part3-intrographanalytics

Matteo Sanchez-Dahl, Latifa Al Jlayel

2022-09-25

# Chapter 3. Results

## 3.1 Introduction to Graph Analytics Outputs

### Import Data

Here we import the data in the same manner as before.

nodes <- as.matrix(read.table('data\_subelj\_jdk/ent\_subelj\_jdk\_jdk\_class\_name.txt'))  
edges <- as.matrix(read.table('data\_subelj\_jdk/out\_subelj\_jdk\_jdk.txt', sep = "", skip = 2))  
edges

## V1 V2  
## [1,] 1 2  
## [2,] 1 3  
## [3,] 1 4  
## [4,] 1 5  
## [5,] 1 5  
## [6,] 1 6  
## [7,] 1 4  
## [8,] 1 5  
## [9,] 1 4  
....

Here we convert the edge list to an igraph object by using the **graph\_from\_edgelist( )** function. We pass in the edge list and indicate that the graph is directed. Using th information from the simplification process, we create subgraphs of the 44 nodes that have a degree greater than 500. We make one version of the graph with numbers as vertex names and another with the corresponding classes.

g\_num <- graph\_from\_edgelist(edges, directed = TRUE)  
net\_num <- induced\_subgraph(g\_num, igraph::V(g\_num)[igraph::degree(g\_num)>500])  
  
# convert the edgelist to an igraph object  
g <- graph\_from\_edgelist(edges, directed = TRUE)  
  
# named nodes  
V(g)$name <- nodes  
  
# create a graph of the simplified network  
net <- induced\_subgraph(g, igraph::V(g)[igraph::degree(g)>500])  
net

## IGRAPH 94bb0b9 DN-- 44 1228 --   
## + attr: name (v/c)  
## + edges from 94bb0b9 (vertex names):  
## [1] java.awt.Dimension->java.lang.String   
## [2] java.awt.Dimension->java.lang.Object   
## [3] java.awt.Dimension->java.lang.Object   
## [4] java.awt.Dimension->java.lang.Class   
## [5] java.awt.Dimension->java.io.Serializable  
## [6] java.lang.String ->java.util.Locale   
## [7] java.lang.String ->java.util.Locale   
## [8] java.lang.String ->java.util.Locale   
## + ... omitted several edges

Here we use the **V( )** function to create a vertex sequence containing all vertices of the graph. The results confirm the 44 vertices we selected.

# get the vertices of a graph  
V(net)

## + 44/44 vertices, named, from 94bb0b9:  
## [1] java.awt.Dimension java.lang.String   
## [3] java.util.Locale java.awt.Image   
## [5] java.awt.Component java.lang.Object   
## [7] java.io.PrintWriter java.io.PrintStream   
## [9] java.awt.Graphics java.beans.PropertyChangeListener   
## [11] java.awt.Font java.lang.Class   
## [13] java.awt.Point java.awt.Event   
## [15] java.util.Set java.awt.Container   
## [17] java.awt.ComponentOrientation java.awt.Insets   
## [19] java.awt.Color java.awt.Cursor   
## + ... omitted several vertices

Here we use the **E( )** function to create an edge sequence containing all the edges in the graph. The results indicate that the graph contains more than one edge between nodes, as we desired.

# get the edges of a graph  
E(net)

## + 1228/1228 edges from 94bb0b9 (vertex names):  
## [1] java.awt.Dimension->java.lang.String   
## [2] java.awt.Dimension->java.lang.Object   
## [3] java.awt.Dimension->java.lang.Object   
## [4] java.awt.Dimension->java.lang.Class   
## [5] java.awt.Dimension->java.io.Serializable  
## [6] java.lang.String ->java.util.Locale   
## [7] java.lang.String ->java.util.Locale   
## [8] java.lang.String ->java.util.Locale   
## [9] java.lang.String ->java.lang.Object   
## [10] java.lang.String ->java.lang.Object   
## + ... omitted several edges

Here we use **as\_adjacency\_matrix( )** function to get the adjacency matrix of the graph.

# get the adjacency matrix  
net.adj <- as\_adjacency\_matrix(net)  
net.adj

## 44 x 44 sparse Matrix of class "dgCMatrix"

## [[ suppressing 44 column names 'java.awt.Dimension', 'java.lang.String', 'java.util.Locale' ... ]]

##   
## java.awt.Dimension . 1 . . . 2 . . . . . 1 . . . . . .  
## java.lang.String . . 3 . . 4 . . . . . 1 . . . . . .  
## java.util.Locale . 20 . . . 3 . . . . . 1 . . . . . .  
## java.awt.Image . 2 . . . 4 . . 1 . . 1 . . . . . .  
## java.awt.Component 14 12 2 7 . 6 2 2 6 4 3 2 9 14 2 3 3 .  
## java.lang.Object . 1 . . . . . . . . . 1 . . . . . .  
## java.io.PrintWriter . 13 2 . . 3 . . . . . 1 . . . . . .  
## java.io.PrintStream . 12 2 . . 3 . . . . . 1 . . . . . .  
## java.awt.Graphics . 2 . 6 . 2 . . . . 3 1 . . . . . .  
....

### Analytic Functions

#### Density

Here we use the **gden( )** function to compute the density of the graph.

An error we encountered here is that the input must be an adjacency matrix/array, edgelist matrix, network, or sparse matrix, or list thereof. Thus, we used the **as\_edgelist( )** function to convert the graph to an edge list.

Another error we encountered when using the graph that has the classes assigned as vertex names is that one of the functions that **gden( )** utilizes doesn’t accept character type variables as arguments.

Ultimately, we passed the edge list of numbers into **gden( )**, and the string “digraph” to indicate that the edges of the graph should be interpreted as directed.

# create edge list  
# calculate density of directed graph  
net.density = gden(as\_edgelist(net\_num), mode = "digraph")  
net.density

## [1] 0.06168145

### Edge Density

Here we use the **edge\_density( )** function to calculate the density of the graph, which is the ratio of the number of edges and the number of possible edges.

# edge density   
igraph::edge\_density(net)

## [1] 0.6490486

Here we explicitly state that self loops should be considered to be possible.

# remove self-loops  
igraph::edge\_density(net, loops = T)

## [1] 0.6342975

### Egocentric Network Analysis

Here we use the **ego.extract( )** function to obtain a list containing the adjacency matrices for the ego nets of each vertex in the graph.

An error we encountered was that the input must be an adjacency matrix/array, network, or list. Thus, we used the **as\_edgelist( )** function to convert the graph to an edge list.

# egocentric network  
net.ego <- sna::ego.extract(as\_edgelist(net))

Here we print the matrix for vertex 1.

net.ego[1]

## $`1`  
## [,1] [,2] [,3]   
## [1,] "0" "java.awt.Dimension" "java.lang.String"  
## [2,] "java.awt.Dimension" "0" "0"   
## [3,] "java.lang.String" "0" "0"

Here we print the matrix for vertex 44.

net.ego[44]

## $`44`  
## [,1] [,2] [,3]   
## [1,] "0" "java.awt.Image" "java.lang.Object"  
## [2,] "java.awt.Image" "0" "0"   
## [3,] "java.lang.Object" "0" "0"

### Degree

Here we use the **degree( )** function to obtain the degree of each vertex in the graph, i.e. the number of adjacent edges

# degree  
igraph::degree(net)

## java.awt.Dimension java.lang.String   
## 81 280   
## java.util.Locale java.awt.Image   
## 51 53   
## java.awt.Component java.lang.Object   
## 205 140   
## java.io.PrintWriter java.io.PrintStream   
## 30 29   
## java.awt.Graphics java.beans.PropertyChangeListener   
## 74 20   
....

### Betweeness Centrality

Here we use the **centr\_betw( )** function to obtain a named list with the following components: **res**, the node-level centrality scores; **centralization**, the graph level centrality index; and **theoretical\_max**, the maximum theoretical graph level centralization score for a graph with the given number of vertices, using the same parameters.

# betweeness centrality  
igraph::centr\_betw(net)

## $res  
## [1] 0.37070376 42.95808676 0.33874795 13.00000000 40.61570474 15.60571429  
## [7] 0.10526316 0.10526316 24.35853210 0.00000000 5.18043508 36.41675470  
## [13] 0.49813024 0.35591926 9.00000000 17.61842105 0.07626841 0.10229133  
## [19] 0.53342881 0.05084561 19.90076570 0.00000000 0.00000000 0.00000000  
## [25] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.56666667  
## [31] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000  
## [37] 0.00000000 27.24205723 2.00000000 0.00000000 0.00000000 0.00000000  
## [43] 0.00000000 0.00000000  
##   
## $centralization  
## [1] 0.0210301  
##   
## $theoretical\_max  
## [1] 77658

### Closeness Centrality

Here we use the **centr\_clo( )** function to obtain a named list with the following components: **res**, the node-level centrality scores; **centralization**, the graph level centrality index; and **theoretical\_max**, the maximum theoretical graph level centralization score for a graph with the given number of vertices, using the same parameters.

# closeness centrality  
igraph::centr\_clo(net)

## $res  
## [1] 0.7500000 0.8333333 0.8333333 0.5600000 0.8666667 0.6250000 0.7500000  
## [8] 0.7500000 0.7368421 NaN 1.0000000 0.8333333 0.7500000 0.7500000  
## [15] 0.4666667 0.8666667 0.8571429 0.7500000 0.6666667 0.7500000 0.8181818  
## [22] NaN 0.3684211 NaN 0.7142857 0.8000000 NaN 0.5192308  
## [29] NaN 0.4736842 0.4285714 0.4285714 0.6666667 0.5833333 0.5454545  
## [36] 0.5000000 0.4909091 0.8888889 0.8888889 0.6666667 0.8918919 0.5600000  
## [43] 0.5076923 0.7777778  
##   
## $centralization  
## [1] NaN  
##   
## $theoretical\_max  
## [1] 42.02273

### Shortest Paths

Here we use the \*\* \*\* function to obtain the shortest path between any two nodes in the graph.

## shortest paths  
net.sp <- igraph::shortest.paths(net)  
net.sp

## java.awt.Dimension java.lang.String  
## java.awt.Dimension 0 1  
## java.lang.String 1 0  
## java.util.Locale 2 1  
## java.awt.Image 2 1  
## java.awt.Component 1 1  
## java.lang.Object 1 1  
## java.io.PrintWriter 2 1  
## java.io.PrintStream 2 1  
## java.awt.Graphics 2 1  
....

Here we use the **get.shortest.paths( )** function to obtain the actual shortest paths from a selected vertex We arbitrarily chose vertex 1.

# get the actual paths (and not just their length)   
igraph::get.shortest.paths(net,1)

## Warning in igraph::get.shortest.paths(net, 1): At core/paths/unweighted.c:368 :  
## Couldn't reach some vertices.

## $vpath  
## $vpath[[1]]  
## + 1/44 vertex, named, from 94bb0b9:  
## [1] java.awt.Dimension  
##   
## $vpath[[2]]  
## + 2/44 vertices, named, from 94bb0b9:  
## [1] java.awt.Dimension java.lang.String   
##   
## $vpath[[3]]  
....

### Geodesic

Here we use the **geodist( )** function

An error we encountered was that the input must be an adjacency matrix/array, edgelist matrix, network, or sparse matrix, or list thereof. Thus, we used the **as\_edgelist( )** function to convert the graph to an edge list. We used the net version with the vertex names as numbers to prevent the Warning: NAs introduced by coercion.

# geodist  
net\_num.gd <- geodist(as\_edgelist(net\_num))  
net\_num.gd

## $counts  
## [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]  
## [1,] 1 2 2 2 2 2 2 2 2 2 2 2 2  
## [2,] 2 1 2 2 2 2 2 2 2 2 2 2 2  
## [3,] 2 2 1 2 2 2 2 2 2 2 2 2 2  
## [4,] 2 2 2 1 2 2 2 2 2 2 2 2 2  
## [5,] 2 2 2 2 1 2 2 2 2 2 2 2 2  
## [6,] 2 2 2 2 2 1 2 2 2 2 2 2 2  
## [7,] 2 2 2 2 2 2 1 2 2 2 2 2 2  
## [8,] 2 2 2 2 2 2 2 1 2 2 2 2 2  
....

### Number of Paths

Here we use multiply the adjacency matrix of the graph by itself to find the number of paths between two nodes

net.np <- net.adj%\*%net.adj  
net.np

## 44 x 44 sparse Matrix of class "dgCMatrix"

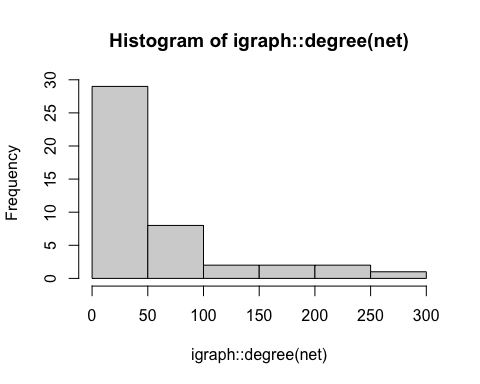
## [[ suppressing 44 column names 'java.awt.Dimension', 'java.lang.String', 'java.util.Locale' ... ]]

##   
## java.awt.Dimension . 14 3 . . 10 . . . . .  
## java.lang.String . 76 . . . 15 . . . . .  
## java.util.Locale . 15 60 . . 86 . . . . .  
## java.awt.Image . 18 6 9 . 16 . . . . 3  
## java.awt.Component 62 322 59 62 57 353 6 6 31 12 27  
## java.lang.Object . 12 3 . . 10 . . . . .  
## java.io.PrintWriter . 55 39 . . 64 . . . . .  
## java.io.PrintStream . 55 36 . . 60 . . . . .  
## java.awt.Graphics 16 126 12 6 . 64 . . 6 . .  
....

### Histogram of Degree of the Nodes

Here we use the **hist( )** function to plot a histogram of the degrees of the nodes in the graph.

# histogram of the degree of the nodes  
hist(igraph::degree(net))



### rgraph Example Functions

Here we use the **edge\_density( )** function to calculate the density of the graph, which is the ratio of the number of edges and the number of possible edges.

# edge density   
igraph::edge\_density(net)

## [1] 0.6490486

Here we explicitly state that self loops should be considered to be possible.

# remove self-loops  
igraph::edge\_density(net, loops = T)

## [1] 0.6342975

Here we use the **diameter( )** function to obtain the diameter of the graph, i.e. the length of the longest geodesic.

# diameter  
net.d <- igraph::diameter(net)  
net.d

## [1] 4

Here we use the **max\_cliques( )** function to find the max-cliques for node 13.

node13 <- c(13)  
net.13clique <- igraph::max\_cliques(net, min = NULL, subset = node13)

## Warning in igraph::max\_cliques(net, min = NULL, subset = node13): At core/  
## cliques/maximal\_cliques\_template.h:269 : Edge directions are ignored for maximal  
## clique calculation.

# At core/cliques/maximal\_cliques\_template.h:269 : Edge directions are ignored for maximal clique calculation.  
net.13clique

## [[1]]  
## + 7/44 vertices, named, from 94bb0b9:  
## [1] java.io.Serializable java.lang.String javax.swing.JComponent  
## [4] java.lang.Class java.awt.Component java.util.Locale   
## [7] java.awt.Font   
##   
## [[2]]  
## + 7/44 vertices, named, from 94bb0b9:  
## [1] java.io.Serializable java.lang.String   
## [3] javax.swing.JComponent java.lang.Class   
....

Here we use the **clique\_num( )** function to find the size of the largest cliques.

# largest cliques  
net.lgcliques <- igraph::clique\_num(net)

## Warning in igraph::clique\_num(net): At core/cliques/  
## maximal\_cliques\_template.h:269 : Edge directions are ignored for maximal clique  
## calculation.

# At core/cliques/maximal\_cliques\_template.h:269 : Edge directions are ignored for maximal clique calculation.  
net.lgcliques

## [1] 11

Exercise: Try to identify the cliques in the plot. Write them down.

### erdos.renyi.game Example Functions

Here we check the names of the vertices and confirm they’re named with their corresponding classes.

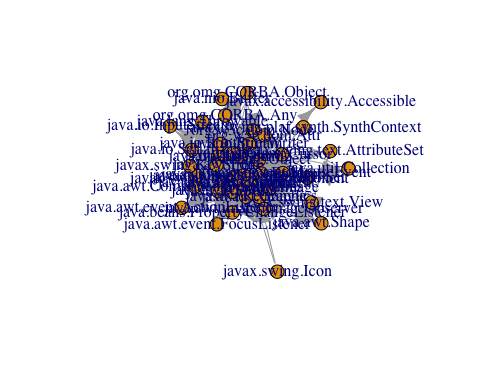
# name vertices  
V(net)$name

## [1] "java.awt.Dimension" "java.lang.String"   
## [3] "java.util.Locale" "java.awt.Image"   
## [5] "java.awt.Component" "java.lang.Object"   
## [7] "java.io.PrintWriter" "java.io.PrintStream"   
## [9] "java.awt.Graphics" "java.beans.PropertyChangeListener"   
## [11] "java.awt.Font" "java.lang.Class"   
## [13] "java.awt.Point" "java.awt.Event"   
## [15] "java.util.Set" "java.awt.Container"   
## [17] "java.awt.ComponentOrientation" "java.awt.Insets"   
## [19] "java.awt.Color" "java.awt.Cursor"   
## [21] "java.awt.Rectangle" "java.awt.event.FocusListener"   
## [23] "java.awt.image.ImageObserver" "javax.accessibility.Accessible"   
## [25] "java.io.InputStream" "java.lang.Throwable"   
## [27] "java.awt.event.ActionListener" "java.awt.event.MouseEvent"   
## [29] "java.io.Serializable" "java.awt.Shape"   
## [31] "java.util.Collection" "javax.swing.text.AttributeSet"   
## [33] "java.nio.Buffer" "org.w3c.dom.Attr"   
## [35] "org.w3c.dom.Node" "org.omg.CORBA.Object"   
## [37] "javax.swing.Icon" "javax.swing.JComponent"   
## [39] "javax.swing.JPopupMenu" "javax.swing.KeyStroke"   
## [41] "javax.swing.text.JTextComponent" "javax.swing.text.View"   
## [43] "javax.swing.plaf.synth.SynthContext" "org.omg.CORBA.Any"

### astrocollab Part 1 Example Functions

Here we create a copy of the graph to manipulate for this section and use the **plot( )** function to plot the graph.

# copy graph to manipulate  
net\_astro <- net  
  
#plot  
plot(net\_astro)



Here we use the **degree( )** function to obtain the degree of each vertex in the graph, i.e. the number of adjacent edges

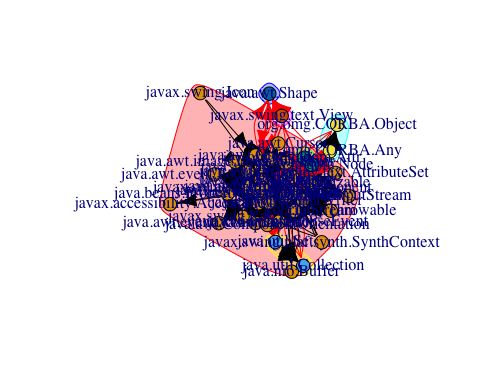
# degree  
igraph::degree(net\_astro)

## java.awt.Dimension java.lang.String   
## 81 280   
## java.util.Locale java.awt.Image   
## 51 53   
## java.awt.Component java.lang.Object   
## 205 140   
## java.io.PrintWriter java.io.PrintStream   
## 30 29   
## java.awt.Graphics java.beans.PropertyChangeListener   
## 74 20   
....

#### Communities

Here we use the \*\*\*\* function

# communities  
wc <- walktrap.community(net\_astro)  
plot(wc, net\_astro, vertex.size = 15, layout = layout.fruchterman.reingold)



#### Simplify

Here we use the **rnorm( )** function to generate random deviates from a normal distribution with n observations, where n is the total number of edges in the graph; the result of which is obtained from the **ecount( )** function. The random deviates are assigned to the weights of the edges in the graph.

Similarly, the **rnorm( )** function is utilized to generate random deviates from a normal distribution with n observations, where n is the total number of vertices in the graph; the result of which is obtained from the **vcount( )** function. The random deviates are assigned to the weights of the vertices in the graph.

# simplify the graph by removing empty entries from the matrix   
E(net\_astro)$weight <- rnorm(ecount(net\_astro))  
V(net\_astro)$weight <- rnorm(vcount(net\_astro))  
net\_astro[1:5, 1:9]

## 5 x 9 sparse Matrix of class "dgCMatrix"  
## java.awt.Dimension java.lang.String java.util.Locale  
## java.awt.Dimension . -0.1991224 .   
## java.lang.String . . 0.8697687  
## java.util.Locale . 10.5180645 .   
## java.awt.Image . 1.8654239 .   
## java.awt.Component -1.200021 -3.7101374 -0.3926849  
## java.awt.Image java.awt.Component java.lang.Object  
## java.awt.Dimension . . -1.3263924  
## java.lang.String . . 0.9643318  
## java.util.Locale . . -3.7978051  
## java.awt.Image . . -0.5005685  
## java.awt.Component 1.259559 . 0.8859188  
## java.io.PrintWriter java.io.PrintStream java.awt.Graphics  
## java.awt.Dimension . . .   
## java.lang.String . . .   
## java.util.Locale . . .   
## java.awt.Image . . 0.4105971  
## java.awt.Component 2.437817 0.245086 -0.7443475

Here we can see the weights assigned to the vertices.

V(net\_astro)$weight

## [1] 1.21329431 -0.14691538 0.34801559 -0.97677315 -0.65176637 -0.74393845  
## [7] 0.85622746 1.08899228 1.94923868 0.03367192 -0.44327094 0.21843879  
## [13] -0.70874931 0.67937144 0.75407984 -0.52794770 1.23192694 0.74979920  
## [19] -0.06171094 0.12007761 -1.19642275 -0.34675526 -0.94666274 1.34036193  
## [25] -0.29625831 -0.36917159 -0.70918026 -1.08833756 -0.36200644 1.19468576  
## [31] 0.31557438 -1.11392797 0.52902740 0.32210653 -0.30831418 -0.39633701  
## [37] -0.40242658 1.17951438 -0.60978745 2.19079818 0.30828869 -0.22677041  
## [43] 0.75312025 1.28563793

Here we use the **induced.subgraph( )** function to create a subgraph with only the vertices specified in the parameter. The **which( )** function is utilized to select the vertices from the graph with positive weights.

In the rubric, the limit for the vertex weight is 2.2, however in this graph there are no vertices with weights that meet that criteria. Thus we set the limit to 0.0 to visualize the resulting plot.

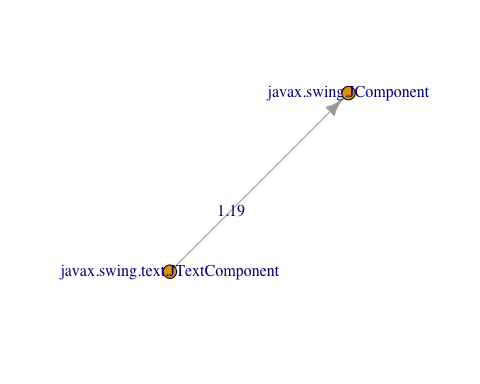
sg <- induced.subgraph(net\_astro, which(V(net\_astro)$weight > 0.0))

In the rubric, the graph is simplified by eliminating empty entries from the matrix, however, in our case this was already done in the initial simplification step. From the earlier histogram of the degree of the nodes in the graph, we can see that the most of the nodes are of a degree less than 50. Thus we chose to eliminate those in this step.

An error that we encountered was that **delete.vertices( )** requires an argument of class network, we instead utilized **delete\_vertices( )** which accepts arguments of type igraph. We label the edges with their values rounded to three decimal places.

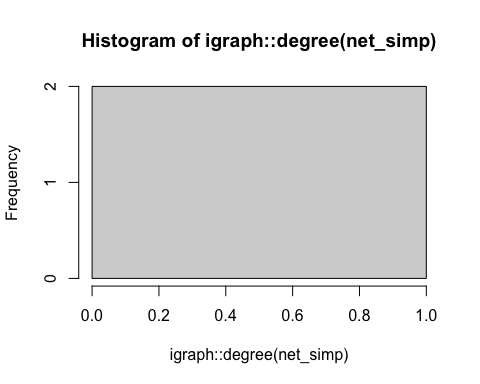
All edge weights were ignored during graph layout because non-positive edge weights were found.

# plot(delete.vertices(sg, igraph::degree(sg)==0), edge.label = round(E(sg)$weight,3))  
# Error in delete.vertices(sg, igraph::degree(sg) == 1) :   
# delete.vertices requires an argument of class network.  
net\_simp <- delete\_vertices(sg, igraph::degree(sg)<50)  
plot(net\_simp, edge.label = round(E(sg)$weight,3))



Here is the histogram of the degree of the remaining nodes. We can conclude these 4 nodes are some of the most important nodes in the software class dependency network of the JDK 1.6.0.7 framework since they remain after simplifying the graph with the weights of the vertices. They also have some of the highest degree centrality. This makes sense since a String class is needed to compile the source code into executable files with binary machine code and Java is an OOP language, which means the Object class is key to its functionality.

hist(igraph::degree(net\_simp))



### NCA Football Conference Example Functions

Here we create a copy of the graph original to manipulate for this section.

# copy graph to manipulate  
net\_fb <- net

Here we use the **vertex\_attr( )** function to query the vertex attributes of the graph.

vertex\_attr(net\_fb)

## $name  
## [1] "java.awt.Dimension" "java.lang.String"   
## [3] "java.util.Locale" "java.awt.Image"   
## [5] "java.awt.Component" "java.lang.Object"   
## [7] "java.io.PrintWriter" "java.io.PrintStream"   
## [9] "java.awt.Graphics" "java.beans.PropertyChangeListener"   
## [11] "java.awt.Font" "java.lang.Class"   
## [13] "java.awt.Point" "java.awt.Event"   
## [15] "java.util.Set" "java.awt.Container"   
## [17] "java.awt.ComponentOrientation" "java.awt.Insets"   
## [19] "java.awt.Color" "java.awt.Cursor"   
## [21] "java.awt.Rectangle" "java.awt.event.FocusListener"   
## [23] "java.awt.image.ImageObserver" "javax.accessibility.Accessible"   
## [25] "java.io.InputStream" "java.lang.Throwable"   
## [27] "java.awt.event.ActionListener" "java.awt.event.MouseEvent"   
## [29] "java.io.Serializable" "java.awt.Shape"   
## [31] "java.util.Collection" "javax.swing.text.AttributeSet"   
## [33] "java.nio.Buffer" "org.w3c.dom.Attr"   
## [35] "org.w3c.dom.Node" "org.omg.CORBA.Object"   
## [37] "javax.swing.Icon" "javax.swing.JComponent"   
## [39] "javax.swing.JPopupMenu" "javax.swing.KeyStroke"   
## [41] "javax.swing.text.JTextComponent" "javax.swing.text.View"   
## [43] "javax.swing.plaf.synth.SynthContext" "org.omg.CORBA.Any"

Here we use the **as\_adjacency\_matrix( )** function to obtain the adjacency matrix reprsesentation of the graph.

adjmatrix <- as\_adjacency\_matrix(net\_fb)  
adjmatrix

## 44 x 44 sparse Matrix of class "dgCMatrix"

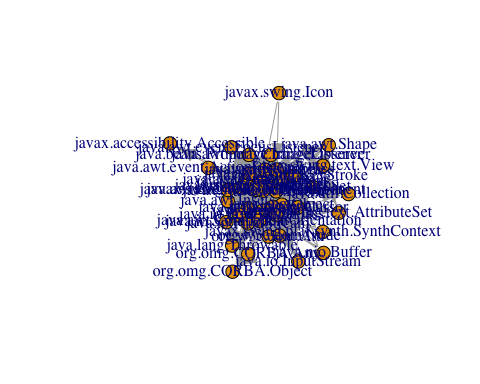
## [[ suppressing 44 column names 'java.awt.Dimension', 'java.lang.String', 'java.util.Locale' ... ]]

##   
## java.awt.Dimension . 1 . . . 2 . . . . . 1 . . . . . .  
## java.lang.String . . 3 . . 4 . . . . . 1 . . . . . .  
## java.util.Locale . 20 . . . 3 . . . . . 1 . . . . . .  
## java.awt.Image . 2 . . . 4 . . 1 . . 1 . . . . . .  
## java.awt.Component 14 12 2 7 . 6 2 2 6 4 3 2 9 14 2 3 3 .  
## java.lang.Object . 1 . . . . . . . . . 1 . . . . . .  
## java.io.PrintWriter . 13 2 . . 3 . . . . . 1 . . . . . .  
## java.io.PrintStream . 12 2 . . 3 . . . . . 1 . . . . . .  
## java.awt.Graphics . 2 . 6 . 2 . . . . 3 1 . . . . . .  
....

Here we use the **plot( )** function to plot the graph.

The graph is difficult to read due to obscruction.

#plot  
plot(net\_fb)



Here we use the **rnorm( )**, **ecount( )**, and **vcount( )** functions to assign weights to the vertices and edges of the graph in a similar fashion as before.

# simplify the graph by removing empty entries from the matrix   
E(net\_fb)$weight <- rnorm(ecount(net\_fb))  
V(net\_fb)$weight <- rnorm(vcount(net\_fb))  
net\_fb[1:5, 1:9]

## 5 x 9 sparse Matrix of class "dgCMatrix"  
## java.awt.Dimension java.lang.String java.util.Locale  
## java.awt.Dimension . 0.6398317 .   
## java.lang.String . . 0.1834898  
## java.util.Locale . 8.1880790 .   
## java.awt.Image . -0.7099989 .   
## java.awt.Component -7.399748 -1.5300949 0.4248419  
## java.awt.Image java.awt.Component java.lang.Object  
## java.awt.Dimension . . -0.7986706  
## java.lang.String . . 3.0216211  
## java.util.Locale . . 0.6747668  
## java.awt.Image . . -2.9158474  
## java.awt.Component -1.434837 . 0.3215144  
## java.io.PrintWriter java.io.PrintStream java.awt.Graphics  
## java.awt.Dimension . . .   
## java.lang.String . . .   
## java.util.Locale . . .   
## java.awt.Image . . 0.3268046  
## java.awt.Component 0.6406312 3.028117 -0.3947206

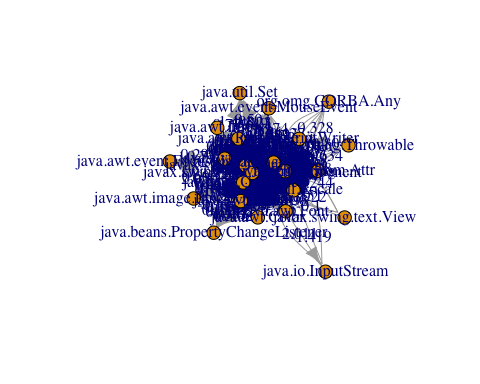
Here we use the **induced.subgraph( )** function to create a subgraph with only the vertices specified in the parameter. The **which( )** function is utilized to select the vertices from the graph with positive weights.

In the rubric, the limit for the vertex weight is 0.7, however we set the limit to 0.0 to select all vertices with positive weights.

We label the edges with their values rounded to three decimal places.

All edge weights were ignored during graph layout because non-positive edge weights were found.

sg <- induced.subgraph(net\_fb, which(V(net\_fb)$weight > 0.0))  
plot(sg, edge.label = round(E(sg)$weight, 3))



Here we use the **is.simple( )** function to check if our graph is simple.

The results confirm what we already know, that the graph is not simple, as we selected to keep multiple edges between nodes.

is.simple(net\_fb)

## [1] FALSE

Here we use the **simplify( )** function to simplify the graph.

# simplify the graph  
net\_sfb <- simplify(net\_fb)  
is.simple(net\_sfb)

## [1] TRUE

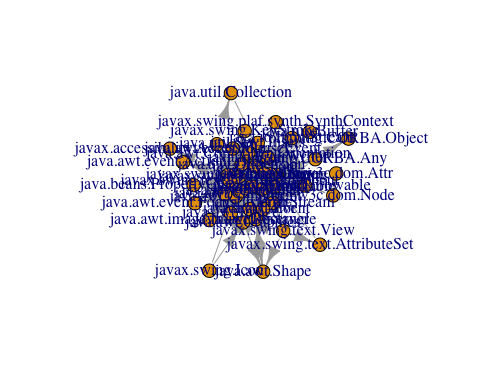
net\_sfb

## IGRAPH 61d082e DNW- 44 258 --   
## + attr: name (v/c), weight (v/n), weight (e/n)  
## + edges from 61d082e (vertex names):  
## [1] java.awt.Dimension->java.lang.String   
## [2] java.awt.Dimension->java.lang.Object   
## [3] java.awt.Dimension->java.lang.Class   
## [4] java.awt.Dimension->java.io.Serializable  
## [5] java.lang.String ->java.util.Locale   
## [6] java.lang.String ->java.lang.Object   
## [7] java.lang.String ->java.lang.Class   
## [8] java.lang.String ->java.io.Serializable  
## + ... omitted several edges

Here we use the **plot( )** function to plot the simiplified graph.

All edge weights were ignored during graph layout because non-positive edge weights were found.

plot(net\_sfb)

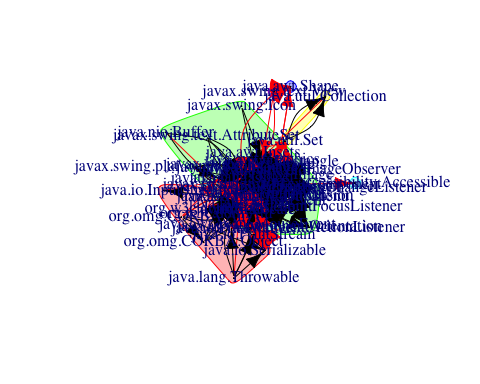


#### Communities

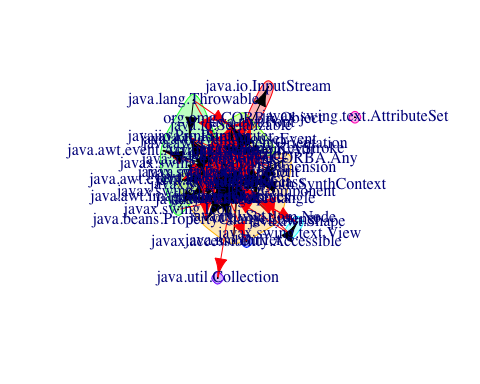
Here we use the \*\* \*\* function

An error we encountered while calculating the communities was that the weight vector must be non-negative.

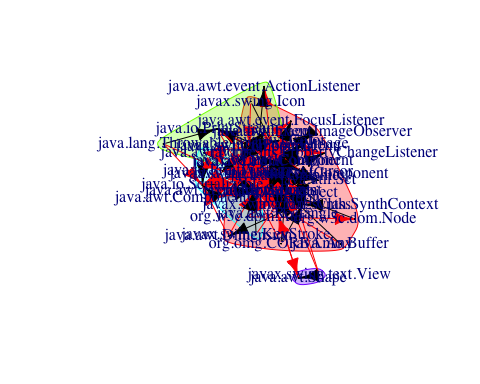
# remove edges with negative weight  
net\_fbpose <- delete\_edges(net\_fb, which(E(net\_fb)$weight < 0.0))   
  
# find communities   
wc <- walktrap.community(net\_fbpose)  
  
# plot  
plot (wc,net\_fbpose, vertex.size=0.5, layout=layout.fruchterman.reingold)

 Here we use the \*\* \*\* function

# remove negative weights from simplified graph  
net\_sfbpose <- delete\_edges(net\_sfb, which(E(net\_sfb)$weight < 0.0))  
  
# find communities in simplified graph  
wcsfb <- walktrap.community(net\_sfbpose)  
  
# plot  
plot (wcsfb, net\_sfbpose, vertex.size=0.5, layout=layout.fruchterman.reingold)



# remove isolates  
net\_ssfb <- delete\_vertices(net\_sfbpose, igraph::V(net\_sfbpose)[igraph::degree(net\_sfbpose)<2])  
   
 #induced.subgraph(net\_sfbpose,  
 #igraph::V(net\_sfbpose)[igraph::degree(net\_sfbpose)>0])  
  
# find communities in simplified graph  
wcssfb <- walktrap.community(net\_ssfb)  
  
# plot  
plot (wcssfb, net\_ssfb, vertex.size=0.5, layout=layout.fruchterman.reingold)

 How does this differ from the wc plot? See where different schools are in the two plots

How to define what is modular?

Here we use the \*\* \*\* function Here we use the \*\* \*\* function Here we use the \*\* \*\* function Here we use the \*\* \*\* function Here we use the \*\* \*\* function Here we use the \*\* \*\* function Here we use the \*\* \*\* function Here we use the \*\* \*\* function