**An Introduction to Graph Analytics**

**This rubric is meant to guide you through the process of performing graph analytics on a data set that represents a large complex graph.**

**You will use the data set that I provide in the Class Project Description.**

Setup:

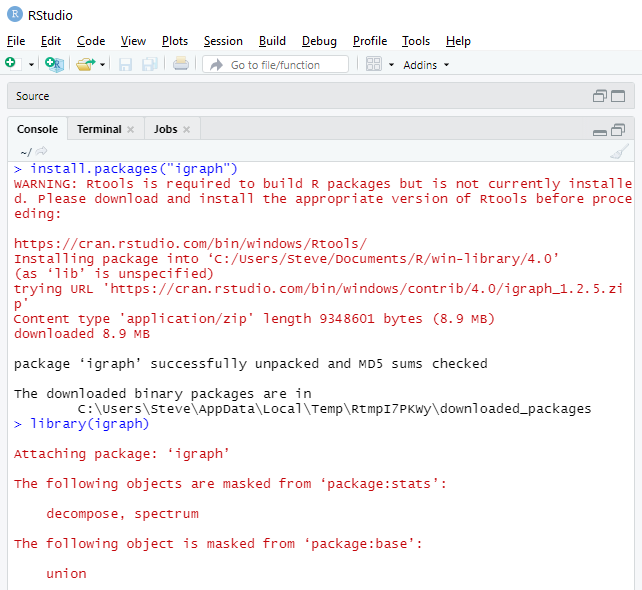
1. Open RStudio (assumes you have already downloaded and installed R).

2. Clear the console (under Edit menu)

3. Clear the Workspace (under the Session menu)

4. Install igraph

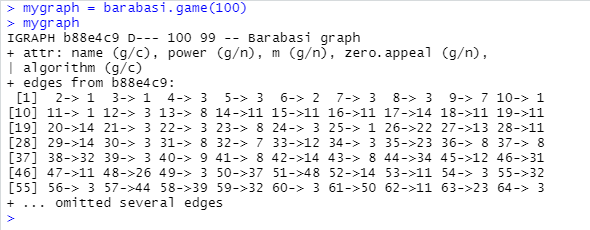
5. Declare it as a library



Don’t worry about Rtools. You are not going to be creating new packages.

Some functions occur in several packages. If they are already loaded into the workspace, they are not reloaded.

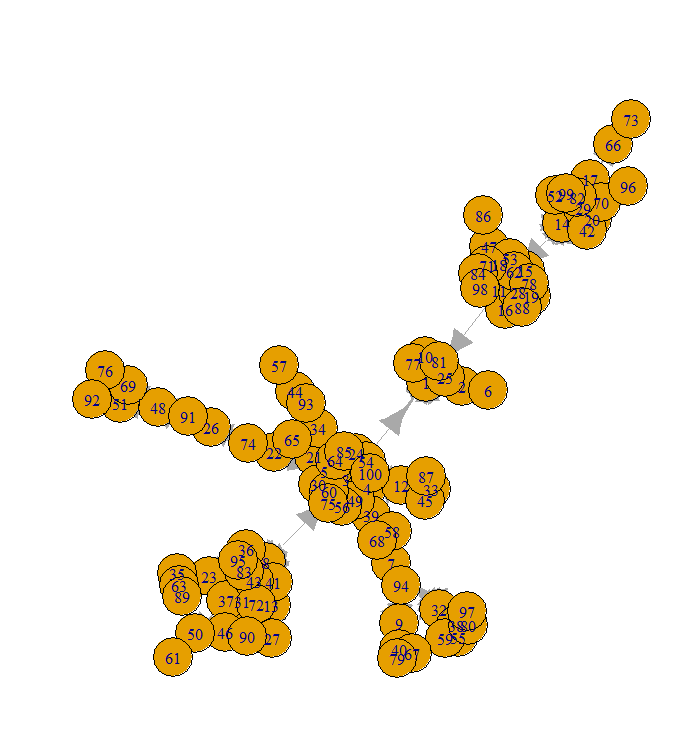
igraph has a built-in function based on the Barabasi-Albert model using a simple stochastic algorithm to create a graph.



Here, mygraph is represented as an *edge list*, e.g., a start node and a destination with an implied edge between them. Edge lists are one way to represent a graph.

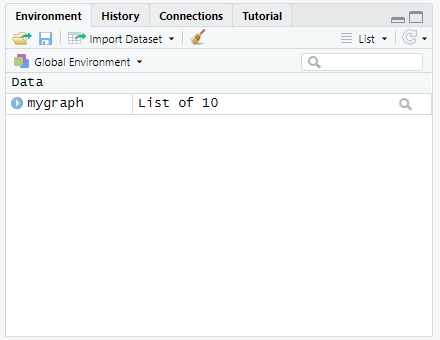
There are a number of ways to plot graphs in igraph.

> plot.igraph(mygraph)



I used a small number of nodes and edges so you could see what a graph looks like. The graph you will be using will be complex with many more nodes.

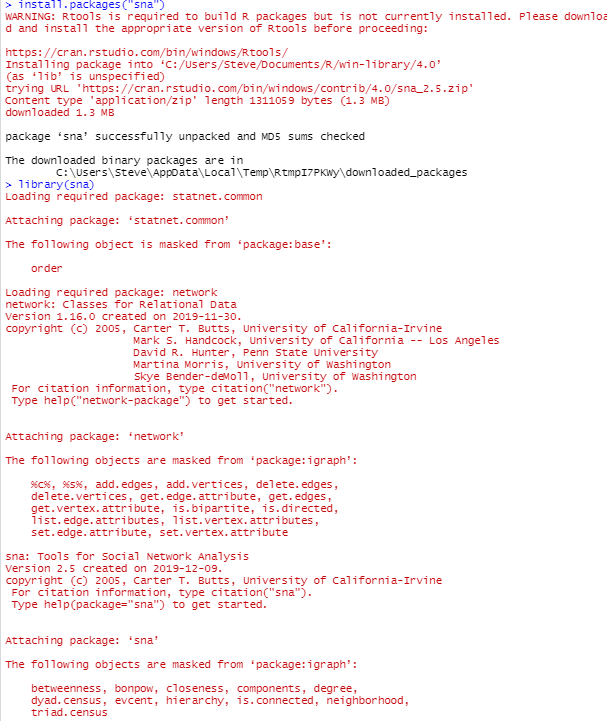
Note that mygraph is a structure that is defined in the workspace.



Using the str() function, you can see its structure. Try it!

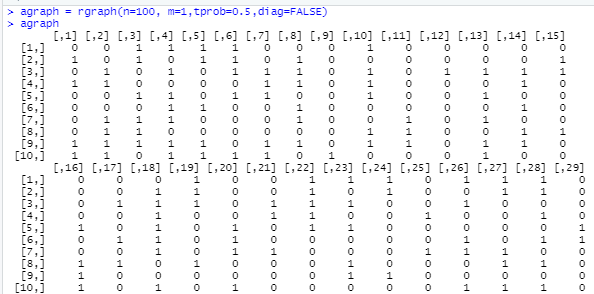
Do str(mygraph). What do you see?

Another package, which we will use later is sna (for social network analysis):



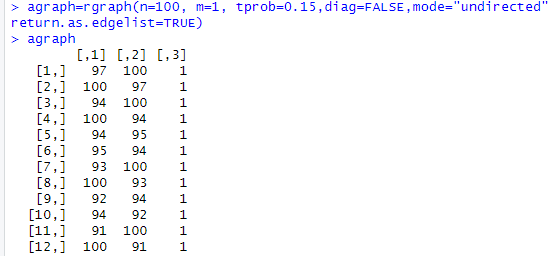
It has a function called rgraph which allows you to create arbitrary graphs for experimentation. It generates a Bernoulli Random graph.

Read the description of the parameters in the Help window (lower right pane).



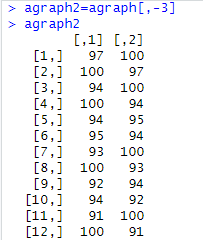
Note that it represents a graph as an *adjacency matrix*. An adjacency matrix is an alternate way of representing a graph. agraph now appears in the workspace (upper right pane).

But, you can return the results as an edgelist. Also, drop the tprob to 0.15agraph2, so we have less edges for illustrative purposes. “undirected” means no arrows on the edges.



Note: it has three columns. So, let’s delete the third column.

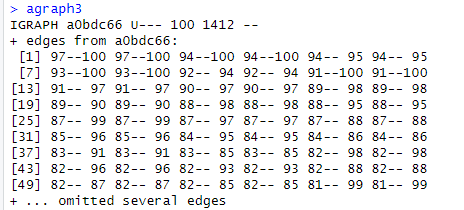
agraph[,-3] means keep all the rows, and all columns except the third.



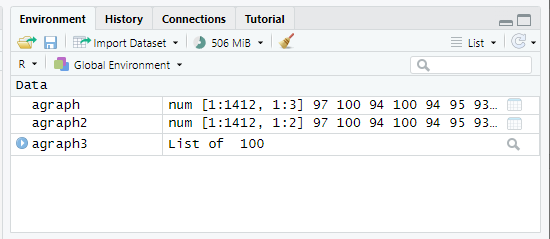
Note that I have omitted most of the printout, but you will see it all when you inspect agraph2.

Let’s convert the edgelist to an igraph object. igraph has an internal representation.





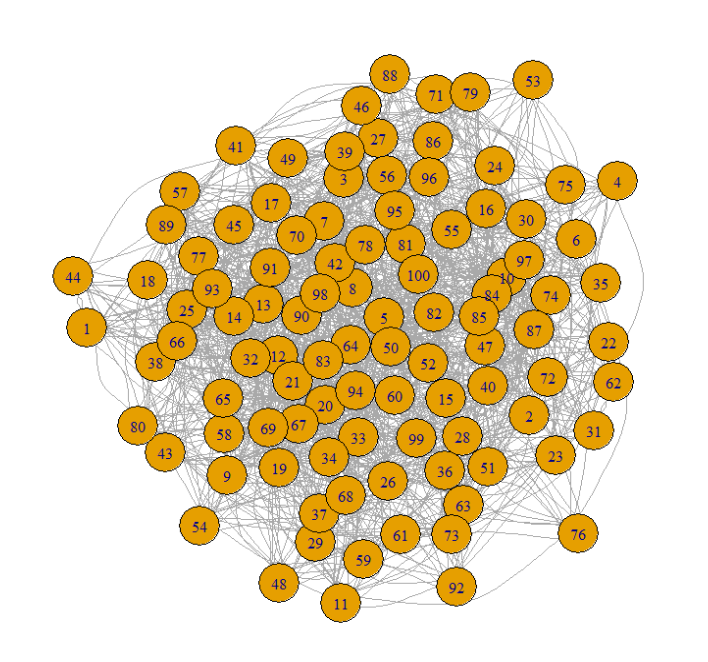
Note: New objects appear in the upper right pane.



Now, let’s plot agraph3:

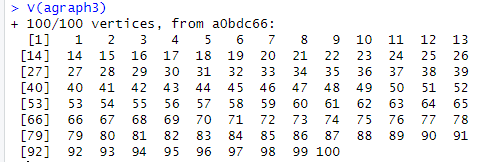
And, we get the blob! We set the probability of connected to 0.15, so there are 1448 edges, e.g., pairs of nodes. And, we set the edges as undirected to make it easier to visualize.

This blob is pretty reasonable in that you can see most of the odes and edges. In RStudio, you can zoom the plot image that may make it more readable.



So, much of Data Science is preparing the data to be analyzed. We’ll see more of this when we examine clustering and predictive analytics.

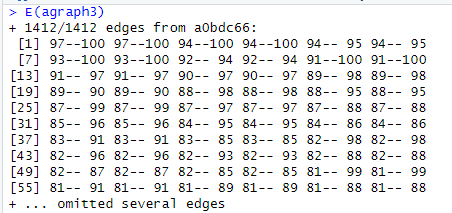
To get the vertices of a graph, we use the V function:



In this case, we created an arbitrary graph, so the nodes are labeled with their number.

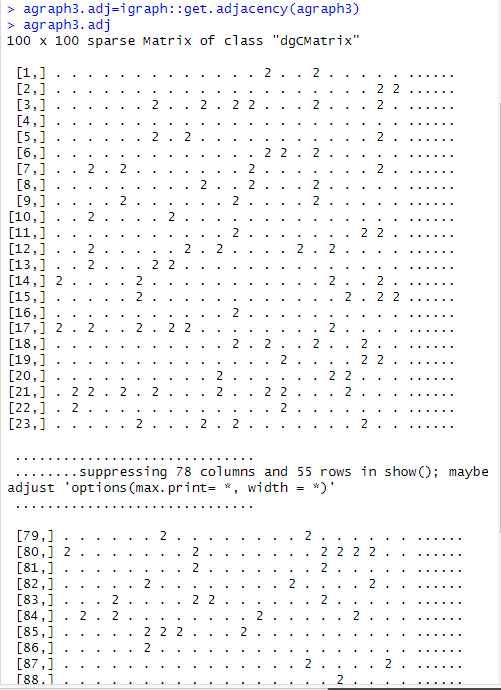
Most real-world graphs will have names associated within individual nodes.

To get the edges of a graph, we use the E function:



Note there are 1448 edges in our graph, so R prints just the first few to save space.

Get the adjacency matrix:



Let’s start looking at some analytic functions:

*Density* = # existing edges/ # possible edges. The # possible edges is the number of edges computed by assuming every node is connected to every other node in the graph. In the real world, this rarely ever occurs.

A graph with higher density is more connected.

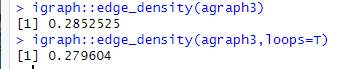
A higher density graph can resist more link failures (e.g., removals).

gden takes an adjacency matrix as its argument.



OK, what we expected. Density is about 0.15 as we specified when we created the graph, but a little less since we allowed random selection of nodes.

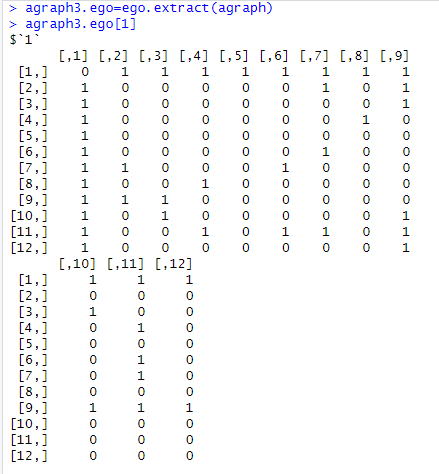
The density of a graph is the ratio of the number of edges and the number of possible edges.



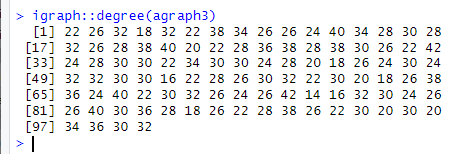
Loops specifies whether to allow loops in the graph. The default is F, but loops=T yields a smalle value because there are different paths with loops.

An *egocentric network* of a vertex v is a subgraph consisting of v and its immediate neighbors. Vertices with lots of neighbors can serve in many roles, such as brokers of information passing through the network.

Here I print the matrices for vertex 1. Vertex 1 has 12 neighbors.



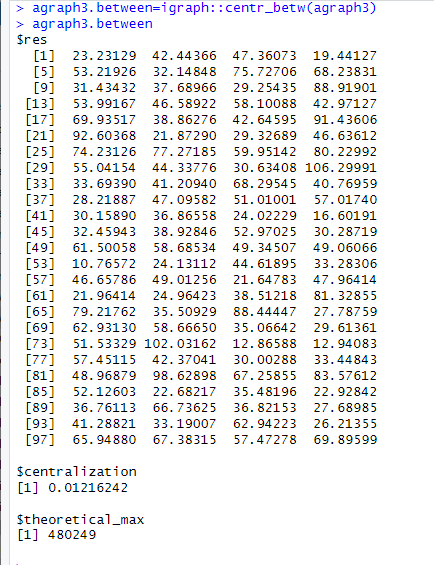
We can find the degree of each node in the graph:



Note: the function ‘degree’ might be defined in several packages. So, sometimes you need to preface the function name with the package name. Here “igraph” is the package name, so the function call is “igraph::degree”. Make sure you have the double colons.

Some centrality metrics:

Betweenness Centrality:



**betweenness centrality** is a measure of [centrality](https://en.wikipedia.org/wiki/Centrality) in a [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) based on [shortest paths](https://en.wikipedia.org/wiki/Shortest_path_problem). For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through (for unweighted graphs) or the sum of the weights of the edges (for weighted graphs) is minimized. The betweenness centrality for each [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) is the number of these shortest paths that pass through the vertex.

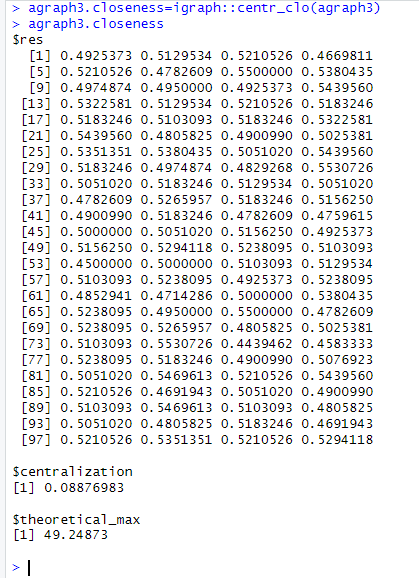
(Wikipedia)

*Closeness Centrality* (CLC) is a measure defined for a given vertex.

In a [connected](https://en.wikipedia.org/wiki/Connected_component_(graph_theory)) [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)), **closeness centrality** (or **closeness**) of a node is a measure of [centrality](https://en.wikipedia.org/wiki/Centrality) in a [network](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)), calculated as the reciprocal of the sum of the length of the [shortest paths](https://en.wikipedia.org/wiki/Shortest_path_problem) between the node and all other nodes in the graph. Thus, the more central a node is, the *closer* it is to all other nodes.

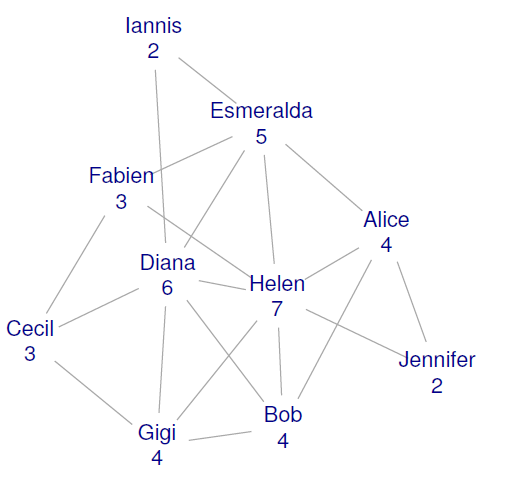
(Wikipedia)

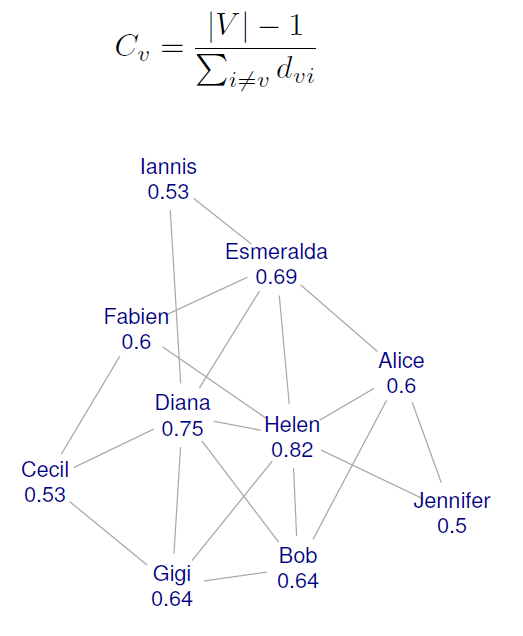
The higher the closeness, the closer the node is to other nodes in the graph.



The normalized form which represents the average length of the shortest paths, where d(x,y) is the distance between nodes x and y.

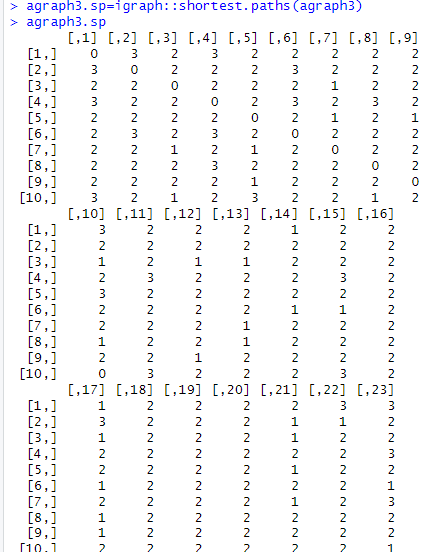
Some examples. Consider this simple graph. We present the closeness C, then the normalized value.





Finding the shortest path between two nodes:

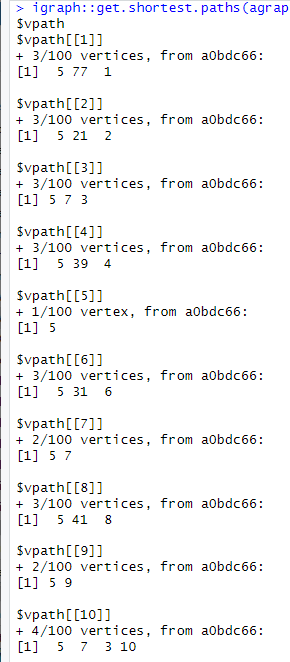
Shortest.paths provides the length of the shortest path between any two nodes in a graph g



Most rows omitted in printout.

If you want to get the actual paths (and not just their length) you can use the function get.shortest.paths

I arbitrarily picked node 5.

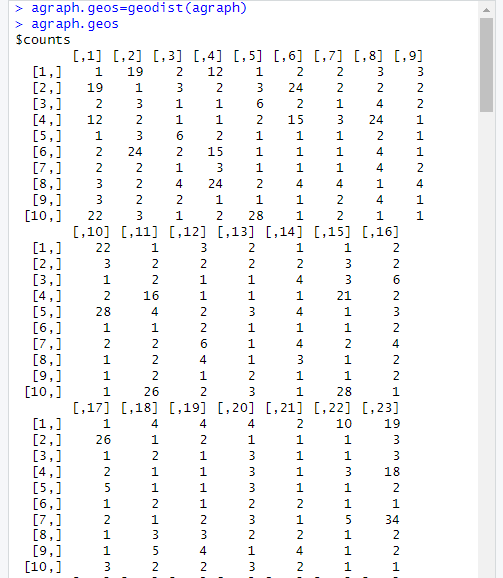


And. Omitted most of the printout.

*A geodesic* is the shortest path between any two nodes in the network.

A node has high betweenness if the geodesics between many pairs of other nodes pass through that node.

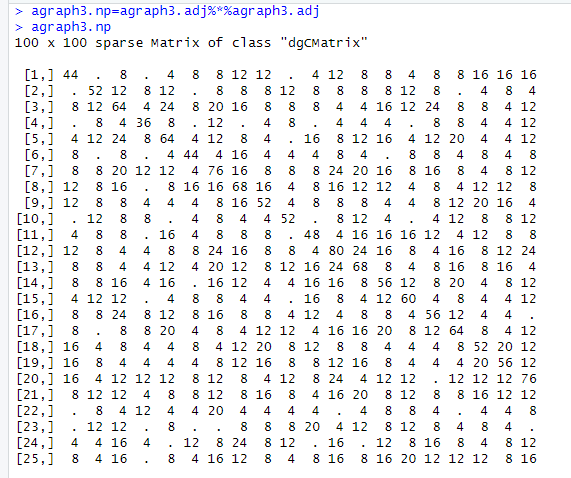
A node with high betweenness, when it fails or is removed, has greater influence on the connectivity of the network.



Most rows omitted.

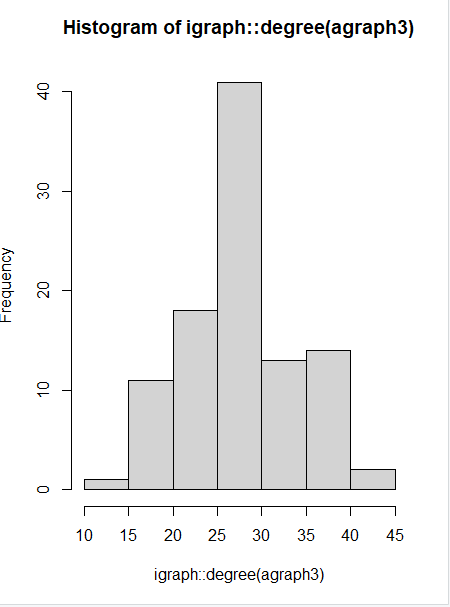
Suppose we want to find the number of paths between two nodes.

We can multiply the adjacency matrix by itself. The cell numbers specify the number of paths.



Hist(You can get a histogram of the degree of the nodes in agraph3:

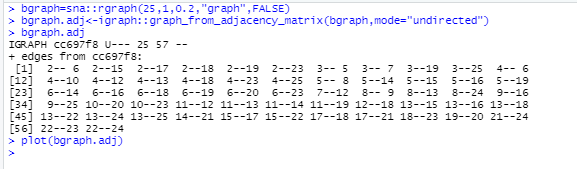
Use hist(igraph::degree(agraph3))

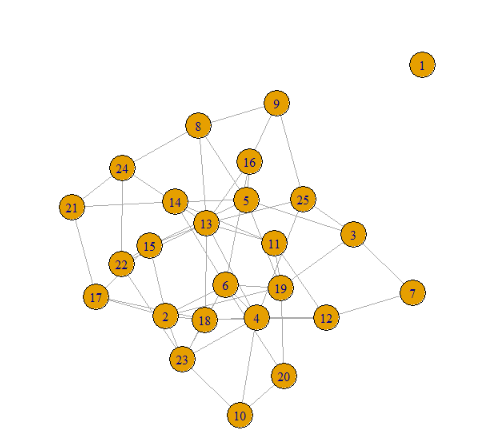


Read the igraph.pdf and you will see there are a lot of functions that you can apply.

One thing I should note is that here are different functions specified in the sna and igraph packages for analyzing graphs, so you have to be careful which ones you use.

Now, let’s go to a smaller graph to get some more visibility. You will be working with a large graph which you may have to do some filtering on to get it to a manageable size.





Notice that there is an *isolate* in this graph.

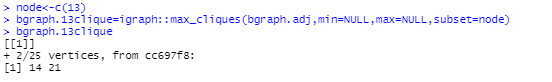
Notice the difference between density with and without loop consideration as the link factor decreases.



Find the diameter of bgraph:



Find the max-cliques for node 13:

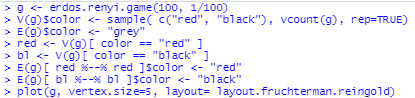


Find the size of the largest cliques:

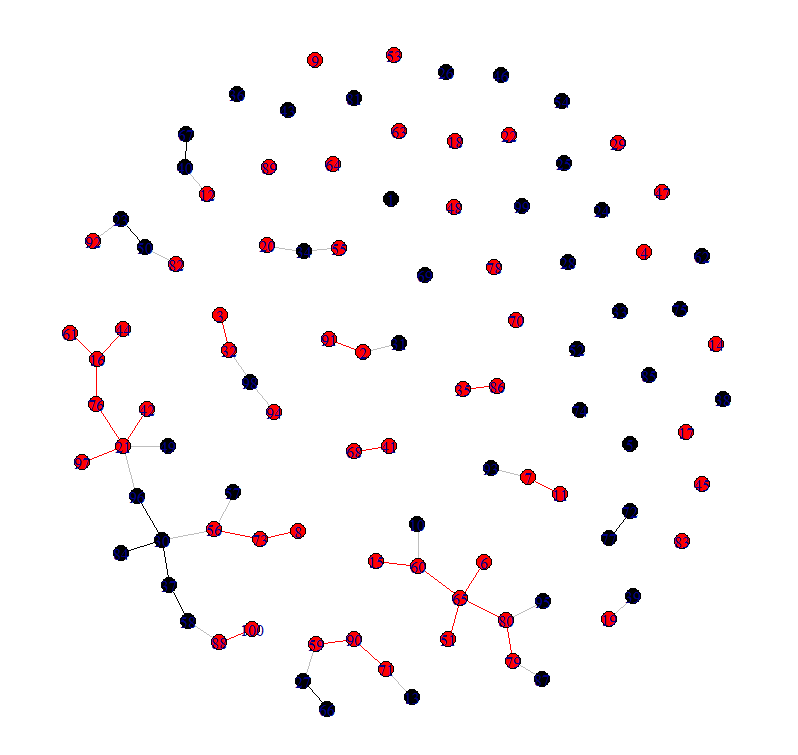


Exercise: Try to identify the cliques in the plot. Write them down.

Another Example: erdos,renyi.game with coloring



And, here is the plot:



You can name vertices using:

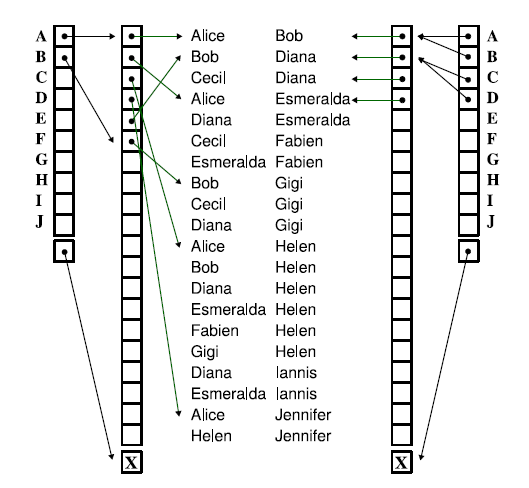
V(gmis)$name

g <- graph.ring(10)

V(g)$name <- sample(letters, vcount(g))

igraph’s representation: Flat data structures, indexed edge lists.

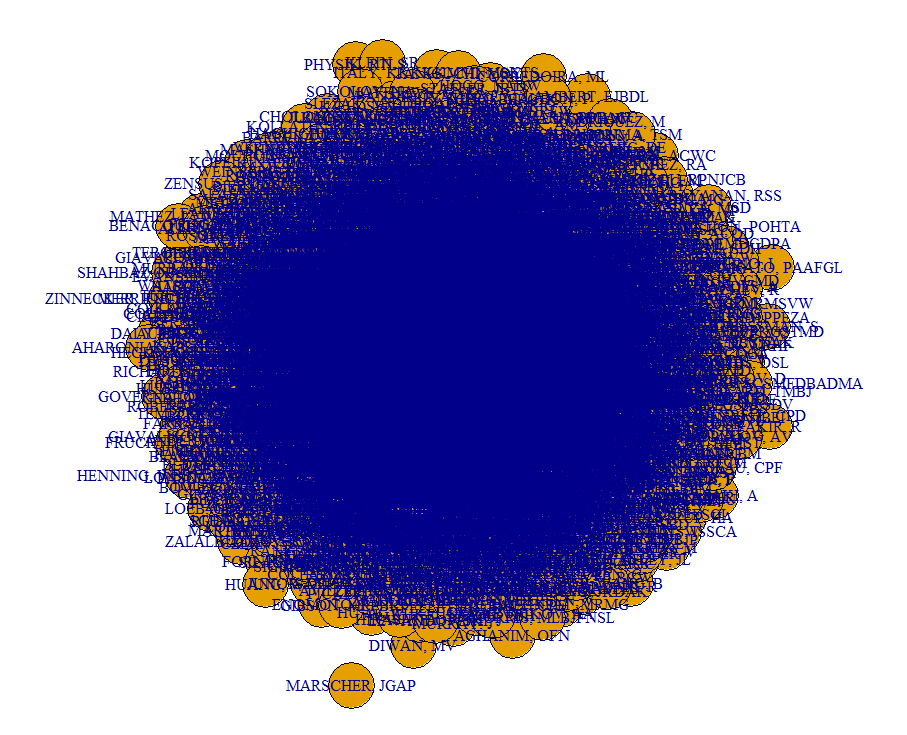
* Easy to handle, good for many kind of questions.



astrocollab is like the data set you will use but with 10x the authors and papers. So, here’s a plot of it:

Note, I used the names of the astrophysicists to label the nodes.

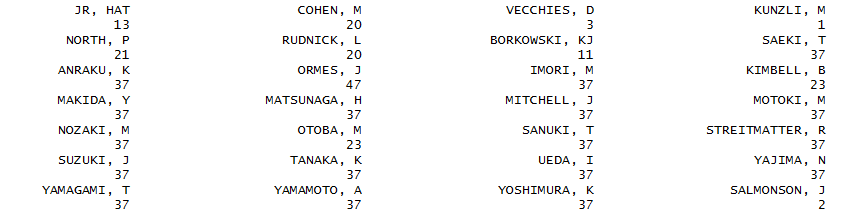
>plot(astrocollab)



Yipes!! Not very helpful, is it? This blob could be very informative if we could simplify it to look at subsets of the data.

You can find the degrees in astrocollab:

degree(astrocollab)

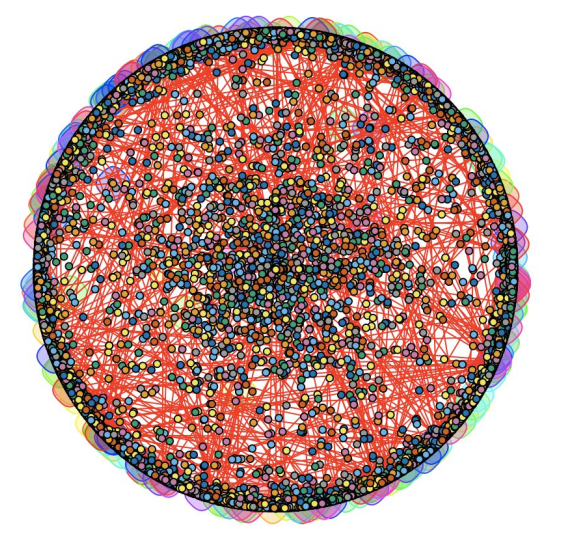


And lots more, of course….

This function tries to find densely connected subgraphs, also called communities in a graph via random walks. The idea is that short random walks tend to stay in the same community.

Here are the communities within AstroCollab:

> wc<-walktrap.community(astrocollab)  
> plot(wc, astrocollab, vertex.size=15, layout=layout.fruchterman.reingold)

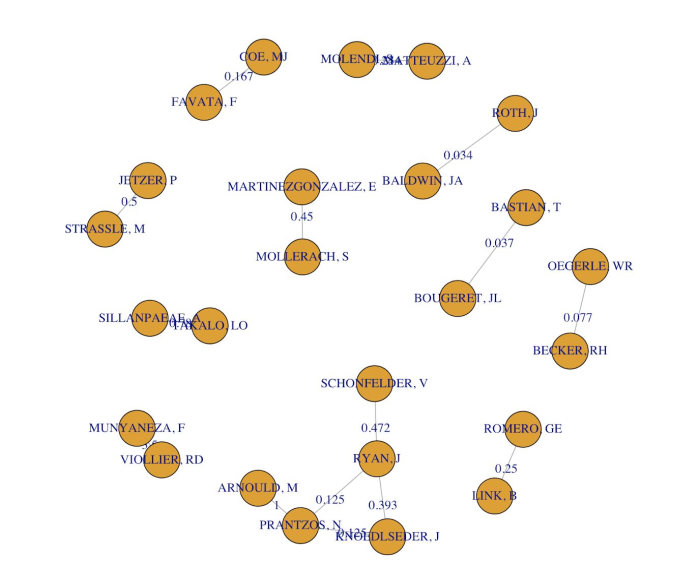


So, each node in astrocollab has the number of degrees, e.g., the sum of its in or out degrees.

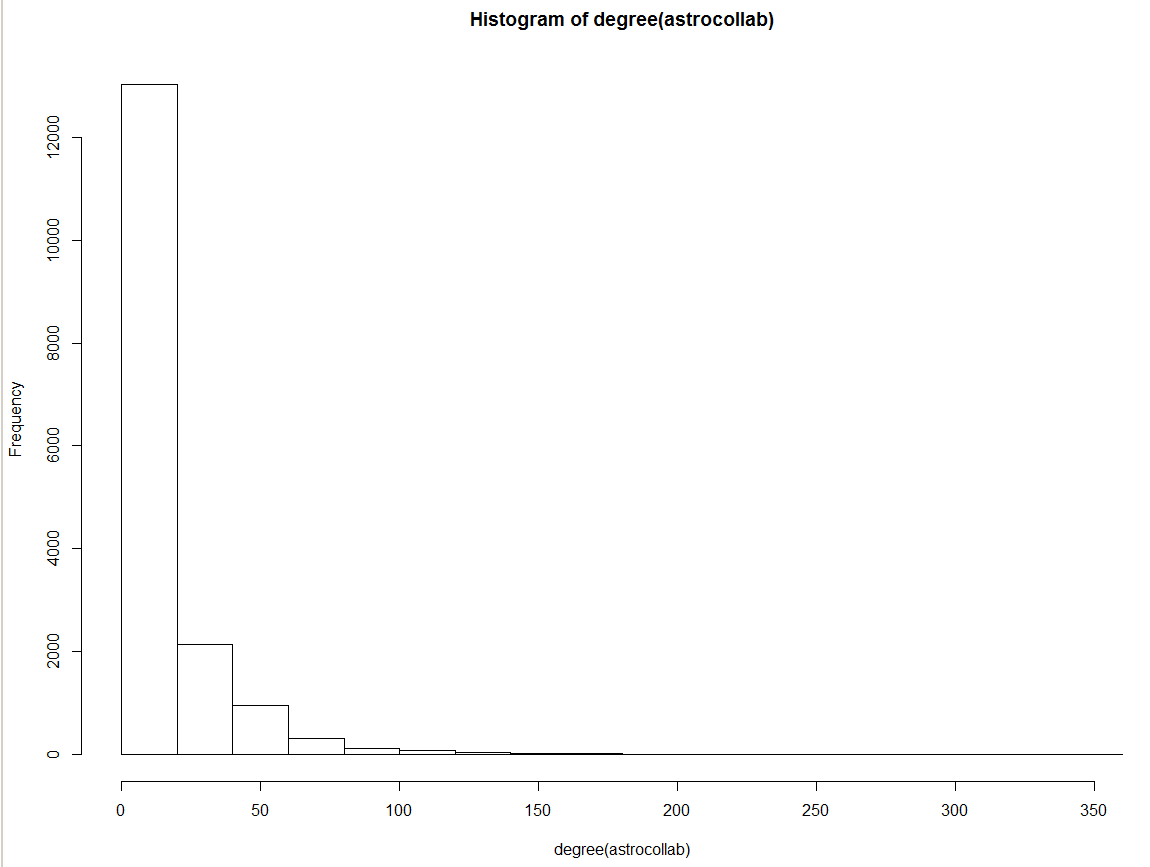
How, can you use this to remove nodes from the graph to simplify it?

We can simplify the graph by removing empty entries from the matrix, using these steps:  
> E(astrocollab)$weight <- rnorm(ecount(astrocollab))  
> V(astrocollab)$weight <- rnorm(vcount(astrocollab))  
> astrocollab[1:5, 1:9]

> sg <- induced.subgraph(astrocollab, which(V(astrocollab)$weight > 2.2))  
> plot(delete.vertices(sg, degree(sg)==0), edge.label = round(E(sg)$weight, 3))



Here’s the histogram:



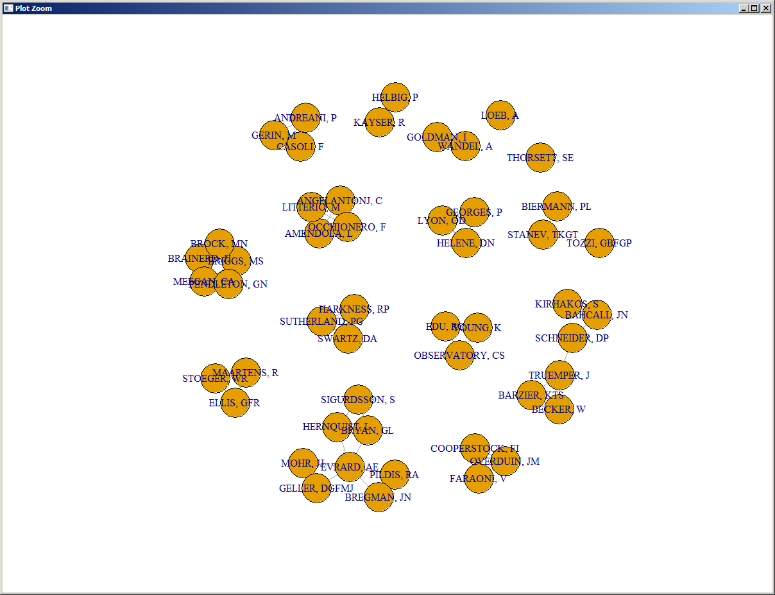
This is typically expected. Few authors have more than a few papers. And, many authors have only a few collaborators.

However, if we were doing the particle physics community, we would find that there are some papers with thousands of authors because o the size of the collaborative teams needed to execute the experiments, such as those done at CERN.

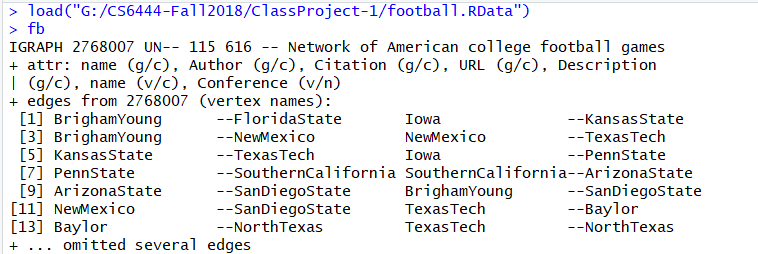
So, consider removing all authors who have just paper and also authors who have only one collaborator.

(Normally, we wouldn’t do this, but I’m trying to show you how to simplify).

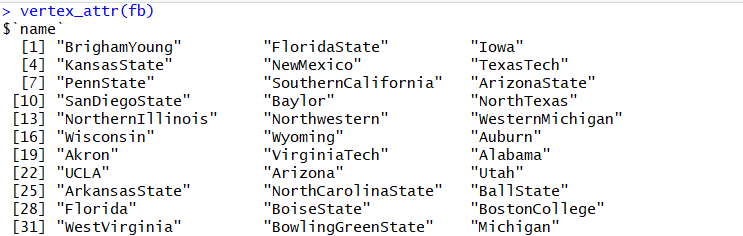
The first 50 collaborators can be plotted as. So, we see clusters of authors.



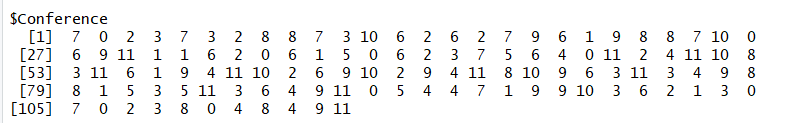
Here is some very old data about NCAA football conferences:



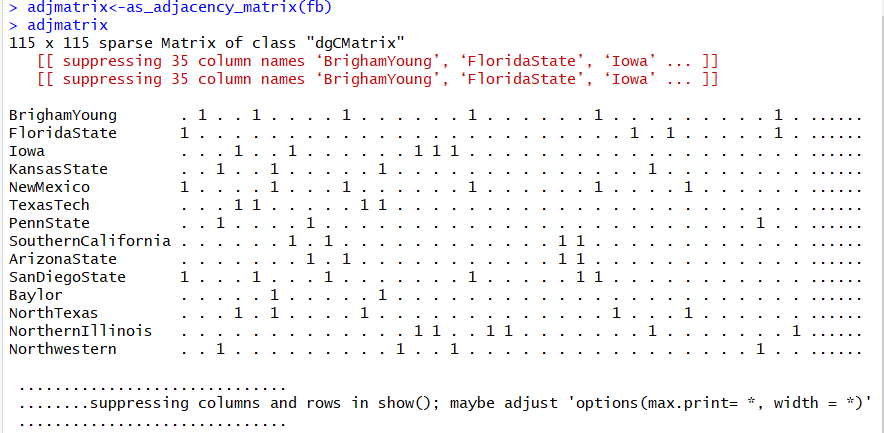
‘Name’ is an attribute



So is the ‘Conference’ they belong to

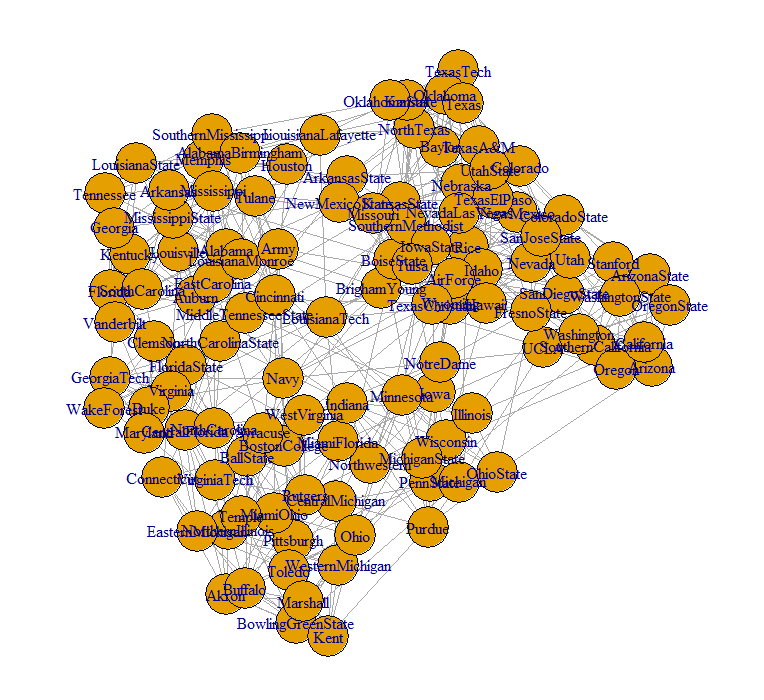


Adjacency matrices can be rather sparse.



plot(fb)

Difficult to read due to obscuration

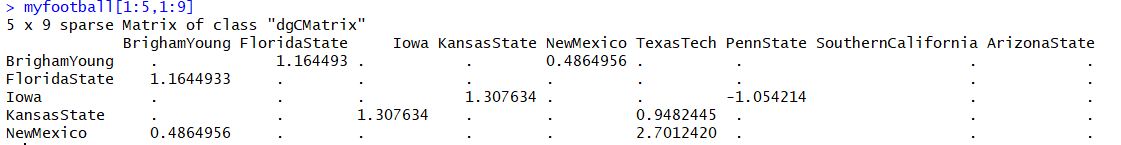


**Axiom: Plotting is your friend!!**

You can add weights to edges and vertices:



Show the weights:

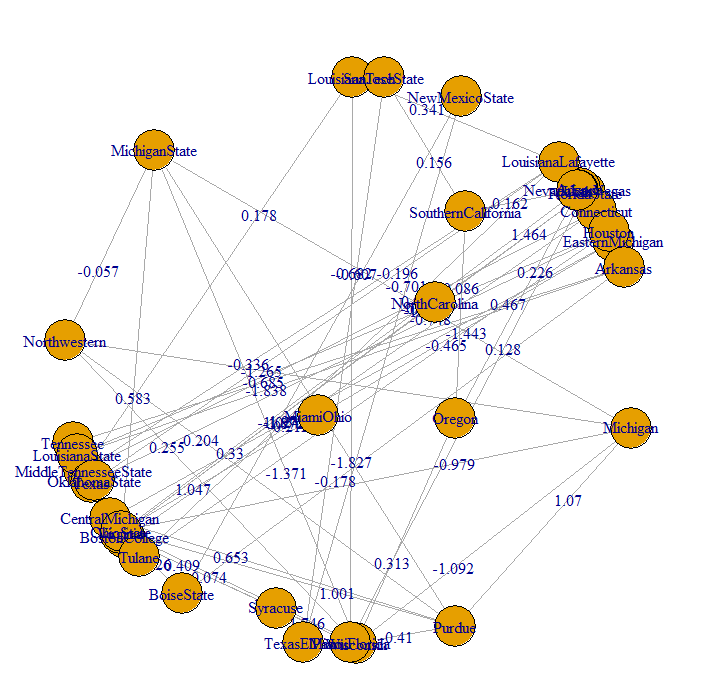


Copy the vertices with positive weight into a subgraph and including the edge weight in the plot.

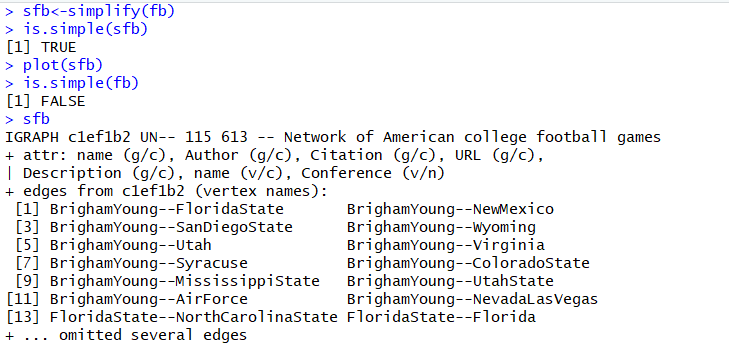


A graph without loops and multiple edges between vertices is called simple. You can check that your graph is simple like so:





You may turn a graph into a simple graph by using the *simplify* function:

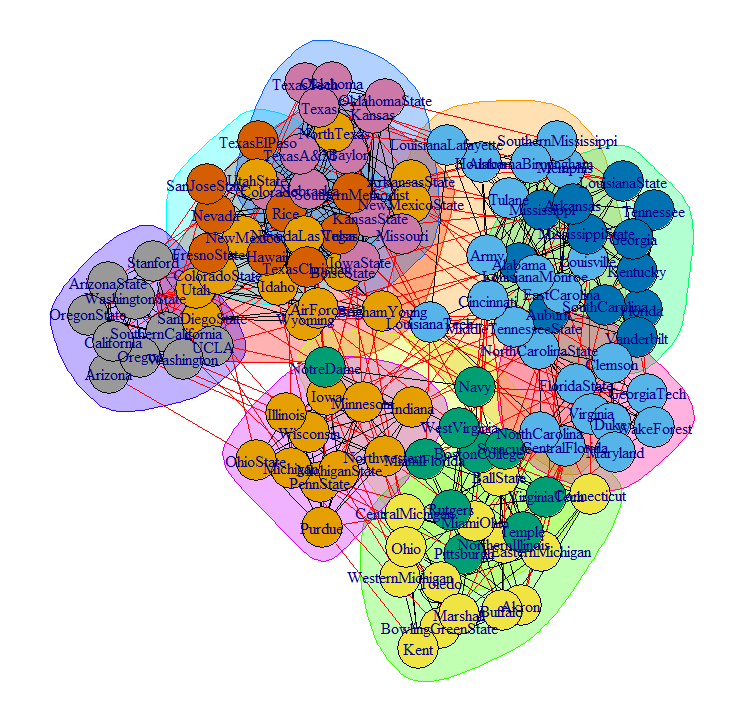


Communities:

Suppose we to find densely connected subgraphs, also called communities in a graph via random walks. The idea is that short random walks tend to stay in the same community.

wc<-walktrap.community(fb)

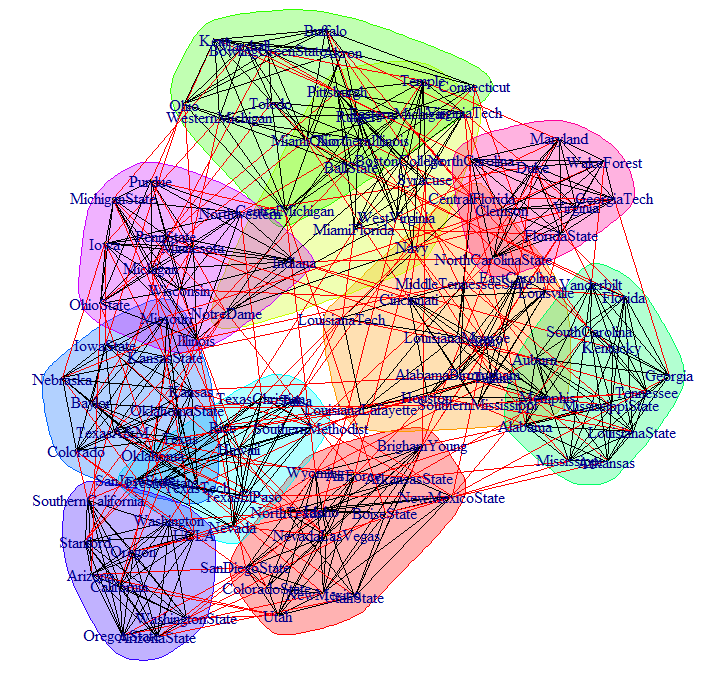
Plot(wc,fb, vertex.size=0.5, layout=layout.fruchterman.reingold)



Clustering objects in order to detect structure:

wcsfb<-walktrap.community(sfb)

Plot(wcsfb,sfb, vertex.size=0.5, layout=layout.fruchterman.reingold)

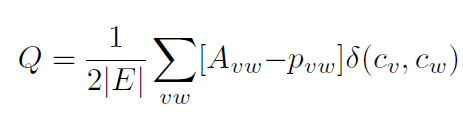


How does this differ from the wc plot?

See where different schools are in the two plots

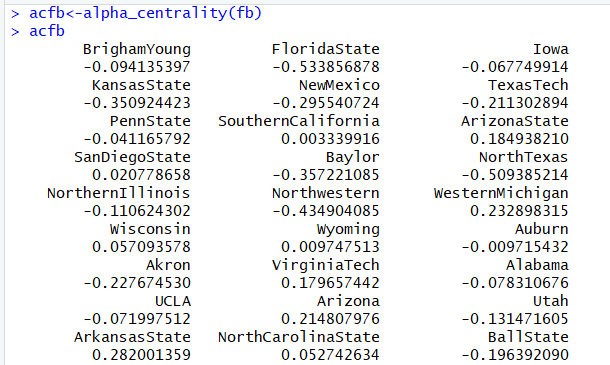
How to define what is modular?

Many proposed definitions, here is a popular one:

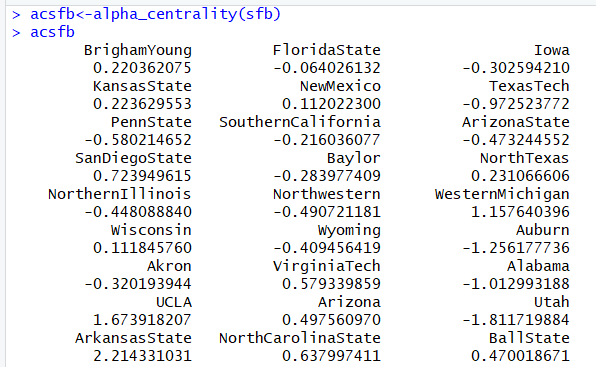


So, it appears there are lots of collaborations. So, let’s see if we can filter some out and get a better idea of what is going on.

*alpha\_centrality* calculates the alpha centrality of some (or all) vertices in a graph. It is an adaptation of [eigenvector centrality](https://en.wikipedia.org/wiki/Eigenvector_centrality) with the addition that nodes are imbued with importance from external sources.



Now, look at the alpha\_centrality of sfb:



The central organization is the one with most degrees, so we just find out theorganization with largest degree. But, there may be many such organizations in a large graph.

If you use astrocollab graph, you may get a message that it was produced with an earlier version of R, as this one was. So, do this:

mygraph = upgrade\_graph("astrocollab")

degreenumber = degree(mygraph, mode = "total")

sort(degreenumber)

The longest path must be in the largest connected component. Use DFS to travel from one vertex, the largest distance will be the result.

sg = induced\_subgraph(mygraph, which(components(mygraph)$membership == 1))

V(sg)$degree = degree(sg)

result = dfs(sg, root = 1, dist = TRUE)$dist

Assign the attribute degree to each vertex. In the largest clique, each vertex has the same minimum degree; we try to find out the minimum degree with a splitting graph, sg. Verify the max clique in sg. We can use binary search to close to the result.

V(mygraph)$degree = degree(mygraph)

sg = induced.subgraph(mygraph, which(V(mygraph)$degree > 50))

vcount(sg)

max\_cliques(sg)

**Since edges in the graph have weight, to find the ego centrality, we need to find the vertex(ices) with the largest sum of edge weights.**

First, we try to reduce the graph, delete all the vertices with degree less than 50. And try to find out the vertex with largest adjacent edges weight sum. The result is 60.75835. The actual weight sum is 80.

Second, we reduce the graph by deleting edges with weight less than 1. The result is 74.38169; the real weight sum is 109.

Third, verify all the results with weight sum larger than 40 in step 2. We got the same result.

V(mygraph)$degree = degree(mygraph)

mynewgraph = delete.edges(mygraph, which(E(mygraph)$weight <= 1))

ecount(mynewgraph)

for (v in V(mynewgraph)) {

edges = incident(mynewgraph, v);

vertex\_attr(mynewgraph, "weight", index = c(v)) <- sum(edges$weight);

}

max\_weight = max(V(mynewgraph)$weight)

max\_weight

target\_single\_vertex\_graph = induced.subgraph(mynewgraph, which(V(mynewgraph)$weight == max\_weight))

V(target\_single\_vertex\_graph)

We can't calculate power centrality directly due to the large scale of the graph. It may run out of memory or just keep on churning with no apparent result.

* Assume 1, the node with power centrality is in the biggest connected component.

Delete the node with degree one, but fail to find out the result.

* Assume 2: the person with a larger number of degrees will be more liley to have greater power of centrality.

At first, we try these people who have degree larger than 300, calculate the power of centrality.

Second, try nodes with degree greater than 200, we got a result.

Then we use binary search method to try degree, 100, 150, 125, 112, 106.

At last, we find 106 is the critical value. We obtain an approximate result due to the limitation of computational power.

result = power\_centrality(mygraph, rescale = TRUE) # Failure

new\_graph = delete.vertices(mygraph, which(degree(newgraph) < 300))

result = power\_centrality(new\_graph, rescale = TRUE)

sort(result) # result 1

new\_graph = delete.vertices(mygraph, which(degree(newgraph) < 200))

result = power\_centrality(new\_graph, rescale = TRUE)

sort(result) # result 2

new\_graph = delete.vertices(mygraph, which(degree(newgraph) < 100))

result = power\_centrality(new\_graph, rescale = TRUE)

sort(result) #Failure

new\_graph = delete.vertices(mygraph, which(degree(newgraph) < 150))

result = power\_centrality(new\_graph, rescale = TRUE)

sort(result) #result 3

new\_graph = delete.vertices(mygraph, which(degree(newgraph) < 125))

result = power\_centrality(new\_graph, rescale = TRUE)

sort(result) #result 4

new\_graph = delete.vertices(mygraph, which(degree(newgraph) < 112))

result = power\_centrality(new\_graph, rescale = TRUE)

sort(result) #result 5

new\_graph = delete.vertices(mygraph, which(degree(newgraph) < 106))

result = power\_centrality(new\_graph, rescale = TRUE)

sort(result) #result 6

The central person could be defined as the person with the most edges = sum of indegree and outdegree.

It could also be defined as the person(s) with the highest betweenness centrality. Note: for complex graphs, there may be many such people with the highest betweenness centrality. Then, other rules and measures must be applied to decide who is mot central.

**ego: find biggest immediate neighborhood (of a single vertex)**

A: Ego-centrality is similar to or equal to betweenness centrality depending on the reference you consult.

Betweenness centrality is a measure of the number of indegree edges to a vertex, usually without regard to any weighting.

Ego-centrality has sometimes been defined as the sum of the weights associated with the indegree edges suitably normalized (such as to whole numbers). The latter must be done with the same scaling factor for all edges in the graph.

A Little More about R:

Special constants include:

* **NA** for missing or undefined data
* **NULL** for empty object (e.g. null/empty lists)
* **Inf** and **-Inf** for positive and negative infinity
* **NaN** for results that cannot be reasonably defined

# NA - missing or undefined data

5 + NA # When used in an expression, the result is generally NA

is.na(5+NA) # Check if missing

# NULL - an empty object, e.g. a null/empty list

10 + NULL # use returns an empty object (length zero)

is.null(NULL) # check if NULL

Inf and -Inf represent positive and negative infinity. They can be returned by mathematical operations like division of a number by zero:

5/0

is.finite(5/0) # Check if a number is finite (it is not).

NaN (Not a Number) - the result of an operation that cannot be reasonably defined, such as dividing zero by zero.

0/0

is.nan(0/0)

**Additional Material:**

* igraph supports multiple network representations
  + Details of the actual format (e.g., punctuation, etc.) might be different for different network libraries
    - However, the main format will be the same
* Assuming that we have network represented as an edgelist we can “read” it in igraph
* Remember to download and read “igraph.pdf”

> library(igraph)

> g <- read.graph(“some graph”,format="ncol”,directed=F)

* Assume you use read.csv to read in data from a CSV file
  + Read in the vertices
  + Read in the links between them
* Use igraph’s graph.data.frame function, which takes two data frames: d and vertices
  + **d** describes the edges of the network. Its first two columns are the IDs of the source and the target node for each edge. The following columns are edge attributes (weight, type, label, or anything else).
  + **vertices** starts with a column of node IDs. Any following columns are interpreted as node attributes.
* Something like this:

net <- graph\_from\_data\_frame(d=links, vertices=nodes, directed=T)

* Use the class function to check the class

class(net)

igraph Plotting Parameters

You can improve the visualization of your graph using some of these parameters:

