

Measuring and Visualizing Networks in R

James M. Cook
University of Maine at Augusta
james.m.cook@maine.edu

Keywords

R; Social Networks; Software; Visualization

1. PRIOR KNOWLEDGE

Previous chapters in this series have:

- introduced the general idea of the social network and its components: relations, ties and nodes,
- distinguished between individual-level and network-level approaches to observing society,
- described different kinds of social networks (directed vs. undirected graphs, networks with dichotomous ties vs. networks with tie strength),
- explained how to represent social networks as edge lists (“edge” = “tie”), adjacency matrices, adjacency lists and graphs, and
- introduced various measurements of the structure of social networks, including characteristics of nodes, ties, paths, subsets of networks and entire networks.

The material in this chapter assumes you have mastered the information in prior chapters. Here, we press forward, considering how a software program called *R* can be used to make the storage, measurement and display of social networks a bit easier.

2. INSTALLING AND USING R

2.1 Installing R

R is a piece of software developed by statisticians, methodologists and programmers for the completion of various tasks in the pursuit of research. R software is used by researchers in the natural sciences, social sciences and humanities, and is available free of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and a full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission from the author and/or a fee.

Copyright James Cook, 2016

charge. R can run on any modern computer with a Linux, Apple Macintosh, or Windows operating system.

From the beginning, R has been designed to be both flexible and open to innovation; as social and natural scientists develop new techniques, they are free to write “packages,” extensions to the software that add new capabilities. To use R for social network analysis, it is necessary to first install R itself, then install a special package named “igraph.”

To install R, follow these steps:

1. Visit the website *r-project.org*.
2. Click the link “download R.” The R Project website is more than a bit cluttered with text; as of October 2016 this link can be found near the top of the R Project home page, at the center.
3. Select a CRAN mirror. “CRAN” is an acronym referring to the Comprehensive R Archive Network, a group of dozens of institutions that make R available for download. R is so popular that if the program were made available on the web page of only a single university or institute, that web page would frequently crash from the strain of so many downloads. The availability of “CRAN Mirrors” – different websites – ensures that the program is always available.
4. After you’ve clicked on a link for a CRAN mirror site near you, click the “Download R” link for the operating system your computer uses, then download the installation file appropriate for your computer.
5. Open the installation file and follow all instructions.

To install the *igraph* package for R, launch the R program and follow these steps specific for different operating systems:

- In any operating system, type “install.packages(“igraph”)” in the R console

window. You may be asked to select a CRAN mirror from which to install the package.

- In the Windows operating system, select *Packages -> Install package(s)...* from the drop-down menus at the top of the screen in the R program. You may be asked to select a CRAN mirror from which to download the package. Find “igraph” in the list of packages, select it as the package to install, and click “OK.”
- In the Apple Macintosh operating system, select *Packages & Data -> Package Installer* from the drop-down menu at the top of the screen in the R program. You may be asked to select a CRAN mirror from which to download the package. Find “igraph” in the list of packages and select it as the package to install. Be sure to select the options “At System Level” and “Install Dependencies” for the most thorough installation possible. Then click “Install Selected.” Finally, visit *Packages & Data -> Package Manager*. If igraph is listed as “not loaded,” click on the checkbox next to igraph to make sure it is loaded.

Finally, the first line of every R script using the igraph package should read as follows in order to make sure that igraph is active and ready to use:

```
library(igraph)
```

What is this “script” that will actively be using igraph? What do “scripts” have to do with social network analysis? The next question addresses these questions.

2.2 Using R: The Console and Script Windows

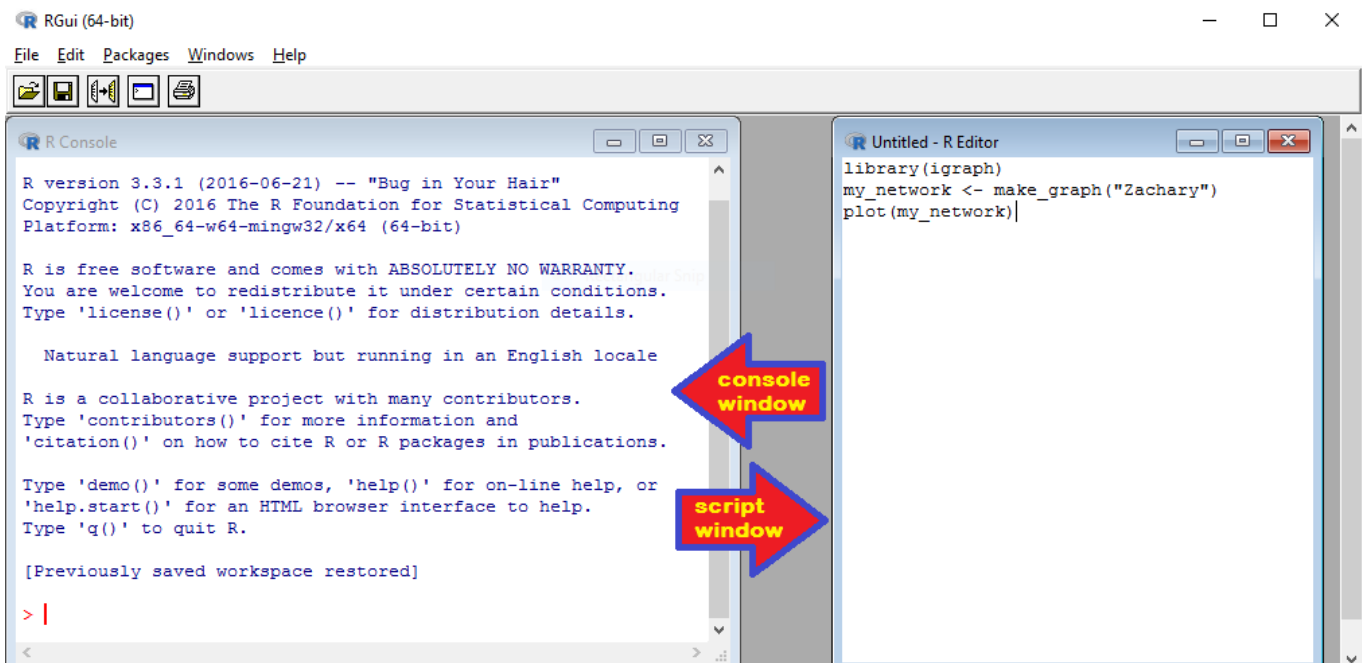
The program R can be tailored to appear in many guises to suit many purposes. But regardless of how it is used, two windows within the program are of primary importance: the Console window and the Script window. These windows are shown in the screen image you’ll find at the bottom of this page.

The Console Window

The Console window is usually the one you’ll find already open when you start the R program. You can type in commands (pieces of text that tell R to know something new or do something new) inside the Console window, and your commands should appear next to the red “>” sign. In order to type commands in the Console window, it has to be selected. All you have to do to select the Console window is hover your computer’s cursor over the Console window and click.

The Console window isn’t just a place where commands are entered; it’s also where the results of your commands will typically be shown. That’s why it’s a good idea to always keep the Console window open when you’re running R.

Console commands can be easily entered one by one, but they’re not saved when you exit a program, and if you need to run your commands multiple times, you’ll need to type your commands in over and over again in



the Console window. The more complicated your goals become in R, the longer your lists of commands will become, and the more of a hassle it will be to type in those commands repeatedly. To reduce the workload of writing commands to R over and over again, we can save them in “scripts.”

The Script Window

A script is nothing more than a sequence of commands. Scripts appear in the Script window, which is typically not open when you start R. To open a new script window in the Windows operating system, select *File* -> *New script* in the drop-down menu at the top of the R program screen. “File” menu option at the top of the R screen, then selecting “New script.” In the Apple Macintosh version of R, scripts are called “documents” instead (an irritating difference without much of a reason, but one you should be able to work through with practice).

The advantage of writing scripts is that you can save them (*File* -> *Save*). Once you have a working script, you can load them (*File* -> *Load*) back in to R and run them over and over and over again. If you want to modify a script to do something slightly different than what you’ve done before, you can save your modified script with a new name (*File* -> *Save as*). Slowly working with scripts and adding new features is a great way to learn analysis and preserve your learning for the future.

3. THE STRUCTURE OF AN R SCRIPT

In an R script, one command is written per line, and the commands are run in order from top to bottom. There are two kinds of R commands: *definitions* and *actions*.

Definitions

R works by defining objects. An object is a name for something you want R to remember, and an object’s definition is the piece of information that you want R to remember whenever the object’s name is used. That’s fairly abstract; let’s show how the definition of an object works in action. Consider the following two commands in R:

```
x <- 7
y <- x + 10
```

These commands each have three parts: 1) the name of the object, 2) what you want the object to be defined as, and 3) an arrow pointing from the definition to the object so that R knows which is which.

The first of these two lines tells R to remember that whenever the object “x” is referred to from now on, “x” is the same as the number 7. That’s pretty simple, but on the next line, R is asked to remember that “y” is the same as x plus 10. R will remember that x refers to the number 7, so y will be remembered as 7 plus 10, or 17.

Objects can be defined as all sorts of things other than numbers, including text...

```
my_name <- “Howard the Duck”
```

... or edge lists...

```
this_network <- graph(edges=c("A","B",
"A","E", "B","C", "C","D", "B","D",
"E","F", "F","A"), directed=FALSE)
```

or even a file somewhere on your computer:

```
my_matrix <- read.csv(file.choose())
```

Reading network files into R is quite a handy trick. We’ll consider this kind of definition in greater detail later in this chapter.

Actions

The other kind of an R command is an *action*. Actions don’t define objects. Instead, they tell R to do something. The first line of any script working with the R package *igraph* is a command...

```
library(igraph)
```

... telling R to take the action of loading *igraph*.

Perhaps the simplest action of all is to print the definition of an object. To do this, simply name the object...

```
y
```

... and R will print the object’s definition in the Console window [in this case, “17”]. If an object is a network and you command it to be printed by simply entering the object’s name...

```
this_network
```

... R will place the object on the screen in text form:

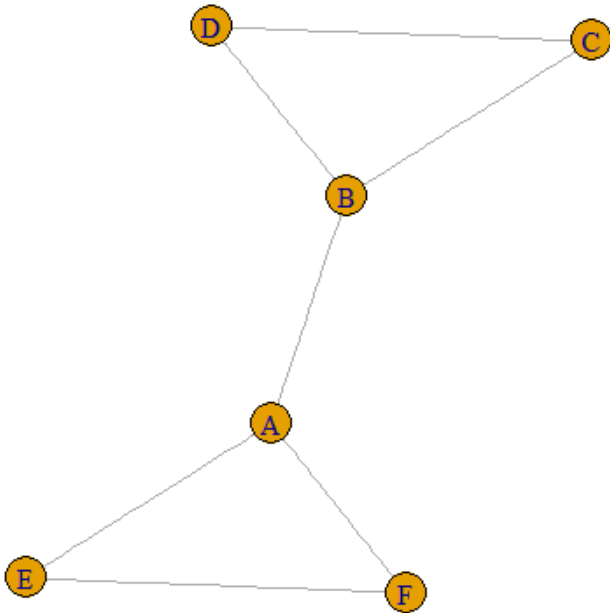
```
> this_network
IGRAPH UN-- 6 7 --
+ attr: name (v/c)
+ edges (vertex names):
[1] A--B A--E B--C C--D B--D E--F A--F
> |
```

To obtain the image of a network’s graph, we’ll use the command `plot`. The `plot` command has many options and tricks up its sleeve, and we’ll look at some of these later in this chapter. For now, it is enough to

know the most basic version of the `plot` command, in which the name of the network object to plot is placed inside parentheses...

```
plot(this_network)
```

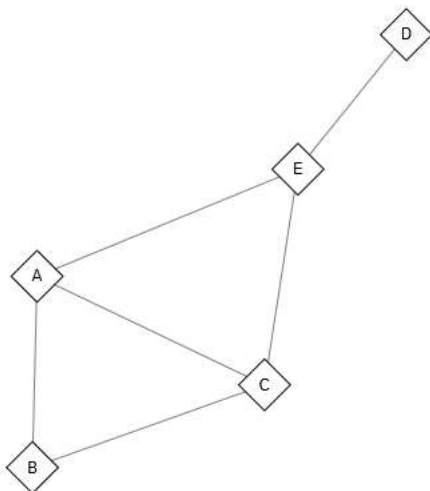
... leading to this result appearing on screen:



The remainder of this chapter is devoted to commands to *define* social network *objects* and to automate the *actions* of social network analysis that you've already learned in previous chapters.

4. LOADING NETWORKS INTO R

In order to work with social networks in R, it is necessary to “load” them in – that is, to define them as objects that R understands. In the section below, we'll consider three different ways to load the following network:



4.1 Typing in an Edge List

When working with a small network, perhaps the easiest way to load a network into R is simply to type it in as an edge list. The script is simple:

```
library(igraph)
network <- graph(edges=c("A","B", "A","C",
"A","E", "B","C", "C","E", "E","D"),
directed=FALSE)
```

To begin our script, as always, we load the `igraph` package. The second sentence defines an object called “network” (you could give it any name) as a graph with edges. What are the edges? They're listed, with the names of the nodes placed in quotes. This list...

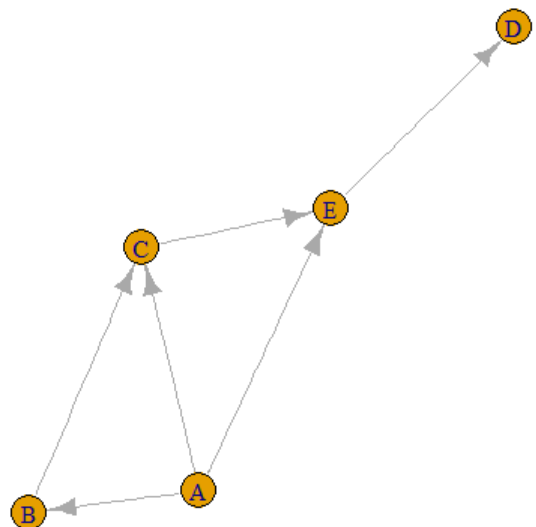
```
c("A","B", "A","C", "A","E", "B","C",
"C","E", "E","D")
```

... is the computer's way of understanding the following edge list:

- A,B
- A,C
- A,E
- B,C
- C,E
- E,D

We know just by looking at the network graph shown in this section that it is an undirected network (no arrows means no direction to ties). For that reason, we add the `directed=FALSE` option to the command.

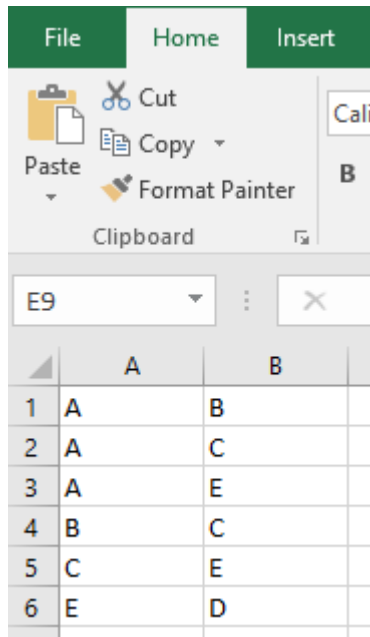
What would happen if we changed `directed = FALSE` to `directed = TRUE`, but kept all other parts of the script the same? Details matter. If we plotted the resulting network with the command `plot(network)`, the graph would look like this:



4.2 Importing an Edge List into R

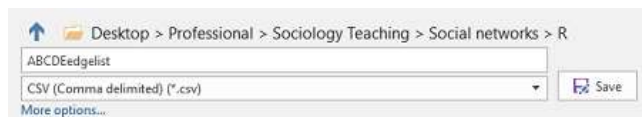
Typing in an edge list is easy when you only have a few ties (“edge” = “tie”). But what happens if you have a very large network with a large set of ties? In such a case, typing isn’t the best option. Instead, we might want to import an edge list from another program.

In Microsoft Excel, an edge list could be placed in the first two columns of a spreadsheet, like this:



	A	B
1	A	B
2	A	C
3	A	E
4	B	C
5	C	E
6	E	D

This edge list can be saved in a special format called CSV by using the command *File -> Save As* and then selecting the CSV option from a drop-down menu of formats:

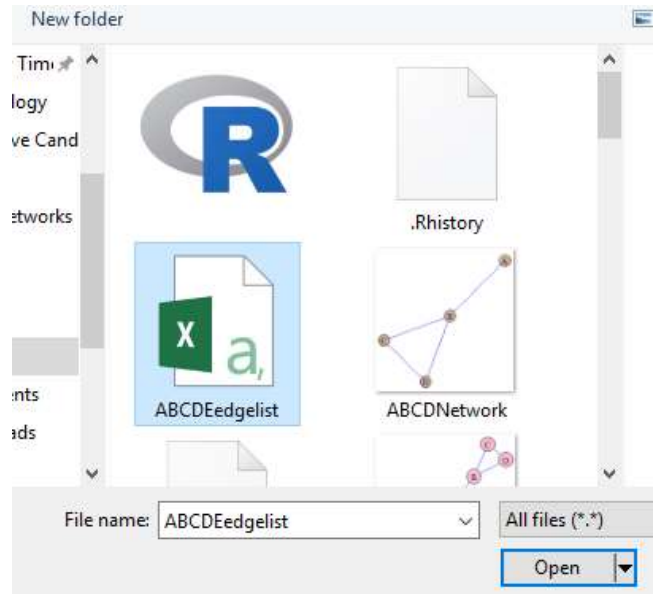


If you’re in Google Sheets, you can follow a similar procedure, saving a CSV file to your computer by using the *File -> Download as -> Comma-separated values* command.

We’re using a small edge list here as an example to save space. But there are many social networks with many, many ties, leading to a very, very long edge list. No matter how small or big your edge list is, the script you’ll use to load your edge list into R is really quite small:

```
library(igraph)
my_data <- read.csv(file.choose(),
header=FALSE)
my_network <- graph.data.frame(my_data,
directed=FALSE)
```

The first line of this script is, as always, used to load the *igraph* package into R. The second line does something special. It creates an object called *my_data*, then tells R the object will be defined by a csv file by using the command *read.csv*. The next part of the command, *file.choose*, tells R to let you find and upload your *.csv* file whenever you run the program:

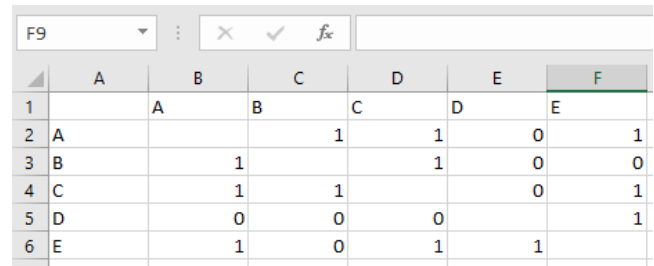


Finally, the *header=FALSE* option in the second line of the script tells R that the first row doesn’t have labels; it just contains a tie like all the other rows.

The third and last command of the script is a bit of magic, using the command *graph.data.frame* to convince R to read your data as a social network, not just any old csv file. The object at the end, *my_network*, will be understood as a social network, and if you added the command *plot(my_network)* at the end, you’d see a network graph appear.

4.3 Importing an Adjacency Matrix

Edge lists can get big, but adjacency matrices get even bigger. In Microsoft Excel, the adjacency matrix for the same network we’ve been using throughout this section looks like this:



	A	B	C	D	E	F
1						
2	A			1	1	0
3	B		1		1	0
4	C		1	1		0
5	D		0	0	0	1
6	E		1	0	1	1

As an adjacency matrix, this network fills three times more cells than it does in an edge list format. Typing in a matrix by hand into R is unwieldy at almost any size. Instead, it's best to build an adjacency matrix using a spreadsheet program such as Microsoft Excel or Google Sheets. Use the first row and the first column to contain node labels. Then save your matrix file in csv format, using the same technique described for an edge list file in Section 4.2 of this chapter.

Again, we're using a small adjacency matrix here as an example to save space. But no matter how small or big your matrix is, the script you'll use to load your edge list into R is short and sweet:

```
library(igraph)
my_data <- read.csv(file.choose(),
header=TRUE, row.names=1)
my_matrix <- as.matrix(my_data)
my_network <- graph.adjacency(my_matrix,
mode="undirected",diag=FALSE)
```

Line 1 of this script loads the package igraph, as always.

Line 2 allows you to choose a csv file on your computer to load into R, then defines it as the object "my_data." Line 2 is a bit different here than for our edge list in Section 4.2, because the adjacency matrix contains node labels in the first row and column (or as R calls them, "headers"). To recognize this fact, we add the option header=TRUE. To show R where the names of the nodes are, we point to the first row with the added option row.names=1.

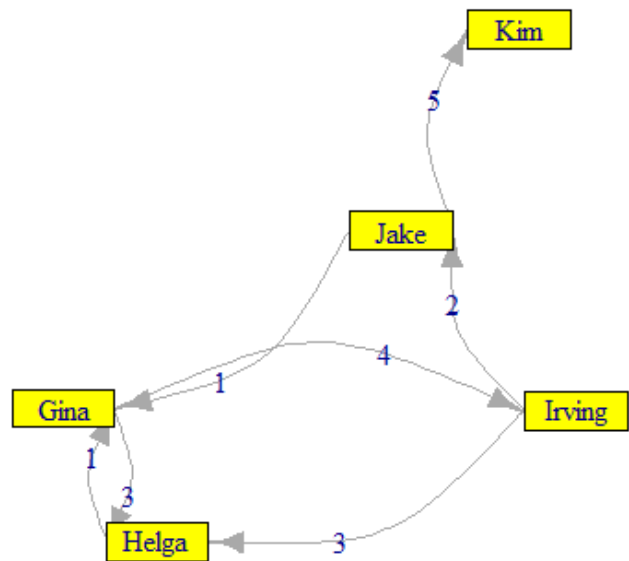
Line 3 and Line 4 take the raw data format of the csv file and tell R how to interpret it, first as a matrix in general and then a special kind of matrix: the adjacency matrix. Line 3 turns the general object "my_data" into a matrix object called "my_matrix." Line 4 turns the matrix object "my_matrix" into a network, "my_network." Line 4 is where we need to tell R what kind of network we're dealing with. This is an undirected network (mode="undirected"), one without tie strength (weighted=FALSE), and one in which the diagonal values are without meaning (diag=FALSE).

You may have noticed that sometimes these options use different kinds of language. The script to load an edge list in Section 4.2 tells us the resulting network is undirected by using the command option directed=FALSE, while the script to load an adjacency matrix in this section conveys the same information by using the command option (mode="undirected"). These differences are an unfortunate consequence of a

happy situation: R is a free piece of software written by many people. When different people work on commands, sometimes they end up writing options for the commands differently. There's no deeper reason for such details; we just have to work with the commands as they're given.

4.4 Importing Networks with Tie Strength

The network we worked with in Sections 4.2 and 4.3 had no tie strength associated with it. But what if we wanted to work with a network that featured tie strength, as with the directed network you see below? Imagine this network represents a fairly violent reality TV show in which the relation is "number of times ___ slapped ___'s face in the past month."



Fortunately, it's fairly easy to add references to tie strength using R when we import either an edge list or an adjacency matrix.

Tie Strength in Imported Edge Lists

In a spreadsheet program, we could represent the "slapping" network like this:

	A	B	C
1	Slapper	Slappee	Times
2	Gina	Helga	3
3	Helga	Gina	1
4	Gina	Irving	4
5	Irving	Helga	3
6	Irving	Jake	2
7	Jake	Gina	1
8	Jake	Kim	5

To import an edge list with tie strength, we've added a new, third column to our csv file. We'll also need to label that third column so we tell R where to find tie strength. Looking at the spreadsheet, you can see the first row now features labels for the columns: "Slapper," "Slappee," and "Times." When we load this csv file in to R using the script shown below, we've added in the option `header=TRUE` in Line 2 so that R can automatically understand those column labels and refer to them.

```
library(igraph)
slapping_data <- read.csv(file.choose(),
header=TRUE)
slapping_network <-
graph.data.frame(slapping_data,
directed=TRUE)
slapping_adjacency_network <-
get.adjacency(slapping_network,
attr='Times',sparse=FALSE)
slapping_adjacency_network
plot(slapping_network,edge.label <-
E(slapping_network)$Times,edge.curved=TRUE)
```

R also automatically understands in Line 3 that the first two columns of `slapping_data` describe the edge list. If we don't do anything else, R will simply ignore the third column, "Times." (Also notice that we've set `directed=TRUE` because this network is a directed network).

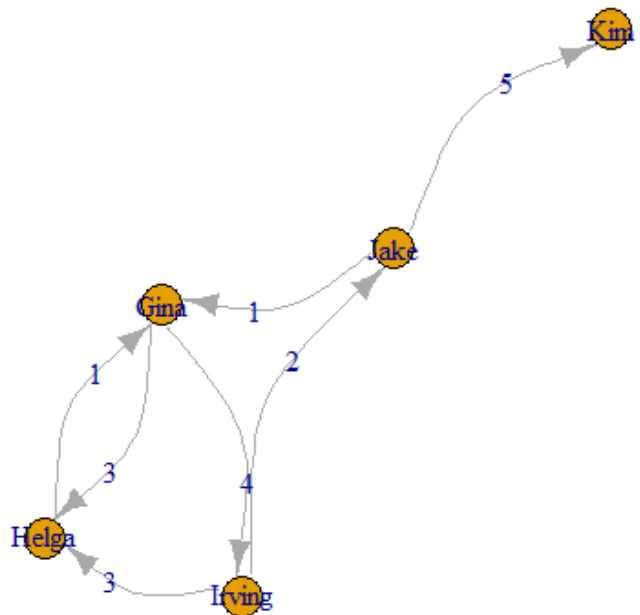
But of course, we don't want to ignore the third column. We want tie strength to be recognized! There are two ways to do that. To add these tie strengths to a network in the form of an adjacency matrix, we'll add a fourth command:

```
slapping_adjacency_network <-
get.adjacency(slapping_network,attr='Times',
,sparse=FALSE)
```

The phrase `get.adjacency` tells R that we're setting up an adjacency matrix. The option `attr='Times'` tells R that the values in the matrix should be taken from the column "Times." The option `sparse=FALSE` tells R to fill all the cells with zero if there are no ties. When we ask for the adjacency matrix to be printed by issuing the command `slapping_adjacency_network` in the next line, this is what we see:

```
> slapping_adjacency_network
      Gina Helga Irving Jake Kim
Gina   0     3     4     0     0
Helga  1     0     0     0     0
Irving 0     3     0     2     0
Jake   1     0     0     0     5
Kim    0     0     0     0     0
```

The last command in the script includes tie strength in a network graph. The option `edge.label=E(slapping_network)$Times` assigns each tie a label according to the value of "Times" for that tie (the "E()" and the "\$" are just part of the syntax). The option `edge.curved=TRUE` is used to curve the ties in the network. This is necessary because in a directional network, ties going different ways between nodes may have different tie strengths. To distinguish between different ties of different strengths between the same two nodes, we need to curve them away from one another so the tie strength labels clearly show which strength is associated with what tie. The result generated by R looks like this:



Tie Strength in Imported Adjacency Matrices

If we're starting out with an adjacency matrix in a spreadsheet, including tie strength information is as simple as including that information in the value of the cells to begin with. To extend the example of our network of reality television slaps, the matrix might look like this:

	A	B	C	D	E	F
1		Gina	Helga	Irving	Jake	Kim
2	Gina		3	4	0	0
3	Helga	1		0	0	0
4	Irving	0	3		2	0
5	Jake	1	0	0		5
6	Kim	0	0	0	0	

The first few lines of a script to work with a csv file generated from this spreadsheet should look familiar by now:

```
library(igraph)
slap_data <- read.csv(file.choose(),
header=TRUE,row.names=1)
slap_matrix <- as.matrix(slap_data)
slap_network <- graph.adjacency(slap_matrix,
mode="directed",weighted=TRUE,diag=FALSE)
plot(slap_network,
edge.label=E(slap_network)$weight,
edge.curved=TRUE)
```

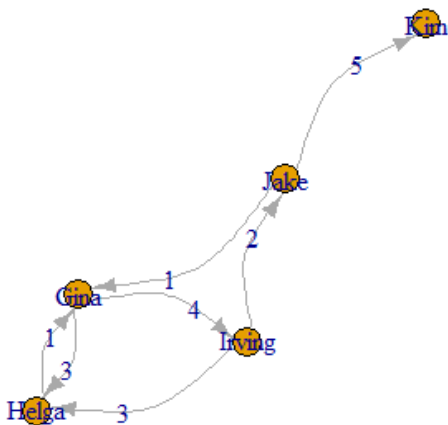
Line 1: loads the igraph package.

Line 2: loads the csv file, with node names

Line 3: defines the csv file as a matrix

Line 4: defines the matrix as a network. Here, two of the options are different. The option `mode="directed"` tells us that this is a directed network, and the option `weighted=TRUE` tells us that the values of the cells in the adjacency matrix contain measures of tie strength.

Line 5: issues a command to plot the network in graph form. Because ties have strength, we use the option `edge.label=E(slap_network)$weight` to place labels on the ties. Because this is a directed network and we have tie strength to express, we'll need to curve the ties to separate them with the `edge.curved=TRUE` option. The resulting graph appears like this:



The graph produced by this script looks much like the graph produced by importing an edge list, which shows that there is more than one way to work with R and get to a good answer. Which strategy you choose to work with network data can often be a matter of your own comfort. However, it's always good to know more than one way to accomplish a task.

5. MEASURING NETWORKS IN R

In a previous chapter, you learned how to make various measurements of a social network by hand. R can automate these measurements for you.

Imagine the following *undirected* network:

A,B
A,E,
B,C
B,D
C,D
D,E
E,F

The following script quickly and efficiently calculates a variety of network measurements for this undirected network, which the script here names "net":

```
library(igraph)
net <- graph(edges=c("A","B", "B","C",
"C","D", "B","D", "A","E", "D","E",
"E","F"), directed=FALSE)

betweenness(net, directed = FALSE)
closeness(net)
degree(net, mode="all")
edge_density(net)
diameter(net)
distances(net, mode = "all")
cliques(net, min=3)
articulation_points(net)
components(net)
```

Most of these measurements of the network are so straightforward as to be self-explanatory: `closeness(net)` will calculate closeness centrality for all nodes in the network called "net," for instance. A few notes are required, however:

- The command `betweenness` requires the inclusion of the option `directed=FALSE` to indicate an undirected network.
- The command `degree` requires the inclusion of the option `mode="all"` to measure degree in an undirected network.
- `edge_density(net)` measures the density of a network.
- `distances(net, mode = "all")` produces a table showing all distances between all pairs of nodes in a network.
- `cliques(net, min=3)` produces a list of all nodes of size 3. For a list of all nodes of size 4, set `min=4`, and so on.
- `articulation_points(net)` produces a list of all cut points in the network called “net.” “Articulation point” is a synonym for “cut point.”
- `components(net)` produces a list of all connected components in a network.

Now imagine the following *directed* network:

G,H
 G,K
 H,I
 H,J
 I,J
 J,I
 K,H
 K,I

The following script quickly and efficiently calculates a variety of network measurements for this directed network, which the script here names “net2”:

```
library(igraph)
net2 <- graph(edges=c("G","H", "H","I",
  "I","J", "J","I", "H","J", "G","K", "K","H",
  "K","I"), directed=TRUE)
```

```
betweenness(net2, directed = TRUE)
closeness(net2)
degree(net2, mode="out")
degree(net2, mode="in")

edge_density(net2)
diameter(net2)
```

```
distances(net2, mode = "out")
articulation_points(net2)
components(net2, mode = "weak")
components(net2, mode = "strong")
```

Again, most of these commands are so straightforward as to be self-explanatory, but a few notes are required for full explanation:

- The command `betweenness` requires the inclusion of the option `directed=TRUE` to indicate an directed network.
- The command `degree` requires the inclusion of the option `mode="out"` to measure outdegree in an undirected network. To measure indegree, enter a separate `degree` command with the option `mode="in."`
- `distances(net, mode = "out")` produces a table showing all distances *from* row nodes *to* column nodes in a directed network.
- In a directed network, there are two kinds of connected components. A *weakly connected component* created with the option `mode = "weak"` in the `components` command is a set of nodes within which each node can *either* reach *or* be reached by every other node along some path.
- On the other hand, a *strongly connected component* created with the option `mode = "strong"` in the `components` command is a set of nodes within which each node can *both* reach *and* be reached by every other node along some path.

6. VISUALIZATION OPTIONS IN R

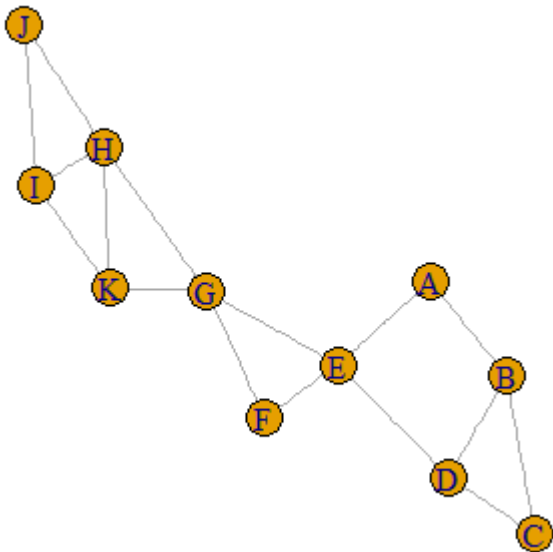
The `plot()` command is your key to creating a network graph, but there are many different ways to visualize network graphs. Visualization is the process by which the layout of a network graph is arranged in order to highlight certain aspects of social network structure. The choices you make in visualizing a network can help shape the conclusions that people draw from looking at your network.

In this section, we’ll be drawing on the following network, which we’ll call “biggnetwork” in R:

A,B
 B,C
 C,D
 B,D
 A,E
 D,E

E,F
 E,G
 F,G
 G,H
 H,I
 I,J
 H,J
 G,K
 K,H
 K,I

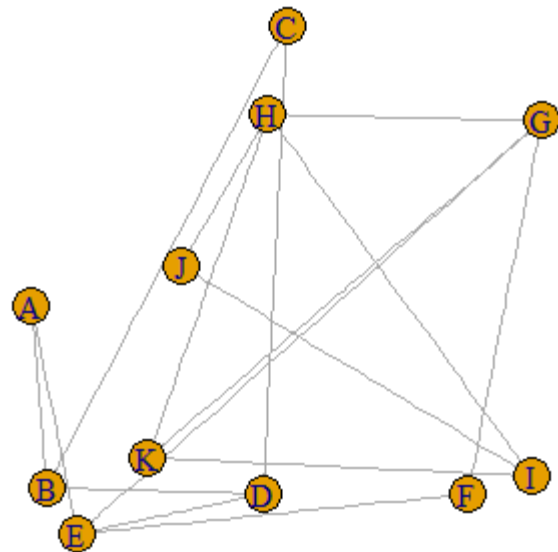
It's hard to see what the structure of the network might be just from looking at the network as an edge list, which is one reason why we create network graphs. If we enter this edge list into R as an undirected network and simply type the command `plot(biggnetwork)`, we'll get a graph that looks like this:



But that's not the only way we could visualize this network. There are choices available to you in R that can change the way a network is seen.

6.1 Options for Layout: Placement of Nodes

Consider the above network graph. Which would you say are the most central nodes in the network? Most people would identify nodes G, F, and E as central, in no small part because these nodes are in the physical center of the network graph image. Now let's look at another network graph. Which nodes are at the center of the following network?



It's quite a bit harder to say who's central, isn't it? Yet *this is the exact same network as the one before it, with exactly the same structure of ties between nodes.* All that has changed is the positioning of the nodes. Appearance matters. The graph you see above was generated using the `plot(biggnetwork, layout=layout.random)` option, an option that literally places nodes at random (within the limitation that they do not overlap one another). The random layout option often obscures underlying patterns in the graph and should usually be avoided.

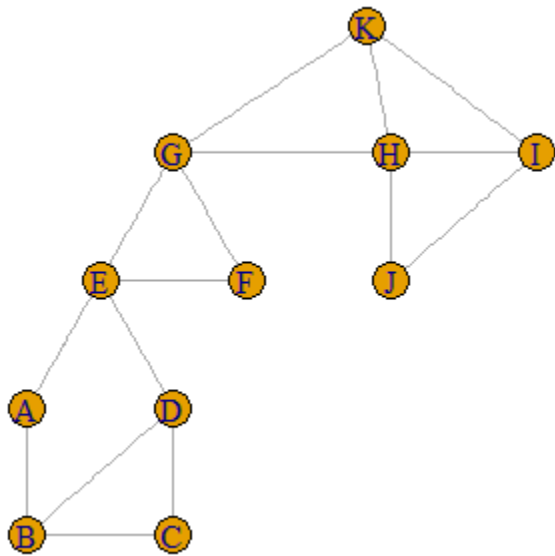
A standard alternative to the random layout function is the "Fruchterman Reingold" layout, written in R as the `plot(biggnetwork, layout=layout.random)` option. This is one of the "spring embedded" layout formats that for smaller graphs are used by default in R. The idea is that ties between nodes are like springs that pull nodes to one another, while nodes otherwise (like electrons) repel one another. In this way, clusters of nodes that have more ties among themselves tend to appear closer together.

Spring-embedding is not the only option available. Consider the "Reingold Tilford" layout, which picks one node and arranges all other nodes according to their network distance from that one chosen node. written in R using an option with the following structure:

```
plot(biggnetwork,
      layout=layout.reingold.tilford(biggnetwork,
                                     root="K"))
```

The `root="K"` part of the option tells R which of the nodes you'd like to be the root – that is the node from which all distances are calculated. This is important.

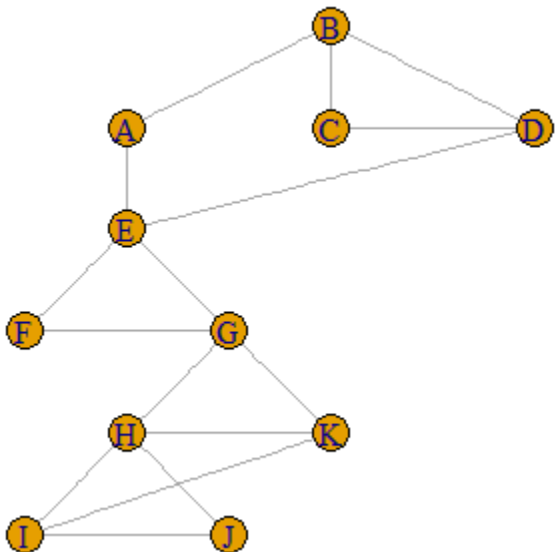
When node K is the root, the Reingold Tilford layout for our network looks like this:



We can use the vertical rows of nodes in this graph to immediately tell us something interesting about their relationship to node K. G, H, and I are in the first row because they are at distance 1 from K. E, F and J are in the second row, at distance 2 from K. Nodes A and D are in the third row, at distance 3 from K. Nodes B and C are in the fourth row at distance 4 from K.

That's all well and good, but the whole network is arranged in terms of node K. What happens if we pick another node to be our central node? What about node B?

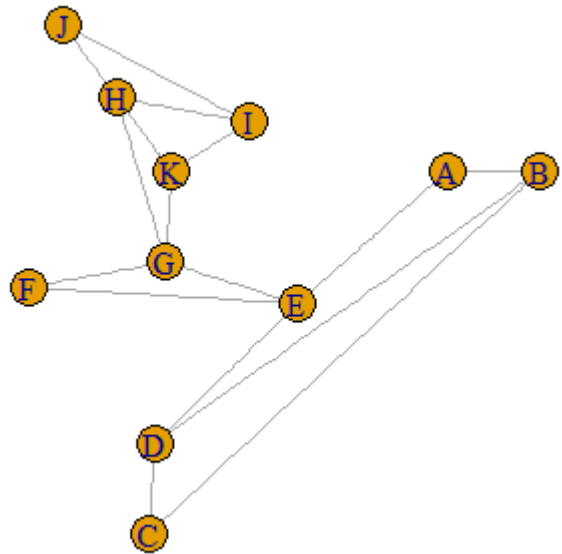
```
plot(biggnetwork,
layout=layout.reingold.tilford(biggnetwork,
root="B")
```



This layout is no less truthful. All layouts are real. But each layout reveals another aspect of the social reality of the social network the viewer sees. This Reingold Tilson layout, laid out according to distance from node B, looks different than the one before and might lead to different conclusions.

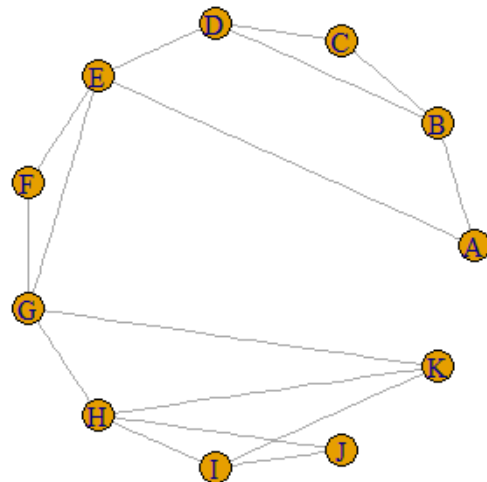
Sometimes revealing truth is not the only concern. Sometimes, we just want a network to *look cool*. What would happen if we wanted to take the Reingold Tilson layout, but put successive layers around the central node in a *circle*? That would *look so cool*:

```
plot(biggnetwork,
layout=layout.reingold.tilford(biggnetwork,
root="K",circular=TRUE)
```



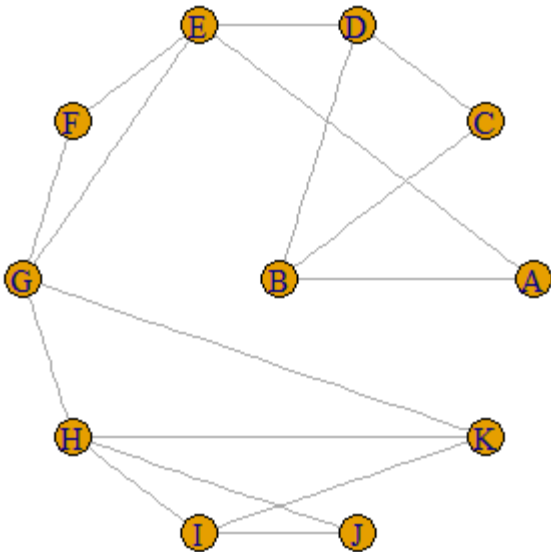
Cool or cluttered? This is where pure science breaks down and aesthetic considerations creep in.

Other layout options include a purely circular layout:
`plot(biggnetwork, layout=layout.circle)`



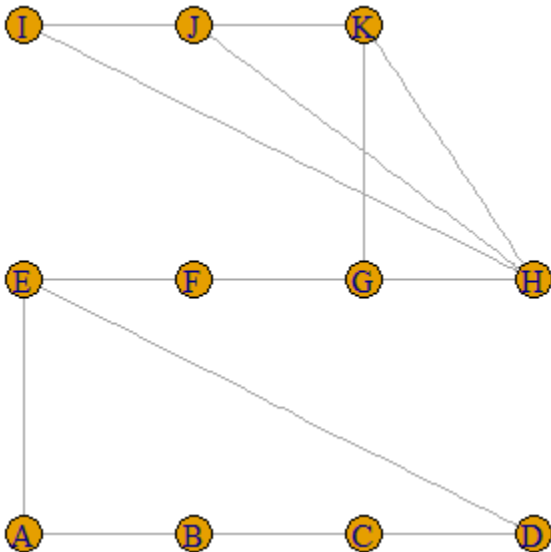
... and a star layout, which is like a circle but allows one node to sit at the middle:

```
plot(biggnetwork,
layout=layout.star(biggnetwork, center="B"))
```



You could use a grid to evenly space nodes:

```
plot(biggnetwork, layout=layout.grid)
```



And if you have a really large network, try the large graph layout algorithm (a.k.a. “LGL”):

```
plot(biggnetwork, layout=layout.lgl)
```

All of these graphs represent the same underlying network. Which layout strategy reveals the “true” network structure the best? I leave that to your consideration.

P.S. If none of these layout plans strikes you as the perfect one for your network, you can always go the

“tk” route. If you are running R on a Windows computer, try changing “plot” to “tkplot”:

```
tkplot(biggnetwork)
```

You’ll be taken to an interactive screen on which you can drag and drop nodes wherever you like them, until you think you’ve got your layout just right to tell the story you want to tell.

6.2 Options for Layout: Features of Nodes

Layout is not the only feature of visualization that you can change. There are a number of ways you can change the look of your nodes, by giving them a different color, shape or size:

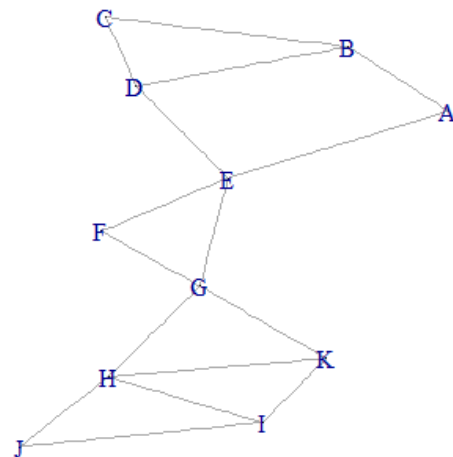
```
plot(biggnetwork, vertex.size=25, vertex.color="yellow", vertex.shape="sphere")
```

Default size is 15; any number below 15 will be smaller than normal, and numbers above 15 will be larger. Try changing color or shape names around. Available shapes include “circle”, “square”, “csquare”, “rectangle”, “crectangle”, “vrectangle,” “sphere,” and “none” (for labels-only nodes). The possibilities for color are nearly limitless (particularly if you use RGB color specifications like #320D44 (a deep purple) or #976B29 (the tan color some M&M candies used to be).

If you’re going to set size or color, why not have those characteristics show off some structural attribute of nodes? The following command sets size according to degree and color according to closeness centrality:

```
plot(biggnetwork, vertex.size=degree(biggnetwork), vertex.color=closeness(biggnetwork), vertex.shape="sphere")
```

The result looks like this:



“Wait a minute,” you may be thinking to yourself. “Where are the nodes? Where is the variation in size?” Let’s find closeness and degree values for the nodes to find out:

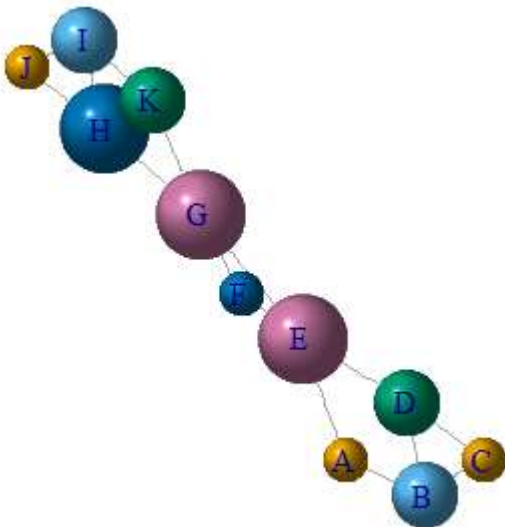
```
closeness(biggnetwork)
degree(biggnetwork)
```

```
> closeness(biggnetwork)
      A      B      C      D
0.04166667 0.03448276 0.03333333 0.04347826
      E      F      G      H
0.05555556 0.04545455 0.05555556 0.04545455
      I      J      K
0.03448276 0.03333333 0.04347826
> degree(biggnetwork, mode="all")
A B C D E F G H I J K
2 3 2 3 4 2 4 4 3 2 3
```

Aha! The values of degree tend to be very small (smaller than the standard node size of 15), and the values of closeness tend to be really, really small. To get more variation, let’s make our numbers bigger through the power of multiplication:

```
plot(biggnetwork, vertex.size=degree(biggnetwork)*10,
      vertex.color=closeness(biggnetwork)*1000,
      vertex.shape="sphere")
```

And now we get something a bit more interesting:



As you can see, there are many options for visualizing your network (and actually, these are only the beginning). Try many different visualizations as you think about presenting your network graphs. Some tinkering is part of the process: this is where play comes in.

7. GLOSSARY

articulation point: a synonym for cut point.

command: a piece of text that tells R to know something new or do something new.

connected component: set of nodes in an undirected network within which each node can reach every other node along some path.

console window: area of the R screen in which individual commands may be typed, and in which results of commands appear.

definition: the piece of information that R remembers to associate with an object.

edge: a synonym for tie.

object: a name for something you want R to remember.

script window: area of the R screen in which commands are entered one per line and executed in order from top to bottom.

strongly connected component: set of nodes in a directed network within which each node can *both* reach *and* be reached by every other node along some path.

visualization: the process by which the layout of a network graph is arranged in order to highlight certain aspects of social network structure.

weakly connected component: set of nodes in a directed network within which each node can *either* reach *or* be reached by every other node along some path.