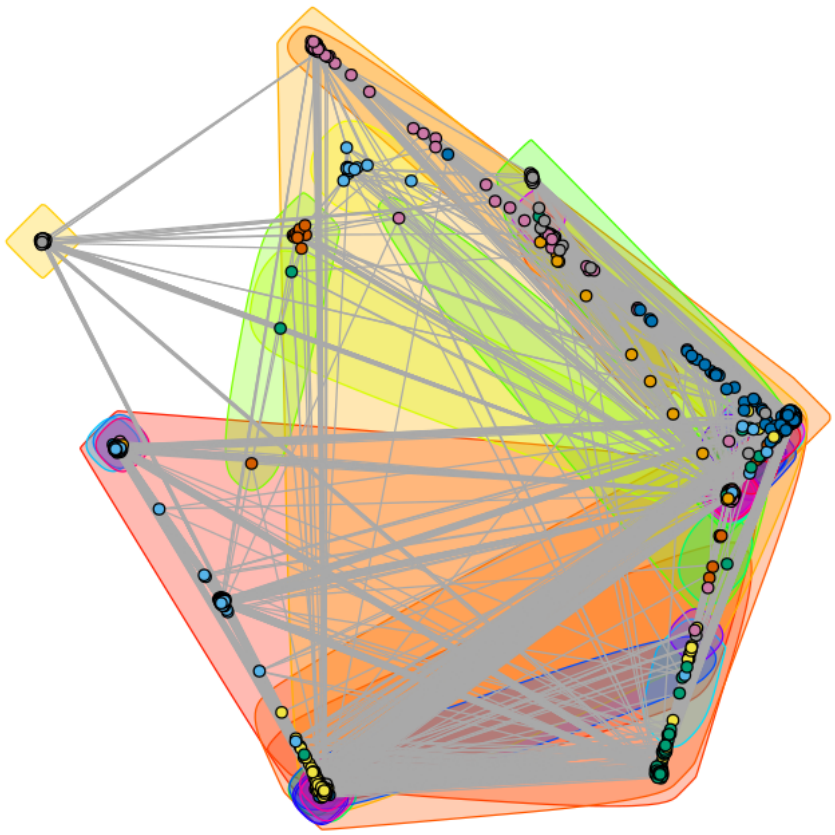

Community Structure, Fast Greedy, Node ID: 1



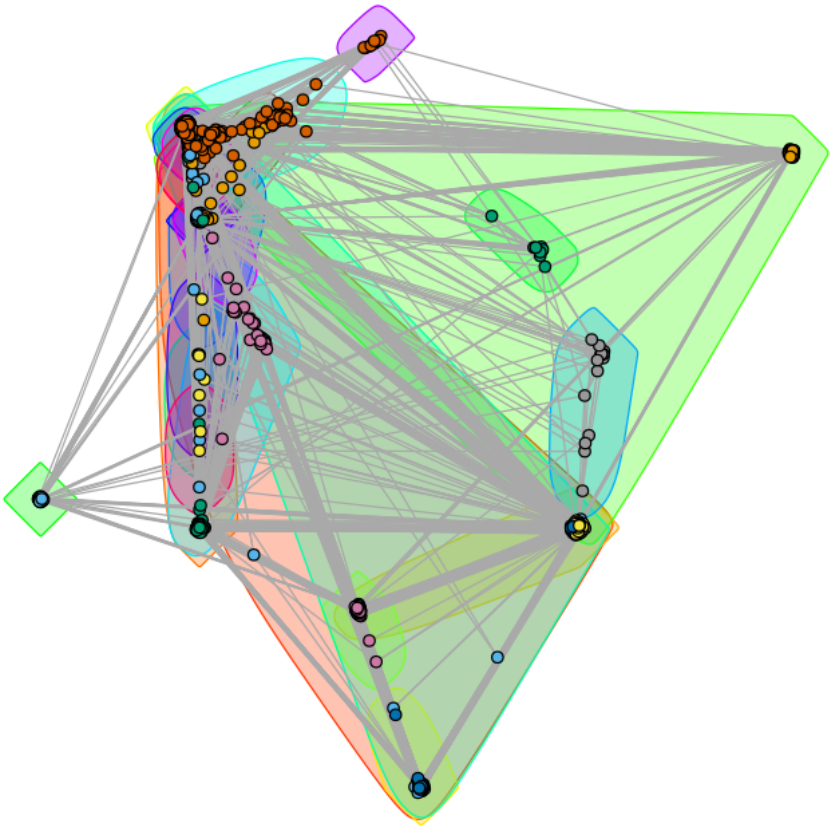

```
[1] "Modularity, Fast Greedy, Node ID: 108: 0.435929"
[1] "Modularity, Edge-Betweenness, Node ID: 108: 0.506755"
[1] "Modularity, Infomap, Node ID: 108: 0.508223"
```

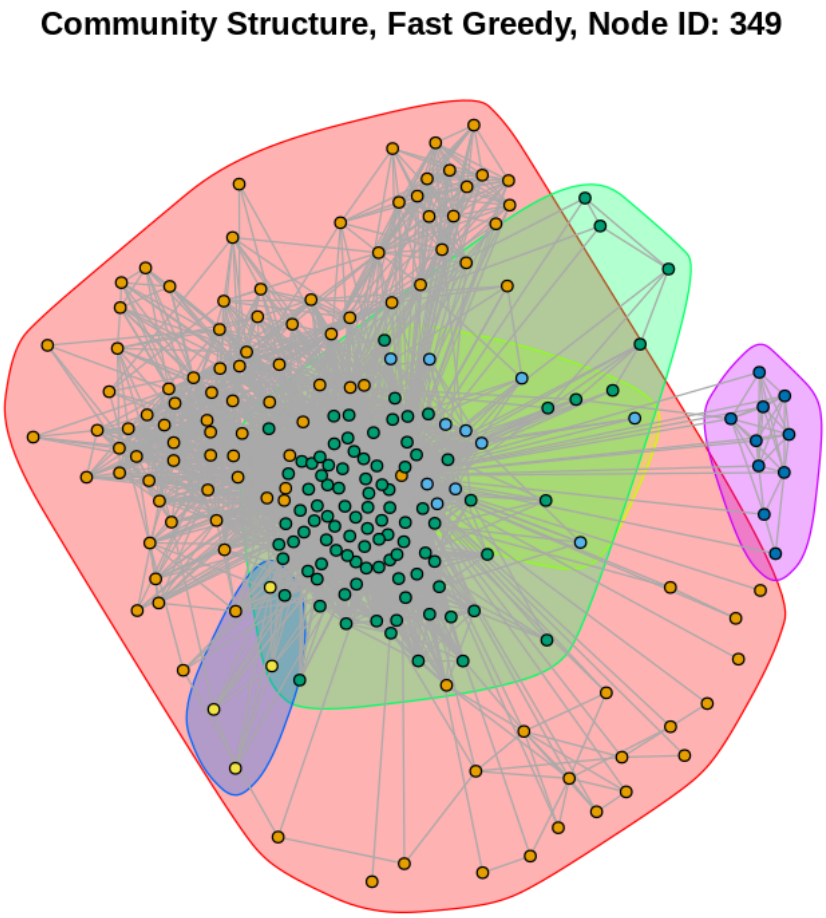

Community Structure, Infomap, Node ID: 1

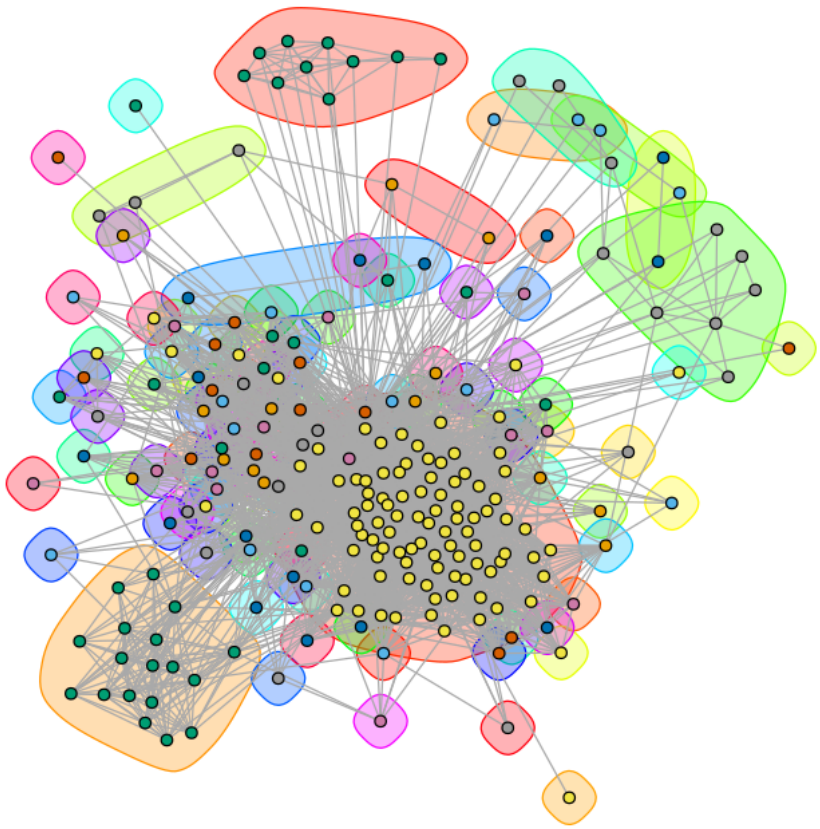

Community Structure, Fast Greedy, Node ID: 108



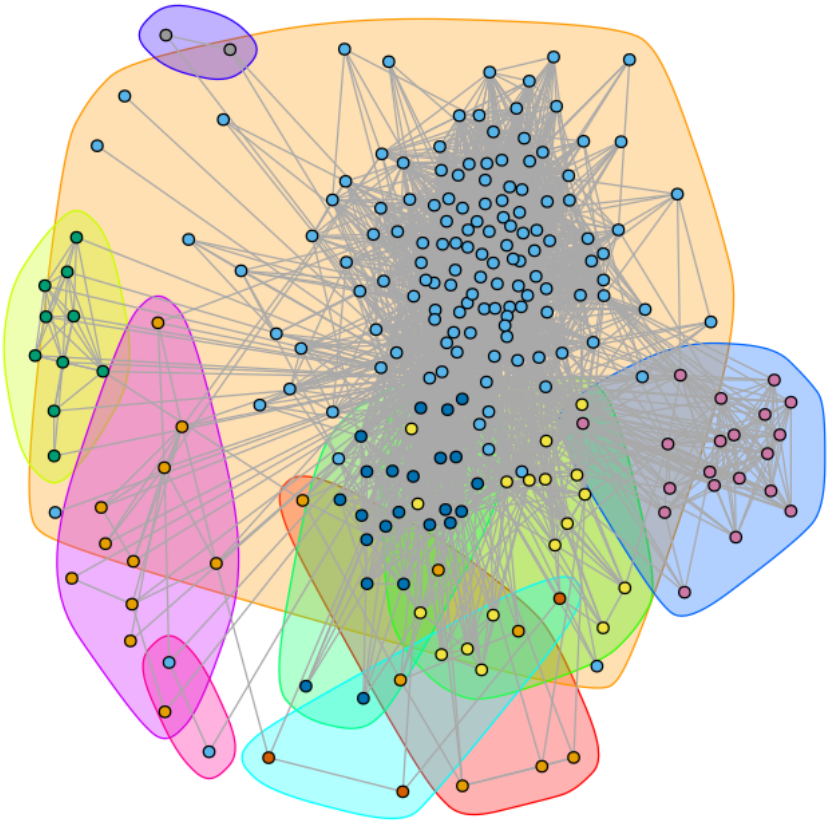

```
[1] "Modularity, Fast Greedy, Node ID: 349: 0.251715"
[1] "Modularity, Edge-Betweenness, Node ID: 349: 0.133528"
[1] "Modularity, Infomap, Node ID: 349: 0.203753"
```

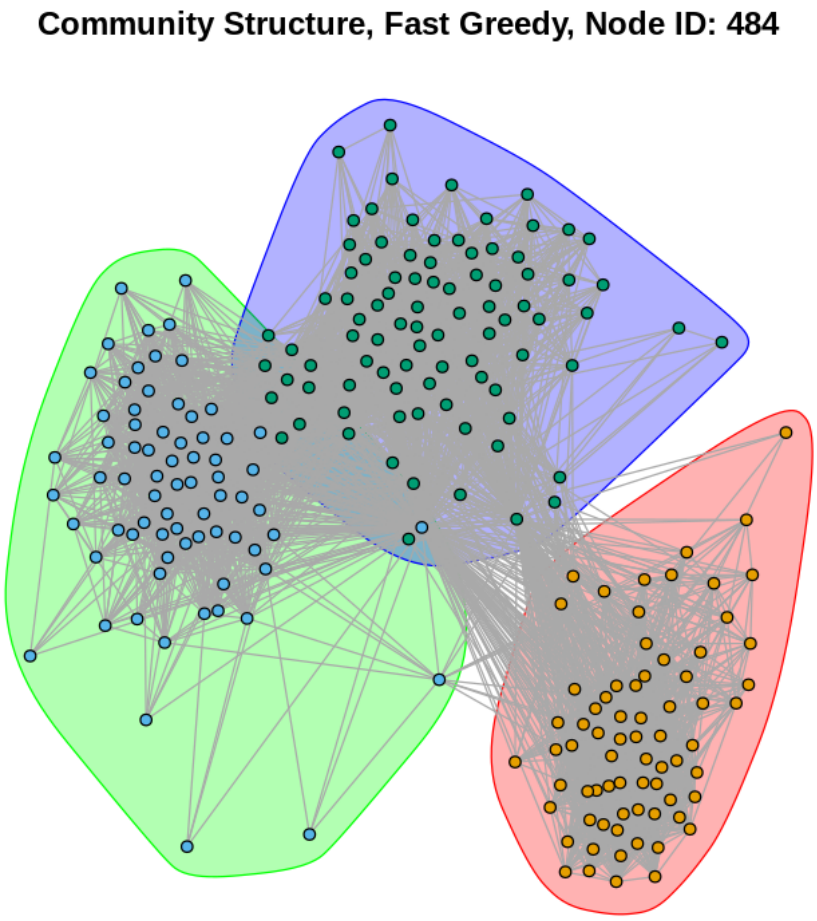

Community Structure, Infomap, Node ID: 108

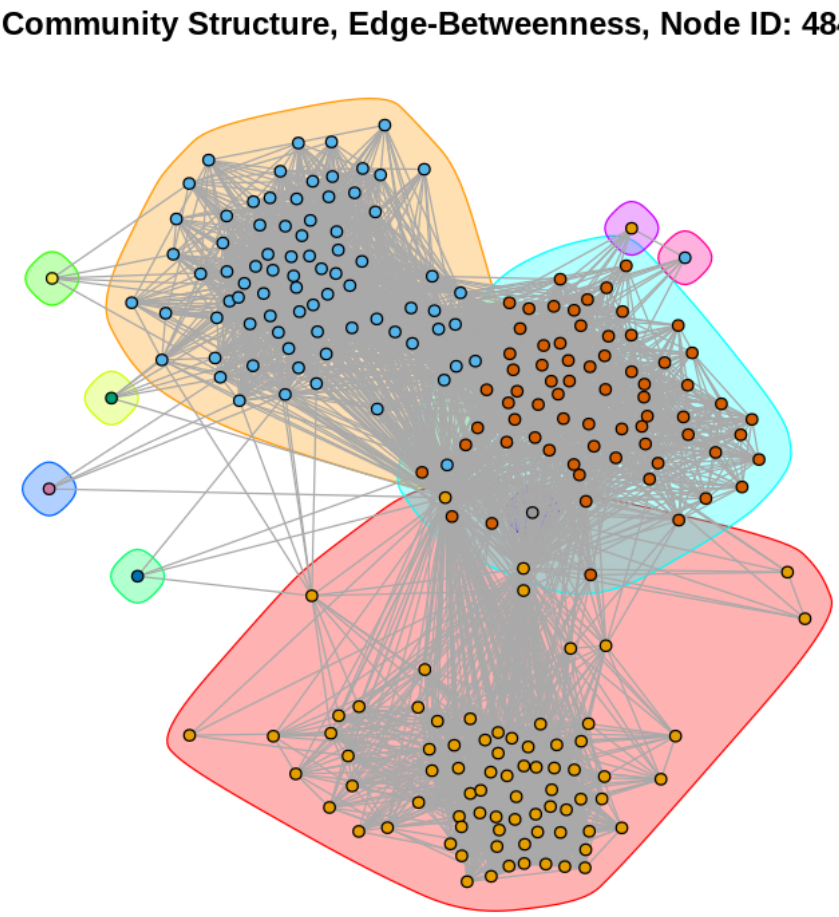

Community Structure, Fast Greedy, Node ID: 349



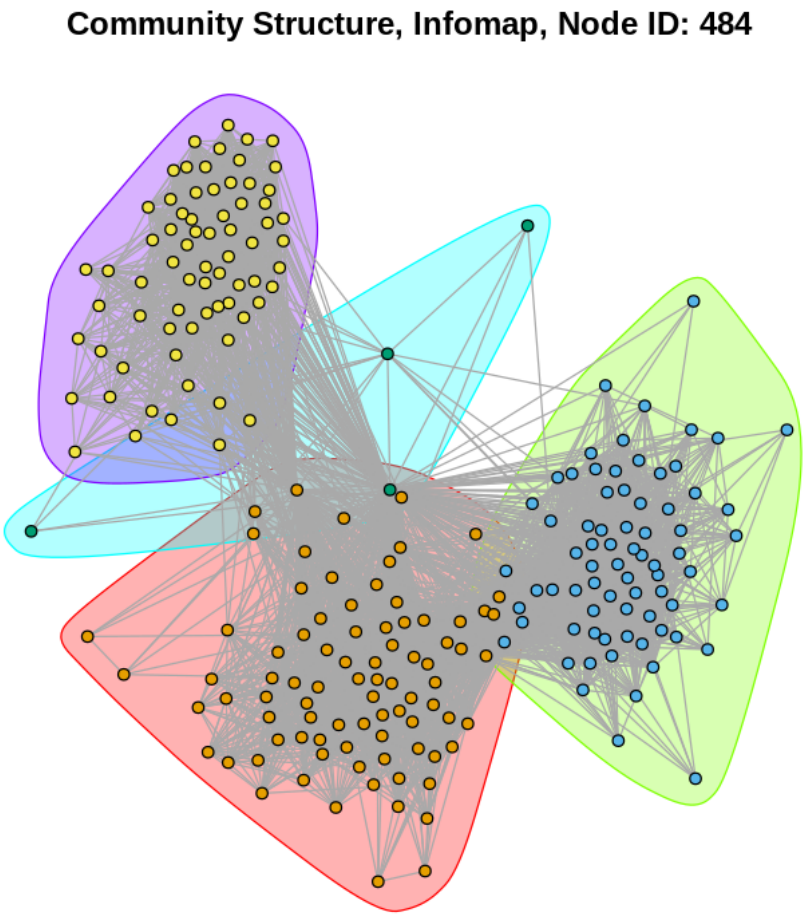

```
[1] "Modularity, Fast Greedy, Node ID: 484: 0.507002"
[1] "Modularity, Edge-Betweenness, Node ID: 484: 0.489095"
[1] "Modularity, Infomap, Node ID: 484: 0.515279"
```

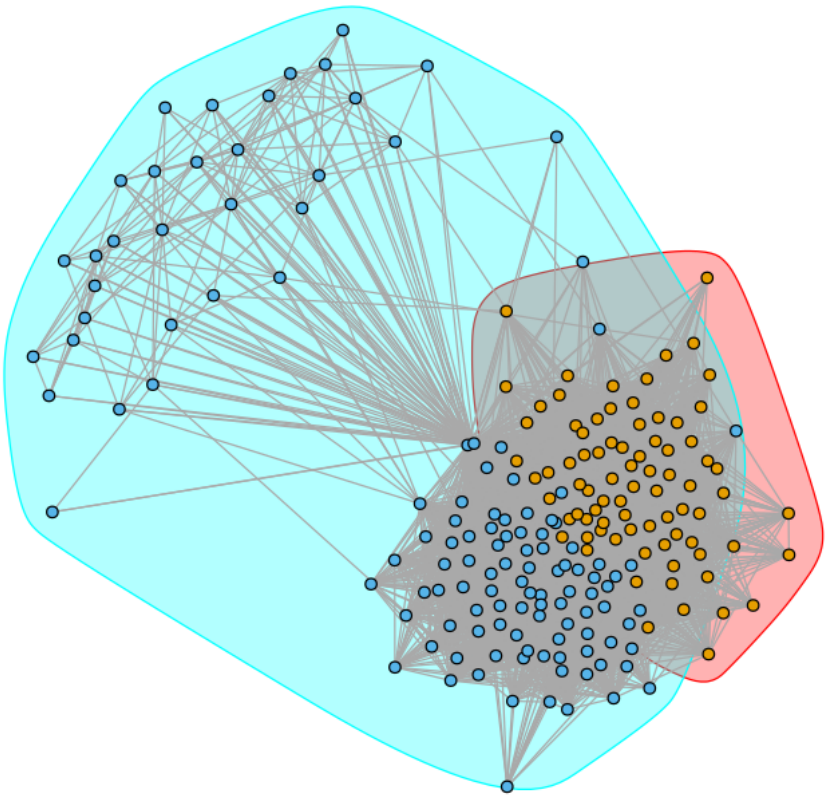

Community Structure, Infomap, Node ID: 349

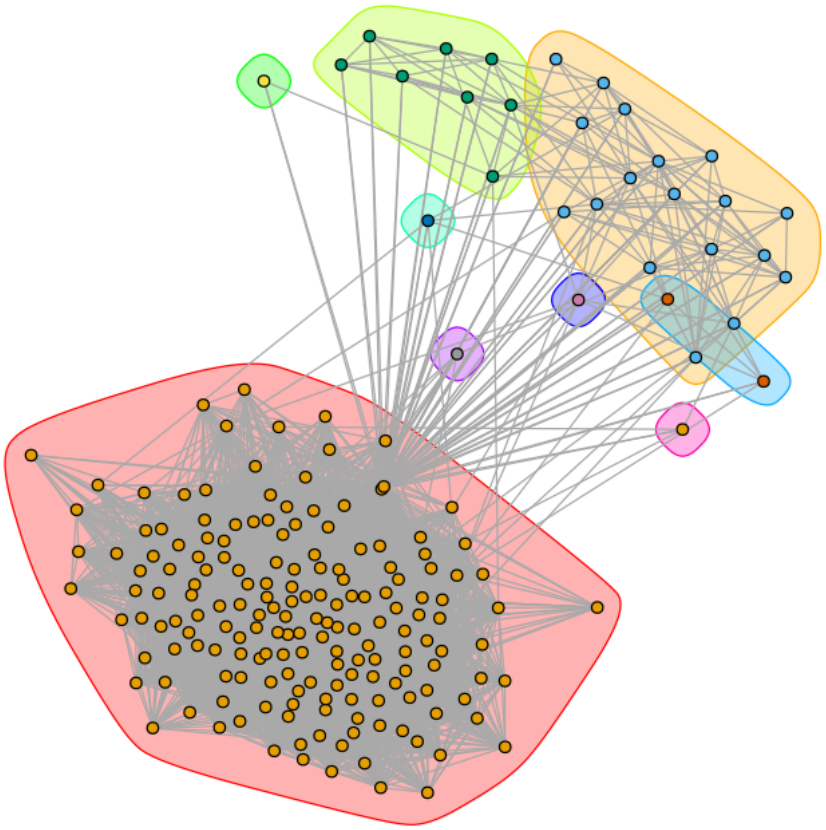

Community Structure, Fast Greedy, Node ID: 484

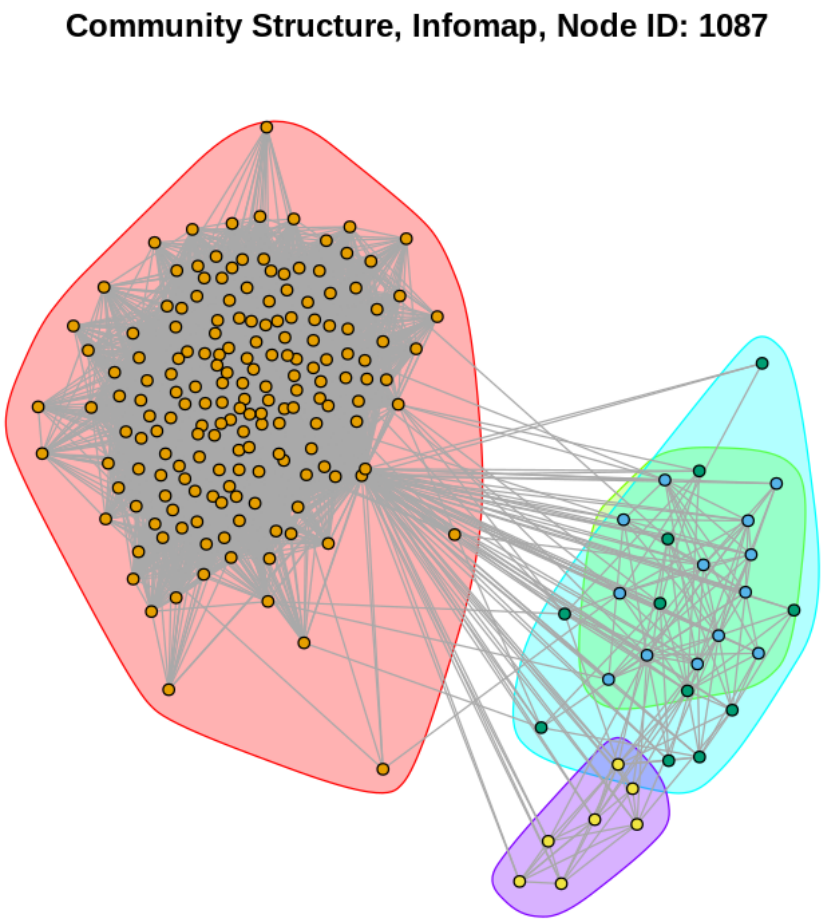



```
[1] "Modularity, Fast Greedy, Node ID: 1087: 0.145531"
[1] "Modularity, Edge-Betweenness, Node ID: 1087: 0.027624"
[1] "Modularity, Infomap, Node ID: 1087: 0.026907"
```


Community Structure, Infomap, Node ID: 484


Community Structure, Fast Greedy, Node ID: 1087


Community Structure, Edge-Betweenness, Node ID: 1087

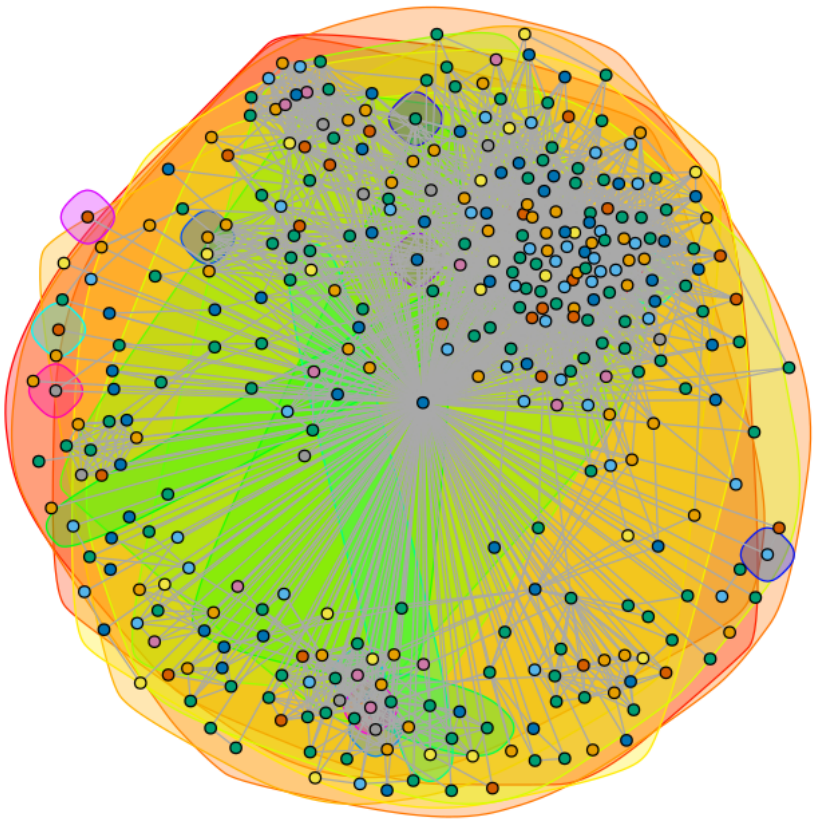


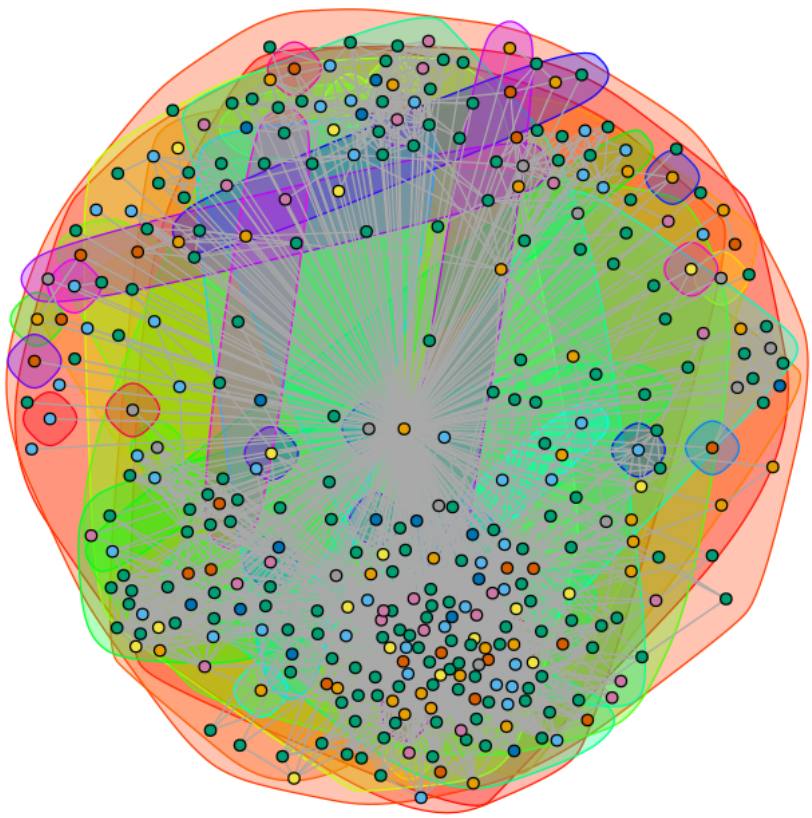
13 Q10

[ ]:
```r
for (i in c(1:5)) {
  net <- induced_subgraph(g, neighbors(g, node_list[i]))
  fg <- cluster_fast_greedy(net)
  eb <- cluster_edge_betweenness(net)
  im <- cluster_infomap(net)
  print(sprintf("Modularity, Fast Greedy, Node ID (without core node): %d:␣
 ↪%f",node_list[i], modularity(fg)))
  print(sprintf("Modularity, Edge-Betweenness, Node ID (without core node): %d:␣
 ↪%f",node_list[i], modularity(eb)))
```

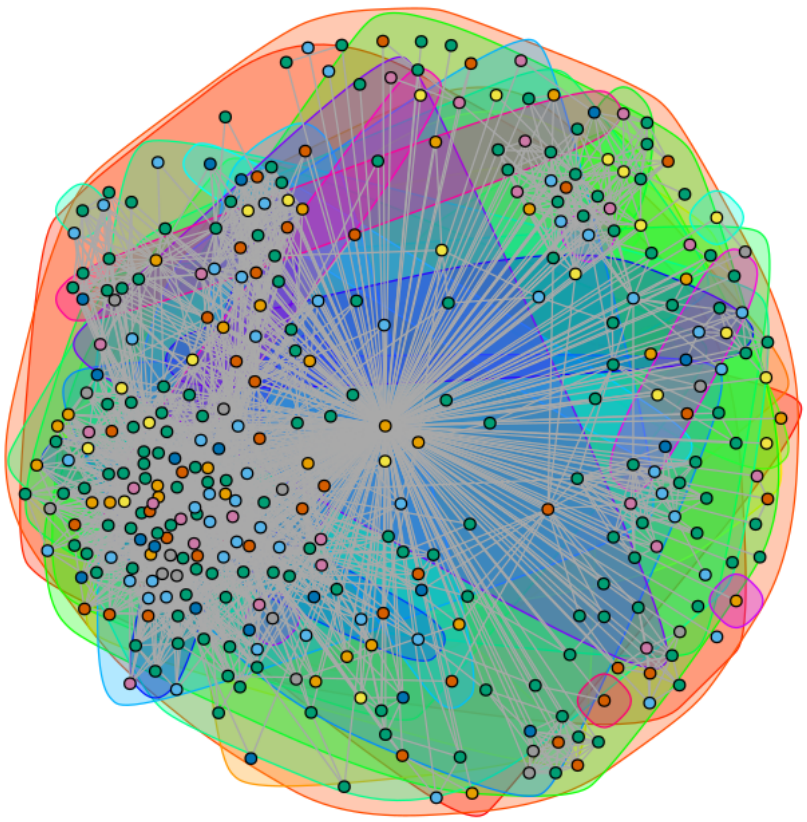
```
  print(sprintf("Modularity, Infomap, Node ID (without core node): %d:␣
 ↪%f",node_list[i], modularity(im)))
  plot(ego[[i]],mark.groups = fg,vertex.size=3,edge.arrow.size=.5,vertex.
 ↪color=fg$membership,vertex.label="",main = sprintf("Community Structure,␣
 ↪Fast Greedy, (WCN) Node ID: %d", node_list[i]))
  plot(ego[[i]],mark.groups = eb,vertex.size=3,edge.arrow.size=.5,vertex.
 ↪color=eb$membership ,vertex.label="",main = sprintf("Community Structure,␣
 ↪Edge-Betweenness, (WCN) Node ID: %d", node_list[i]))
  plot(ego[[i]],mark.groups = im,vertex.size=3,edge.arrow.size=.5,vertex.
 ↪color=im$membership,vertex.label="",main = sprintf("Community Structure,␣
 ↪Infomap, (WCN) Node ID: %d", node_list[i]))
}
```

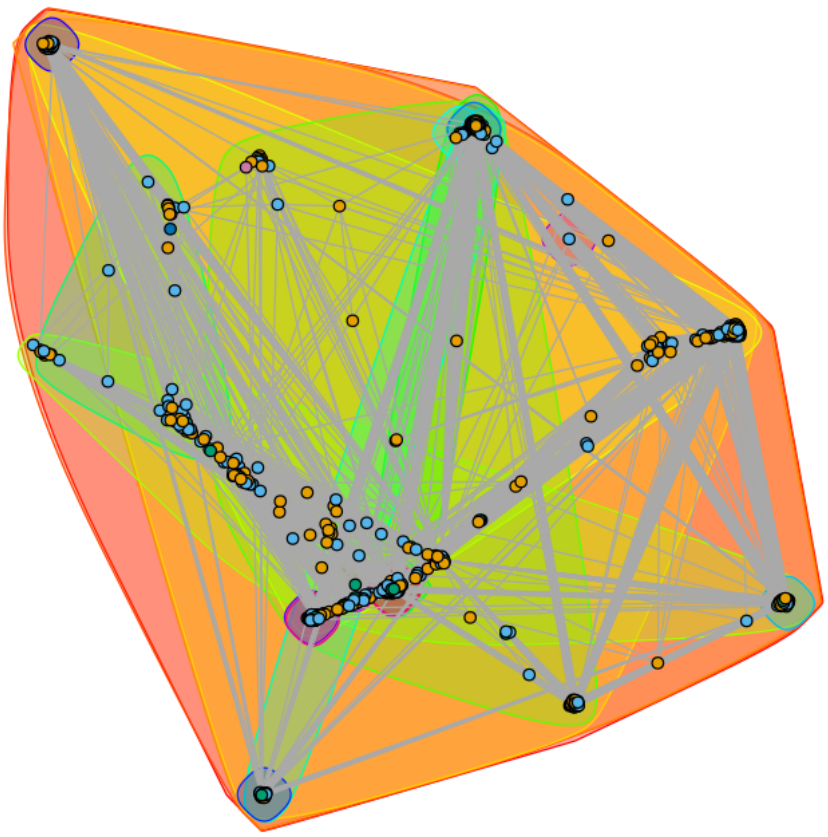
```
[1] "Modularity, Fast Greedy, Node ID (without core node): 1: 0.441853"
[1] "Modularity, Edge-Betweenness, Node ID (without core node): 1: 0.416146"
[1] "Modularity, Infomap, Node ID (without core node): 1: 0.418008"
```

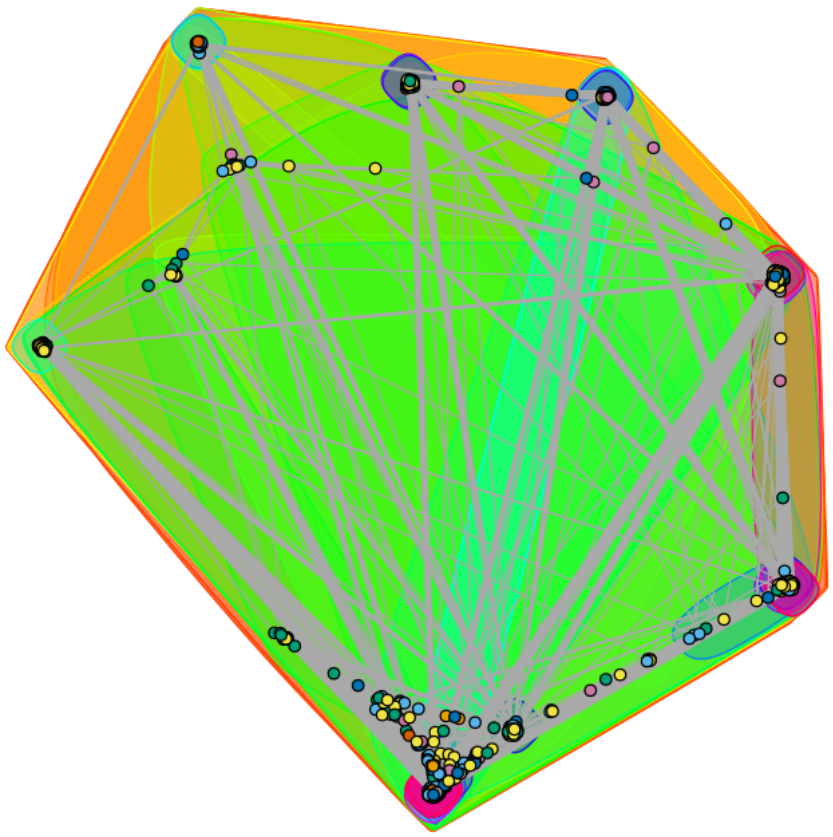

Community Structure, Fast Greedy, (WCN) Node ID: 1



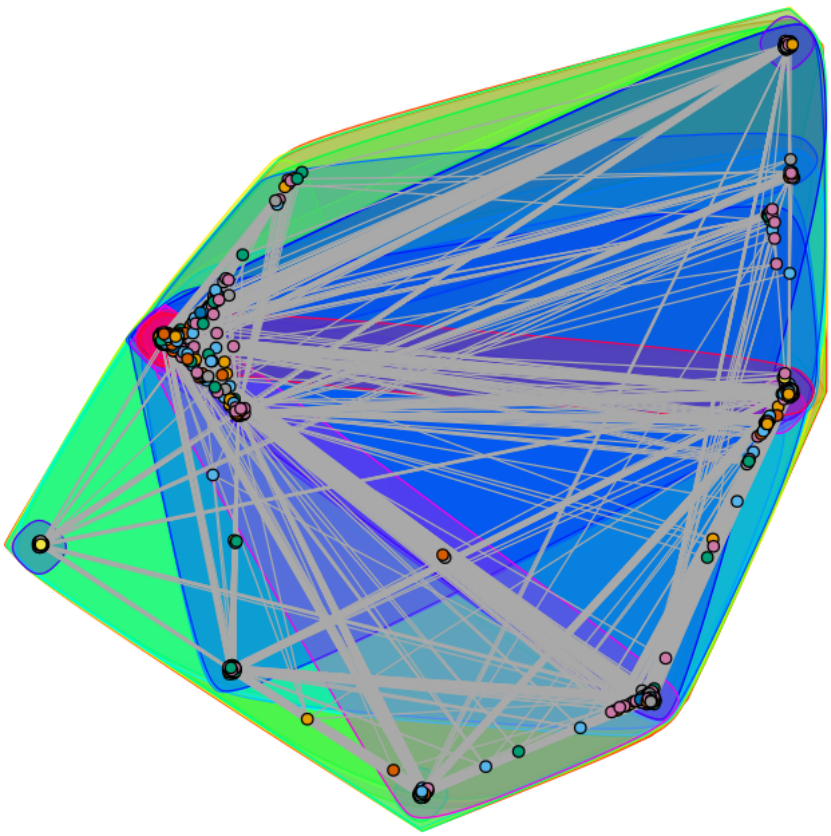

```
[1] "Modularity, Fast Greedy, Node ID (without core node): 108: 0.458127"
[1] "Modularity, Edge-Betweenness, Node ID (without core node): 108: 0.521322"
[1] "Modularity, Infomap, Node ID (without core node): 108: 0.520760"
```

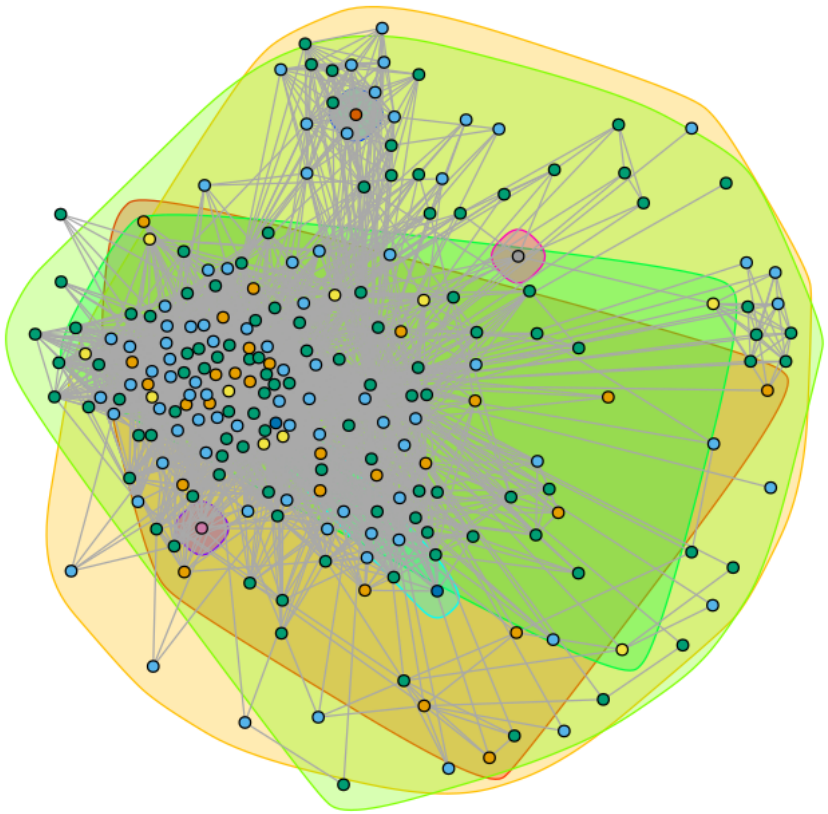

Community Structure, Infomap, (WCN) Node ID: 1

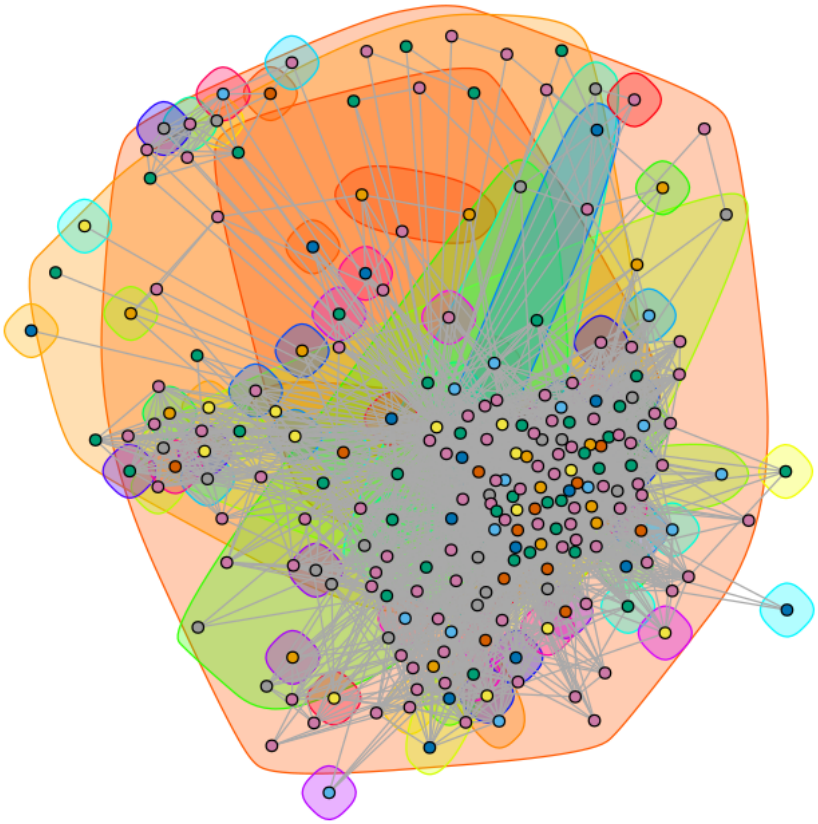

Community Structure, Fast Greedy, (WCN) Node ID: 108



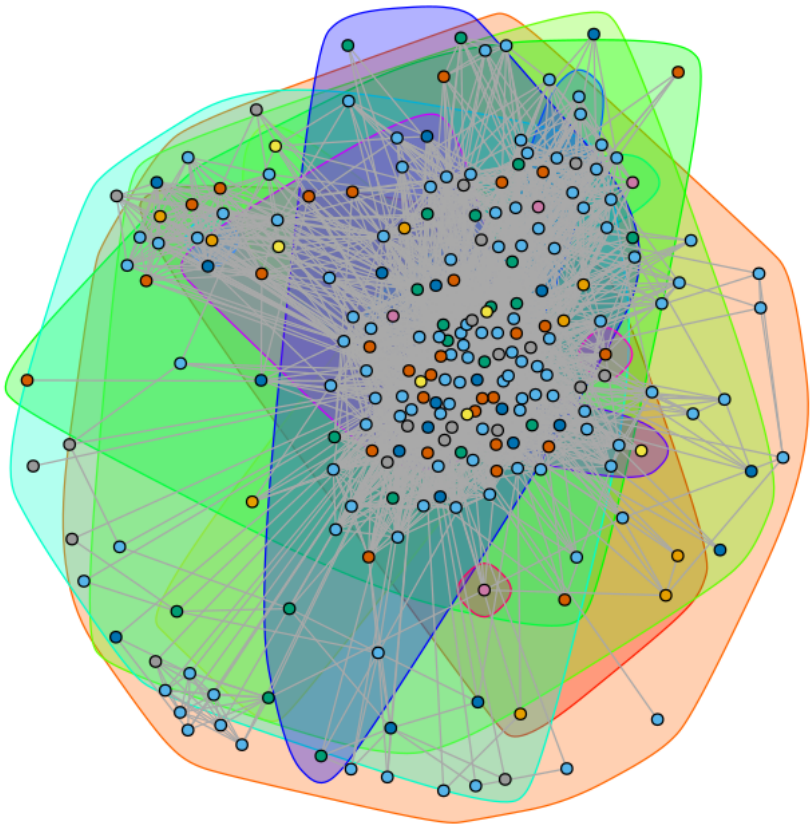

```
[1] "Modularity, Fast Greedy, Node ID (without core node): 349: 0.245692"
[1] "Modularity, Edge-Betweenness, Node ID (without core node): 349: 0.150566"
[1] "Modularity, Infomap, Node ID (without core node): 349: 0.244816"
```

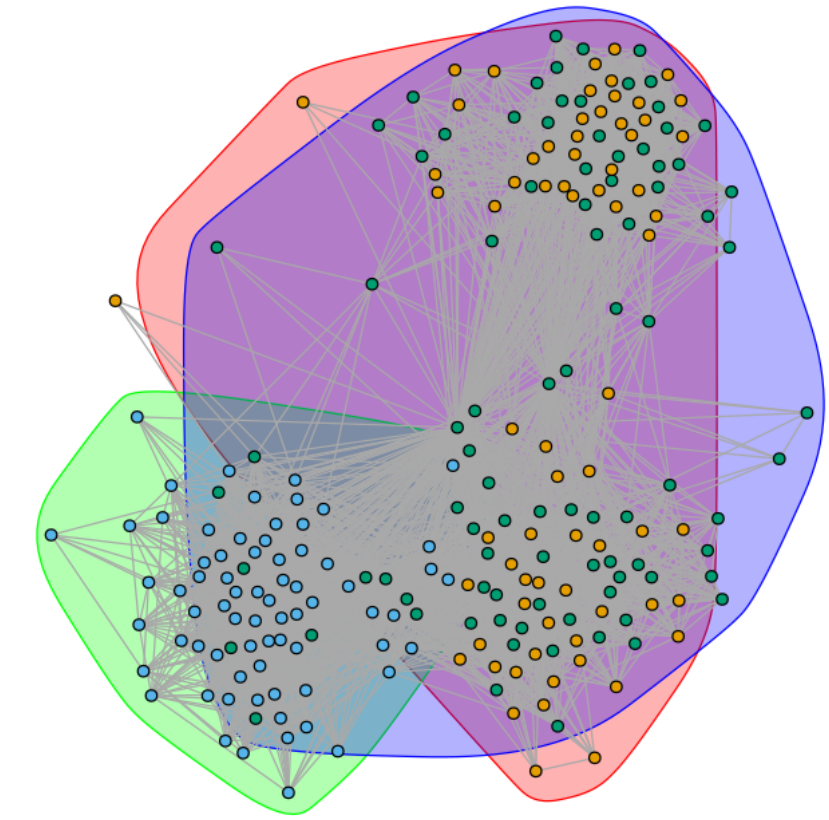

Community Structure, Infomap, (WCN) Node ID: 108

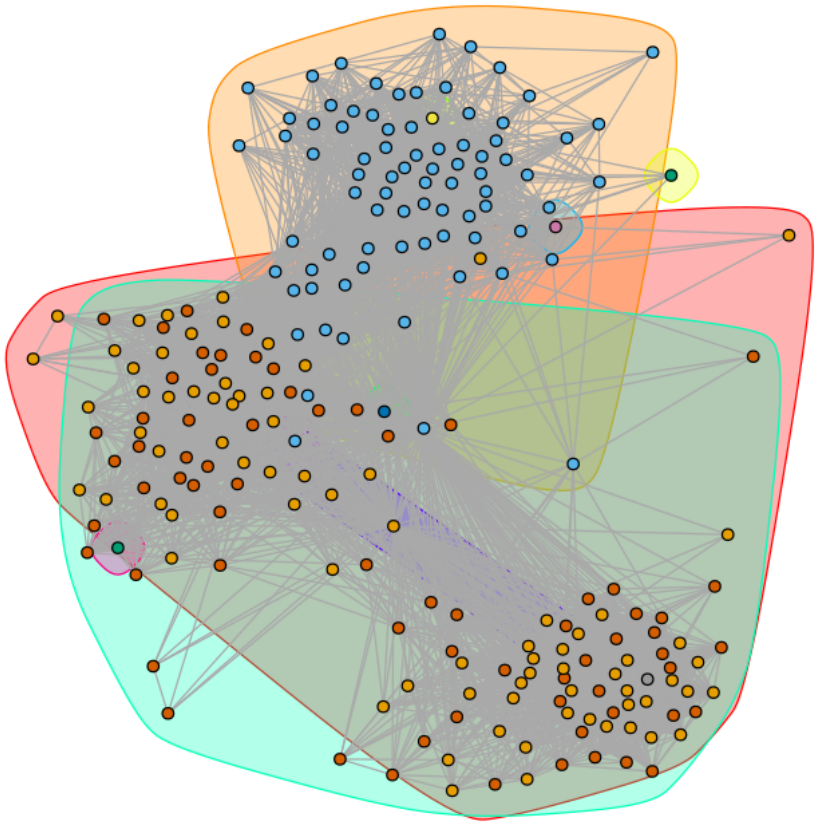

Community Structure, Fast Greedy, (WCN) Node ID: 349



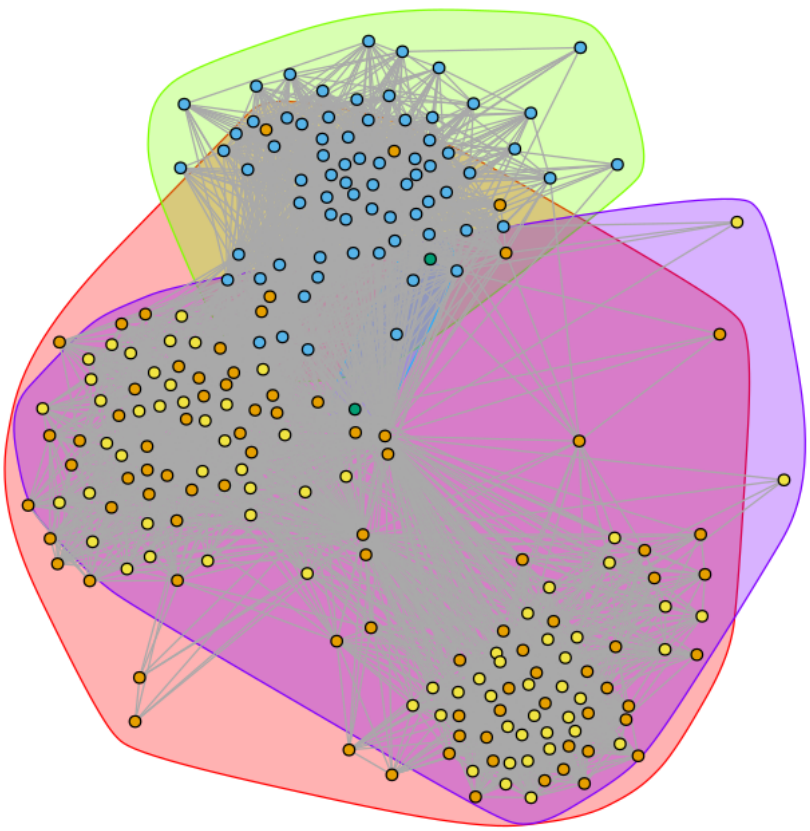

```
[1] "Modularity, Fast Greedy, Node ID (without core node): 484: 0.534214"
[1] "Modularity, Edge-Betweenness, Node ID (without core node): 484: 0.515441"
[1] "Modularity, Infomap, Node ID (without core node): 484: 0.543444"
```

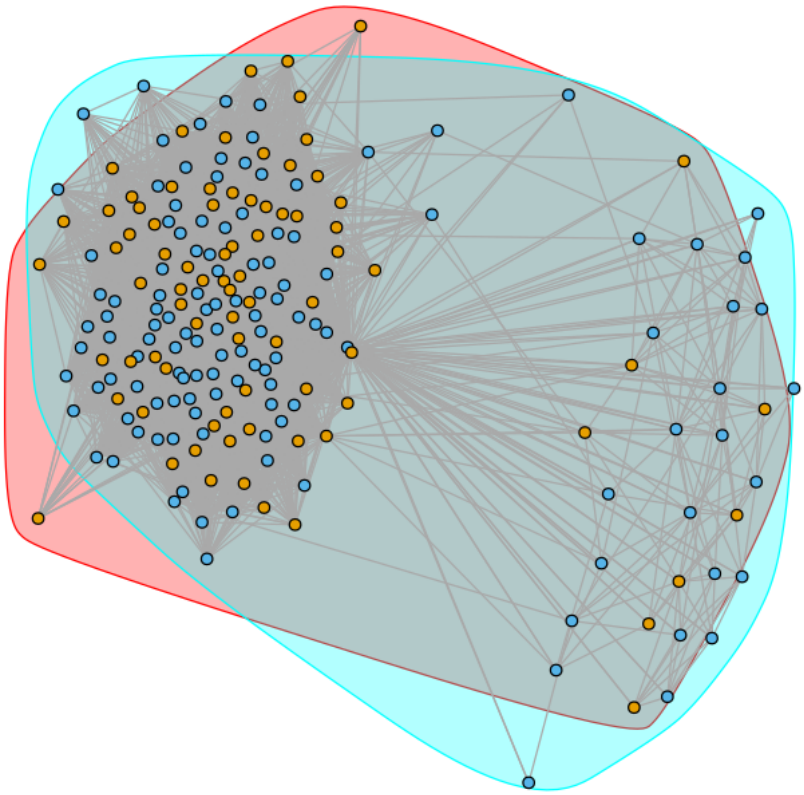

Community Structure, Infomap, (WCN) Node ID: 349

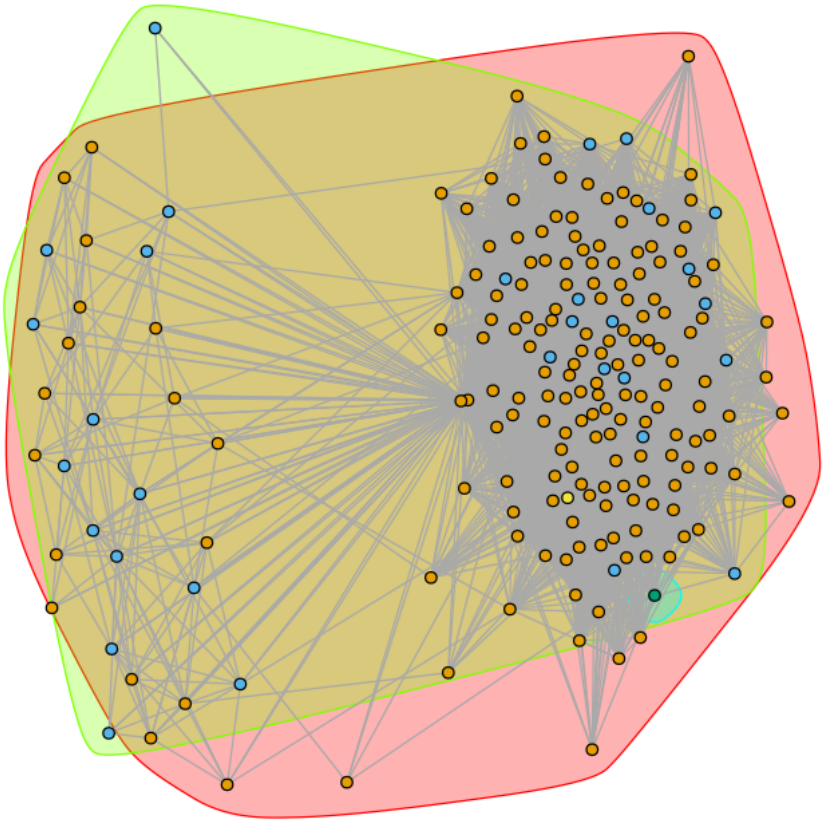

Community Structure, Fast Greedy, (WCN) Node ID: 484

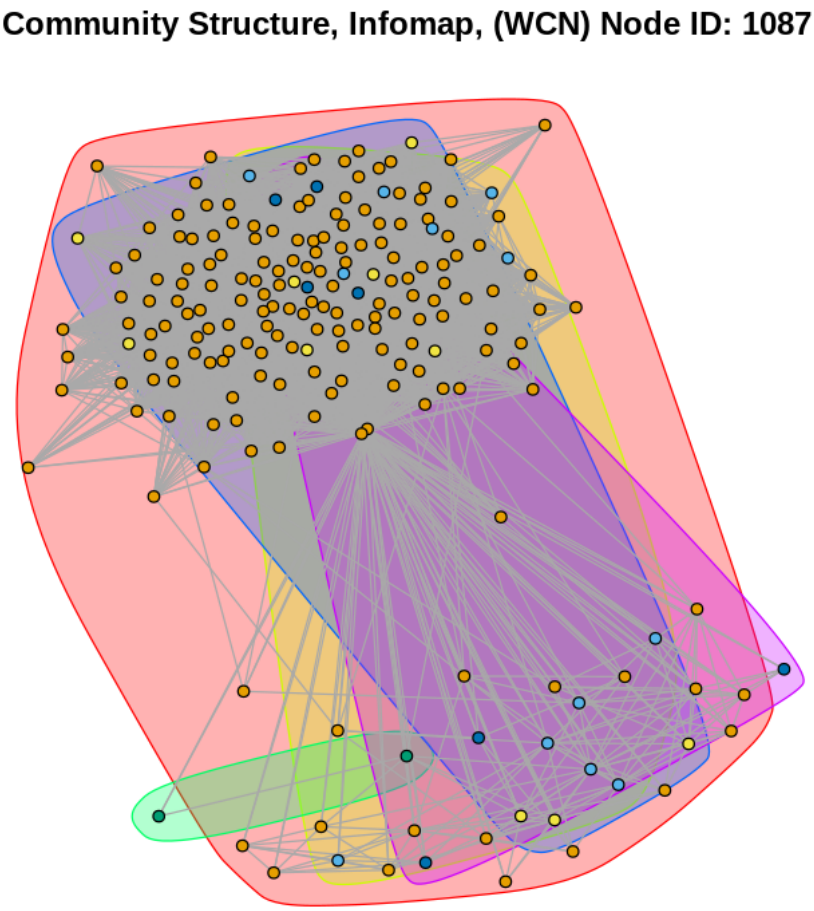



```
[1] "Modularity, Fast Greedy, Node ID (without core node): 1087: 0.148196"
[1] "Modularity, Edge-Betweenness, Node ID (without core node): 1087: 0.032495"
[1] "Modularity, Infomap, Node ID (without core node): 1087: 0.027372"
```


Community Structure, Infomap, (WCN) Node ID: 484


Community Structure, Fast Greedy, (WCN) Node ID: 1087


Community Structure, Edge-Betweenness, (WCN) Node ID: 1087



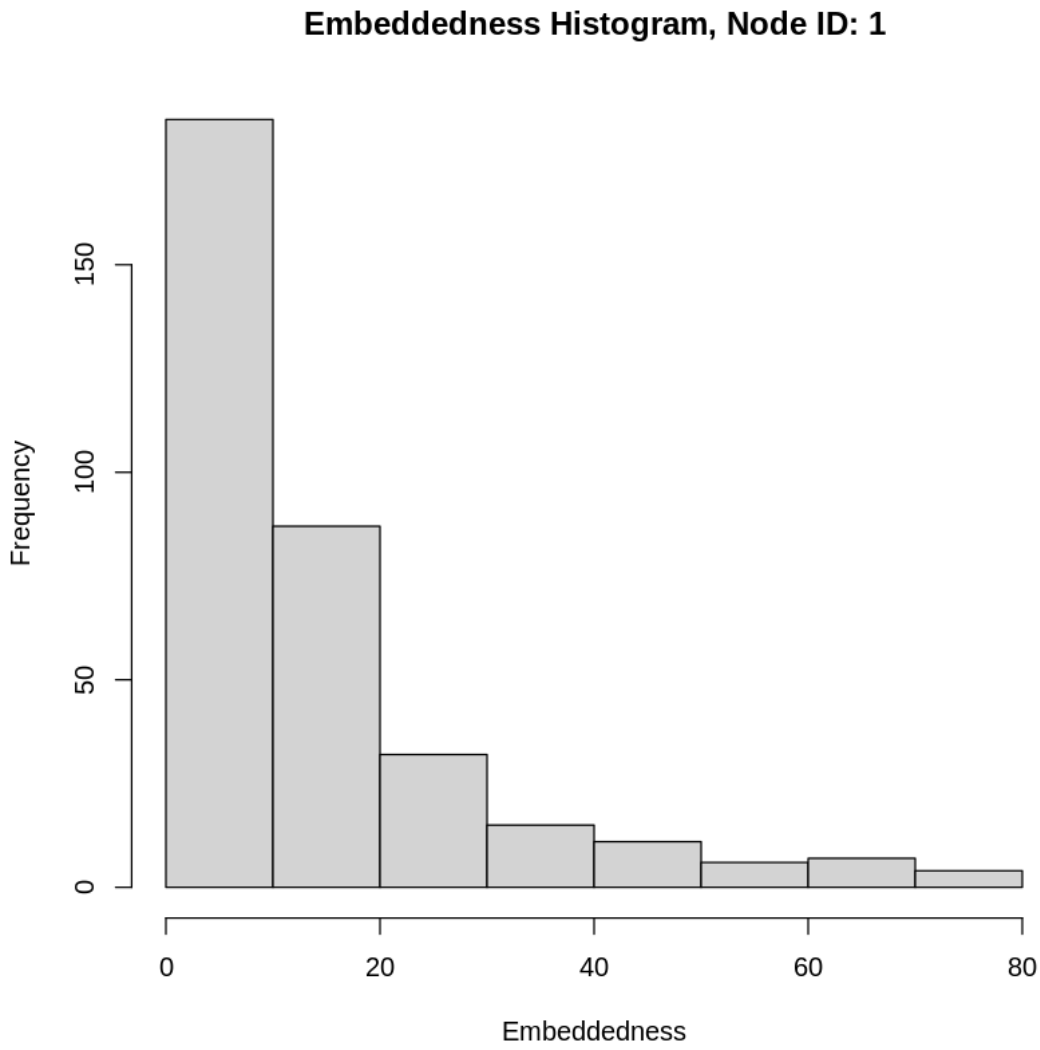
14 Q11

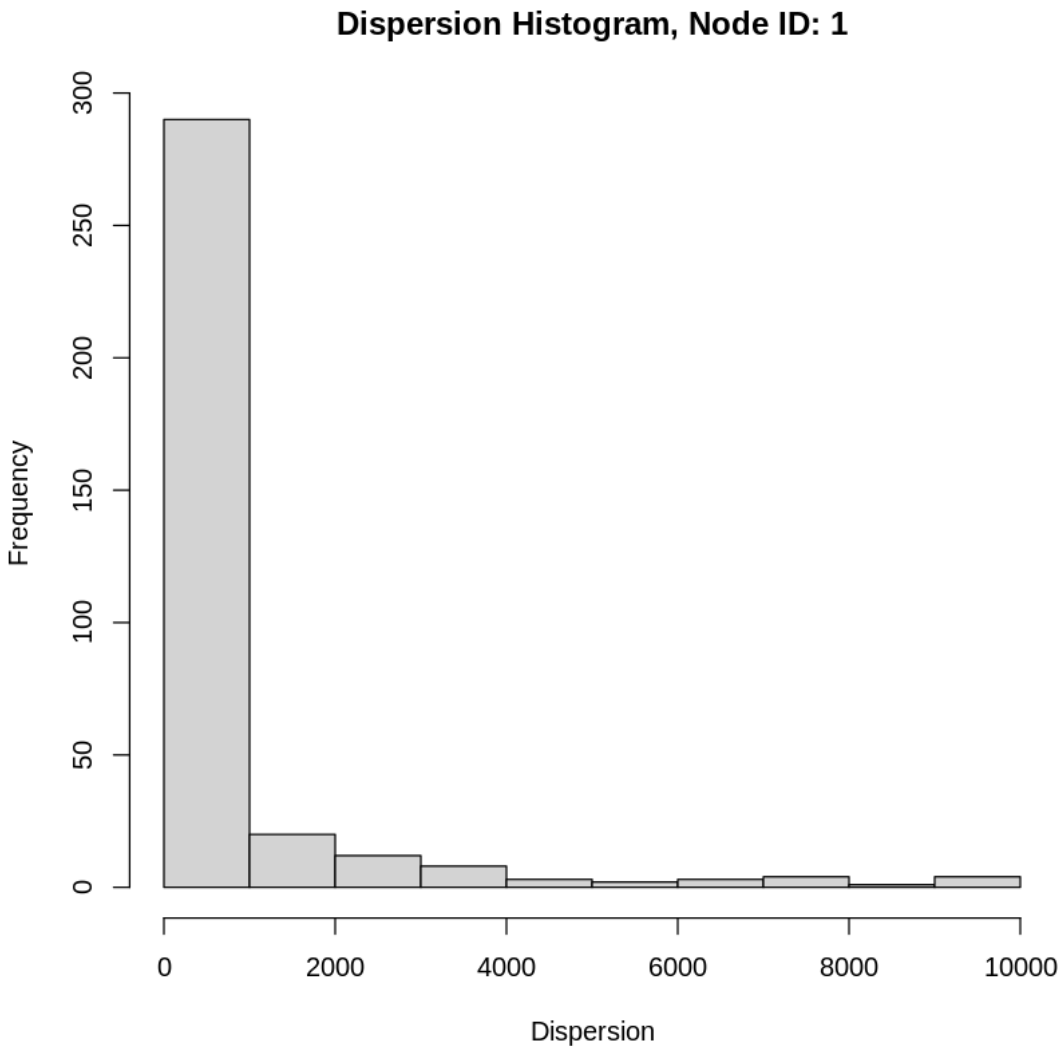
The embeddedness of a node v is defined as the number of mutual vertices a given node shares with the core node v_c. Hence, we can formulate our equation as following

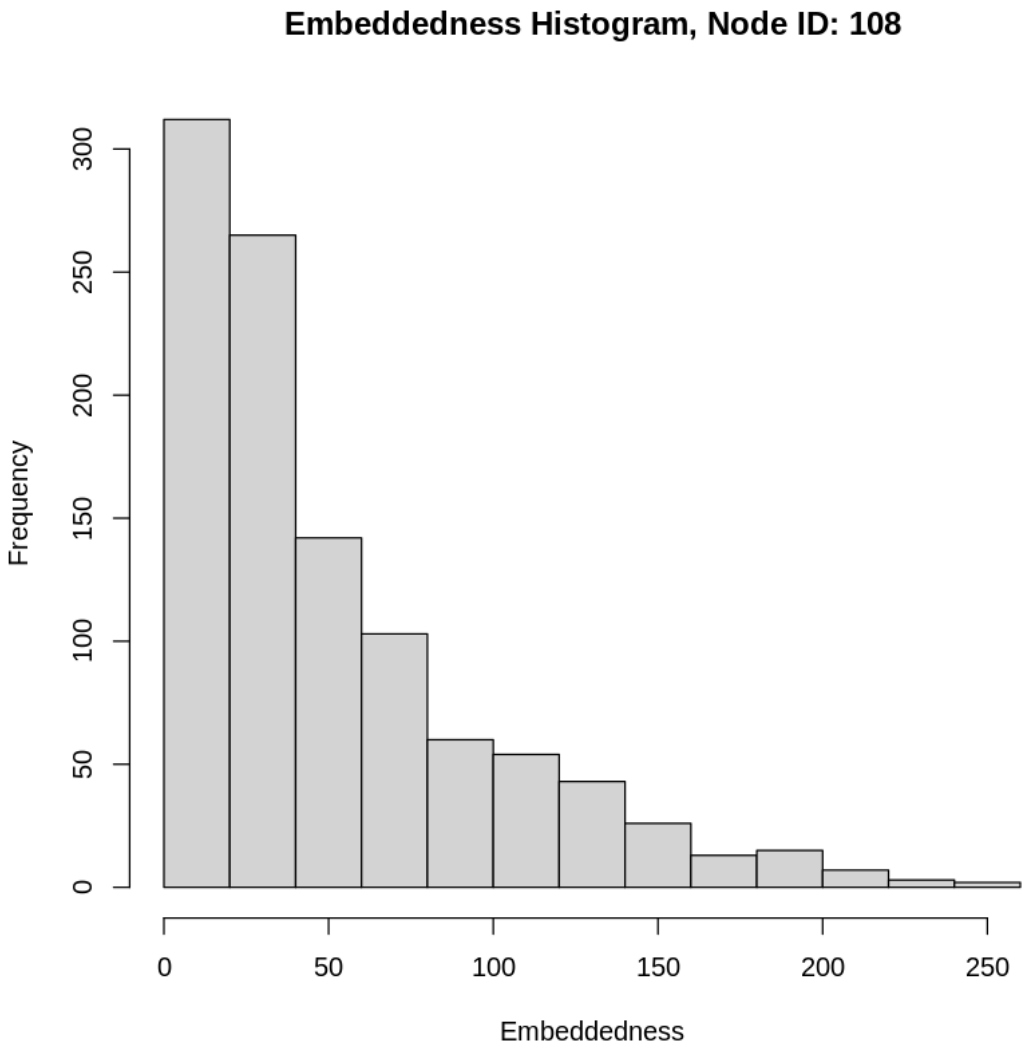
$Embeddedness_v = |deg(v_c) \cap deg(v)| - 1$, where $deg(v)$ is the set of adjacent vertices of v

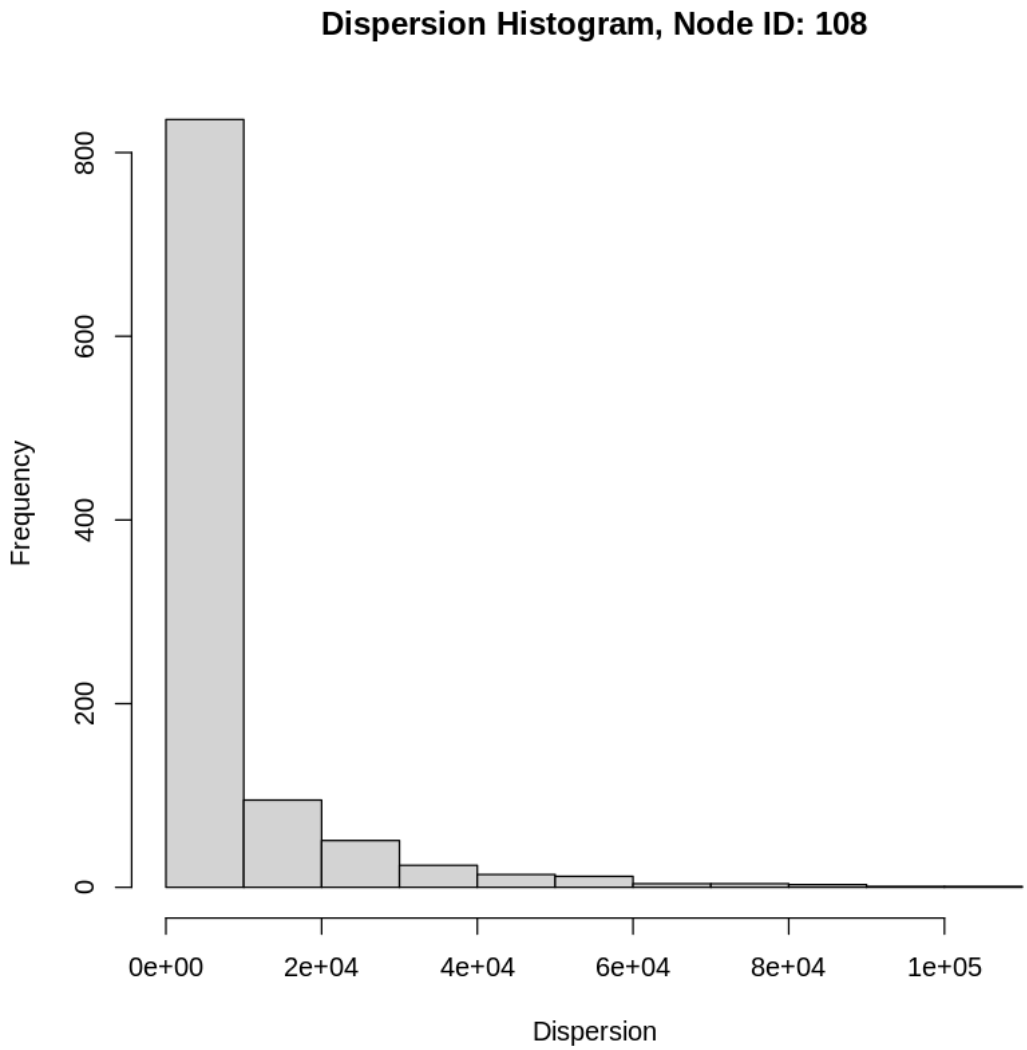
15 Q12

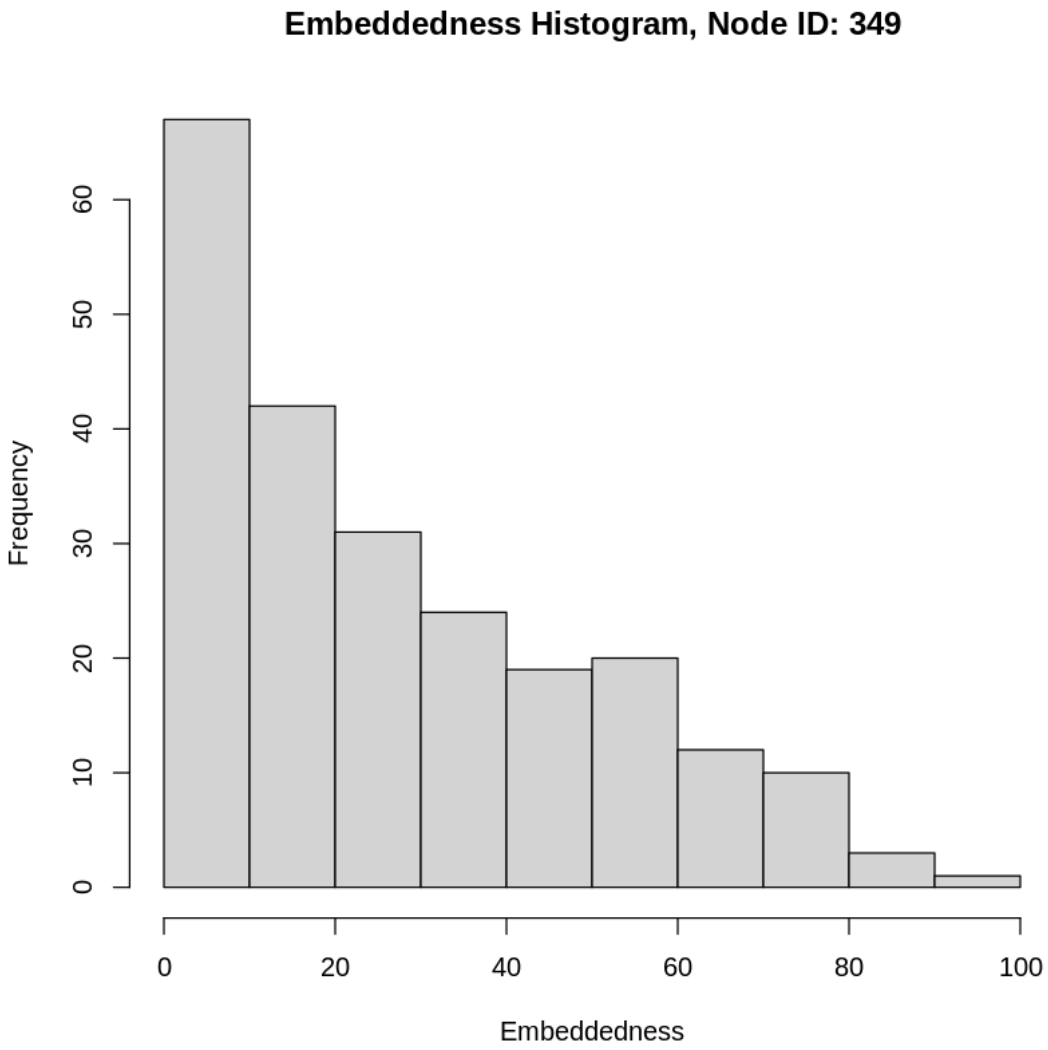
[ ]:
```
node_list_str = c("0","107","348","483","1086")
for(j in c(1:5)){

    nodelist = unlist(ego(g, order=1, nodes=node_list_str[j]))
    pn = induced.subgraph(g,nodelist)
    pn$name = sort(nodelist)
    embeddedness <- c()
    disp <- c()
    deg <- c()
    i=1
    for(v in vertex_attr(pn)$name){
        if(v==node_list_str[j])
            next
        disp[i] = 0
        neh_ver = neighbors(pn, node_list_str[j])
        neh_core = neighbors(pn,v)
        inter = intersection(neh_ver, neh_core)
        embeddedness[i] = length(inter)
        deg[i] = degree(pn,v)
        eg2 = delete.vertices(pn, c(node_list_str[j], v))
        if(embeddedness[i]>1){
            ver = c()
            for(m in 1:length(inter)){
                ver = c(ver,vertex_attr(pn)$name[inter[m]])
            }
            ver1 = c()
            for(m in 1:length(ver)){
                ver1=c(ver1,which(vertex_attr(eg2)$name==ver[m]))
            }
            disp_mat = distances(eg2,v=ver1, to=ver1)
            disp_mat[disp_mat==Inf]<-diameter(eg2)+1
            disp[i] = sum(disp_mat)
        }
        i=i+1

    }
    hist(embeddedness,main=sprintf("Embeddedness Histogram, Node ID:␣
 ↪%d",strtoi(node_list_str[j])+1), xlab="Embeddedness",ylab="Frequency")
    hist(disp,main=sprintf("Dispersion Histogram, Node ID:␣
 ↪%d",strtoi(node_list_str[j])+1), xlab="Dispersion",ylab="Frequency")
}
```

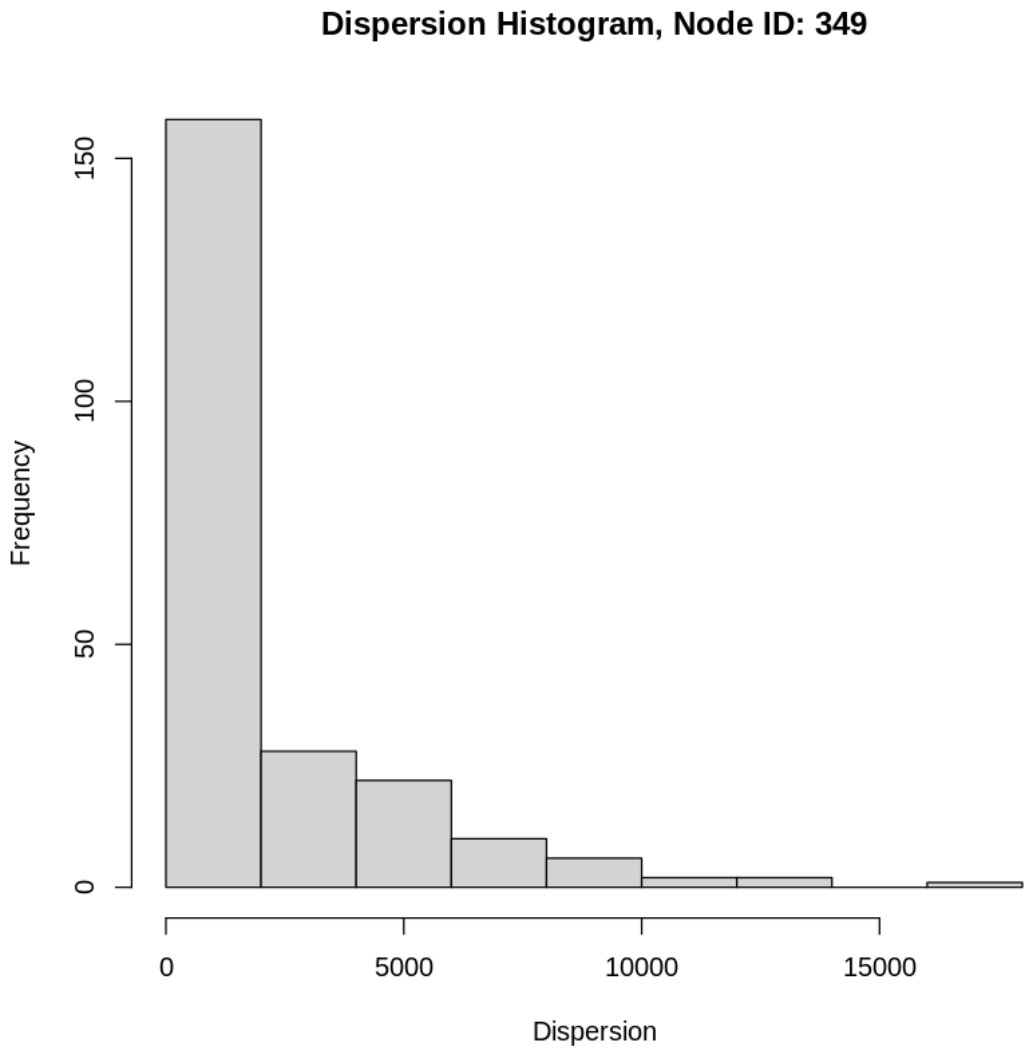

Embeddedness Histogram, Node ID: 1
Frequency
0
50
100
150
0
20
40
60
80
Embeddedness

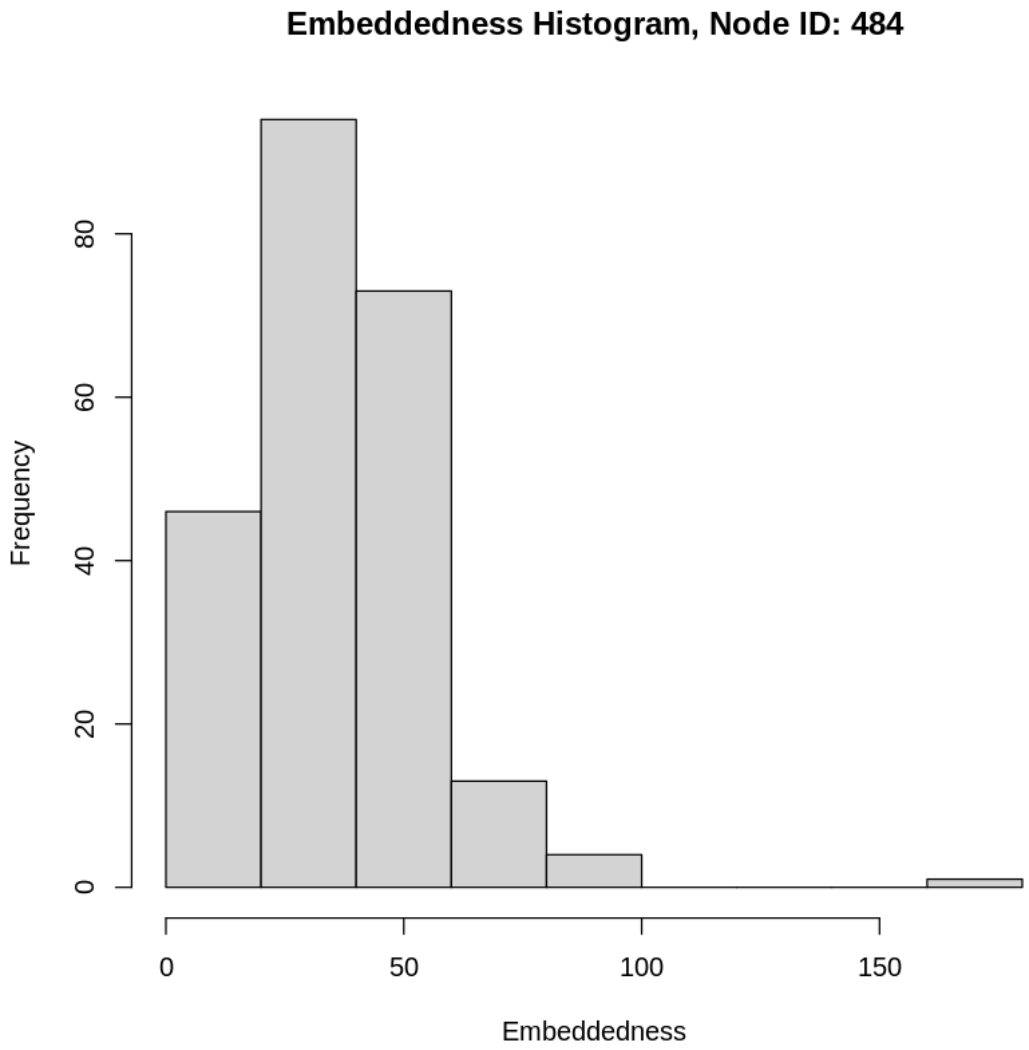

Dispersion Histogram, Node ID: 1
Frequency
0
50
100
150
200
250
300
0
2000
4000
6000
8000
10000
Dispersion

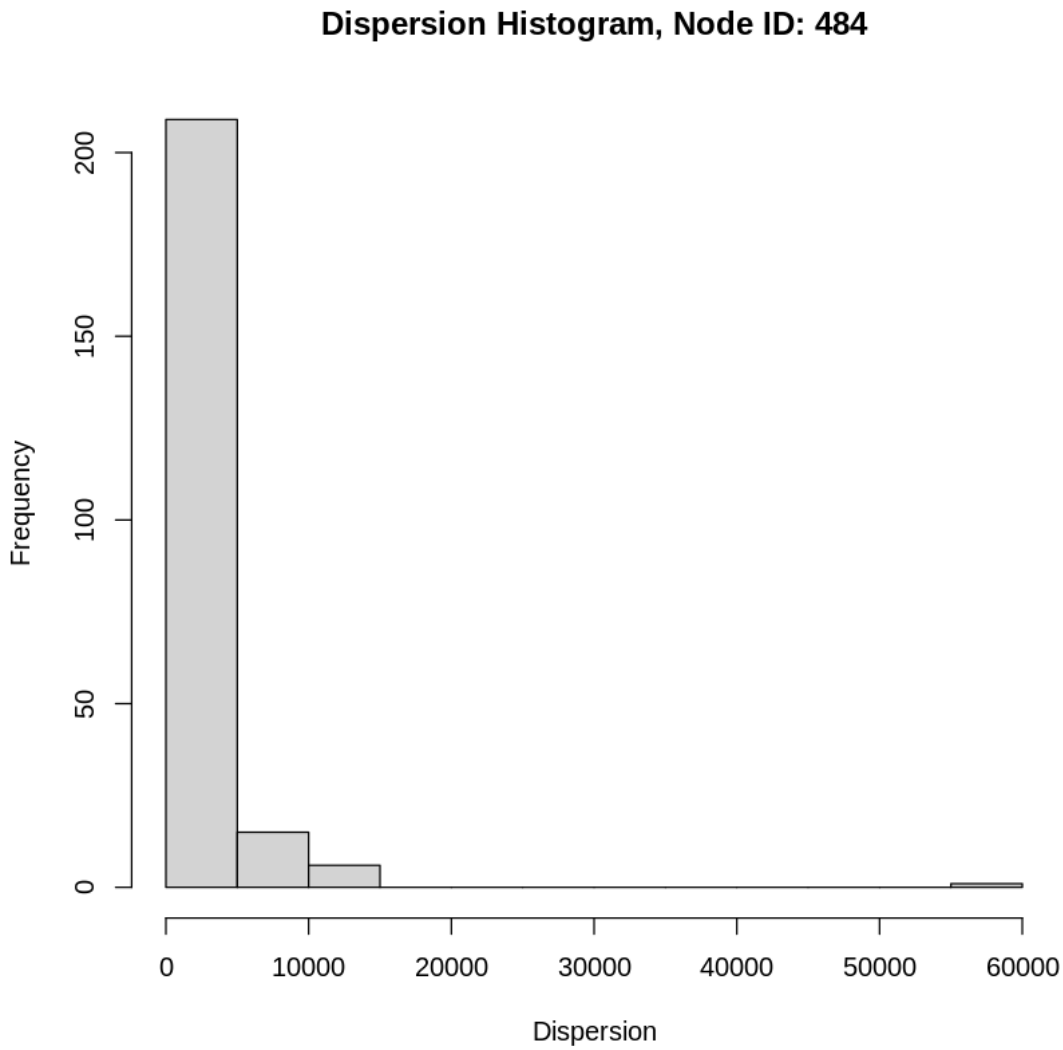

Embeddedness Histogram, Node ID: 108
Frequency
0
50
100
150
200
250
300
0
50
100
150
200
250
Embeddedness

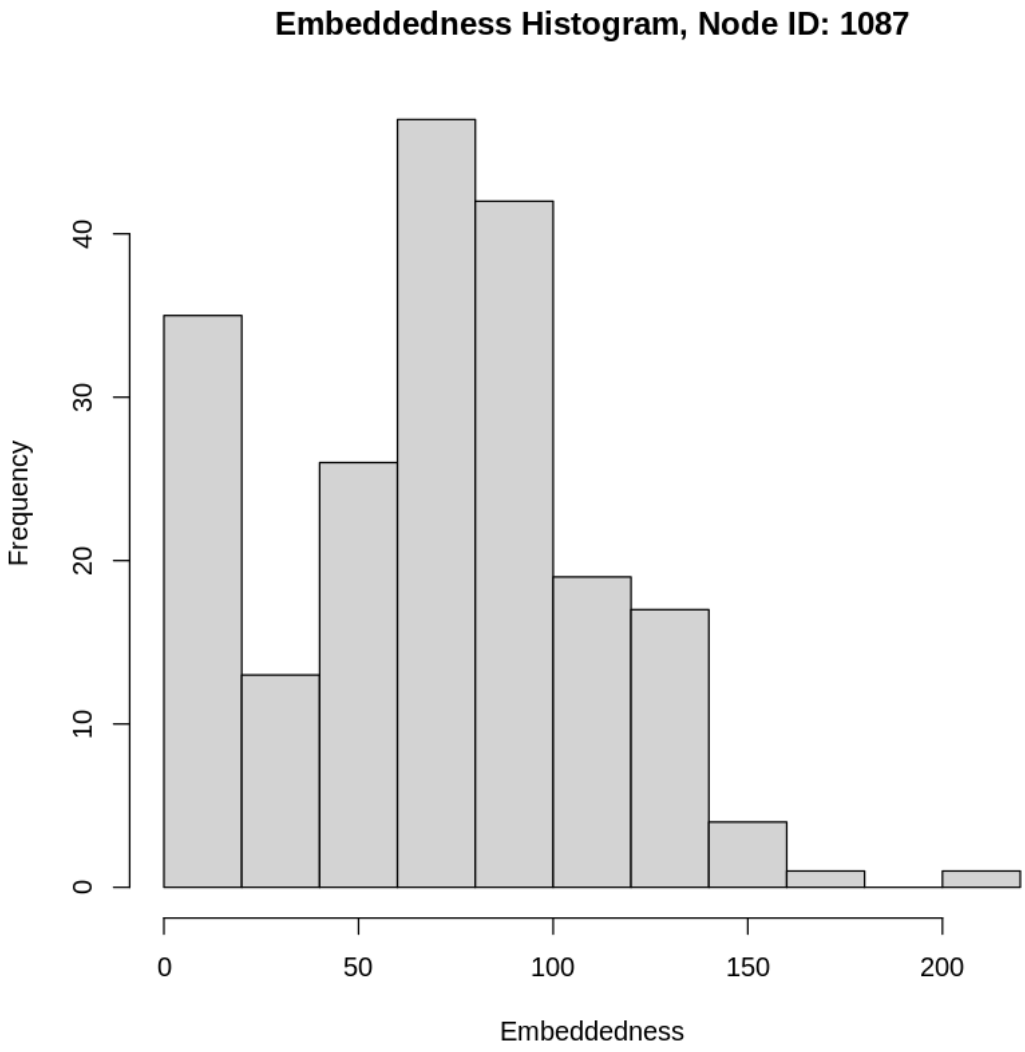

Dispersion Histogram, Node ID: 108
Frequency
0
200
400
600
800
0e+00
2e+04
4e+04
6e+04
8e+04
1e+05
Dispersion

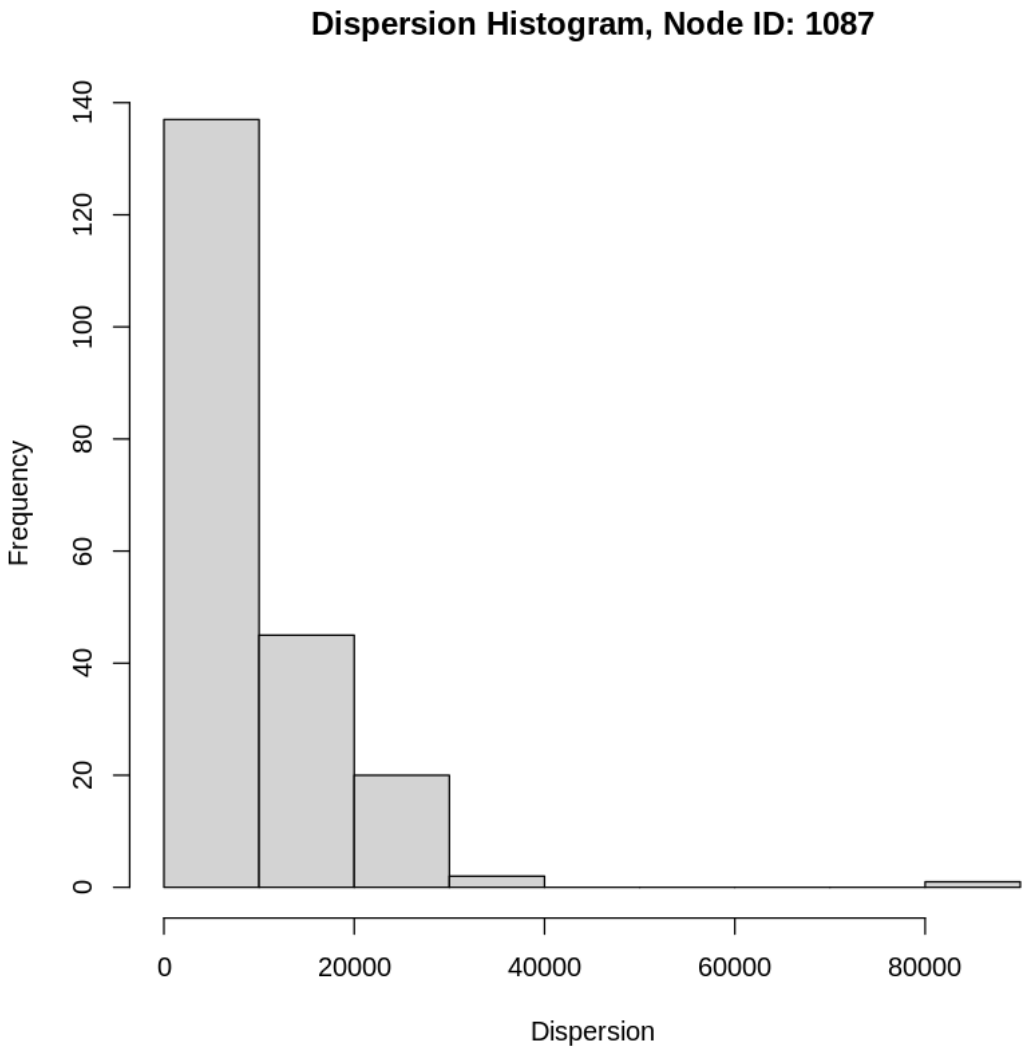

Embeddedness Histogram, Node ID: 349
Frequency
0
10
20
30
40
50
60
0
20
40
60
80
100
Embeddedness


Dispersion Histogram, Node ID: 349
Frequency
0
50
100
150
0
5000
10000
15000
Dispersion


Embeddedness Histogram, Node ID: 484
Frequency
0
20
40
60
80
0
50
100
150
Embeddedness


Dispersion Histogram, Node ID: 484
Frequency
0
50
100
150
200
0
10000
20000
30000
40000
50000
60000
Dispersion


Embeddedness Histogram, Node ID: 1087
Frequency
0
10
20
30
40
0
50
100
150
200
Embeddedness

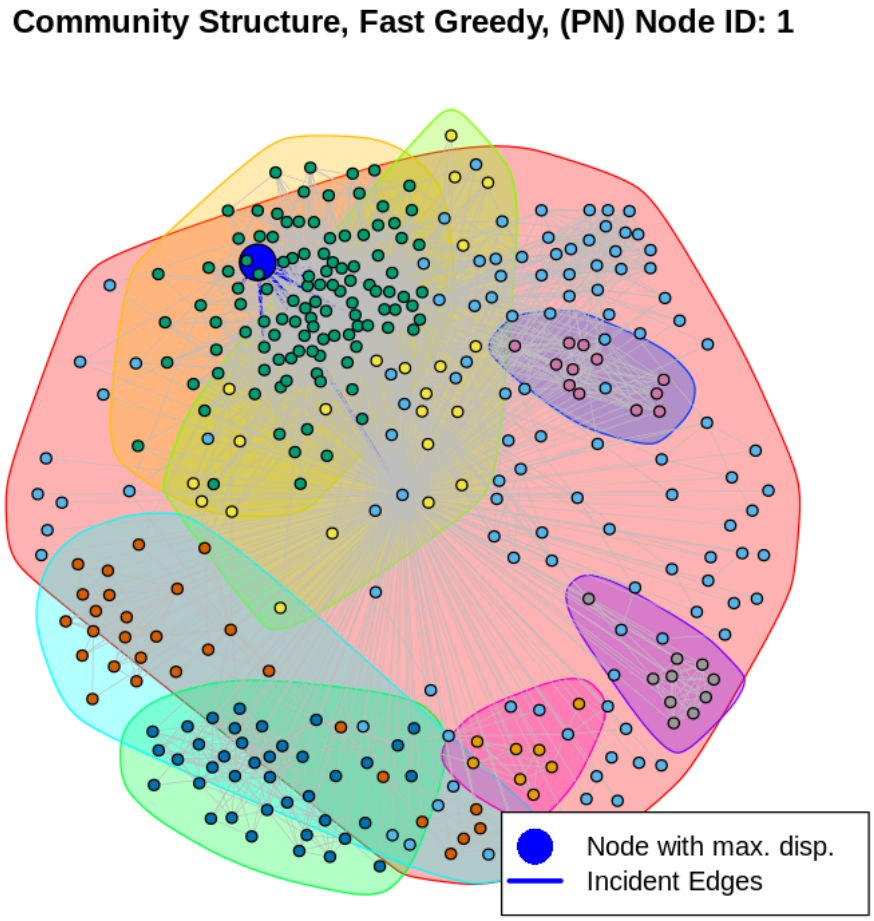


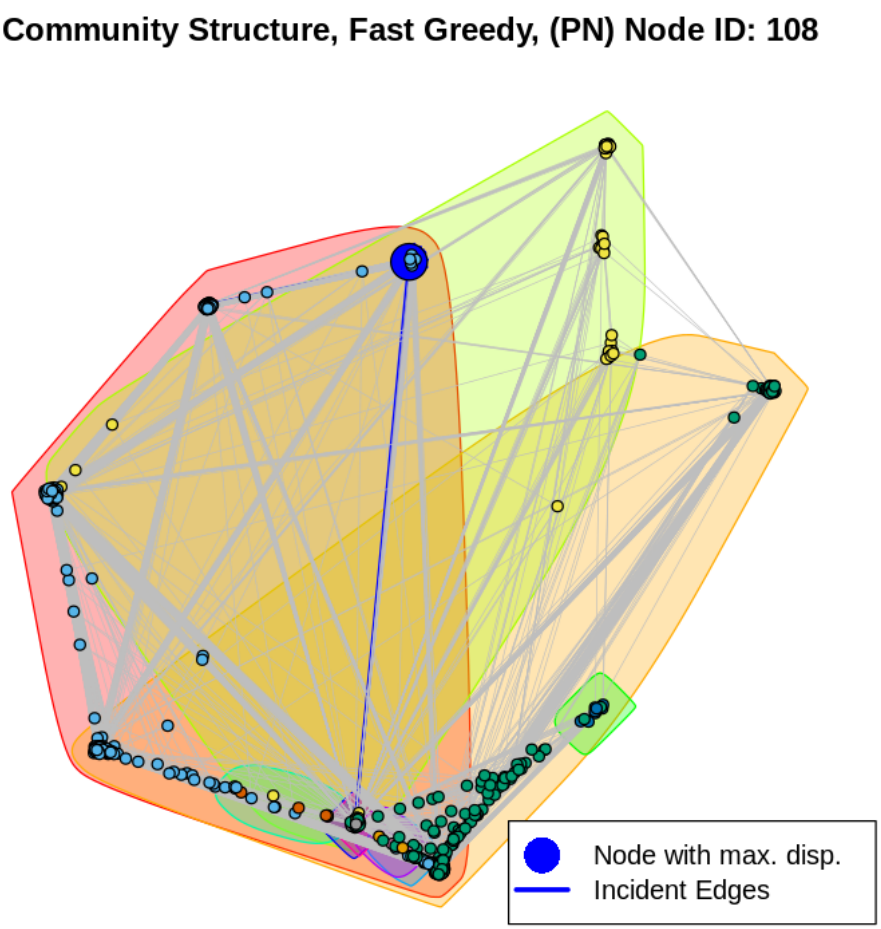
16 Q13

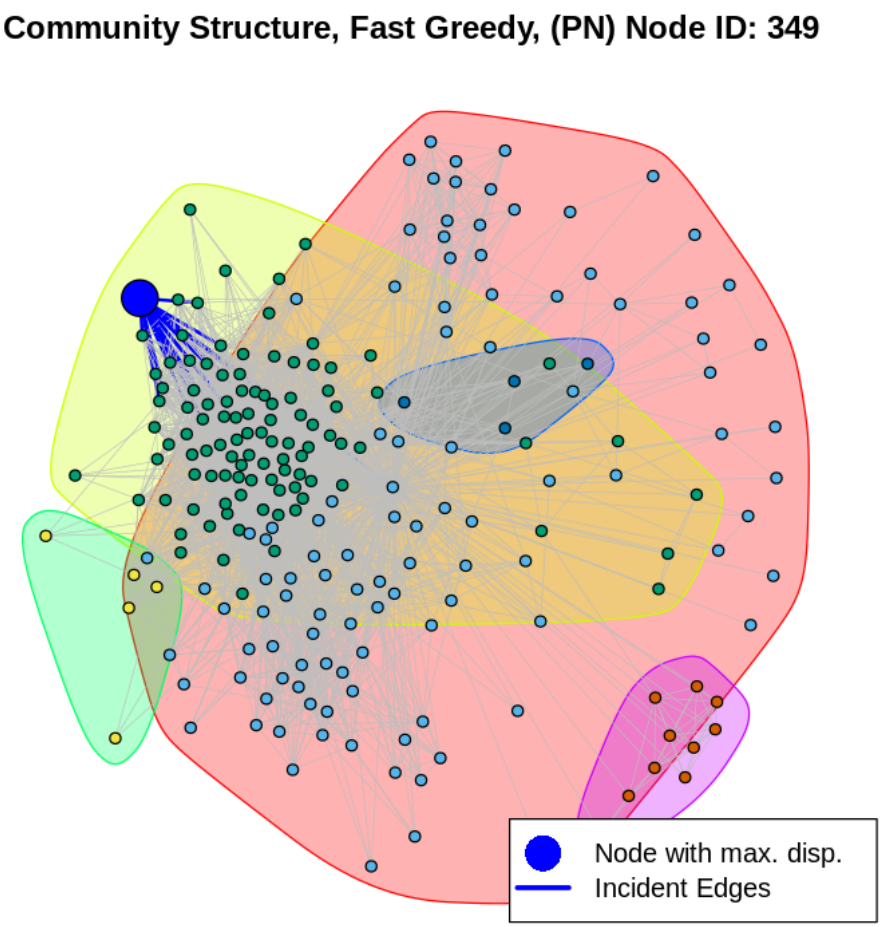
[ ]:
```
for(j in c(1:5)){

    nodelist = unlist(ego(g, order=1, nodes=node_list_str[j]))
    pn = induced.subgraph(g,nodelist)
    pn$name = sort(nodelist)
    embeddedness <- c()
    disp <- c()
    deg <- c()
    i=1
    for(v in vertex_attr(pn)$name){
```

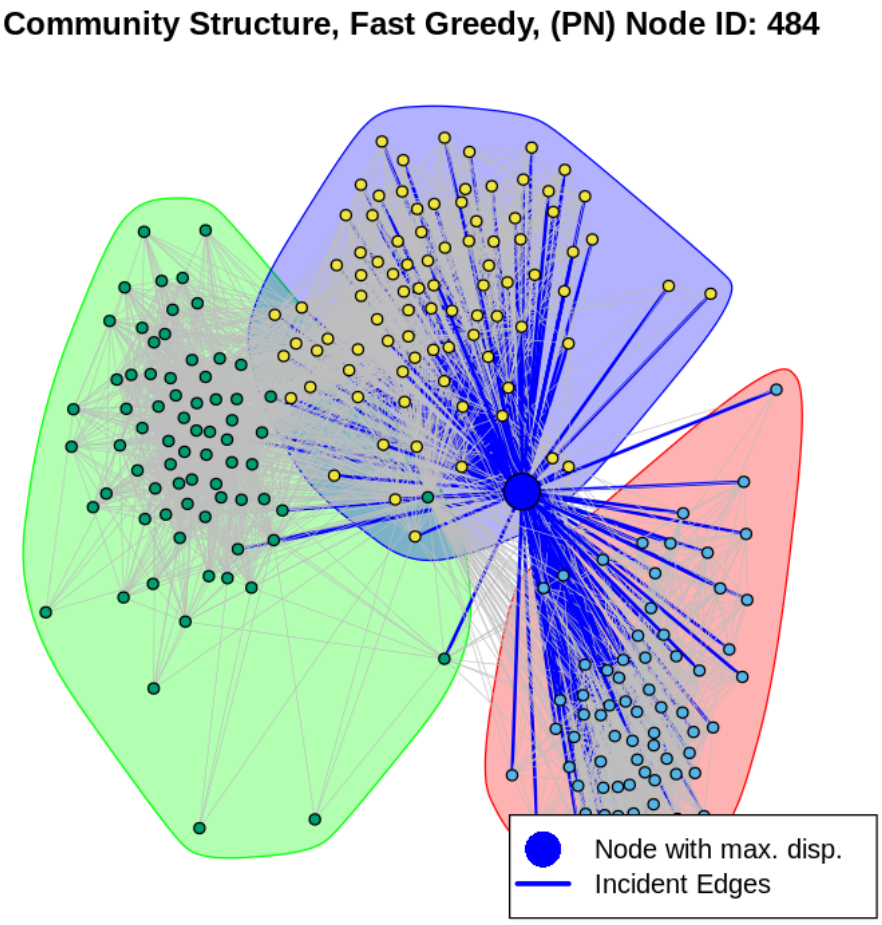
```
        if(v==node_list_str[j])
            next
        disp[i] = 0
        neh_ver = neighbors(pn, node_list_str[j])
        neh_core = neighbors(pn,v)
        inter = intersection(neh_ver, neh_core)
        embeddedness[i] = length(inter)
        deg[i] = degree(pn,v)
        eg2 = delete.vertices(pn, c(node_list_str[j], v))
        if(embeddedness[i]>1){
            ver = c()
            for(m in 1:length(inter)){
                ver =  c(ver,vertex_attr(pn)$name[inter[m]])
            }
            ver1=c()
            for(m in 1:length(ver)){
                ver1=c(ver1,which(vertex_attr(eg2)$name==ver[m]))
            }
            disp_mat = distances(eg2,v=ver1, to=ver1)
            disp_mat[disp_mat==Inf]<-diameter(eg2)+1
            disp[i] = sum(disp_mat)
        }
        i=i+1

    }

    maxdisp = pn$name[which(disp==max(disp))]
    maxnode = which(pn$name==maxdisp)

    fg <- cluster_fast_greedy(pn)
    vert_col = fg$membership+1
    vert_col[maxnode] = "blue"
    vert_size = rep(3, length(vert_col))
    vert_size[maxnode] = 10
    edge_col = rep("grey", length(E(pn)))
    edge_col[which(get.edgelist(pn, name = FALSE)[,1] == maxnode | get.
↪edgelist(pn, name = FALSE)[,2] == maxnode)] = "blue"
    edge_wid = rep(0.5,length(E(pn)))
    edge_wid[which(get.edgelist(pn, name = FALSE)[,1] == maxnode |  get.
↪edgelist(pn, name = FALSE)[,2] == maxnode)] = 2;
    plot(pn,mark.groups = fg,vertex.size=vert_size,edge.arrow.size=.5,vertex.
↪color=vert_col,edge.color = edge_col,edge.width=edge_wid,
         vertex.label="",main = sprintf("Community Structure, Fast Greedy, (PN)␣
↪Node ID: %d", strtoi(node_list_str[j])+1))
    legend('bottomright', legend = c("Node with max. disp.", "Incident Edges"),
       lty = c(0, 1), lwd = c(5,3), pch=c(16,NA),
       col = c('blue','blue'), pt.cex=3)
```

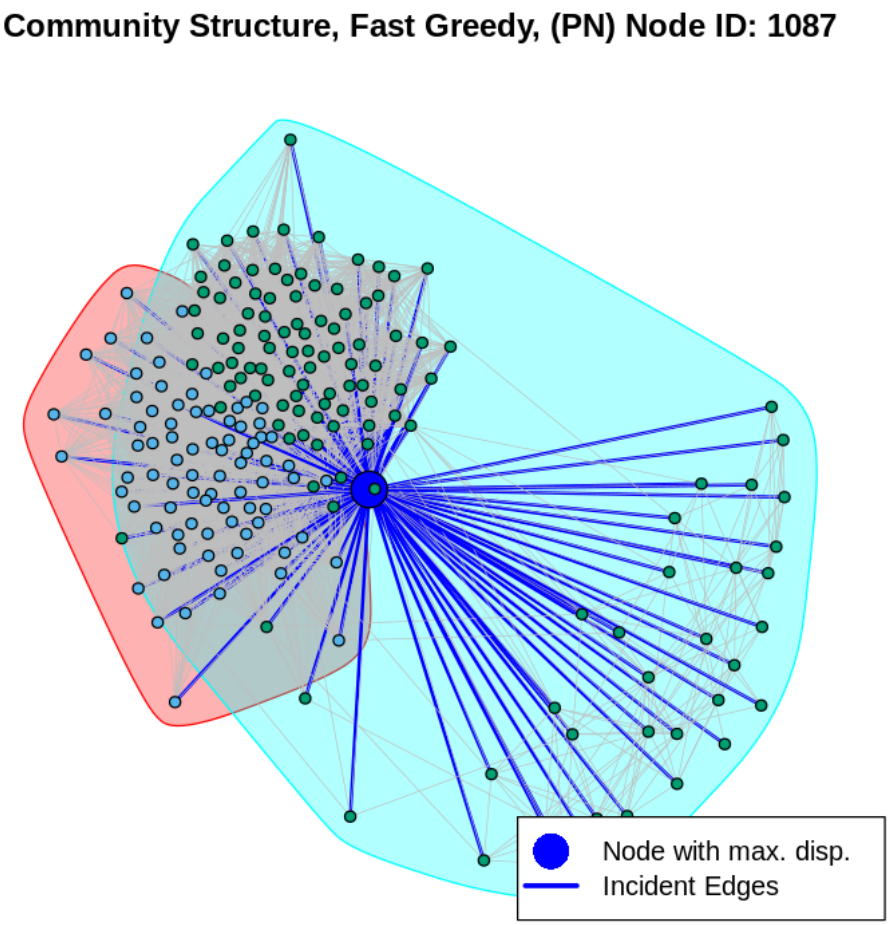
```
}
```


Community Structure, Fast Greedy, (PN) Node ID: 1
Node with max. disp.
Incident Edges


Community Structure, Fast Greedy, (PN) Node ID: 108
Node with max. disp.
Incident Edges


Community Structure, Fast Greedy, (PN) Node ID: 349
Node with max. disp.
Incident Edges


Community Structure, Fast Greedy, (PN) Node ID: 484
Node with max. disp.
Incident Edges



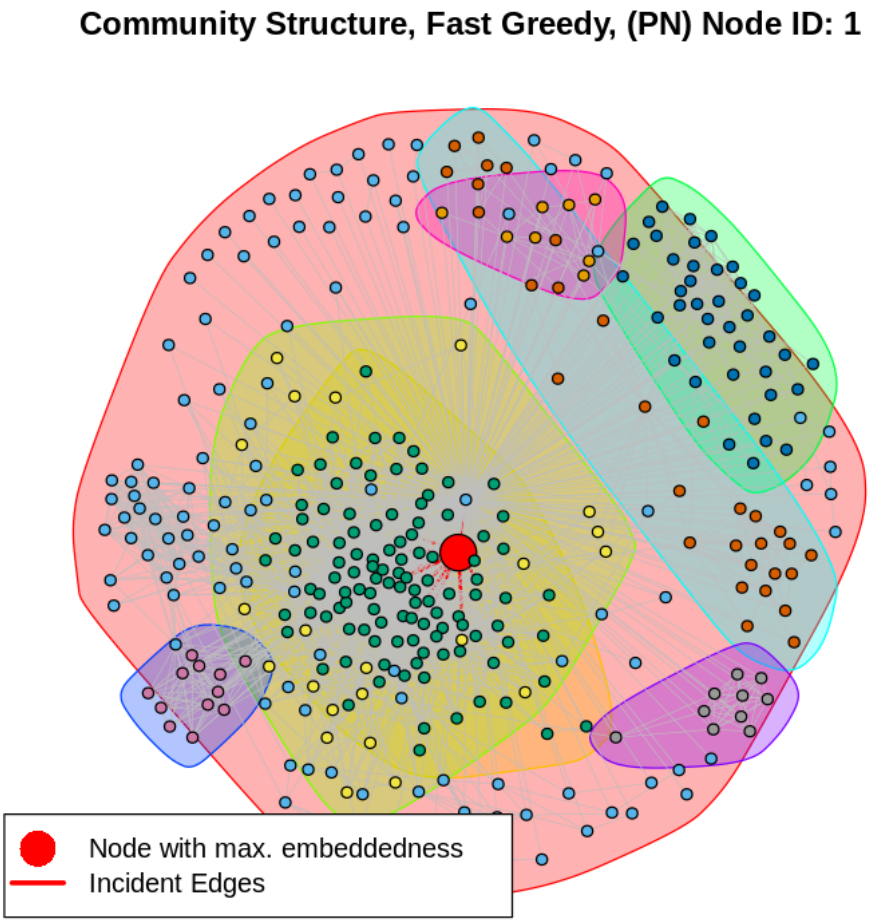
17 Q14

[ ]:
```
for(j in c(1:5)){

    nodelist = unlist(ego(g, order=1, nodes=node_list_str[j]))
    pn = induced.subgraph(g,nodelist)
    pn$name = sort(nodelist)
    embeddedness <- c()
    disp <- c()
    deg <- c()
    i=1
    for(v in vertex_attr(pn)$name){
```

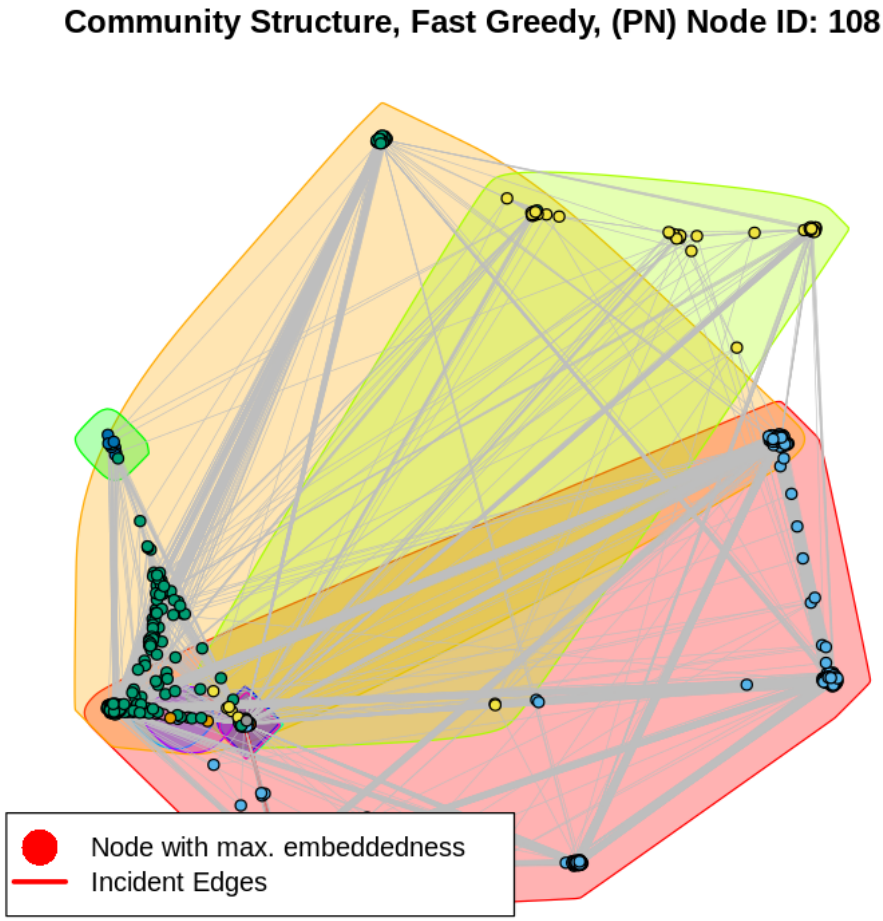
```
        if(v==node_list_str[j])
            next

        disp[i] = 0
        neh_ver = neighbors(pn, node_list_str[j])
        neh_core = neighbors(pn,v)
        inter = intersection(neh_ver, neh_core)
        embeddedness[i] = length(inter)

        deg[i] = degree(pn,v)
        eg2 = delete.vertices(pn, c(node_list_str[j], v))
        if(embeddedness[i]>1){
            ver = c()
            for(m in 1:length(inter)){
                ver =  c(ver,vertex_attr(pn)$name[inter[m]])
            }
            ver1=c()
            for(m in 1:length(ver)){
                ver1=c(ver1,which(vertex_attr(eg2)$name==ver[m]))
            }
            disp_mat = distances(eg2,v=ver1, to=ver1)
            disp_mat[disp_mat==Inf]<-diameter(eg2)+1
            disp[i] = sum(disp_mat)
        }
        i=i+1

    }

    maxemb = pn$name[which(embeddedness==max(embeddedness))]
    maxnode = which(pn$name==maxemb)
    print(sprintf("ID of node with max. emb. for core node with ID %d:␣
↪%d",strtoi(node_list_str[j])+1,maxnode))

    fg <- cluster_fast_greedy(pn)
    vert_col = fg$membership+1
    vert_col[maxnode] = "red"
    vert_size = rep(3, length(vert_col))
    vert_size[maxnode] = 10
    edge_col = rep("grey", length(E(pn)))
    edge_col[which(get.edgelist(pn, name = FALSE)[,1] == maxnode | get.
↪edgelist(pn, name = FALSE)[,2] == maxnode)] = "red"
    edge_wid = rep(0.5,length(E(pn)))
    edge_wid[which(get.edgelist(pn, name = FALSE)[,1] == maxnode |  get.
↪edgelist(pn, name = FALSE)[,2] == maxnode)] = 2;
    plot(pn,mark.groups = fg,vertex.size=vert_size,edge.arrow.size=.5,vertex.
↪color=vert_col,edge.color = edge_col,edge.width=edge_wid,
```

```
        vertex.label="",main = sprintf("Community Structure, Fast Greedy, (PN)␣
↪Node ID: %d", strtoi(node_list_str[j])+1))
   legend('bottomleft', legend = c("Node with max. embeddedness", "Incident␣
↪Edges"),
       lty = c(0, 1), lwd = c(5,3), pch=c(16,NA),
       col = c('red','red'), pt.cex=3)

}
```

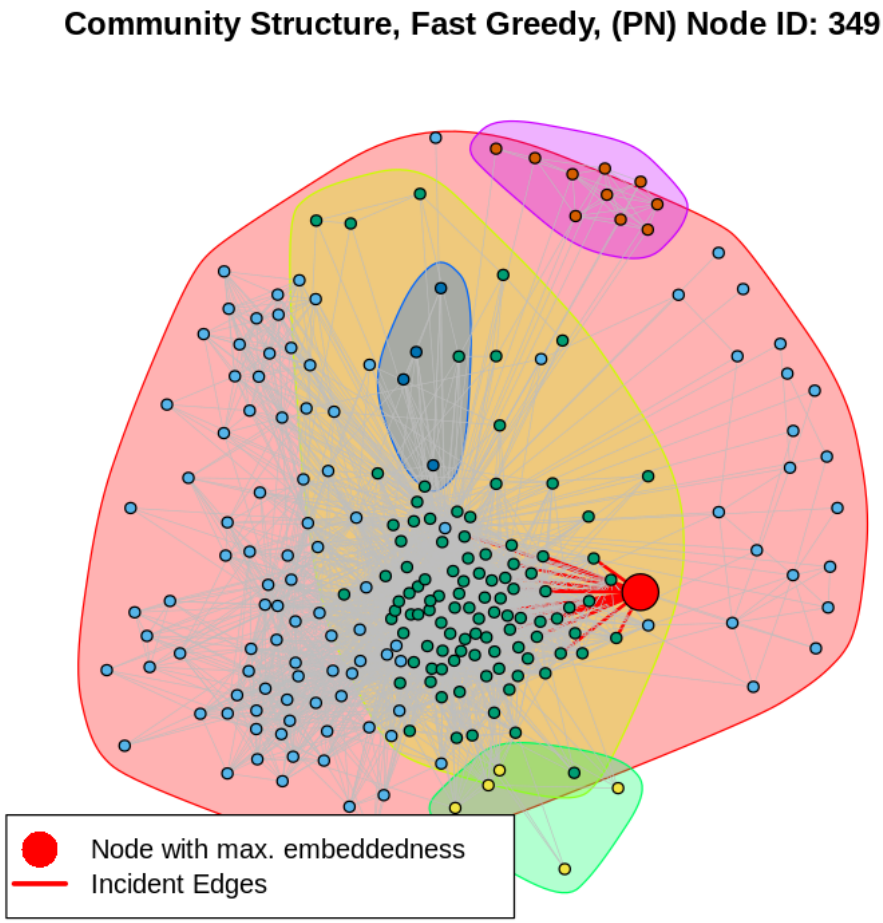
```
[1] "ID of node with max. emb. for core node with ID 1: 49"
[1] "ID of node with max. emb. for core node with ID 108: 977"
```



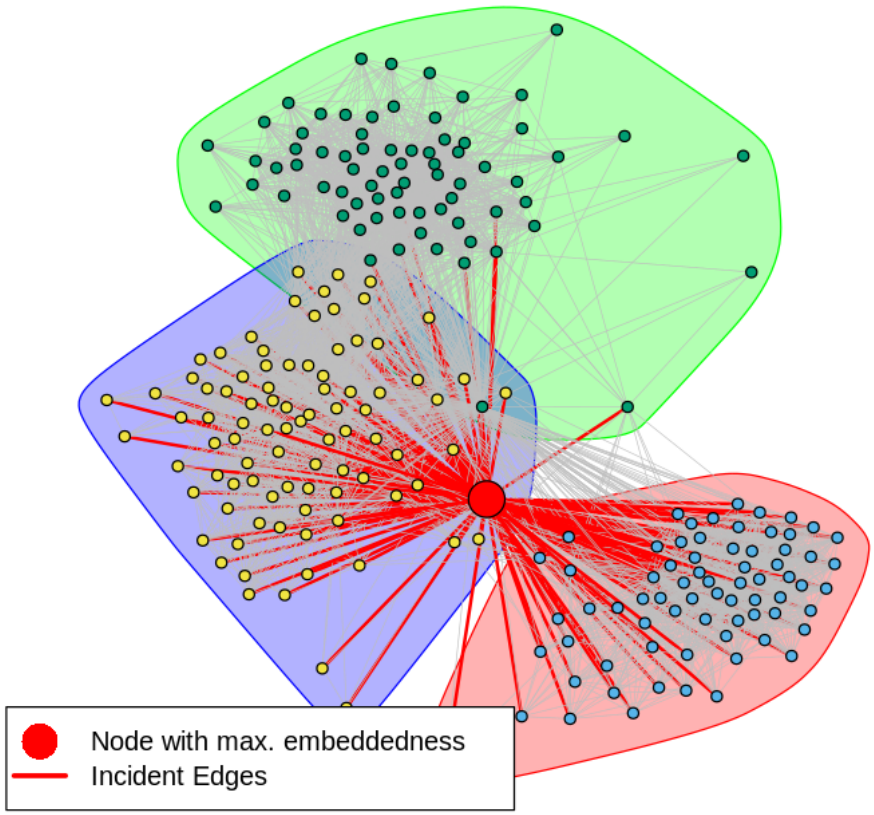

```
[1] "ID of node with max. emb. for core node with ID 349: 31"
```

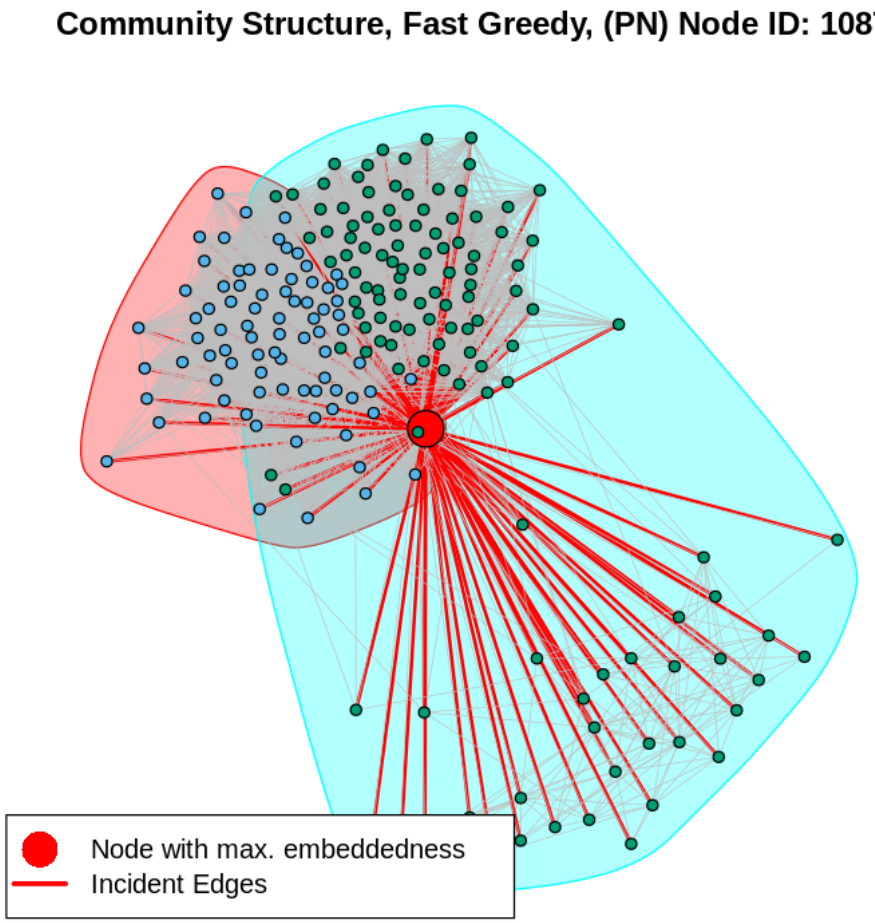



```
[1] "ID of node with max. emb. for core node with ID 484: 1"
```




```
[1] "ID of node with max. emb. for core node with ID 1087: 1"
```


Community Structure, Fast Greedy, (PN) Node ID: 484
Node with max. embeddedness
Incident Edges



18 Q15

Embeddedness:

Embeddedness is defined as the mutual connections in the social network of two users. The higher the number of nodes and connectivity among them in the social network, the greater is the embeddedness. The expected value of embeddedness is dependent on the number of edges in the personalized network. In addition, large communities are likely to have a node with large value of embeddedness by definition of embeddedness, which is proportional to the degree of the node. However, this is not a good measure of the strength of ties, as users with strong relationships are likely to be associated with users from various different communities.

Dispersion:

Dispersion is defined as the sum of distances between every pair of the mutual vertices the node shares with the core node, calculated in a modified subgraph graph with the target node and core node removed. Dispersion measures the extent to which two people's mutual friends are not themselves well-connected, in other words, the mutual nodes of two strongly connected nodes are not well connected and must display a dispersed structure. Therefore, the density of incident edges from a node with maximum dispersion will likely be lower than a node with maximum embeddedness, since the network must display a dispersed structure with connectedness among users from various different communities. Dispersion is going to be higher for those nodes whose connections are farther apart than the nodes with dense connectedness. As a result, dispersion is a better measure of strength of ties or relationship over embeddedness.

Dispersion/Embeddedness:

The maximum value of dispersion/embeddedness happens when dispersion of a node is very high while the embeddedness is very low. Such a node is likely to have mutual friends from different communities with stronger ties. The ratio is higher for smaller networks with dispersed structures, with friends that are themselves not strongly connected.

19 Q16

[ ]:
```
g_pn = make_ego_graph(g, nodes=c('414'))
Nr = which(degree(g_pn[[1]]) == 24)
print(length(Nr))
```

```
[1] 11
```

the length is 11

20 Q17

[ ]:
```
common.neighbors <- function(graph, i, j) {
    s_i <- neighbors(graph, i, mode = c("all"))
    s_j <- neighbors(graph, j, mode = c("all"))
    inter <- intersection(s_i, s_j)
    length(inter)
}

jaccard <- function(graph, i, j) {
    s_i <- neighbors(graph, i, mode = c("all"))
    s_j <- neighbors(graph, j, mode = c("all"))
    inter <- intersection(s_i, s_j)
    length(inter)/(length(s_i)+length(s_j))
}

adamic.adar <- function(graph, i, j) {
    s_i <- neighbors(graph, i, mode = c("all"))
    s_j <- neighbors(graph, j, mode = c("all"))
    inter <- intersection(s_i, s_j)
```

```
    sum <- 0
    for(k in inter) {
        s_k <- neighbors(graph, k, mode = c("all"))
        sum <- sum + 1/(log(length(s_k)))
    }
    sum
}

not.friends <- function(graph, i) {
    ver <- V(graph)
    s_i <- neighbors(graph, i, mode = c("all"))
    not_friend <- ver[!ver %in% s_i]
    not_friend
}

friend_recs <- function(graph, i, neighborhood_measure, num_of_recs) {
    measure <- c()
    not_friends <- not.friends(graph, i)
    for(j in not_friends){
        if(neighborhood_measure == "common neighbors"){
            measure <- c(measure, common.neighbors(graph, i, j))
        } else if(neighborhood_measure == "jaccard") {
            measure <- c(measure, jaccard(graph, i, j))
        } else if(neighborhood_measure == "adamic adar") {
            measure <- c(measure, adamic.adar(graph, i, j))
        }
    }
    measure_sorted <- sort(measure, decreasing = TRUE, index.return=TRUE)
    recd_friends <- not_friends[measure_sorted$ix]
    recd_friends[c(1:num_of_recs)]
}

select.edges <- function(graph, i, p) {
    incident_edges <-incident(graph, v=i, mode = c("all"))
    sample(incident_edges, p*degree(graph,v=i))
}

delete.edges <- function(graph, edges) {
    delete_edges(graph, edges)
}

add.edges <- function(graph, i, edges) {
    for(j in 1:length(edges)) {
        graph <- graph + edge(i, edges[j])
    }
    graph
}
```

```
accuracy <- function(graph, graph3, i) {
    s_i <- neighbors(graph, i, mode = c("all"))
    s_j <- neighbors(graph3, i, mode = c("all"))
    inter <- intersection(s_i, s_j)
    length(inter)/length(s_i)
}

rec.accuracy <- function(graph, i, p, user_list, neighborhood_measure) {
    edges_to_delete <- select.edges(graph, i, p)
    graph2 <- delete.edges(graph, edges_to_delete)
    recommended <- friend_recs(graph2, i, neighborhood_measure,␣
 ↪length(edges_to_delete))
    graph3 <- add.edges(graph2, i, recommended)
    accuracy(graph, graph3, i)
}
```

[ ]:
```
node_ind <- c()
eg_tmp <- g_pn[[1]]
ver_tmp <- V(eg_tmp)
for(i in 1:length(degree(eg_tmp))) {
    if(degree(eg_tmp, v=i) == 24) {
        node_ind <- c(node_ind,i)
    }
}
user_list <- ver_tmp[node_ind]
graph <- g_pn[[1]]
p <- 0.75
neighborhood_measure <- c("common neighbors", "jaccard", "adamic adar")

for(measure in neighborhood_measure) {
    avg <- c()
    for(i in user_list) {
        sum <- 0
        for(m in 1:10) {
            recs <- rec.accuracy(graph, i, p, user_list, measure)
            sum <- sum + recs
        }
        avg <- c(avg, sum/10)
    }
    print(measure)
    print(mean(avg))
}
```

```
[1] "common neighbors"
[1] 0.8787879
[1] "jaccard"
```

```
[1] 0.8530303
[1] "adamic adar"
[1] 0.8825758
```

By the outcome above, we can see that common neighbors is the best.

21 Part2 - Google Network

[ ]:
```
install.packages("igraph")
install.packages("entropy")
install.packages("infotheo")
install.packages("googledrive")
install.packages("httpuv")
library("googledrive")
library('igraph')
library('entropy')
library('infotheo')
library("httpuv")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

also installing the dependencies ‘Rcpp’, ‘promises’, ‘later’



Attaching package: ‘igraph’


The following objects are masked from ‘package:stats’:

    decompose, spectrum


The following object is masked from ‘package:base’:
```

```
    union


Attaching package: ‘infotheo’


The following objects are masked from ‘package:entropy’:

    discretize, entropy
```

[ ]:
```r
if (file.exists("/usr/local/lib/python3.7/dist-packages/google/colab/_ipython.
  ↪py")) {
  install.packages("R.utils")
  library("R.utils")
  library("httr")
  my_check <- function() {return(TRUE)}
  reassignInPackage("is_interactive", pkgName = "httr", my_check)
  options(rlang_interactive=TRUE)
}
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

also installing the dependencies ‘R.oo’, ‘R.methodsS3’


Loading required package: R.oo

Loading required package: R.methodsS3

R.methodsS3 v1.8.1 (2020-08-26 16:20:06 UTC) successfully loaded. See
?R.methodsS3 for help.

R.oo v1.24.0 (2020-08-26 16:11:58 UTC) successfully loaded. See ?R.oo for help.


Attaching package: ‘R.oo’


The following object is masked from ‘package:R.methodsS3’:

    throw
```

```
The following object is masked from ‘package:igraph’:

    hierarchy


The following objects are masked from ‘package:methods’:

    getClasses, getMethods


The following objects are masked from ‘package:base’:

    attach, detach, load, save


R.utils v2.11.0 (2021-09-26 08:30:02 UTC) successfully loaded. See ?R.utils for
help.


Attaching package: ‘R.utils’


The following object is masked from ‘package:utils’:

    timestamp


The following objects are masked from ‘package:base’:

    cat, commandArgs, getOption, inherits, isOpen, nullfile, parse,
    warnings
```

[ ]:
```
# # authorize google drive
# drive_auth(
#   email = gargle::gargle_oauth_email(),
#   path = NULL,
#   scopes = "https://www.googleapis.com/auth/drive",
#   cache = gargle::gargle_oauth_cache(),
#   use_oob = gargle::gargle_oob_default(),
#   token = NULL
# )
```

```
Is it OK to cache OAuth access credentials in the folder
~/.cache/gargle
between R sessions?
```

```
Error in `drive_auth()`:
! Can't get Google credentials
  Are you running googledrive in a non-interactive session? Consider:
• `drive_deauth()` to prevent the attempt to get credentials
• Call `drive_auth()` directly with all necessary specifics
  See gargle's "Non-interactive auth" vignette for more details:
  <https://gargle.r-lib.org/articles/non-interactive-auth.html>
Traceback:

1. drive_auth(email = gargle::gargle_oauth_email(), path = NULL,
 .     scopes = "https://www.googleapis.com/auth/drive", cache = gargle::
 ↪gargle_oauth_cache(),
 .     use_oob = gargle::gargle_oob_default(), token = NULL)
2. drive_abort(c("Can't get Google credentials", i = "Are you running␣
 ↪googledrive in a non-interactive session? \\\n             Consider:",
 .     `*` = "{.fun drive_deauth} to prevent the attempt to get credentials",
 .     `*` = "Call {.fun drive_auth} directly with all necessary specifics",
 .     i = "See gargle's \"Non-interactive auth\" vignette for more details:",
 .     i = "{.url https://gargle.r-lib.org/articles/non-interactive-auth.html}" )
3. cli::cli_abort(message = message, …, .envir = .envir)
4. rlang::abort(message, …, call = call, use_cli_format = TRUE)
5. signal_abort(cnd, .file)
```

[ ]:
```
drive_auth(use_oob = TRUE, cache = TRUE)

zip_file <- drive_get("gplus.tar.gz")

drive_download(zip_file,overwrite = 1)

untar('gplus.tar.gz')
```

```
Please point your browser to the following url:

https://accounts.google.com/o/oauth2/auth?client_id=603366585132-dpeg5tt0et3go5o
f2374d83ifevk5086.apps.googleusercontent.com&scope=https%3A%2F%2Fwww.googleapis.
com%2Fauth%2Fdrive%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.email&re
direct_uri=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&response_type=code


Enter authorization code:
4/1AX4XfWiQTL32pbSjzLlRKC1xsQRqPXmxduAJQjGPGO7dHbDCcu0OGoJErco



 The input `path` resolved to exactly 1 file.

File downloaded:
```

- gplus.tar.gz <id: 1hwohgpWZlGRPRbKI-QanWC3FGri2fp2j>

Saved locally as:

- gplus.tar.gz

[ ]:
```
file_path ="gplus/"
edge_files = list.files(path=file_path,pattern="edges")
circles_files = list.files(path=file_path,pattern="circles")
fts_files = list.files(path=file_path,pattern="feat")
initial_graph = list()
final_graph = list()
graph_circles = list()
ego_nodes = list()
```

22 Q18

[ ]:
```
cnt = 0
node_names = c()
for(i in 1:length(edge_files)){
    # get node id
    node = strsplit(edge_files[i],".edges")[[1]]
    node_names <-c(node_names,node)
    #print(node)
    ego_nodes[i] = node
    fc = file(paste(file_path,node,".circles",sep=""),open="r")
    if(length(fc)>0){
        file_lines <- readLines(fc)
        if(length(file_lines)>0){
            circles =list()
            for(j in 1:length(file_lines)){
                circle_users = strsplit(file_lines[j],"\t")
                circles[[j]] <- circle_users[[1]][-1]
              }
            # find users who have more than 2 circles
            if(length(circles)>2){
                cnt = cnt + 1
                initial_graph[[i]] <- read.
↪graph(paste(file_path,edge_files[i],sep=""),format="ncol",directed=TRUE)
                graph_circles[[i]] <- circles
                graph_nodes <- V(initial_graph[[i]])
                print(length(graph_nodes))
                print(node)
```

```
                # add the core node to his neighbor list and construct the graph
                final_graph[[i]] <- add.vertices(initial_graph[[i]],1,name=node)
                core_index = which(V(final_graph[[i]])$name==node)
                core_node_edges = list()
                ### add edges connecting to this core node
                for(k in 1:length(graph_nodes)){
                    core_node_edges = c(core_node_edges, c(core_index, k))
                }
                final_graph[[i]] <- add.edges(final_graph[[i]],core_node_edges)
            }
        }
    }
    close(fc)
}
```

```
[1] 1187
[1] "100535338638690515335"
[1] 559
[1] "100962871525684315897"
[1] 2614
[1] "101130571432010257170"
[1] 1052
[1] "101185748996927059931"
[1] 2455
[1] "101263615503715477581"
[1] 3814
[1] "101373961279443806744"
[1] 488
[1] "101541879642294398860"
[1] 4712
[1] "101626577406833098387"
[1] 2224
[1] "102170431816592344972"
[1] 102
[1] "102615863344410467759"
[1] 1371
[1] "102778563580121606331"
[1] 2332
[1] "103236949470535942612"
[1] 2009
[1] "103892332449873403244"
[1] 452
[1] "104105354262797387583"
[1] 3091
[1] "104607825525972194062"
[1] 475
[1] "104672614700283598130"
```

```
[1] 1730
[1] "104987932455782713675"
[1] 4289
[1] "106186407539128840569"
[1] 4842
[1] "106228758905254036967"
[1] 4903
[1] "106382433884876652170"
[1] 1083
[1] "106837574755355833243"
[1] 4369
[1] "107040353898400532534"
[1] 2156
[1] "107203023379915799071"
[1] 2512
[1] "107223200089245371832"
[1] 562
[1] "107459220492917008623"
[1] 4362
[1] "107489144252174167638"
[1] 1743
[1] "108883879052307976051"
[1] 773
[1] "109327480479767108490"
[1] 4720
[1] "109596373340495798827"
[1] 1889
[1] "110538600381916983600"
[1] 338
[1] "110614416163543421878"
[1] 1556
[1] "110701307803962595019"
[1] 97
[1] "110809308822849680310"
[1] 1548
[1] "110971010308065250763"
[1] 4160
[1] "111048918866742956374"
[1] 4938
[1] "111091089527727420853"
[1] 498
[1] "112317819390625199896"
[1] 851
[1] "112724573277710080670"
[1] 1662
[1] "113112256846010263985"
[1] 1604
[1] "113356364521839061717"
```

```
[1] 769
[1] "113881433443048137993"
[1] 4229
[1] "114147483140782280818"
[1] 1374
[1] "115121555137256496805"
[1] 511
[1] "115360471097759949621"
[1] 3490
[1] "115455024457484679647"
[1] 923
[1] "115625564993990145546"
[1] 1175
[1] "116247667398036716276"
[1] 1243
[1] "116315897040732668413"
[1] 4872
[1] "116807883656585676940"
[1] 4604
[1] "116825083494890429556"
[1] 4831
[1] "116931379084245069738"
[1] 252
[1] "117412175333096244275"
[1] 307
[1] "117503822947457399073"
[1] 1473
[1] "117668392750579292609"
[1] 4805
[1] "117734260411963901771"
[1] 2426
[1] "118107045405823607895"
[1] 694
[1] "118379821279745746467"
```

[ ]:
```
#Personal Networks
cnt
```

57

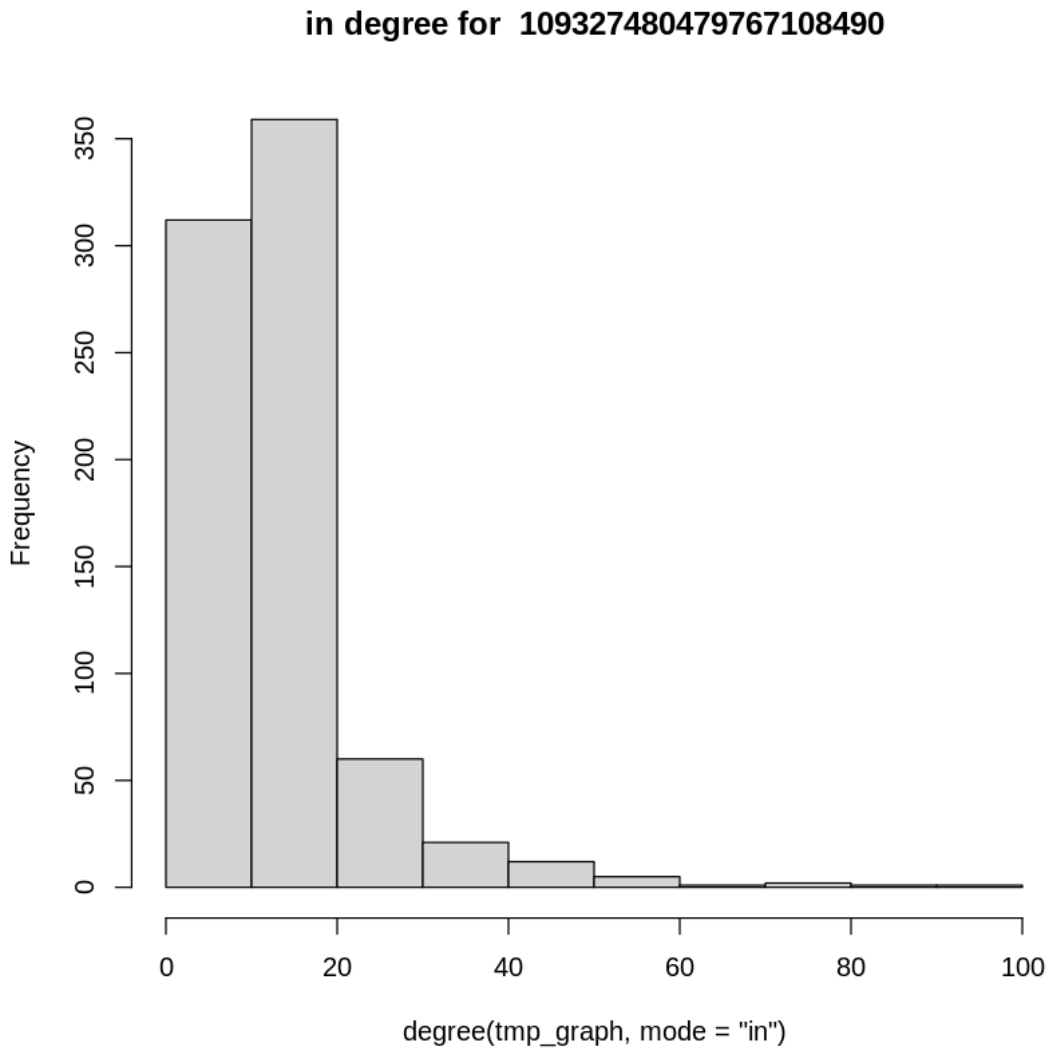
[ ]:
```
#Nodes
length(edge_files)
```

132

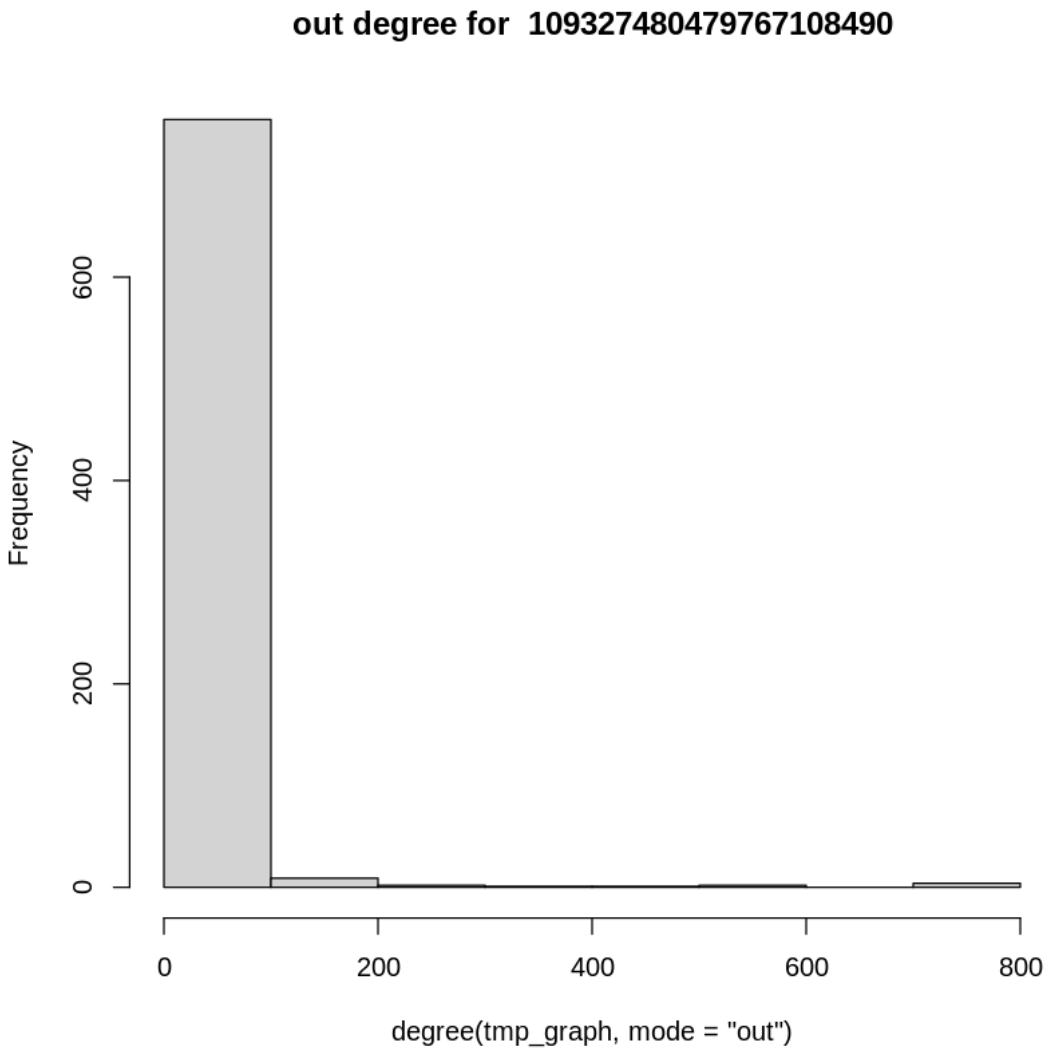
23 Q19

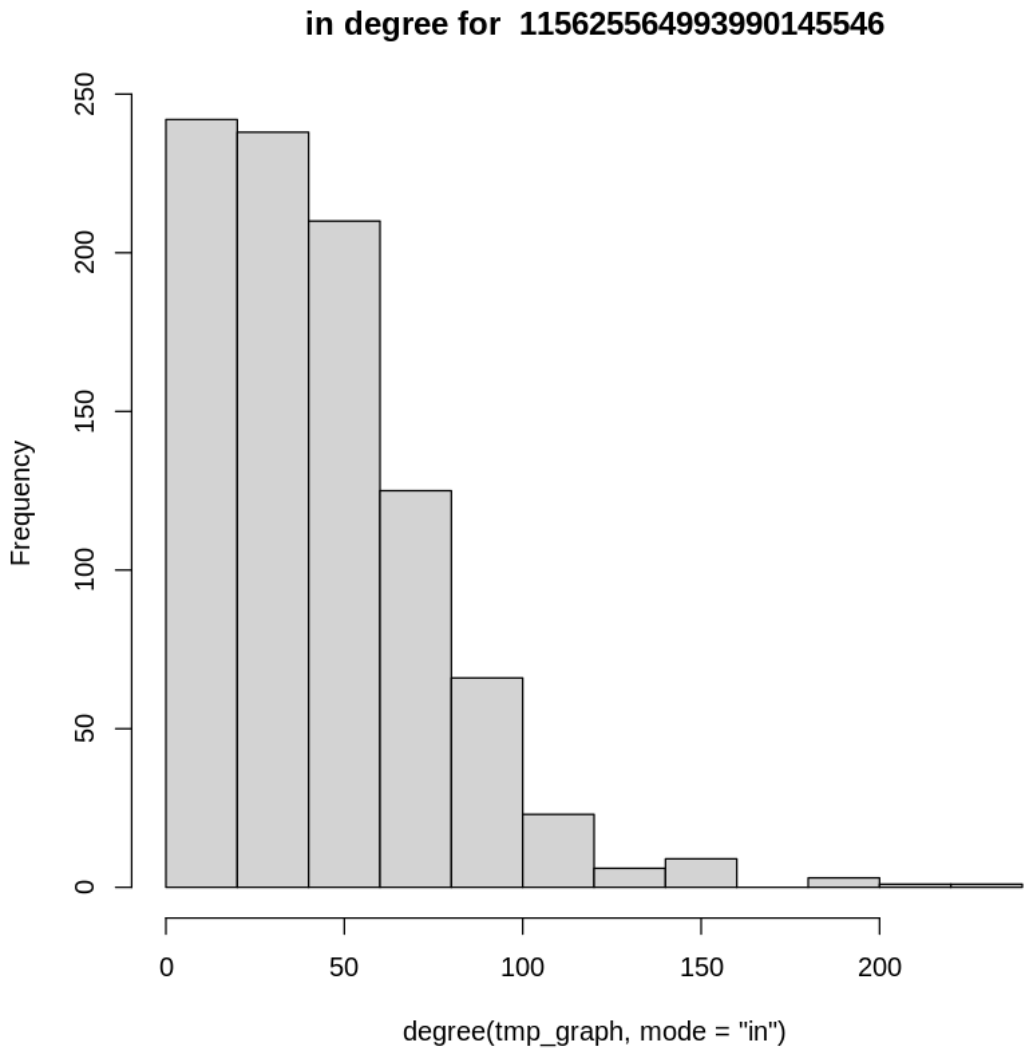
[ ]:
```
interest_node = c('109327480479767108490',␣
 ↪'115625564993990145546','101373961279443806744')
graph_inds = c()
for (i in 1: length(interest_node)){
    graph_ind <- which(node_names==interest_node[i])
    graph_inds <- c(graph_inds, graph_ind)
    print(graph_ind)
    tmp_graph = final_graph[[graph_ind]]
    hist(degree(tmp_graph, mode="in"),main = paste("in degree for ",␣
 ↪interest_node[i]))
    hist(degree(tmp_graph, mode="out"),main = paste("out degree for ",␣
 ↪interest_node[i]))
}
```

```
[1] 69
```

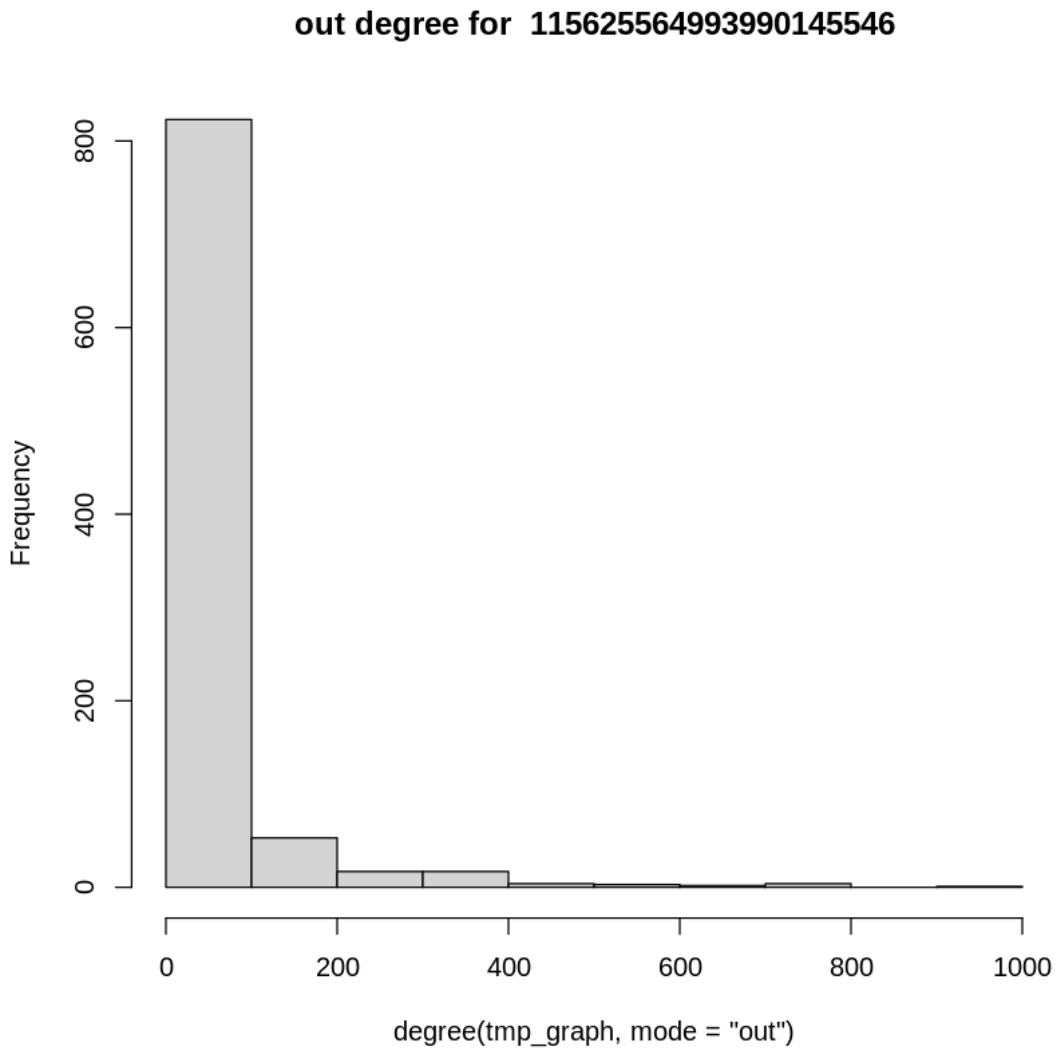

in degree for 10932748047976710849
Frequency
0
50
100
150
200
250
300
350
0
20
40
60
80
100
degree(tmp_graph, mode = "in")

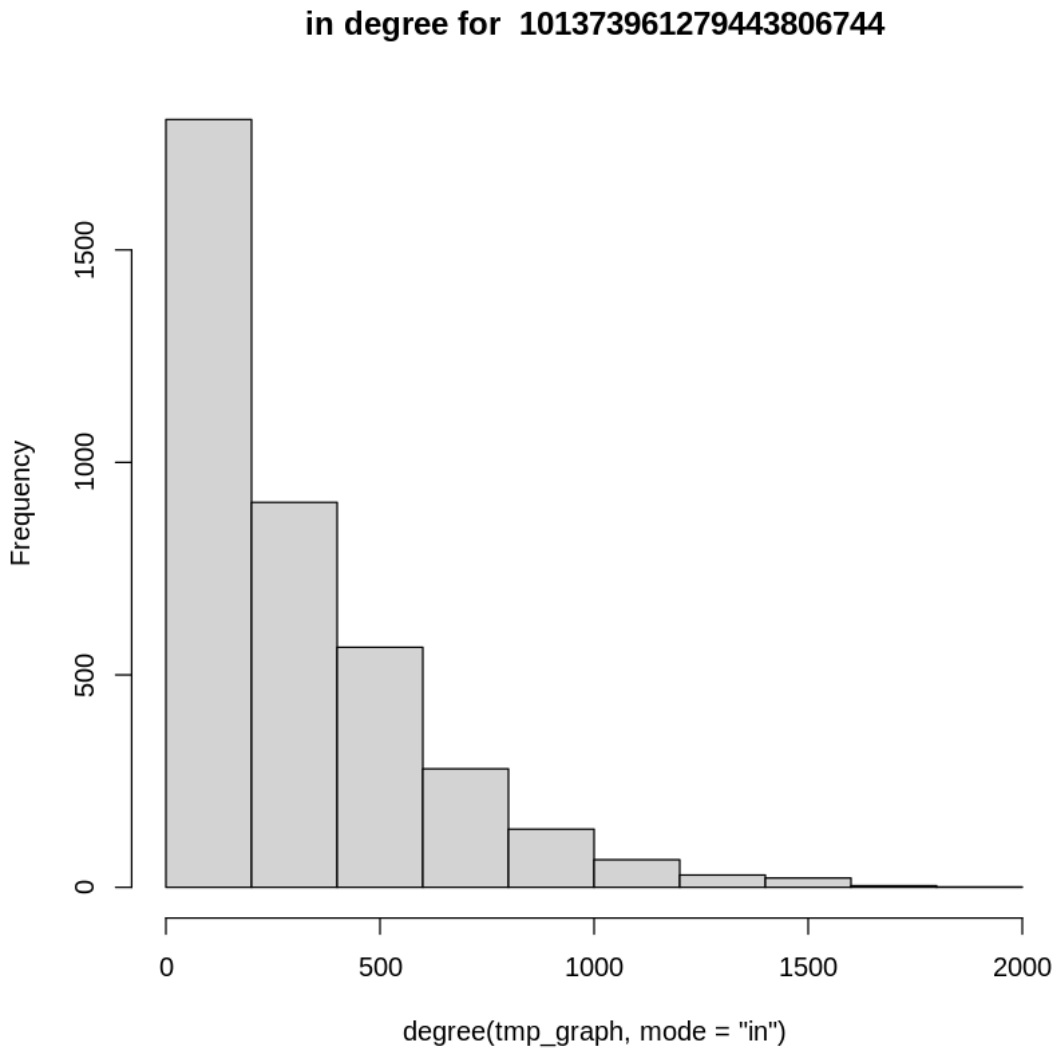
```
[1] 115
```

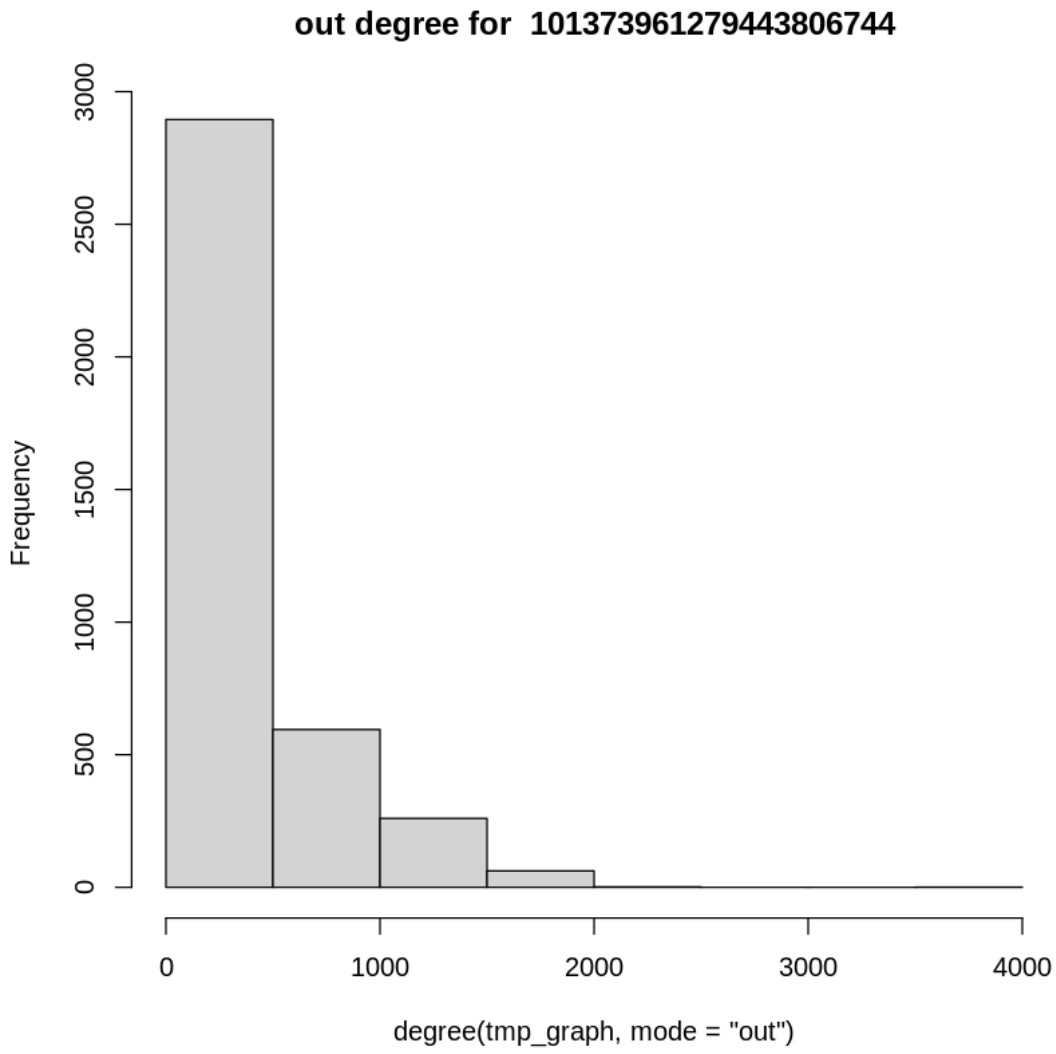

out degree for 109327480479767108490
Frequency
0
200
400
600
0
200
400
600
800
degree(tmp_graph, mode = "out")


in degree for 11562556499399014554 6
Frequency
0
50
100
150
200
250
0
50
100
150
200
degree(tmp_graph, mode = "in")

```
[1] 17
```


out degree for 11562556499399014554 6
Frequency
0
200
400
600
800
0
200
400
600
800
1000
degree(tmp_graph, mode = "out")


in degree for 101373961279443806744
Frequency
0
500
1000
1500
0
500
1000
1500
2000
degree(tmp_graph, mode = "in")

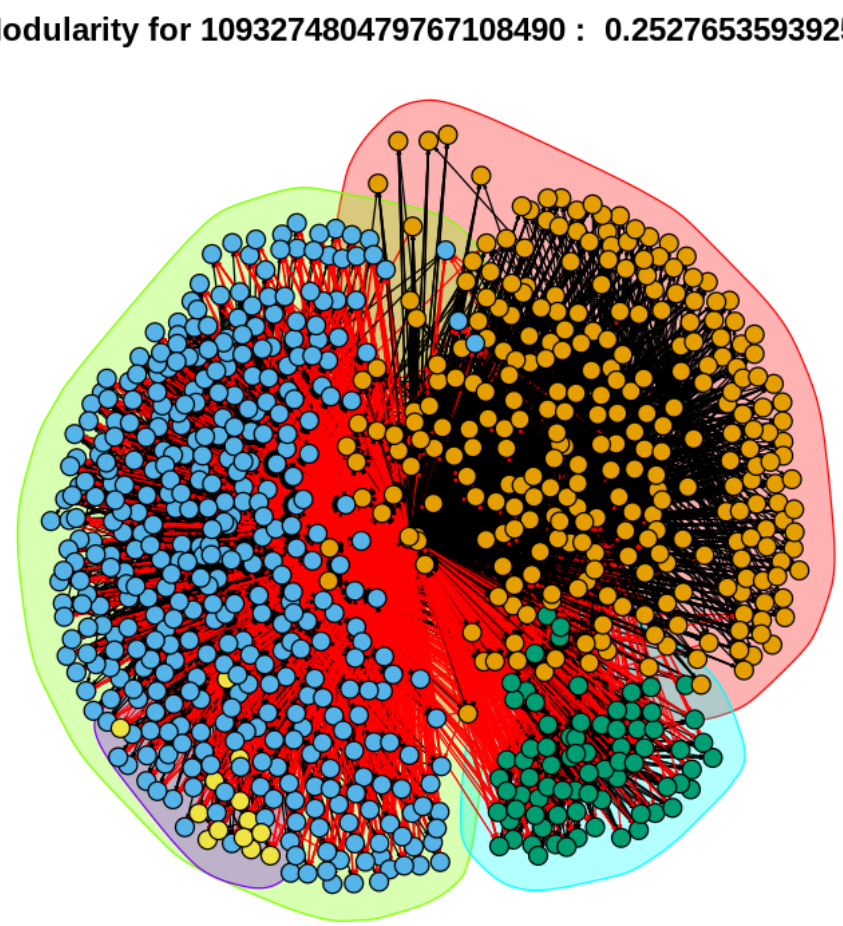


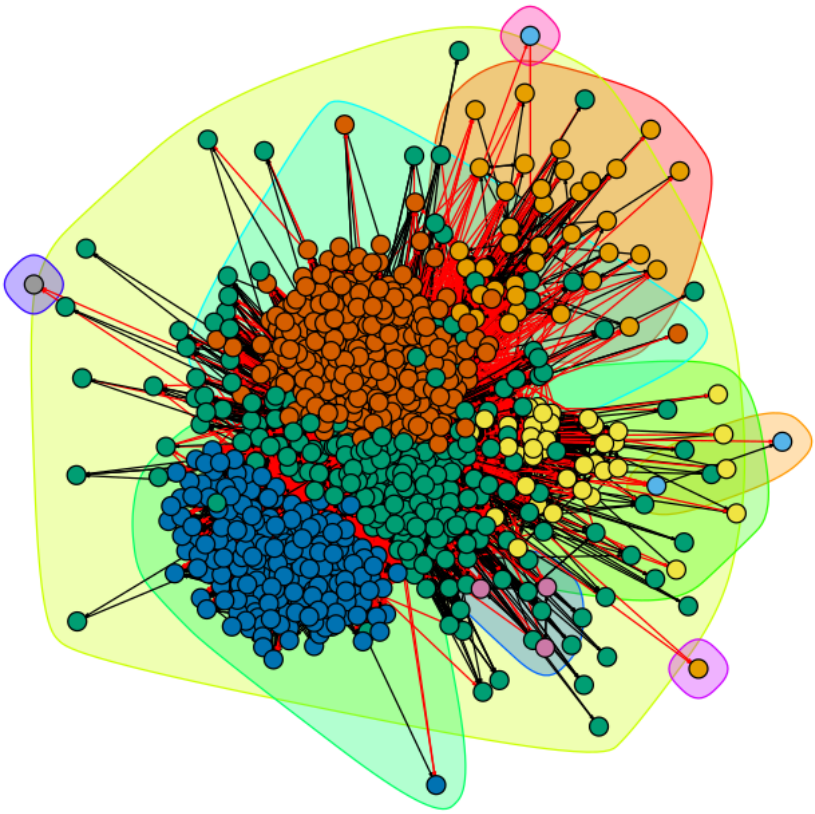
The networks do indeed have similar out degree distributions. The in degree distributions seems to vary witht he first network having a more skewed graph centering smaller degrees as where the second graph is slightly less and the last one seems to be the most unskewed.

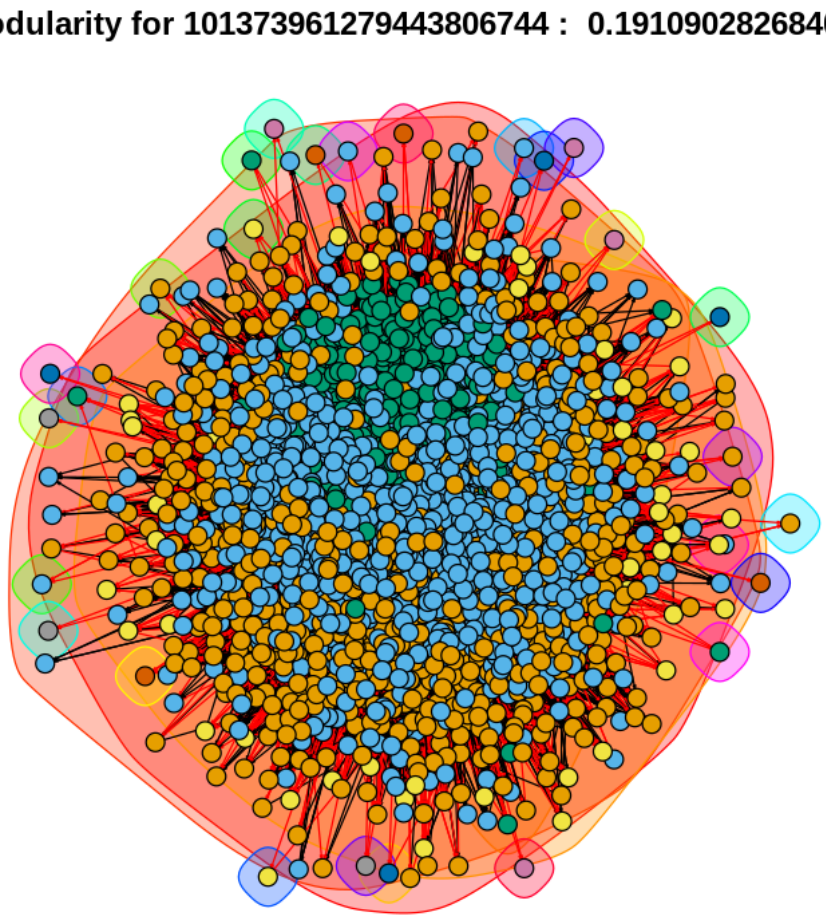
24 Q20

[ ]:
```
for (i in 1: length(interest_node)){
    graph_ind <- which(node_names==interest_node[i])
    tmp_graph = final_graph[[graph_ind]]
    walktrap = cluster_walktrap(tmp_graph)
    # cat("Modularity for", interest_node[i], ": ", modularity(walktrap), "\n")
    plot(walktrap, tmp_graph,
```

```
    vertex.label = NA, vertex.size = 5, edge.arrow.size = 0.1,␣
 ↪main=paste("Modularity for", interest_node[i], ": ", modularity(walktrap)))
}
```




Modularity for 115625564993990145546 : 0.319472554647349



The modularity scores varies in the above 3 plots with ID: 115625564993990145546 having the highest modularity and ID: 10137396127944380674 4 having the lowest modularity.

25 Q21

Homogeneity is a measure of similarity between samples (or rather, how many samples of the same class are put together) in a cluster. Completeness is a measure of how many similar samples (samples of the same class) are assigned to the same cluster using the clustering algorithm.

26 Q22

[ ]:
```
for (i in 1:length(interest_node)){
    data = paste("gplus/", interest_node[i], ".circles", sep = "")
    circle_users = c()
    circle = c()
    comm = c()
    graph_ind = which(node_names==interest_node[i])
    tmp_graph = final_graph[[graph_ind]]
    walktrap = cluster_walktrap(tmp_graph)
    for (j in c(1:length(readLines(data)))) {
      strings = strsplit(readLines(data)[j], "\t")
      strings = tail(strings[[1]], -1)
      circle_users = c(circle_users, strings)
      circle = c(circle, rep(j,length(strings)))
      }
    for (k in circle_users) {
      for (l in k){
        circle_id = membership(walktrap)[[l]]
        comm = c(comm, circle_id)
      }
    }

    HC = entropy(circle)
    HK = entropy(comm)
    HCK = condentropy(circle, comm)
    HKC = condentropy(comm, circle)
    h = 1-(HCK/HC)
    c = 1-(HKC/HK)
    cat(interest_node[i], "h:", h, "c:", c, "\n")
}
```

```
109327480479767108490 h: 0.5249326 c: 0.5497246
115625564993990145546 h: 0.1060277 c: 0.3821831
101373961279443806744 h: 0.0003744895 c: 0.0008590604
```

There are no negative values however, the values for each ID varies widely. Since h represents the similarity between samples in a given cluster and c represents the similarity of class samples in each cluster, the first ID has the highest similarities between samples and cluster, they have the deepest communities. The last ID indicates the least heterogeneity and completeness score, indicating that their communities are the most sparse/shallow.

27 Q23

27.1 Initialization

[ ]:
```
!pip install stellargraph
import stellargraph as sg
from stellargraph.mapper import FullBatchNodeGenerator
from stellargraph.layer import GCN

from tensorflow.keras import layers, optimizers, losses, metrics, Model
from sklearn import preprocessing, model_selection
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

from tensorflow.keras.callbacks import EarlyStopping
```

27.2 Data Preparation

[ ]:
```
#import dataset
dataset = sg.datasets.Cora()
G, node_subjects = dataset.load()

print(G.info())
```

```
StellarGraph: Undirected multigraph
 Nodes: 2708, Edges: 5429

 Node types:
  paper: [2708]
    Features: float32 vector, length 1433
    Edge types: paper-cites->paper

 Edge types:
    paper-cites->paper: [5429]
        Weights: all 1 (default)
        Features: none
```

[ ]:
```
#train-test splitting
train_subjects, test_subjects = model_selection.train_test_split(
    node_subjects, train_size=140, test_size=None, stratify=node_subjects
)
val_subjects, test_subjects = model_selection.train_test_split(
    test_subjects, train_size=800, test_size=None, stratify=test_subjects
)
```

[ ]:
```
#one-hot encoding
target_encoding = preprocessing.LabelBinarizer()

train_targets = target_encoding.fit_transform(train_subjects)
val_targets = target_encoding.transform(val_subjects)
test_targets = target_encoding.transform(test_subjects)
```

27.3 Model Training

[ ]:
```
generator = FullBatchNodeGenerator(G, method="gcn")
train_gen = generator.flow(train_subjects.index, train_targets)
gcn = GCN(
    layer_sizes=[16, 16], activations=["relu", "relu"], generator=generator,␣
 ↪dropout=0.5
)
x_inp, x_out = gcn.in_out_tensors()
predictions = layers.Dense(units=train_targets.shape[1],␣
 ↪activation="softmax")(x_out)
```

Using GCN (local pooling) filters…

[ ]:
```
model = Model(inputs=x_inp, outputs=predictions)
model.compile(
    optimizer=optimizers.Adam(lr=0.01),
    loss=losses.categorical_crossentropy,
    metrics=["acc"],
)

val_gen = generator.flow(val_subjects.index, val_targets)

es_callback = EarlyStopping(monitor="val_acc", patience=50,␣
 ↪restore_best_weights=True)

history = model.fit(
    train_gen,
    epochs=200,
    validation_data=val_gen,
    verbose=2,
    shuffle=False,
    callbacks=[es_callback],
)

sg.utils.plot_history(history)

test_gen = generator.flow(test_subjects.index, test_targets)

test_metrics = model.evaluate(test_gen)
```

```
print("\nTest Set Metrics:")
for name, val in zip(model.metrics_names, test_metrics):
    print("\t{}: {:0.4f}".format(name, val))
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105:
UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

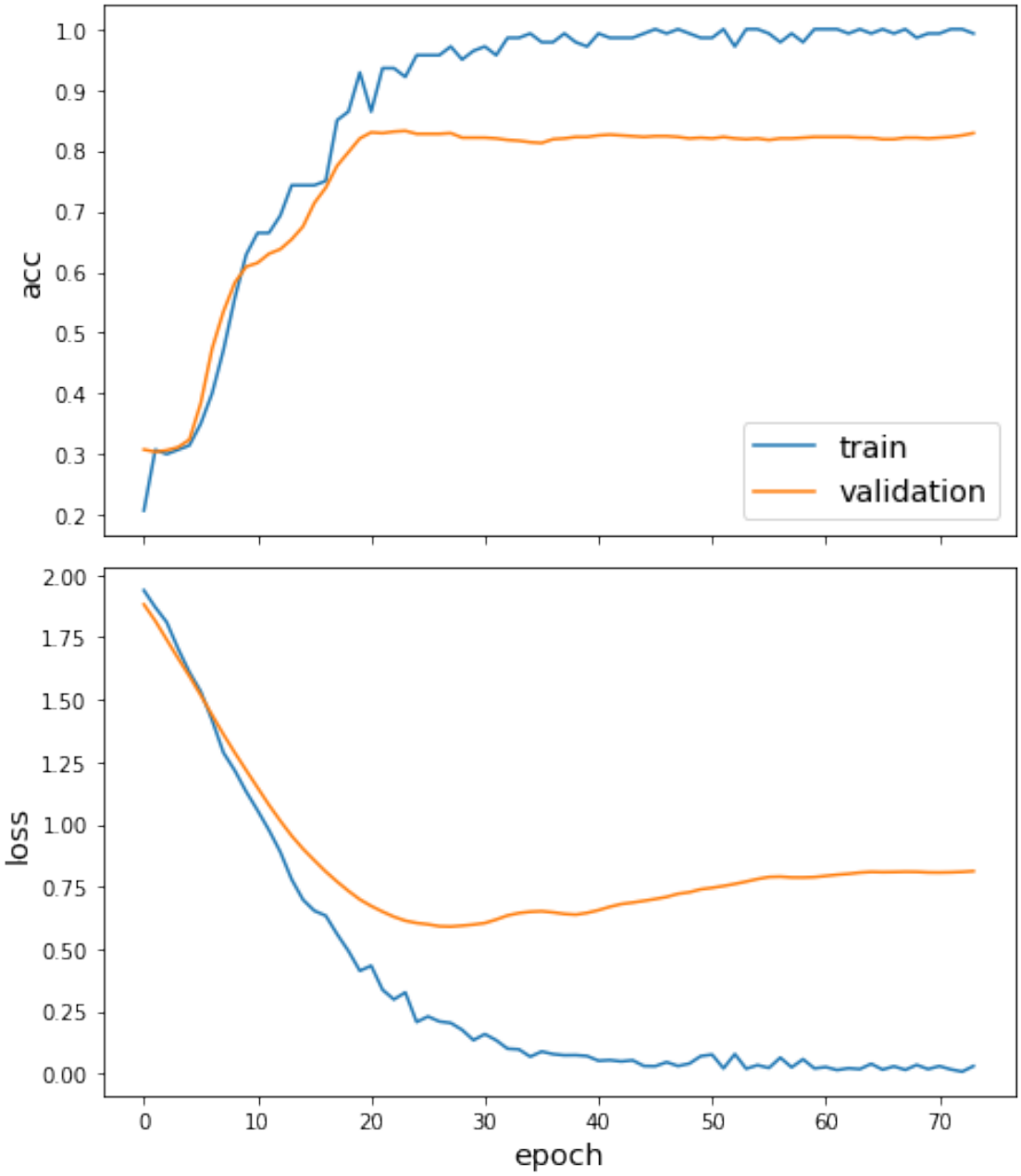
```
Epoch 1/200
1/1 - 2s - loss: 1.9377 - acc: 0.2071 - val_loss: 1.8812 - val_acc: 0.3075 -
2s/epoch - 2s/step
Epoch 2/200
1/1 - 0s - loss: 1.8701 - acc: 0.3071 - val_loss: 1.8152 - val_acc: 0.3038 -
174ms/epoch - 174ms/step
Epoch 3/200
1/1 - 0s - loss: 1.8106 - acc: 0.3000 - val_loss: 1.7405 - val_acc: 0.3063 -
160ms/epoch - 160ms/step
Epoch 4/200
1/1 - 0s - loss: 1.7046 - acc: 0.3071 - val_loss: 1.6659 - val_acc: 0.3113 -
281ms/epoch - 281ms/step
Epoch 5/200
1/1 - 0s - loss: 1.6100 - acc: 0.3143 - val_loss: 1.5940 - val_acc: 0.3237 -
271ms/epoch - 271ms/step
Epoch 6/200
1/1 - 0s - loss: 1.5324 - acc: 0.3500 - val_loss: 1.5175 - val_acc: 0.3837 -
281ms/epoch - 281ms/step
Epoch 7/200
1/1 - 0s - loss: 1.4170 - acc: 0.4000 - val_loss: 1.4378 - val_acc: 0.4737 -
297ms/epoch - 297ms/step
Epoch 8/200
1/1 - 0s - loss: 1.2869 - acc: 0.4714 - val_loss: 1.3602 - val_acc: 0.5350 -
338ms/epoch - 338ms/step
Epoch 9/200
1/1 - 0s - loss: 1.2150 - acc: 0.5571 - val_loss: 1.2867 - val_acc: 0.5825 -
236ms/epoch - 236ms/step
Epoch 10/200
1/1 - 0s - loss: 1.1306 - acc: 0.6286 - val_loss: 1.2163 - val_acc: 0.6087 -
274ms/epoch - 274ms/step
Epoch 11/200
1/1 - 0s - loss: 1.0556 - acc: 0.6643 - val_loss: 1.1469 - val_acc: 0.6150 -
326ms/epoch - 326ms/step
Epoch 12/200
1/1 - 0s - loss: 0.9766 - acc: 0.6643 - val_loss: 1.0779 - val_acc: 0.6300 -
271ms/epoch - 271ms/step
Epoch 13/200
1/1 - 0s - loss: 0.8891 - acc: 0.6929 - val_loss: 1.0137 - val_acc: 0.6375 -
256ms/epoch - 256ms/step
Epoch 14/200
```

```
1/1 - 0s - loss: 0.7789 - acc: 0.7429 - val_loss: 0.9536 - val_acc: 0.6538 -
323ms/epoch - 323ms/step
Epoch 15/200
1/1 - 0s - loss: 0.6974 - acc: 0.7429 - val_loss: 0.9013 - val_acc: 0.6750 -
269ms/epoch - 269ms/step
Epoch 16/200
1/1 - 0s - loss: 0.6530 - acc: 0.7429 - val_loss: 0.8551 - val_acc: 0.7138 -
315ms/epoch - 315ms/step
Epoch 17/200
1/1 - 0s - loss: 0.6334 - acc: 0.7500 - val_loss: 0.8100 - val_acc: 0.7387 -
279ms/epoch - 279ms/step
Epoch 18/200
1/1 - 0s - loss: 0.5592 - acc: 0.8500 - val_loss: 0.7699 - val_acc: 0.7750 -
282ms/epoch - 282ms/step
Epoch 19/200
1/1 - 0s - loss: 0.4918 - acc: 0.8643 - val_loss: 0.7324 - val_acc: 0.7975 -
269ms/epoch - 269ms/step
Epoch 20/200
1/1 - 0s - loss: 0.4123 - acc: 0.9286 - val_loss: 0.6984 - val_acc: 0.8200 -
470ms/epoch - 470ms/step
Epoch 21/200
1/1 - 0s - loss: 0.4326 - acc: 0.8643 - val_loss: 0.6721 - val_acc: 0.8300 -
244ms/epoch - 244ms/step
Epoch 22/200
1/1 - 0s - loss: 0.3362 - acc: 0.9357 - val_loss: 0.6494 - val_acc: 0.8288 -
289ms/epoch - 289ms/step
Epoch 23/200
1/1 - 0s - loss: 0.2975 - acc: 0.9357 - val_loss: 0.6290 - val_acc: 0.8313 -
315ms/epoch - 315ms/step
Epoch 24/200
1/1 - 0s - loss: 0.3255 - acc: 0.9214 - val_loss: 0.6131 - val_acc: 0.8325 -
270ms/epoch - 270ms/step
Epoch 25/200
1/1 - 0s - loss: 0.2071 - acc: 0.9571 - val_loss: 0.6039 - val_acc: 0.8275 -
296ms/epoch - 296ms/step
Epoch 26/200
1/1 - 0s - loss: 0.2293 - acc: 0.9571 - val_loss: 0.5986 - val_acc: 0.8275 -
374ms/epoch - 374ms/step
Epoch 27/200
1/1 - 0s - loss: 0.2091 - acc: 0.9571 - val_loss: 0.5910 - val_acc: 0.8275 -
258ms/epoch - 258ms/step
Epoch 28/200
1/1 - 0s - loss: 0.2033 - acc: 0.9714 - val_loss: 0.5897 - val_acc: 0.8288 -
281ms/epoch - 281ms/step
Epoch 29/200
1/1 - 0s - loss: 0.1757 - acc: 0.9500 - val_loss: 0.5933 - val_acc: 0.8213 -
289ms/epoch - 289ms/step
Epoch 30/200
```

```
1/1 - 0s - loss: 0.1343 - acc: 0.9643 - val_loss: 0.5976 - val_acc: 0.8213 -
346ms/epoch - 346ms/step
Epoch 31/200
1/1 - 0s - loss: 0.1585 - acc: 0.9714 - val_loss: 0.6037 - val_acc: 0.8213 -
262ms/epoch - 262ms/step
Epoch 32/200
1/1 - 0s - loss: 0.1339 - acc: 0.9571 - val_loss: 0.6178 - val_acc: 0.8200 -
290ms/epoch - 290ms/step
Epoch 33/200
1/1 - 0s - loss: 0.1000 - acc: 0.9857 - val_loss: 0.6342 - val_acc: 0.8175 -
263ms/epoch - 263ms/step
Epoch 34/200
1/1 - 0s - loss: 0.0973 - acc: 0.9857 - val_loss: 0.6438 - val_acc: 0.8163 -
310ms/epoch - 310ms/step
Epoch 35/200
1/1 - 0s - loss: 0.0679 - acc: 0.9929 - val_loss: 0.6490 - val_acc: 0.8138 -
407ms/epoch - 407ms/step
Epoch 36/200
1/1 - 0s - loss: 0.0888 - acc: 0.9786 - val_loss: 0.6512 - val_acc: 0.8125 -
471ms/epoch - 471ms/step
Epoch 37/200
1/1 - 0s - loss: 0.0784 - acc: 0.9786 - val_loss: 0.6466 - val_acc: 0.8188 -
328ms/epoch - 328ms/step
Epoch 38/200
1/1 - 0s - loss: 0.0737 - acc: 0.9929 - val_loss: 0.6404 - val_acc: 0.8200 -
276ms/epoch - 276ms/step
Epoch 39/200
1/1 - 0s - loss: 0.0742 - acc: 0.9786 - val_loss: 0.6375 - val_acc: 0.8225 -
404ms/epoch - 404ms/step
Epoch 40/200
1/1 - 0s - loss: 0.0706 - acc: 0.9714 - val_loss: 0.6445 - val_acc: 0.8225 -
455ms/epoch - 455ms/step
Epoch 41/200
1/1 - 0s - loss: 0.0518 - acc: 0.9929 - val_loss: 0.6554 - val_acc: 0.8250 -
228ms/epoch - 228ms/step
Epoch 42/200
1/1 - 0s - loss: 0.0542 - acc: 0.9857 - val_loss: 0.6689 - val_acc: 0.8263 -
424ms/epoch - 424ms/step
Epoch 43/200
1/1 - 0s - loss: 0.0491 - acc: 0.9857 - val_loss: 0.6800 - val_acc: 0.8250 -
430ms/epoch - 430ms/step
Epoch 44/200
1/1 - 0s - loss: 0.0536 - acc: 0.9857 - val_loss: 0.6859 - val_acc: 0.8238 -
287ms/epoch - 287ms/step
Epoch 45/200
1/1 - 0s - loss: 0.0307 - acc: 0.9929 - val_loss: 0.6928 - val_acc: 0.8225 -
276ms/epoch - 276ms/step
Epoch 46/200
```

```
1/1 - 0s - loss: 0.0299 - acc: 1.0000 - val_loss: 0.7002 - val_acc: 0.8238 -
333ms/epoch - 333ms/step
Epoch 47/200
1/1 - 0s - loss: 0.0461 - acc: 0.9929 - val_loss: 0.7085 - val_acc: 0.8238 -
276ms/epoch - 276ms/step
Epoch 48/200
1/1 - 0s - loss: 0.0307 - acc: 1.0000 - val_loss: 0.7210 - val_acc: 0.8225 -
322ms/epoch - 322ms/step
Epoch 49/200
1/1 - 0s - loss: 0.0403 - acc: 0.9929 - val_loss: 0.7273 - val_acc: 0.8200 -
277ms/epoch - 277ms/step
Epoch 50/200
1/1 - 0s - loss: 0.0703 - acc: 0.9857 - val_loss: 0.7392 - val_acc: 0.8213 -
283ms/epoch - 283ms/step
Epoch 51/200
1/1 - 0s - loss: 0.0765 - acc: 0.9857 - val_loss: 0.7450 - val_acc: 0.8200 -
306ms/epoch - 306ms/step
Epoch 52/200
1/1 - 0s - loss: 0.0211 - acc: 1.0000 - val_loss: 0.7523 - val_acc: 0.8225 -
494ms/epoch - 494ms/step
Epoch 53/200
1/1 - 0s - loss: 0.0786 - acc: 0.9714 - val_loss: 0.7605 - val_acc: 0.8200 -
351ms/epoch - 351ms/step
Epoch 54/200
1/1 - 0s - loss: 0.0198 - acc: 1.0000 - val_loss: 0.7701 - val_acc: 0.8188 -
316ms/epoch - 316ms/step
Epoch 55/200
1/1 - 0s - loss: 0.0342 - acc: 1.0000 - val_loss: 0.7804 - val_acc: 0.8200 -
461ms/epoch - 461ms/step
Epoch 56/200
1/1 - 0s - loss: 0.0229 - acc: 0.9929 - val_loss: 0.7884 - val_acc: 0.8175 -
297ms/epoch - 297ms/step
Epoch 57/200
1/1 - 0s - loss: 0.0641 - acc: 0.9786 - val_loss: 0.7898 - val_acc: 0.8200 -
245ms/epoch - 245ms/step
Epoch 58/200
1/1 - 0s - loss: 0.0254 - acc: 0.9929 - val_loss: 0.7867 - val_acc: 0.8200 -
343ms/epoch - 343ms/step
Epoch 59/200
1/1 - 0s - loss: 0.0577 - acc: 0.9786 - val_loss: 0.7864 - val_acc: 0.8213 -
366ms/epoch - 366ms/step
Epoch 60/200
1/1 - 0s - loss: 0.0211 - acc: 1.0000 - val_loss: 0.7880 - val_acc: 0.8225 -
336ms/epoch - 336ms/step
Epoch 61/200
1/1 - 0s - loss: 0.0265 - acc: 1.0000 - val_loss: 0.7927 - val_acc: 0.8225 -
406ms/epoch - 406ms/step
Epoch 62/200
```

```
1/1 - 0s - loss: 0.0145 - acc: 1.0000 - val_loss: 0.7976 - val_acc: 0.8225 -
313ms/epoch - 313ms/step
Epoch 63/200
1/1 - 0s - loss: 0.0217 - acc: 0.9929 - val_loss: 0.8011 - val_acc: 0.8225 -
339ms/epoch - 339ms/step
Epoch 64/200
1/1 - 0s - loss: 0.0183 - acc: 1.0000 - val_loss: 0.8060 - val_acc: 0.8213 -
288ms/epoch - 288ms/step
Epoch 65/200
1/1 - 0s - loss: 0.0393 - acc: 0.9929 - val_loss: 0.8089 - val_acc: 0.8213 -
342ms/epoch - 342ms/step
Epoch 66/200
1/1 - 0s - loss: 0.0162 - acc: 1.0000 - val_loss: 0.8079 - val_acc: 0.8188 -
290ms/epoch - 290ms/step
Epoch 67/200
1/1 - 0s - loss: 0.0294 - acc: 0.9929 - val_loss: 0.8084 - val_acc: 0.8188 -
356ms/epoch - 356ms/step
Epoch 68/200
1/1 - 0s - loss: 0.0152 - acc: 1.0000 - val_loss: 0.8095 - val_acc: 0.8213 -
238ms/epoch - 238ms/step
Epoch 69/200
1/1 - 0s - loss: 0.0351 - acc: 0.9857 - val_loss: 0.8090 - val_acc: 0.8213 -
301ms/epoch - 301ms/step
Epoch 70/200
1/1 - 0s - loss: 0.0182 - acc: 0.9929 - val_loss: 0.8067 - val_acc: 0.8200 -
341ms/epoch - 341ms/step
Epoch 71/200
1/1 - 0s - loss: 0.0302 - acc: 0.9929 - val_loss: 0.8063 - val_acc: 0.8213 -
307ms/epoch - 307ms/step
Epoch 72/200
1/1 - 0s - loss: 0.0171 - acc: 1.0000 - val_loss: 0.8072 - val_acc: 0.8225 -
254ms/epoch - 254ms/step
Epoch 73/200
1/1 - 0s - loss: 0.0080 - acc: 1.0000 - val_loss: 0.8089 - val_acc: 0.8250 -
281ms/epoch - 281ms/step
Epoch 74/200
1/1 - 0s - loss: 0.0303 - acc: 0.9929 - val_loss: 0.8120 - val_acc: 0.8288 -
322ms/epoch - 322ms/step
1/1 [==============================] - 0s 108ms/step - loss: 0.6814 - acc:
0.8009

Test Set Metrics:
        loss: 0.6814
        acc: 0.8009
```



27.4 Evaluation

[ ]:
```
all_nodes = node_subjects.index
all_gen = generator.flow(all_nodes)
all_predictions = model.predict(all_gen)

node_predictions = target_encoding.inverse_transform(all_predictions.squeeze())
```

```
df = pd.DataFrame({"Predicted": node_predictions, "True": node_subjects})
df.head(20)
```

[ ]:

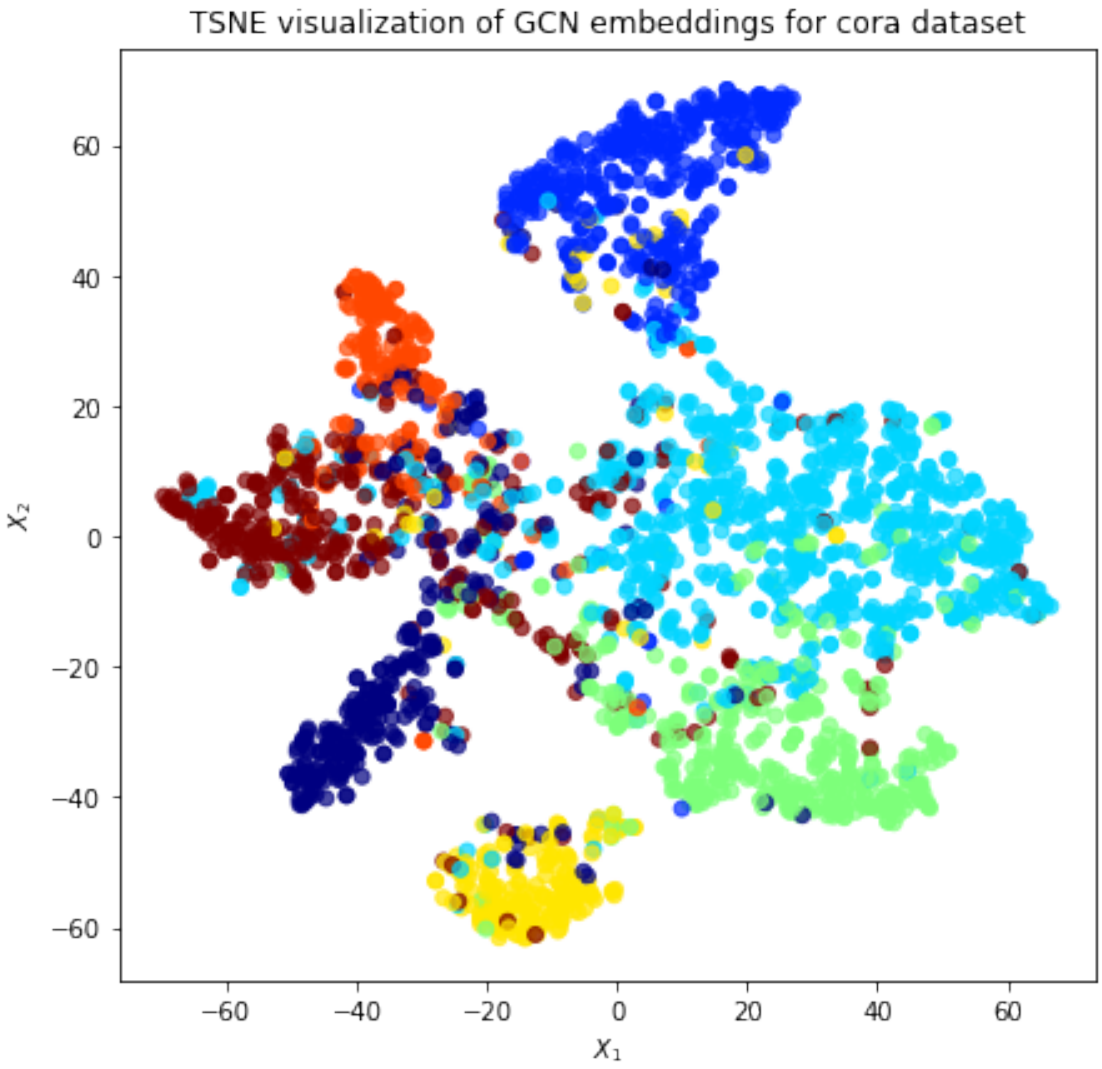
	Predicted	True
31336	Neural_Networks	Neural_Networks
1061127	Rule_Learning	Rule_Learning
1106406	Reinforcement_Learning	Reinforcement_Learning
13195	Reinforcement_Learning	Reinforcement_Learning
37879	Probabilistic_Methods	Probabilistic_Methods
1126012	Probabilistic_Methods	Probabilistic_Methods
1107140	Case_Based	Theory
1102850	Neural_Networks	Neural_Networks
31349	Neural_Networks	Neural_Networks
1106418	Theory	Theory
1123188	Neural_Networks	Neural_Networks
1128990	Genetic_Algorithms	Genetic_Algorithms
109323	Probabilistic_Methods	Probabilistic_Methods
217139	Genetic_Algorithms	Case_Based
31353	Neural_Networks	Neural_Networks
32083	Neural_Networks	Neural_Networks
1126029	Reinforcement_Learning	Reinforcement_Learning
1118017	Neural_Networks	Neural_Networks
49482	Neural_Networks	Neural_Networks
753265	Theory	Neural_Networks

[ ]:
```
transform = TSNE

embedding_model = Model(inputs=x_inp, outputs=x_out)
emb = embedding_model.predict(all_gen)
emb.shape

X = emb.squeeze(0)
X.shape

trans = transform(n_components=2)
X_reduced = trans.fit_transform(X)
X_reduced.shape

fig, ax = plt.subplots(figsize=(7, 7))
ax.scatter(
    X_reduced[:, 0],
    X_reduced[:, 1],
    c=node_subjects.astype("category").cat.codes,
    cmap="jet",
    alpha=0.7,
)
ax.set(
```

```
    aspect="equal",
    xlabel="$X_1$",
    ylabel="$X_2$",
    title=f"{transform.__name__} visualization of GCN embeddings for cora␣
  ↪dataset",
)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783:
FutureWarning: The default initialization in TSNE will change from 'random' to
'pca' in 1.2.
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793:
FutureWarning: The default learning rate in TSNE will change from 200.0 to
'auto' in 1.2.
  FutureWarning,
```

[ ]:
```
[Text(0, 0.5, '$X_2$'),
 Text(0.5, 0, '$X_1$'),
 Text(0.5, 1.0, 'TSNE visualization of GCN embeddings for cora dataset'),
 None]
```



28 Q24

28.1 Node2vec

[ ]:
```
!pip install stellargraph

import matplotlib.pyplot as plt

from sklearn.manifold import TSNE
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

```
import os
import networkx as nx
import numpy as np
import pandas as pd

from stellargraph.data import BiasedRandomWalk
from stellargraph import StellarGraph
from stellargraph import datasets
```

[ ]:
```
dataset = datasets.Cora()
G, node_subjects = dataset.load(largest_connected_component_only=True)
print(G.info())
```

```
StellarGraph: Undirected multigraph
 Nodes: 2485, Edges: 5209

 Node types:
  paper: [2485]
    Features: float32 vector, length 1433
    Edge types: paper-cites->paper

 Edge types:
    paper-cites->paper: [5209]
        Weights: all 1 (default)
        Features: none
```

[ ]:
```
rw = BiasedRandomWalk(G)

walks = rw.run(
    nodes=list(G.nodes()),
    length=100,
    n=10,
    p=0.5,
    q=2.0,
)
print("Number of random walks: {}".format(len(walks)))
```

```
Number of random walks: 24850
```

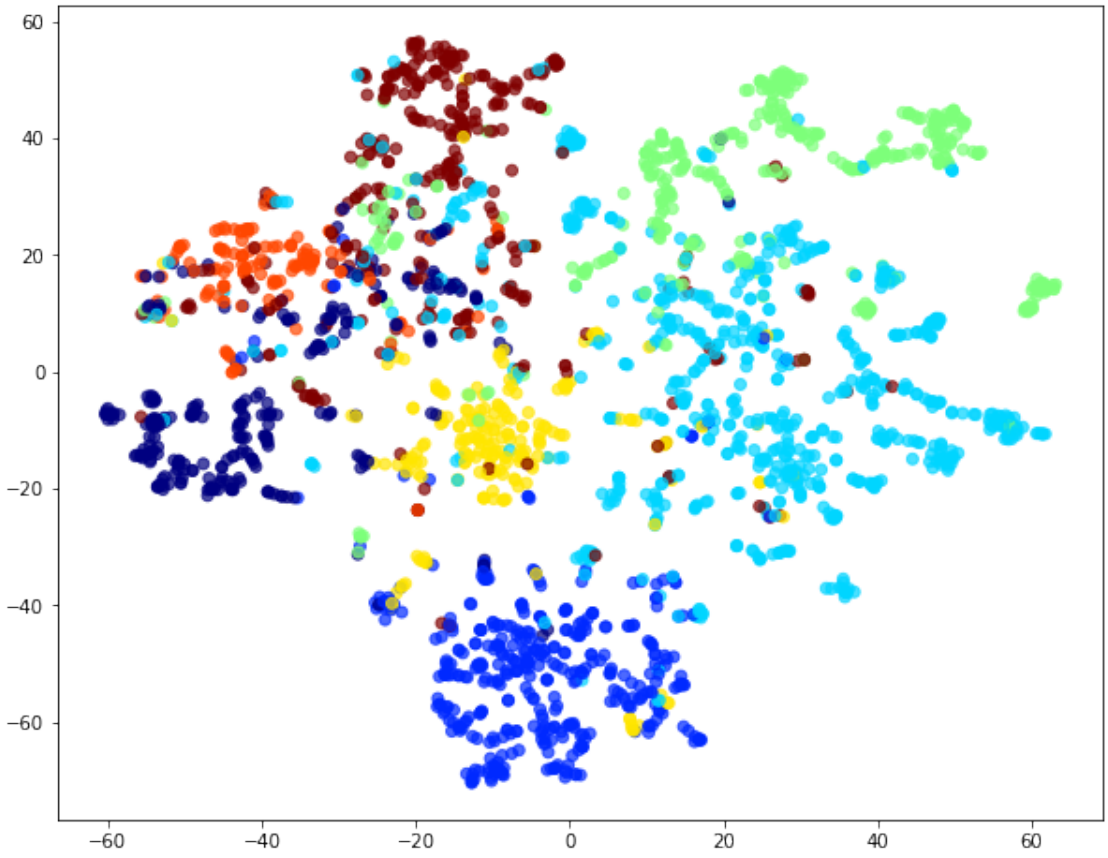
[ ]:
```
from gensim.models import Word2Vec

str_walks = [[str(n) for n in walk] for walk in walks]
model = Word2Vec(str_walks, size=128, window=5, min_count=0, sg=1, workers=2,␣
 ↪iter=1)
```

[ ]:
```
node_ids = model.wv.index2word
node_embeddings = (
```

```
    model.wv.vectors
)
node_targets = node_subjects[[int(node_id) for node_id in node_ids]]

tsne = TSNE(n_components=2)
node_embeddings_2d = tsne.fit_transform(node_embeddings)

alpha = 0.7
label_map = {l: i for i, l in enumerate(np.unique(node_targets))}
node_colours = [label_map[target] for target in node_targets]

plt.figure(figsize=(10, 8))
plt.scatter(
    node_embeddings_2d[:, 0],
    node_embeddings_2d[:, 1],
    c=node_colours,
    cmap="jet",
    alpha=alpha,
)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783:
FutureWarning: The default initialization in TSNE will change from 'random' to
'pca' in 1.2.
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793:
FutureWarning: The default learning rate in TSNE will change from 200.0 to
'auto' in 1.2.
  FutureWarning,
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x7ff24b50c8d0>
```



[ ]:
```
X = node_embeddings
y = np.array(node_targets)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

clf1 = RandomForestClassifier()
clf1.fit(X_train, y_train)

y_pred = clf1.predict(X_test)

print("Accuracy Score: ", accuracy_score(y_test, y_pred))
```

```
Accuracy Score:  0.8485254691689008
```

28.2 Text features

[ ]:
```
cora_cites_file = os.path.join(dataset.base_directory, "cora.cites")
cora_content_file = os.path.join(dataset.base_directory, "cora.content")

cora_cites = pd.read_csv(
```

```
    cora_cites_file,
    sep="\t",
    header=None,
    names=["target", "source"],
)
cora_feature_names = [f"w{i}" for i in range(1433)]
cora_raw_content = pd.read_csv(
    cora_content_file,
    sep="\t",
    header=None,
    names=["id", *cora_feature_names, "subject"],
)

#one-hot encoding
counter = 0
encoding_list = []

while counter < cora_content_str_subject["subject"].shape[0]:
  category = cora_raw_content.at[counter,"subject"]
  if category == "Case_Based":
    encoding_list.append(0)
  elif category == "Genetic_Algorithms":
    encoding_list.append(1)
  elif category == "Neural_Networks":
    encoding_list.append(2)
  elif category == "Probabilistic_Methods":
    encoding_list.append(3)
  elif category == "Reinforcement_Learning":
    encoding_list.append(4)
  elif category == "Rule_Learning":
    encoding_list.append(5)
  elif category == "Theory":
    encoding_list.append(6)
  counter += 1

cora_raw_content["one_hot"] = encoding_list
cora_raw_content
```

[ ]:

	id	w0	w1	w2	w3	w4	w5	w6	w7	w8	…	w1425	w1426	w1427	\
0	31336	0	0	0	0	0	0	0	0	0	…	0	1	0	
1	1061127	0	0	0	0	0	0	0	0	0	…	1	0	0	
2	1106406	0	0	0	0	0	0	0	0	0	…	0	0	0	
3	13195	0	0	0	0	0	0	0	0	0	…	0	0	0	
4	37879	0	0	0	0	0	0	0	0	0	…	0	0	0	
…	…	..	..	..	..	..	..	..	..	..	…	…	…	…	
2703	1128975	0	0	0	0	0	0	0	0	0	…	0	0	0	
2704	1128977	0	0	0	0	0	0	0	0	0	…	0	0	0	

```
2705  1128978   0   0   0   0   0   0   0   0   0  …      0      0      0
2706   117328   0   0   0   0   1   0   0   0   0  …      0      0      0
2707    24043   0   0   0   0   0   0   0   0   0  …      0      0      0

      w1428  w1429  w1430  w1431  w1432                 subject  one_hot
0         0      0      0      0      0         Neural_Networks        2
1         0      0      0      0      0           Rule_Learning        5
2         0      0      0      0      0  Reinforcement_Learning        4
3         0      0      0      0      0  Reinforcement_Learning        4
4         0      0      0      0      0   Probabilistic_Methods        3
…     …     …     …     …     …                       …      …
2703      0      0      0      0      0      Genetic_Algorithms        1
2704      0      0      0      0      0      Genetic_Algorithms        1
2705      0      0      0      0      0      Genetic_Algorithms        1
2706      0      0      0      0      0               Case_Based        0
2707      0      0      0      0      0         Neural_Networks        2

[2708 rows x 1436 columns]
```

[ ]:
```
#convert dataframe to numpy array
cora_input = cora_raw_content.drop(columns=['id','subject', 'one_hot'])
cora_target = cora_raw_content[['one_hot']]

cora_input = cora_input.to_numpy()
cora_target = cora_target.to_numpy()
```

[ ]:
```
#split test, train
X_train, X_test, y_train, y_test = train_test_split(cora_input, cora_target,␣
 ↪test_size=0.3)
print(X_train.shape)
print(X_test.shape)
```

```
(1895, 1433)
(813, 1433)
```

[ ]:
```
# Scale Data
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[ ]:
```
# Dimension Reduction
from sklearn.decomposition import PCA

pca = PCA(n_components=200)
```

```
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

[ ]:
```
# Classifier
clf2 = RandomForestClassifier()
clf2.fit(X_train, y_train)

y_pred = clf2.predict(X_test)

from sklearn.metrics import accuracy_score
print("Accuracy Score: ", accuracy_score(y_test, y_pred))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples,), for example using
ravel().
  This is separate from the ipykernel package so we can avoid doing imports
until

Accuracy Score:  0.6961869618696187
```

The accuracy in prediction results using text features peaks at ~0.70 indicating that Node2Vec outperforms as it achieves a higher accuracy score of 0.85.

28.3 Combined

[ ]:
```
#Ensembling--Averaging
from sklearn.metrics import mean_squared_error

X_train, X_test, y_train, y_test = train_test_split(cora_input, cora_target,␣
 ↪test_size=0.3)

clf1.fit(X_train, y_train)
clf2.fit(X_train, y_train)

pred_1 = clf1.predict(X_test)
pred_2 = clf2.predict(X_test)


pred_final = np.round_((pred_1+pred_2)/2)

print("RMSE: ", mean_squared_error(y_test, pred_final))
print("Accuracy Score: ", accuracy_score(y_test, pred_final))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:6:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples,), for example using
ravel().
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:7:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples,), for example using
ravel().
  import sys
```

```
RMSE:  2.060270602706027
Accuracy Score:  0.7306273062730627
```

29 Q25

[ ]:
```
import numpy as np
import os
import networkx as nx
from sklearn.model_selection import train_test_split
import pandas as pd
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from numpy import dot
from numpy.linalg import norm


from collections import Counter
import matplotlib.pyplot as plt
```

[ ]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

[ ]:
```
all_data = []
all_edges = []
filepath = "/content/gdrive/My Drive/cora/"

for root,dirs,files in os.walk(filepath):
    for file in files:
        if '.content' in file:
            with open(os.path.join(root,file),'r') as f:
                all_data.extend(f.read().splitlines())
        elif '.cites' in file:
            with open(os.path.join(root,file),'r') as f:
                all_edges.extend(f.read().splitlines())

# random_state = 42
# all_data = shuffle(all_data,random_state=random_state)
```

[ ]:
```
categories =  ['Reinforcement_Learning', 'Theory', 'Case_Based',␣
 ↪'Genetic_Algorithms', 'Probabilistic_Methods', 'Neural_Networks',␣
 ↪'Rule_Learning']
sorted(categories)
label_encoder = {}
i = 0
for cat in sorted(categories):
  label_encoder[cat] = i
  i +=1
label_encoder
```

[ ]:
```
{'Case_Based': 0,
 'Genetic_Algorithms': 1,
 'Neural_Networks': 2,
 'Probabilistic_Methods': 3,
 'Reinforcement_Learning': 4,
 'Rule_Learning': 5,
 'Theory': 6}
```

[ ]:
```
#parse the data
labels = []
nodes = []
X = []
element_to_ind  = {}

for i,data in enumerate(all_data):
    elements = data.split('\t')
    labels.append(label_encoder[elements[-1]])
    X.append(elements[1:-1])
    nodes.append(elements[0])
    element_to_ind[elements[0]]= i
X = np.array(X,dtype=int)
N = X.shape[0] #the number of nodes
F = X.shape[1] #the size of node features
print('X shape: ', X.shape)


#parse the edge
edge_list=[]
for edge in all_edges:
    e = edge.split('\t')
    edge_list.append((e[0],e[1]))

print('\nNumber of nodes (N): ', N)
print('\nNumber of features (F) of each node: ', F)
print('\nCategories: ', set(labels))
```

```
num_classes = len(set(labels))
print('\nNumber of classes: ', num_classes)
```

```
X shape:  (2708, 1433)

Number of nodes (N):  2708

Number of features (F) of each node:  1433

Categories:  {0, 1, 2, 3, 4, 5, 6}

Number of classes:  7
```

[ ]:
```
G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edge_list)
G = nx.relabel_nodes(G, element_to_ind)
print('Graph info: ', nx.info(G))
```

```
Graph info:  Graph with 2708 nodes and 5278 edges
```

[ ]:
```
nodes = list(G.nodes)
print(len(nodes))
list(G.neighbors(0))
```

```
2708
```

[ ]:
```
[258, 544, 8, 435, 14]
```

[ ]:
```
df = pd.DataFrame(list(zip(nodes, labels,X)),columns =['node',␣
↪'label','features'])
print(len(df))
df.head()
```

```
2708
```

[ ]:

	node	label	features
0	0	2	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
1	1	5	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, …
2	2	4	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
3	3	4	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
4	4	3	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …

[ ]:
```
Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
G = G.subgraph(Gcc[0])
gcc_nodes = list(G.nodes)
```

[ ]:
```
df = df.loc[df['node'].isin(gcc_nodes)]
df['node'] = list(range(len(df))) #rename nodes
df.head()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

[ ]:

	node	label	features
0	0	2	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
1	1	5	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, …
2	2	4	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
3	3	4	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
4	4	3	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …

[ ]:
```
train = df.groupby('label', group_keys=False).apply(lambda x: x.sample(20))
G = nx.relabel_nodes(G, df['node'])
```

[ ]:
```
def create_transition_matrix(g):
    vs = list(g.nodes)
    n = len(vs)
    adj = nx.adjacency_matrix(g)
    transition_matrix = adj/adj.sum(axis=1)

    return transition_matrix
```

[ ]:
```
def random_walk(g, num_steps, start_node, transition_matrix = None):
  if transition_matrix is None:
    transition_matrix = create_transition_matrix(g)
  #perform a random walk
  #return v
```

[ ]:
```
seeds_dict = {predicted:list(train[train['label'] == predicted]['node']) for␣
 ↪predicted in range(7)}

def random_walk_with_teleportation(g, num_steps, start_node,tp,predicted,␣
 ↪transition_matrix = None):
  #modify random walk code to add teleportation.
  #you can only teleport to a node belonging to the same class as the start node
  #if start_node_class = c1: teleport only to nodes with c1
  #write random walk
```

```
  #return v
```

```
  File "<ipython-input-15-3a04561676c5>", line 9
    #return
           ^
SyntaxError: unexpected EOF while parsing
```

[ ]:
```
#pagerank. NO teleportation, NO tfidf.
transition_matrix = create_transition_matrix(G)

num_samples = 1000
num_walk_steps = 100

visiting_freq_label = []
for i in range(transition_matrix.shape[0]):
  visiting_freq_label.append([0,0,0,0,0,0,0])

visiting_freq = [0 for i in range(transition_matrix.shape[0])]

for train_node,predicted in zip(train['node'],train['label']):
  #print (train_node,predicted)
  for i in range(num_samples):
      start_point = train_node
      end_node = random_walk(G, num_walk_steps, start_point, transition_matrix)
      visiting_freq_label[end_node][predicted] += 1
      visiting_freq[end_node] +=1
```

[ ]:
```
count = 0
for vf in visiting_freq:
  if vf ==0:
    count+=1
print('unvisited = ', count)
visiting_freq_label = np.asarray(visiting_freq_label)
preds = np.argmax(visiting_freq_label,axis = 1)
print(classification_report(df['label'], preds))
accuracy_score(df['label'], preds)
```

[ ]:
```
#pagerank. WITH telportation, without tfidf

#repeat above expeiment but this time use the teleportation random walk

#get metrics
```

[ ]:
```
vs = list(G.nodes)
n = len(vs)
```

```
adj = nx.adjacency_matrix(G)
transition = np.zeros((len(G.nodes), len(G.nodes)))

#for n1 in nodes:
  #for n2 in nodes:
    # if there is an edge between n1 and n2:
      # cos_sim = compute cosine similarity between features of n1 and n2
      # transition[n1,n2] = np.exp(cos_sim) #neumerator of softmax. #why do we need softmax?
#divide the values in transition by denominator of softmax. how will you do this?
```

[ ]:
```
#pagerank. Without teleportation. WITH TFIDF
transition_matrix = transition

#perfrom pagerank using our tf_idf based transition matrix
#use randon walk without teleporation
#get metrics
```

[ ]:
```
#pagerank. WITH teleportation WITH TFIDF
transition_matrix = transition

#same as above, except use random walk with teleportation
#get metrics
```