Introduction to Network Analysis in `igraph`

Cynthia S. Q. Siew

2024-10-18

Contents

Introduction to Network Analysis in `igraph`

0.0 What is this book about?

Network Science is a branch of complexity science that uses methods from graph theory to study the structure and dynamics of complex systems from a wide range of topics, such as the Internet, social networks, and ecological systems. In recent years, network science approaches have achieved some level of interest from the social sciences. In my own discipline of psychology, network science has been used to study the structure of a variety of psychological networks, ranging from the human brain, cognitive and language networks, social systems, and psychometric data. The aim of this book is to introduce the basics of network analysis using the `igraph` package in R, since R is a programming language that most psychologists are already familiar with. Hence, this book is positioned more as a "how-to" book, and gets into the specifics of the code and programming rather than explaining how network measures can be used to explore research topics in the social sciences (for this, I strongly recommend Thomas Hills' Behavioral Network Science).

0.1 Why write this book?

Although many researchers find networks interesting and want to adopt some aspects of the network science framework into their own work, I have observed that there is a gap between that desire and the practical knowledge of how to go about doing that. In my teaching, I have also observed that students in my PL4246 Networks in Psychology course frequently struggle with the technical/programming aspects of network analysis in the R environment. As far as I can tell, there is no dedicated textbook or open workshops or freely available online resources dedicated to teaching people about how to use `igraph` for network analysis (if I am wrong, please let me know!). Personally, I have had to learn how to use `igraph` for my own research through trial

and error (mostly error), and by reading the wonderful online documentation (https://r.igraph.org/reference/index.html), over a period of several years since my PhD days (and I still think that there are loads more to learn!). The `igraph` package is rich, dense, and full of useful functions and tricks, but it is difficult to learn them readily as an outsider. By writing this book, I hope that this gap can be narrowed, especially for my students and for anyone who is "network-curious", but struggle to have a concrete sense of what these networks look like and what useful information can be extracted from them.

0.2 Acknowledgements

This book is dedicated to all my PL4246 students, past and present. I have learned tremendously about the limits of my own knowledge and gaps in my teaching strategy and approach through their fierce commitment to learn about network analysis. I would like to acknowledge their contributions in shaping the structure and contents of this book.

I would also like to acknowledge the developers of `igraph` for their hard work in developing and maintaining the package over all these years.

If you have any comments about this book or spot any glaring errors (which I am sure exists in troves since this is still an evolving and growing project!), please write to me at cynthia@nus.edu.sg. Thank you :)

0.3 About the author

Dr. Cynthia Siew is an Assistant Professor of Psychology at the National University of Singapore. She is a psycholinguist and cognitive scientist who uses network analysis to study cognitive structures, such as the mental lexicon and semantic memory. To learn more about her research and teaching, please see http://hello.csqsiew.xyz/.

0.4 Table of Contents

Section 1: Foundations for Network Analysis

Section 2: Describing the Network

Section 3: Modeling Networks

Section 4: Beyond `igraph`

Bonus Chapters

Chapter 1

Chapter 1: Installation and set up

1.1 Installing R and RStudio

Before we can get started with network analysis, we need to prepare a computing environment for doing so. We will analyze networks using the `igraph` R package, and so the very first step is to install both R and RStudio into the device that you wish to use for network analysis. It is important to note that R and RStudio are two different things that you have to install. R is the programming language itself (like Python or C++) whereas RStudio is the software application that we will use to interact with R. RStudio is known as an *integrated development environment* (other examples of IDEs are Visual Studio and PyCharm).

First, install R from https://www.r-project.org/. Note that you will have to choose a CRAN mirror that you wish to download R from - which one you pick does not matter. Since I am based in Singapore I typically choose one from nearby regions (Taiwan, Korea, Japan, Indonesia, or Worldwide). Download the latest version.

Then, install RStudio from https://posit.co/. Download the correct RStudio Desktop for your computer OS. (https://posit.co/download/rstudio-desktop/)

Note that both R and RStudio are free downloads. You should not need to pay for anything.

1.2 RStudio Cloud

If you do not have a computer/laptop that you are able to install programs on, you can try using RStudio in the cloud. Basically, you can run RStudio on the

internet without downloading anything into your computer.

The main consideration is that the free version has a cap on usage. Below is reproduced from their website (https://posit.cloud/plans/free):

> "Compute hours represent how much time a project is open or running with a particular configuration, computed as (RAM + CPUs allocated) / 2 x hours. For example, working with a project for 1 hour with 1 GB of RAM and 1 CPU allocated consumes 1 project hour. The Cloud Free plan has a cap each month of 25 project hours. Once you reach the cap, you can no longer open or create projects during your current month."

1.3 Installing R packages

Now that you have R and RStudio set up, let's try to install the R packages that we will use in this book. First, what is a "R package"? Sometimes they are known as libraries, but you can think of packages/libraries as a collection of R functions that do specialized things. Because we make use of several functions from the `igraph` package to do network analysis, we need to first download `igraph`.

To download `igraph`, open RStudio and type the following into your Console (the panel with the ">" symbol - this is where you type in instructions for R to execute).

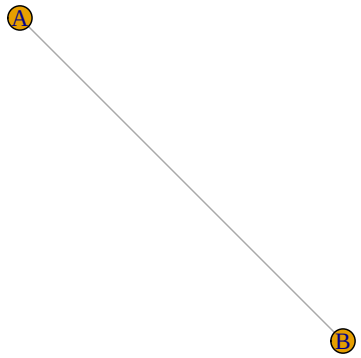
```
install.packages("igraph")
```

Be patient and wait for the package to finish downloading. After it is downloaded, we can only start using the functions in `igraph` *after* we have loaded the library into our workspace. To do this, type the following into the Console:

```
library(igraph)
```

If everything is working fine, you should see a plot of two connected nodes when you run the following line of code in the Console:

```
plot(graph_from_literal(A-B))
```



Important! You only need to install a package **ONCE** (unless you wish to update it). But you will always need to load the package at the start of the R session (i.e., run `library(igraph)` in the Console whenever you open RStudio) so that you can use these special network functions.

1.3.1 Exercise

1. Try installing the package called `tidyverse`. `tidyverse` is a powerful set of packages that enable people to do cool data science things in R. It is highly recommended to learn it if you are interested in becoming a proficient user of R. See https://www.tidyverse.org/ for more information.

2. You know if you have succeeded if you are able to load `tidyverse` into your workspace. What is the code needed to do this?

Chapter 2

Chapter 2: Programming and data science basics

2.1 The RStudio Environment

Recall from the previous chapter that R is a programming language, whereas RStudio is a program that lets you program in R in an efficient way (it is an *integrated development environment*).

Once you have installed R and RStudio into your computer, you only need to open RStudio (not R) to get started.

2.1.1 Setting up a Project

A good practice before starting any kind of programming foray is to create a dedicated folder on your computer for that project, and then create a new R project that is associated with that folder. You can think of a R project as a dedicated workspace for everything that you will do in that context, which makes it easy to save your progress and return to the same state on a different day.

To create a new Project, simply go to *File -> New Project*. Depending on whether you already have a project folder set up in your computer, you would choose either "New Directory" (so you create both a new folder and project file) or "Existing Directory" (so you navigate to the folder and create a project file for it).

You should notice in your folder a new file called `your-folder-name.Rproj`. Whenever you are done with your R session, you can choose to save all the objects in your environment (we will discuss the concepts of 'objects' and 'environ-

ment' in a bit) by agreeing to 'Save' the prompt window. Then, when you wish to return to the project, all you have to do is to open the `your-folder-name.Rproj` and viola! All of your data objects, scripts should be right where you left it.

> *Try this for yourself!* Set up a new R project for a folder dedicated to this chapter and work through the contents, creating R scripts and data objects. Then, exit RStudio (remember to save the workspace image). Click on the .Rproj file to see if you are able to return to your previous workspace state.

2.1.2 Layout of RStudio

The default view of RStudio has 4 quadrants (the more useful and important tabs are mentioned below):

1. Upper left: R scripts
2. Lower left: Console
3. Upper right: Environment, History
4. Lower right: File, Plots, Packages, Help

Note that the positions of these quadrants may differ slightly depending on how you have set up your RStudio. You can re-size and re-organize the structure of these quadrants/panes depending on your own preferences (*Tools -> Global Options -> Pane Layout*). It may also be useful for you to adjust minor things like font size and type, color themes, etc. (*Tools -> Global Options -> Appearance*) to your liking, so that you feel happy and relaxed while programming.

2.1.3 R scripts vs. Console

You can enter commands (instructions to the computer) directly into the Console (where the '>' is) and hit Enter/Return.

Try entering the following into your Console:

```
10 + 20
```

```
## [1] 30
```

Notice that the answer of this math equation has been returned to you in the Console.

You can also open a blank file (*File -> New File -> R Script*), write the command in it, and then execute it with `Ctrl-Enter` (Win) or `Cmd-Return` (Mac), or clicking on "Run" near the top right corner with the cursor on the same line. The result should also appear in the Console.

2.1.4 Working with R scripts is recommended!

Where possible it is highly recommended to write your code in a R script before executing it, rather than to type it directly into the Console. This is especially true when your code becomes longer and more complex. There are also other benefits, such as:

- You can save your code in a .R file and re-open it in any R programming environment. This is important for reproducibility of your research.
- You can leave comments in the R script to remind yourself of your work. Comments are not interpreted by R and occur after the # symbol. Writing useful comments for your future self is both a skill and an art.
- You can break up a complex command into multiple lines for ease of writing and editing, and then run it all by highlighting the lines and pressing Ctrl-Enter.

```
10 + 20 # this is a comment
```

```
## [1] 30
```

```
# 20 * 20 (will not be run)
```

Advanced R users can consider using Rmarkdown or Quarto for literate programming and publishing.

2.2 Variables and Data Types

2.2.1 What are variables?

A key aspect of programming is the ability to store information into *variables*. In R sometimes these variables are also known as *objects*. Try executing the following:

```
x <- 7
my_height <- 163
```

Other than numeric data, you can also store text (character) and logical data too.

```
my_name <- "Cynthia"
SundayToday <- FALSE
```

Notice the use of `<-` above: this is an *assignment operator* to "assign" values to a name/label that you provide.

In addition, variable names should not contain any weird characters and should not begin with a number. I recommend using letters and underscores.

You should notice that while no output is printed to the console, there are new lines in the Environment tab of RStudio. Have a look and see if the items make sense to you. Basically, by running the above code, the value of `x` and `my_height` (as well as `my_name` and `SundayToday`) has been stored in the "memory" of RStudio.

Then you can begin to do interesting things with the variables:

```
# what is Cynthia's height in metres?
my_height/100
```

```
## [1] 1.63
```

```
my_height * x # a random math equation
```

```
## [1] 1141
```

```
SundayToday # this just prints the value of the variable
```

```
## [1] FALSE
```

2.2.2 Removing objects

If your workspace/environment gets too messy, you can remove variables by using the `rm()` function:

```
rm(x)
```

Notice that `x` is missing from the Environment tab.

To remove everything in the Environment, click on the Broom icon in the Environment tab or use the following command `rm(list=ls())`.

2.2.3 Saving objects

Although it doesn't make sense to save variables that contain only one value, it is useful to save variables that are more complex (e.g., data frames or network

objects), and to save the entire workspace so that you can come back to your project later.

Here is how you do it:

```
# single objects
save(my_height, file = 'CS_height.RData')

# entire workspace
save.image(file = 'everything.RData')
```

Notice that you need to specify a `file` argument which is the name of the file that you want to save the information to, and that the file extension is `.RData`.

You can also save all of the objects in the workspace by clicking on the Floppy Disk icon in the Environment tab (NOT the Scripts tab).

Recall from 'Setting up a Project' that you are automatically prompted to save all of the objects in the workspace before exiting the R session - this creates a hidden file called `.RData` where the information is saved to and is automatically loaded when you start a new session for the same project.

2.2.4 Loading objects

First, wipe the Environment with the Broom icon. The Environment should be empty. Then run the following:

```
load('everything.RData')
```

All your variables have magically returned!

2.2.5 What are vectors?

Vectors are a kind of a special variable that can store multiple values or data points, instead of just one.

```
a_vector <- c(1,9,6,8)

a_vector
```

```
## [1] 1 9 6 8
```

Notice the use of a special function `c()` to combine single values into the vector.

2.2.6 Cool things you can do with vectors

Here are some interesting things you can do with vectors - try running each command and see if you can figure out what it does by examining the output. The comments below give you several hints!

```
# extraction
a_vector[3]
```

```
## [1] 6
```

```
# removal
a_vector[-2]
```

```
## [1] 1 6 8
```

```
a_vector[-(1:3)] # this removes the first 3 elements
```

```
## [1] 8
```

```
# editing
a_vector[1] <- 0

a_vector
```

```
## [1] 0 9 6 8
```

```
# naming
names(a_vector) <- c('a', 'b', 'c', 'd')

a_vector
```

```
## a b c d
## 0 9 6 8
```

A key concept is the use of [square brackets] to subset the vector based on the order of the elements in the vector (numbered from left to right starting with 1).

2.3 Functions and Arguments

2.3.1 What are functions?

You can think of functions as special instructions that R knows of that you can use as shortcuts to do something. In fact, you have already been using a number of functions: `load(...)`, `rm(...)`, `save(...)`. The clue is that functions have a name that is typed out and has brackets following it.

2.3.2 What are arguments?

Again, you already have been using arguments in functions. For instance, `load('everything.RData')` has the argument inside the brackets which is the name of the file that you wish to load into your workspace.

The clue is that arguments are found inside the brackets of functions. You can think of arguments as additional instructions or parameters that will inform the function what it should do in specific ways.

2.3.3 Multiple arguments

Some functions can take on several arguments. When this is the case, it is good practice to explicitly name the argument.

Consider the following:

```
round(x = 131.485783, digits = 1)
```

```
## [1] 131.5
```

`round()` is a function that takes two arguments, the first argument (x) is the number that is being rounded, and the second argument (*digits*) indicating the number of decimal places.

If you don't specify the argument names, then R assumes that they follow the internal expected order:

```
round(131.485783, 1)
```

```
## [1] 131.5
```

```
# try the following and think about the output
round(131.485783, 2)
```

```
## [1] 131.49
```

```
round(1, 131.485783)
```

```
## [1] 1
```

```
round(131.485783)
```

```
## [1] 131
```

For the final command, if no argument is specified (in the second location) and there is a default value (here it is 0), then R will use the default. To find out about the internal ordering of arguments and default values for any function, you can search for the function name in the Help tab, or type `?function_name` into the Console. This gives you handy documentation about a specific function that you can quickly review.

2.4 File Paths and Working Directory

2.4.1 Navigation in computers

All of your digital data in computers are stored in folders and sub-folders. When you want open a file in a program, you need to navigate to the location where that specific file is stored.

Similarly, when programming in R, if you want to load datasets and/or previously saved workspaces into your workspace you need to first "point" RStudio to the correct location where these files are stored.

2.4.2 Helping RStudio find your stuff

First you need to know where it is currently looking at (i.e., your current *working directory*):

```
getwd()
```

```
## [1] "/home/cynthia/Documents/Research-Workspace/intro-to-igraph"
```

Make a new folder called "test_folder" in that location (using Finder or Files in your computer system). For learning purposes you can also move the `everything.RData` file there. Then try out the following:

```
load('everything.RData') # notice that this does not work

# let's have RStudio look in the correct folder
setwd('./test_folder') # note that in Windows you will need backslashes instead
# the period means "from the current location"

getwd() # this should be updated

load('everything.RData') # does it work now?
```

Now when you place a `.RData` file in `test_folder` you should be able to load it into your workspace.

An alternate (and possibly more intuitive) solution via the GUI is to do the either of the following: (1) *Session -> Set Working Directory -> Choose Working Directory* or (2) Files tab: navigate to desired location and click on the Gear icon. Then choose "Set Working Directory".

Note that when you are working in a dedicated R project workspace, the default working directory is basically the project folder where the .Rproj file lives in. If you have set up the workspace as recommended in "Setting up a Project", then the easier approach is to simply move any data files that you need for your project into that folder, rather than to keep changing the working directory to "find" your files. This is good practice in any case; better to consolidate all the files associated with a given project rather than have them scattered across multiple folders.

2.5 Packages

2.5.1 What are packages/libraries?

Packages contain a set of dedicated functions, sometimes they are often called "libraries". While many packages come pre-installed with R, many other packages (especially the ones we need for this class) need to be installed from CRAN before we can start to use them.

2.5.2 Installation vs. Loading

Remember that you only need to install a package once, but you need to load them each time you begin a new RStudio session so that you can use the special functions of that package.

```
# install ONCE
install.packages('igraph')

# load EACH TIME
library(igraph)
```

You can also install and load packages via the Packages tab in RStudio. Click on "Install" and search for the package you need. Then you can search for the packages installed on your system and load the ones you want with the check box. It is also convenient to Update packages here.

2.5.3 Detaching packages

If you need to remove an already loaded package in your workspace, you can uncheck the package name in Packages tab.

Or you can run the following in your console:

```
detach("package:igraph", unload=TRUE)
```

2.6 Special Datatypes

So far we know that we can store single values as objects or variables in RStudio, as well as a group of values in a vector. In this module you are also going to run into two special variable types: *dataframes* and *igraph network objects*. We will discuss network objects in the next chapter.

2.6.1 Dataframes

A data frame is a rectangular, 2-dimensional variable with multiple rows and columns. You should be familiar with this type of data from introductory statistics where you would load such data from Excel (or some other spreadsheet program) into SPSS or JASP for analysis.

2.6.2 Loading data into RStudio

First, the data should be saved as a `.csv` file. This is basically a "text" version of the Excel format (look at Save As... in your Excel program). I recommend this because special packages and functions are needed to load data from proprietary file formats. The csv format is much more flexible (it would also work with SPSS/JASP).

To see this in action, first create a .csv file called `numbers.csv` using your preferred spreadsheet program. Make two columns of random numbers and provide column headers. An example is shown below:

Column A	Column B
1	4
2	5
3	6

Save this file into your current working directory.

To load the file via the Console:

```
my_data <- read.csv('numbers.csv', header = TRUE)
```

Notice that we are using a function called `read.csv` to read in .csv files (there are other functions for other kinds of files but we will mostly work with .csv files in this book). This function contains two arguments: the name of the file in the first position, and a second argument which states that there are column headers in the file. If there are no column headers, then switch this to `FALSE` and R will automatically provide labels. Finally, we assign the contents of the .csv file to a data frame object called `my_data`, which now shows up in the Environment tab. You are of course free to change this to a different label that describes your data better.

There is an alternative approach using the RStudio GUI. To load the file via the GUI: Navigate to the file on Files tab, click on file and select "Import Dataset". A helpful window opens to preview how the data will look like, and gives you options to change the name of the data object, specify if there are column names/headers, etc.

> Common mistake: Selecting "View File" opens the data as a window in RStudio, but does NOT actually load the information into the workspace, which means that you are unable to manipulate the data in any way.

Chapter 3

Chapter 3: Introduction to network objects

3.1 What is a network?

A network consists of two sets of entities: A list of *nodes* and a list of *edges*. Nodes correspond to the individual elements whose pairwise relational structure is specified by edges or links. So, if node A and node B are related to each other based on some criteria set by the modeler (you!) then you would specify an edge between nodes A and B.

When networks are small (on the order of a handful of nodes), it would not too be difficult to manually specify the list of nodes and their connections. However, this quickly gets too onerous once the number of nodes increases because the number of possible edges increases exponentially. For instance, a network of 5 nodes could have a maximum of 10 edges, whereas a network of 50 nodes could have a maximum of 1,225 edges.

In practice, the networks that a social scientist are likely to be working with are going to be quite large, and the networks are derived from large datasets that are external to the R programming environment. In the next chapter we will explore how to get these datasets into RStudio and how to convert them into network. In this chapter, we focus on understanding the structure of the network object, a special datatype that `igraph` uses for network analysis.

3.2 Your first network

Network objects are a special variable that can only be analyzed with functions from the `igraph` library (that you previously installed). Let's check out a simple

example.

```
library(igraph) # remember that we need to load this

# create a network of 2 nodes that are connected
g <- graph_from_literal('cat'-'bat')
```

`graph_from_literal` is a special function that allows the user to quickly build small networks by explicitly stating the nodes and edges of the network. Here we are creating a network of 2 nodes, labeled 'cat' and 'bat', and these 2 nodes are connected to each other.

You should find a new object `g` in Environment. But the description of the object does not seem like a network. That is OK! R knows what it is.

When we print `g` in the Console, we find that it has a number of special properties. If you don't want to see all the edges being printed, you can type `summary(g)` instead.

```
g
```

```
## IGRAPH 908e438 UN-- 2 1 --
## + attr: name (v/c)
## + edge from 908e438 (vertex names):
## [1] cat--bat
```

```
summary(g)
```

```
## IGRAPH 908e438 UN-- 2 1 --
## + attr: name (v/c)
```

Let's try to make sense of the output in Console when we "print" the network object `g`.

This first line already packs a ton of information:

```
IGRAPH f977daf UN-- 2 1 --
```

1. The IGRAPH label is a good sign, it means that this object is need a network object that `igraph` understands. This is followed by a UUID generated to uniquely identify the graph.
2. The first letter in caps can either be **D** or **U**, which corresponds to *Directed* or *Undirected* edges, respectively. In this example we have a **U**, which means that the edges in this network are undirected.

3. The second letter in caps can either be **N** or **-**, which corresponds to nodes either having *Names* or are left unlabeled, respectively. In this example we have a **N**, which means that the nodes are labeled.
4. The third letter in caps: **W** or **-**, which corresponds to edges either having *Weights* or are unweighted, respectively. In this example we have a **-**, which means that the nodes are unweighted.
5. It is worth noting that it is possible to have a fourth letter, **B** or **-**, which corresponds to the network being a *Bipartite* graph or not, respectively. We do not cover bipartite networks in this book (for now), and so we should expect to see a **-**, as we do in this example.
6. The first number refers to the number of nodes.
7. The second number refers to the number of edges.

> A good tip for network analysis is to always check the summary of the network object to make sure that the network that you are analyzing is indeed what you think it is. For instance, if you wanted to analyze a network with weighted edges, but do not see the **W** in your network summary, this indicates that `igraph` is not interpreting the network as a weighted graph and suggests that something went wrong during the network construction/specification phase.

The second line reads:

```
+ attr: name (v/c)
```

This prints a list of the various node, edge, or network-level attributes associated with the network. You can think of "attributes" like "properties" of the nodes and edges. Here we have a single attribute, labeled 'name', and it is a node (**v**ertex) attribute of **c**haracter class. This can be inferred from '(v/c)' following 'name'.

It is possible for edges to have attributes, and for the network itself to have attributes. These will be labeled with e and g respectively in the character *before* the forward slash.

It is possible for the attribute to take on various data classes, such as numeric, character, logical. These will be labeld with n, c, and l respectively in the character *after* the forward slash.

The final lines read:

```
+ edge from f977daf (vertex names):
[1] cat--bat
```

This simply prints out all of the edges in the network, and this output will be truncated for very large network.

3.3 Your second network

To broaden our understanding, let's create a slightly different network and explore its properties.

```
# let's create a slightly different network
g_directed <- graph_from_literal('cat'-+'bat')

g_directed
```

```
## IGRAPH 3bc6019 DN-- 2 1 --
## + attr: name (v/c)
## + edge from 3bc6019 (vertex names):
## [1] cat->bat
```

First, are you able to tell what was different from the code used to create the second network?

It is a subtle change, but an additional + symbol is added to the dash. The function interprets + as the arrow head of the edge, and creates a *directed* edge pointing from 'cat' to 'bat'.

Now's examine the summary of `g_directed`.

```
IGRAPH ba5e381 DN-- 2 1 --
```

1. Notice that the first character is now **D**; this indicates that the network has directed edges.
2. The number of nodes and edges remains the same.

```
+ edge from ba5e381 (vertex names):
[1] cat->bat
```

Notice that the "directedness" of edges is also represented in the output, with arrowheads.

3.3.1 Exercise

Run the following code, and answer the questions below:

```
g_exercise <- graph_from_literal(A-+B, B-+C, C-+A)
E(g_exercise)$weight <- c(1,2,1)

summary(g_exercise)
```

```
## IGRAPH 3bfe7cc DNW- 3 3 --
## + attr: name (v/c), weight (e/n)
```

1. How many nodes and edges does this network have?
2. Are the edges in this network undirected or directed, and unweighted or weighted? Substantiate your answer.
3. How many attributes does this network have? Describe what these attributes are.

3.4 Node and edge attributes

In the previous exercise you might have noticed that I snuck in some new code: `E(g_exercise)$weight <- c(1,2,1)`. This section explains what this code does and how you can explore and manipulate the node and edge attributes of your network.

Try running the following and see what gets printed in the Console:

```
V(g_exercise)
```

```
## + 3/3 vertices, named, from 3bfe7cc:
## [1] A B C
```

```
E(g_exercise)
```

```
## + 3/3 edges from 3bfe7cc (vertex names):
## [1] A->B B->C C->A
```

As you can see, `V()` and `E()` are special functions in `igraph` which you can use to print a list of the nodes and edges associated with a particular network object.

Now let's see what we get with the following code:

```
V(g_exercise)$name
```

```
## [1] "A" "B" "C"
```

```
E(g_exercise)$weight
```

```
## [1] 1 2 1
```

The `$` is a special operator because it acts like a kind of a "selector" of a specific part of a larger thing. So you could interpret `V(g_exercise)$name` as the names of the nodes in the `g_exercise` network. Likewise, `E(g_exercise)$weight` refers to the weights of the edges in the `g_exercise` network.

Notice that the output is a *vector* (Chapter 2) of numbers and characters. This means we can also manipulate them using the special functions that we've learned about in Chapter 2.

```
V(g_exercise)$name <- c('Alice', 'Bob', 'Colin')

V(g_exercise)$name
```

```
## [1] "Alice" "Bob"   "Colin"
```

Based on this new knowledge, think back to this line of code `E(g_exercise)$weight <- c(1,2,1)` and see if you can figure out what is happening here. A neat thing that happened here is that `g_exercise` did not initially have an edge attribute called `weight` (you can verify this for yourself by removing `g_exercise` from your Environment and re-running only the line `g_exercise <- graph_from_literal(A-+B, B-+C, C-+A)` and checking the summary.) So you can actually specify your own node/edge attributes and add the corresponding information. Of course, this becomes an error-prone process once networks get even a little bit bigger. You will likely include these attributes at the initial stage of the network construction when importing the data used to create the network, so that these information are already "linked" to the nodes and edges of the network. We will explore this in the next chapter.

3.4.1 Exercise

Run the following in your Console and try to explain the output you observe.

```
V(g_exercise)$weight

E(g_exercise)$name
```

> *Answer: We get a NULL output because there is no node attribute called* `weight` *and no edge attribute called* `name`*.*

3.5 Graph attributes

Earlier in this chapter I mentioned that it is possible for the network itself to have an attribute. This is not particularly relevant for network analysis, but

for completeness the code below demonstrates how you can specify a graph attribute and how it would look like in the summary output. One possible use for this is to label a network object that you have painstakingly build with a reference so that when you share this network with others they can then cite and acknowledge your work.

```
g_exercise$citation <- "Please cite: Siew, C. S. Q. Sample network for Chapter 3."

summary(g_exercise) # pay attention to the attributes section!
```

```
## IGRAPH 3bfe7cc DNW- 3 3 --
## + attr: citation (g/c), name (v/c), weight (e/n)
```

```
g_exercise$citation
```

```
## [1] "Please cite: Siew, C. S. Q. Sample network for Chapter 3."
```

3.6 Useful functions for working with attributes

`igraph` offers some useful helper functions for working with attributes.

To get a list of attributes of each type

- `graph_attr_names(network)`
- `vertex_attr_names(network)`
- `edge_attr_names(network)`

where `network` = name of your network object.

To set an attribute of each type

- `set_graph_attr(network, name, value)`
- `set_vertex_attr(network, name, value)`
- `set_edge_attr(network, name, value)`

where `network` = name of your network object, `name` = name of attribute, `value` = the values associated for nodes/edges/graph for that attribute. This is an alternative approach and identical to `V(network)$name <- value` (for node attributes).

To remove an attribute of each type

- `delete_graph_attr(network, name)`
- `delete_vertex_attr(network, name)`
- `delete_edge_attr(network, name)`

where `network` = name of your network object, `name` = name of attribute.

Chapter 4

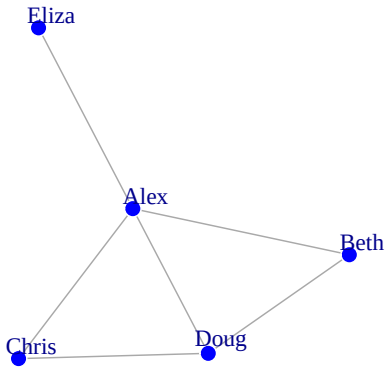
Chapter 4: From data to networks

4.1 How are networks represented?

It helps to have a very concrete idea of how your network is depicted in this programming environment. To illustrate this, let's create a simple network of 5 people. Each person is represented by a circle, called a node. If they are friends with each other, a connection or an edge is placed between them.

```
# create a simple example
g <- graph_from_literal(Alex-Beth, Chris-Doug, Eliza-Alex, Beth-Doug,
                        Alex-Chris, Doug-Alex)

# visualize the network
plot(g, vertex.color = 'blue', vertex.frame.color = 'white', vertex.size = 10,
     vertex.label.dist = 1.5)
```

Alex	Beth
Alex	Chris
Alex	Doug
Alex	Eliza
Beth	Doug
Chris	Doug



All of the information associated with the connectivity structure of these 5 people can be represented in two ways: in an *edge list* or an *adjacency matrix*. The following sections will demonstrate what data in an edge list and adjacency matrix looks like, and how to import the data into RStudio so that it corresponds to the network representation that you want.

4.2 Edge List

An edge list is exactly what it says it is. A list of edges (connections) in the network. Below you can see that each edge is represented on a separate row and the name of each node in the relationship is listed in separate columns.

Specifically, an edge list refers to a type of network data representation where each row represents a single edge in the network with at least 2 columns where the labels of connected nodes are provided. The number of rows in the edge list corresponds to the number of edges in the network. Additional columns can

	Alex	Beth	Chris	Doug	Eliza
Alex	0	1	1	1	1
Beth	1	0	0	1	0
Chris	1	0	0	1	0
Doug	1	1	1	0	0
Eliza	1	0	0	0	0

be specified that provide more information about the edges (e.g., type, weight). These become additional "edge attributes" of the network.

4.3 Adjacency Matrix

An adjacency matrix refers to a type of network data representation known as an adjacency matrix where the edge connectivity is represented in the matrix. The number of rows and columns correspond to the number of nodes in the network. A non-zero value in the [i,j] element of the adjacency matrix indicates the presence of a link between node i and node j.

So in this case, the matrix has 5 rows and 5 columns (a square 5 x 5 matrix). The rows and columns are labelled with the node names. The cell carries a value of 1 if two people are connected, 0 if not connected. In this example, if we look at the first row, we can easily tell that Alex is the social butterfly of the group; he's friends with everyone in the network!

> Eagle-eyed readers will realize that this matrix is symmetrical! This means that if you slice the matrix on its diagonal starting from the top left to the bottom right, the positions of filled cells are mirrored on the other side. This makes sense if we assume that the friendship of Alex and Beth goes both ways (Alex considers Beth a friend and Beth considers Alex a friend). In network science terms, we say that there is an undirected edge between Alex and Beth. We'll get into the nitty gritty about the different types of edges a network could have later on.

4.4 Importing your data and converting into a network

After reading the previous section, you should be able to identify whether your data set takes the form of an edge list or adjacency matrix, and if not, take the necessary steps to convert the data into one of these two formats. Depending whether it is an edge list or adjacency matrix, we will make use of different

functions to convert the connectivity information into an `igraph` network object so that we can start to analyze it further.

Another point to note is that you as the modeler will have to make decisions about the type of edges your network will have. Should the edges be undirected or directed, and/or unweighted or weighted? While there are no objectively right or wrong decisions, there are probably better or worse decisions that would depend mostly on your own research questions, interpretation of the data set, and the extant network science literature on similar topics.

A final point to note is that I have created sample .csv files based on the famous Zachary Karate Club network for demonstrating the code below. All supplementary data can be obtained from Github (https://github.com/csqsiew/csqsiew.github.io/tree/main/data), and are stored in a subdirectory called `/data` from which I am reading from. Based on chapter 2 (finding files) you should be able to make the corresponding modifications to the `read.csv()` parts of the code.

4.4.1 Importing edge lists

A. You want a network with *undirected, unweighted* edges.

```
karate_el <- read.csv('data/karate_el.csv', header = FALSE)

karate_uu <- graph_from_data_frame(karate_el, directed = FALSE)

summary(karate_uu)
```

```
## IGRAPH 372404e UN-- 34 78 --
## + attr: name (v/c)
```

Things to note:

1. `header = FALSE` in the first line - this is because there are no column headers in the `karate_el.csv` file! You can inspect and verify this for yourself.
2. `directed = FALSE` in the second line - this tells R that the edges are to be interpreted as undirected.
3. `graph_from_data_frame` is the function you would use for edge lists, which have been imported into RStudio as a data frame object.

B. You want a network with *directed, unweighted* edges.

```
karate_el <- read.csv('data/karate_el.csv', header = FALSE)

karate_du <- graph_from_data_frame(karate_el, directed = TRUE)

summary(karate_du)
```

```
## IGRAPH 6f42e05 DN-- 34 78 --
## + attr: name (v/c)
```

Things to note:

1. `directed = TRUE` in the second line - this tells R that the edges are to be interpreted as directed. The direction of the edges runs from the nodes in the first column of the edge list to the nodes in the second column of the edge list.

C. You want a network with *undirected, weighted* edges.

```
karate_el <- read.csv('data/karate_el_weights.csv', header = TRUE)

karate_uw <- graph_from_data_frame(karate_el, directed = FALSE)

summary(karate_uw)
```

```
## IGRAPH 24ba2bd UNW- 34 78 --
## + attr: name (v/c), weight (e/n)
```

Things to note:

1. A different file is used: `karate_el_weights.csv`. Open the file and see for yourself that there is an additional column called `weight`. We can consider this to be a representation of the strength of relationship between two karate club members. Additional columns in an edge list are interpreted as edge attributes - see if you can spot it in the summary output of `karate_uw`.
2. `header = TRUE` in the first line - this is because there *are* column headers in the `karate_el_weights.csv` file!

D. You want a network with *directed, weighted* edges.

```r
karate_el <- read.csv('data/karate_el_weights.csv', header = TRUE)

karate_dw <- graph_from_data_frame(karate_el, directed = TRUE)

summary(karate_dw)
```

```
## IGRAPH bbe89b6 DNW- 34 78 --
## + attr: name (v/c), weight (e/n)
```

4.4.2 Importing adjacency matrices

A. You want a network with *undirected, unweighted* edges.

```r
karate_am <- read.csv('data/karate_adj.csv', header = TRUE, row.names = 1)

karate_am <- as.matrix(karate_am) # converts dataframe into a matrix object

isSymmetric(karate_am) # sanity check that the matrix is symmetric
```

```
## [1] TRUE
```

```r
karate_uu <- graph_from_adjacency_matrix(karate_am, mode = 'undirected')

summary(karate_uu)
```

```
## IGRAPH b81ea0e UN-- 34 78 --
## + attr: name (v/c)
```

Things to note:

1. The arguments `header = TRUE` and `row.names = 1` in the `read.csv` function specify that there are column headers and row names in the first column of the dataset. Notice that the column and row labels (which correspond to the node names) are identical across both in `karate_adj.csv`.
2. We need to use the function `as.matrix` to convert the dataframe object into a matrix object, since the `graph_from_adjacency_matrix` expects a matrix object.
3. I included a sanity check - using `isSymmetric` to check if the matrix is symmetric down the diagonal. This is good to do because when creating an undirected network from an adjacency matrix, `graph_from_adjacency_matrix` expects a symmetric matrix, and returns a warning message if this is not true.

4. Notice that we need to use a different function, `graph_from_adjacency_matrix`, for converting the matrix into a network.

B. You want a network with *directed, unweighted* edges.

```
karate_am <- read.csv('data/karate_adj.csv', header = TRUE, row.names = 1)

karate_am <- as.matrix(karate_am) # converts dataframe into a matrix object

isSymmetric(karate_am) # sanity check that the matrix is symmetric
```

```
## [1] TRUE
```

```
karate_du <- graph_from_adjacency_matrix(karate_am, mode = 'directed')

summary(karate_du)
```

```
## IGRAPH c2d6f07 DN-- 34 156 --
## + attr: name (v/c)
```

Things to note:

1. `directed = TRUE` in the `graph_from_adjacency_matrix` function - this tells R that the edges are to be interpreted as directed. For each non-empty cell in the matrix, the direction of that edge would be **from** the node in the **row**, *to* the node in the *column*. See figure below for a depiction.
2. Because of this directedness interpretation, and given that the matrix is symmetric, this implies that the network's edges are "double counted". A directed connection from node A to B is counted separately from the directed connection from node B to A. You can see that the number of edges in the `summary` output of this network is twice that of the previous network.

C. You want a network with *undirected, weighted* edges.

```
karate_am <- read.csv('data/karate_adj_weights.csv', header = TRUE, row.names = 1)

karate_am <- as.matrix(karate_am) # converts dataframe into a matrix object

isSymmetric(karate_am) # sanity check that the matrix is symmetric
```

```
## [1] FALSE
```

```
karate_uw <- graph_from_adjacency_matrix(karate_am, mode = 'undirected', weighted = 'we
```

```
## Warning: The `adjmatrix` argument of `graph_from_adjacency_matrix()` must be symmetri
## i Use mode = "max" to achieve the original behavior.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated
```

```
summary(karate_uw)
```

```
## IGRAPH 94bf759 UNW- 34 100 --
## + attr: name (v/c), weight (e/n)
```

Things to note:

1. See if you can tell how `karate_adj_weights.csv` is different from `karate_adj.csv`. The numbers in the cells for the former file can take on values other than 0s and 1s. This is typically used to depict the weight of the connection between two nodes.
2. `weighted = 'weight'` informs `igraph` that the values in the cell (which range from 1 to 5) are to be assigned as values of an edge attribute called `'weight'`. This is how you get a network with weighted edges.

D. You want a network with *directed, weighted* edges

```
karate_am <- read.csv('data/karate_adj_weights.csv', header = TRUE, row.names = 1)

karate_am <- as.matrix(karate_am) # converts dataframe into a matrix object

isSymmetric(karate_am) # sanity check that the matrix is symmetric
```

```
## [1] FALSE
```

```
karate_dw <- graph_from_adjacency_matrix(karate_am, mode = 'directed', weighted = 'weig

summary(karate_dw)
```

```
## IGRAPH 8b4b333 DNW- 34 103 --
## + attr: name (v/c), weight (e/n)
```

4.5 Exercise

Janelle is interested in modeling the social network structure of her cats. In order to construct the network, she observed the interaction patterns of her cats for a week. Here are her observations:

1. Niko and Russell seem to get on quite well. Niko likes to eat Russell's food.
2. Annabelle and Sammy were adopted at the same time and tend to stick close to each other.
3. Janelle noticed that Jo has been napping next to Jack in the study room.
4. Sammy, Russell, and Groucho love playing with the ball of twine in the living room.
5. Groucho and Russell are elderly cats who seem to have a mutual respect for each other.
6. The only cat that Dottie is able to tolerate is Niko.

Your task is to:

(i) create the raw data file for this network (a .csv file)
(ii) convert the data into a network object in igraph
(iii) visualize the network using `plot(network_name)`
(iv) print the output of the network object
(v) save your network object as a .RData (and see if you are able to reload it into RStudio)

Chapter 5

Chapter 5: Manipulating network objects

In this chapter we explore how network objects can be "manipulated" based on their node or edge attributes. This can be especially useful if you wish to segment or subset your network so that you can analyze networks of different types. For instance, you may wish to explore how your network analysis results change depending on the threshold used to remove edges of varying weights (edge attribute). Another example could be that you are only interested in analyzing language networks where the nodes represent nouns, and not other parts of speech like verbs or adjectives (node attribute). We will also learn how to add these node and edge attributes to the network.

5.1 Set up

We will use the `karate` network as our example in this chapter.

```
library(igraph)
library(igraphdata)
library(tidyverse)

data('karate')

summary(karate)
```

```
## This graph was created by an old(er) igraph version.
##   Call upgrade_graph() on it to use with the current igraph version
##   For now we convert it on the fly...
```

```
## IGRAPH 4b458a1 UNW- 34 78 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

5.2 Adding node attributes

An important feature of psychological networks is that we are usually interested in more than just the underlying mathematics of the network, but rather we want to know how these network measures relate to behavior. This implies that the nodes in psychological networks are typically associated with rich psychological variables. For instance, nodes depicting individuals of different ages and genders, or nodes depicting words with different values on lexical-semantic variables like valence and arousal.

We need to add this node information as node attributes to the network representation.

In the code below, we output the list of nodes so that we can merge this data frame to the node attributes (can be done external to RStudio or internally within RStudio using data wrangling methods). This is because node attributes are additional variables that you have knowledge of or have collected in your study.

```
# export your node names, enter attributes manually outside of R
write.csv(data.frame(node = V(karate)$name), file = 'karate_nodes.csv', row.names = F)
```

In the `karate_node_added.csv` file, sample node attributes for the `karate` network have been added. The code below shows you how to read in this data and add this information as node attributes to the network using the `set_vertex_attribute` function.

```
# import your node attributes
node_info <- read.csv('data/karate_nodes_added.csv', header = T)

# very important to ensure that node order is identical
node_info <- node_info %>% arrange(factor(node, levels = V(karate)$name))

identical(V(karate)$name, node_info$node) # sanity check that the node name order is i
```

```
## [1] TRUE
```

```
# add the 'gender' attribute
karate <- set_vertex_attr(karate,
                          name = 'gender',
                          value = node_info$gender)
```

```
# add the 'belt' attribute
karate <- set_vertex_attr(karate,
                          name = 'belt',
                          value = node_info$belt)

# add the 'age' attribute
karate <- set_vertex_attr(karate,
                          name = 'age',
                          value = node_info$age)

summary(karate) # these vertex attributes should show up in the summary
```

```
## IGRAPH 4b458a1 UNW- 34 78 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label (v/c), colo
```

A key point is to ensure that the node ordering in your .csv file is identical to that of the network - this can be assured using the `arrange` function from `tidyverse`. This is to ensure accurate mapping from your data frame to the network.

When you run `summary(karate)` hopefully you will see that there are additional labels (gender, belt, age) included in the attributes section.

5.3 Adding edge attributes

Because there are usually more edges than nodes in the network, we usually do not manually insert edge attributes by hand (although it is possible!). In Chapter 4 recall that edge weights are usually included in the network at the point of network construction, as edge weights are in an additional column in the edge list that you are importing into RStudio.

Here we explore a simple case where you want to "tag" and color edges differently depending on whether the tie is between *same* or *different* genders.

```
# initialize all edges with the same label
E(karate)$edge_type <- 'same'

# re-assign those with mixed edges to a new label
E(karate)$edge_type[E(karate)[V(karate)[V(karate)$gender == 'male'] %--% V(karate)[V(karate)$gend

summary(karate) # there should be a new edge type
```

```
## IGRAPH 4b458a1 UNW- 34 78 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label (v/c), colo
```

```
E(karate)$edge_type
```

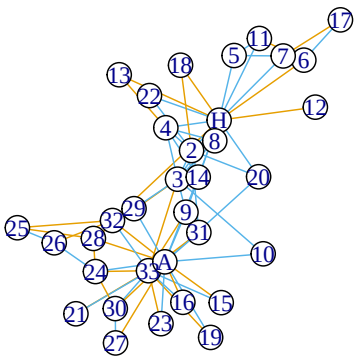
```
##  [1] "same"      "same"      "same"      "same"      "different" "same"      "same"
## [18] "same"      "same"      "same"      "different" "same"      "same"      "same"
## [35] "same"      "same"      "same"      "different" "different" "same"      "differ
## [52] "same"      "same"      "different" "same"      "different" "same"      "same"
## [69] "different" "different" "same"      "same"      "different" "different" "same"
```

```
# this tells you the ordering of the levels so you can tell that red = different and b
factor(E(karate)$edge_type) |> levels()
```

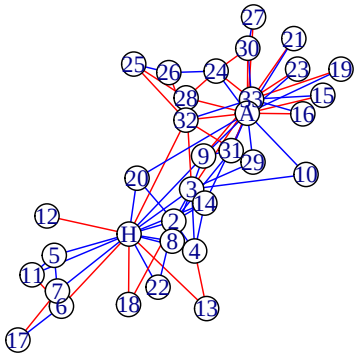
```
## [1] "different" "same"
```

We can then easily visualize the different edge types in the network.

```
# note the use of factor()
plot(karate, vertex.color = 'white', edge.color = factor(E(karate)$edge_type))
```



```
# specifying your own colors
plot(karate, vertex.color = 'white', edge.color = c('red', 'blue')[factor(E(karate)$edg
```



5.4 Subsetting the network

5.4.1 Subset by node attributes

The example below subsets the `karate` networks to only retain male individuals.

```
karate_male <- induced_subgraph(graph = karate,
                                vids = V(karate)$gender == 'female')

summary(karate_male)
```

```
## IGRAPH 8946ba7 UNW- 12 6 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label (v/c), colo
```

How would you change the code to retain female individuals?

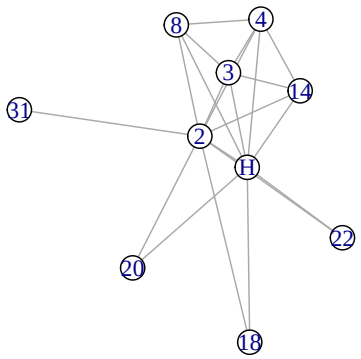
5.4.2 Subset by a target node

The example below shows you how to extract the immediate neighborhood structure of a particular node (the ego).

```
mrhi <- make_ego_graph(karate, order = 1, nodes = 'Actor 2')[[1]] # order = 1 (immedia

summary(mrhi)
```

```
## IGRAPH 30d3270 UNW- 10 21 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

```
plot(mrhi, vertex.color = 'white')
```

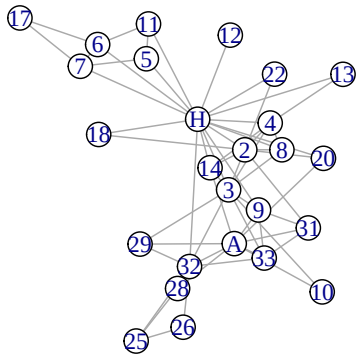


We can also extract the ego's 2-hop neighborhood - this includes their immediate connections, and the connections of the connections. This can be easily done by changing the `order` argument.

```
mrhi2 <- make_ego_graph(karate, order = 2, nodes = 'Mr Hi')[[1]] # order = 2 (immediat

summary(mrhi2)
```

```
## IGRAPH 4f0b4d4 UNW- 26 59 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

```
plot(mrhi2, vertex.color = 'white')
```

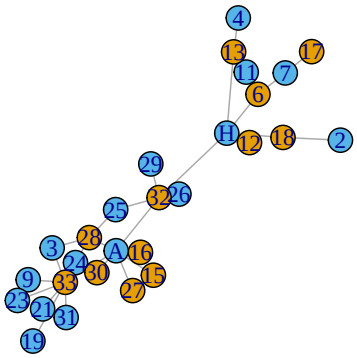


5.4.3 Subset by edge attributes

We can also subset a network by their edge attributes. Below shows an example where we only want to retain edges that between males and females (i.e., different genders).

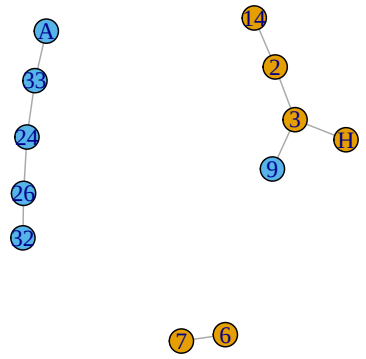
The `eids` argument looks a bit crazy, but what it is doing to subset from all of the edges in the original network, the set of edges that connect nodes that are female and male. `%--%` is a special `igraph` operator that looks for the edges that join two sets of specified vertices (here we are using the square operator to select nodes of various attributes). *Think about what aspect of this argument you would be modify in order to retain the edges between females only.*

```
# retain edges of the network where the connection is between a male and a female (i.e., mixed ed
mixed_network <- subgraph.edges(graph = karate,
                                eids = E(karate)[V(karate)[V(karate)$gender == 'female'] %--% V(k
                                )

# eids = E(network)$weight > 4

plot(mixed_network, vertex.color = factor(V(mixed_network)$gender))
```



If you have a weighted graph, it is also possible to subset the network based on edge weights. The code below retains edges that have a weight greater than 4. Play around with this argument and see if you can subset the network based on different weight thresholds. For instance, if you are interested in studying acquaintance relations, then you may decide to retain weak edges, but if you are interested in studying close relations, then you may decide to retain the strongest edges.

```
example_network <- subgraph.edges(graph = karate,
                                  eids = E(karate)[E(karate)$weight > 4]
                                  )

plot(example_network)
```



5.4.4 Subsetting by mutual edges

This section is most applicable for directed networks, as you may wish to subset a network based on whether edges are mutual (1–>2 AND 2–>1) or asymmetrical (1–>2 but there is no connection from 2 to 1).

To illustrate this, we first have to convert our `karate` network into a directed graph.

```
karate_d1 <- as.directed(karate, mode = 'random') # convert undirected edges into directed edges
karate_d2 <- as.directed(karate, mode = 'mutual') # convert undirected edges into mutual, directe

summary(karate_d1)
```

```
## IGRAPH c266607 DNW- 34 78 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label (v/c), colo
```

```
summary(karate_d2)
```

```
## IGRAPH e6b4530 DNW- 34 156 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label (v/c), colo
```

Notice from the summary output that `karate_d2` has twice the number of edges than `karate_d1`. This is because each undirected edge has been converted into two directed edges (mutual) connecting the node pair in both directions.

We can use the function `which_mutual` to detect mutual directed edges in the network and use this to subset the network accordingly. Notice that to select for the non-mutual edges, we can use the `!` or NOT operator.

```
# retain mutual edges
subgraph.edges(karate_d1, eids = E(karate_d1)[which_mutual(karate_d1)], delete.vertices
  summary() # empty graph
```

```
## IGRAPH e62e115 DNW- 0 0 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

```
subgraph.edges(karate_d2, eids = E(karate_d2)[which_mutual(karate_d2)], delete.vertices
  summary()
```

```
## IGRAPH 6e7e930 DNW- 34 156 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

```
# retain non-mutual edges
subgraph.edges(karate_d1, eids = E(karate_d1)[!which_mutual(karate_d1)], delete.vertic
  summary()
```

```
## IGRAPH 821ca94 DNW- 34 78 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

```
subgraph.edges(karate_d2, eids = E(karate_d2)[!which_mutual(karate_d2)], delete.vertic
  summary() # empty graph
```

```
## IGRAPH c8d8682 DNW- 0 0 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

Notice from the summary output that you get different results depending on the way that the directed graph is constructed. In these toy examples, the directed edges in the network were either ALL mutual edges or ALL non-mutual edges. If you have a network with varying proportions of mutual and non-mutual edges these functions can be useful for assessing the overall mutuality of the directed graph.

5.5 Useful functions

Finally, below are some useful functions for network manipulations that you can learn more about by reading the documentatoin.

```
as.directed() # convert your undirected edges to directed edges, see documentation

as.undirected() # convert your directed edges to undirected edges, see documentation

new_network <- remove.edge.attribute(network, 'weight') # to convert weighted to unweighted graph
```

Chapter 6

Chapter 6: Macro-level network measures

In this section, we will review network science measures that describe the overall or global structure of the entire network. You can think of these measures as providing a "bird's eye view" of your network, and they are useful for comparing different network representations.

To facilitate the demonstrations below, we will use the undirected, unweighted karate network that we have previously seen in Chapter 4. In the rest of the chapter we will compute various macro-level network measures for this network.

```
karate_el <- read.csv('data/karate_el.csv', header = FALSE)

karate_uu <- graph_from_data_frame(karate_el, directed = FALSE)

summary(karate_uu)
```

```
## IGRAPH 5f98de6 UN-- 34 78 --
## + attr: name (v/c)
```

Although it is possible to compute macro-level network measures for directed, weighted networks in certain instances (and this will be pointed out where appropriate), I choose to focus on the undirected, unweighted version for this chapter because macro-level measures are more complex to interpret and less intuitive when additional information like edge weights and direction are included. For instance, *network density* refers to the ratio of the number of (existing) edges and the number of possible edges among nodes in the network–it is not clear how this measure be adapted if we wanted to use weighted edges instead.

6.1 Average Degree

A common method of characterizing the global structure of networks is to simply compute **average degree**, which is the mean of the degree of all nodes in the network. A node's degree refers to the number of connections or edges that are incident to the node.

We first use `degree` to compute degree for each node (see Chapter 7: Degree) and then pipe the output to get the mean degree for the entire network.

```
degree(karate_uu) |> mean()
```

```
## [1] 4.588235
```

On average, each node in the karate network has 4.5 connections.

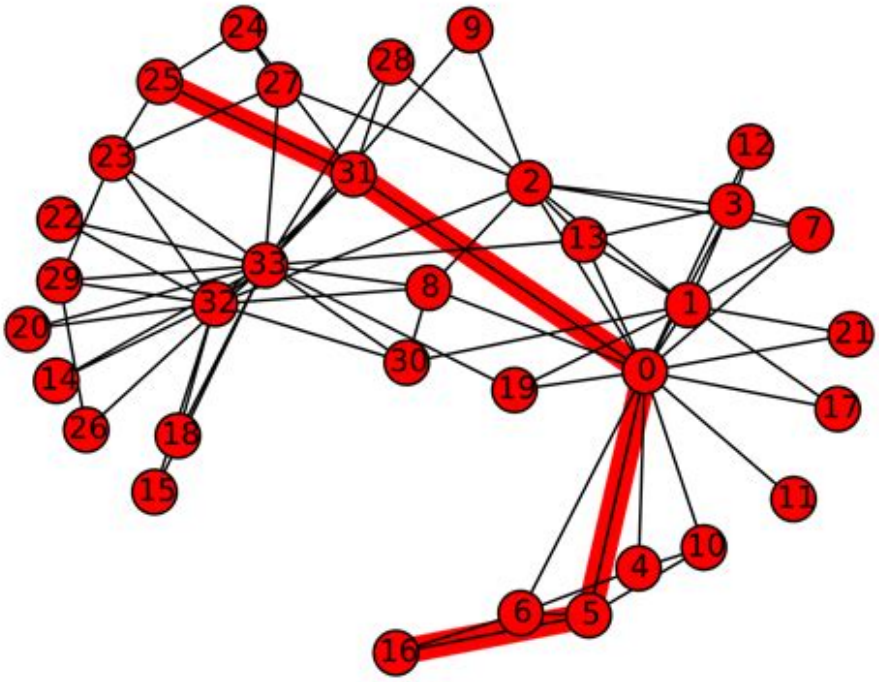
In Chapter 7 we will learn that it is possible to compute different variants of degree for directed graphs. In-degree refers to the number of edges incoming to the target node, out-degree refers to the number of edges outgoing from to the target node, and all-degree refers to all the edges (the sum of in- and out-degree). Try running the code below on a network with directed edges.

```
# mean in-degree
degree(karate_uu, mode = 'in') |> mean()

# mean out-degree
degree(karate_uu, mode = 'out') |> mean()

# mean all-degree
degree(karate_uu, mode = 'all') |> mean()
```

6.2 Average Shortest Path Length

Average shortest path length (ASPL) refers to the mean of the shortest possible path between all possible pairs of nodes in the network. (This loosely corresponds to the idea of "six degrees of separation" in social networks.)

The figure below depicts the shortest path between nodes 25 and 16 (path length = 4).



To compute the ASPL of the karate network:

```
mean_distance(graph = karate_uu)
```

```
## [1] 2.4082
```

On average, it takes 2.4 steps to connect any two randomly selected nodes in the karate network.

It is possible to compute ASPL based on shortest paths that consider the direction and weights of the edges.

```
mean_distance(karate_uu, weights = NA, directed = F) # unweighted, undirected
```

```
## [1] 2.4082
```

```
mean_distance(karate_uu, weights = NA, directed = T) # unweighted, directed
```

```
## [1] 2.4082
```

```
mean_distance(karate_uu, directed = F) # weighted, undirected
```

```
## [1] 2.4082
```

```
mean_distance(karate_uu, directed = T) # weighted, directed
```

```
## [1] 2.4082
```

Because `karate_uu` is undirected and unweighted, the result is unsurprisingly identical across all 4 version (try this with a directed, weighted network!). When a graph contains edge weights, then they are automatically considered unless you instruct otherwise using `weights = NA`. Use the `directed` argument to specify if the direction of the edges should be considered in the computation of short paths.

An important point to note is that edge weights are interpreted as *distances* rather than *strength of connection*: See Chapter 7 for more information about this.

6.3 Global Clustering Coefficient

Global clustering coefficient, or global C, refers to the number of closed triangles in the network relative to the number of possible triangles. It is a measure of overall level of local connectivity among nodes in the network.

A simple way of thinking about this concept is that it is measuring the probability that each pair of "friends" of a given node are also friends with each other.

To computer global clustering coefficient, we use the `transitivity` function as follows:

```
transitivity(karate_uu, type = 'global')
```

```
## [1] 0.2556818
```

We can interpret this value to mean that for each pair of nodes that are directly connected to a target node, the probability that these two nodes are also connected to each other is 25.6% (i.e., the probability of a completed "triangle", when it is possible, is 25.6%). This value is usually quite high in social networks, because people tend to introduce their friends to each other, resulting in *triadic closure*.

Contrast this with Chapter 7: Local clustering coefficient–notice that although we use the same function, the `type` argument specifies if we want to obtain the global clustering coefficient for the whole network (overall local connectivity), or local clustering coefficient for individual nodes (local connectivity centered on a single node).

6.4 Small World Index

The term "small world" has a specific meaning in network science as compared to the layperson's. A network is considered to have small world characteristics if (i) its ASPL is shorter than that of a randomly generated network with the same number of nodes and edges, and (ii) its global C is larger than that of a randomly generated network with the same number of nodes and edges. There are various ways to compute a value that quantifies the "small worldness" of a network, although we do not cover all of them here (see Neal, 2017, for a comparison of different methods). In this section we go through an approach to quantifying small-worldness based on Humphries and Gurney (2008).

In practice, researchers compute a measure known as the **small world index** which combines the global clustering coefficient and ASPL of the target network relative to its equally sized random graphs (with the same number of nodes and edges) to derive a proportional score.

Mathematically,

$$SWI = \frac{C_i / \overline{C}_{rand}}{L_i / \overline{L}_{rand}}$$

where C = global clustering coefficient, L = average shortest path length, i = network of interest, $rand$ = random network, $\overline{x}$ denotes mean of the random networks' measures.

In order to compute the Small World Index we need to first generate networks that form a baseline for comparison to the target network (see also Chapter 9: Network Models). Here, we generate 5 size-matched Erdos-Renyi (ER) networks, which is a type of random network model where edges are distributed randomly across a set of nodes. We specify that the number of nodes and edges of the ER network to be identical to the target network (`karate_uu`). Then we take the mean ASPL and C of the ER networks, which becomes the $\overline{L}_{rand}$ and $\overline{C}_{rand}$ in the equation above respectively.

```
## generate ER networks
set.seed(2)

er_graphs <- list()

for(i in 1:5) { # number of networks to generate
```

```
  er_graphs[[i]] <- sample_gnm(n = gorder(karate_uu), m = gsize(karate_uu), directed =
}

# compute the L and C for each ER network, take the mean
sapply(er_graphs, mean_distance) |> mean() # L_rand
```

```
## [1] 2.384314
```

```
sapply(er_graphs, transitivity, type = 'global') |> mean() # C_rand
```

```
## [1] 0.1188022
```

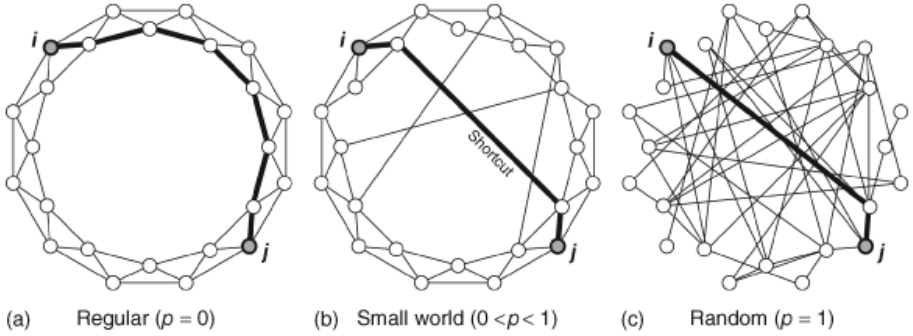
We can now plug in these values together with the actual C and L of the target network to compute SWI:

```
# SWI for karate_uu
(0.2556818/0.1188022)/(2.4082/2.384314)
```

```
## [1] 2.130817
```

Humphries and Gurney (2008) write that SWI larger than 1 indicates presence of a small world structure, since this implies that the target network has more local clustering than random expectation, while having relatively similar ASPLs. SWIs less than 1 or close to 1 indicates that there is no evidence to support the claim that the target network is a small world. Here, we have some evidence supporting the claim that our karate network has a small world structure. In practice, up to 1,000 random ER networks are generated to compute $\overline{L}_{rand}$ and $\overline{C}_{rand}$; although this can be a slow process depending on your network size. The number of random networks to generate can be adjusted in the for loop above.

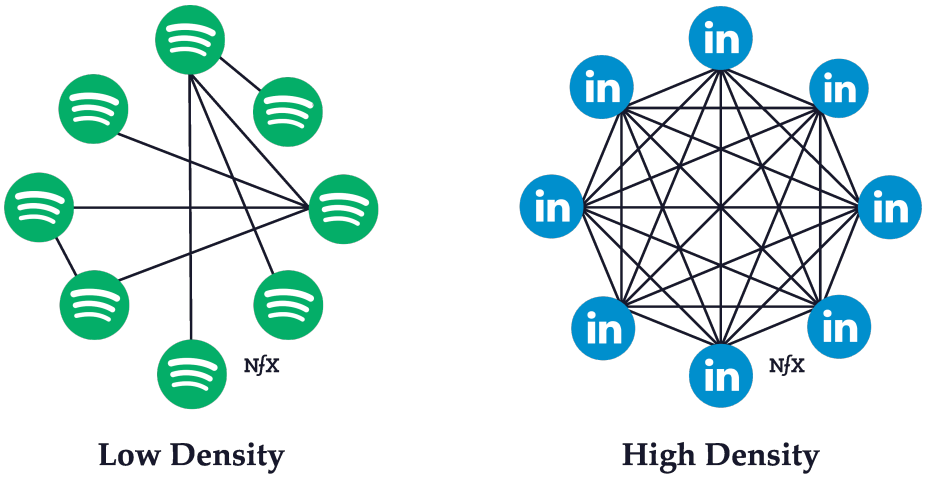
The main take home message is that a small world network has high levels of local clustering (nodes whose neighbors are also neighbors of each other), but also contains a number of shortcuts that drastically reduces the overall distances/path lengths between nodes. See below for an illustration of this idea.



6.5 Network Density

Network density refers to the ratio of the number of (existing) edges and the number of possible edges among nodes in the network.

Simple example of networks with lower and higher network densities.



To obtain network density of a network:

```
edge_density(karate_uu)
```

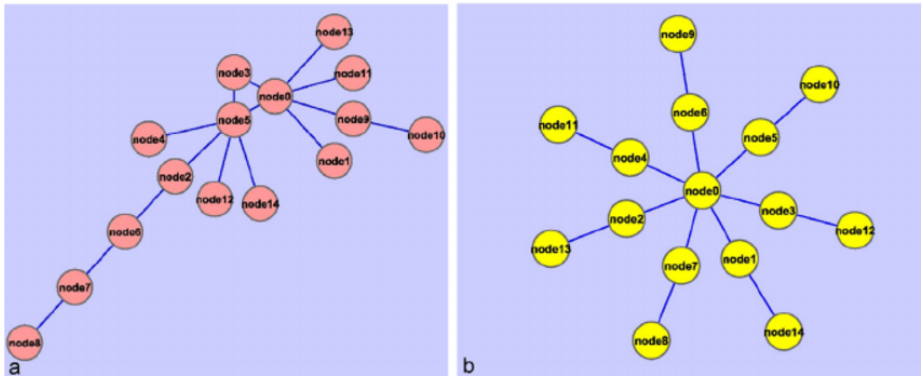
```
## [1] 0.1390374
```

This means that 13.9% of all possible edges in the network are actual edges.

6.6 Network Diameter

Network diameter refers to length of the longest shortest path between nodes in the network. Instead of getting the mean of all the shortest paths as you did in ASPL, what is the maximum length of those short paths?

Simple example of networks with higher and lower network diameters



To obtain network diameter of a network:

```
diameter(karate_uu, directed = F)
```

```
## [1] 5
```

```
# diameter(karate_uu, directed = T) # directed edges
```

This means that the length of the longest shortest path between two nodes in the network is 5.

6.7 Assortative Mixing

Assortative mixing is a measure of *homophily* in the network, based on some internal/external property of the nodes. Are nodes that are connected to each other tend to have similar properties?

6.7.1 Assortative Mixing by Degree (continuous attributes)

A common extension of assortative mixing is **assortative mixing by degree**, which measures the correlation of degrees between neighboring nodes. Are nodes that are connected to each other tend to have similar degrees?

A positive value indicates that nodes that are many (few) neighbors tend to be connected to nodes that also have many (few) neighbors. A negative value indicates that nodes that are many (few) neighbors tend to be connected to nodes that have few (many) neighbors. Values very close to zero indicates no correlation.

```
assortativity_degree(karate_uu, directed = F)
```

```
## [1] -0.4756131
```

```
# assortativity_degree(karate_uu, directed = T) # for directed edges
```

There is *dissortativity mixing by degree* in the karate network–nodes with higher degree have a higher tendency to be connected to nodes with lower degree.

It is possible to add your own node attributes to the network (see Chapter 5) and compute its assortativity. The script below demonstrates this with a randomly created set of numbers for the nodes (i.e., a continuous variable ranging from 1 to 5, randomly selected with replacement for each of the 34 nodes in the network). Because these numbers are randomly generated, we expect assortativity to be close to 0 here.

```
# create a random vector of numbers for demonstration
set.seed(3)
V(karate_uu)$random1 <- sample(1:5, gorder(karate_uu), replace = T)

# compute the assortativity of this node attribute
assortativity(karate_uu, values = V(karate_uu)$random1, directed = F)
```

```
## [1] -0.06819853
```

6.7.2 Assortative Mixing by Node Attributes (categorical attributes)

It is also possible to compute assortativity for categorical labels assigned to nodes. For instance, in a social network of male and female actors, is there gender homophily such that nodes tend to be connected to other nodes that share the same label?

A positive value indicates that nodes with the same labels tend to be connected to each other. A negative value indicates that nodes tend to be connected to nodes with a different label from itself. Values very close to zero indicates no correlation.

It is possible to add your own node attributes to the network (see Chapter 5) and compute its assortativity. The script below demonstrates this with a randomly created set of labels for the nodes (i.e., a categorical variable). Because these labels are randomly generated, we expect assortativity to be close to 0 here.

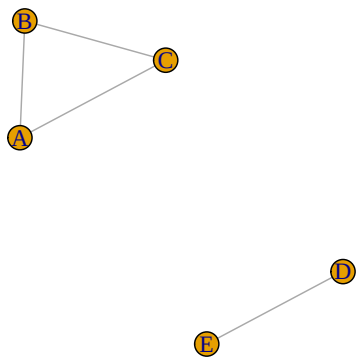
```
# create a random vector of labels for demonstration
set.seed(3)
V(karate_uu)$random2 <- sample(c('A', 'B', 'C'), gorder(karate_uu), replace = T)

# compute the assortativity of this node attribute
assortativity_nominal(karate_uu, types = factor(V(karate_uu)$random2), directed = F)
```

```
## [1] -0.0850826
```

6.8 Network Components

Thus far, we have been assuming that the networks that we have worked with form a single component. In other words, all nodes are part of the same component, and there are no separate components. But, as shown in the example network below, it is certainly possible for nodes in a network to fragment into distinct components.

```
# create a sample network with two distinct components
g <- graph_from_literal(A-B, B-C, A-C, D-E)
plot(g)
```



When working with networks derived from behavioral data, it can be common to observe that the network consists of distinct network components. This has implications for the computing and interpretation of the macro-level network measures, because several of them rely on the computation of shortest path between node pairs. For instance, ASPL computes the shortest paths for all possible node pairs and takes the mean–but what happens if there is no path between nodes A and D, for example?

Node pairs that do not have a path are ignored in the computation. This is the default behavior in `igraph`. Hence, this value reflects the mean of all "computable" short paths (i.e., when node pairs are found in the same component) and this is computed across all network components in the network.

```
mean_distance(g, unconnected = TRUE) # ignore infinite paths, default
```

```
## [1] 1
```

```
mean_distance(g, unconnected = FALSE) # consider infinite paths
```

```
## [1] Inf
```

As you can see, the second output is `Inf`, reflecting the presence of infinite paths in the network. From the `igraph` documentation: "If TRUE, only the lengths of the existing paths are considered and averaged; if FALSE, the length

of the missing paths are considered as having infinite length, making the mean distance infinite as well."

Hence, it is good practice to first check if your network consists of a single component, or multiple components. We can use the following script to find out: how many components, the size of each components (number of nodes), and obtain a list of nodes and their component membership. Basically, `components` is a special function that stores these pieces of information as its output, which you can then retrieve using the `$`: `csize` for size of component, `no` for number of components, `membership` for the component membership for each node.

```
g_comp <- components(g)

g_comp$csize # size of each component
```

```
## [1] 3 2
```

```
g_comp$no # number of components
```

```
## [1] 2
```

```
data.frame(node = V(g)$name, membership = g_comp$membership)
```

```
##   node membership
## A    A          1
## B    B          1
## C    C          1
## D    D          2
## E    E          2
```

```
# you should know how to export the results into a .csv file!
```

Typically, real-world networks have a giant or largest connected component (LCC) which contains the majority of nodes in the network, and researchers prefer to only compute network-level metrics on the LCC. This component is extracted for further analysis, removing the smaller connected components or hermits (isolates in the network where the node has no neighbors).

The code below shows you how to extract the largest connected component of the sample network:

```
g_lcc <- induced_subgraph(graph = g,
                          vids = components(g)$membership == which.max(components(g)$cs

summary(g_lcc)
```

```
## IGRAPH 3487066 UN-- 3 3 --
## + attr: name (v/c)
```

It is also possible to extract other components of the network. The code below retains nodes that do not belong to the largest connected component of the network.

```
g_2 <- induced_subgraph(graph = g,
                        vids = components(g)$membership != which.max(components(g)$csize))

summary(g_2)
```

```
## IGRAPH e23b4aa UN-- 2 1 --
## + attr: name (v/c)
```

6.9 Exercise

Your task is to:

1. Load the `UKfaculty` network from the `igraphdata` library using the following code:

```
library(igraphdata) # have you downloaded the package?

data("UKfaculty") # you should see a UKfaculty object appear in your Environment

# I've included this code to make it an undirected, unweighted network for this exercise
# for more information about network manipulation, see Chapter 5
UKfaculty <- UKfaculty |> delete_edge_attr("weight") |> as.undirected(mode = 'mutual')
```

{TODO: Briefly describe the network.}

2. Answer the following questions:

- How many nodes and edges does the network have?
- How many network components does this network have? If it has more than one component, only retain the largest connected component before proceeding to the next part.
- Compute the following macro-level network measures for this network:
  - average degree

- average shortest path length
- global clustering coefficient
- network density
- network diameter
- assortativity (by degree and by group) {footnote}
- small world index

Footnote: `UKfaculty` contains a `Group` node attribute which denotes the department that each faculty member belonged to.

Chapter 7

Chapter 7: Micro-level network measures

Micro-level network measures provide you with information about specific nodes in the network. These are generally known as centrality measures in the network science literature. Centrality is the network scientist's way of quantifying the relative "importance" of a given node relative to other nodes in the network. There are *many* different definitions of what counts as "central", as you will see in the following subsections, and also from this Periodic Table of Centrality Indices. There is no single "correct" or "best" metric - which metrics are most useful to you will depend on the nature of the system that you are modeling as well as the network behavior that you interested in.

To facilitate the demonstrations below, we will use the directed, weighted karate network that we have previously seen in Chapter 4. In the rest of the chapter we will compute various micro-level network measures for the nodes in this network.

```
karate_el <- read.csv('data/karate_el_weights.csv', header = TRUE)

karate_dw <- graph_from_data_frame(karate_el, directed = TRUE)

summary(karate_dw)
```

```
## IGRAPH 66f644c DNW- 34 78 --
## + attr: name (v/c), weight (e/n)
```

7.1 Degree

The **degree** of node i, frequently denoted as k_i in the network science literature, refers to the number of edges or links directly connected to node i.

To get the degree of all nodes in the network, try running the following code:

```
degree(graph = karate_dw)
```

```
##    Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 9 Actor 10 Act
##       16        9       10        6        3        4        4        5        2
## Actor 32 Actor 33  Actor 8 Actor 11 Actor 12 Actor 13 Actor 18 Actor 22 Actor 17
##        6       12        4        3        1        2        2        2        2
```

If you only want the degree for a specific node:

```
degree(graph = karate_dw, v = 'Mr Hi')
```

```
## Mr Hi
##    16
```

```
degree(graph = karate_dw, v = c('Mr Hi', 'Actor 5')) # for more than 1 node
```

```
##   Mr Hi Actor 5
##      16       3
```

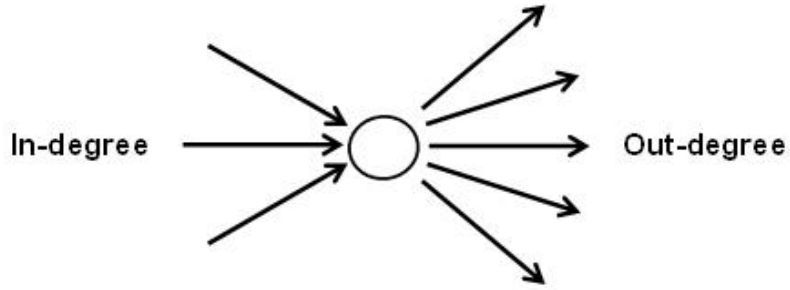
It would be better to "save" the outputs as an object (especially if your network is very large!) so that you can manipulate or export it later.

```
karate_degree <- degree(karate_dw)
```

Notice that you have a named vector called `karate_degree` in your Environment.

7.1.1 In-, out-, all-degree

If your network has directed edges, **in-degree** refers to the number of edges that are going towards the target node, whereas **out-degree** refers to the number of edges that are going away from the target node. **all-degree** is the sum of in-degree and out-degree, and is the same as *degree*.



To get the in- or out-degree of a node, we need to include the `mode = 'in'` or `mode = 'out'` argument accordingly.

```
degree(graph = karate_dw, v = 'Actor 5', mode = 'in') # in-degree = incoming edges
```

```
## Actor 5
##       1
```

```
degree(graph = karate_dw, v = 'Actor 5', mode = 'out') # out-degree = outgoing edges
```

```
## Actor 5
##       2
```

Notice that Actor 5 has an in-degree of 1 and out-degree of 2, and in the previous section we saw that Actor 5 had a degree of 3. If no arguments are specified, `igraph` will compute all-degree, which is the sum of a node's in- and out-degree, by default for directed graphs. Keep this in mind!

```
degree(graph = karate_dw, v = 'Actor 5') # the default is all-degree
```

```
## Actor 5
##       3
```

```
degree(graph = karate_dw, v = 'Actor 5', mode = 'all')
```

```
## Actor 5
##       3
```

```
degree(graph = karate_dw, v = 'Actor 5', mode = 'total') # all is the same as total
```

```
## Actor 5
##       3
```

7.2 Strength

The **strength** of node i refers to the sum of its adjacent edge weights. This measure is only applicable to weighted networks.

To get the strength of all nodes in the network, try running the following code:

```
strength(graph = karate_dw)
```

```
##    Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 9 Actor 10 Act
##       45       23       33       16        4       12       11       16        5
## Actor 32 Actor 33  Actor 8 Actor 11 Actor 12 Actor 13 Actor 18 Actor 22 Actor 17
##       17       31        8        5        1        7        7        3        5
```

If you only want the strength for a specific node(s):

```
strength(graph = karate_dw, v = 'Mr Hi')
```

```
## Mr Hi
##    45
```

```
strength(graph = karate_dw, v = c('Mr Hi', 'Actor 5')) # for more than 1 node
```

```
##   Mr Hi Actor 5
##      45       4
```

7.2.1 In-, out-, all-strength

If your network has directed edges, we can have variations to the original strength measure just as we saw for degree. **In-strength** refers to the sum of the edge weights that are going towards the target node, whereas **out-strength** refers to the sum of the edge weights that are going away from the target node. **all-strength** is the sum of in-strength and out-strength, and is the same as *strength*.

To get the in- or out-strength of a node, we need to include the `mode = 'in'` or `mode = 'out'` argument accordingly.

```
strength(graph = karate_dw, v = 'Actor 5', mode = 'in') # in-degree = incoming edges
```

```
## Actor 5
##       2
```

```
strength(graph = karate_dw, v = 'Actor 5', mode = 'out') # out-degree = outgoing edges
```

```
## Actor 5
##       2
```

It is worth noting here that the in-strength and out-strength of Actor 5 is the same, but recall from the degree section that the in-degree and out-degree of Actor 5 differed. This is a key point to remember, is that strength does *not* provide you with information about the *number* of edges the node has - it is possible for a node to have high strength due to a few highly weighted edges or several weakly weighted edges.

Once again, if no arguments are specified, `igraph` will compute all-strength, which is the sum of a node's in- and out-strength, by default for directed graphs.

```
strength(graph = karate_dw, v = 'Actor 5') # the default is all-strength
```

```
## Actor 5
##       4
```

```
strength(graph = karate_dw, v = 'Actor 5', mode = 'all')
```

```
## Actor 5
##       4
```

```
strength(graph = karate_dw, v = 'Actor 5', mode = 'total') # all is the same as total
```

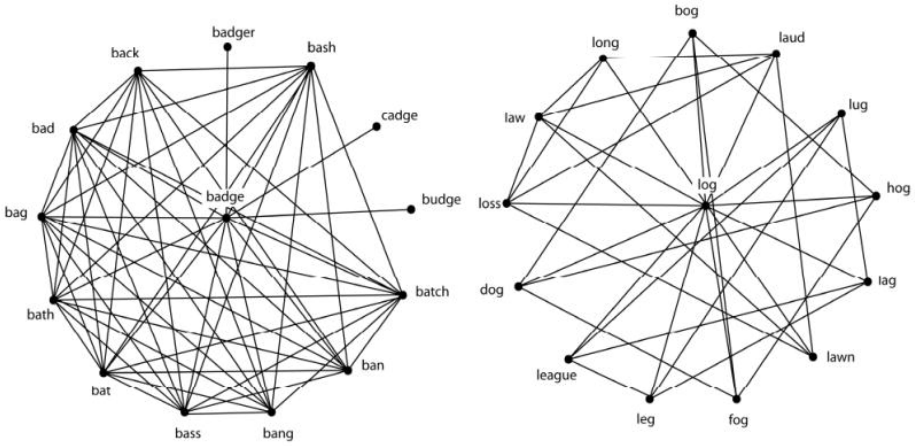
```
## Actor 5
##       4
```

7.3 Local clustering coefficient

The **local clustering coefficient**, C, of node i measures the ratio of the actual number of edges existing among nodes directly connected to the target node i to the number of all possible edges that could exist among these nodes.

C ranges from 0 to 1. When $C = 0$, none of the neighbors of the target node are connected to each other. When $C = 1$, each neighbor of the target node is connected to all the other neighbors of the target word.

You can think of local clustering coefficient as providing a measure of the *level of interconnectivity among the local neighborhood of the target node*. If you look at the figure below, you can easily see that it is possible for nodes to have the same degree, but different local clustering coefficients.



To get the local clustering coefficient of all nodes in the network, try running the following code:

```
transitivity(karate_dw, type = 'local', isolates = 'zero')
```

```
##     Mr Hi   Actor 2   Actor 3   Actor 4   Actor 5   Actor 6   Actor 7   Actor 9  Act
## 0.1500000 0.3333333 0.2444444 0.6666667 0.6666667 0.5000000 0.5000000 0.5000000 0.0
##  Actor 29  Actor 30  Actor 31  Actor 32  Actor 33   Actor 8  Actor 11  Actor 12  Act
## 0.3333333 0.6666667 0.5000000 0.2000000 0.1969697 1.0000000 0.6666667 0.0000000 1.0
```

A couple of things to note:

1. It is important to specify `type = local` for local clustering coefficients (one value for each node), as compared to the global clustering coefficient of the entire graph (this is a macro-level measure that we learned about in Chapter 6).

2. For nodes that have a degree of 0 or 1, it would not be possible to compute a local clustering coefficient value for those nodes. Be default `igraph` will return a `NaN` value (see Actor 12's value below). By specifying `isolates = 'zero'` these `NaN` values will be reported as zeros instead.

```
transitivity(karate_dw, type = 'local')
```

```
##     Mr Hi   Actor 2   Actor 3   Actor 4   Actor 5   Actor 6   Actor 7   Actor 9  Actor 10  Act
## 0.1500000 0.3333333 0.2444444 0.6666667 0.6666667 0.5000000 0.5000000 0.5000000 0.0000000 0.60
##  Actor 29  Actor 30  Actor 31  Actor 32  Actor 33   Actor 8  Actor 11  Actor 12  Actor 13  Act
## 0.3333333 0.6666667 0.5000000 0.2000000 0.1969697 1.0000000 0.6666667       NaN 1.0000000 1.00
```

7.3.1 Weighted local clustering coefficient

If you have a weighted network, you can compute *weighted* local clustering coefficients using Barrat et al.'s (2004) generalization of transitivity to weighted networks by specifying `type = 'weighted'`. If your network is unweighted, this generalization will simply return unweighted C values.

```
transitivity(karate_dw, type = 'weighted')
```

```
##      Mr Hi    Actor 2    Actor 3    Actor 4    Actor 5    Actor 6    Actor 7    Actor 9   Acto
## 0.13629630 0.31521739 0.25925926 0.63750000 0.75000000 0.58333333 0.54545455 0.48437500 0.0000
##   Actor 27   Actor 28   Actor 29   Actor 30   Actor 31   Actor 32   Actor 33    Actor 8   Acto
## 1.00000000 0.15384615 0.27777778 0.75000000 0.60606061 0.16470588 0.15542522 1.00000000 0.8000
```

7.4 Closeness centrality

Closeness centrality of node i is the inverse of the average of the length of the shortest path between node i and all other nodes in the network. If a node has high closeness centrality, it means that on average, it takes few steps to travel from that node to all other nodes in the network. If a node has low closeness centrality, it means that on average, it takes more steps to travel from that node to all other nodes in the network.

Closeness centrality is commonly viewed as an indicator of the accessibility of a given node in the network from all other locations in the network.

To get the closeness centrality of all nodes in the network, try running the following code:

```
closeness(karate_dw, normalized = T) # weights are automatically considered
```

```
##     Mr Hi   Actor 2   Actor 3   Actor 4   Actor 5   Actor 6   Actor 7   Actor 9  Actor 10  Act
## 0.2771084 0.2051282 0.2666667 0.2666667 0.5000000 0.4285714 0.3333333 0.4285714 0.5000000 1.00
##  Actor 29  Actor 30  Actor 31  Actor 32  Actor 33   Actor 8  Actor 11  Actor 12  Actor 13  Act
## 0.3333333 0.4000000 0.2857143 0.5000000 1.0000000       NaN       NaN       NaN       NaN
```

It is typical to include `normalized = T` so that the values are normalized with respect to the size of the network.

Note that closeness centrality can only be meaningfully computed for connected graphs (so that a path exists between any pair of nodes; see Chapter 6: Network Components). If there are distinct network components in the network, this means that for some sets of node pairs, the path between them does not exist and closeness cannot be computed. Usually, network scientists focus their analysis on the largest connected component of the network and ignore the smaller connected components (which are typically viewed as outliers). Chapter 6 provides you with the code needed to extract the largest connected component from the initial network.

You can specify the `mode` and `weights` arguments accordingly if you have directed/weighted networks to get the corresponding versions of closeness centrality computed.

7.4.1 Weighted closeness centrality

If your network is weighted, weights are automatically included in the computation of closeness centrality. If you prefer that edge weights are *not* considered, include `weights = NA` in the argument. Notice that some of the values have changed depending on whether weights are considered or not.

If your network is unweighted, then weights are not considered by default.

```
closeness(karate_dw, normalized = T) # weights are automatically considered
```

```
##     Mr Hi   Actor 2   Actor 3   Actor 4   Actor 5   Actor 6   Actor 7   Actor 9  Act
## 0.2771084 0.2051282 0.2666667 0.2666667 0.5000000 0.4285714 0.3333333 0.4285714 0.5(
##  Actor 29  Actor 30  Actor 31  Actor 32  Actor 33   Actor 8  Actor 11  Actor 12  Act
## 0.3333333 0.4000000 0.2857143 0.5000000 1.0000000       NaN       NaN       NaN
```

```
closeness(karate_dw, normalized = T, weights = NA) # weights are ignored
```

```
##     Mr Hi   Actor 2   Actor 3   Actor 4   Actor 5   Actor 6   Actor 7   Actor 9  Act
## 0.7666667 0.6400000 0.7500000 0.8000000 0.7500000 1.0000000 1.0000000 1.0000000 1.00
##  Actor 29  Actor 30  Actor 31  Actor 32  Actor 33   Actor 8  Actor 11  Actor 12  Act
## 0.7500000 1.0000000 1.0000000 1.0000000 1.0000000       NaN       NaN       NaN
```

7.4.2 Interpretation of edge weights

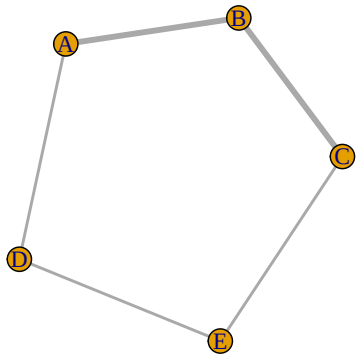
Caution is needed as the interpretation of edge weights in this context is to treat them as *distances* rather than as connection strengths. In other words,

when computing closeness centralities, `igraph` treats higher edge weights as longer distances. As reproduced from the `igraph` documentation: "If the graph has a weight edge attribute, then this is used by default. Weights are used for calculating weighted shortest paths, so they are interpreted as distances.").

Notice that this is different from how we would normally want to interpret edge weights, usually as indicator of the strength of the relation between two nodes. If we want the weights to be interpreted in the way that we intend (as relational strength) we need to include the following to "trick" igraph: `weights = 1/E(network)$weight`. This inverts the relation so that now bigger numbers for edge weight corresponds to shorter distances (and hence shorter paths).

To demonstrate this point consider the simple example below where we wish to compute the shortest path between two nodes in unweighted and weighted networks. Notice that the result changes depending on the weights assigned to the edges–whereby smaller weights are considered to provide "shorter paths" between pairs of nodes. This corresponds to the "distance" interpretation of edge weights. Because closeness centrality relies on the same `shortest_paths` function under the hood, this is something that we need to keep in mind when deciding to computed weighted closeness centrality.

```
# unweighted graph
g <- graph_from_literal(A-B,B-C,A-D,D-E,E-C)

# weighted graph
g_w <- g
E(g_w)$weight <- c(2,1,2,1,1)

plot(g_w, edge.width = E(g_w)$weight*2)
```




```
# what is the shortest path between nodes A and C?
shortest_paths(g_w, from = "A", to = "C")$vpath # the shortest path is not A-B-C!
```

```
## [[1]]
## + 4/5 vertices, named, from 066e36b:
## [1] A D E C
```

```
shortest_paths(g, from = "A", to = "C")$vpath # the shortest path is A-B-C!
```

```
## [[1]]
## + 3/5 vertices, named, from 066e36b:
## [1] A B C
```

```
# the reason is between bigger weights correspond to larger distances.
# and hence longer paths

# the shortest path is A-B-C after we invert the edge weights
E(g_w)$weight <- 1/c(2,1,2,1,1)
shortest_paths(g_w, from = "A", to = "C")$vpath
```

```
## [[1]]
## + 3/5 vertices, named, from 066e36b:
## [1] A B C
```

To reiterate, if you have a weighted graph and you would like to compute weighted closeness centrality for your nodes AND your edge weights are already interpreted as *distances* this is the pseudo R code that you need:

```
closeness(network, normalized = T, weights = E(network)$weight)
```

If your edge weights are interpreted as *strengths*, then you need to mathematically "flip" or invert the edge weights.

```
# take the inverse
closeness(network, normalized = T, weights = 1/E(network)$weight)

# if your weights are Likert ratings, this code flips the scale around
closeness(network, normalized = T,
          weights = abs(E(network)$weight - (max(E(network)$weight) + 1)))
```

7.4.3 Directed closeness centrality

As shown in the example below for *John A*, different closeness centrality values are returned depending on whether closeness centrality should be computed based on both incoming and outgoing edges (`mode = 'all'`), incoming edges only (`mode = 'in'`), or outgoing edges only (`mode = 'out'`). Because *John A* has an out-degree of 0 (see if you know how to verify this for yourself!), it has a closeness centrality of NaN when `mode = 'out'` is specified.

```
closeness(karate_dw, normalized = T, weights = NA, mode = 'all', vids = 'John A')
```

```
## John A
##   0.55
```

```
closeness(karate_dw, normalized = T, weights = NA, mode = 'in', vids = 'John A')
```

```
##    John A
## 0.7931034
```

```
closeness(karate_dw, normalized = T, weights = NA, mode = 'out', vids = 'John A')
```

```
## John A
##    NaN
```

7.5 Betweenness centrality

Betweenness centrality is a measure of the degree to which nodes stand in between each other. A node with a high betweenness centrality is a node that is frequently found in the short paths of other pairs of nodes in the network. In contrast, a node with a low betweenness centrality is a node that is not usually found in the short paths of node pairs. Betweeenness can be viewed as an indicator of whether a node represents a "bottleneck" in the system.

The same considerations (i.e., about connected graphs, additional arguments for weighted and directed graphs, normalization, interpretation of weights as distances) from the closeness centrality section applies to betweenness centrality as well, because betweenness centrality also relies on the `shortest_paths` function for its computation.

To get the betweenness centrality of all nodes in the network, try running the following code:

```
igraph::betweenness(karate_dw, normalized = T) # weights are automatically considered
```

```
##        Mr Hi      Actor 2      Actor 3      Actor 4      Actor 5      Actor 6      A
## 0.0000000000 0.0021306818 0.0075757576 0.0040246212 0.0014204545 0.0000000000 0.0018
##     Actor 24     Actor 25     Actor 26     Actor 27     Actor 28     Actor 29     Ac
## 0.0000000000 0.0000000000 0.0037878788 0.0000000000 0.0000000000 0.0018939394 0.0009
##     Actor 17       John A
## 0.0000000000 0.0000000000
```

7.5.1 Weighted betweenness centrality

If your network is weighted, weights are automatically included in the computation of betweenness centrality. If you prefer that edge weights are *not* considered, include `weights = NA` in the argument. Notice that some of the values have changed depending on whether weights are considered or not.

If your network is unweighted, then weights are not considered by default.

```
igraph::betweenness(karate_dw, normalized = T) # weights are automatically considered
```

```
##        Mr Hi      Actor 2      Actor 3      Actor 4      Actor 5      Actor 6      A
## 0.0000000000 0.0021306818 0.0075757576 0.0040246212 0.0014204545 0.0000000000 0.0018
##     Actor 24     Actor 25     Actor 26     Actor 27     Actor 28     Actor 29     Ac
## 0.0000000000 0.0000000000 0.0037878788 0.0000000000 0.0000000000 0.0018939394 0.0009
##     Actor 17       John A
## 0.0000000000 0.0000000000
```

```
igraph::betweenness(karate_dw, normalized = T, weights = NA) # weights are ignored
```

```
##        Mr Hi      Actor 2      Actor 3      Actor 4      Actor 5      Actor 6      Actor 7
## 0.0000000000 0.0004734848 0.0083648990 0.0018939394 0.0000000000 0.0004734848 0.0014204545 0.0
##     Actor 24     Actor 25     Actor 26     Actor 27     Actor 28     Actor 29     Actor 30
## 0.0000000000 0.0000000000 0.0009469697 0.0000000000 0.0006313131 0.0020517677 0.0009469697 0.0
##     Actor 17       John A
## 0.0000000000 0.0000000000
```

7.5.2 Directed betweenness centrality

As shown in the example below for *John A*, different betweenness centrality values are returned depending on whether betweenness centrality should be computed by considering the directionality of the edges (`directed = TRUE`), or if edge direction should be ignored (`directed = FALSE`). Basically, if `directed = TRUE`, the shortest paths are only possible between node pairs where the edge directions are aligned (i.e., A->B->C), but if `directed = FALSE` then shortest paths are possible as long as there is an existing edge and direction of the edges is ignored.

```
igraph::betweenness(karate_dw, normalized = T, weights = NA, directed = TRUE, v = 'John A')
```

```
## John A
##      0
```

```
igraph::betweenness(karate_dw, normalized = T, weights = NA, directed = FALSE, v = 'John A')
```

```
##   John A
## 0.304075
```

Quick note: I've included `igraph::betweenness` to tell R to use the `betweenness` function from the `igraph` library and not the one from `influenceR` library (Chapter 12). This is important because sometimes different libraries have functions with the same name.

7.6 Page Rank centrality

Page Rank centrality is a centrality measure developed by Google to rank webpages (the historic paper describing the algorithm can be viewed here). The general idea is that a random walker will traverse the network space and their

paths are biased by the link connectivity structure of the network. The random walker restarts the walk after some time (simulating "boredom" of the surfer). The number of visits received by a node provides an indicator of its importance in the network. Intuitively, we expect that nodes have a high Page Rank score if there are many nodes that point to it, or if the nodes that point to it themselves have a high Page Rank score.

The `weights` and `directed` arguments can be adjusted depending on your graph type. It is important to note that the interpretation of edge weights here is that of "connection strength" (Reproduced from the `igraph` manual: "This function interprets edge weights as connection strengths. In the random surfer model, an edge with a larger weight is more likely to be selected by the surfer."). This is different from the "distance" interpretation of edge weights in the closeness and betweenness centralities sections.

To get the Page Rank centrality of all nodes in the network, try running the following code:

```
# undirected, unweighted centralities
page_rank(graph = karate_dw, directed = F, weights = NA)$vector
```

```
##       Mr Hi     Actor 2     Actor 3     Actor 4     Actor 5     Actor 6     Actor 7
## 0.096997285 0.052876924 0.057078509 0.035859858 0.021977952 0.029111155 0.029111155
##    Actor 26    Actor 27    Actor 28    Actor 29    Actor 30    Actor 31    Actor 32
## 0.021006197 0.015044038 0.025639767 0.019573459 0.026288538 0.024590155 0.037158087
```

```
# directed, weighted centralities (weights used by default)
page_rank(graph = karate_dw, directed = T)$vector
```

```
##      Mr Hi    Actor 2    Actor 3    Actor 4    Actor 5    Actor 6    Actor 7    Acto
## 0.01523537 0.01552315 0.01938528 0.01886916 0.01581093 0.01667427 0.03091729 0.0188
##   Actor 27   Actor 28   Actor 29   Actor 30   Actor 31   Actor 32   Actor 33    Acto
## 0.01523537 0.02484945 0.01798162 0.02141609 0.01939474 0.04512689 0.08402038 0.0235
```

```
# notice we include the "$vector" after the function because the function outputs
# a list object, and we only want the part of the object that contains the
# centrality values for the individual nodes
```

7.7 Other forms of node centrality

`igraph` is also able to compute the following node centrality scores:

- authority score

- eigenvector centrality; directed generalization is alpha centrality
- harmonic centrality
- hub centrality
- power centrality

If you interested to learn about these the links to the `igraph` documentation are included above. {May be further discussed in later versions of the book.}

7.8 Exporting node centrality measures for data analysis

As you can see from this chapter, it is fairly straightforward to obtain node centrality measures–you need to know which functions to use, and think about whether you want a variant of the measure that reflects properties of the network that are important for your research (e.g., directedness and weightedness of the edges). However, printing out the values in the Console is unwieldy, and makes it quite difficult to manipulate and analyze this information.

Below is a code chunk that you should be able to easily edit to extract the measures that you want for all nodes in a given network. It creates a data frame containing each centrality measure in a separate column and node names in another column. You can add more or fewer centrality measures in the data frame. Then you analyze the data frame object as per normal (e.g., are degree and local clustering coefficient of nodes positively correlated in this network?) or export the data frame so that you can merge it with other measures (e.g., gender of each person) and conduct statistical tests in another software (e.g., on average, do males have a higher closeness centrality than females?).

```
node_centrality_data <- data.frame(
    degree = degree(karate_dw, mode = 'all'), # column_name = centrality function
    out_strength = strength(karate_dw, mode = 'out'),
    uu_closeness = closeness(karate_dw, normalized = TRUE, mode = 'all', weights = NA)
    )

write.csv(node_centrality_data, file = 'results.csv')
```

7.9 Exercise

Your task is to:

1. Load the famous Krackhardt's Kite network from the `igraphdata` library using the following code:

```
library(igraphdata) # have you downloaded the package?

data("kite") # you should see a kite object appear in your Environment
```

2. Answer the following questions:

- How many nodes and edges does the network have?
- What type of edges does the network have?
- Find out which node(s) has:
  - the highest/lowest degree in the network?
  - the highest/lowest local clustering coefficient?
  - the highest/lowest betweenness centrality?
  - the highest/lowest closeness centrality?
  - the highest/lowest Page Rank centrality?

3. Save the node-level measures as a `.csv` file and open it in a spreadsheet program of your choice.

4. Do the same nodes have the highest degree, closeness, and betweenness centrality scores in this network? Try plotting the network and explain why different nodes have the highest scores on these centrality measures.

Chapter 8

Chapter 8: Meso-level network measures

A common feature of many real-world networks is that they have robust community structure. Nodes are considered to be part of the same community if the density of connections among those nodes is relatively higher than the density of connections between nodes from different communities (Newman, 2006).

Modularity, Q, is a measure of the density of links inside communities in relation to the density of links between communities (Fortunato, 2010). Networks with higher Q are said to show strong evidence of community structure.

8.1 How do network scientists "find" communities in networks?

Many community detection methods have been developed by network scientists to detect communities in networks. Each differs in their implementation, and reflects the creator's implicit definition of what is a community. In this section we will explore just 4 of these methods.

If you are interested to learn more about community detection, check out Fortunato (2010) who provided a comprehensive comparison of various community detection techniques.

In this section we will use the undirected, unweighted version of the karate network from Chapter 4 for demonstration.

```
karate_el <- read.csv('data/karate_el.csv', header = FALSE)
```

```
karate_uu <- graph_from_data_frame(karate_el, directed = FALSE)

summary(karate_uu)
```

```
## IGRAPH 5acc868 UN-- 34 78 --
## + attr: name (v/c)
```

As you will see in the rest of this chapter, the template for conducting a community detection analysis is identical across the four methods. `igraph` contains additional community detection algorithms not covered in this chapter, but you should be able to generalize the application of these algorithms based on the code provided below. Generally, community detection algorithms have in-built default parameters that are used to optimize performance. At the completion of the method, it returns the communities (i.e., how the individual nodes are grouped), and the modularity score based on this final grouping.

8.2 Edge betweenness ("divisive method")

The core idea behind the edge betweenness community detection method is that edges connecting separate communities tend to have high edge betweenness as all the shortest paths from one community to another must necessarily traverse through these edges.

The algorithm works by calculating the edge betweenness of all edges of the graph, and then removing the edge with the highest edge betweenness score. It recalculates the edge betweenness of remaining edges and again removes the one with the highest score. This repeats until modularity cannot be improved further.

Because there may be random components in the community detection algorithm, it is a good idea to use `set.seed` to ensure that your results are reproducible. Usually the algorithm is repeated several times, and the results are compared across runs. For now we will just run the algorithm once and return to this point at the end of this section.

For the edge betweenness method, we use the `cluster_edge_betweenness` function. Functions that begin with `cluster_` are community detection methods in `igraph`. The outputs of the community detection are saved into the `karate_edge` object.

```
set.seed(1)

karate_edge <- cluster_edge_betweenness(karate_uu)
```

Normally, we wish to know at least these 3 things from the community detection results, which we can easily retrieve from the `karate_edge` object:

1. The community membership of each node in the network (i.e., which nodes belong to the same community?). In the code below, this is merged with the node names and only the first 10 nodes are shown. It would be straightforward to save and export this information for further analyses; see the commented code below.
2. We also want to know how nodes are distributed across the communities. By using the `table` function we get a count of the community labels. Here, we can see that there are a total of 5 communities, the largest one consisting of 12 nodes (community 4) and the smallest one consisting of a single node (community 5).
3. The modularity of the final community grouping, which gives us a sense of the quality of partitions retrieved by the algorithm. Here, Q is 0.401.

```
# 1. membership of nodes in each community
data.frame(node = V(karate_uu)$name, community = karate_edge$membership) |>
  head(10) # view the first 10
```

```
##         node community
## 1     Mr Hi         1
## 2   Actor 2         1
## 3   Actor 3         2
## 4   Actor 4         1
## 5   Actor 5         3
## 6   Actor 6         3
## 7   Actor 7         3
## 8   Actor 9         4
## 9  Actor 10         5
## 10 Actor 14         1
```

```
### saving the results
# results <- data.frame(node = V(karate_uu)$name, community = karate_edge$membership)
# write.csv(results, file = 'results.csv')

# 2. distribution of nodes across communities
table(karate_edge$membership)
```

```
##
##  1  2  3  4  5
## 10  6  5 12  1
```

```
# 3. modularity of the network
modularity(karate_edge)
```

```
## [1] 0.4012985
```

In the subsequent sections, you will notice that the structure of the code/output is identical, except for the specific community detection function used.

8.3 Louvain method ("greedy, maximization method")

The core idea behind the Louvain community detection method is that communities are essentially "mergers" of small communities (Blondel et al., 2008), reflecting the self-similar nature of complex networks.

1. Each node is assigned to one community such that there are as many communities as there are nodes. Then remove node i from its community and place it in the community of the neighbor which yields the greatest gain in modularity. Repeat for all nodes in the network.
2. A new network is built where nodes are the communities found in the previous phase. Repeat Step 1.
3. Repeat Step 1 and 2 until it is not possible to further increase the value of Q.

For the Louvain method, we use the `cluster_louvain` function. The outputs of the community detection are saved into the `karate_louvain` object.

```
set.seed(1)

karate_louvain <- cluster_louvain(karate_uu)

# 1. membership of nodes in each community
data.frame(node = V(karate_uu)$name, community = karate_louvain$membership) |>
  head(10) # view the first 10
```

```
##        node community
## 1    Mr Hi         1
## 2   Actor 2         1
## 3   Actor 3         1
## 4   Actor 4         1
```

```
## 5   Actor 5         2
## 6   Actor 6         2
## 7   Actor 7         2
## 8   Actor 9         3
## 9  Actor 10         1
## 10 Actor 14         1
```

```
# 2. distribution of nodes across communities
table(karate_louvain$membership)
```

```
##
##  1  2  3  4
## 12  5 11  6
```

```
# 3. modularity of the network
modularity(karate_louvain)
```

```
## [1] 0.4188034
```

Notice that the Louvain method returned 4 communities ranging in size from 5 to 12, with a Q of 0.419.

8.4 Random walker ("dynamic method")

The core idea behind the random walker community detection method is that if there are communities in the network, a random walker will tend to spend more time inside the community than outside.

The Walktrap algorithm groups nodes together based on the similarities of the paths taken by the random walker starting from that node. The idea is to merge sets of vertices that have low "distance" from each other.

For the random walker method, we use the `cluster_walktrap` function. The outputs of the community detection are saved into the `karate_walktrap` object.

```
set.seed(1)

karate_walktrap <- cluster_walktrap(karate_uu)

# 1. membership of nodes in each community
data.frame(node = V(karate_uu)$name, community = karate_walktrap$membership) |>
  head(10) # view the first 10
```

```
##        node community
## 1     Mr Hi         1
## 2   Actor 2         1
## 3   Actor 3         2
## 4   Actor 4         1
## 5   Actor 5         5
## 6   Actor 6         5
## 7   Actor 7         5
## 8   Actor 9         2
## 9  Actor 10         2
## 10 Actor 14         2
```

```
# 2. distribution of nodes across communities
table(karate_walktrap$membership)
```

```
##
## 1 2 3 4 5
## 9 7 9 4 5
```

```
# 3. modularity of the network
modularity(karate_walktrap)
```

```
## [1] 0.3532216
```

Notice that the random walker method returned 5 communities ranging in size from 5 to 9, with a Q of 0.353.

8.5 Infomap ("information-theoretic method")

The core idea behind the Infomap community detection method is to leverage on information-theoretic methods to "describe" the information flow of the entire system (based on random walks).

The Infomap algorithm attempts to describe the random walker's trajectory using the fewest number of "bits" of information. Communities are groups of nodes that receive new "names" during the compression process.

For the Infomap method, we use the `cluster_infomap` function. The outputs of the community detection are saved into the `karate_infomap` object.

```
set.seed(1)

karate_infomap <- cluster_infomap(karate_uu)
```

```
# 1. membership of nodes in each community
data.frame(node = V(karate_uu)$name, community = karate_infomap$membership) |>
  head(10) # view the first 10
```

```
##        node community
## 1     Mr Hi         1
## 2   Actor 2         1
## 3   Actor 3         1
## 4   Actor 4         1
## 5   Actor 5         2
## 6   Actor 6         2
## 7   Actor 7         2
## 8   Actor 9         3
## 9  Actor 10         1
## 10 Actor 14         1
```

```
# 2. distribution of nodes across communities
table(karate_infomap$membership)
```

```
##
##  1  2  3
## 12  5 17
```

```
# 3. modularity of the network
modularity(karate_infomap)
```

```
## [1] 0.4020381
```

Notice that the Infomap method returned 3 communities ranging in size from 5 to 17, with a Q of 0.402.

8.6 Comparison of methods

Fortunato (2010) conducted a comprehensive comparison of community detection techniques. Generally, Rosvall & Bergstorm's Infomap and Blondel et al.'s greedy modularity maximization method performed the best. Both also were relatively fast algorithms that were efficient for large networks.

The code below illustrates the similarities and differences in the results of the various community detection methods.

8.6.1 Comparing modularity

In this example, the Louvain method yielded community groupings that led to the highest Q score of 0.419, whereas the Walktrap method yielded the lowest Q score of 0.353. Overall, there is moderately strong evidence of community structure in the karate network with modularity scores approximating ~0.40 across the four community detection methods used.

```
modularity(karate_edge)
```

```
## [1] 0.4012985
```

```
modularity(karate_louvain)
```

```
## [1] 0.4188034
```

```
modularity(karate_walktrap)
```

```
## [1] 0.3532216
```

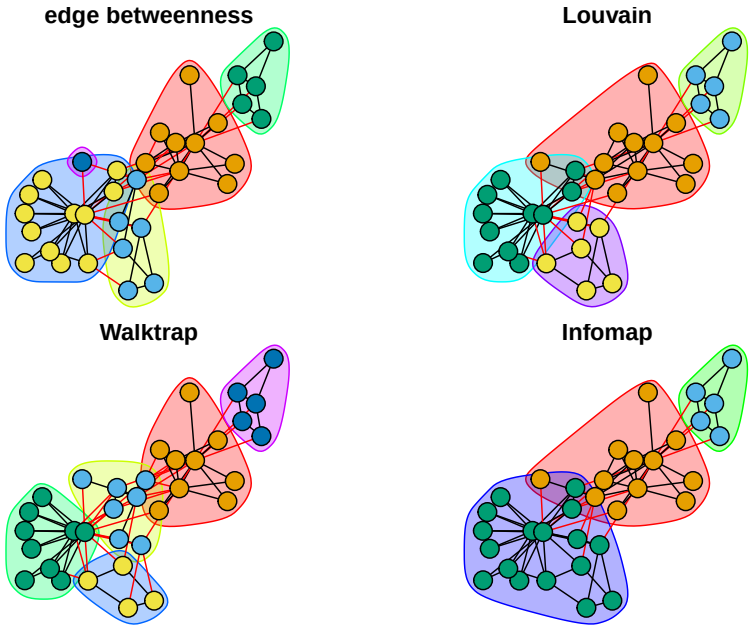
```
modularity(karate_infomap)
```

```
## [1] 0.4020381
```

8.6.2 Comparing community membership

The code below provides the following visualization that compares how different methods are grouping nodes into communities.

```
par(mar=c(0,0,0,0)+.6, mfrow = c(2,2)) # reduce margins and plot both networks togethe

set.seed(1)
fixed_l <- layout_with_fr(karate_uu) # to fix node layout across plots

plot(karate_edge, karate_uu, layout = fixed_l, main = 'edge betweenness', vertex.label
plot(karate_louvain, karate_uu, layout = fixed_l, main = 'Louvain', vertex.label = NA)
plot(karate_walktrap, karate_uu, layout = fixed_l, main = 'Walktrap', vertex.label = N
plot(karate_infomap, karate_uu, layout = fixed_l, main = 'Infomap', vertex.label = NA)
```



More formally, we can make use of statistical methods such as the Fleiss' Kappa test to quantify the extent to which the groupings are consistent across the 4 methods.

```
# first, compile the community membership results from all methods
compare_membership <- data.frame(
  edge = karate_edge$membership,
  louvain = karate_louvain$membership,
  walktrap = karate_walktrap$membership,
  infomap = karate_infomap$membership
)

# compute consistency that two nodes tend to be found in the same cluster
library(irr) # we need this library for the kappa test of interrater agreement
```

```
## Loading required package: lpSolve
```

```
kappam.fleiss(compare_membership)
```

```
##  Fleiss' Kappa for m Raters
##
##  Subjects = 34
##    Raters = 4
##     Kappa = 0.332
```

```
##
##         z = 8.46
##   p-value = 0
```

Although the Kappa test is significant at $p < .05$, indicating that the groupings are more consistent than expected from chance, the Kappa value of 0.332 is relatively low. This suggests that the community detection methods are returning similar but also somewhat different results in the community membership of individual nodes.

{Note: You may need to relabel so that '1' corresponds to most popular category, etc.}

8.7 Things to note

Before ending this chapter it is worth highlighting the following.

8.7.1 Weighted and directed graphs

Some of the community detection methods listed above can make use of weighted and directed information in the community detection process; please refer to the documentation for details. Typically, you would need to modify the arguments in the `cluster_` function to indicate that edge weights and direction should be considered. Please note that the interpretation of edge weights as distances or connection strength (see Chapter 7) can differ depending on the community detection method. Again, refer to the documentation for more information.

8.7.2 Networks with multiple components

As discussed in Chapter 6, not all nodes in the network necessarily forms a single connected component. In networks containing multiple network components, it is very likely that nodes in the smaller components are grouped together as a community in any community detection method. This is because doing so naturally optimizes modularity; hence, communities would never contain nodes that come from different components. It may make more sense to focus the community detection on the largest connected component of the network.

8.7.3 Calculating modularity for self-defined communities

It is possible to compute modularity based on pre-specified communities. For instance, you may have prior information on the subgroups that each node in the network belongs to (i.e., community labels that are extrinsic to the network

structure; e.g., student interest groups or majors) and you want to see how those groupings correspond to the connectivity structure.

If the modularity of those labels are quite high and comparable to the community detection results, it would suggest that nodes that belong to the same extrinsic group also have relatively higher density of connections internally, relative to the density of connections across different groups.

In the toy example below, I provided "made-up" groups to the karate network through the node-attribute "suburb". Because these labels were randomly generated, we would expect modularity computed based on these groups to be very low or close to zero.

```
# let's assume we have information about the suburb that each karate member lives in
V(karate_uu)$suburb <- sample(1:5, gorder(karate_uu), replace = TRUE)

modularity(karate_uu, membership = V(karate_uu)$suburb) # specify your own labels
```

```
## [1] -0.06402038
```

8.7.4 Comparing different runs

As mentioned in the introduction of this chapter, you can get slightly different results even when re-running the same community detection method on the same network (assuming that you did not initially set a stable random seed). It is a good idea to repeat this multiple times to check if your results are generally consistent. The code below shows you how to run the algorithm multiple times and save the outputs into a list object that you can later investigate further.

As you can see there are minor variations in the modularity scores for each instance, and the very high Kappa score of 0.95 indicates highly consistent communities across runs.

```
cluster_results <- list()

for(i in 1:5) { # change 5 to a different number if you want more runs
  set.seed(i)

  cluster_results[[i]] <- cluster_louvain(karate_uu)
}

# compare Q
sapply(cluster_results, modularity)
```

```
## [1] 0.4188034 0.4188034 0.4188034 0.4155983 0.4155983
```

```
# compare consistency of membership
# first, compile the community membership results from all runs into a single df
df1 <- lapply(cluster_results, function(x) x$membership)
df1_merged <- do.call(cbind, df1)

kappam.fleiss(df1_merged)
```

```
##  Fleiss' Kappa for m Raters
##
##  Subjects = 34
##    Raters = 5
##     Kappa = 0.95
##
##         z = 28.6
##   p-value = 0
```

8.7.5 Other community detection methods

Several other community detection methods can be implemented in `igraph`:

- cluster_fast_greedy - fast greedy modularity optimization algorithm
- cluster_fluid_communities - community detection algorithm based on interacting fluids
- cluster_label_prop - community detection based on majority voting of label in the neighborhood of the vertex
- cluster_leading_eigen - community detection based on the leading eigenvector
- cluster_leiden - Leiden algorithm
- cluster_optimal - community detection by maximizing the modularity measure over all possible partitions.
- cluster_spinglass - find communities in graphs via a spin-glass model and simulated annealing

8.8 Exercise

1. Pick a network for your own choosing for this exercise.

2. Pick one of the 4 community detection methods that you've learned about. Conduct a community detection analysis with that method.

- What is the modularity, Q, of the network?

- How many communities were detected? What was the size of the largest one? What was the size of the smallest one?

3. Bonus questions:

- Can you save the output of the node community membership results as a `.csv` file?
- Which community detection method produced the highest modularity value?
- What is the meaning of these communities specific to the network that you picked for this exercise?

Chapter 9

Chapter 9: Network models

Under construction.

Planned topics: - ER networks - WS networks - Configuration networks - Preferential attachment network model

Chapter 10

Chapter 10: Behaviors in the network

Networks are ultimately theoretical, mathematical objects that offer behavioral scientists a useful framework to study and characterize phenomena that they are interested in studying. As we have seen in the previous chapters, relational data from various subfields of psychology can be represented as a network, and network science provides the tools and measures that enable us to study its micro, meso, and macro-level structure. However, what arguably makes networks really meaningful are the processes that operate on the network.

A key reason why we spend so much time discussing measures of network structure is because an underlying assumption is that the network structure has meaningful implications on network processes, and it is through these network processes whereby complex behaviors emerge from the networks. For instance, the presence of echo chambers in social networks and the emergence of epilepsy episodes in brain activation networks highlight how it is not just the structure of the network that contributes to these phenomena, but also the mechanisms and processes that occur within that network structure. In other words, the mechanism implemented on the network structure represents a key theoretical linkage between the abstract mathematical representation and the psychological construct.

In this first iteration of this chapter, we explore how *spreading activation*, which is a key construct in the cognitive sciences, can be implemented on a language network to provide a computational account of empirical phenomena. In future iterations, I plan to include other approaches of implementing "behaviors" in the network, including random walks and SIR models.

10.1 Introduction to spreadr

First, let's download and load the spreadr R package.

```
# install.packages('spreadr')
# remotes::install_github('csqsiew/spreadr') # to get the most updated version from my

library(spreadr)
library(igraph)
```

spreadr is a R package that enables the user to implement the spreading activation mechanism in a network structure (Siew, 2019). Although the concept of spreading activation is very prominent in psychology research (Collins & Loftus, 1975; Dell, 1986), there are not many accessible ways of formally implementing this idea on an explicit cognitive structure. spreadr assumes that activation is a limited cognitive resource that can be passed from one node to another node, as long as those two nodes are connected. This "passing" of activation to a neighboring node mimics the idea that an activated node can activate other related nodes. This process is assumed to proceed in a parallel fashion across all nodes in the network that have a non-zero activation value assigned to them, and are connected to other nodes that could receive activation from them. For details on the algorithm used, see Siew (2019).

In order to implement the spreading activation mechanism, we need to specify a number of parameters:

1. network: the network igraph object to conduct the simulation on
2. time: number of time steps to continue for (not actual time)
3. start_run: a data frame object that has two columns ("node" and "activation") that specifies which nodes receive how many activation units at time = 0 (advanced: it is possible to specify different times at which the node receive activation as well)
4. retention: This represents the proportion of activation that remains in the node (not spread) at each time step. Then, 1 - retention of the activation at each node is spread to neighbouring nodes.
5. decay: Number from 0 to 1 (inclusive) representing the proportion of activation that is lost at each time step.
6. suppress: Number representing the maximum amount of activation in a node for it to be set to 0, at each time step.

```
# use the pnet network that comes in the spreadr package
pnet
```

```
## IGRAPH a7c0363 UN-- 34 96 --
```

```
## + attr: name (v/c)
## + edges from a7c0363 (vertex names):
##  [1] spike --speak  spike --spoke  spike --speck  spike --spook  spoke --spook  spoke --speck
## [15] speed --stead  speed --seed   speed --speech speak --speed  stead --seed   seed  --seek
## [29] speak --speech speech--peach  peach --peace  peek  --peach  peach --peep   peach --pea
## [43] peach --pitch  peach --patch  peach --pooch  peach --poach  peach --preach peach --reach
## [57] peep  --peat   peace --peep   peek  --peep   peel  --peas   peat  --peel   peace --peel
## [71] leach --reach  leach --each   leach --beach  teach --beach  teach --each   teach --reach
## [85] poach --perch  patch --perch  pouch --patch  pitch --patch  patch --pooch  patch --poach
```

```r
# starting activation values (time = 0)
start_run <- data.frame(
  node = c("beach"),
  activation = c(20))

# run the simulation
result <- spreadr(network = pnet, start_run = start_run,
                  retention = 0.5, decay = 0, suppress = 0,
                  time = 2, include_t0 = TRUE)

# view the result
result
```

```
##       node activation time
## 1    spike  0.0000000    0
## 2    speak  0.0000000    0
## 3    spoke  0.0000000    0
## 4    speck  0.0000000    0
## 5    spook  0.0000000    0
## 6     sped  0.0000000    0
## 7    speed  0.0000000    0
## 8     spud  0.0000000    0
## 9    stead  0.0000000    0
## 10    seed  0.0000000    0
## 11  speech  0.0000000    0
## 12    seek  0.0000000    0
## 13   sneak  0.0000000    0
## 14   sleek  0.0000000    0
## 15    peek  0.0000000    0
## 16   peach  0.0000000    0
## 17   peace  0.0000000    0
## 18    peep  0.0000000    0
## 19     pea  0.0000000    0
## 20    peat  0.0000000    0
## 21    peel  0.0000000    0
```

```
## 22    peas  0.0000000    0
## 23   teach  0.0000000    0
## 24   leach  0.0000000    0
## 25   beach 20.0000000    0
## 26    each  0.0000000    0
## 27   pouch  0.0000000    0
## 28   pitch  0.0000000    0
## 29   patch  0.0000000    0
## 30   pooch  0.0000000    0
## 31   poach  0.0000000    0
## 32  preach  0.0000000    0
## 33   reach  0.0000000    0
## 34   perch  0.0000000    0
## 35   spike  0.0000000    1
## 36   speak  0.0000000    1
## 37   spoke  0.0000000    1
## 38   speck  0.0000000    1
## 39   spook  0.0000000    1
## 40    sped  0.0000000    1
## 41   speed  0.0000000    1
## 42    spud  0.0000000    1
## 43   stead  0.0000000    1
## 44    seed  0.0000000    1
## 45  speech  0.0000000    1
## 46    seek  0.0000000    1
## 47   sneak  0.0000000    1
## 48   sleek  0.0000000    1
## 49    peek  0.0000000    1
## 50   peach  2.0000000    1
## 51   peace  0.0000000    1
## 52    peep  0.0000000    1
## 53     pea  0.0000000    1
## 54    peat  0.0000000    1
## 55    peel  0.0000000    1
## 56    peas  0.0000000    1
## 57   teach  2.0000000    1
## 58   leach  2.0000000    1
## 59   beach 10.0000000    1
## 60    each  2.0000000    1
## 61   pouch  0.0000000    1
## 62   pitch  0.0000000    1
## 63   patch  0.0000000    1
## 64   pooch  0.0000000    1
## 65   poach  0.0000000    1
## 66  preach  0.0000000    1
## 67   reach  2.0000000    1
```

```
## 68   perch  0.0000000    1
## 69   spike  0.0000000    2
## 70   speak  0.0000000    2
## 71   spoke  0.0000000    2
## 72   speck  0.0000000    2
## 73   spook  0.0000000    2
## 74    sped  0.0000000    2
## 75   speed  0.0000000    2
## 76    spud  0.0000000    2
## 77   stead  0.0000000    2
## 78    seed  0.0000000    2
## 79  speech  0.0500000    2
## 80    seek  0.0000000    2
## 81   sneak  0.0000000    2
## 82   sleek  0.0000000    2
## 83    peek  0.0500000    2
## 84   peach  2.7666667    2
## 85   peace  0.0500000    2
## 86    peep  0.0500000    2
## 87     pea  0.0500000    2
## 88    peat  0.0500000    2
## 89    peel  0.0500000    2
## 90    peas  0.0500000    2
## 91   teach  2.6166667    2
## 92   leach  2.6166667    2
## 93   beach  5.8166667    2
## 94    each  2.6166667    2
## 95   pouch  0.0500000    2
## 96   pitch  0.0500000    2
## 97   patch  0.0500000    2
## 98   pooch  0.0500000    2
## 99   poach  0.0500000    2
## 100 preach  0.2166667    2
## 101  reach  2.6500000    2
## 102  perch  0.0500000    2
```

It is possible to change various parameters of the simulation in order to get it to mimic the situation that you want. For instance, you could assign activation to more than 1 node in the network, different starting activation values, different network structures, different values for the retention, decay, and suppress parameters. You can also allow the simulation to continue for several time steps. There are also multiple ways to analyze the end result. You may choose to focus on a couple of target nodes' final activation levels, or look at how activation is distributed across the entire network or among a subset of nodes. At the end of the day, what you decide to do should align with your research questions so that the simulations can help you test a specific idea or question that you had

of your network.

10.2 Case study: False memories

Vitevitch et al. (2012) (Experiment 1) conducted a false memory task comparing false alarm rates for words with high and low clustering coefficients. In this task, the phonological neighbors of the target word were presented to the participants, but the target word was never presented. Then, the participants were asked to recall as many words as they could. The authors found that participants falsely recognized more words with low C than high C, suggesting that the lower connectivity structure of the low C word led to more activation spreading to the target word, increasing false alarm rates.

In this simulation, we explore if their empirical result could be replicated computationally using `spreadr`. The `chapter10-networks.RData` contains two phonological networks containing the two target words each, "seethe" and "wrist". Although both words have the same degree of 16, "seethe" has a higher local C (0.49) than "wrist" (0.16).

In the simulation below, equal levels of activation are assigned to all neighbors of the target, but never the target node itself. Activation is allowed to spread for a fixed number of time steps, and the final activation values of the target nodes are retrieved at the final time step. A higher final activation of target node is taken as an indicator of more false alarms in memory retrieval.

Question: Based on the outputs of the simulation below, which word has the higher final activation value? Does this result align with the empirical result reported by Vitevitch et al. (2012)?

```
load('data/chapter10-networks.RData')

library(tidyverse)

# low C network
start_run_low <- data.frame(
  node = neighbors(lowC, v = 'rIst;wrist')$name,
  activation = rep(10, 16))

result_low <- spreadr(network = lowC, start_run = start_run_low,
                  retention = 0.5, decay = 0, suppress = 0,
                  time = 5, include_t0 = TRUE)

result_low |> filter(node == 'rIst;wrist', time == 5)
```

```
##         node activation time
## 1 rIst;wrist   8.516831    5
```

```
# high C network
start_run_high <- data.frame(
  node = neighbors(highC, v = 'siD;seethe')$name,
  activation = rep(10, 16))

result_high <- spreadr(network = highC, start_run = start_run_high,
                   retention = 0.5, decay = 0, suppress = 0,
                   time = 5, include_t0 = TRUE)

result_high |> filter(node == 'siD;seethe', time == 5)
```

```
##         node activation time
## 1 siD;seethe   3.149976    5
```

10.3 Exercise: Design your experiment!

For this exercise, try to design a simulation study using spreading activation, on any network of your choosing. Some questions to help you with this:

- What would I learn from doing this simulation?
- What does the notion of "spreading activation" mean for the specific network that I've chosen?
- How should the simulation be set up? What are the starting values and parameters? And why were these specific values chosen?
- How should the outputs be analyzed? Why?
- What are your expected results? Did the actual results align with your initial expectations?
- What did I learn from doing this simulation?

10.4 References

Collins, A. M., & Loftus, E. F. (1975). A spreading-activation theory of semantic processing. *Psychological Review, 82*(6), 407–428.

Dell, G. S. (1986). A spreading-activation theory of retrieval in sentence production. *Psychological Review, 93*(3), 283.

Siew, C. S. Q. (2019). spreadr: An R package to simulate spreading activation in a network. *Behavior Research Methods, 51*(2), 910–929. https://doi.org/10.3758/s13428-018-1186-5

Vitevitch, M. S., Ercal, G., & Adagarla, B. (2011). Simulating retrieval from a highly clustered network: Implications for spoken word recognition. *Frontiers in Psychology, 2*, 369.

Vitevitch, M. S., Chan, K. Y., & Roodenrys, S. (2012). Complex network structure influences processing in long-term and short-term memory. *Journal of Memory and Language, 67*(1), 30–44. https://doi.org/10.1016/j.jml.2012.02.008

10.5 Future topics under this chapter

- random walker
- SIR models

Chapter 11

Chapter 11: `keyplayer` package

Borgatti (2006) formalized the key player problem based on two different goals that the set of key players to be identified in the social network is expected to fulfil.

The negative version of the key player problem searches for nodes that, when removed from the network, causes the maximum amount of fragmentation of the network.

The positive version of the key player problem searches for nodes that, when activated, can spread information to the largest proportion of the rest of the network.

1. Key Player Problem-Negative (maximize fragmentation)
2. Key Player Problem-Positive (maximize cohesiveness)

The rationale for developing algorithms that maximize fragmentation or cohesiveness is due to the limitations of standard network measures like closeness or betweenness centrality, which Borgatti (2006) argues are not optimized to solve the key player problem. Borgatti (2006) also argues that selecting a set, or ensemble, of nodes that work together to solve the key player problem is more optimal than selecting the "top" nodes based on their individual centrality values.

The general steps of a key player analysis are as follows:

1. Select the size of your keyplayer set (up to you and depends on your application; a general rule of thumb is no larger than 10% of the total number of nodes in the network).

2. A random set of nodes is selected at first (hence important to set a seed).
3. (For the current set) Compute the Fragmentation/Cohesion score.
4. Choose a different set of nodes (guided by various node centrality measures depending on the specific algorithm used).
5. Repeat steps 3 and 4 until Fragmentation/Cohesion score cannot be increased further after a certain number of attempts.

11.1 Set up

First, let's install the `keyplayer` R package:

```
install.packages('keyplayer')
```

We will use the `macaque` network from the `igraphdata` R package for demonstration. This dataset comes from Négyessy et al., where they used network science to study the cortical pathways from the primary somatosensory cortex to the primary visual cortex in the macaque monkey brain. The network consists of 45 nodes representing 45 brain areas (30 visual and 15 sensorimotor), and 463 directed and unweighted edges. An edge indicates the presence of a pathway or axonal tracts *from* brain area i *to* brain area j, as identified through the use of tracers.

```
library(igraphdata)

data("macaque")

# load the other packages that we need
library(igraph)
library(keyplayer)
```

```
##
## Attaching package: 'keyplayer'
```

```
## The following object is masked from 'package:igraph':
##
##     contract
```

11.2 Key Player Problem-Negative

The goal of the KPP-Neg is to maximize Fragmentation.

Fragmentation, F, is the ratio between the number of node pairs that are not connected once the set of key players have been removed, and the total number of node pairs in the original fully connected network.

$F_{min} = 0$ indicates that the network consists of a single component. $F_{max} = 1$ indicates that the network has been completely fractured, solely consisting of isolates, or nodes with no connections (i.e., every node is unreachable). The KPP-Neg aims to find the set of key players that would maximize F.

To run the key player analysis, we make use of the `kpset` function from the `keyplayer` library. Notice that there are a number of additional parameters that need to be specified.

(i) `adj.matrix`: which refers to the network, that needs to be first converted into an adjacency matrix for the function to work. We can use the `igraph` function `as_adjacency_matrix` to do this.
(ii) `size`: which refers to the number of key players or size of the key player set
(iii) `type`: needs to be `"fragment"` for the negative version
(iv) `method`: grouping criterion; documentation suggests that the `"min"` method should be used for fragment centrality.
(v) `binary`: I set this to *TRUE* so that the edges are treated as unweighted (which is the case for the `macaque` network anyway), but change this to *FALSE* if you would like to have the edge weights be included.

```
set.seed(1)

results <- kpset(adj.matrix = as_adjacency_matrix(macaque),
                 size = 3,
                 type = "fragment",
                 method = "min",
                 binary = T)
```

```
## This graph was created by an old(er) igraph version.
##   Call upgrade_graph() on it to use with the current igraph version
##   For now we convert it on the fly...
```

```
results
```

```
## $keyplayers
## [1] 13 15 30
##
## $centrality
## [1] 0.484427
```

Notice that the results of the key player algorithm is stored in an object called `results`, which is a list object containing two elements. The `centrality` element (which is accessed via `results$centrality`) provides the fragmentation score of the final set of keyplayers (i.e., the extent to which the network fragments when these nodes are removed). The second element is the optimal set of key players based on the goal of fragmentation which is stored in `results$keyplayers`. Because it records the "position" of the keyplayer nodes within the node order of the graph, we can treat it as a vector to extract the names of the keyplayer nodes as shown below.

```
# fragmentation score
results$centrality
```

```
## [1] 0.484427
```

```
# to map the keyplayer number to node names
V(macaque)$name[results$keyplayers]
```

```
## [1] "LIP" "VIP" "46"
```

The set of 3 brain areas that leads to the most amount of fragmentation in the macaque brain network when removed is [LIP, VIP, and 46], with a fragmentation score, F, of 0.484.

11.3 Key Player Problem-Positive

The goal of KPP-Pos is to maximize Cohesion.

Cohesion, C, is defined as the amount of connection between the key player set and the rest of the graph. It measures the number of unique nodes that can be reached from the key player set in a given number of steps (usually, steps = 1).

$C_{min} = 0$ indicates that the KP set is infinitely far from all other nodes in the network. $C_{max} = 1$ indicates that the KP set is immediately adjacent (steps = 1) to all other nodes in the network (excluding the keyplayer nodes themselves). The KPP-Pos aims to find the set of key players that would maximize C.

To run the key player analysis, we make use of the `kpset` function from the `keyplayer` library. Notice that there are a number of additional parameters that need to be specified.

(i) `adj.matrix`: which refers to the network, that needs to be first converted into an adjacency matrix for the function to work. We can use the `igraph` function `as_adjacency_matrix` to do this.

(ii) `size`: which refers to the number of key players or size of the key player set
(iii) `type`: needs to be `"diffusion"` for the positive version
(iv) `method`: grouping criterion; documentation suggests that the `"union"` method should be used for cohesion centrality. `T = 1` indicates that we allow for the diffusion to spread for 1 step from the set of keyplayer nodes when computing for cohesion centrality.
(v) `binary`: I set this to *TRUE* so that the edges are treated as unweighted (which is the case for the `macaque` network anyway), but change this to *FALSE* if you would like to have the edge weights be included.

```
set.seed(1)

results <- kpset(as_adjacency_matrix(macaque),
                 size = 3,
                 type = "diffusion",
                 method = "union",
                 T = 1,
                 binary = T)

results
```

```
## $keyplayers
## [1]  5 15 27
##
## $centrality
## [1] 38
```

Notice that the results of the key player algorithm is stored in an object called `results`, which is a list object containing two elements. The `centrality` element (which is accessed via `results$centrality`) provides the cohesion score of the final set of keyplayers (i.e., the number of nodes in the network that is accessible from the key player set by 1 step (i.e., `T = 1`), but excluding the key players themselves). It would be important to normalize this value by the total number of nodes in the network (minus the number of nodes in the key player set) so that we get a cohesion score that reflects the proportion of coverage of the network offered by the key players. The second element is the optimal set of key players based on the goal of fragmentation which is stored in `results$keyplayers`. Because it records the "position" of the keyplayer nodes within the node order of the graph, we can treat it as a vector to extract the names of the keyplayer nodes as shown below.

```
# cohesion score
results$centrality/(gorder(macaque) - length(results$keyplayers))
```

```
## [1] 0.9047619
```

```
# notice that we subtract the keyplayers from the full network size

# to map the keyplayer number to node names
V(macaque)$name[results$keyplayers]
```

```
## [1] "V4"  "VIP" "TF"
```

11.4 References

Borgatti, S. P. (2006). Identifying sets of key players in a social network. Computational and Mathematical Organization Theory, 12(1), 21-34.

Négyessy, L., Nepusz, T., Kocsis, L., & Bazsó, F. (2006). Prediction of the main cortical areas and connections involved in the tactile function of the visual cortex by network analysis. European Journal of Neuroscience, 23(7), 1919-1930.

11.5 Exercise

Repeat the key player analysis (both versions) on the macaque brain network, but with the following changes:

- change the size of the keyplayer set to a number other than 3
- repeat the analysis a few times but change the seed each time

1. Compare your results across these attempts and between the positive and negative version of the key player problem. How consistent are your results? Which nodes are commonly chosen as key players?
2. How do the fragmentation and cohesion scores change as the size of the key player set increases?
3. What are the implications of the key players for the macaque brain network?

Chapter 12

Chapter 12: `influenceR` package

In this chapter, we explore two additional micro-level network metrics that are developed in the domain of social network analysis. For this, we need to install the `influenceR` R package which provides the functions for computing bridging scores and effective network sizes for nodes in the network.

12.1 Set up

First, let's install the `influenceR` R package:

```
install.packages('influenceR')
```

We will continue to use the `macaque` network from Chapter 11 for demonstration.

```
library(igraphdata)

data("macaque")

# load the other packages that we need
library(igraph)
library(influenceR)
```

```
##
## Attaching package: 'influenceR'
```

```
## The following objects are masked from 'package:igraph':
##
##     betweenness, constraint
```

```
# to make edges undirected by collapsing directed edges
macaque <- as.undirected(macaque, mode = 'collapse')
```

```
## This graph was created by an old(er) igraph version.
##   Call upgrade_graph() on it to use with the current igraph version
##   For now we convert it on the fly...
```

12.2 Bridging score

Granovetter (1973) observed that edges that reduce the overall distance of a network are structurally important bridges. These are usually the "weak" edges in a social network where the nature of the relationship between those two individuals are on an acquaintance basis. Nevertheless, the presence of these edges enable connection between distant parts of the network.

Valente and Fujimoto (2010) built on that insight and developed a "node-centric" measure of bridging that considers the impact of deleting an edge or link on the cohesiveness of the network. In other words, a node with a higher bridging score tends to possess a structurally important edge, such that deleting it substantially reduces the overall connectivity of the network.

A node's bridging score, B_i, is the average decrease in cohesiveness when each of its edges are systematically removed from the graph.

$B_i = \frac{\sum \Delta C}{k_i}$

where k_i = the degree of node i and ΔC = change in network cohesiveness.

For each connected edge of a target node:

1. Remove the edge and recompute network cohesiveness (inverse of distance)
2. Take the difference from the original network cohesiveness (i.e., the decrease)

Then, take the sum of the change in cohesiveness, divide by the number of edges (degree).

Below, we can easily use the `bridging` function to compute this information for us.

```
bridging(macaque) # from influenceR
```

```
##           V1           V2           V3          V3A           V4          V4t          VOT
## 0.0003156566 0.0003310887 0.0002976190 0.0003108003 0.0005071549 0.0003282828 0.0002693603 0.0
##           7a          FST         PITd         PITv         CITd         CITv         AITd
## 0.0003900112 0.0002702463 0.0003367003 0.0002840909 0.0004349046 0.0003682660 0.0003198653 0.0
##            1            2            5           Ri          SII           7b            4
## 0.0003787879 0.0003749617 0.0003979186 0.0004103535 0.0005536131 0.0003561254 0.0003114478 0.0
```

```
bridging_data <- data.frame(node = V(macaque)$name,
                            bridge_score = bridging(macaque))

head(bridging_data)
```

```
##     node bridge_score
## V1    V1 0.0003156566
## V2    V2 0.0003310887
## V3    V3 0.0002976190
## V3A  V3A 0.0003108003
## V4    V4 0.0005071549
## V4t  V4t 0.0003282828
```

12.2.1 Important note about directed graphs

The `bridging` function does not work as expected when edges are directed. Negative scores and inconsistent scores could arise from running this function on graphs with directed edges. Hence, please convert directed graphs into undirected graphs before proceeding (see Chapter 5.5, `as.undirected()` function). The root cause of this is because of a parallelization invoked in the function to compute the change in the cohesion when removing each edge, which does not behave consistently when edges are directed.

12.3 Effective network science

Burt (2004) argues that people who are located near "holes" in a social network tend to have better ideas. This occurs because such individuals are "brokers" across different social groups and can accumulate social capital more quickly.

The effective size of a node's ego network is based on the concept of redundancy. A person's ego network has redundancy to the extent that their contacts are connected to each other as well. The nonredundant part of a person's relationships is the effective size of their ego network. The maximum ENS is the degree

of the ego. The minimum ENS is 1, as if one "effectively" only had 1 single friend.

A node's effective network size, E_i, is computed as follows:

$E_i = k - 2t/k$

where t = number of ties in the ego network and k = the degree of the node.

For each node:

1. Extract the ego network of the node. The ego network is the immediate neighborhood of the node.
2. Check the extent to which its neighbors are connected by counting number of ties, t.

Below, we can easily use the `ens` function to compute this information for us.

```
ens(macaque) # from influenceR
```

```
##        V1        V2        V3       V3A        V4       V4t       VOT        VP
##  2.000000  6.200000  5.428571  4.076923 12.900000  3.200000  3.000000  6.000000  6.1
##      CITv      AITd      AITv      STPp      STPa        TF        TH       FEF
##  3.750000  5.600000  3.000000  6.400000  1.800000 11.941176  7.333333 13.454545 13.9
##        Id        35        36
##  2.500000  3.000000  4.666667
```

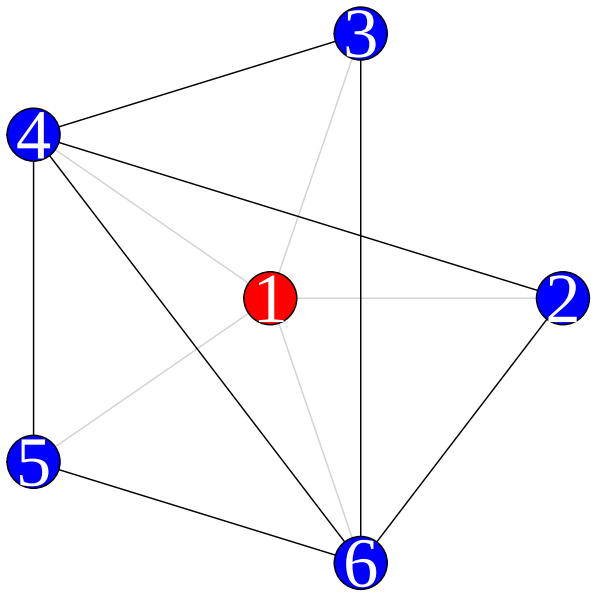
```
ens_data <- data.frame(node = V(macaque)$name,
                       ens_score = ens(macaque))

head(ens_data)
```

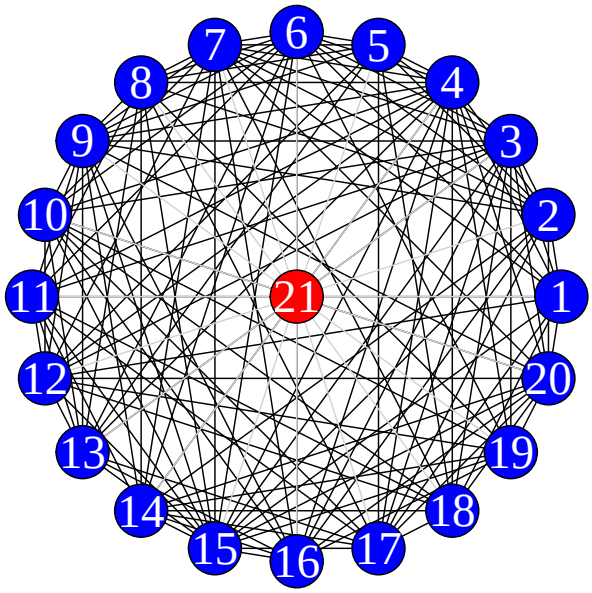
```
##     node ens_score
## V1    V1  2.000000
## V2    V2  6.200000
## V3    V3  5.428571
## V3A  V3A  4.076923
## V4    V4 12.900000
## V4t  V4t  3.200000
```

By now, you should know how to export the data frames as a .csv file for further exploration! (Hint: `write.csv(...)`).

12.3.1 How is effective network science different from local clustering coefficient?

A common question that arises is whether ENS is really capturing information that is different from that of local clustering coefficient (Chapter 7.3), since both measures are micro-level network measures that quantify the density of the internal connectivity structure of a node's immediate neighbors. To illustrate how they are different we consider two toy networks (visualized below). For the first network we are interested in ego node 1, which has a degree of 5, and for the second network we are interested in ego node 21, which has a degree of 20.





Let's compute the ENS and local C for the ego node in the smaller neighborhood (using both the corresponding functions and the mathematical equations):

```
# the ego node is node 1 in the g_small network

transitivity(g_small, type = 'local', vids = 1)
```

```
## [1] 0.7
```

```
# number of ties / number of possible ties among neighbors
# t / (k(k-1)/2)
7/(5*(5-1)/2)
```

```
## [1] 0.7
```

```
ens(g_small)[1]
```

```
## [1] 2.2
```

```
# k - (2t/k)
5 - (2*7/5)
```

```
## [1] 2.2
```

Let's also compute the ENS and local C for the ego node in the larger neighborhood (using both the corresponding functions and the mathematical equations):

```
# the ego node is node 21 in the g_large network

transitivity(g_large, type = 'local', vids = 21)
```

```
##  21
## 0.7
```

```
# number of ties / number of possible ties among neighbors
# t / (k(k-1)/2)
133/(20*(20-1)/2)
```

```
## [1] 0.7
```

```
ens(g_large)[21]
```

```
##  21
## 6.7
```

```
# k - (2t/k)
20 - (2*133/20)
```

```
## [1] 6.7
```

What you will notice is that although both nodes have a local C of 0.7, indicating that 70% of all possible connections in their immediate neighborhoods are fulfilled, the two nodes have different effective network sizes. The node from the larger neighborhood has a ENS of 6.7 whereas the node from the smaller neighborhood has a ENS of 2.2. The difference in ENS reflects the fact that the node from the larger neighborhood has more potential in its ability to fill/exploit structural holes in their network than the node from the smaller neighborhood, due to the former having a larger neighborhood to begin with. This is despite both nodes having the same level of interconnectivity within their neighborhoods.

12.3.2 Important note about directed graphs

The `ens` function does not work as expected when edges are directed. Negative scores and scores less than 1 could arise from running this function on graphs with directed edges. Hence, please convert directed graphs into undirected graphs before proceeding (see Chapter 5.5, `as.undirected()` function).

12.4 References

Burt, R. S. (2004). Structural holes and good ideas. *American Journal of Sociology, 110*(2), 349-399.

Granovetter, M. S. (1973). The strength of weak ties. *American Journal of Sociology, 78*(6), 1360-1380.

Valente, T. W., & Fujimoto, K. (2010). Bridging: locating critical connectors in a network. *Social Networks, 32*(3), 212-220.

12.5 Exercise

Conduct the following analysis on a social network of your choosing, where the nodes correspond to individual persons. If you do not have a social network, you could conduct the analysis on the `karate` or `UKfaculty` networks from the `igraphdata` library.

1. Identify the individuals with the highest and lowest bridging scores.

- What is the implication of having a high bridging score for this particular network?
- What is the implication of having a low bridging score for this particular network?

2. Identify the individuals with the highest and lowest effective network sizes.

- What is the implication of having a high effective network size for this particular network?
- What is the implication of having a low effective network size for this particular network?

Chapter 13

Chapter 13: `bootnet` package

Psychometrics is the science of psychological measurement. It attempts to measure psychological constructs that are not usually directly observable. The core question in psychometrics is how to relate observable information (questionnaire responses, behaviors, test performance) to theoretical psychological constructs.

The underlying assumption is that the relationship between observables and psychological constructs is that of *common cause*. In other words, there is a hypothesized latent variable (representing the unobservable construct; e.g., intelligence) that causes or leads to the behaviors and responses that are observed (e.g., scores on Raven's Progressive Matrices).

In recent years, a new area known as **network psychometrics** has developed and gained substantial attention among psychometricians. Instead of trying to uncover a latent variable from observable data, network psychometrics focuses on analyzing the patterns of covariances among observable data as a network representation. This enables researchers to leverage on the network science framework in order to quantify their structural properties.

For instance, Borsboom and Cramer (2013) argue that the complexity of psychopathology demands more sophisticated models that acknowledge its complexity. Traditional psychometric methods that assume a latent variable implicitly assume that clinical symptoms are caused by the disorder itself, but the evidence suggests that symptoms of mental disorders have reliable patterns of covariance that cannot be easily reduced to a single cause or latent variable. Instead of treating mental disorders as a "disease" that can be solved by finding the underlying cause(s) (i.e., the "disease" model in western medicine), psychometic network models view mental disorders as interactive systems of symptoms with reinforcing negative feedback loops and causal relationships.

13.1 Chapter outline

Throughout this book we have seen various examples of behavioral networks, and learned about the importance of clearly defining what the nodes and edges are representing. What are the nodes and edges of a psychometric network?

- Nodes: observed variables. In the example that we will work through, these would be the items on the survey that correspond to symptoms associated with post-traumatic stress disorder (PTSD).
- Edges: strength of association between the observed variables. For instance, the strength of the correlation of two symptoms in the PTSD symptom list. This information has to be estimated from the data itself.

In the rest of this chapter we will work through a concrete example to show you how to convert data from a survey to a network representation that you can analyze using the functions you've learned about earlier. Thereafter you should be able to apply the same steps to your own data to get the corresponding psychometric network.

13.2 Installation of `bootnet`

We need to first install the `bootnet` R library.

```
install.packages('bootnet')
```

13.2.1 `qgraph` vs. `igraph`

It is important to keep in mind that that network objects created by the `bootnet` R package are *not* `igraph` objects; instead, they are `qgraph` objects, a different class of data objects specially created for `bootnet` and other packages for psychometric network analysis. In order for you to use the code provided in this book, we will need to first extract the adjacency matrix representation of this object and convert it to an `igraph` network object. I will show you how to do this later in the chapter.

13.3 Dataset

The dataset that we will use in this chapter comes from 359 women who were enrolled in a community substance program and met the criteria for either PTSD or sub-threshold PTSD, based on the DSM-IV. This data was obtained from Hien et al. (2009) and adapted slightly for teaching purposes.

They completed the PTSD Symptom Scale-Self Report (PSS-SR; Foa et al., 1993) by rating the frequency at which they experienced each of symptoms on the following scale: 1 (not at all), 2 (once a week), 3 (2 - 4 times a week), 4 (5 or more times a week).

Foa, E. B., Riggs, D. S., Dancu, C. V., & Rothbaum, B. O. (1993). Reliability and validity of a brief instrument for assessing post-traumatic stress disorder. Journal of Traumatic Stress, 6(4), 459-473.

Hien DA, Wells EA, Jiang H, Suarez-Morales L, Campbell AN, Cohen LR, Zhang Y, et al. Multisite randomized trial of behavioral interventions for women with co-occurring ptsd and substance use disorders. Journal of Consulting and Clinical Psychology. 2009;77(4):607–619. doi: 10.1037/a0016227

As we can see below, there are 17 items in the PSS-SR, corresponding to the columns of the data frame, and 359 responses corresponding to the rows of the data frame. The responses are ordinal and range from 1 to 5.

```
load('data/ptsd_freq.RData') # load pre-processed data

qs_freq_wide[1:3,1:5] # each row = 1 subject, each column = 1 item
```

```
## # A tibble: 3 x 5
##   `AVOID REMINDERS OF THE TRAUMA` `BAD DREAMS ABOUT THE TRAUMA` `BEING JUMPY OR EASILY STARTLE
##                             <dbl>                         <dbl>                             <db
## 1                               4                             1
## 2                               1                             3
## 3                               4                             2
```

```
colnames(qs_freq_wide) # list of items
```

```
##  [1] "AVOID REMINDERS OF THE TRAUMA"  "BAD DREAMS ABOUT THE TRAUMA"    "BEING JUMPY OR EASILY
##  [7] "FEELING IRRITABLE"              "FEELING PLANS WONT COME TRUE"   "HAVING TROUBLE CONCENT
## [13] "NOT THINKING ABOUT TRAUMA"      "PHYSICAL REACTIONS"             "RELIVING THE TRAUMA"
```

13.4 Estimate a partial correlation network

To represent the associations across the 17 items in our dataset, we will estimate a *partial correlation* network. The partial correlation depicts the strength of association of 2 variables after conditioning on other variables.

We make use of the `estimateNetwork` function from bootnet to help us do this. To estimate a partial correlation network, we need to specify an additional argument, `default = "pcor"`.

```
library(bootnet)
```

```
## This is bootnet 1.6
```

```
## For questions and issues, please see github.com/SachaEpskamp/bootnet.
```

```
ptsd_network <- estimateNetwork(qs_freq_wide, default = "pcor")
```

```
## Estimating Network. Using package::function:
##   - qgraph::qgraph(..., graph = 'pcor') for network computation
##   - psych::corr.p for significance thresholding
```

```
# pcor = partial correlation network

summary(ptsd_network) # how many nodes and edges does this network have?
```

```
##
## === Estimated network ===
## Number of nodes: 17
## Number of non-zero edges: 136 / 136
## Mean weight: 0.05363509
## Network stored in object$graph
##
## Default set used: pcor
##
## Use plot(object) to plot estimated network
## Use bootnet(object) to bootstrap edge weights and centrality indices
##
## Relevant references:
##
##      Epskamp, S., Borsboom, D., & Fried, E. I. (2016). Estimating psychological netw
```

First, notice, that the `summary` function gives us a very different output than what we are used to. This is because `ptsd_network` is a `qgraph` object, not an `igraph` object. That said, the output is quite straightforward and you should be able to tell that there are 17 nodes and 136/136 non-zero edges.

Before proceeding, note that there are several different methods to estimate a network from psychometric data (type in `?estimateNetwork` into the console to see all of the different estimator methods that `bootnet` can use). For demonstration purposes, I have shown you just one of those approaches (i.e., partial correlation). In practice, which method you decide on depends on the properties of your data (i.e, is it categorical, continuous, normally distributed?) and how conservative you want your model to be. See "Additional resources".

13.4.1 Thresholding

`ptsd_network` contains 17 nodes and 136/136 non-zero edges. Notice that the number 136 corresponds to the maximum number of edges possible in a network of 17 nodes (test this for your yourself by calculating the number of possible combinations of 2 from 17 options). The presence of 136 non-zero edges indicates that this is in fact a fully connected graph.

A fully connected network is a rather meaningless representation to work with as there will be little to no variance in the node-level measures. Because many of these correlations depict very weak relationships, we can remove such non-significant correlations by including the argument `threshold = 'sig'`.

```
ptsd_network <- estimateNetwork(qs_freq_wide, default = "pcor",
                                threshold = 'sig')
```

```
## Estimating Network. Using package::function:
##   - qgraph::qgraph(..., graph = 'pcor') for network computation
##   - psych::corr.p for significance thresholding
```

```
# pcor = partial correlation network, only retain significant edges

summary(ptsd_network) # how many nodes and edges does this network have?
```

```
##
## === Estimated network ===
## Number of nodes: 17
## Number of non-zero edges: 31 / 136
## Mean weight: 0.03630966
## Network stored in object$graph
##
## Default set used: pcor
##
## Use plot(object) to plot estimated network
## Use bootnet(object) to bootstrap edge weights and centrality indices
##
## Relevant references:
##
##      Epskamp, S., Borsboom, D., & Fried, E. I. (2016). Estimating psychological networks and t
```

After including statistical significance as a threshold to remove edges depicting weak correlations, `ptsd_network` contains 17 nodes and 31 edges.

13.4.2 Absolutizing

We can view the adjacency matrix of the network from running the following:

```
ptsd_network$graph
```

```
##            AROTT      BDATT      BJOES       BOA    DOCOFP       FEN       FEI     F
## AROTT  0.0000000  0.0000000  0.0000000 0.1555554 0.0000000 0.0000000 0.0000000 0.000
## BDATT  0.0000000  0.0000000  0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.000
## BJOES  0.0000000  0.0000000  0.0000000 0.4075190 0.0000000 0.0000000 0.0000000 0.000
## BOA    0.1555554  0.0000000  0.4075190 0.0000000 0.0000000 0.1423142 0.0000000 0.000
## DOCOFP 0.0000000  0.0000000  0.0000000 0.0000000 0.0000000 0.2357181 0.1334469 0.000
## FEN    0.0000000  0.0000000  0.0000000 0.1423142 0.2357181 0.0000000 0.1410481 0.147
## FEI    0.0000000  0.0000000  0.0000000 0.0000000 0.1334469 0.1410481 0.0000000 0.121
## FPWCT  0.0000000  0.0000000  0.0000000 0.0000000 0.0000000 0.1474119 0.1214582 0.000
## HTC    0.0000000 -0.1221041  0.0000000 0.0000000 0.0000000 0.0000000 0.1793871 0.000
## HTS    0.0000000  0.0000000  0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.000
## LIIA   0.0000000  0.0000000  0.1129132 0.0000000 0.3509726 0.0000000 0.0000000 0.204
## NATR   0.0000000  0.0000000  0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.000
## NTAT   0.1515577  0.0000000  0.1204094 0.0000000 0.0000000 0.0000000 0.0000000 0.000
## PHR    0.1746399  0.1089284  0.1692879 0.0000000 0.0000000 0.0000000 0.0000000 0.000
## RTT    0.0000000  0.0000000  0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.000
## UWROT  0.0000000  0.1461690 -0.1109757 0.0000000 0.0000000 0.0000000 0.0000000 0.000
## UTOI   0.0000000  0.1710409  0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.000
```

You may notice from the adjacency matrix that some partial correlations are negative. For instance, the correlation between HTC and BDATT is -0.1221041. However, when edge weights are negative, certain network measures cannot be computed and so we need to convert them to a positive value (but maintain the magnitude of the effect). Edges that depict a "positive" or "negative" meaning are associated with the theory of signed networks, which are beyond the scope of this book. Interested readers can refer to the `signnet` R package for further information about how to analyze signed networks.

The code below shows you how to absolutize the matrix and assigns the output as a new data object `my_adj_mat`.

```
# make all edge weights positive
my_adj_mat <- abs(ptsd_network$graph)

my_adj_mat[1:10,1:7]
```

```
##            AROTT      BDATT    BJOES       BOA    DOCOFP       FEN       FEI
## AROTT  0.0000000 0.0000000 0.000000 0.1555554 0.0000000 0.0000000 0.0000000
## BDATT  0.0000000 0.0000000 0.000000 0.0000000 0.0000000 0.0000000 0.0000000
```

```
## BJOES  0.0000000 0.0000000 0.000000 0.4075190 0.0000000 0.0000000 0.0000000
## BOA    0.1555554 0.0000000 0.407519 0.0000000 0.0000000 0.1423142 0.0000000
## DOCOFP 0.0000000 0.0000000 0.000000 0.0000000 0.0000000 0.2357181 0.1334469
## FEN    0.0000000 0.0000000 0.000000 0.1423142 0.2357181 0.0000000 0.1410481
## FEI    0.0000000 0.0000000 0.000000 0.0000000 0.1334469 0.1410481 0.0000000
## FPWCT  0.0000000 0.0000000 0.000000 0.0000000 0.0000000 0.1474119 0.1214582
## HTC    0.0000000 0.1221041 0.000000 0.0000000 0.0000000 0.0000000 0.1793871
## HTS    0.0000000 0.0000000 0.000000 0.0000000 0.0000000 0.0000000 0.0000000
```

13.4.3 Conversion to igraph

In the final step, we need to convert the matrix object into an `igraph` network, using a function that should be familiar to you by now! (see Chapter 2/3).

```
library(igraph) # load the package

# why we should use this particular function?
ptsd_network_igraph <- graph_from_adjacency_matrix(my_adj_mat,
                                                   mode = 'undirected',
                                                   weighted = TRUE)

summary(ptsd_network_igraph)
```

```
## IGRAPH d4fe76d UNW- 17 31 --
## + attr: name (v/c), weight (e/n)
```

We can easily see that the network contains 17 nodes and 31 edges, and is an undirected and weighted graph. The weights correspond to the strength of the partial correlation between those two items, and only significant correlations were retained as edges. The edges are undirected because direction of causality cannot be inferred from the correlations.

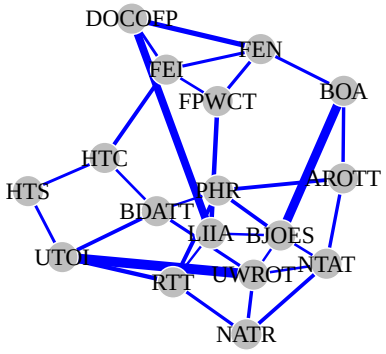
Thereafter, we can easily compute various network measures for the network.

```
# degree of nodes
degree(ptsd_network_igraph)
```

```
##  AROTT  BDATT  BJOES    BOA DOCOFP    FEN    FEI  FPWCT    HTC    HTS   LIIA   NATR   NTAT
##      3      4      5      3      3      4      4      3      3      2      4      3      4
```

```
# global clustering coefficient
transitivity(ptsd_network_igraph, type = 'global')
```

```
## [1] 0.1724138
```



13.5 Sample sizes

A common pitfall when trying to use `bootnet` to estimate a network occurs when sample sizes are simply too small in order to robustly estimate the covariance structure (`bootnet` throws an error or returns a network with 0 edges). In such cases, the best solution is to collect more data or find larger datasets. If one insists on sticking with the low-quality dataset, it is always possible to estimate a correlation matrix through other packages and conduct your own thresholding, since it is non-trivial to convert any numeric (adjacency) matrix into a network. But this is not recommended practice.

13.6 Additional resources

Epskamp, S., Borsboom, D., & Fried, E. I. (2018). Estimating psychological networks and their accuracy: A tutorial paper. Behavior Research Methods, 50(1), 195-212.

Snijders, T. A., & Borgatti, S. P. (1999). Non-parametric standard errors and tests for network statistics. Connections, 22(2), 161-170.

Isvoranu, A.-M., Epskamp, S., Waldorp, L., & Borsboom, D. (Eds.). (2022). Network Psychometrics with R: A Guide for Behavioral and Social Scientists (1st ed.). Routledge. https://doi.org/10.4324/9781003111238

Chapter 14

Chapter 14: Network visualization in igraph

This chapter aims to introduce you the basics of network visualization. Note that for a very large network (> 100 nodes) it becomes very difficult to create good visualizations using basic `igraph` plotting functions. You may need to venture to other R packages (see Chapter 15 - under development). A nice open-source software for creating great network visualizations is Gephi.

14.1 Set up

We will use the `karate` network from Chapter 5, which contains various node and edge attributes.

```
library(igraph)
library(igraphdata)
library(tidyverse)

data('karate')

# import your node attributes
node_info <- read.csv('data/karate_nodes_added.csv', header = T)

# very important to ensure that node order is identical
node_info <- node_info %>% arrange(factor(node, levels = V(karate)$name))
```

```
## This graph was created by an old(er) igraph version.
##   Call upgrade_graph() on it to use with the current igraph version
```

```
##   For now we convert it on the fly...
```

```
identical(V(karate)$name, node_info$node) # sanity check that the node name order is i
```

```
## [1] TRUE
```

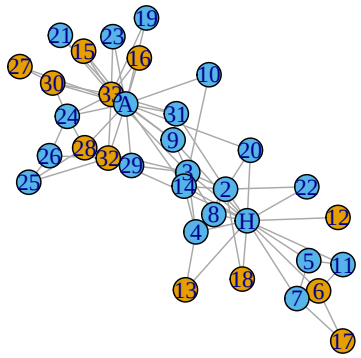
```
# add the 'gender' attribute
karate <- set_vertex_attr(karate,
                          name = 'gender',
                          value = node_info$gender)

# add the 'belt' attribute
karate <- set_vertex_attr(karate,
                          name = 'belt',
                          value = node_info$belt)

# add the 'age' attribute
karate <- set_vertex_attr(karate,
                          name = 'age',
                          value = node_info$age)

# initialize all edges with the same label
E(karate)$edge_type <- 'same'

# re-assign those with mixed edges to a new label
E(karate)$edge_type[E(karate)[V(karate)[V(karate)$gender == 'male'] %--% V(karate)[V(ka

summary(karate) # there should be a new edge type
```

```
## IGRAPH 4b458a1 UNW- 34 78 -- Zachary's karate club network
## + attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name (v/c), label
```

14.2 Visualizing node attributes

14.2.1 Color nodes by their categorical attributes

```
# color nodes by their attributes (categorical)
plot(karate, vertex.color = factor( V(karate)$gender )) # notice the use of factor()
```



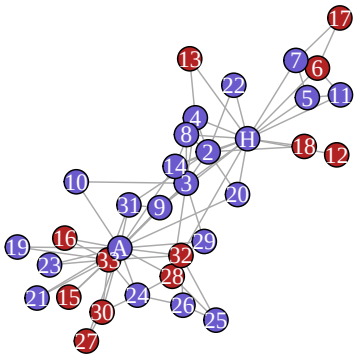

```
# plot(karate, vertex.color = V(karate)$gender) # this throws an error
```

If you would like to specify your own colors, you can refer to this website for the names of colors in R and assign them to each category as shown in the following code:

```
factor( V(karate)$gender ) |> levels() # order of levels is female male
```

```
## [1] "female" "male"
```

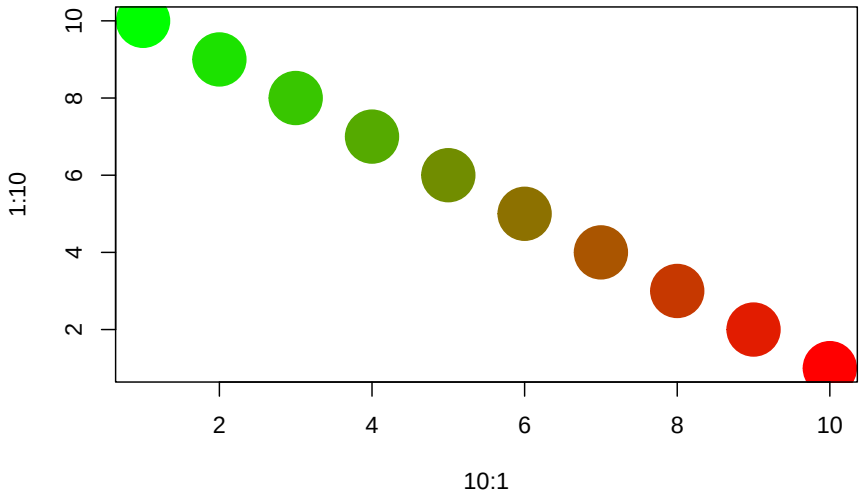
```
my_colors <- c('firebrick', 'slateblue') # firebrick = female, slateblue = male

# use of square brackets to map the colors onto the labels
plot(karate, vertex.color = my_colors[ factor( V(karate)$gender ) ] ,
     vertex.label.color = 'white' # to make the labels visible
     )
```



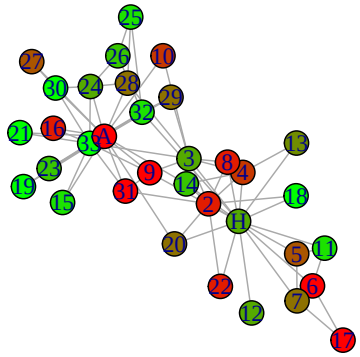
Can you try replacing the `gender` attribute with the `belt` attribute?

14.2.2 Color nodes by their continuous attributes

```
# color nodes by their attributes (continuous)

# a function to generate color gradients from a pre-specified palette
palf <- colorRampPalette(c("red", "green")) # smaller values = white

plot(x=10:1, y=1:10, pch=19, cex=5, col=palf(10))
```



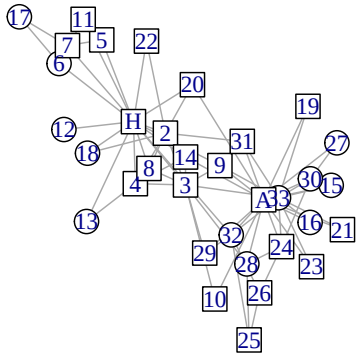

```
plot(karate, vertex.color = palf(10)[cut(V(karate)$age, 10)]) # notice the use of cut()
```



```
# cut() chops up the age distribution into 10 bins, each bin is assigned to the color
# the number of bins and color bins must be the same
```

14.2.3 Node shapes by their categorical attributes

```
# give categorical attributes different shapes
plot(karate,
     vertex.shape = c('circle', 'square')[factor(V(karate)$gender)], # notice the use
     vertex.color = 'white')
```

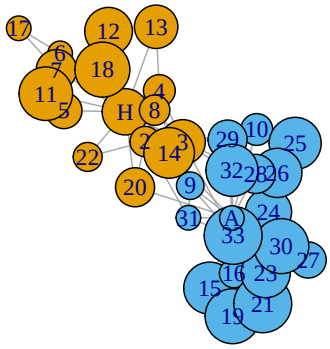



```
factor(V(karate)$gender) |> levels() # female/circle, male/square
```

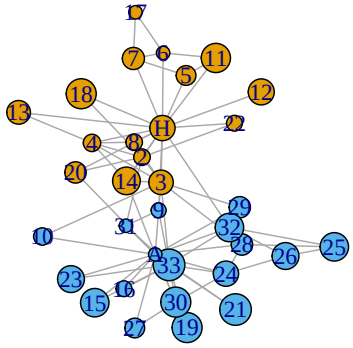
```
## [1] "female" "male"
```

14.2.4 Node size by their continuous attributes

```
# give continuous attributes different sizes

plot(karate, vertex.size = V(karate)$age) # notice that the sizes are not very well sc
```

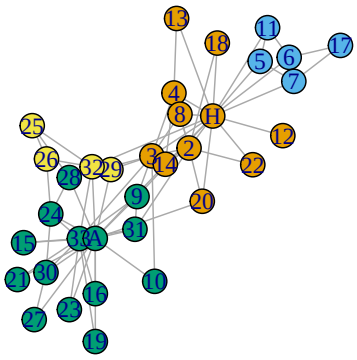


```
plot(karate, vertex.size = 0.5*V(karate)$age) # you will have to play around to find a suitable s
```



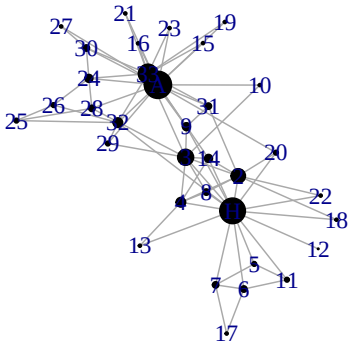
14.2.5 Example 1: Color nodes by their community membership

```
set.seed(1)
community_result <- cluster_louvain(karate)

V(karate)$community <- community_result$membership

plot(karate, vertex.color = V(karate)$community)
```



14.2.6 Example 2: Size nodes by their degree

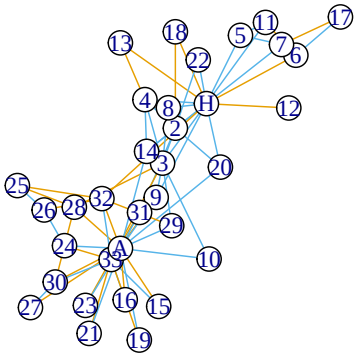
```
plot(karate, vertex.size = degree(karate), vertex.color = 'black') # the scaling issue
```



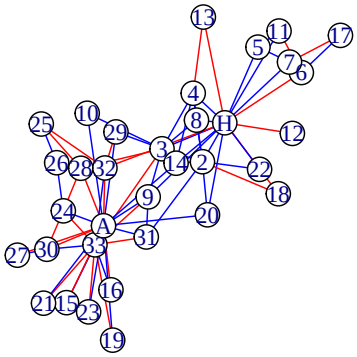
14.3 Visualizing edge attributes

14.3.1 Color edges by their categorical attributes

```
plot(karate, vertex.color = 'white', edge.color = factor(E(karate)$edge_type)) # note the use of
```




```
plot(karate, vertex.color = 'white', edge.color = c('red', 'blue')[factor(E(karate)$edg
```

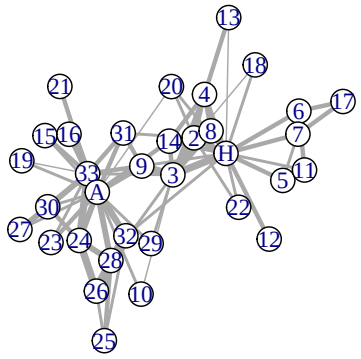


```
factor(E(karate)$edge_type) |> levels() # this tells you the ordering of the levels so you can te
```

```
## [1] "different" "same"
```

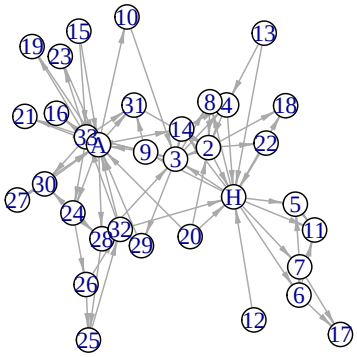
14.3.2 Edge thickness based on their continuous attributes

```
plot(karate, vertex.color = 'white', edge.width = E(karate)$weight)
```



14.3.3 Arrows for directed edges

```
plot(as.directed(karate, mode = 'random'), vertex.color = 'white', edge.arrow.width = 0.5, edge.a
```

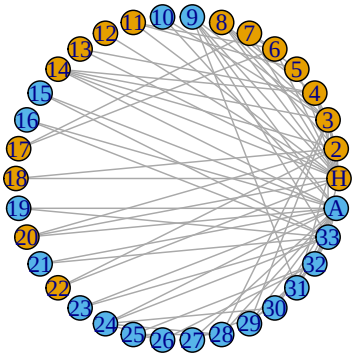


14.4 Other

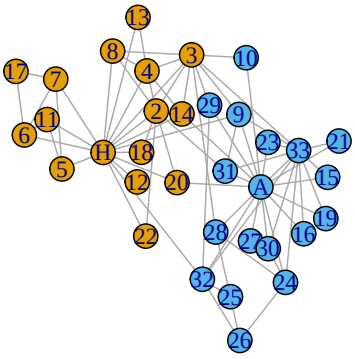
14.4.1 Graph layouts

It is good practice to set a seed before executing the graph layout functions, so that you are able to "save" the output and replicate it in the future. Notice that the layout function takes the network as its argument and outputs a 2 column matrix which is the x- and y-coordinates for each of the nodes in the network (explore the `my_layout1` R object). Then you can insert that as the input to the layout parameter when you want to plot the network.

```
## saving a fixed layout for future use
set.seed(1)
my_layout1 <- layout_in_circle(karate)

set.seed(2)
my_layout2 <- layout_with_dh(karate)

plot(karate, layout = my_layout1)
```



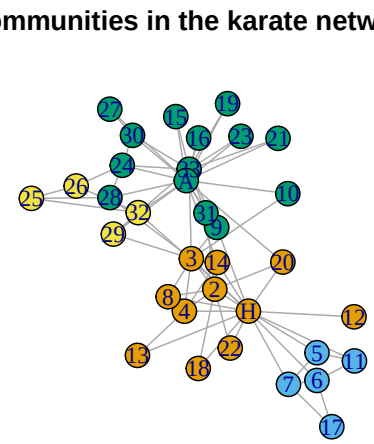

```
plot(karate, layout = my_layout2)
```



See https://kateto.net/network-visualization (Section 4.2 Network layouts) to learn more about the different plotting formats available in `igraph`.

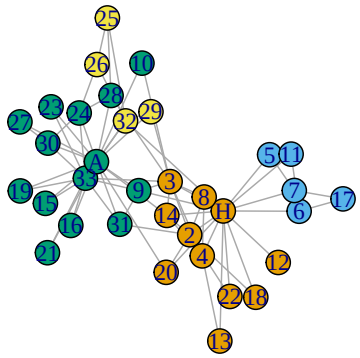
14.4.2 Plot title

```
## add a plot title ----
plot(karate, vertex.color = V(karate)$community, main = 'Communities in the karate netw
```



14.4.3 Add a subtitle

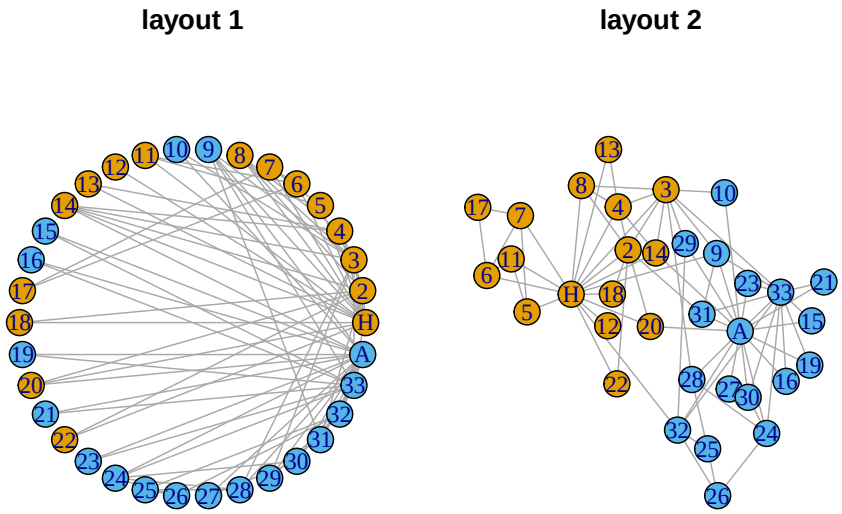
```
## add a subtitle ----
plot(karate, vertex.color = V(karate)$community, main = 'Communities in the karate netw
     sub = 'The Louvain community detection method was used.')
```



The Louvain community detection method was used.

14.4.4 Multiple plots

```
## multiple plots ----

par(mar=c(0,0,0,0)+1, mfrow = c(1,2)) # adjust margins, 1 row, 2 columns

plot(karate, layout = my_layout1, main = 'layout 1')
plot(karate, layout = my_layout2, main = 'layout 2')
```



14.4.5 Useful links

Introduction to network visualization in R: https://kateto.net/network-visualization

Colors in R: https://r-charts.com/colors/

Chapter 15

Chapter 15: Network visualization with other packages

Under construction.

Planned topics: - gephi - networkD3 - threejs - visNetwork

Chapter 16

Chapter 16: Analyzing degree distributions

Early work in the era of modern network science has revealed that the degree distributions of real-world networks followed a power-law (Barabasi & Albert, 2002). This was an important discovery because the fact that a power law could characterize the distribution of node degrees across a wide, diverse range of networks suggested that a common mechanism or organizing principle could be responsible for such a pattern. This led Barabasi and colleagues to propose a prominent model of network growth, known as *preferential attachment*, that provided a possible mechanism for the emergence of a power law degree distributions in networks.

However, we need to first understand two concepts before proceeding: *degree distribution* and *power law*.

Degree distribution refers to the probability distribution of node degrees in a given network. For instance, what proportion of nodes in the network have a degree of x, where x represents various degree values in the network?

Power law is a class of long-tailed distributions where the relationship between two variables is best characterized by an exponent (power relation). If a degree distribution follows a power law, the relation between the probability distribution of node degrees and the value of the degree itself is characterized by an exponential function, as follows:

$p(x) = Cx^{-\alpha}$

Where $p(x)$ is the probability of x occurring in the dataset, C = a constant, α = exponent or power of the relationship between x and $p(x)$. The exponent is negative because it is a negative relationship: As x (node degree) increases, the probability (of it occurring) decreases. This mathematical relationship captures

the idea that hubs (nodes with extremely high degree) are more uncommon in the network than nodes with lower degree.

In this Chapter, we will work through an example of how to analyze the distribution of a variable, and to statistically assess if its distribution is indeed similar to a power law distribution. The example is drawn from the tutorials provided by the creator of the `poweRlaw` R package, see the vignettes on https://cran.r-project.org/web/packages/poweRlaw/index.html for a more detailed walk through.

16.1 Set up

First, let's make sure to download and then load the `poweRLaw` R package into our workspace.

```
# install.packages('poweRlaw')

library('poweRlaw')
```

For this example, we will use the Moby Dick data set provided in the `poweRlaw` package. The `moby` R object is a vector of numbers, which each number corresponds to the raw frequency count of words found in the novel Moby Dick. The vector is also organized from the most frequently occurring word, to the least frequently occurring word. Notice that the identity of the words was not provided in the vector, but usually function words are the most frequently words in language corpora (e.g., the, of, to, etc.). Finally, although the example is on word frequencies, the same procedure can be applied to any quantity, e.g. node degrees in the network.

```
data("moby", package = "poweRlaw")

head(moby)
```

```
## [1] 14086  6414  6260  4573  4484  4040
```

```
tail(moby)
```

```
## [1] 1 1 1 1 1 1
```

16.2 Fitting the power law distribution

Then, we run the following code to fit the data to a power law distribution and infer two key parameters, alpha and x_{min}.

```
# assign a power law distribution "object" to the data
m_pl = displ$new(moby)

# infer parameters (alpha, minimum cut off)
est = estimate_xmin(m_pl)
est
```

```
## $gof
## [1] 0.008252634
##
## $xmin
## [1] 7
##
## $pars
## [1] 1.952728
##
## $ntail
## [1] 2958
##
## $distance
## [1] "ks"
##
## attr(,"class")
## [1] "estimate_xmin"
```

```
# update the power law object with parameters
m_pl$setXmin(est)
m_pl
```

```
## Reference class object of class "displ"
## Field "xmin":
## [1] 7
## Field "pars":
## [1] 1.952728
## Field "no_pars":
## [1] 1
```

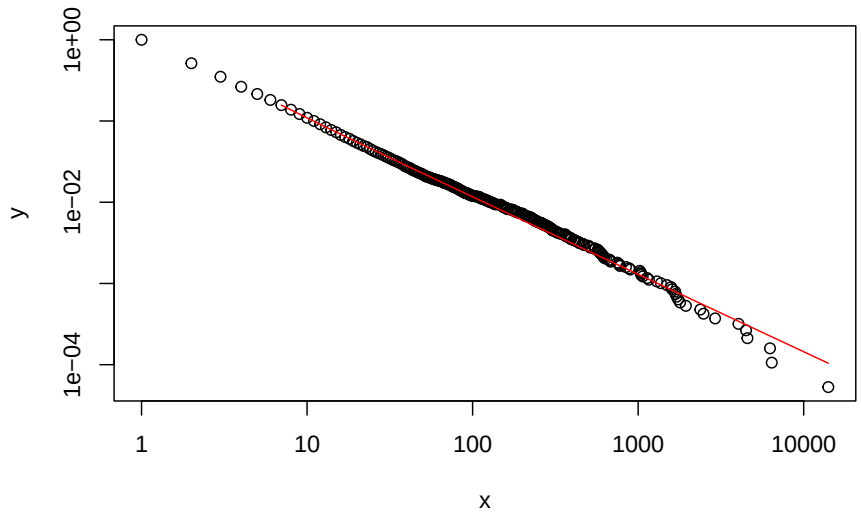
The output looks a bit crazy! Let's try to unpack it. The first output `est` contains the results of fitting the power law distribution to the word frequency data. `$gof` tells us the goodness of fit statistic as estimated from using the Kolmogorov-Smirnov test to measure the distance between the actual distribution and the fitted power law distribution. `$pars` is the alpha exponent of the power law distribution. `$xmin` tells us the cut off point at which the power law up to. You can think of this as the part of the distribution where there is a lot

of jitter and noise (common in real world data) and the power law is not fitted to the data where x is less than that cut off. In the last part of the code, we plug these estimated values back into the power law object; this is to enable us to conduct statistical testing later on.

16.3 Visualization

We can also visualize the fitted power law distribution (in red; notice how the line does not continue beyond $x_{min} < 7$), plotted on log-log scale. The (negative) slope of the best-fit line corresponds to the alpha (exponent) parameter.

```
# visualization
plot(m_pl)
lines(m_pl, col = 'red')
```



16.4 Statistical testing of distribution fit

16.4.1 Method 1: Bootstrapping the data

Because it is possible to fit a power-law distribution to any data set, Clauset et al. strongly recommend that we test whether the observed data set actually

follows a power-law. The method used below follows Clauset et al.'s suggestion to test the null hypothesis (that the data came from a power law distribution) using a goodness-of-fit test with a bootstrapping procedure. The basic concept is to perform a hypothesis test by generating multiple data sets (with the previously estimated parameters x_{min} and alpha) and then "re-inferring" the model parameters. The original model parameter is then compared to the simulated set of parameters. If the p-value is greater than .05, we cannot reject the null hypothesis that the data could be generated from a power law distribution with similar parameters. If the p-value is less than .05, we reject the null hypothesis, and conclude that the data is not convincingly fit by a power law.

```
# parameter uncertainty and statistical testing

## 1000 bootstraps using four cores
bs_p = bootstrap_p(m_pl, no_of_sims = 1000, threads = 4, seed = 1) # can be slow
```

```
## Expected total run time for 1000 sims, using 4 threads is 217 seconds.
```

```
bs_p$bootstraps |> head()
```

```
##           gof xmin     pars ntail
## 1 0.009839299    6 1.917765  3515
## 2 0.007466091   10 1.969352  2095
## 3 0.009205848    9 1.943017  2290
## 4 0.013751881    7 1.942744  2989
## 5 0.004895829    6 1.956929  3469
## 6 0.008659503    7 1.952541  2943
```

```
mean(bs_p$bootstraps[, 2]) # M of xmin
```

```
## [1] 7.668
```

```
mean(bs_p$bootstraps[, 3]) # M of pars (alpha)
```

```
## [1] 1.950474
```

```
sd(bs_p$bootstraps[, 2]) # SD of xmin
```

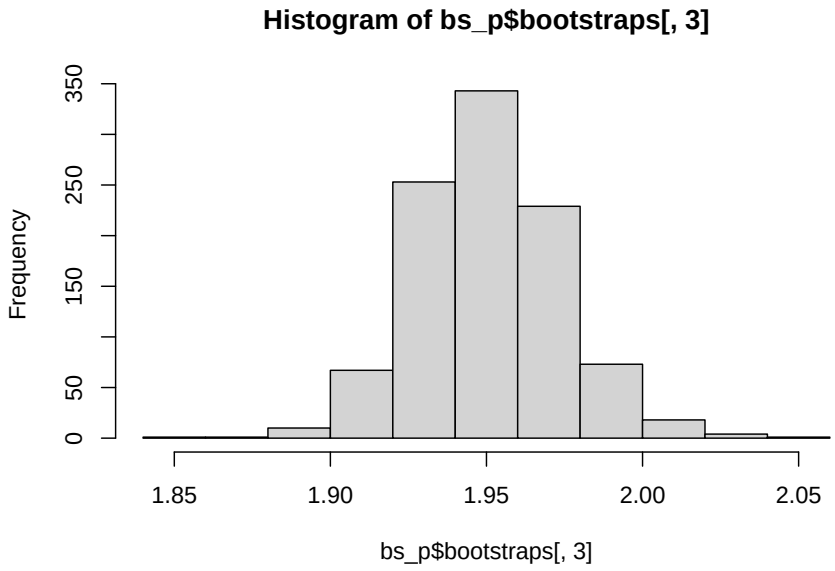
```
## [1] 3.341398
```

```
sd(bs_p$bootstraps[, 3]) # SD of pars (alpha)
```

```
## [1] 0.02341784
```

```
hist(bs_p$bootstraps[, 2])
```


```
hist(bs_p$bootstraps[, 3])
```




```
# p-value
bs_p$p # null hypothesis: distribution came from a power law distribution, since p > 0.05, cannot
```

```
## [1] 0.694
```

16.4.2 Method 2: Comparing alternative distributions

An alternative method is to fit other long-tailed distributions to the data, and then compare these distributions directly using a likelihood ratio test. The code below shows you how to compare the power law distribution to the log-normal distribution (although other long-tailed distributions can also be tested). Basically, there is a similar process of fitting the data to the new distribution and updating the parameters. Then the `compare_distributions` function is used to provide a likelihood ratio test comparing the two distributions. Since the test was not statistically significant, both distributions are equally far from the true distribution and hence it is not possible to determine which is the best fitting model.

```
# make a log-normal object for moby
# fix the xmin parameter to be the same as m_pl
m_ln <- dislnorm$new(moby)
m_ln$setXmin(7)
```

```
# estimate the parameters for log normal distribution
est <- estimate_pars(m_ln)

# update the parameters into the object
m_ln$setPars(est)

# use Vuong's test to compare PL vs LN distributions
comp <- compare_distributions(m_pl, m_ln)

comp$p_two_sided
```

```
## [1] 0.6773366
```

16.5 Exercise

Analyze the degree distribution of the following network:

```
library(igraphdata)

data('UKfaculty')
```

To get a vector of "degree counts" (c.f., word frequencies in Moby Dick), use the following code and see if this degree distribution follows a power law distribution using the bootstrapping approach.

```
degree_frequencies <- table(degree(UKfaculty, mode = 'all'))
```

16.6 References

For an example of how this approach has been used to explore the degree distributions of networks, see:

Siew, C. S. Q., & Vitevitch, M. S. (2020). Investigating the Influence of Inverse Preferential Attachment on Network Development. *Entropy, 22*(9), Article 9. https://doi.org/10.3390/e22091029

Clauset, A., Shalizi, C. R., & Newman, M. E. (2009). Power-law distributions in empirical data. *SIAM Review, 51*(4), 661–703.

Gillespie, C. S. (2015). PoweRlaw: Analysis of heavy tailed distributions. R Package Version 0.30. 0, URL Http://CRAN. R-Project. Org/Package= poweRlaw.