Impact of Social Media Information Spread Analysis on X

Progress Report

Colin Song (261049870)

I. Abstract

This project investigates the flow of information on X (formerly Twitter) by analyzing the platform's network structure and user interactions. The primary objective of this project is to identify subgraphs and their properties to

gain deeper insight into how information flows on X. Characterizing subgraphs and their properties reveals patterns of influence, the roles of gatekeepers, and the mechanisms by which information reaches distant parts of the network. These findings allow for a better understanding of the structural and behavioral factors of subgraphs that ultimately reveal how information flows on X.

II. Project Description

2.1 Purpose

The purpose of this project is to gain a better understanding of how information flows on social media platforms primarily on X. Information flow is crucial for understanding how ideas, news, and opinions spread across networks, influence public perspectives, and affect people's decision-making. By studying these processes, we can reveal patterns of influence, understand who is influenced, and calculate how quickly information reaches different parts of the network. This knowledge is key to addressing challenges such as preventing an increase in news of misinformation, optimizing the spread of accurate information, and improving communication strategies.

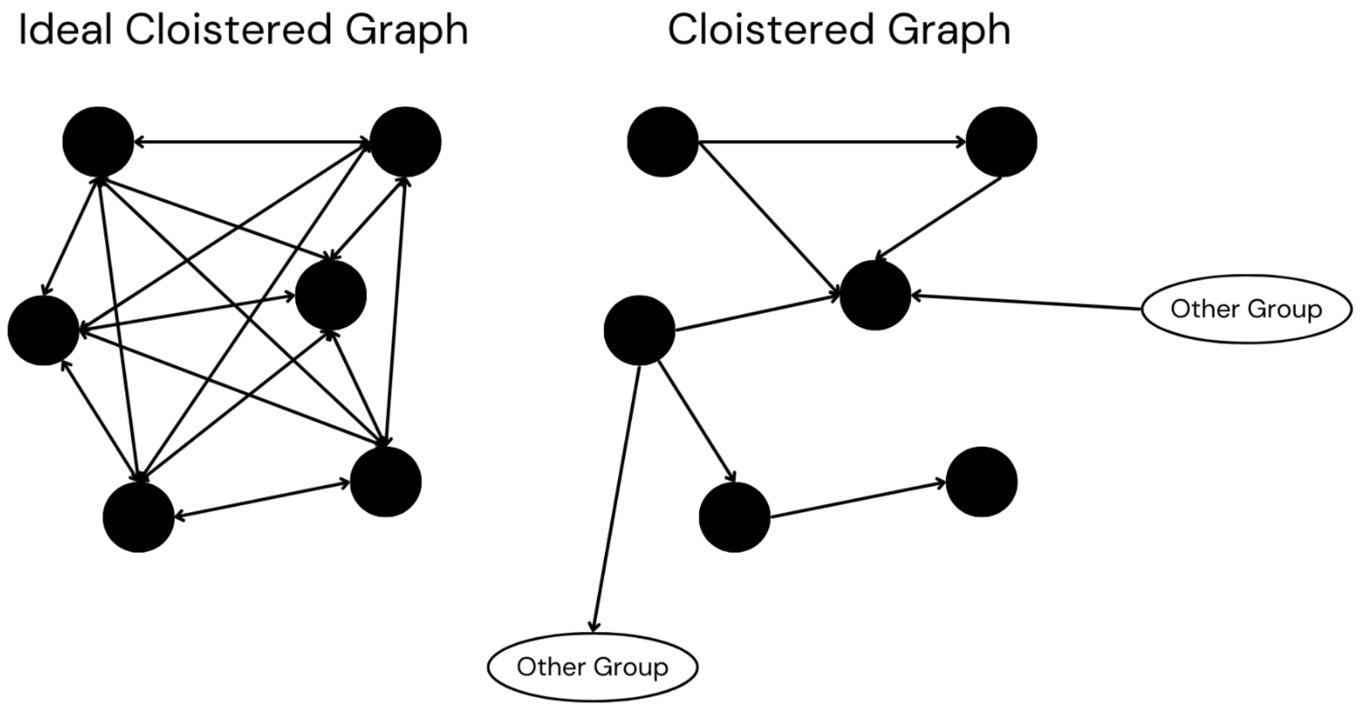
Analyzing communities within social media networks plays a critical role in achieving said goals. Communities, which represent smaller subsets of the network, allow us to focus on specific areas of the network. This localized analysis is valuable for understanding the behavior and dynamics of clustered communities and their impact on the network. Community analysis helps identify significant nodes—such as influencers and gatekeepers—that control the flow of information. Studying communities also makes network analysis more manageable. By focusing on smaller areas of a network, we can gain understanding that apply to the entire network without being overwhelmed by its complexity.

2.2 Description

In this project, the primary focus was to analyze the flow of information on X by exploring how different graph structures influence the network. Three distinct graph types were studied: cloistered graph, tree graph, and line branch graph. Each structure represents a different analysis and understanding of how information flows through a network. I will provide a summary of each type of subgraph

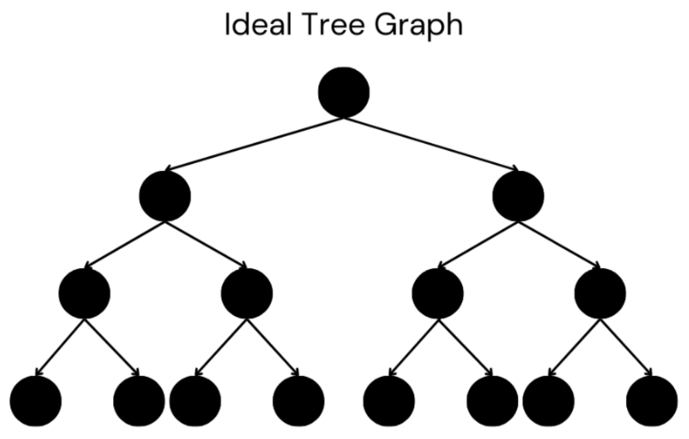
Cloistered Graph:

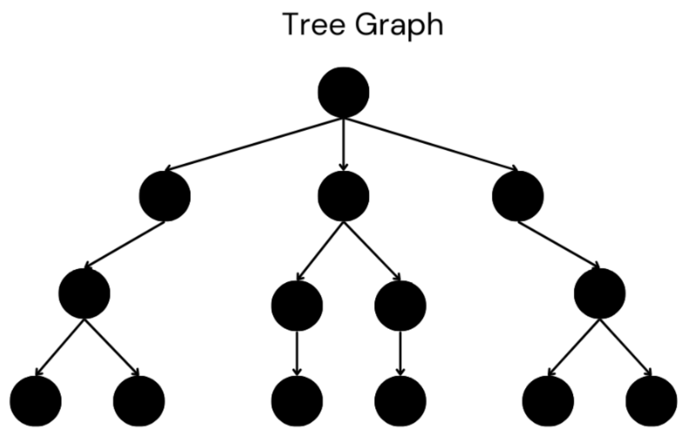
The cloistered graph simulates tightly knit communities, also referred to as groups, within a network, where each user has many connections with others in the same community. An ideal cloistered graph would be a disconnected complete graph. However, since such groups are difficult to see in a real-world setting, a more practical definition for a cloistered graph would be a group that has few to none edges connected to other groups within a network. This structure would be analogous to closed groups on X, such as private discussions or niche communities where information tends to flow among active users.



Tree Graph:

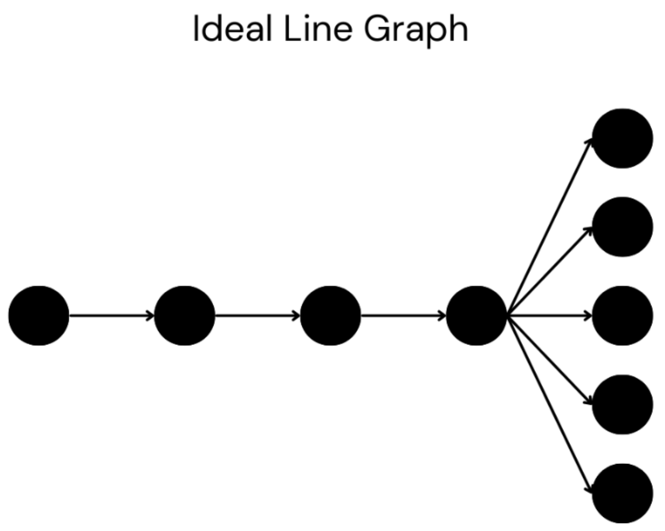
The tree graph is a hierarchical data structure where an ideal tree graph is a noncyclic graph that develops from a single node (root node) and each node splits into 2 child nodes. This model reflects scenarios on X where a central figure (an influencer or organization) posts information to their followers, which then spreads through subsequent layers of users.

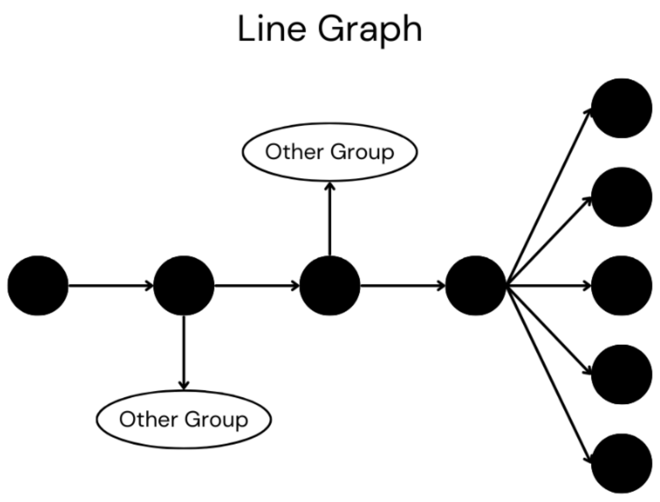




Line Branch Graph:

The line branch graph is a line of nodes where the final node in the line is connected to several nodes. An ideal line branch graph would be a group where the line of nodes only has connections to the node before and after itself, and the final node in the line has several connections to other nodes. This sequence of nodes would be a group where information passes sequentially from one user to the next, representing the retweet chain seen in niche topics or during initial phases of information propagation.





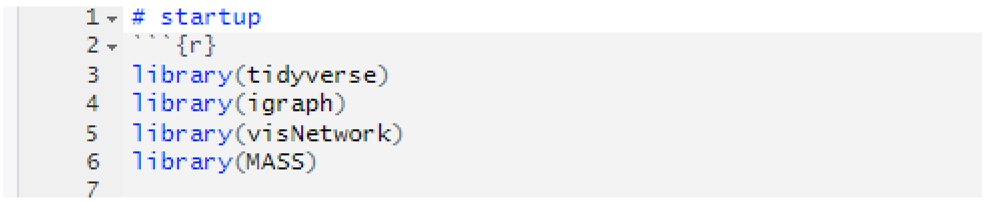
Analyzing these three graph types revealed how certain community configurations affect the speed, reach, and flow of information in networks on X. These findings contribute to understanding the properties of real-world social networks and provide a baseline for further exploration of how information flows.

2.3 What was built

To implement the analysis of information flow on X, the project focused on graphing the network and analyzing the network through the lens of three distinct graph structures: a cloistered graph, tree graph, and line branch graph. These graph types were chosen to represent different community configurations commonly observed in networks.

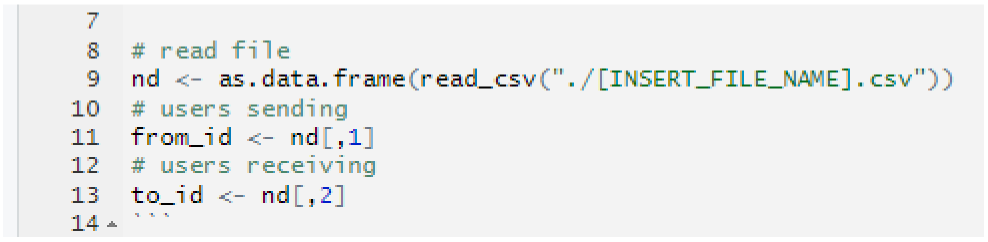
The implementation was carried out using R, leveraging its robust libraries for graph representation and analysis. The program written in R is structured to organize the data, graph the network, and analyze the communities in networks by categorizing them into subgraphs.

The program starts with loading the necessary packages for the program.

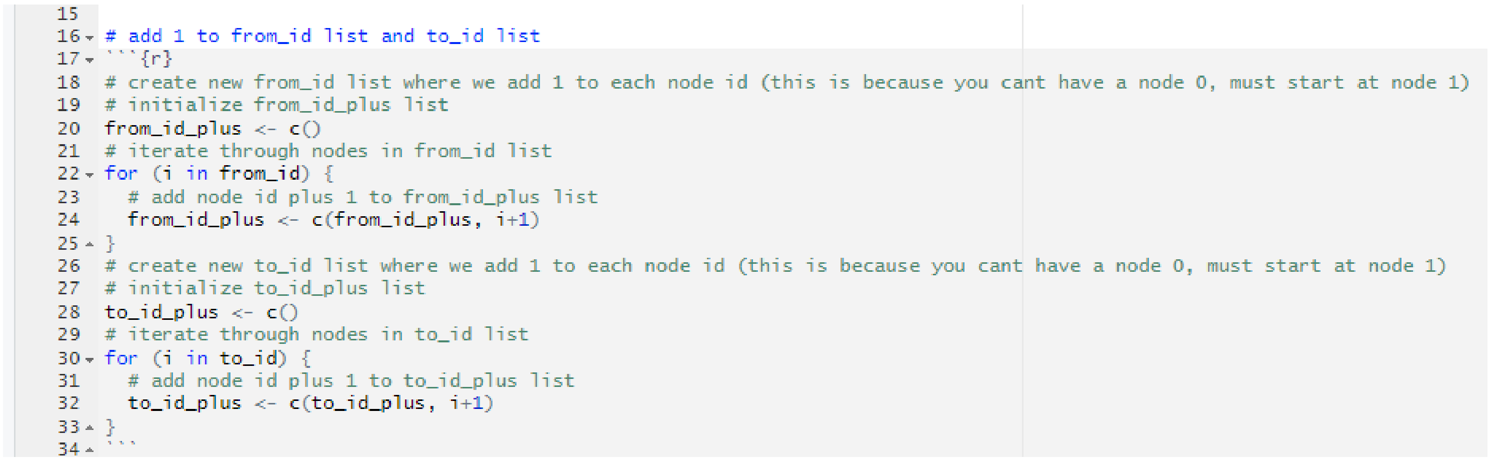


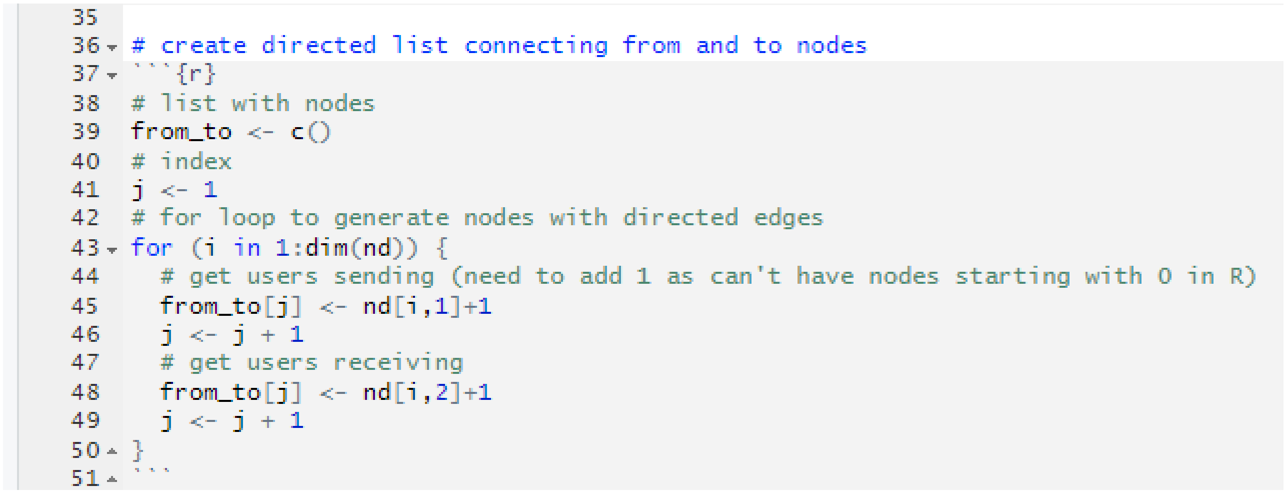
~~~
# startup
```{r}
library(tidyverse)
library(igraph)
library(visNetwork)
library(MASS)
~~~

The program then reads from a CSV file that contains the data representing the network. The following lines in the program then organize data by splitting the data into two lists: nodes that point to other nodes and nodes that are pointed to.

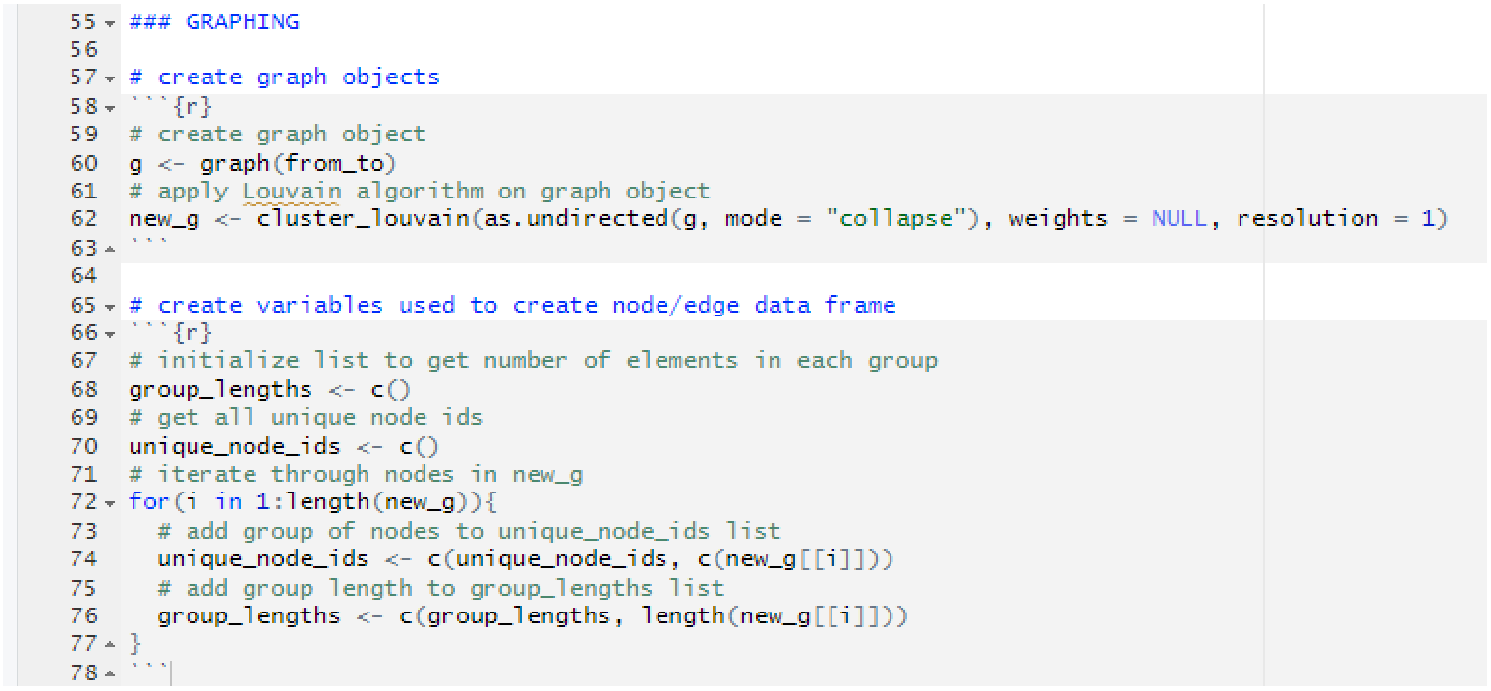


```

# read file
nd <- as.data.frame(read_csv("./[INSERT_FILE_NAME].csv"))
# users sending
from_id <- nd[,1]
# users receiving
to_id <- nd[,2]
```
```

After organizing the data, the program then adds one to every node id in the nodes pointing to other nodes and nodes that are pointed to lists. This is due to R not being able to graph nodes that start with 0 or search through lists using an index of 0.
```

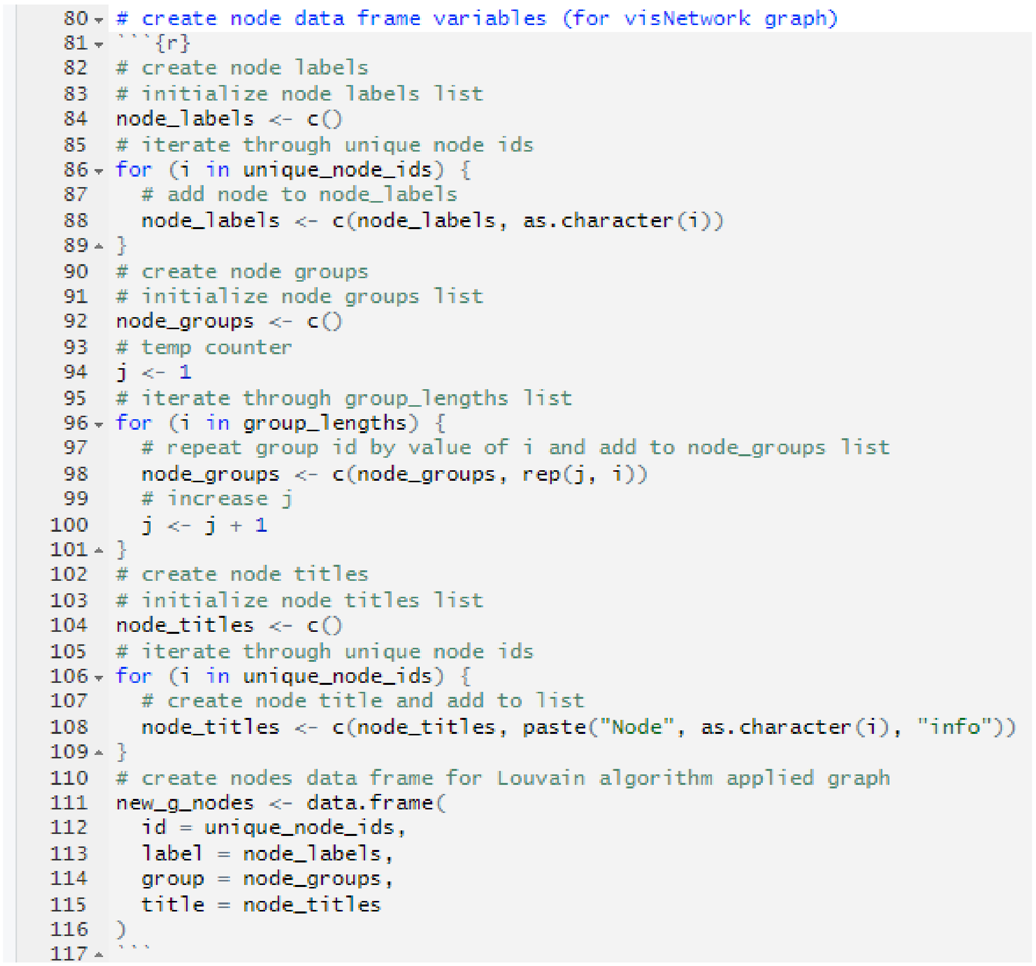


~~~
# add 1 to from_id list and to_id list
```{r}
# create new from_id list where we add 1 to each node id (this is because you cant have a node 0, must start at node 1)
# initialize from_id_plus list
from_id_plus <- c()
# iterate through nodes in from_id list
for (i in from_id) {
  # add node id plus 1 to from_id_plus list
  from_id_plus <- c(from_id_plus, i+1)
}
# create new to_id list where we add 1 to each node id (this is because you cant have a node 0, must start at node 1)
# initialize to_id_plus list
to_id_plus <- c()
# iterate through nodes in to_id list
for (i in to_id) {
  # add node id plus 1 to to_id_plus list
  to_id_plus <- c(to_id_plus, i+1)
}
~~~
```

The program then creates a directed list that connects the nodes using the two lists created above.



```
# create directed list connecting from and to nodes
```{r}
# list with nodes
from_to <- c()
# index
j <- 1
# for loop to generate nodes with directed edges
for (i in 1:dim(nd)) {
  # get users sending (need to add 1 as can't have nodes starting with 0 in R)
  from_to[j] <- nd[i,1]+1
  j <- j + 1
  # get users receiving
  from_to[j] <- nd[i,2]+1
  j <- j + 1
}
```
```

A graph object is created (from the *igraph* package) using the directed list. The program then applies the Louvain algorithm to identify communities in the network. This algorithm is a built-in function (from the *igraph* package) that detects communities in a network by optimizing a parameter called modularity, which quantifies how well a
```

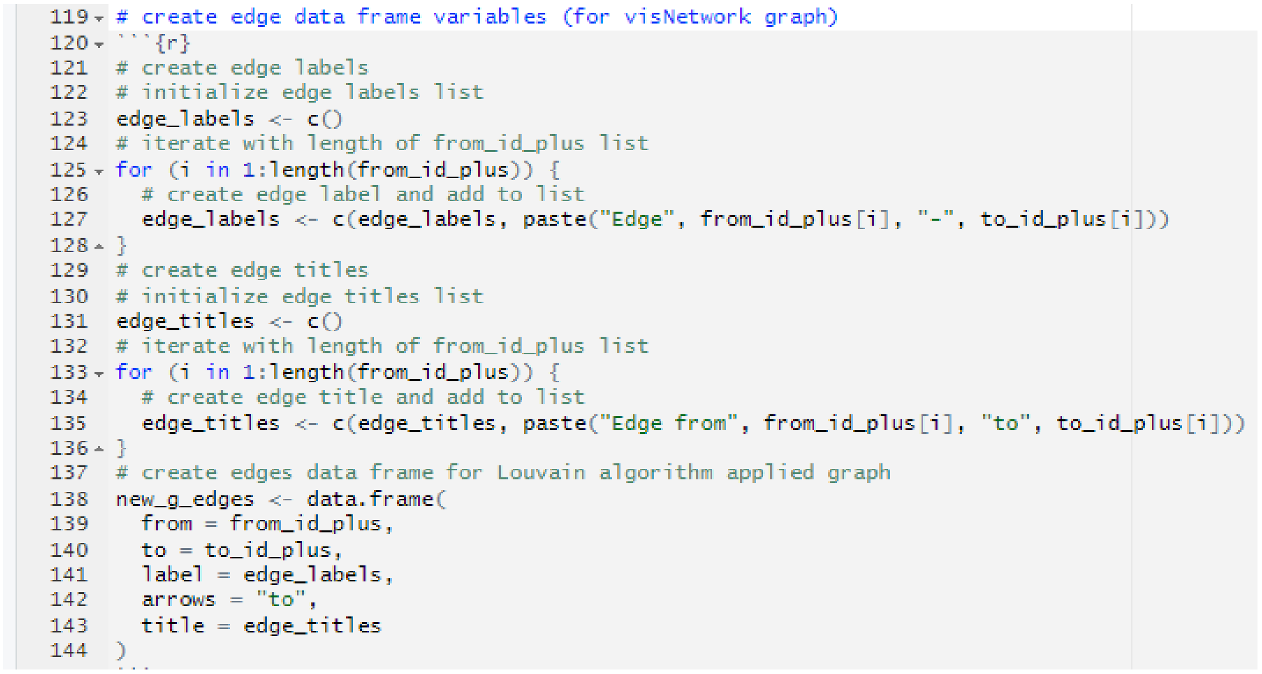
network can be divided into distinct groups. It separates the network into groups of nodes that are more densely connected to each other than to other nodes. Since my objective was to sort communities into subgraphs, the Louvain algorithm was an excellent choice. Its efficiency on large datasets made it perfect for analyzing my data, which included networks with over 1 million connections. The following lines apply the Louvain algorithm to the graph object and manipulate the data to prepare a visual representation of the network.


~~~
### GRAPHING

# create graph objects
```{r}
# create graph object
g <- graph(from_to)
# apply Louvain algorithm on graph object
new_g <- cluster_louvain(as.undirected(g, mode = "collapse"), weights = NULL, resolution = 1)
~~~

create variables used to create node/edge data frame
```{r}
# initialize list to get number of elements in each group
group_lengths <- c()
# get all unique node ids
unique_node_ids <- c()
# iterate through nodes in new_g
for(i in 1:length(new_g)){
  # add group of nodes to unique_node_ids list
  unique_node_ids <- c(unique_node_ids, c(new_g[[i]]))
  # add group length to group_lengths list
  group_lengths <- c(group_lengths, length(new_g[[i]]))
}
```
```

The program creates two data frames that contain information about the nodes and edges in the network. These two data frames are used to assist with graphing the network.
```

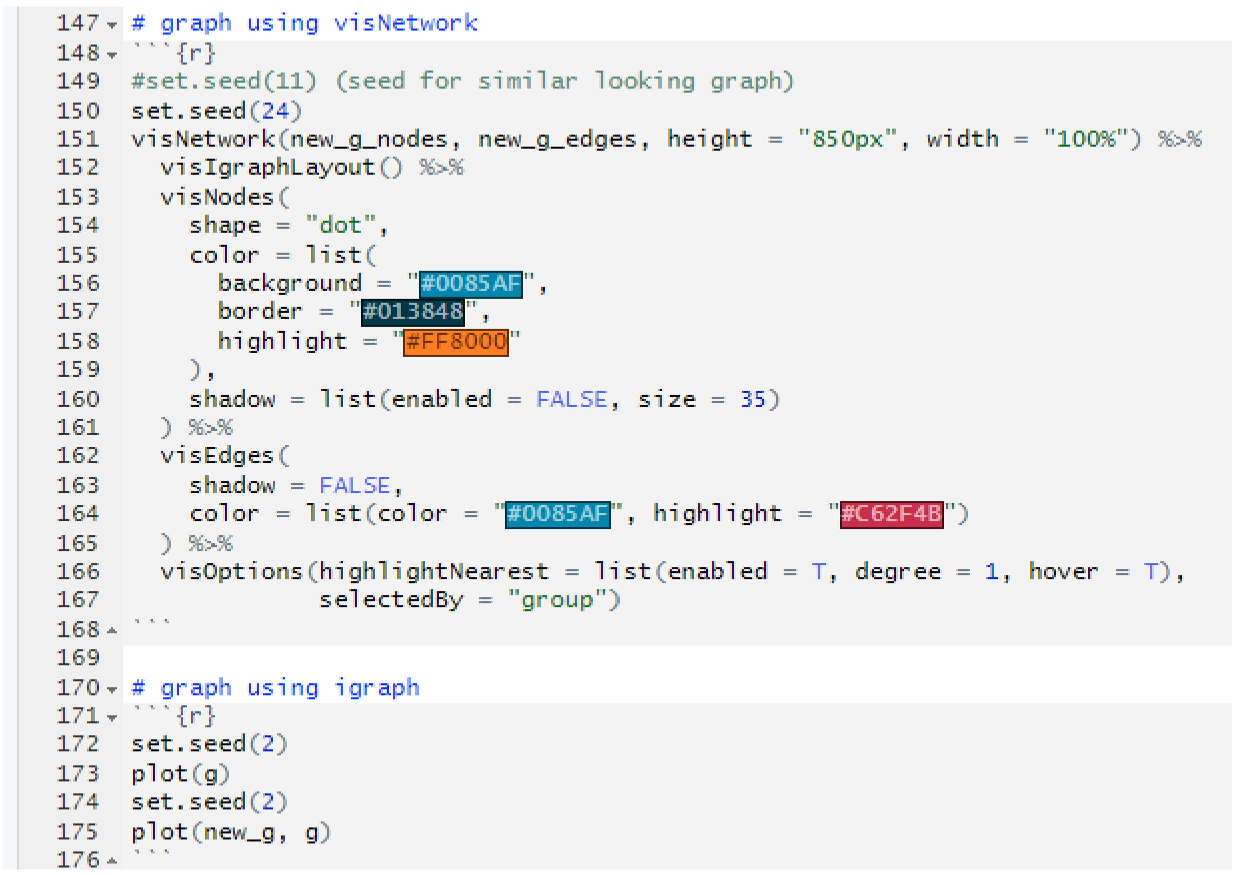


~~~
# create node data frame variables (for visNetwork graph)
```{r}
# create node labels
# initialize node labels list
node_labels <- c()
# iterate through unique node ids
for (i in unique_node_ids) {
  # add node to node_labels
  node_labels <- c(node_labels, as.character(i))
}
# create node groups
# initialize node groups list
node_groups <- c()
# temp counter
j <- 1
# iterate through group_lengths list
for (i in group_lengths) {
  # repeat group id by value of i and add to node_groups list
  node_groups <- c(node_groups, rep(j, i))
  # increase j
  j <- j + 1
}
# create node titles
# initialize node titles list
node_titles <- c()
# iterate through unique node ids
for (i in unique_node_ids) {
  # create node title and add to list
  node_titles <- c(node_titles, paste("Node", as.character(i), "info"))
}
# create nodes data frame for Louvain algorithm applied graph
new_g_nodes <- data.frame(
  id = unique_node_ids,
  label = node_labels,
  group = node_groups,
  title = node_titles
)
```
~~~



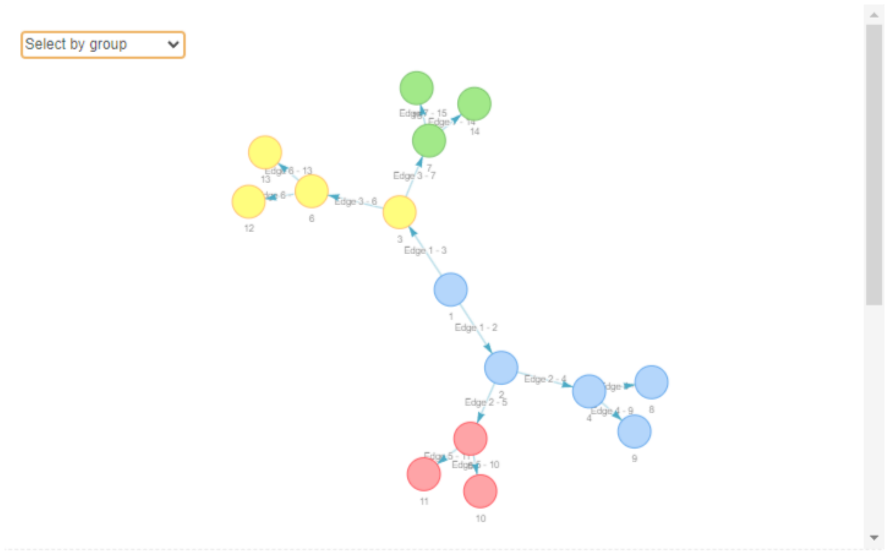
~~~
# create edge data frame variables (for visNetwork graph)
```{r}
# create edge labels
# initialize edge labels list
edge_labels <- c()
# iterate with length of from_id_plus list
for (i in 1:length(from_id_plus)) {
  # create edge label and add to list
  edge_labels <- c(edge_labels, paste("Edge", from_id_plus[i], "-", to_id_plus[i]))
}
# create edge titles
# initialize edge titles list
edge_titles <- c()
# iterate with length of from_id_plus list
for (i in 1:length(from_id_plus)) {
  # create edge title and add to list
  edge_titles <- c(edge_titles, paste("Edge from", from_id_plus[i], "to", to_id_plus[i]))
}
# create edges data frame for Louvain algorithm applied graph
new_g_edges <- data.frame(
  from = from_id_plus,
  to = to_id_plus,
  label = edge_labels,
  arrows = "to",
  title = edge_titles
)
~~~

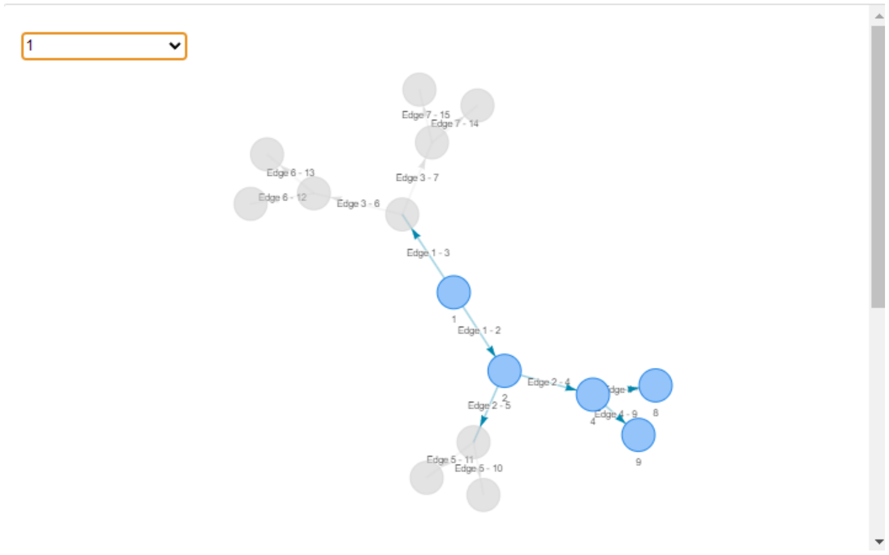
After preparing the data, the program graphs the network using *visNetwork* and *igraph*. While both graphs represent the same network, they offer slightly different perspectives and insights. *visNetwork* provides an interactive representation of the network as well as an option to export the representation as an HTML file. *igraph* provides a static image of the network and focuses on network analysis.

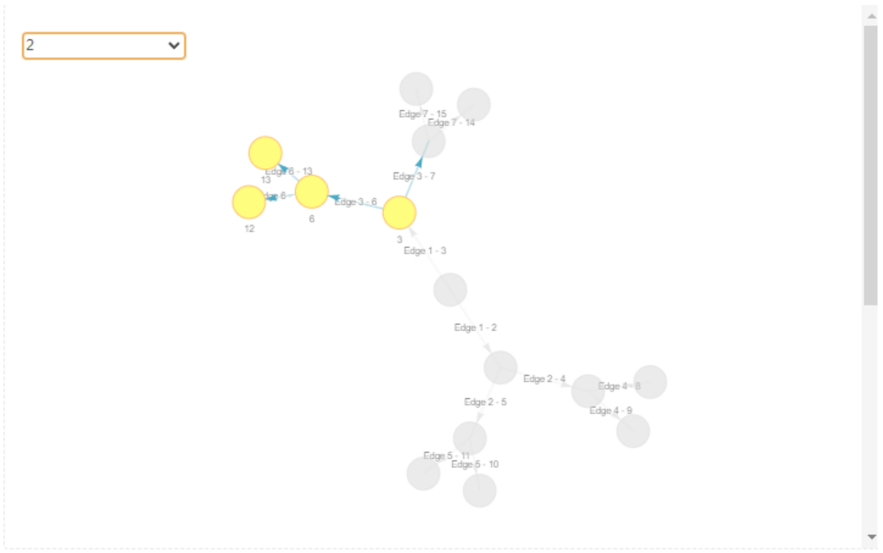


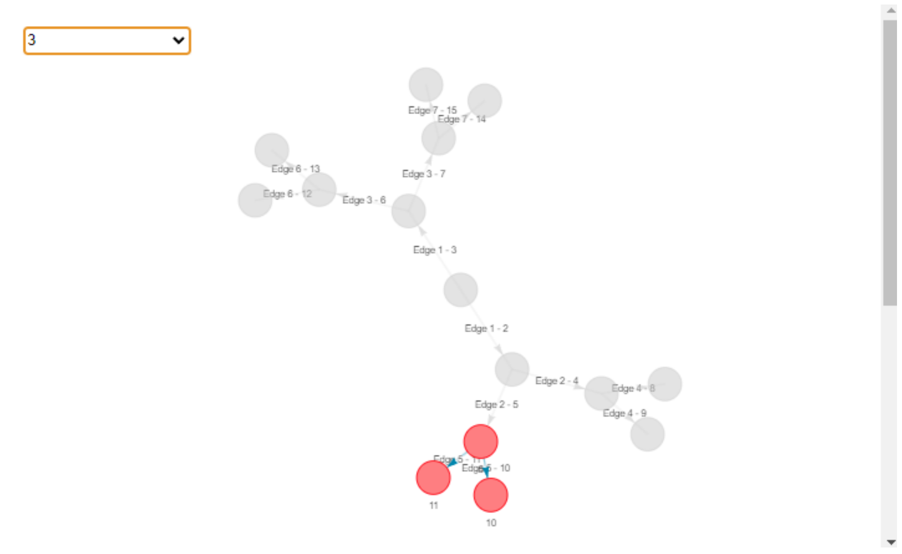
~~~
# graph using visNetwork
```{r}
#set.seed(11) (seed for similar looking graph)
set.seed(24)
visNetwork(new_g_nodes, new_g_edges, height = "850px", width = "100%") %>%
  visIgraphLayout() %>%
  visNodes(
    shape = "dot",
    color = list(
      background = "#0085AF",
      border = "#013848",
      highlight = "#FF8000"
    ),
    shadow = list(enabled = FALSE, size = 35)
  ) %>%
  visEdges(
    shadow = FALSE,
    color = list(color = "#0085AF", highlight = "#C62F4B")
  ) %>%
  visOptions(highlightNearest = list(enabled = T, degree = 1, hover = T),
             selectedBy = "group")
~~~

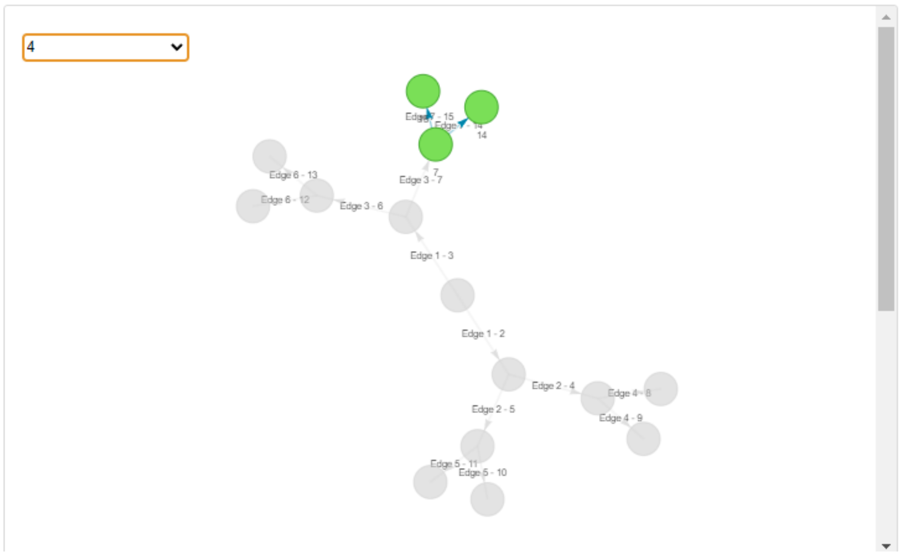
graph using igraph
```{r}
set.seed(2)
plot(g)
set.seed(2)
plot(new_g, g)
```
```

Sample graph using *visNetwork*:
```

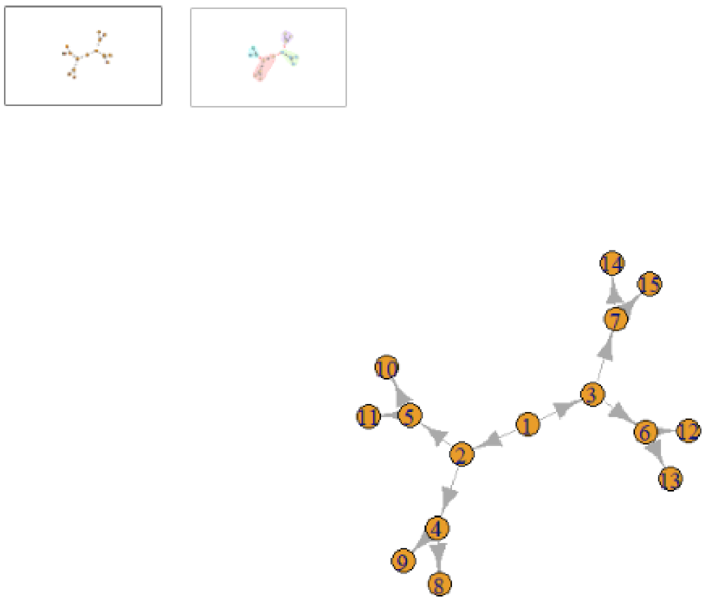

Select by group
Edge 7 - 15
Edge 7 - 14
Edge 3 - 7
Edge 6 - 13
Edge 6 - 12
Edge 3 - 6
Edge 1 - 3
Edge 1 - 2
Edge 2 - 4
Edge 4 - 8
Edge 4 - 9
Edge 2 - 5
Edge 5 - 11
Edge 5 - 10
1
2
3
4
6
7
8
9
10
11
12
13
14

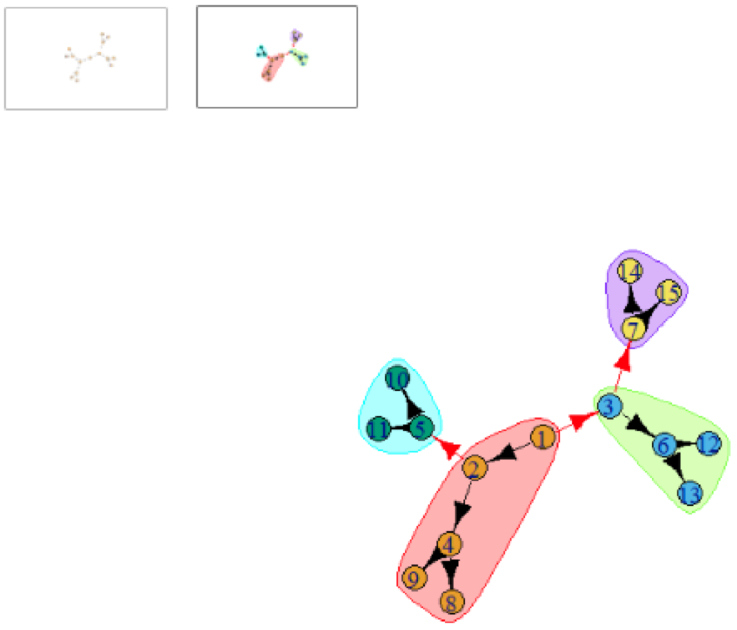

1
Edge 7 - 15
Edge 7 - 14
Edge 3 - 7
Edge 6 - 13
Edge 6 - 12
Edge 3 - 6
Edge 1 - 3
Edge 1 - 2
Edge 2 - 4
Edge 4 - 8
Edge 4 - 9
Edge 2 - 5
Edge 5 - 11
Edge 5 - 10
1
2
4
8
9


2
Edge 7 - 15
Edge 7 - 14
Edge 3 - 7
Edge 6 - 13
Edge 6 - 12
Edge 3 - 6
Edge 1 - 3
Edge 1 - 2
Edge 2 - 4
Edge 4 - 8
Edge 4 - 9
Edge 2 - 5
Edge 5 - 11
Edge 5 - 10
3
6
12
13


3
Edge 7 - 15
Edge 7 - 14
Edge 6 - 13
Edge 3 - 7
Edge 6 - 12
Edge 3 - 6
Edge 1 - 3
Edge 1 - 2
Edge 2 - 4
Edge 4 - 8
Edge 2 - 5
Edge 4 - 9
Edge 5 - 11
Edge 5 - 10
11
10


4
Edge 7 - 15
Edge 7 - 14
14
7
Edge 6 - 13
Edge 3 - 7
Edge 6 - 12
Edge 3 - 6
Edge 1 - 3
Edge 1 - 2
Edge 2 - 4
Edge 4 - 8
Edge 2 - 5
Edge 4 - 9
Edge 5 - 11
Edge 5 - 10

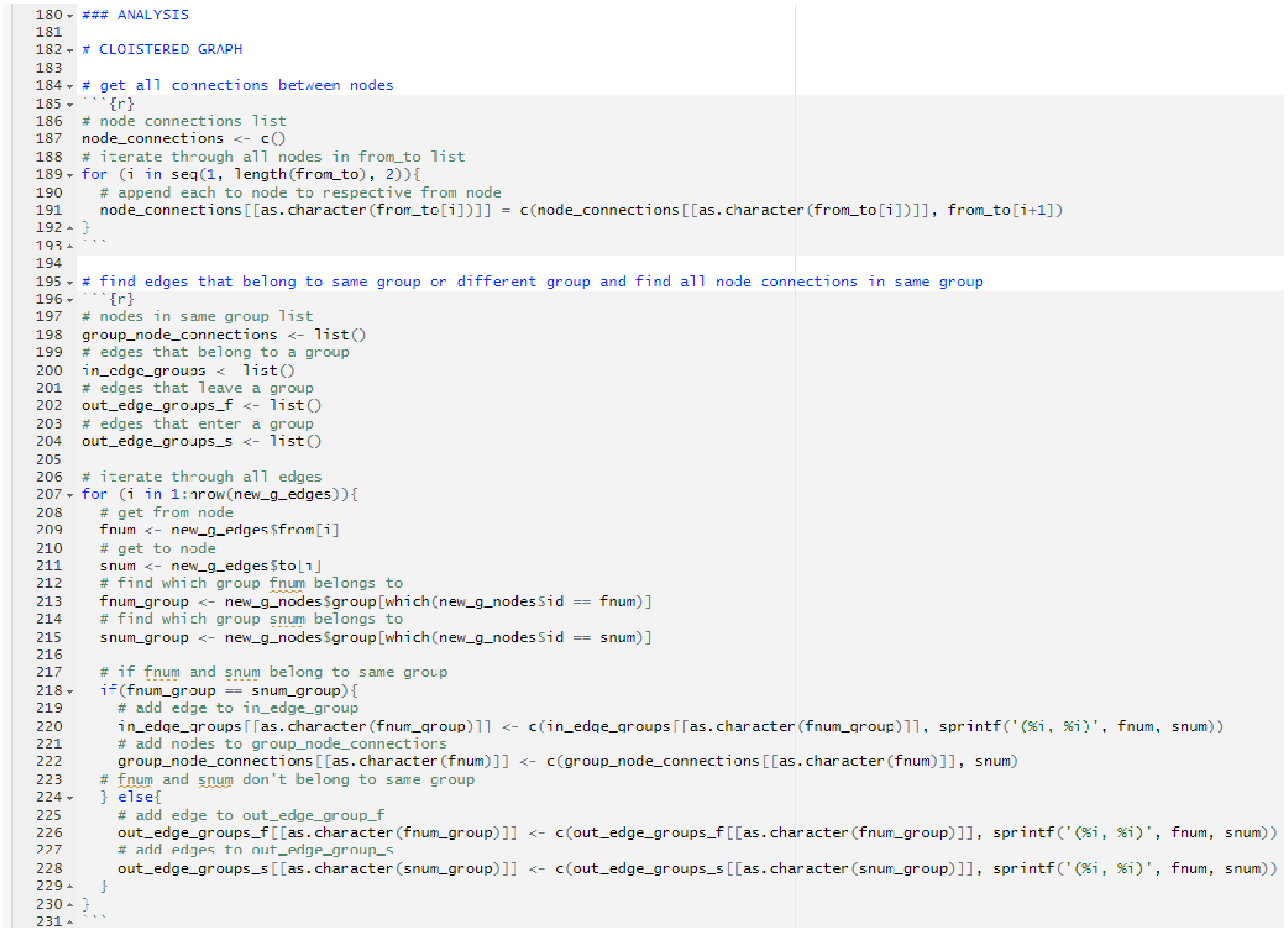




After dividing the network into communities and graphing it, the program then analyzes the communities and applies the subgraph analysis.

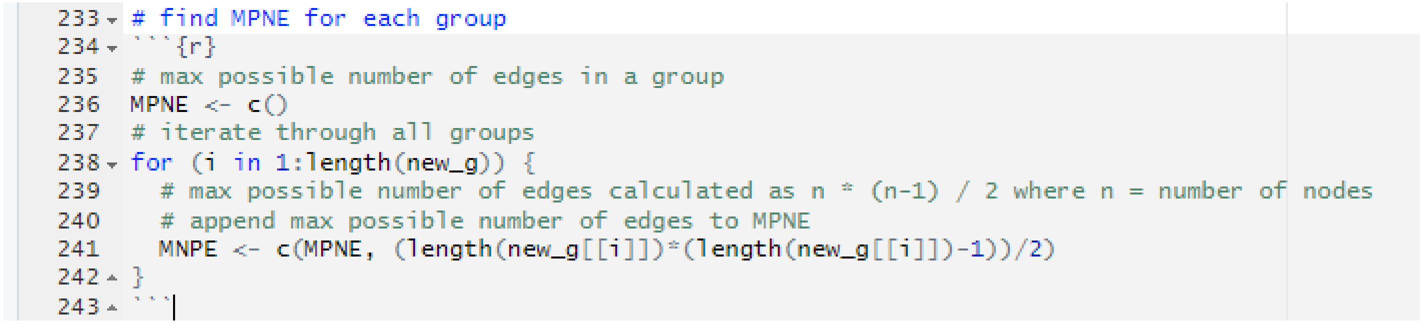
To identify all communities that resemble cloistered graphs, the program compares the number of edges in the group to the maximum number of possible edges in the group.

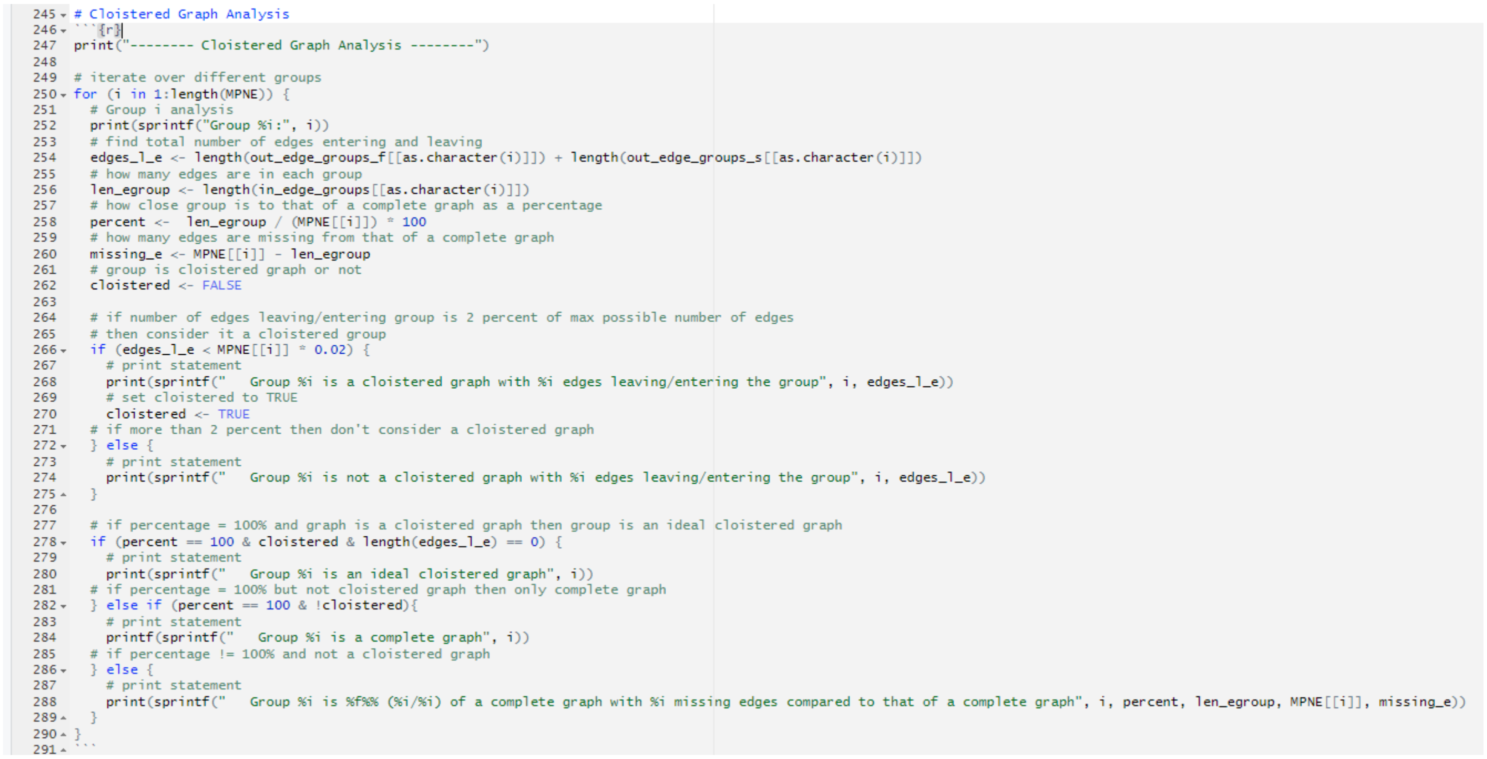
I created a few helper functions to assist with the analysis. The first helper function finds every connection between nodes in the data. The second helper function finds every edge that connects to a different community and every node that belong to the same community. The third helper function finds the MPNE (maximum possible number of edges) of the group.



~~~
### ANALYSIS

# CLOISTERED GRAPH

# get all connections between nodes
```{r}
# node connections list
node_connections <- c()
# iterate through all nodes in from_to list
for (i in seq(1, length(from_to), 2)){
  # append each to node to respective from node
  node_connections[[as.character(from_to[i])]] = c(node_connections[[as.character(from_to[i])]], from_to[i+1])
}
~~~

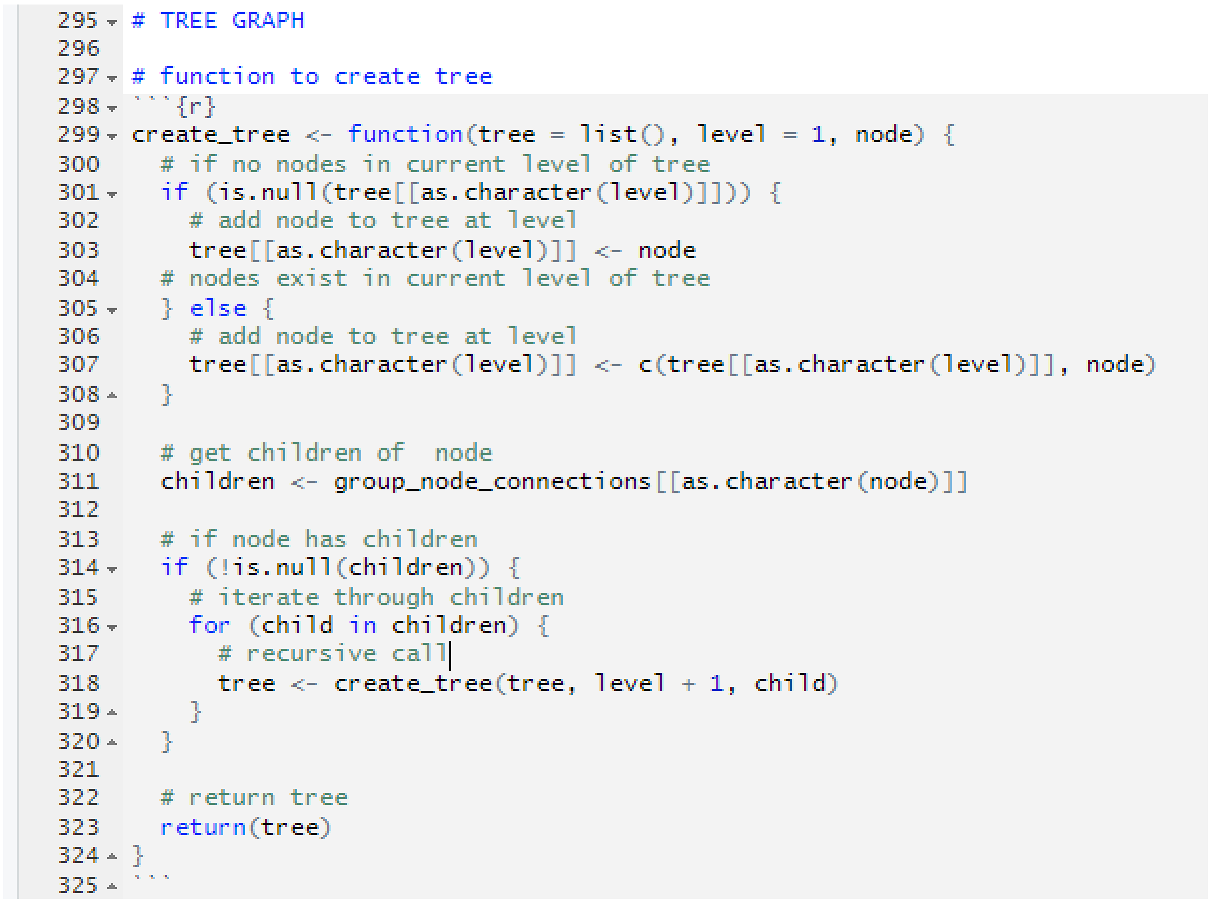
# find edges that belong to same group or different group and find all node connections in same group
```{r}
# nodes in same group list
group_node_connections <- list()
# edges that belong to a group
in_edge_groups <- list()
# edges that leave a group
out_edge_groups_f <- list()
# edges that enter a group
out_edge_groups_s <- list()

# iterate through all edges
for (i in 1:nrow(new_g_edges)){
  # get from node
  fnum <- new_g_edges$from[i]
  # get to node
  snum <- new_g_edges$to[i]
  # find which group fnum belongs to
  fnum_group <- new_g_nodes$group[which(new_g_nodes$id == fnum)]
  # find which group snum belongs to
  snum_group <- new_g_nodes$group[which(new_g_nodes$id == snum)]

  # if fnum and snum belong to same group
  if(fnum_group == snum_group){
    # add edge to in_edge_group
    in_edge_groups[[as.character(fnum_group)]] <- c(in_edge_groups[[as.character(fnum_group)]], sprintf('(%i, %i)', fnum, snum))
    # add nodes to group_node_connections
    group_node_connections[[as.character(fnum)]] <- c(group_node_connections[[as.character(fnum)]], snum)
  # fnum and snum don't belong to same group
  } else{
    # add edge to out_edge_group_f
    out_edge_groups_f[[as.character(fnum_group)]] <- c(out_edge_groups_f[[as.character(fnum_group)]], sprintf('(%i, %i)', fnum, snum))
    # add edges to out_edge_group_s
    out_edge_groups_s[[as.character(snum_group)]] <- c(out_edge_groups_s[[as.character(snum_group)]], sprintf('(%i, %i)', fnum, snum))
  }
}
```
```
```



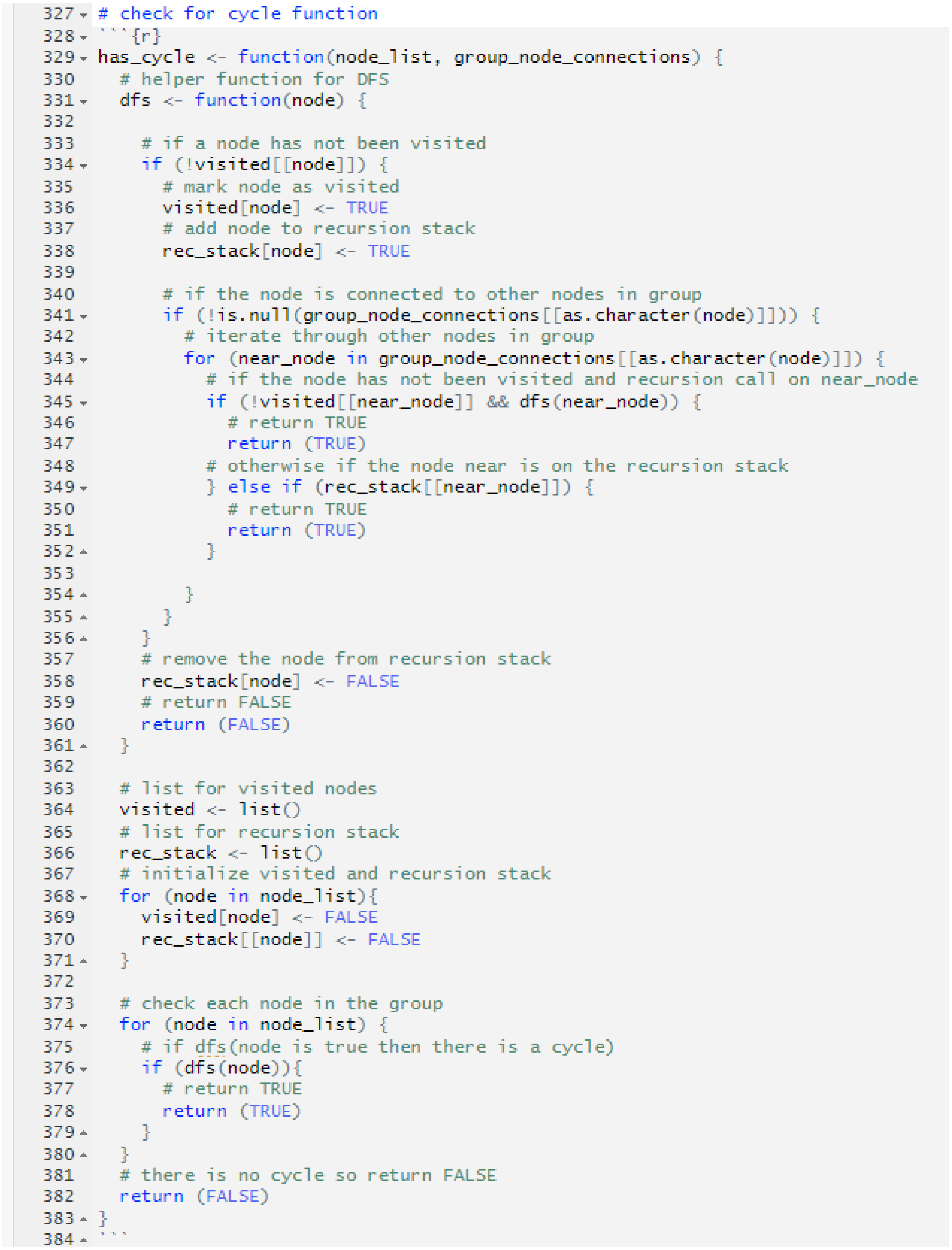
~~~
# find MPNE for each group
```{r}
# max possible number of edges in a group
MPNE <- c()
# iterate through all groups
for (i in 1:length(new_g)) {
  # max possible number of edges calculated as n * (n-1) / 2 where n = number of nodes
  # append max possible number of edges to MPNE
  MNPE <- c(MPNE, (length(new_g[[i]])*(length(new_g[[i]])-1))/2)
}
~~~
```

Using these three helper functions, the program then performs the cloistered graph identification analysis on every group within the network.



```
# Cloistered Graph Analysis
```{r}
print("-------- Cloistered Graph Analysis --------")

# iterate over different groups
for (i in 1:length(MPNE)) {
  # Group i analysis
  print(sprintf("Group %i:", i))
  # find total number of edges entering and leaving
  edges_l_e <- length(out_edge_groups_f[[as.character(i)]]) + length(out_edge_groups_s[[as.character(i)]])
  # how many edges are in each group
  len_egroup <- length(in_edge_groups[[as.character(i)]])
  # how close group is to that of a complete graph as a percentage
  percent <-  len_egroup / (MPNE[[i]]) * 100
  # how many edges are missing from that of a complete graph
  missing_e <- MPNE[[i]] - len_egroup
  # group is cloistered graph or not
  cloistered <- FALSE

  # if number of edges leaving/entering group is 2 percent of max possible number of edges
  # then consider it a cloistered group
  if (edges_l_e < MPNE[[i]] * 0.02) {
    # print statement
    print(sprintf("   Group %i is a cloistered graph with %i edges leaving/entering the group", i, edges_l_e))
    # set cloistered to TRUE
    cloistered <- TRUE
  # if more than 2 percent then don't consider a cloistered graph
  } else {
    # print statement
    print(sprintf("   Group %i is not a cloistered graph with %i edges leaving/entering the group", i, edges_l_e))
  }

  # if percentage = 100% and graph is a cloistered graph then group is an ideal cloistered graph
  if (percent == 100 & cloistered & length(edges_l_e) == 0) {
    # print statement
    print(sprintf("   Group %i is an ideal cloistered graph", i))
  # if percentage = 100% but not cloistered graph then only complete graph
  } else if (percent == 100 & !cloistered){
    # print statement
    printf(sprintf("   Group %i is a complete graph", i))
  # if percentage != 100% and not a cloistered graph
  } else {
    # print statement
    print(sprintf("   Group %i is %f%% (%i/%i) of a complete graph with %i missing edges compared to that of a complete graph", i, percent, len_egroup, MPNE[[i]], missing_e))
  }
}
```
```
```

Next, the program determines which communities are tree graphs by comparing every community against the properties of an ideal tree graph.

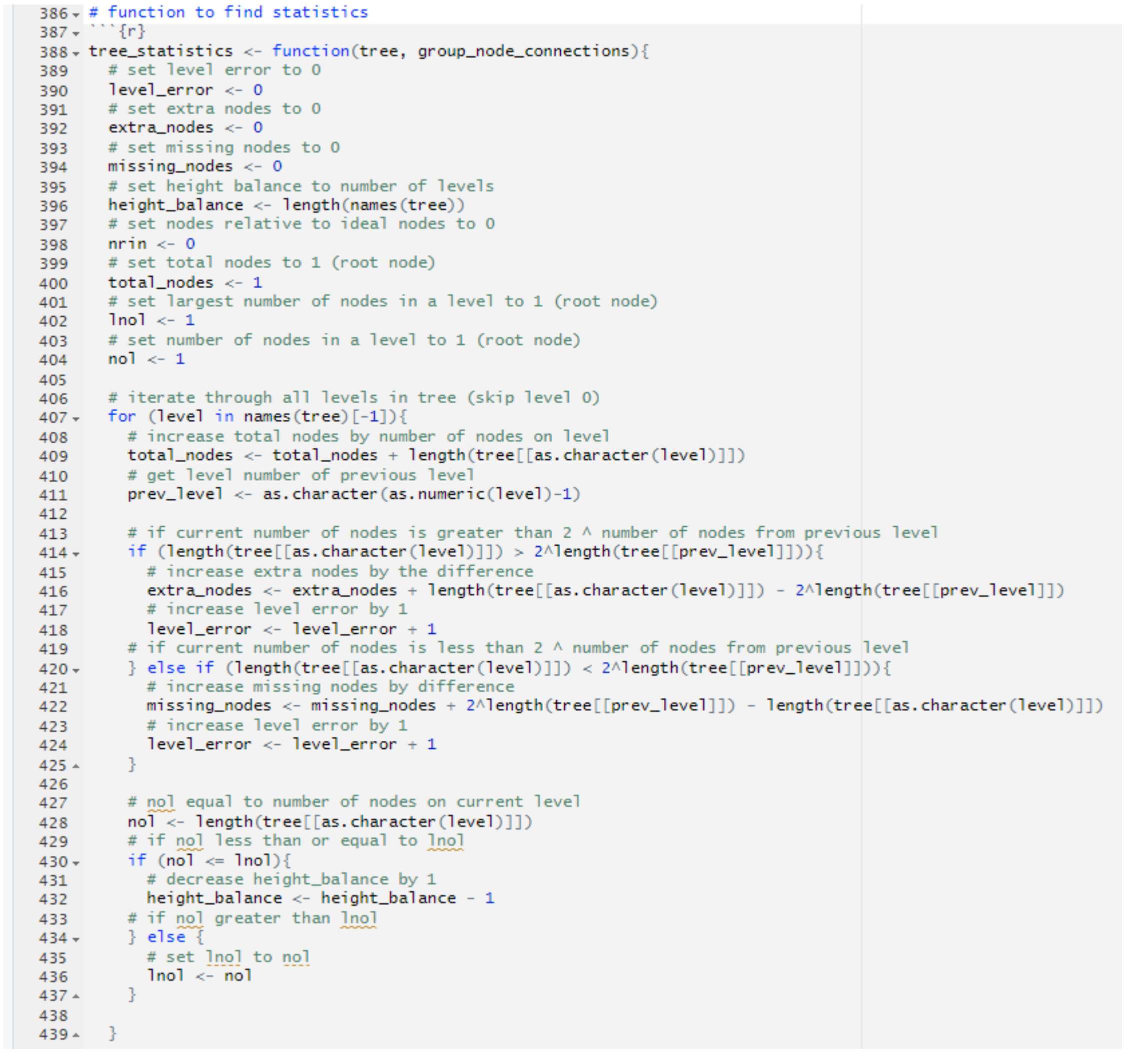
Once again, I created several helper functions to assist with the analysis. The first helper function constructs a tree structure using the nodes within a group. The second helper function checks for cycles within a group. The third helper function examines the group in its tree structure and calculates specific statistics, which will be discussed in Section 3.1. The fourth helper function evaluates these statistics to determine how similar the tree structure is to an ideal tree graph.



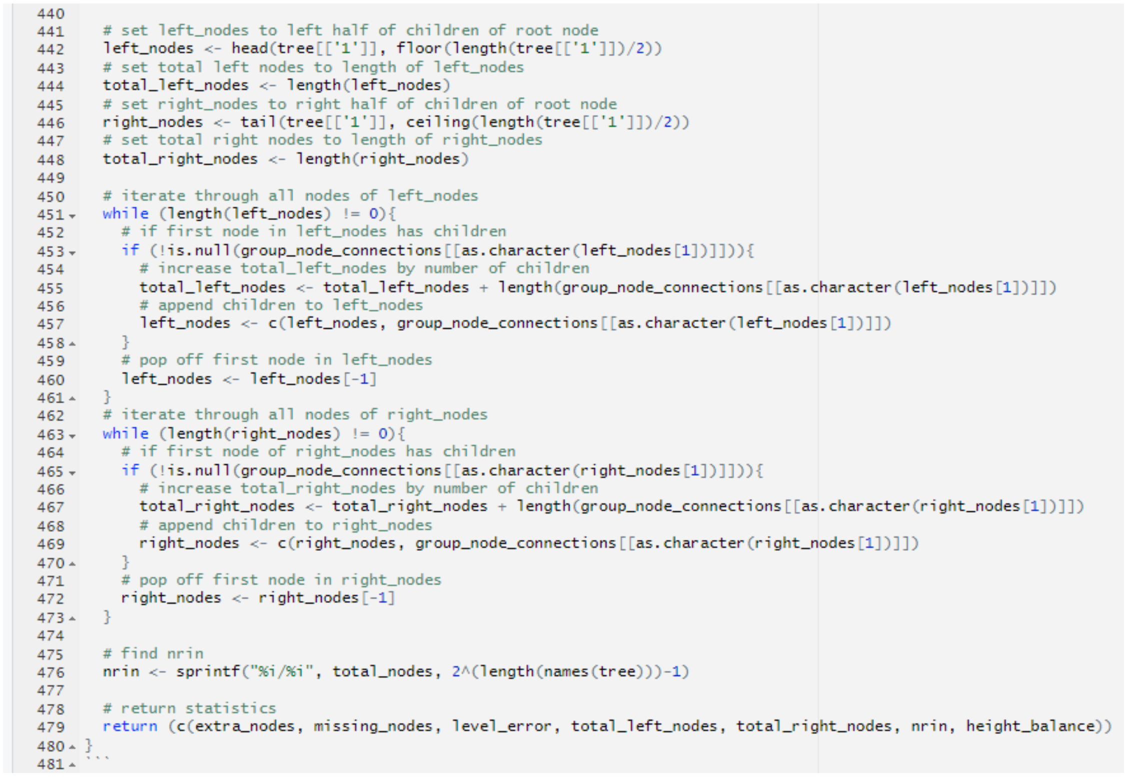
~~~
# TREE GRAPH

# function to create tree
```{r}
create_tree <- function(tree = list(), level = 1, node) {
  # if no nodes in current level of tree
  if (is.null(tree[[as.character(level)]])) {
    # add node to tree at level
    tree[[as.character(level)]] <- node
  # nodes exist in current level of tree
  } else {
    # add node to tree at level
    tree[[as.character(level)]] <- c(tree[[as.character(level)]], node)
  }

  # get children of  node
  children <- group_node_connections[[as.character(node)]]

  # if node has children
  if (!is.null(children)) {
    # iterate through children
    for (child in children) {
      # recursive call
      tree <- create_tree(tree, level + 1, child)
    }
  }

  # return tree
  return(tree)
}
```
~~~



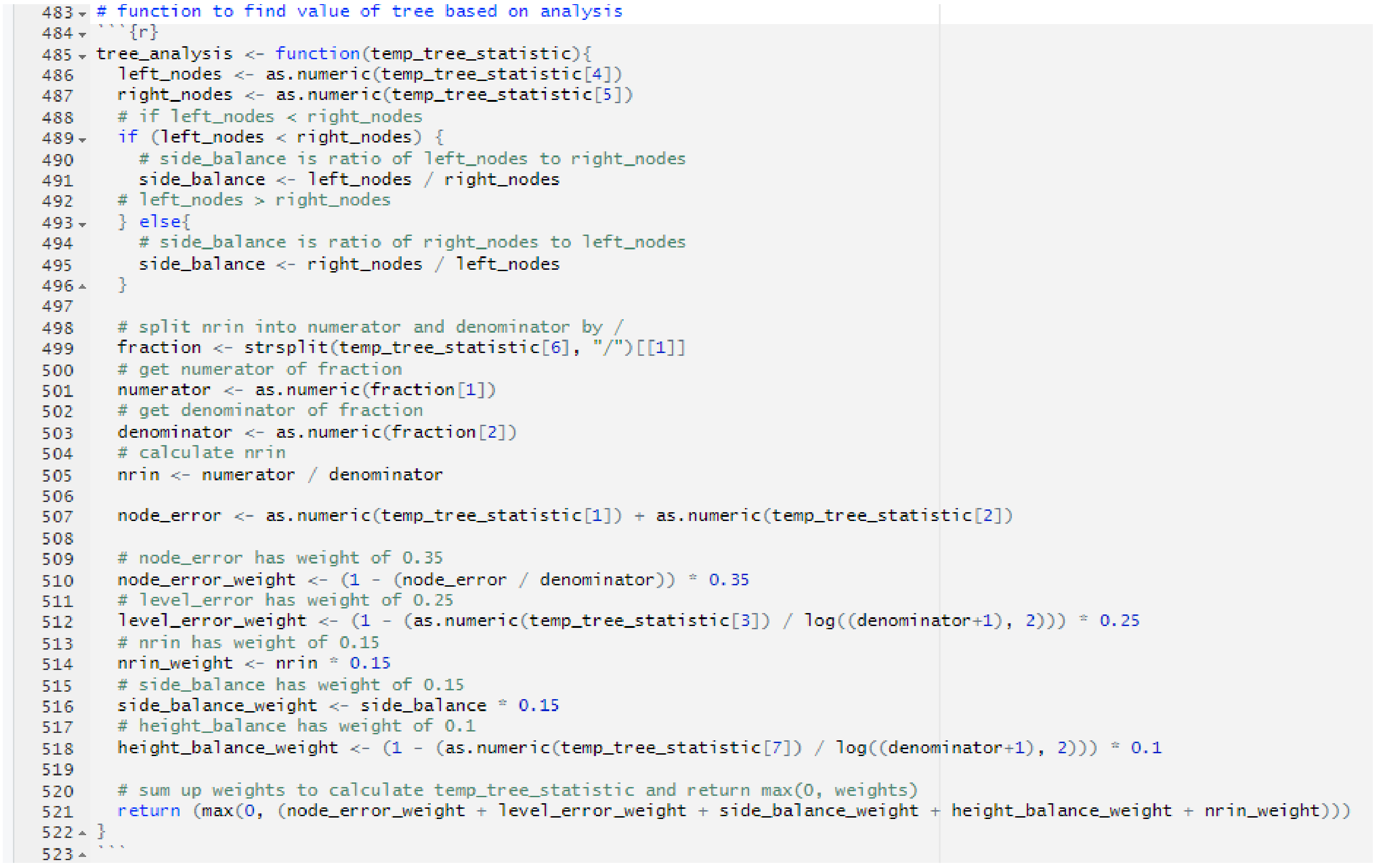
~~~
# check for cycle function
```{r}
has_cycle <- function(node_list, group_node_connections) {
  # helper function for DFS
  dfs <- function(node) {

    # if a node has not been visited
    if (!visited[[node]]) {
      # mark node as visited
      visited[node] <- TRUE
      # add node to recursion stack
      rec_stack[node] <- TRUE

      # if the node is connected to other nodes in group
      if (!is.null(group_node_connections[[as.character(node)]])) {
        # iterate through other nodes in group
        for (near_node in group_node_connections[[as.character(node)]]) {
          # if the node has not been visited and recursion call on near_node
          if (!visited[[near_node]] && dfs(near_node)) {
            # return TRUE
            return (TRUE)
          # otherwise if the node near is on the recursion stack
          } else if (rec_stack[[near_node]]) {
            # return TRUE
            return (TRUE)
          }

        }
      }
    }
    # remove the node from recursion stack
    rec_stack[node] <- FALSE
    # return FALSE
    return (FALSE)
  }

  # list for visited nodes
  visited <- list()
  # list for recursion stack
  rec_stack <- list()
  # initialize visited and recursion stack
  for (node in node_list){
    visited[node] <- FALSE
    rec_stack[[node]] <- FALSE
  }

  # check each node in the group
  for (node in node_list) {
    # if dfs(node is true then there is a cycle)
    if (dfs(node)){
      # return TRUE
      return (TRUE)
    }
  }
  # there is no cycle so return FALSE
  return (FALSE)
}
```
~~~



~~~
# function to find statistics
```{r}
tree_statistics <- function(tree, group_node_connections){
  # set level error to 0
  level_error <- 0
  # set extra nodes to 0
  extra_nodes <- 0
  # set missing nodes to 0
  missing_nodes <- 0
  # set height balance to number of levels
  height_balance <- length(names(tree))
  # set nodes relative to ideal nodes to 0
  nrin <- 0
  # set total nodes to 1 (root node)
  total_nodes <- 1
  # set largest number of nodes in a level to 1 (root node)
  lnol <- 1
  # set number of nodes in a level to 1 (root node)
  nol <- 1

  # iterate through all levels in tree (skip level 0)
  for (level in names(tree)[-1]){
    # increase total nodes by number of nodes on level
    total_nodes <- total_nodes + length(tree[[as.character(level)]])
    # get level number of previous level
    prev_level <- as.character(as.numeric(level)-1)

    # if current number of nodes is greater than 2 ^ number of nodes from previous level
    if (length(tree[[as.character(level)]]) > 2^length(tree[[prev_level]])){
      # increase extra nodes by the difference
      extra_nodes <- extra_nodes + length(tree[[as.character(level)]]) - 2^length(tree[[prev_level]])
      # increase level error by 1
      level_error <- level_error + 1
    # if current number of nodes is less than 2 ^ number of nodes from previous level
    } else if (length(tree[[as.character(level)]]) < 2^length(tree[[prev_level]])){
      # increase missing nodes by difference
      missing_nodes <- missing_nodes + 2^length(tree[[prev_level]]) - length(tree[[as.character(level)]])
      # increase level error by 1
      level_error <- level_error + 1
    }

    # nol equal to number of nodes on current level
    nol <- length(tree[[as.character(level)]])
    # if nol less than or equal to lnol
    if (nol <= lnol){
      # decrease height_balance by 1
      height_balance <- height_balance - 1
    # if nol greater than lnol
    } else {
      # set lnol to nol
      lnol <- nol
    }

  }
~~~

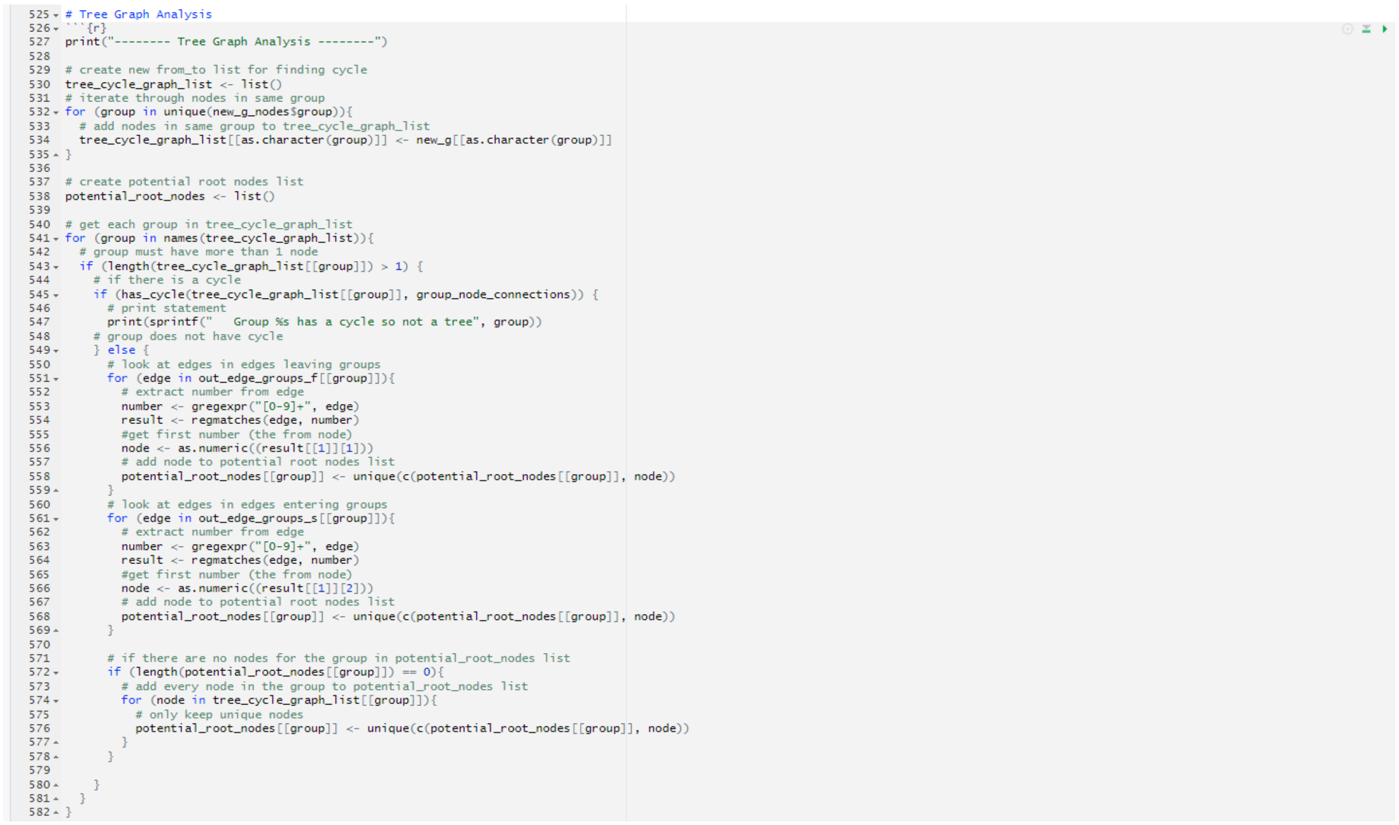


```

  # set left_nodes to left half of children of root node
  left_nodes <- head(tree[['1']], floor(length(tree[['1']])/2))
  # set total left nodes to length of left_nodes
  total_left_nodes <- length(left_nodes)
  # set right_nodes to right half of children of root node
  right_nodes <- tail(tree[['1']], ceiling(length(tree[['1']])/2))
  # set total right nodes to length of right_nodes
  total_right_nodes <- length(right_nodes)

  # iterate through all nodes of left_nodes
  while (length(left_nodes) != 0){
    # if first node in left_nodes has children
    if (!is.null(group_node_connections[[as.character(left_nodes[1])]])){
      # increase total_left_nodes by number of children
      total_left_nodes <- total_left_nodes + length(group_node_connections[[as.character(left_nodes[1])]])
      # append children to left_nodes
      left_nodes <- c(left_nodes, group_node_connections[[as.character(left_nodes[1])]])
    }
    # pop off first node in left_nodes
    left_nodes <- left_nodes[-1]
  }
  # iterate through all nodes of right_nodes
  while (length(right_nodes) != 0){
    # if first node of right_nodes has children
    if (!is.null(group_node_connections[[as.character(right_nodes[1])]])){
      # increase total_right_nodes by number of children
      total_right_nodes <- total_right_nodes + length(group_node_connections[[as.character(right_nodes[1])]])
      # append children to right_nodes
      right_nodes <- c(right_nodes, group_node_connections[[as.character(right_nodes[1])]])
    }
    # pop off first node in right_nodes
    right_nodes <- right_nodes[-1]
  }

  # find nrin
  nrin <- sprintf("%i/%i", total_nodes, 2^(length(names(tree)))-1)

  # return statistics
  return (c(extra_nodes, missing_nodes, level_error, total_left_nodes, total_right_nodes, nrin, height_balance))
}
```
```
```

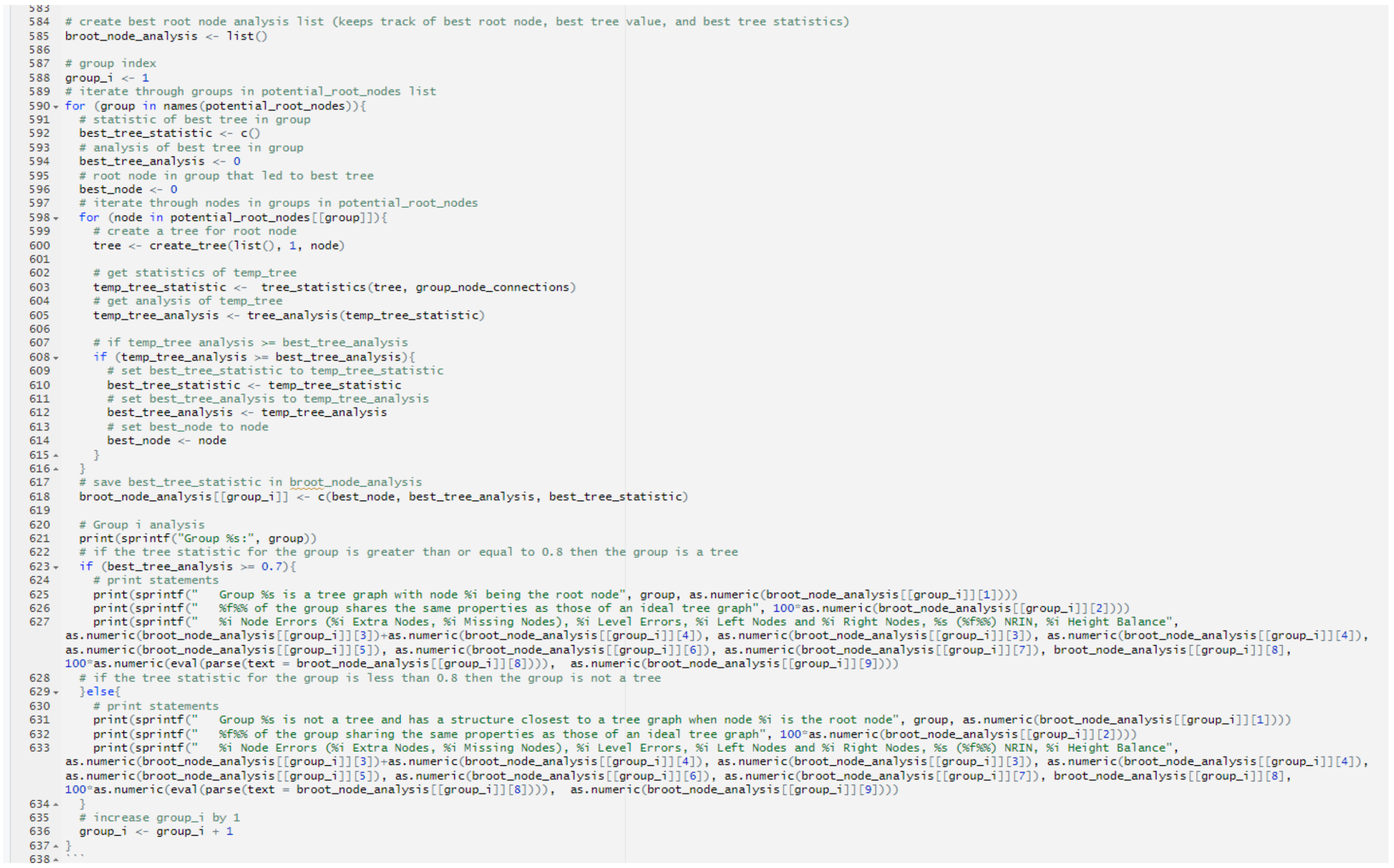


~~~
# function to find value of tree based on analysis
```{r}
tree_analysis <- function(temp_tree_statistic){
  left_nodes <- as.numeric(temp_tree_statistic[4])
  right_nodes <- as.numeric(temp_tree_statistic[5])
  # if left_nodes < right_nodes
  if (left_nodes < right_nodes) {
    # side_balance is ratio of left_nodes to right_nodes
    side_balance <- left_nodes / right_nodes
  # left_nodes > right_nodes
  } else{
    # side_balance is ratio of right_nodes to left_nodes
    side_balance <- right_nodes / left_nodes
  }

  # split nrin into numerator and denominator by /
  fraction <- strsplit(temp_tree_statistic[6], "/")[[1]]
  # get numerator of fraction
  numerator <- as.numeric(fraction[1])
  # get denominator of fraction
  denominator <- as.numeric(fraction[2])
  # calculate nrin
  nrin <- numerator / denominator

  node_error <- as.numeric(temp_tree_statistic[1]) + as.numeric(temp_tree_statistic[2])

  # node_error has weight of 0.35
  node_error_weight <- (1 - (node_error / denominator)) * 0.35
  # level_error has weight of 0.25
  level_error_weight <- (1 - (as.numeric(temp_tree_statistic[3]) / log((denominator+1), 2))) * 0.25
  # nrin has weight of 0.15
  nrin_weight <- nrin * 0.15
  # side_balance has weight of 0.15
  side_balance_weight <- side_balance * 0.15
  # height_balance has weight of 0.1
  height_balance_weight <- (1 - (as.numeric(temp_tree_statistic[7]) / log((denominator+1), 2))) * 0.1

  # sum up weights to calculate temp_tree_statistic and return max(0, weights)
  return (max(0, (node_error_weight + level_error_weight + side_balance_weight + height_balance_weight + nrin_weight)))
}
```
~~~

Using these four helper functions, the program then performs the tree graph identification analysis on every group within the network.



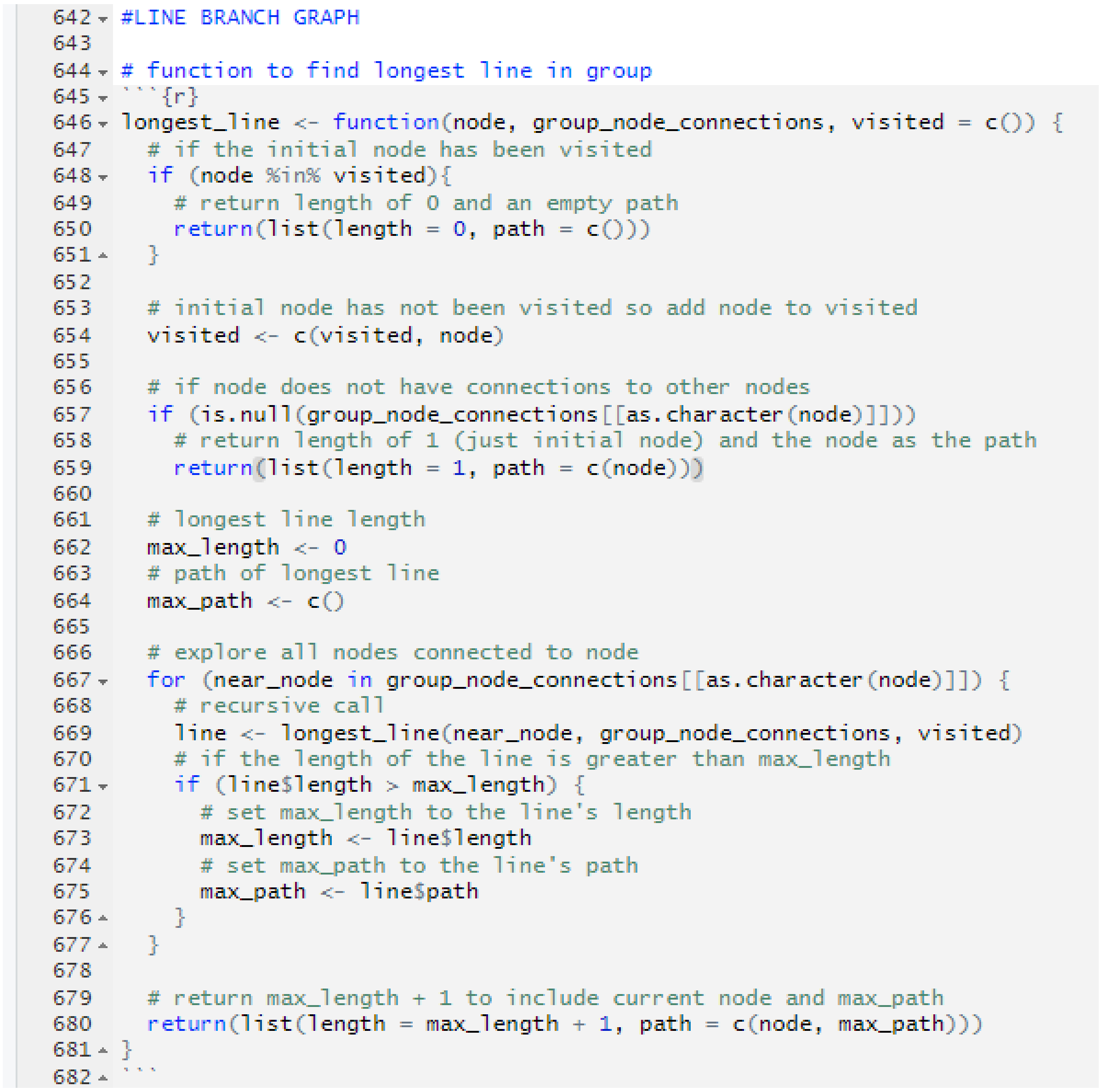
~~~r
# Tree Graph Analysis
```{r}
print("-------- Tree Graph Analysis --------")

# create new from_to list for finding cycle
tree_cycle_graph_list <- list()
# iterate through nodes in same group
for (group in unique(new_g_nodes$group)){
  # add nodes in same group to tree_cycle_graph_list
  tree_cycle_graph_list[[as.character(group)]] <- new_g[[as.character(group)]]
}

# create potential root nodes list
potential_root_nodes <- list()

# get each group in tree_cycle_graph_list
for (group in names(tree_cycle_graph_list)){
  # group must have more than 1 node
  if (length(tree_cycle_graph_list[[group]]) > 1) {
    # if there is a cycle
    if (has_cycle(tree_cycle_graph_list[[group]], group_node_connections)) {
      # print statement
      print(sprintf("   Group %s has a cycle so not a tree", group))
    # group does not have cycle
    } else {
      # look at edges in edges leaving groups
      for (edge in out_edge_groups_f[[group]]){
        # extract number from edge
        number <- gregexpr("[0-9]+", edge)
        result <- regmatches(edge, number)
        #get first number (the from node)
        node <- as.numeric((result[[1]][1]))
        # add node to potential root nodes list
        potential_root_nodes[[group]] <- unique(c(potential_root_nodes[[group]], node))
      }
      # look at edges in edges entering groups
      for (edge in out_edge_groups_s[[group]]){
        # extract number from edge
        number <- gregexpr("[0-9]+", edge)
        result <- regmatches(edge, number)
        #get first number (the from node)
        node <- as.numeric((result[[1]][2]))
        # add node to potential root nodes list
        potential_root_nodes[[group]] <- unique(c(potential_root_nodes[[group]], node))
      }

      # if there are no nodes for the group in potential_root_nodes list
      if (length(potential_root_nodes[[group]]) == 0){
        # add every node in the group to potential_root_nodes list
        for (node in tree_cycle_graph_list[[group]]){
          # only keep unique nodes
          potential_root_nodes[[group]] <- unique(c(potential_root_nodes[[group]], node))
        }
      }

    }
  }
}
~~~



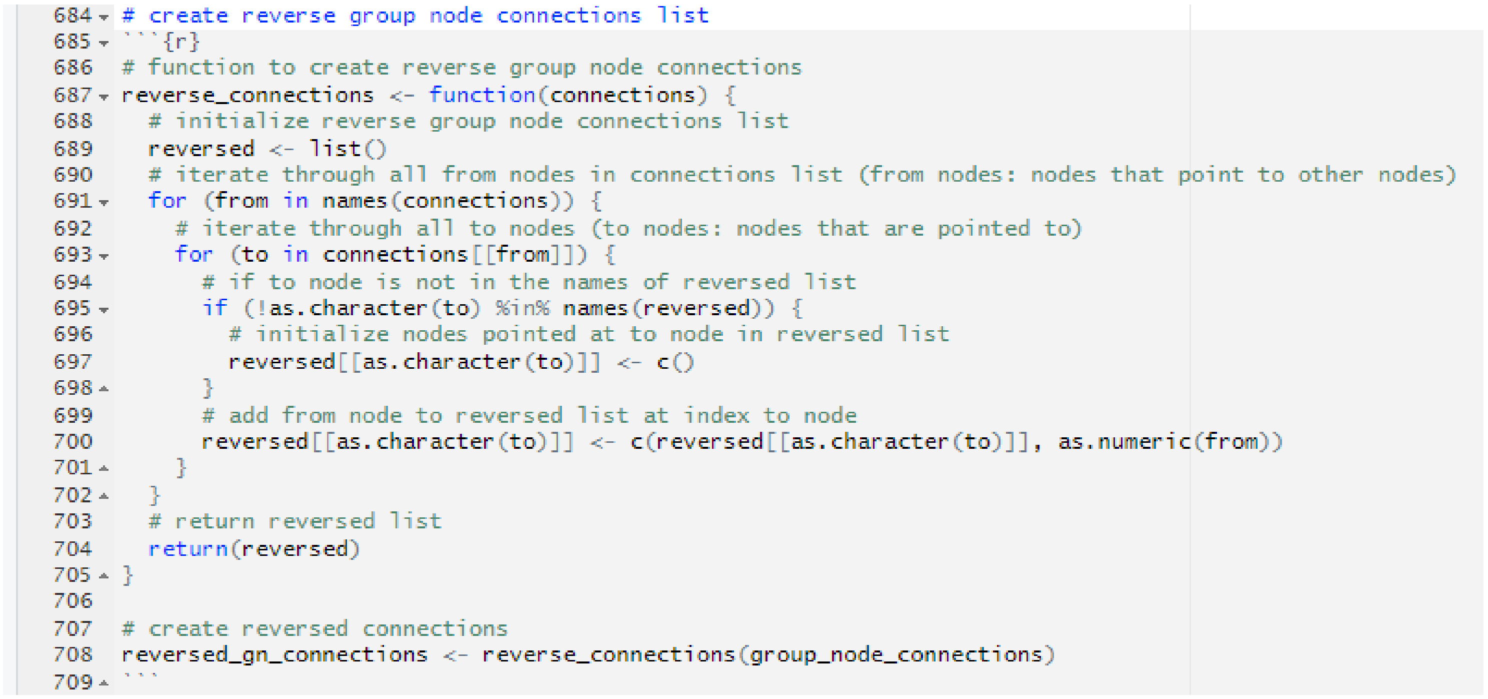
```r

# create best root node analysis list (keeps track of best root node, best tree value, and best tree statistics)
broot_node_analysis <- list()

# group index
group_i <- 1
# iterate through groups in potential_root_nodes list
for (group in names(potential_root_nodes)){
  # statistic of best tree in group
  best_tree_statistic <- c()
  # analysis of best tree in group
  best_tree_analysis <- 0
  # root node in group that led to best tree
  best_node <- 0
  # iterate through nodes in groups in potential_root_nodes
  for (node in potential_root_nodes[[group]]){
    # create a tree for root node
    tree <- create_tree(list(), 1, node)

    # get statistics of temp_tree
    temp_tree_statistic <-  tree_statistics(tree, group_node_connections)
    # get analysis of temp_tree
    temp_tree_analysis <- tree_analysis(temp_tree_statistic)

    # if temp_tree analysis >= best_tree_analysis
    if (temp_tree_analysis >= best_tree_analysis){
      # set best_tree_statistic to temp_tree_statistic
      best_tree_statistic <- temp_tree_statistic
      # set best_tree_analysis to temp_tree_analysis
      best_tree_analysis <- temp_tree_analysis
      # set best_node to node
      best_node <- node
    }
  }
  # save best_tree_statistic in broot_node_analysis
  broot_node_analysis[[group_i]] <- c(best_node, best_tree_analysis, best_tree_statistic)

  # Group i analysis
  print(sprintf("Group %s:", group))
  # if the tree statistic for the group is greater than or equal to 0.8 then the group is a tree
  if (best_tree_analysis >= 0.7){
    # print statements
    print(sprintf("   Group %s is a tree graph with node %i being the root node", group, as.numeric(broot_node_analysis[[group_i]][1])))
    print(sprintf("   %f%% of the group shares the same properties as those of an ideal tree graph", 100*as.numeric(broot_node_analysis[[group_i]][2])))
    print(sprintf("   %i Node Errors (%i Extra Nodes, %i Missing Nodes), %i Level Errors, %i Left Nodes and %i Right Nodes, %s (%f%%) NRIN, %i Height Balance",
as.numeric(broot_node_analysis[[group_i]][3])+as.numeric(broot_node_analysis[[group_i]][4]), as.numeric(broot_node_analysis[[group_i]][3]), as.numeric(broot_node_analysis[[group_i]][4]),
as.numeric(broot_node_analysis[[group_i]][5]), as.numeric(broot_node_analysis[[group_i]][6]), as.numeric(broot_node_analysis[[group_i]][7]), broot_node_analysis[[group_i]][8],
100*as.numeric(eval(parse(text = broot_node_analysis[[group_i]][8]))),  as.numeric(broot_node_analysis[[group_i]][9])))
  # if the tree statistic for the group is less than 0.8 then the group is not a tree
  }else{
    # print statements
    print(sprintf("   Group %s is not a tree and has a structure closest to a tree graph when node %i is the root node", group, as.numeric(broot_node_analysis[[group_i]][1])))
    print(sprintf("   %f%% of the group sharing the same properties as those of an ideal tree graph", 100*as.numeric(broot_node_analysis[[group_i]][2])))
    print(sprintf("   %i Node Errors (%i Extra Nodes, %i Missing Nodes), %i Level Errors, %i Left Nodes and %i Right Nodes, %s (%f%%) NRIN, %i Height Balance",
as.numeric(broot_node_analysis[[group_i]][3])+as.numeric(broot_node_analysis[[group_i]][4]), as.numeric(broot_node_analysis[[group_i]][3]), as.numeric(broot_node_analysis[[group_i]][4]),
as.numeric(broot_node_analysis[[group_i]][5]), as.numeric(broot_node_analysis[[group_i]][6]), as.numeric(broot_node_analysis[[group_i]][7]), broot_node_analysis[[group_i]][8],
100*as.numeric(eval(parse(text = broot_node_analysis[[group_i]][8]))),  as.numeric(broot_node_analysis[[group_i]][9])))
  }
  # increase group_i by 1
  group_i <- group_i + 1
}
```
```
```

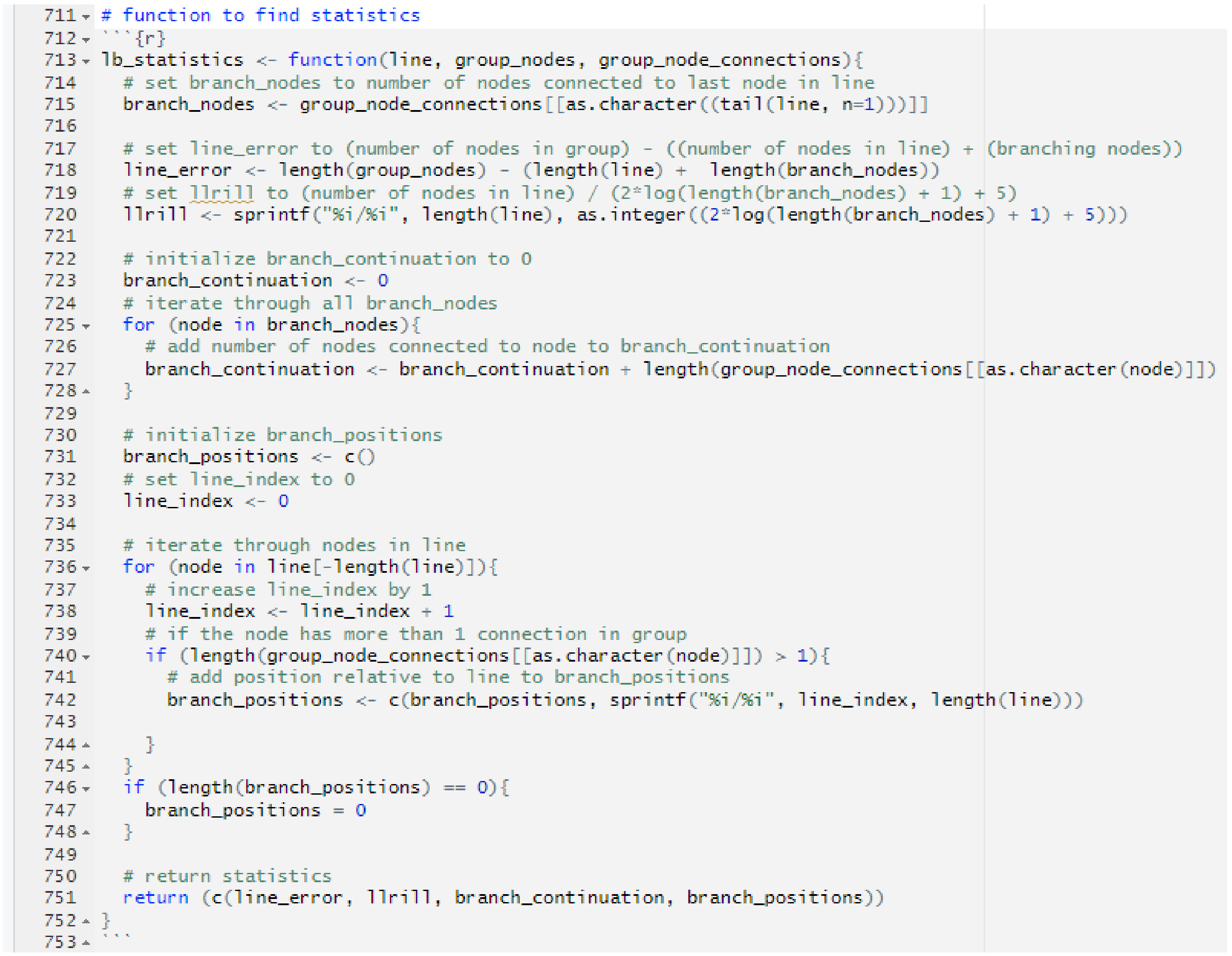
Next, the program determines which communities are line branch graphs by comparing every community against the properties of an ideal line branch graph.

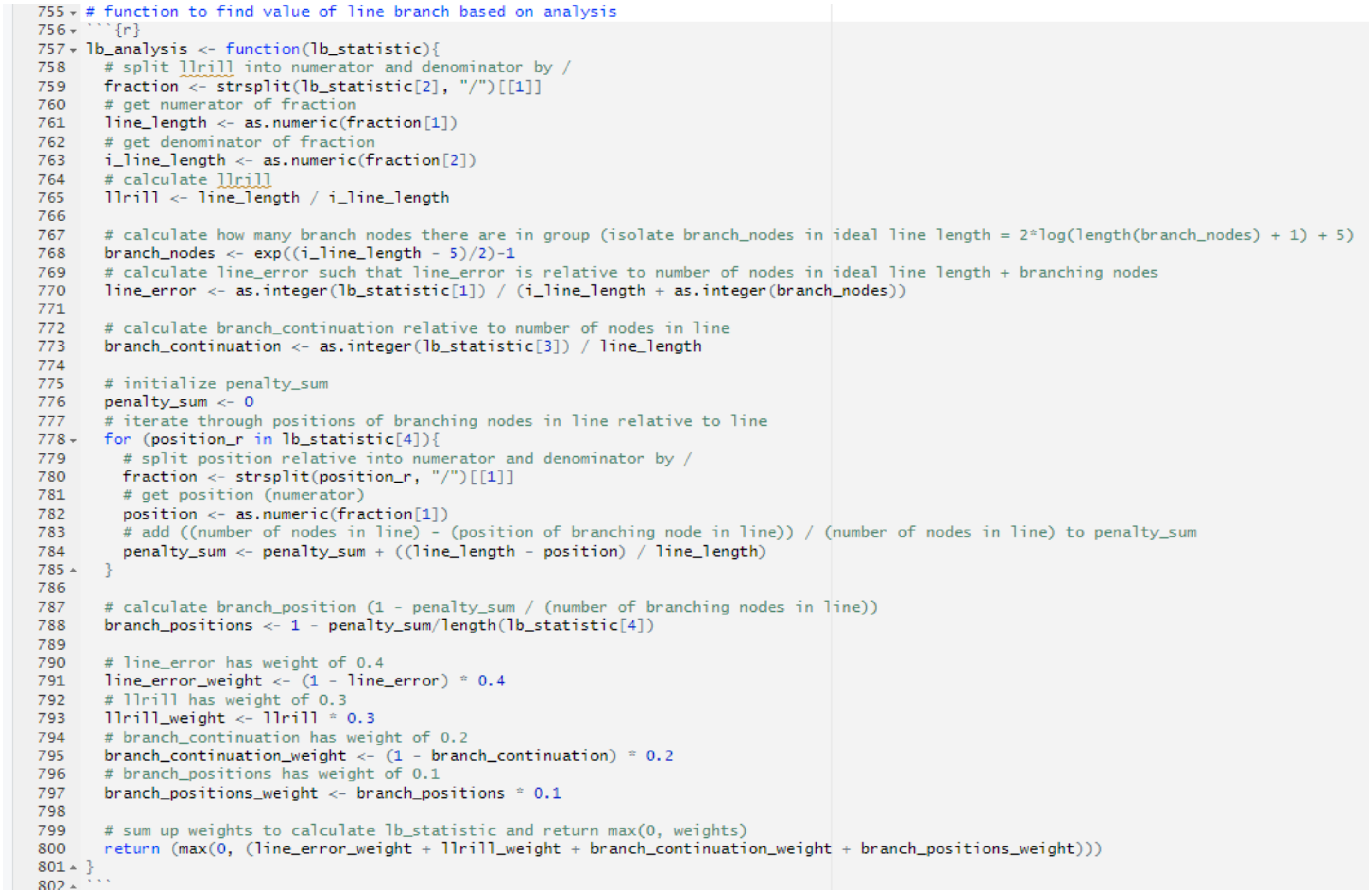
Like the analyzation of the cloistered graph and tree graph, I created several helper functions to assist with the analysis. The first helper function finds the longest line that ends with a given node in a group. The second function finds every reversed node connection within the same group. The third helper function examines the group in its line branch structure and calculates specific statistics, which will be discussed in Section 3.1. The fourth helper function evaluates these statistics to determine how similar the line branch structure is to an ideal line branch graph.



~~~
#LINE BRANCH GRAPH

# function to find longest line in group
```{r}
longest_line <- function(node, group_node_connections, visited = c()) {
  # if the initial node has been visited
  if (node %in% visited){
    # return length of 0 and an empty path
    return(list(length = 0, path = c()))
  }

  # initial node has not been visited so add node to visited
  visited <- c(visited, node)

  # if node does not have connections to other nodes
  if (is.null(group_node_connections[[as.character(node)]]))
    # return length of 1 (just initial node) and the node as the path
    return(list(length = 1, path = c(node)))

  # longest line length
  max_length <- 0
  # path of longest line
  max_path <- c()

  # explore all nodes connected to node
  for (near_node in group_node_connections[[as.character(node)]]) {
    # recursive call
    line <- longest_line(near_node, group_node_connections, visited)
    # if the length of the line is greater than max_length
    if (line$length > max_length) {
      # set max_length to the line's length
      max_length <- line$length
      # set max_path to the line's path
      max_path <- line$path
    }
  }

  # return max_length + 1 to include current node and max_path
  return(list(length = max_length + 1, path = c(node, max_path)))
}
```
~~~

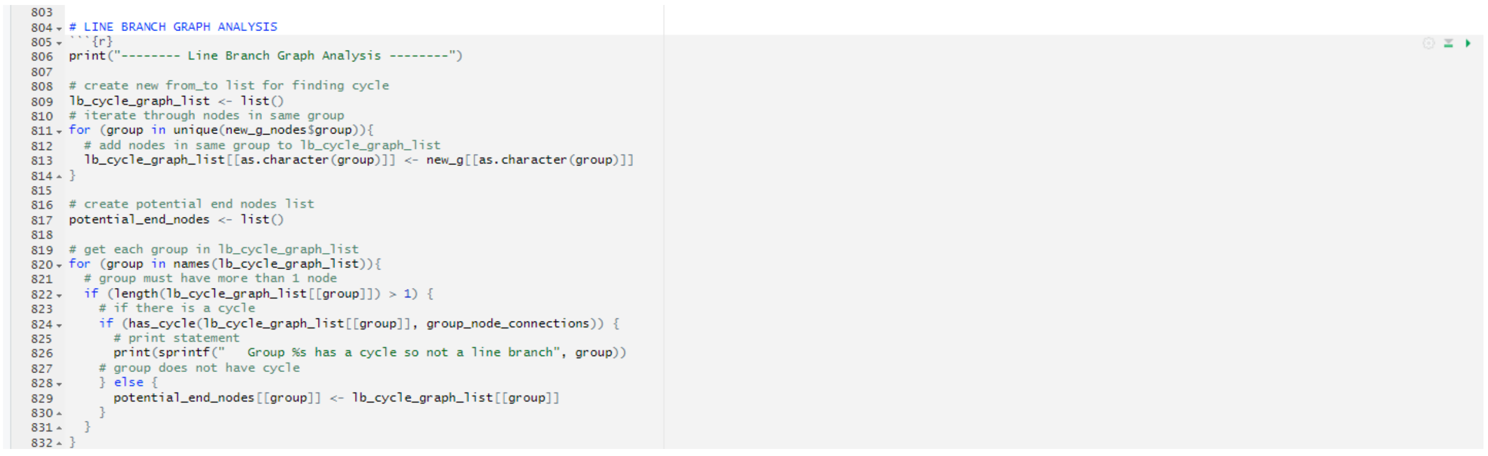


~~~
# create reverse group node connections list
```{r}
# function to create reverse group node connections
reverse_connections <- function(connections) {
  # initialize reverse group node connections list
  reversed <- list()
  # iterate through all from nodes in connections list (from nodes: nodes that point to other nodes)
  for (from in names(connections)) {
    # iterate through all to nodes (to nodes: nodes that are pointed to)
    for (to in connections[[from]]) {
      # if to node is not in the names of reversed list
      if (!as.character(to) %in% names(reversed)) {
        # initialize nodes pointed at to node in reversed list
        reversed[[as.character(to)]] <- c()
      }
      # add from node to reversed list at index to node
      reversed[[as.character(to)]] <- c(reversed[[as.character(to)]], as.numeric(from))
    }
  }
  # return reversed list
  return(reversed)
}

# create reversed connections
reversed_gn_connections <- reverse_connections(group_node_connections)
~~~
```



```
function to find statistics
```{r}
lb_statistics <- function(line, group_nodes, group_node_connections){
  # set branch_nodes to number of nodes connected to last node in line
  branch_nodes <- group_node_connections[[as.character((tail(line, n=1)))]]

  # set line_error to (number of nodes in group) - ((number of nodes in line) + (branching nodes))
  line_error <- length(group_nodes) - (length(line) +  length(branch_nodes))
  # set llrill to (number of nodes in line) / (2*log(length(branch_nodes) + 1) + 5)
  llrill <- sprintf("%i/%i", length(line), as.integer((2*log(length(branch_nodes) + 1) + 5)))

  # initialize branch_continuation to 0
  branch_continuation <- 0
  # iterate through all branch_nodes
  for (node in branch_nodes){
    # add number of nodes connected to node to branch_continuation
    branch_continuation <- branch_continuation + length(group_node_connections[[as.character(node)]])
  }

  # initialize branch_positions
  branch_positions <- c()
  # set line_index to 0
  line_index <- 0

  # iterate through nodes in line
  for (node in line[-length(line)]){
    # increase line_index by 1
    line_index <- line_index + 1
    # if the node has more than 1 connection in group
    if (length(group_node_connections[[as.character(node)]]) > 1){
      # add position relative to line to branch_positions
      branch_positions <- c(branch_positions, sprintf("%i/%i", line_index, length(line)))

    }
  }
  if (length(branch_positions) == 0){
    branch_positions = 0
  }

  # return statistics
  return (c(line_error, llrill, branch_continuation, branch_positions))
}
```
```
```

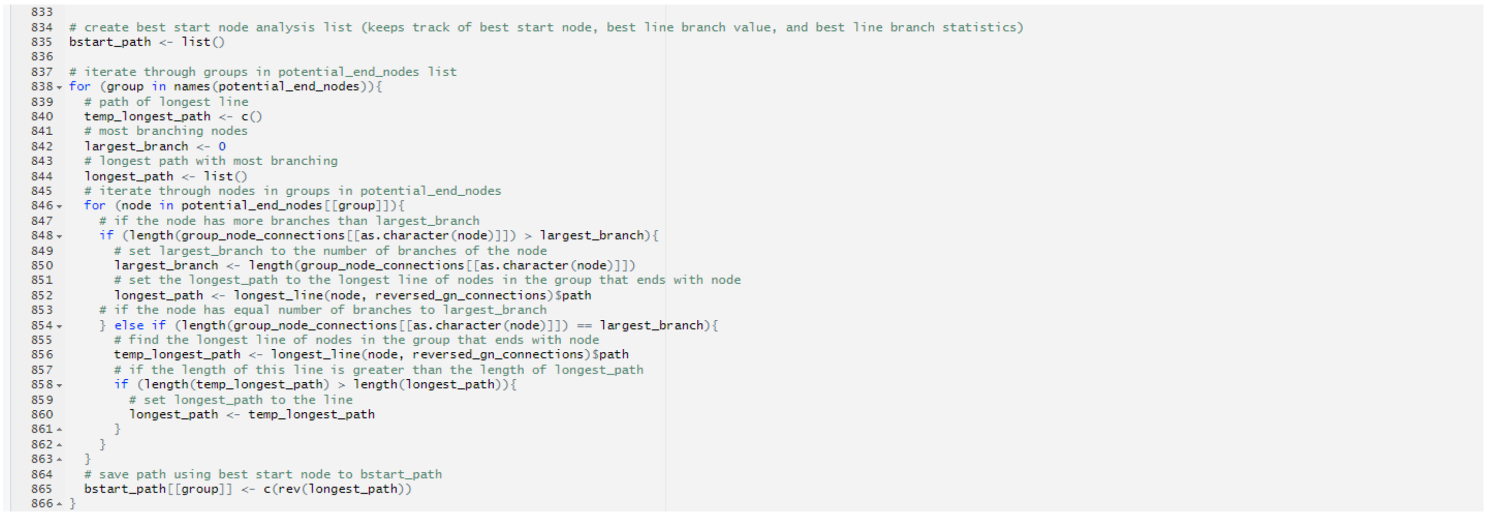


~~~
# function to find value of line branch based on analysis
```{r}
lb_analysis <- function(lb_statistic){
  # split llrill into numerator and denominator by /
  fraction <- strsplit(lb_statistic[2], "/")[[1]]
  # get numerator of fraction
  line_length <- as.numeric(fraction[1])
  # get denominator of fraction
  i_line_length <- as.numeric(fraction[2])
  # calculate llrill
  llrill <- line_length / i_line_length

  # calculate how many branch nodes there are in group (isolate branch_nodes in ideal line length = 2*log(length(branch_nodes) + 1) + 5)
  branch_nodes <- exp((i_line_length - 5)/2)-1
  # calculate line_error such that line_error is relative to number of nodes in ideal line length + branching nodes
  line_error <- as.integer(lb_statistic[1]) / (i_line_length + as.integer(branch_nodes))

  # calculate branch_continuation relative to number of nodes in line
  branch_continuation <- as.integer(lb_statistic[3]) / line_length

  # initialize penalty_sum
  penalty_sum <- 0
  # iterate through positions of branching nodes in line relative to line
  for (position_r in lb_statistic[4]){
    # split position relative into numerator and denominator by /
    fraction <- strsplit(position_r, "/")[[1]]
    # get position (numerator)
    position <- as.numeric(fraction[1])
    # add ((number of nodes in line) - (position of branching node in line)) / (number of nodes in line) to penalty_sum
    penalty_sum <- penalty_sum + ((line_length - position) / line_length)
  }

  # calculate branch_position (1 - penalty_sum / (number of branching nodes in line))
  branch_positions <- 1 - penalty_sum/length(lb_statistic[4])

  # line_error has weight of 0.4
  line_error_weight <- (1 - line_error) * 0.4
  # llrill has weight of 0.3
  llrill_weight <- llrill * 0.3
  # branch_continuation has weight of 0.2
  branch_continuation_weight <- (1 - branch_continuation) * 0.2
  # branch_positions has weight of 0.1
  branch_positions_weight <- branch_positions * 0.1

  # sum up weights to calculate lb_statistic and return max(0, weights)
  return (max(0, (line_error_weight + llrill_weight + branch_continuation_weight + branch_positions_weight)))
}
```
~~~

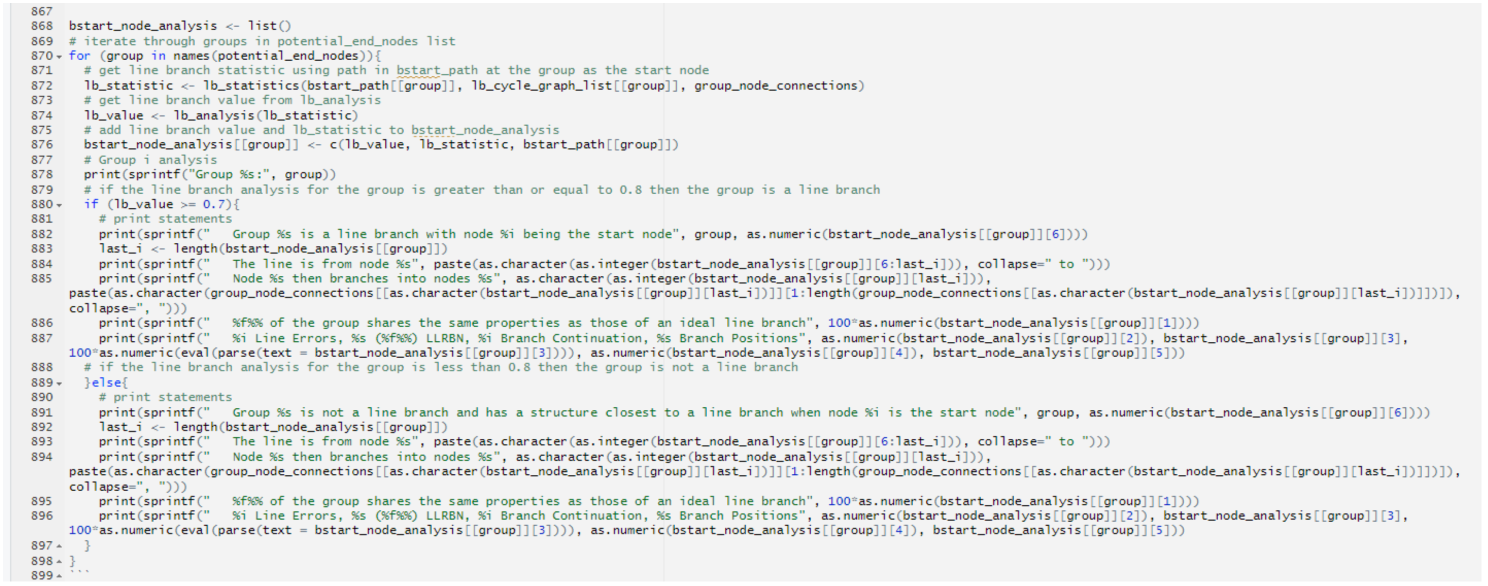
Using these four helper functions, the program then performs the line branch graph identification analysis on every group within the network.



~~~
# LINE BRANCH GRAPH ANALYSIS
```{r}
print("-------- Line Branch Graph Analysis --------")

# create new from_to list for finding cycle
lb_cycle_graph_list <- list()
# iterate through nodes in same group
for (group in unique(new_g_nodes$group)){
  # add nodes in same group to lb_cycle_graph_list
  lb_cycle_graph_list[[as.character(group)]] <- new_g[[as.character(group)]]
}

# create potential end nodes list
potential_end_nodes <- list()

# get each group in lb_cycle_graph_list
for (group in names(lb_cycle_graph_list)){
  # group must have more than 1 node
  if (length(lb_cycle_graph_list[[group]]) > 1) {
    # if there is a cycle
    if (has_cycle(lb_cycle_graph_list[[group]], group_node_connections)) {
      # print statement
      print(sprintf("   Group %s has a cycle so not a line branch", group))
    # group does not have cycle
    } else {
      potential_end_nodes[[group]] <- lb_cycle_graph_list[[group]]
    }
  }
}
~~~



```

# create best start node analysis list (keeps track of best start node, best line branch value, and best line branch statistics)
bstart_path <- list()

# iterate through groups in potential_end_nodes list
for (group in names(potential_end_nodes)){
  # path of longest line
  temp_longest_path <- c()
  # most branching nodes
  largest_branch <- 0
  # longest path with most branching
  longest_path <- list()
  # iterate through nodes in groups in potential_end_nodes
  for (node in potential_end_nodes[[group]]){
    # if the node has more branches than largest_branch
    if (length(group_node_connections[[as.character(node)]]) > largest_branch){
      # set largest_branch to the number of branches of the node
      largest_branch <- length(group_node_connections[[as.character(node)]])
      # set the longest_path to the longest line of nodes in the group that ends with node
      longest_path <- longest_line(node, reversed_gn_connections)$path
    # if the node has equal number of branches to largest_branch
    } else if (length(group_node_connections[[as.character(node)]]) == largest_branch){
      # find the longest line of nodes in the group that ends with node
      temp_longest_path <- longest_line(node, reversed_gn_connections)$path
      # if the length of this line is greater than the length of longest_path
      if (length(temp_longest_path) > length(longest_path)){
        # set longest_path to the line
        longest_path <- temp_longest_path
      }
    }
  }
  # save path using best start node to bstart_path
  bstart_path[[group]] <- c(rev(longest_path))
}
```



```

bstart_node_analysis <- list()
# iterate through groups in potential_end_nodes list
for (group in names(potential_end_nodes)){
  # get line branch statistic using path in bstart_path at the group as the start node
  lb_statistic <- lb_statistics(bstart_path[[group]], lb_cycle_graph_list[[group]], group_node_connections)
  # get line branch value from lb_analysis
  lb_value <- lb_analysis(lb_statistic)
  # add line branch value and lb_statistic to bstart_node_analysis
  bstart_node_analysis[[group]] <- c(lb_value, lb_statistic, bstart_path[[group]])
  # Group i analysis
  print(sprintf("Group %s:", group))
  # if the line branch analysis for the group is greater than or equal to 0.8 then the group is a line branch
  if (lb_value >= 0.7){
    # print statements
    print(sprintf("   Group %s is a line branch with node %i being the start node", group, as.numeric(bstart_node_analysis[[group]][6])))
    last_i <- length(bstart_node_analysis[[group]])
    print(sprintf("   The line is from node %s", paste(as.character(as.integer(bstart_node_analysis[[group]][6:last_i])), collapse=" to ")))
    print(sprintf("   Node %s then branches into nodes %s", as.character(as.integer(bstart_node_analysis[[group]][last_i])),
paste(as.character(group_node_connections[[as.character(bstart_node_analysis[[group]][last_i])]][1:length(group_node_connections[[as.character(bstart_node_analysis[[group]][last_i])]])]),
collapse=", ")))
    print(sprintf("   %f%% of the group shares the same properties as those of an ideal line branch", 100*as.numeric(bstart_node_analysis[[group]][1])))
    print(sprintf("   %i Line Errors, %s (%f%%) LLRBN, %i Branch Continuation, %s Branch Positions", as.numeric(bstart_node_analysis[[group]][2]), bstart_node_analysis[[group]][3],
100*as.numeric(eval(parse(text = bstart_node_analysis[[group]][3]))), as.numeric(bstart_node_analysis[[group]][4]), bstart_node_analysis[[group]][5]))
  # if the line branch analysis for the group is less than 0.8 then the group is not a line branch
  }else{
    # print statements
    print(sprintf("   Group %s is not a line branch and has a structure closest to a line branch when node %i is the start node", group, as.numeric(bstart_node_analysis[[group]][6])))
    last_i <- length(bstart_node_analysis[[group]])
    print(sprintf("   The line is from node %s", paste(as.character(as.integer(bstart_node_analysis[[group]][6:last_i])), collapse=" to ")))
    print(sprintf("   Node %s then branches into nodes %s", as.character(as.integer(bstart_node_analysis[[group]][last_i])),
paste(as.character(group_node_connections[[as.character(bstart_node_analysis[[group]][last_i])]][1:length(group_node_connections[[as.character(bstart_node_analysis[[group]][last_i])]])]),
collapse=", ")))
    print(sprintf("   %f%% of the group shares the same properties as those of an ideal line branch", 100*as.numeric(bstart_node_analysis[[group]][1])))
    print(sprintf("   %i Line Errors, %s (%f%%) LLRBN, %i Branch Continuation, %s Branch Positions", as.numeric(bstart_node_analysis[[group]][2]), bstart_node_analysis[[group]][3],
100*as.numeric(eval(parse(text = bstart_node_analysis[[group]][3]))), as.numeric(bstart_node_analysis[[group]][4]), bstart_node_analysis[[group]][5]))
  }
}
```
```
```

2.4 What does it do

The program allows for both visualization and analysis of networks by graphing the entire network and identifying subgraphs among communities within the network. The program also incorporates a community detection algorithm to split the network into distinct communities. The network is then visually represented as a directed graph, highlighting the connections between nodes (users) and the direction of the connection (flow of the information).

Once the network is split into communities, the program studies the communities by analyzing properties to identify subgraphs. By analyzing structural properties and the flow of information, the program calculates if a community is a cloistered graph and assesses the similarities of communities to the ideal form of tree and line branch graphs. The program is a dynamic tool designed to visualize the network, analyze its structure, and evaluate the flow of information.

2.5 How does it work

To provide a clear understanding of how the program operates and identifies subgraphs within the network, the following lines of pseudocode break down and explain each segment of the code and its underlying ideas

```
1 load packages
2 read csv file
3 split data in csv file into nodes that point to other nodes (from id list) and nodes that are pointed to (to id list)

4 iterate through every row in csv file to create directed list (from_to list) using from and to id list

### GRAPHING
5 create a graph object (from igraph package)
6 apply Louvain algorithm on graph object to create new graph object

7 iterate through nodes in graph object to find unique nodes id's and number of nodes in each group

8 iterate through unique node id's to find node labels (node id but as a char)
9 iterate through number of nodes in each group to find node groups (group that node belongs to)
10 iterate through unique node id's to create node titles (title of the node)
11 create node data frame that holds the following info
     unique node id's
     node labels
     node groups
     node titles
```

```
12 iterate through from id list and add 1 to every node id to create from id plus one list
(nodes/indexing in R must start at 1)
13 iterate through to id list and add 1 to every node id to create to id plus one list

14 iterate through from id plus one list to create edge labels (every edge between two
nodes)
15 iterate through from id plus one list to create edge titles (title of every edge)
16 create edge data frame that holds the following info
     from id plus one list
     to id plus one list
     edge labels
     edge titles

17 using node data frame and edge data frame, graph network using visNetwork
18 using graph object, graph network using igraph

### ANALYSIS
# CLOISTERED GRAPH
19 iterate through from_to list and find all node connections for every node

20 iterate through every edge
21     if nodes in edges belong to the same group
22          add the edge to same group edge list
23          add nodes to group node connections list
24     else
25          add edge to different group edge list

26 iterate through graph object to find maximum possible number of edges (MPNE) for every
group

27 iterate through every group
28      find number of edges leaving/entering group
29      find number of edges in group
30      calculate how close group is to a complete graph using number of edges in group and
MPNE (percentage)
31     find how many edges are missing for group to be a complete graph

32     if number of edges leaving/entering a group is less than 2% of MPNE
33          then group is a cloistered graph
34          print statistics and analysis
35     else
36          group is not a cloistered graph
37          print statistics and analysis

38     if percentage is 100% and group is a cloistered graph and group is disconnected
39          then group is an ideal cloistered graph
```

```
         print analysis
    else if percentage is 100% and group is not a cloistered graph
         then group is only a complete graph
         print analysis
    else
         group is value of percentage of a complete graph
         print statistics and analysis


# TREE GRAPH
function create_tree(tree = list(), level = 1, node)
  """
    create_tree makes a tree structure given a node
    :tree: tree structure is contained in a list
    :level: current level of tree as an int
    :node: node as an int to either add to tree or add children of node to tree
    :return: returns tree structure with node as the root node
  """
    if tree has no nodes
         add node to tree at level 1
    else
         add the node to tree at current level

    get children of node (its connections in group node connections list)

    if there are children
         iterate through every child
              recursively call create_tree(tree, level+1, child)

    return tree


function has_cycle(node_list, group_node_connections)
  """
    has_cycle checks if there a cycle given the nodes in a group and the group node connections list
    :node_list: vectors of ints for nodes in a group
    :group_node_connections: group node connections list
    :return: boolean value indicating whether a group has a cycle
  """
         function dfs(node)
       """
         dfs uses dfs to check if a node has been seen before
         :node: int of node id that needs to be checked if node has been seen before
         :return: boolean value that indicates whether node has been seen before
       """
```

```
        if the node has not been visited before
            set value in visited and rec_stack list at index node to TRUE
            if node has other connections in group
                iterate through all connections
                    if other nodes have not been visited and dfs(other nodes)
                        return TRUE
                    else if other nodes are in rec_stack
                        return TRUE

        set value in rec_stack at index node to FAlSE
        return FALSE

    create visited and recursion stack list

    iterate through nodes in node_list
        set every value of visited and rec_stack at index node to FALSE as we have not seen these nodes yet

    iterate through nodes in node_list
        if dfs(node)
            return TRUE

    return FALSE


function tree_statistics(tree, group_node_connections)
   """
    tree_statistics finds the statistics of a tree
    :tree: tree structure as a list
    :group_node_connections: group node connections list
    :return: statistics of a tree as a vector
   """
    initialize line error, extra nodes, missing nodes, height balance, nrin, total nodes, lnol, nol variables

    iterate through every level in tree
        add number of nodes on level to total nodes

        if number of nodes on level is greater than 2 ^ (number of nodes on previous level)
            add difference to extra nodes
            increase level error by 1
        else if less
            add difference to missing nodes
            increase level error by 1
        set nol to number of nodes on level
        if nol is less than or equal to lnol
            increase height balance by 1
        else
            set lnol to nol

    set left nodes list to left half of children of root node
    set total left nodes to number of left nodes
    set right nodes list to right half of children of root node
    set total right nodes to number of right nodes

    while left nodes list is not empty
        if first node in left nodes list has children
            append children to left nodes list
            add number of children to total left nodes
        pop off first node in left nodes list

    while right nodes list is not empty
        if first node in right nodes list has children
            append children to right nodes list
            add number of children to total right nodes
        pop off first node in right nodes list

    set nrin = (total number of nodes in group) / (2 ^ (number of levels in tree) - 1)

    return (extra nodes, missing nodes, level error, total left nodes, total right nodes, nrin, height balance)


function tree_analysis(tree statistics)
    """
    tree_analysis analyzes statistics of a tree and gives a rating of how well the tree is
    :tree statistics: statistics of a tree as a vector
    :return: float
    """
    get left nodes and right nodes from tree statistics (total left nodes and total right nodes)
    if left nodes is less than right nodes
        side balance = left nodes / right nodes
    else
        side balance = right nodes / left nodes

    calculate nrin as a float

    get node error = extra nodes + missing nodes
    node error has weight of 0.35
    level error has weight of 0.25
    nrin has weight of 0.15
    side balance has weight of 0.15
    height balance has weight of 0.1

    find value of node error given its weight and its relative value
    find value of level error given its weight and its relative value
    find value of nrin given its weight
    find value of side balance given its weight
    find value of height balance given its weight and its relative value
    sum up values

    return max(0, sum of values)


create potential root nodes list
iterate through every group in network
    if group has more than one node
        check if group has a cycle by calling has_cycle(group, group_node_connections)
            if group has a cycle then group is not a tree
        else
            iterate through edges leaving/entering group
                add nodes in group connected to other groups to potential root nodes list

            if there are no nodes in potential root nodes list
                add every node in group to potential root nodes list

create best root node analysis list
iterate through every group
    set best tree statistics, best tree analysis, and best node to 0
    iterate through nodes in potential root nodes list that belong to same group
        create tree by calling create_tree(list(), 1, node)
        get statistics of tree by calling tree_statistics(tree, group_node_connections)
        get analysis of tree by calling tree_analysis(tree statistics)

        if tree analysis > best tree analysis
            set best tree statistics to tree statistics
            set best tree analysis to tree analysis
            set best node to node

    save best node, best tree analysis, and best tree statistics to best root node analysis list at index group
```

```
167    if best tree analysis is greater than or equal to 0.7
168         then group is a tree with best node being the root node
169         print statistics and analysis
170    else
171         group is not a tree but best node being the root node leads to a tree structure
closest to an ideal tree compared to other nodes in group as the root node
172         print statistics and analysis


# LINE BRANCH GRAPH
173 function longest_line(node, group_node_connections, visited = c())
     """
174    longest_line finds the longest line of nodes in a group that ends with node
175    :node: node id as an int of last node in longest line of nodes for a group
176    :group_node_connections: group node connections list
177    :visited: visited nodes in group as a list
178    :return: list with length of longest line and nodes in longest line
     """
179    if node is in visited list
180         return (length = 0, path = c())

181    add node to visited list

182    if node does not have connections in group
183         return (length = 1, path = c(node))

184    set max length and max path to 0
185    iterate through nodes that node is connected to in group
186         recursively call longest_line(near_node, group_node_connections, visited)
187         if the length of line is greater than max length
188              set max length to length of line
189              set max path to path of line

190    return (length = max length, path = c(node, max path))


191 function reverse_connections(connections)
     """
192    reverse_connections reverses a connections list (reverses 'from nodes' -> 'to nodes'
to 'to nodes' -> 'from nodes')
193    :connections: connections list
194    :return: connections list but reversed
     """
195    initialize reversed list
196    iterate through all from nodes in connections list
```

```
        iterate through all to nodes that are connected to from node
            add to node and its from node to reversed connections list

    return reversed connections list


create reversed group node connections by calling reverse_connections(group node connections)



function lb_statistics(line, group_nodes, group_node_connections)
  """
    lb_statistics finds the statistics of a line branch
    :line: line of nodes as a vector of ints
    :group_nodes: all nodes in the group as a vector of ints
    :group_node_connections: group node connections list
    :return: statistics of a line branch as a vector
  """
    set branching nodes to the connections of last node in line

    set line error = group_nodes - ((number of nodes in line) + (number of branching nodes))

    set llrill = (number of nodes in line) / (2*log(number of branching nodes + 1) + 5)

    initialize branch continuation to 0
        iterate through all nodes in branching nodes
        add number of connections each node has to branch continuation

    initialize branch positions to 0
        iterate through all nodes in line
            if node has more than one connection in group
                add the position of node to branch positions

    return (line error, llrill, branch continuation, branch positions)


function lb_analysis(lb_statistic)
  """
    lb_analysis analyzes statistics of a line branch and gives a rating of well the line branch is
    :lb_statistic: statistics of a line branch as a vector
    :return: float
  """
    set line length to numerator of llrill
    set ideal line length to denominator of llrill
    calculate llrill as a float

    calculate how many branch nodes there are in group

    calculate line error = line error / ((ideal line length) + (number of branch nodes))

    calculate branch continuation = branch continuation / line length


    set penalty sum to 0
    iterate through branch positions
        get node position
        add (line length - node position) / (line length) to penalty sum
    calculate branch positions = 1 - (penalty sum) / (number of positions in branch positions)

    line error has weight of 0.4
    llrill has weight of 0.3
    branch continuation has weight of 0.2
    branch position has weight of 0.1

    find value of line error given its weight and its relative value
    find value of lrrill given its weight
    find value of branch continuation given its weight and its relative value
    find value of branch position given its weight
    sum up values

    return max(0, sum of values)


create potential start nodes list
iterate through every group in network
    if group has more than one node
        if has_cycle(group, group_node_connections)
            group has a cycle so not a line branch
        else
            add every node in group to potential start nodes list

create best start path list
iterate through every group
    set temporary longest path, largest branch, longest path to 0
    iterate through nodes in potential start nodes list that belong to same group
        if node has more connections than largest branch
            set largest branch to number of connections node has
            set longest path to path ending with node by calling longest_line(node,
reversed group node connections) and getting its path
        else if node has number of connections same as largest branch
            set temporary longest path to path from longest_line(node, reversed group node connections)
            if number of nodes in temporary longest path is greater than number of nodes in longest path
                set longest path to temporary longest path

    save longest path to best start path list at index group

create best start node analysis list
iterate through every group
    get statistics of line branch by calling lb_statistics(path at index group in best start path list, nodes in group, group node connections)
    get analysis of line branch by calling lb_analysis(lb_statistic)
    add line branch analysis and statistic and path to best start node analysis list at index group

    if line branch analysis is greater than or equal to 0.7
        then group is a line branch
        print statistics and analysis
    else
        group is not a line branch
        print statistics and analysis
```

III. Detailed Project Description

3.1 Domain Model

1. Cloistered Graph

Definitions:

Cloistered Graph: Group that has few to none edges leaving or entering the group

Ideal Cloistered Graph: Complete graph that is disconnected

MPNE: Maximum possible number of edges in a group. Calculated by taking every node in group and finding all possible connections between every node in same group.

Method to identify cloistered graphs: Compare Number of Edges to Max Possible Number of Edges (MPNE)

1. Find how many edges are in the group
2. Find the number of edges that leave/enter the group
3. Calculate MPNE
4. If number of edges leaving/entering the group is less than 2% of MPNE of that group, then group is a cloistered group. If greater than 2% of MPNE, then group is not a cloistered group
5. Divide number of edges by MPNE. This percentage represents how close the group is to that of a complete graph.

Example:

Group has 12 nodes with 11 inner group edges (1 node connects to the other 11 nodes in the group). The group also has 2 outer group edges.

11 edges inside group
2 edges leaving/entering the group
MPNE: 12 nodes * (12 nodes – 1 node) / 2 = 66 edges
*Because 2 / 66 * 100% = 3.03% (> 2%) of nodes leave/enter the group, the group is not a cloistered group*

Accuracy Percentage: 11 edges / MPNE * 100% = 16.67%
Group is 16.67% of a complete graph

2. Tree

Definitions:

Ideal Tree: A noncyclic graph where the graph develops from a single node (root node) and each node splits into 2 child nodes

Tree Finder Algorithm (TFA): Properties to check/find to identify whether a group is a tree

What the TFA checks for and analyzes:

- Node Error
- Level Error
- Side Balance
- Height Balance
- Nodes Relative to Number of Ideal Nodes

Node Error: Sum of missing nodes and extra nodes that should not exist in a tree (if there is an extra node that should not exist, then assume for the remaining levels that the deeper nodes should have two additional child nodes to ensure that the error is not counted more than once). This statistic helps identify deviations from an ideal tree by analyzing node counts at ever level.

Level Error: Number of levels with an incorrect number of nodes. For each level, calculate the expected number of nodes based on the previous level (e.g. if level 2 has 3 nodes, level 3 should have 6 nodes). This statistic is assessed on a per-level basis and evaluates how closely the tree follows the property of each parent node having two children nodes.

Side Balance: Determines how evenly distributed the nodes are between the left and right sides of the tree. Split the root node into two sides: left and right. Then count the number of nodes on each side. This statistic finds how balanced the tree is in terms of side distribution.

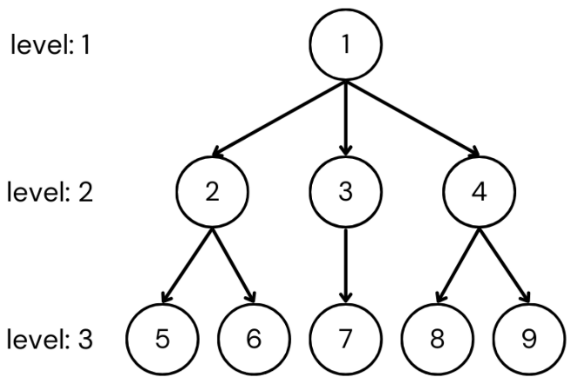
Height Balance: Evaluates the distribution of nodes across the levels of the tree. Iterate through all levels and find how many levels have fewer nodes than the largest number of nodes observed so far on a level (lnol). This statistic determines whether the tree continues to grow in terms of number of nodes per level.

Nodes Relative to Number of Ideal Nodes (NRIN): Total number of nodes in tree relative to number of nodes in an ideal tree with the same depth. This statistic determines how close the tree is to an ideal tree by comparing the actual number of nodes to the theoretical ideal number of nodes given the depth level.

Method to identify trees:

   a. Check for cycles: if the group contains a cycle, it cannot be a tree
   b. Identify potential root nodes: create a list of potential root nodes for each group if no cycle exists in that group
   c. Evaluate each root node: for each node in the group's potential root nodes list
      i. Consider current node to be root node
      ii. Create tree structure using root node
      iii. Apply TFA on tree structure to get statistics
      iv. Compare statistics to best statistics observed so far. If statistics are better, then save statistics as best statistics and save root node as best root node
   d. Repeat until group potential root nodes exhausted: repeat step c) until list of nodes from same group in potential root nodes list is exhausted
   e. Output results: from the list of best statistics, identify the group and output the best node and statistics that form a tree closest to an ideal tree when the best node is used as the root node

Example:



Node Error: 2
Node 1 should only have 2 children (has 1 extra child).
Node 3 should have 2 children (has 1 missing child)
1 extra child + 1 missing child = 2 node errors

Level Error: 2
Level 2 should have 2 nodes (has 3 nodes)
Level 3 should have 6 nodes (has 5 nodes)
Level 2 and Level 3 have errors, so Level Error = 2

Side Balance = 1
Both Left Side and Right Side have same number of nodes

Height Balance: 3
Number of nodes increases every level, because there are 3 levels then Height Balance = 3

NRIN: 9/7
9 total nodes and ideal tree of height 3 should have 7 nodes

Value of Tree:
(1 - (2/7)) * (0.35) + (1 - (2/3)) * 0.25 + 9/7 * 0.15 + 1 * 0.15 + (1 - (3/3)) * 0.1 = 0.67619047619

3. Line Branch

Definitions:

Ideal Line Branch: a sequence of nodes connected in a path sequence where the final node in the path branches into multiple children nodes

Line Branch Finder Algorithm (LBFA): Properties to check/find to identify whether a group is a line branch

What the LBFA checks for and the analyzes:

- Line Error
- Line Length Relative to Ideal Line Length
- Branch Continuation
- Branch Positions

Line Error: Finds how many errors there are in the line (nodes in the line should not branch before the final node in line). Calculate the number of nodes that branch prematurely, as well as the total number of nodes in branches that should not exist. This statistic determines how much the line branch deviates from an ideal line branch.

Line Length Relative to Ideal Line Length (LLRILL): Compares actual number of nodes in a line to the ideal line length. The ideal line length L is calculated by

$L = a \cdot log(B + 1) + c$

where a is the scaling factor, B is the number of branching nodes after the final node in the line, and c is the baseline line length. This statistic evaluates how accurately the group aligns with the ideal line length.

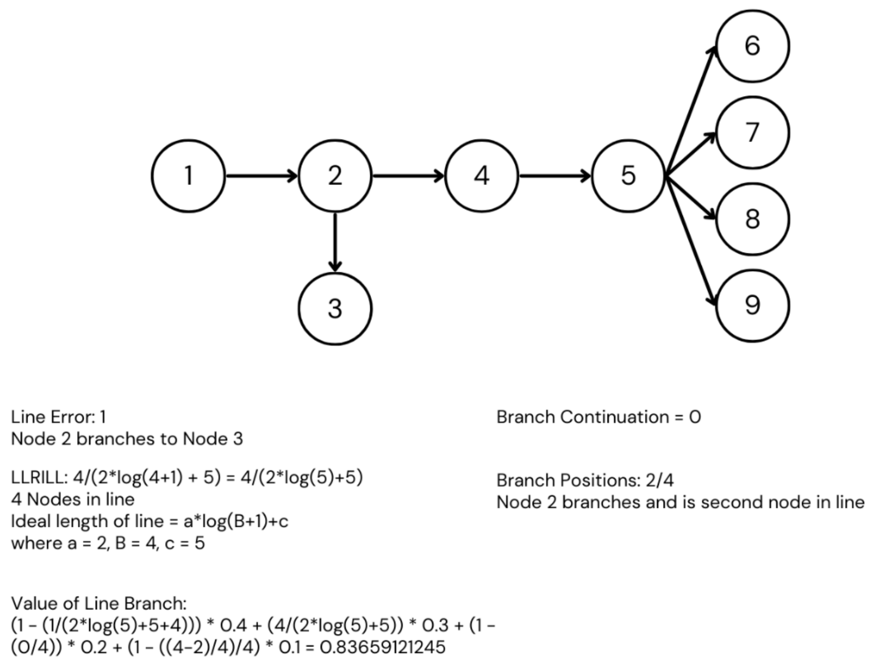
Branch Continuation: Finds whether additional branching occurs after the initial branching from the final node in the line. This statistic determines how closely the group follows the structure of an ideal line branch.

Branch Positions: Finds the positions of nodes in the line that branch early. This statistic will tell the user whether branching occurs, and if so, whether it happens earlier or later in the line branch.

Method to identify line branch:

a. Check for cycles: if a cycle exists in the group, it cannot be a line branch
b. Identify potential ending nodes: create a list of potential ending nodes for each group if no cycle exists for that group. Then initialize most branching nodes and best path (longest path in the group that ends the line with the node that led to most branching nodes) and iterate over each node in the group's potential ending nodes list
    i. Consider current node as the ending node of the line
    ii. Find number of branching nodes ending node has
    iii. If number of branching nodes is greater than most branching nodes, get longest path in the group that ends with the ending node and set best path to the current path and set most branching nodes to number of branching nodes from ending node. If number of branching nodes is same as most branching nodes, get longest path in the group that ends with the ending node. Compare the length of longest path to length of best path. If longest path is longer than best path, then set best path to longest path and most branching nodes to number of branching nodes from ending node
c. Evaluate each best path for each group: apply LBFA on each best path for each group to get statistics and analysis.
d. Output results: identify each group and output the path and statistics that form a line branch closest to an ideal line branch

Example:



Line Error: 1
Node 2 branches to Node 3

LLRILL: 4/(2*log(4+1) + 5) = 4/(2*log(5)+5)
4 Nodes in line
Ideal length of line = a*log(B+1)+c
where a = 2, B = 4, c = 5

Value of Line Branch:
(1 - (1/(2*log(5)+5+4))) * 0.4 + (4/(2*log(5)+5)) * 0.3 + (1 - (0/4)) * 0.2 + (1 - ((4-2)/4)/4) * 0.1 = 0.83659121245

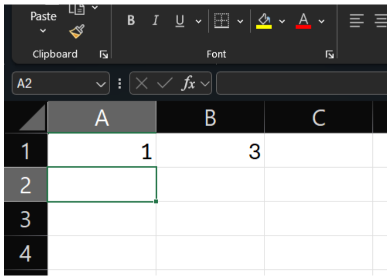
Branch Continuation = 0

Branch Positions: 2/4
Node 2 branches and is second node in line

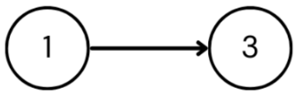
3.2 Deployment Description

This section outlines how to run the program, including organizing the CSV file, understanding the libraries used, and executing the program.

The program requires a CSV file as input to represent the network. The file must follow a specific structure for the data to be properly processed and analyzed. The first column should contain the source nodes, representing the nodes that initiate connections (nodes that point towards other nodes). The second column should contain the target nodes, representing the nodes that receive connections (nodes that are pointed towards). Therefore, each row in the CSV file corresponds to a directed edge in the network, where connections point from the source nodes to target nodes. For example, a row with 1 in the first column and 3 in the second column represents node 1 pointing to node 3. This format for the CSV file must be maintained to ensure no errors occur during the program's execution.



	A	B	C
1	1	3	
2			
3			
4			



The program utilizes several R libraries to carry out data manipulation, graph analysis, and visualization
Tidyverse: R packages that assist with data wrangling, parsing, reading, and writing.
igraph: Allows for graph visualisation and graph analysis
visNetwork: Allows for interactive graph visualization as well as dynamic network diagrams
MASS: R packages that include statistical tools and functions used to analyze data

Steps to run the program (R Markdown File)
1. Prepare the CSV file and specify file path in the program
2. Install required libraries
3. Run each code chunk to view analysis and results

3.3 Information for Next Development Team

This project was done individually and, while it explores a similar topic to that of another team (the project done on Information Spread Models on X), it branches away significantly in its approach and focus. The emphasis here was on visually representing a network and identifying subgraphs within a network.

For more information about this project or any inquiries, contact Colin Song
Email: colin.song@mail.mcgill.ca

IV. Experiment or Test Cases

Describe Experiment and Results

The primary focus of this project was to visually represent the network and identify subgraphs within a network. As a result, extensive testing was not a central component of this project. However, deeper analysis was conducted on cloistered graphs, tree graphs, and line branch graphs. After defining these subgraphs, I evaluated their characteristics to understand the differences in their structures and how these subgraphs influence information flow.

This analysis involved studying the structural properties of each subgraph, using metrics such as connectivity and node distribution. For cloistered graphs, I analyzed the density of internal connections and the role of gatekeeper nodes. Analysis revealed that cloistered graphs often create self-contained cycles of information with limited spread beyond the group. For tree graphs, I examined properties such as height and branching patterns. The analysis showed that the branching factor and depth of the tree significantly influences the spread of information, with bottlenecking at critical nodes slowing the spread of further information. For line branch graphs, I focused on the length of the line and the branching behavior. I found that the linear structure of a line branch allowed for predictable spread of information, however, the line often ended shortly and soon branched into many nodes. The line branch is a structure where the source of the information is less influential than the final user, who interacts with and impacts many of their followers. Although these analyses provide valuable information into the nature of the subgraphs, more testing must be done to further understand the impact of these subgraphs and improve the program's capabilities.

VI. Reflection

6.1 What would you do differently (what have you learned negative)

My primary goals for this project were to develop a program to visually represent a network and implement three algorithms to identify three subgraphs: cloistered graphs, tree graphs, and line branch graphs. While I was successful in achieving these goals, I ended up coding a program of 900 lines. I often found myself getting lost in the complexity of the code as well as the sheer number of variables used in the program. This made the code harder to follow and required me to rewrite certain sections of the program multiple times to fix issues or improve the clarity.

Looking back, I would focus on better organization for not only the coding process but also for the planning process of implementing the three algorithms. Although I had clear definitions and steps for each algorithm, I did not effectively plan out how to translate those steps into code. This created numerous challenges during the development of the program, as I often found myself unsure of how to proceed with the next step. I would have also dedicated more time to designing the structure of the code and planning how to implement the algorithms programmatically. This approach would have relieved much of the stress I experienced from being overwhelmed by the complexity of the code and ensured a smoother implementation process.

6.2 What would you do again (what have you learned positive)

Looking back, one decision I would make again is choosing to code this project in R. R proved to be an excellent choice due to many tools for statistical analysis, data visualization, and graph modeling the software offers. Libraries such as *igraph* and *tidyverse* allowed for constructing and analyzing the network, while *visNetwork* allowed for effective and interactive visualizations. These tools streamlined many aspects of the project and let me focus more on my primary goals.

Another decision I would repeat again is the decision to create my own definitions for the subgraphs—cloistered graphs, tree graphs, and line branch graphs—and to develop properties for analyzing these structures. The definitions and properties I came up with proved to be the foundation of this project and provided a unique perspective for evaluating groups of nodes within a network. By breaking down the steps to analyze each subgraph, I was able to have a better understanding of not only the influences subgraphs have on the network but how information flows. The process of defining these subgraphs provided clarity and a consistent method of analyzing communities.

Finally, the algorithms I designed to identify and classify subgraphs within a network is a decision I would make again. By breaking down the steps systematically, I developed a procedure to identify any community as a specific subgraph, given it matches the properties of the ideal subgraph it is being identified as. Although the algorithms can be improved, they currently succeed in achieving the goals I set out and demonstrated to be very effective when analyzing large complex network structures. These decisions not only contributed to the success of the project but also provides a strong foundation for future development.

Future Work

Things to Fix

One thing to fix involves tuning the parameters used in the program. One example of this is when calculating the ideal line length for a line branch graph. The values of the scaling factor a and the baseline length c in the formula $L = a \cdot log(B + 1) + c$ require further tuning to align with real-world data. Testing numerous datasets would help determine optimal values for these parameters, ensuring the formula better represents realistic line structures. This adjustment would improve the accuracy of the line branch graph analysis.

One major challenge I faced during my research was graphing an entire network. Early on, I attempted to graph a dataset with one million connections, which took the program an hour to complete. However, the resulting graph was unreadable as such a dense network appeared as a blob of colors. Networks with many nodes quickly became visually overwhelming, making it hard to distinguish meaningful patterns or relationships. The only fix I had around this issue was to focus on smaller areas of the network to make graphing and analyzing more manageable.

Lastly, the need for new data remains an unresolved issue. The current project heavily relies on existing datasets, which may limit the scope of analysis and the ability to simulate diverse scenarios. Developing a method to generate or acquire new data—such as creating datasets, simulating user interactions, or collaborating with data providers—would expand the project's capabilities and allow for more robust testing of the algorithms. Access to new and varied data would also enable the program to adapt to evolving network dynamics and remain relevant for future applications.

Features to Add/Future Work

1. Testing and Tuning Parameters

Future development should involve more extensive testing to fine-tune the formula for the ideal line length. The current formula, $L = a \cdot log(B + 1) + c$, uses parameters a (scaling factor) and c (baseline length). I currently have a set to be 2 and c set to be 5, however these values should be adjusted to better reflect real-world network properties.

Weights used when analyzing tree and line branch graphs should also be adjusted. These weights determine whether a group is fit to be a tree or a line branch graph. Due to the significant role these weights play in identifying subgraphs within a network, they must be tested rigorously against large datasets to ensure accurate identification

Experimenting with these parameters can help refine the identification process to ensure it aligns more closely with observed patterns in social media networks. Testing various datasets and edge cases will provide valuable insights into how well the program adapts to different types of networks.

2. Using the Program to Evaluate Information Spread

The program has the potential to simulate and evaluate how information flows in networks. Future teams can build on my program to analyze the flow of information through specific subgraphs, focusing on factors such as:

- Speed of Propagation: How quickly information reaches distant nodes.
- Influence of Key Nodes: Identifying nodes that play pivotal roles in spreading information.
- Impact of Communities: How tightly-knit groups influence or impede the flow of information.

These factors can be extended to simulate real-world scenarios, such as tracking viral tweets or examining how misinformation spreads. By using the program to analyze real data, the algorithms could be tested in more practical scenarios and reveal how information flows when subgraphs are identified. This could include studying how tightly connected cloistered graphs gatekeep information, how hierarchical tree structures spread information, and how line branch graphs influence the reach and longevity of information. Analyzing real data would uncover patterns in information spread that are unique to real-world networks.

3. Adding More Properties to Subgraphs

To further solidify the definitions of each subgraph, additional properties and metrics can be incorporated into the analysis. Examples include:

- Clustering Coefficient: Measure how tightly nodes are connected within a subgraph.
- Edge Weight Analysis: Include weights on edges to represent the strength of relationships or interaction frequencies.
- Centrality Metrics: Analyze measures such as betweenness or eigenvector centrality to better understand the roles of nodes within subgraphs.

Improving the subgraph properties will provide a more comprehensive understanding of their structure and behavior, enabling more accurate identification and deeper analysis. While the current metrics provide a strong foundation, adding more properties would help solidify the definitions of cloistered graphs, tree graphs, and line branch graphs. Incorporating metrics could provide deeper insights into the behavior of each subgraph. This would not only improve the identification process but also offer a better understanding of how these subgraphs function within larger networks.

Suggestions

1. Plan ahead: Before starting, organize the steps required to complete the project and outline each task
2. Research: Take the time to read articles to become familiar with the topics involved in the project
3. Focus on your goals: Always relate your work back to the project's main objective

VII. Supplemental Materials

During the course of this project, several materials and resources were studied that were ultimately not used for the final implementation of my research. However, this information could be valuable for future work. These include unused statistical metrics and hydrating tweets using Twarc.

Unused Statistical Metrics

In the process of analyzing subgraphs, I gathered information about several additional statistical metrics that were not incorporated into the project. These metrics could provide deeper insights into network behavior and help refine subgraph analysis.

1. Modularity
   - Compares edges inside communities to edges between communities. Quantifies how well a graph can be divided into separate communities

2. Number of Communities
   - Total number of distinct communities in network. Large number of communities may indicate highly fragmented network. Small may indicate more centralized structure

3. Distribution of Community Size
   - Distribution of sizes of detected communities. Shows presence of large communities vs. small

4. Average # of Edges in Community
   - Average number of edges in communities. Shows how well-connected per node in community

5. Number of In-Degrees and Out-Degrees
   - Number of edges coming to a vertex and number of edges coming out of vertex. Can find users who are at central areas within community

6. Conductance
   - Ratio of in-degrees to total number of edges. Lower conductance indicates more tightly-knit, well-defined community, higher indicates community has more connections outside

7. Conductance of Whole Network
   - Sum of conductance values of all communities. Measures how well network is divided into communities. Lower value suggests well-defined community structures

8. Edge Density
   - Ratio of number of edges to max number of possible edges. An idea of how dense a graph is in terms of edge connectivity

9. Clustering Coefficient
   - The degree to which nodes in a community tend to cluster together. High coefficient suggests that nodes form tight-knit groups

10. Eigenvector Centrality
    - Finds centrality for a node, proportional to other nodes. Nodes that are pointed to by "important" nodes will also be considered "important"

11. Betweenness Centrality
    - Measure of centrality based on shortest path. Node with vlaue will have more information passed through

12. Intra-Community/Inter-Community Interactions
    - Ratio of edges within communities to edges between communities. High ratios suggest strong community boundaries, low ratios suggest overlap/interaction between communities

13. Overlapping Communities
    - Identifies nodes that belong to multiple communities. Reveals users that participate in various groups

Hydrate Tweets

Another supplemental material involves the use of *Twarc*, a Python library for working with Twitter data. *Twarc* allows for hydrating tweet IDs, which retrieves the full metadata of tweets from their IDs (this requires an X Developer Account and access to the X API). While this approach was not utilized in this project due to limited API call rates, I invested considerable time exploring it and believed it had the potential to provide valuable datasets.

References and Background Material (accumulated)

Communities. Communities - NetworkX 3.4.2 documentation. (n.d.). https://networkx.org/documentation/stable/reference/algorithms/community.html

Datta, S. S. (2024, March 18). *Graph density*. Baeldung on Computer Science. https://www.baeldung.com/cs/graph-density#:~:text=Graph%20density%20represents%20the%20ratio,new%20edges%20to%20the%20network

Du, D. (n.d.). *Social Network Analysis: Lecture 3-network characteristics*. Social Network Analysis. https://ddu.ext.unb.ca/6634/Lecture_notes/Lec3_network_statistics_handout.pdf

Eigenvector_centrality. eigenvector_centrality - NetworkX 3.4.2 documentation. (n.d.). https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.eigenvector_centrality.html

Finding community structure by multi-level optimization of modularity. R igraph manual pages. (n.d.-a). https://igraph.org/r/doc/cluster_louvain.html

GeeksforGeeks. (2022, July 21). *Betweenness centrality (centrality measure)*. https://www.geeksforgeeks.org/betweenness-centrality-centrality-measure/

GeeksforGeeks. (2024, March 20). *Difference between graph and tree*. https://www.geeksforgeeks.org/difference-between-graph-and-tree/

Junker, N. (2020, March 4). *Community detection with Louvain and Infomap: R-bloggers*. R-bloggers. https://www.r-bloggers.com/2020/03/community-detection-with-louvain-and-infomap/

Junker, N. (2022, August 17). *Interactive network visualization with R*. statworx. https://www.statworx.com/en/content-hub/blog/interactive-network-visualization-with-r/

Lizardo, O., & Jilbert, I. (2020, January 6). *2.7 average degree: Social Networks: An introduction*. Social Networks: An Introduction. https://bookdown.org/omarlizardo/_main/2-7-average-degree.html

Louf, B., McDiarmid, C., & Skerman, F. (n.d.). *Modularity and graph expansion - drops*. Modularity and Graph Expansion. https://drops.dagstuhl.de/storage/00lipics/lipics-vol287-itcs2024/LIPIcs.ITCS.2024.78/LIPIcs.ITCS.2024.78.pdf

Modularity. modularity - NetworkX 3.4.2 documentation. (n.d.). https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.quality.modularity.html#r6937dc4d2017-4

Newman, M. (n.d.). *Modularity and community structure in networks*. Proceedings of the National Academy of Sciences. https://www.pnas.org/doi/full/10.1073/pnas.0601602103

Pisharody, A. (2022, March 8). *Learn TWARC!*. Twarc Command Basics · Learn Twarc! https://scholarslab.lib.virginia.edu/learn-twarc/06-twarc-command-basics

Popova, K. (2024, June 5). *Network analysis and community detection on political tweets*. Medium. https://medium.com/@pop.kristina1/network-analysis-and-community-detection-on-political-tweets-9e0f21294e31

R igraph manual pages. (n.d.-b). https://igraph.org/r/doc/communities.html

Terzi, E. (n.d.). *Graph clustering*. CAS CS 565, Data Mining. https://cs-people.bu.edu/evimaria/cs565-16/lect10.pdf

Wolitzky, A. (n.d.). *Lecture 3: Eigenvector centrality measures*. MIT 14.15 Networks. https://ocw.mit.edu/courses/14-15-networks-spring-2022/mit14_15s22_lec3.pdf

Databases:

Higgs twitter dataset. SNAP. (n.d.-a). https://snap.stanford.edu/data/higgs-twitter.html

Social circles: Twitter. SNAP. (n.d.-b). https://snap.stanford.edu/data/ego-Twitter.html

Stanford Large Network Dataset Collection. SNAP. (n.d.-c). https://snap.stanford.edu/data/index.html#twitter

Twitter Interaction Network for the US congress. SNAP. (n.d.-d). https://snap.stanford.edu/data/congress-twitter.html

Databases Not Used (not APA):

https://github.com/shaypal5/awesome-twitter-data?tab=readme-ov-file#twitter-datasets

https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi%3A10.7910%2FDVN%2FPDI7IN

https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/LW0BTB

https://data.world/socialmediadata/j-k-rowling-tweets-and-retweets/workspace/project-summary?agentid=socialmediadata&datasetid=j-k-rowling-tweets-and-retweets

https://github.com/echen102/COVID-19-TweetIDs