**An Introduction to Graph Analytics**

Setup:

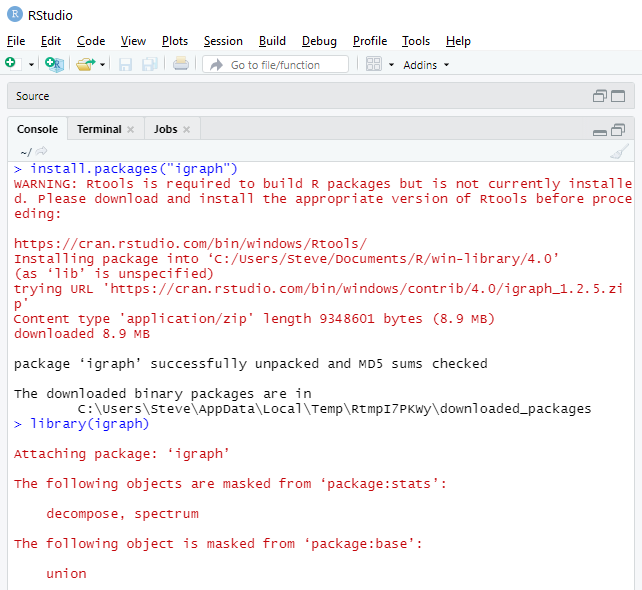
1. Open RStudio

2. Clear the console (under Edit menu)

3. Clear the Workspace (under the Session menu)

4. Install igraph

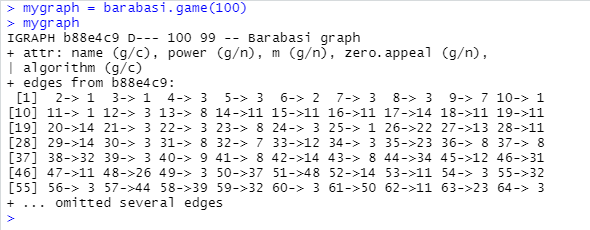
5. Declare it a library



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.

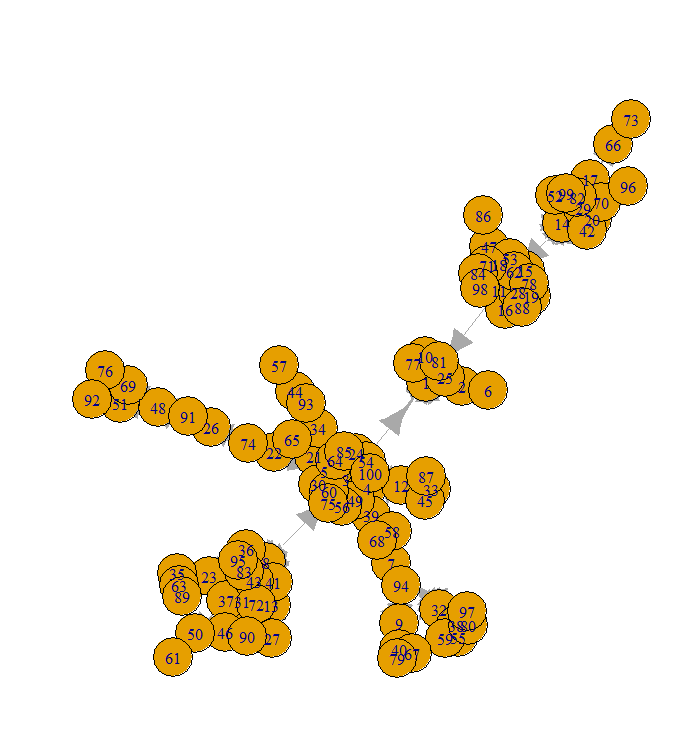
igraph has a built-in function based on the Barabasi-Albert model using a simple stochastic algorithm to create a graph.



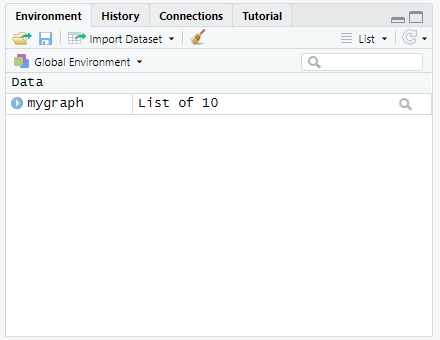
Here, mygraph is represented as an *edge list*, e.g., a start node and a destination with an implied edge between them.

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

> plot.igraph(mygraph)

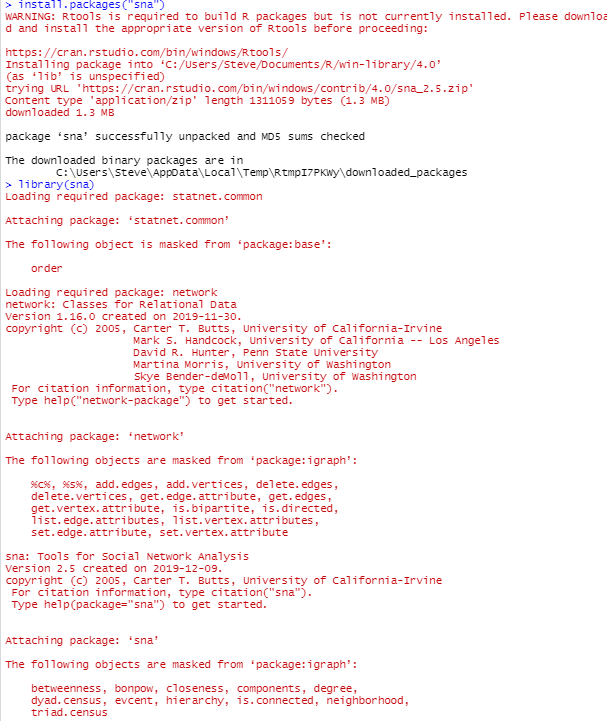


Note that mygraph is a structure that is defined in the workspace.



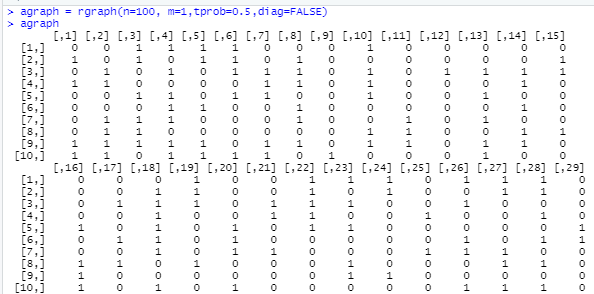
Using the str() function, you can see its structure. Try it!

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



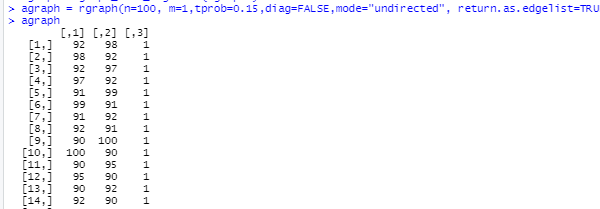
It has a function called rgraph which allows you to create arbitrary graphs for experimentation. It generates a Bernoulli Random graph.

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



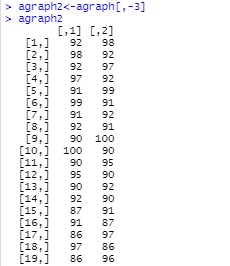
Note that it represents a graph as an *adjacency matrix*. agraph now appears in the workspace (upper right pane).

But, you can return the results as an edgelist. Also, drop the tprob to 0.3, so we have less edges for illustrative purposes. “undirected” means no arrows on the edges.



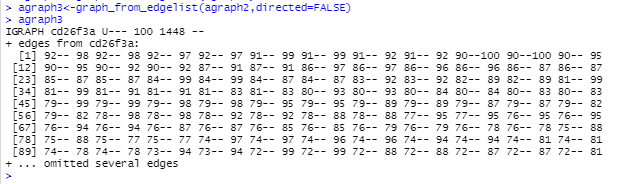
Note: it has three columns. So, let’s delete the third column.

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



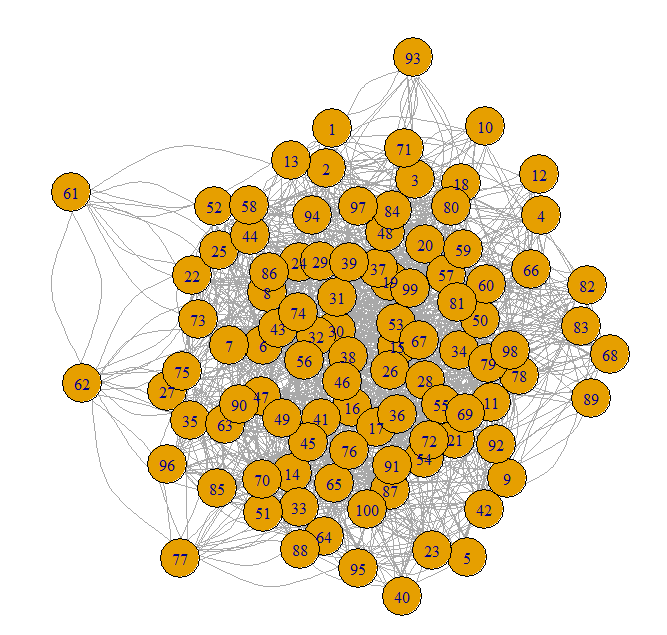
Note that I have omitted most of the printout, but you will see it all when you inspect agraph2.

Let’s convert the edgelist to an igraph object:



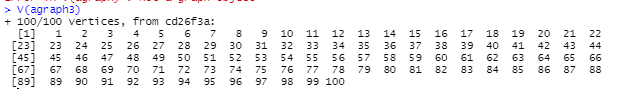
>plot.igraph(agraph3)

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.



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.

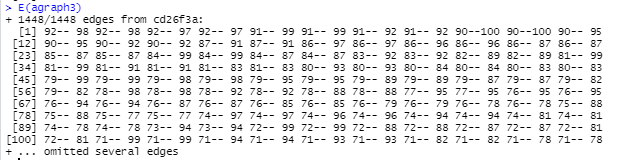
To get the vertices of a graph, we use the V function:



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

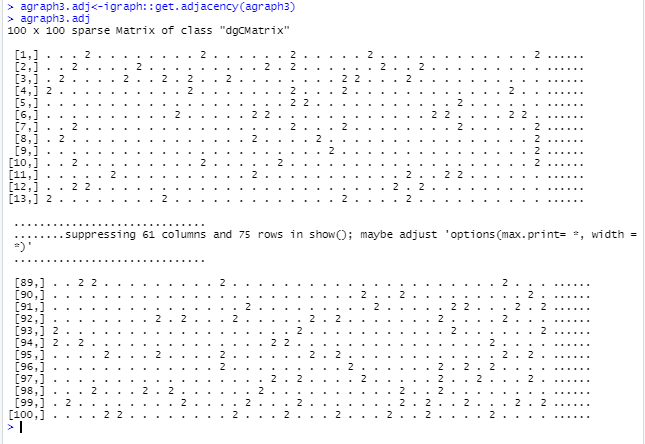
Most real world graphs will have names associated within individual nodes.

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

A graph with higher density is more connected

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

gden works on adjacency matrix.



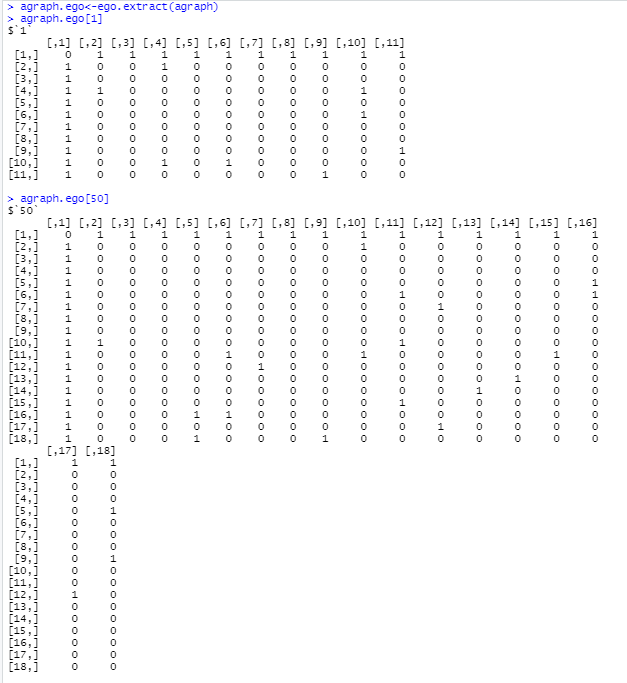
OK. what we expected. Density is about 0.15 as we specified when we created the graph.

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

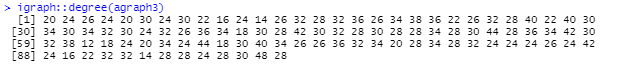


An *egocentric network* of a vertex v is a subgraph consisting of v and its immediate neighbors. Vertices with lots of neighbors can serve in many roles, such as brokers of information passing through the network.

Here I print the matrices for vertices 1 and 50. Vertex 1 has 10 neighbors and vertex 50 has 17 neighbors.



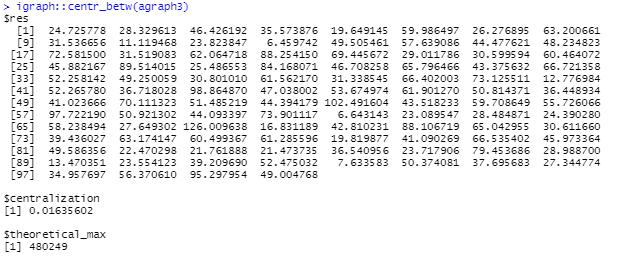
We can find the degree of each node in the graph:



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 double colons.

Some centrality metrics:

Betweenness Centrality:



Centralization: The graph level centrality index

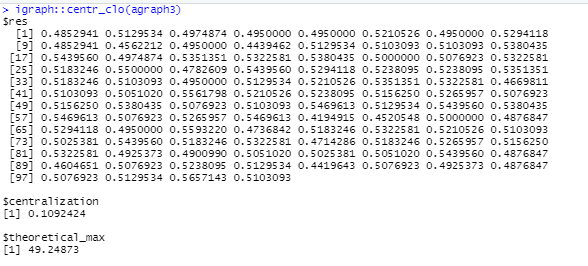
Theoretical\_max: The maximum theoretical graph level centralization score for a graph with the given number of vertices, using the same parameters.

Closeness Centrality:

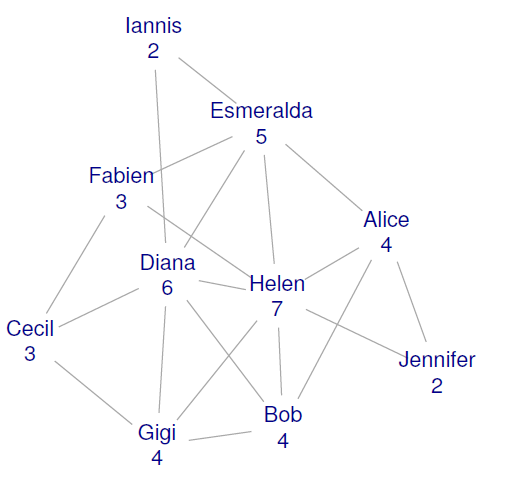
*Closeness Centrality* (CLC) is a measure defined for a given vertex.

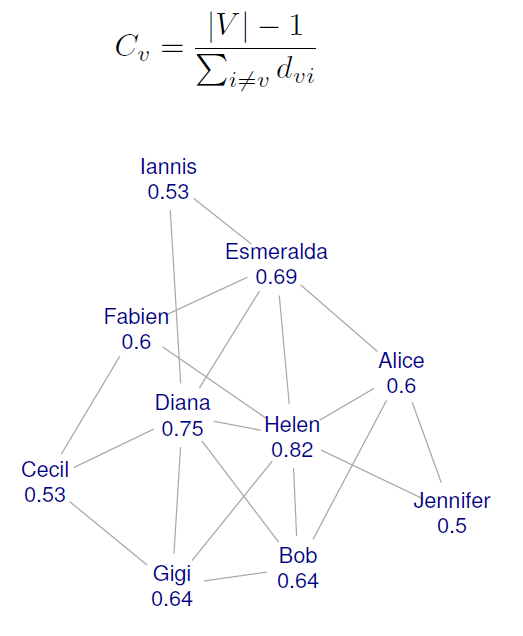
The higher the closeness, the closer the node is to other nodes in the graph.

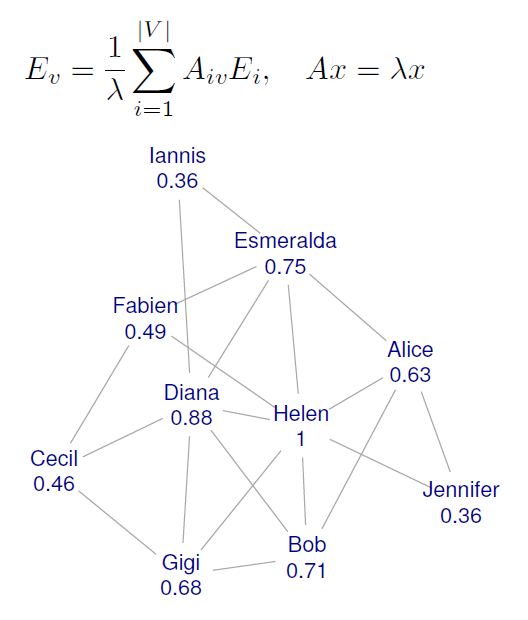
The ratio of the total number of nodes in the graph minus one to the sum of the geodesics from v to every other vertex in the graph.

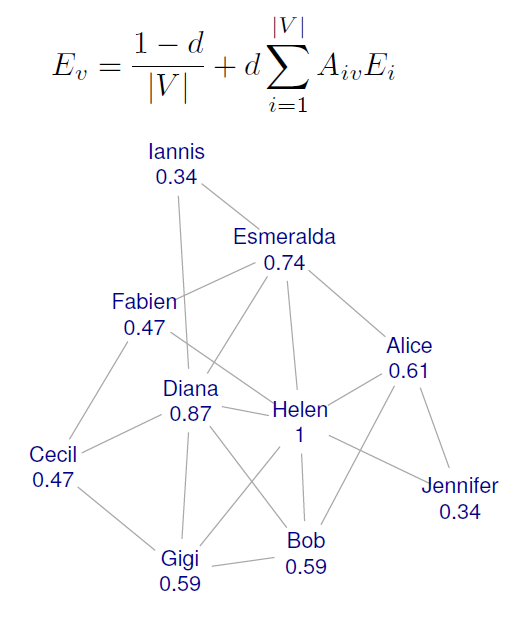


Consider this simple graph:



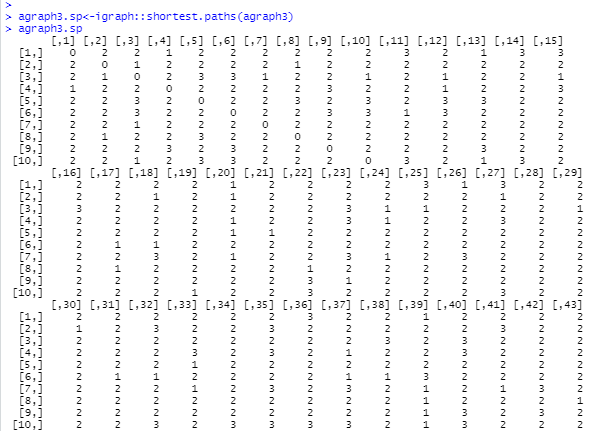






Finding the shortest path between two nodes:

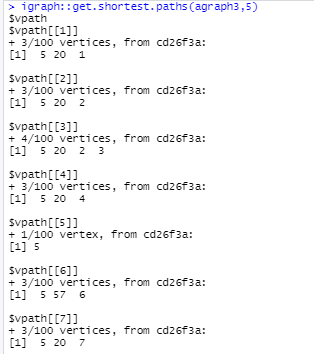
Shortest.paths provides the length of the shortest path between any two nodes in a graph g



Almost 90 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

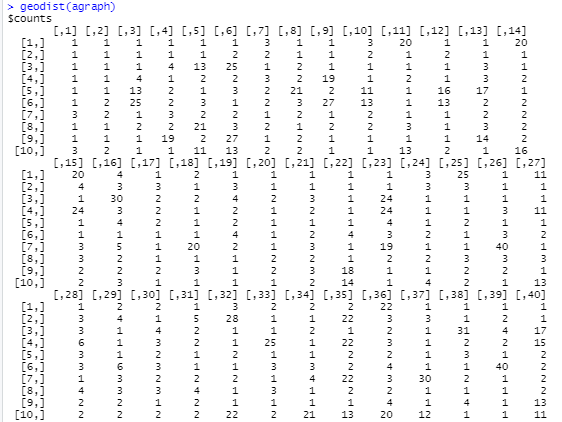
I arbitrarily picked node 5.



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

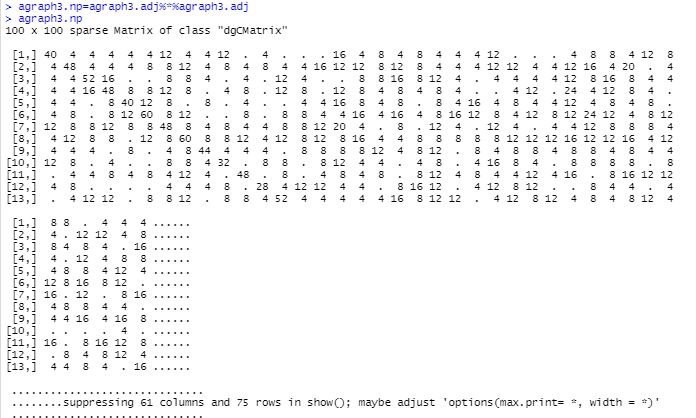
A node has high betweenness if the geodesics between many pairs of other nodes pass through that node.

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



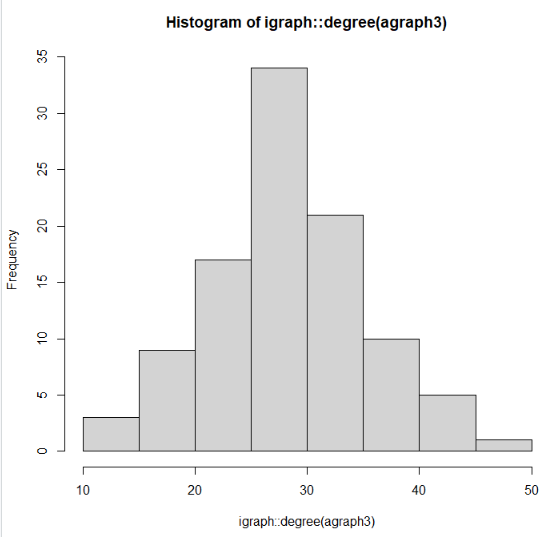
Suppose we want to find the number of paths between two nodes.

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



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

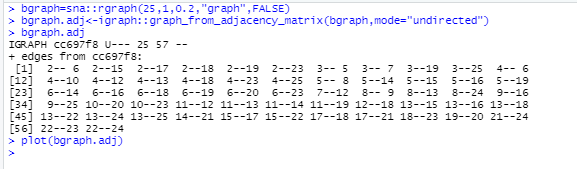
Use hist(igraph::degree(agraph3))

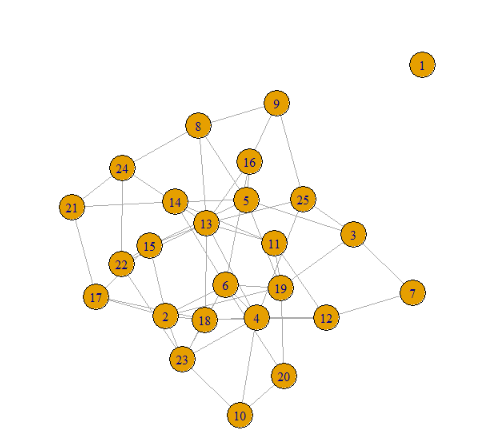


Read the igraph.pdf and you will see there are a lot of functions that you can apply.

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.

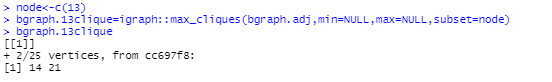
Notice the difference between density with and without loop consideration as the link factor decreases.



Find the diameter of bgraph:



Find the max-cliques for node 13:

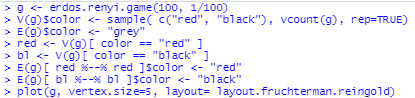


Find the size of the largest cliques:

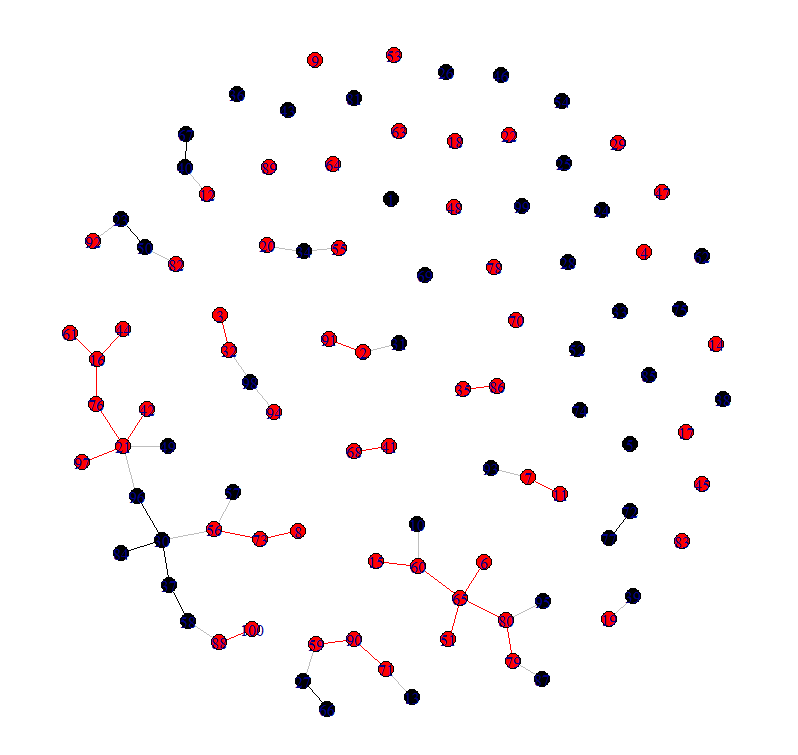


Exercise: Try to identify the cliques in the plot. Write them down.

Another Example: erdos,renyi.game with coloring



And, here is the plot:



You can name vertices using:

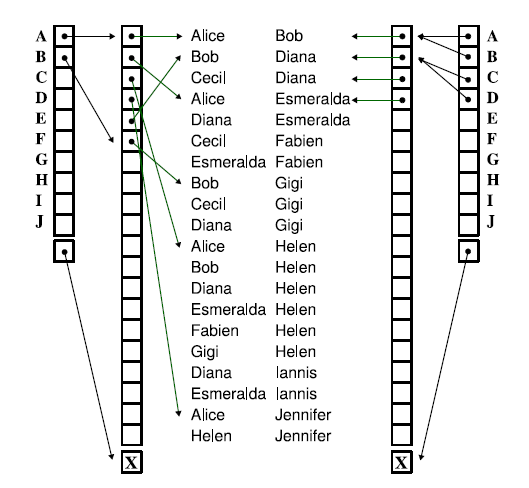
V(gmis)$name

g <- graph.ring(10)

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

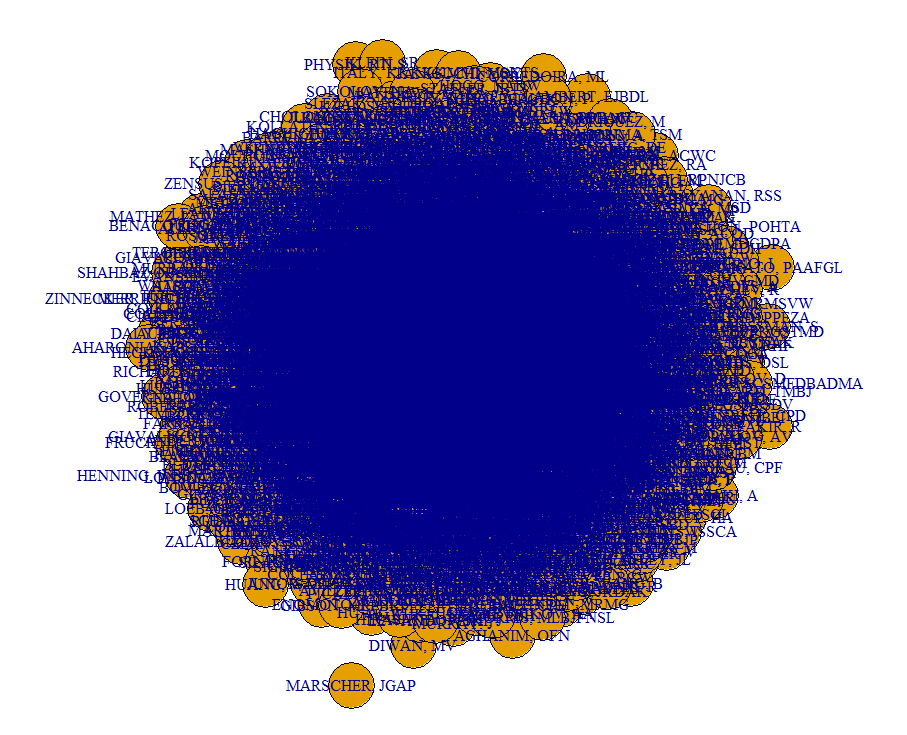
igraph’s representation: Flat data structures, indexed edge lists.

* 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:

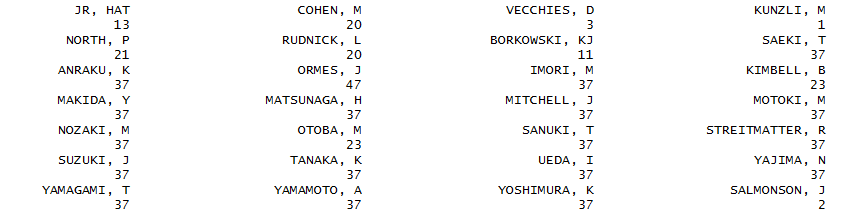
>plot(astrocollab)



Yipes!! Not very helpful, is it?

You can find the degrees in astrocollab:

degree(astrocollab)



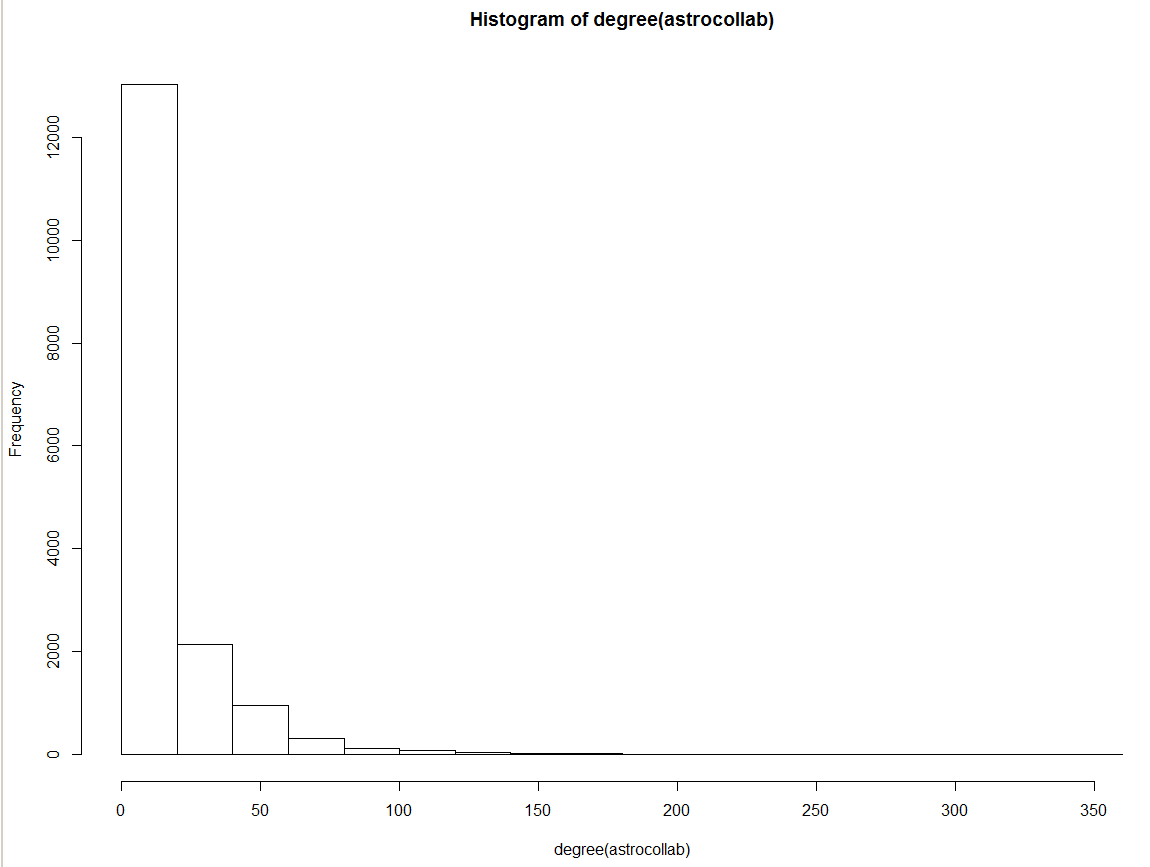
And lots more, of course….

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 form the graph to simplify it?

Here’s the histogram:



This is typically expected.

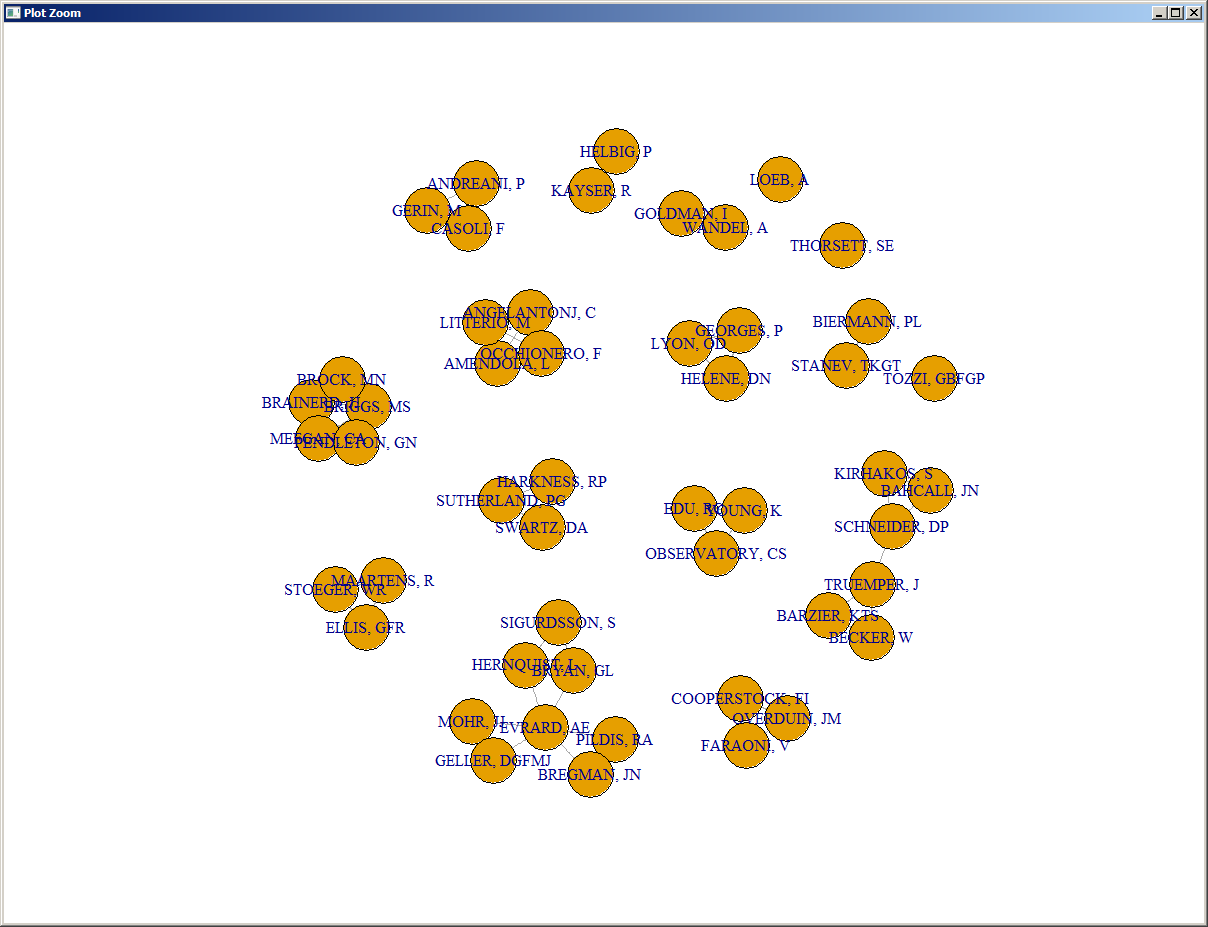
Few authors have more than a few papers.

And, many authors have only a few collaborators.

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

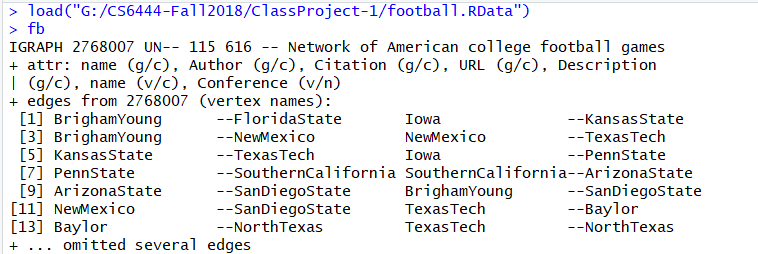
(Normally, we wouldn’t do this, but I’m trying to show you how to simplify).

The first 50 collaborators can be plotted as:

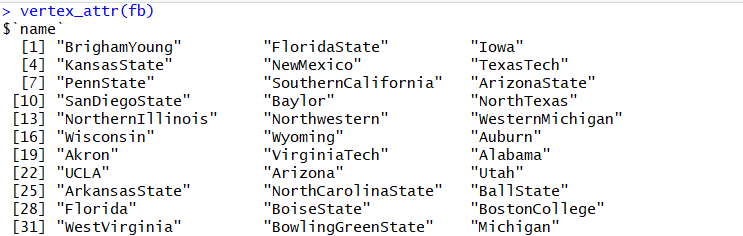


So, we see clusters of authors.

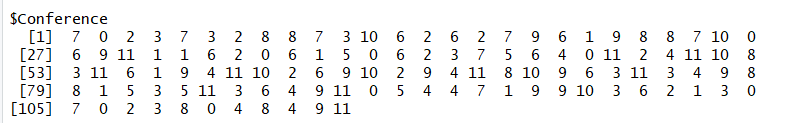
Here is some old data about NCA football conferences:



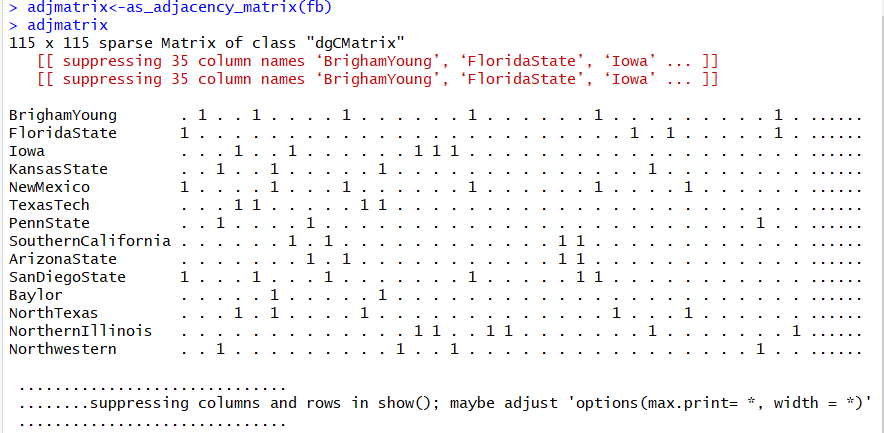
‘Name’ is an attribute



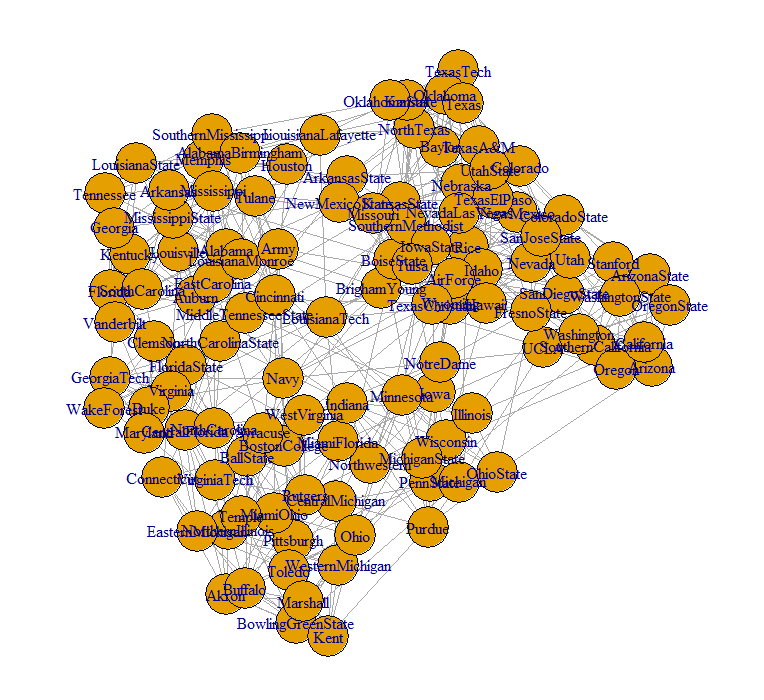
So is ‘Conference’ they belong to



Adjacency matrices can be rather sparse.



* plot(fb)
* Difficult to read due to obscuration

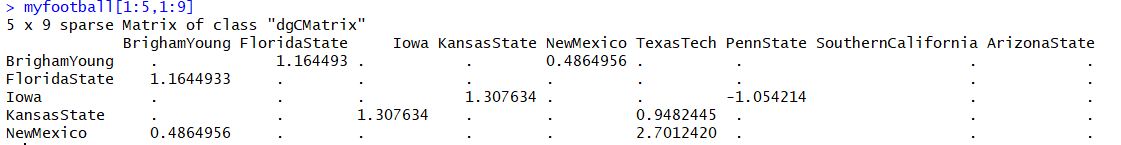


**Axiom: Plotting is your friend!!**

You can add weights to edges and vertices:



Show the weights:

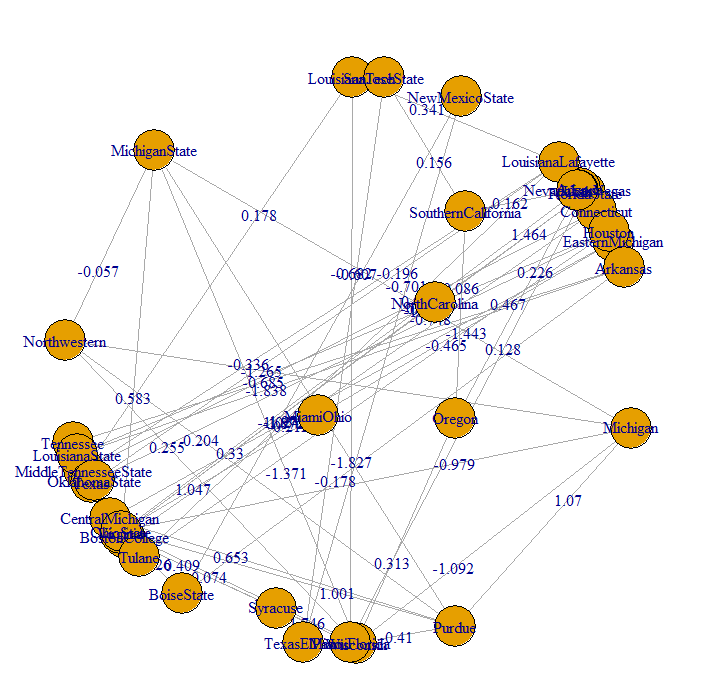


Copy the vertices with positive weight into a subgraph and including the edge weight in the plot (next slide)

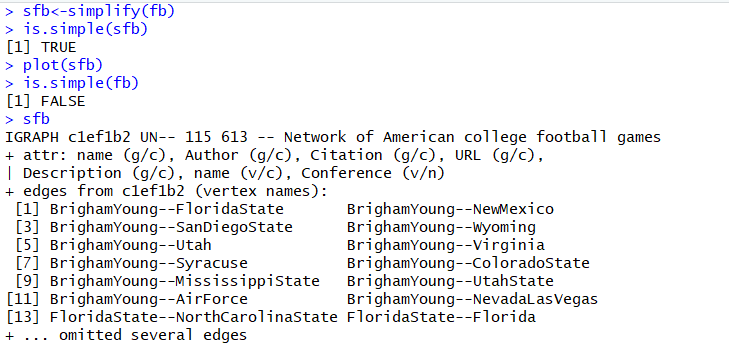


A graph without loops and multiple edges between vertices is called simple. You can check that your graph is simple like so:



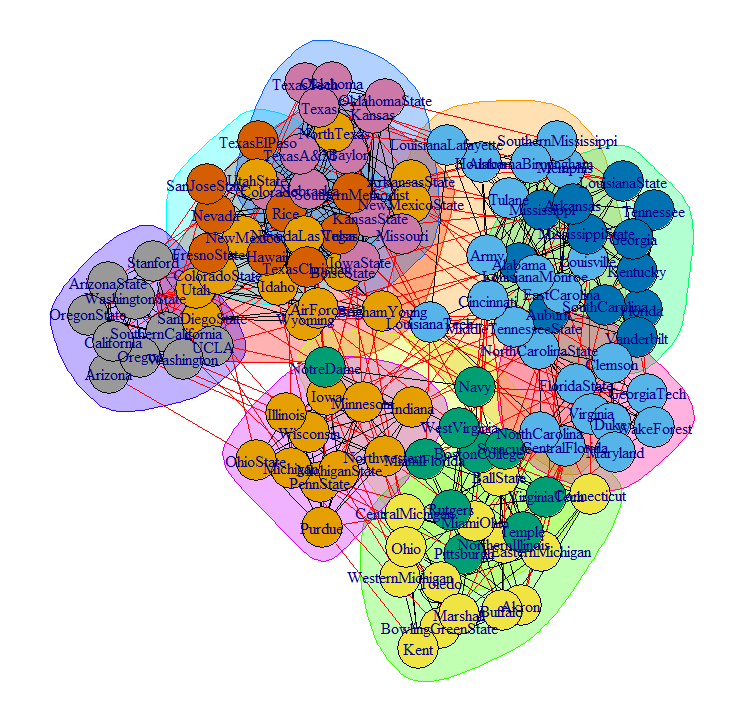


You may turn a graph into a simple graph by using the *simplify* function:



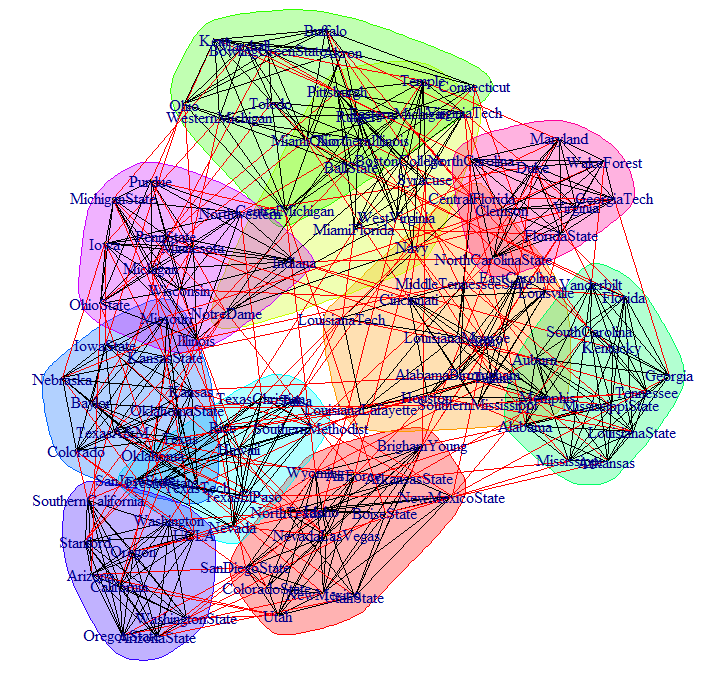
Communities:

* 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:

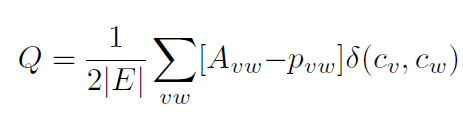
* wcsfb<-walktrap.community(sfb)
* Plot(wcsfb,sfb, vertex.size=0.5, layout=layout.fruchterman.reingold)



* How does this differ from the wc plot?

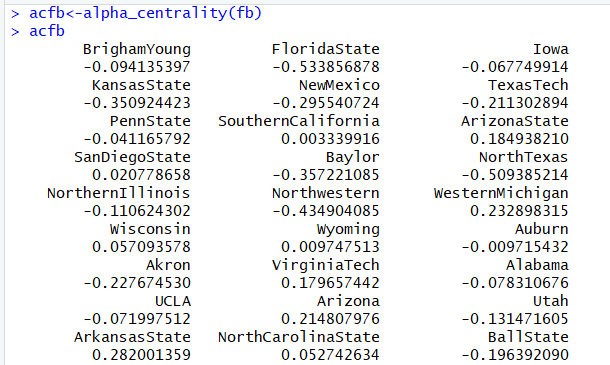
See where different schools are in the two plots

* How to define what is modular?
* Many proposed definitions, here is a popular one:

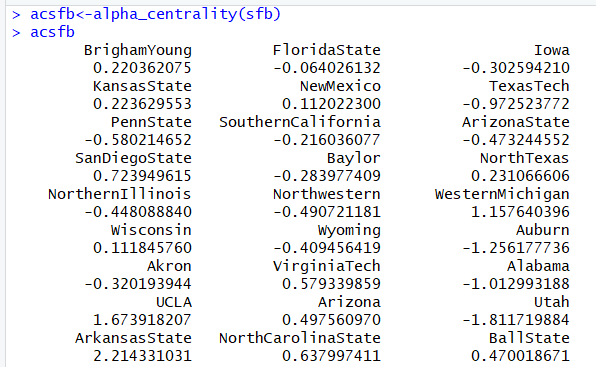


So, it appears there are lots of collaborations. So, let’s see if we can filter some out and get a better idea of what is going on.

*alpha\_centrality* calculates the alpha centrality of some (or all) vertices in a graph. Read about alpha\_centrality in the documentation



Now, look at the alpha\_centrality of sfb:



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:

