

**INTRODUCTION TO NETWORK ANALYSIS IN R:
A HANDBOOK**

August 12, 2016

Daizaburo Shizuka
School of Biological Sciences
University of Nebraska-Lincoln

Table of Contents

ABOUT THIS HANDBOOK

This handbook contains course material to be used in the NAOC 2016 pre-conference workshop titled “Introduction to Network Analysis in R”.

My goal with this booklet is to provide you with detailed materials that you can take home so that you will have something you can revisit months or years after this workshop. This will hopefully help you retain the material. We may veer off the materials presented here during the workshop, but this booklet should cover almost all topics we end up covering.

Disclaimers:

There are likely some errors or bugs in the codes presented in this handbook. If any such bugs come up, we will address them during the workshop. This booklet is not meant to be cited in publications. Where appropriate, I have done my best to provide references that can be cited.

ABOUT THE AUTHOR:

Dai Shizuka is an Assistant Professor in the School of Biological Sciences at University of Nebraska-Lincoln. He is an evolutionary ecologist working primarily with birds. He especially enjoys thinking about how ecology influences learning and recognition systems as well as social networks, and in turn, how these factors influence evolutionary processes such as coevolution, signal evolution and hybridization/speciation. He teaches courses such as Field Animal Behavior, Ecology and Evolution, Networks in Ecology and Evolution, and Introduction to R.

Dai also maintains a webpage with tutorials on networks analysis in R (among a few other things):

www.shizukalab.com/toolkits

Any comments or suggestions on this page are welcome.

BEFORE YOU START

1. Install Rstudio

2. Install Packages

- 'igraph'
- 'bipartite'
- 'asnipe'
- 'assortnet'
- 'popgraph'
- 'ggplot2'
- 'ggmap'
- 'rnetcarto'
- 'ecodist'
- 'igraphdata'
- 'statnet'
- 'RColorBrewer'

An easy way to do this is to run the following code:

```
install.packages(c('igraph', 'bipartite', 'asnipe', 'assortnet', 'popgraph', 'ggplot2',  
'ggmap', 'rnetcarto', 'ecodist', 'igraphdata', 'statnet', 'RColorBrewer'))
```

3. Download data

- Go to https://dshizuka.github.io/NAOC2016/NAOCworkshop_SampleData.zip and download the compressed data files.
- Save the file into a folder that you can remember.

This will allow us to run some of the examples even if internet access fails.

1. INTRODUCTION

1.1 The R programming language

R is a free, cross-platform, open-source programming language that allows you to do data manipulation, conduct a wide variety of tasks such as data manipulation, data analysis, produce beautiful graphs, put together and run simple models, simulations, etc.

One of the huge advantages is that there is a large community of users within the scientific community producing new and improved ‘packages’, or bundles of functions and datasets that are useful for a particular set of tasks. Anyone can put together an R package. However, there are some guidelines (e.g., there must be a help file with certain level of detailed information). R packages that have been vetted are archived in the Comprehensive R Archive Network (CRAN), and those that are deposited on this server can be downloaded from within the R command console

For the purposes of this workshop, I will assume that you are already familiar with R. However, I try to explain codes in a friendly way so that complete novices should be able to mostly keep up.

Rstudio

There are many ways to run R. You can just run R directly or use software that is designed to provide a computing environment that helps you stay organized and efficient (called ‘integrated development environment’, or IDE). In this workshop, we will use one such software called *Rstudio*. The benefit to Rstudio for me is that it looks the same in Mac and Windows, so it makes it easier to demonstrate codes. It is also very popular so many participants are likely to be familiar with it.

1.2 The igraph Package

As with many other types of analyses, there are a ton of R packages available for network analysis. In this course, we will be primarily using the *igraph* package (Gabor & Nepusz 2006, <http://igraph.org>), which combines ease of use and high-level computation (it is also available for Python and C/C++). This package has proven to be very useful, and it covers a lot of basic network analysis methods as well as plotting capabilities. However, there may be certain situations in which you will want to use other network packages. We will be introducing some of these other packages as we go.

Let’s go ahead and install and load the *igraph* package using the following command:

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

Now that you've installed the package, you want to find out some basic information. Try:

```
library(help='igraph')
```

This should open a new window that gives you a lot of info. The Index shows all of the functions that come as part of the package.

Now we want to try getting information on some of these. However, first, we have to *load the package* before we can begin to use it. Then, try looking up some functions:

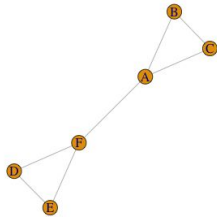
```
library(igraph) #load the package
?graph_from_adjacency_matrix
```

**NOTE: Each time you quit R (or R Studio), your packages will be unloaded. That means that you have to load the package again each time you start the program. The best way to deal with this is to include the command `library(igraph)` at the beginning of your script.*

1.2.1 Our first network!

Let's jump right in and try making a network using the *igraph* package. Here, we will manually make a network using the `make_graph()` function. You likely won't use this function that often, but it is very useful for the purpose of demonstration.

```
library(igraph)
g=make_graph(~A-B-C-A, D-E-F-D, A-F)
plot(g)
```



Congratulations! You made your first network.

So what did we do in the codes above? We used a `make_graph()` function to two make two triangles (A-B-C-A and D-E-F-D) and connect two nodes from those two triangles (A-F). We then used the `plot()` function to make the network figure.

NOTE that your network plot might be oriented differently than this one. Try running this command over and over again. You will notice that the plot will come out slightly differently each time. Think about what changes and what does not change. Does this mean the network changes? We will tackle issues regarding network layout a little bit later.

Now let's learn a little bit about this object, `g`, that we have created.

```
class(g)
```

```
> class(g)
[1] "igraph"
```

You can see that `g` is an 'igraph object'. This means that it is now a *graph* or *network*. You can see more details about this object by simply:

```
g
```

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

As you can see, simply printing `g` will display the bare minimum information: “UN” means “Undirected network with Names of vertices”, and the numbers indicate that there are 6 vertices and 7 edges. The second line shows that there is attribute called 'name'. The characters inside the parentheses (v/c) indicates that it is a 'vertex attribute' and that it is a 'character' attribute (i.e., rather than numeric or integers). The third and fourth lines show us the 7 edges that exist in this network.

We can look up the vertices and edges using this special syntax:

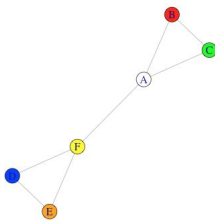
```
V(g) #look up vertices
E(g) #look up edges
```

In addition, we can look up vertex attributes. We currently only have one vertex attribute, called ‘name’:

```
V(g)$name  
> V(g)$name  
[1] "A" "B" "C" "D" "E" "F"
```

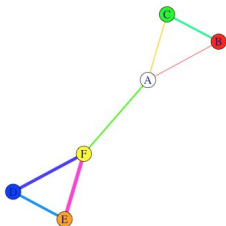
We can also create new vertex attributes using this syntax. Certain attribute names can be directly interpreted by igraph--for example, the vertex attribute ‘color’ will automatically be interpreted for plotting the network. Let’s try this out:

```
V(g)$color=c("white", "red", "green", "blue", "orange", "yellow") #a random set of colors  
plot(g)
```



We can also add edge attributes. Let’s try adding two edge attributes, *width* and *color*.

```
E(g)$width=1:7  
E(g)$color=rainbow(7) #rainbow() function chooses a specified number of colors  
plot(g)
```



You can see that now the edges are slightly different widths--these could represent the variations in the strength of relationships between nodes in a *weighted network*.

1.3 Basic Data Formats

There are three basic data formats that can be used to describe networks: *adjacency matrix*, *edge list*, and *adjacency list*. Each format has its pros and cons. There are other variations on these (e.g., a *biadjacency matrix* for bipartite networks).

1.3.1 Adjacency Matrix

An **adjacency matrix** is a matrix in which the rows and columns represent different *nodes*. In an unweighted adjacency matrix, the *edges* (i.e., lines) are represented by 0 or 1, with 1 indicating that these two nodes are connected. If two nodes are connected, they are said to be adjacent (hence the name, adjacency matrix). In a weighted matrix, however, you can have different values, indicating different edge qualities (or tie strengths).

We can extract the adjacency matrix of the network we created, called **g**:

```
as_adjacency_matrix(g, sparse=F)
> as_adjacency_matrix(g, sparse=F)
  A B C D E F
A 0 1 1 0 0 1
B 1 0 1 0 0 0
C 1 1 0 0 0 0
D 0 0 0 0 1 1
E 0 0 0 1 0 1
F 1 0 0 1 1 0
```

Note the argument `sparse=F` in the code above. This displays the adjacency matrix with 0s. If `sparse=T`, the output is a special format of the matrix where the 0s are replaced with a period (this is to make it easier to see very large matrices).

Also note that, because the network is undirected and unweighted, the corresponding adjacency matrix is symmetrical (value for row A, column B is identical to row B, column A) and binary (values are 0 or 1).

1.3.2 Edge List

An **edge list** is a two-column list of the two nodes that are connected in a network. In the case of a directed network, the convention is that the edge goes from the vertex in the first column to the vertex in the second column. In an undirected network, the order of the vertices don't matter. For weighted networks, you may have a third column that indicates the edge weight.

You can get the edgelist of any igraph object as well:

```
as_edgelist(g)
  [,1] [,2]
[1,] "A"  "B"
[2,] "A"  "C"
[3,] "A"  "F"
[4,] "B"  "C"
[5,] "D"  "E"
[6,] "D"  "F"
[7,] "E"  "F"
```


1.3.3 Adjacency List

An **adjacency list**, also known as a node list, presents the 'focal' node on the first column, and then all the other nodes that are connected to it (i.e., adjacent to it) as columns to the right of it. In a spreadsheet, would be a table with rows with different number of columns, which is often very awkward to deal with, like this:

Focal Node	Neighbor 1	Neighbor 2	Neighbor 3
A	B	C	F
B	A	C	
C	A	B	
D	E	F	
E	D	F	
F	A	D	E

In R, you can display an adjacency list as an actual 'list object', with each item representing neighbors of each focal node:

```
as_adj_list(g)
> as_adj_list(g)
$A
+ 3/6 vertices, named:
[1] B C F
$B
+ 2/6 vertices, named:
[1] A C
$C
+ 2/6 vertices, named:
[1] A B
...
```

1.3.4 Data formats for directed and weighted networks

Let's consider some important aspects of data formats that come with networks that are directed or weighted. I will keep this short by listing some important things to consider, and a line of code that will display this.

Directed networks

Recall that we have previously created an igraph object for a *directed network* called **dir.g**.

For directed networks, the adjacency matrix is not symmetrical. Rather, the cell value is 1 if the edge goes from the *row vertex* to the *column vertex*.

```
as_adjacency_matrix(dir.g, sparse=F)
> as_adjacency_matrix(dir.g, sparse=F)
  A B C D E F
A 0 1 0 0 0 1
B 0 0 1 0 0 0
C 1 0 0 0 0 0
D 0 0 0 0 1 0
E 0 0 0 0 0 1
F 1 0 0 1 0 0
```

For directed networks with mutual edges (represented by double-edged arrows), the edge list lists both directions separately:

```
as_edgelist(dir.g)
> as_edgelist(dir.g)
  [,1] [,2]
[1,] "A" "B"
[2,] "A" "F"
[3,] "B" "C"
[4,] "C" "A"
[5,] "D" "E"
[6,] "E" "F"
[7,] "F" "A"
[8,] "F" "D"
```

You can see that, since the `dir.g` network object contains one mutual edge ($A \leftrightarrow F$), the edge list has 8 rows, while the edgelist for the undirected version of the network has 7 rows.

Weighted networks

Let's now consider what the data formats would look like. To do this, let's go back to our original network, `g`. Let's say that the edge widths that we added represent edge weights or values. Then, the adjacency matrix for this network can be shown by using the `attr=` argument within the function to call the adjacency matrix to specify the edge weights:

```
as_adjacency_matrix(g, sparse=F, attr="width")
  A B C D E F
A 0 1 2 0 0 3
B 1 0 4 0 0 0
C 2 4 0 0 0 0
D 0 0 0 0 5 6
E 0 0 0 5 0 7
F 3 0 0 6 7 0
```

You can display the edge weights as an edgelist as well. In fact, `igraph` has a convenient function that will display all of the edge attributes together as a data frame:

```
as_data_frame(g)
  from to width  color
1   A  B     1 #FF0000FF
2   A  C     2 #FFDB00FF
3   A  F     3 #49FF00FF
4   B  C     4 #00FF92FF
5   D  E     5 #0092FFFF
6   D  F     6 #4900FFFF
7   E  F     7 #FF00DBFF
```

Recall that in undirected networks, the “from” and “to” designation are arbitrary (it is simply organized in alphabetical order here).

If you want to show an edge list as a three-column matrix with the two nodes and edge weights only, you can just specify which edge attribute you want to use as the edge weight, e.g. (output not shown):

```
as_data_frame(g)[,c("from", "to", "width")]
```

1.4 Creating a Network From a Dataset

In this section, we will provide an overview of how to take a dataset and generate a network. Typically, we would start from a .csv or .txt file that is organized as an adjacency matrix or edge list (see above).

Super quick R tutorial: Three ways to import .csv files

(1) A common method for reading data from a .csv file is simply:

```
#for .csv files
dat=read.csv("completepathname.csv", header=T)
```

This script creates a dataframe called "dat", which will contain your data. header=T indicates that the first row is a header row (though I think this is default). On Macs, the easiest way to do this is to find the file in your Finder Window, and then drag the file into your R Editor. This displays the file path in the Editor.

(2) The second method is to set a working directory to the folder where your file resides.

```
setwd("path-to-folder-here")
dat=read.csv(file="SampleData_1.csv",header=T)
```

This is similar to the first method. It can be particularly useful if you want to batch-process a bunch of files.

(3) The final method to import data from a file is:

```
dat=read.csv(file.choose(), header=T)
```

This will call a prompt that will let you choose the file. You can then find and choose the file you want to import.

1.4.1 Creating a network from your edge list

Creating a network from an edgelist that you have created is easy. First, import the .csv file called "sample_edgelist.csv".

```
edge.dat=read.csv("https://dshizuka.github.io/NAOC2016/Sample_edgelist.csv")
edge.dat
```

```
>edge.dat=read.csv("https://dshizuka.github.io/NAOC2016/Sample_edgelist.csv")
> edge.dat
```

	V1	V2	weight
1	Adam	Betty	1
2	Adam	Daniel	1
3	Adam	Frank	4
4	Betty	Charles	3
5	Betty	Daniel	1
6	Betty	Frank	2
7	Daniel	Esther	1
8	Daniel	Frank	1
9	Esther	Frank	1
10	Frank	Gina	2

So this data frame has three columns: the first two columns are the edge list, and the third column is an edge value we called "weight". If we have the data organized this way, we can simply use a function called graph.data.frame() to create a network we will call eg (output not shown).

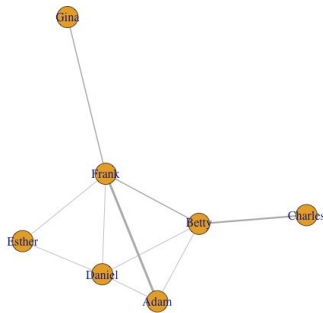
```
eg=graph_from_data_frame(edge.dat, directed=FALSE)
eg
plot(eg, edge.width=E(eg)$weight)
```

```

> eg
IGRAPH UNW- 7 10 --
+ attr: name (v/c), weight (e/n)
+ edges (vertex names):
[1] Adam --Betty    Adam --Daniel  Adam --Frank   Betty --Charles Betty --Daniel
Betty --Frank
[7] Daniel--Esther Daniel--Frank Esther--Frank   Frank --Gina

```

This has created an undirected, weighted network with 7 nodes and 10 edges. There is one edge attribute called “weight”. The plot function uses this edge weight as the edge width.



1.4.2 Creating a network from your adjacency matrix

Importing an adjacency matrix written in .csv format is just slightly trickier. This is because you want R to know that first row is a header AND the first row contains row names rather than data. You also want R to recognize this data as a “matrix object”. We can use just one line of code to do this:

```

am=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016/Sample_adjmatrix.csv", header=T,
row.names=1))
am

```

```

> am
      Adam Betty Charles Daniel Esther Frank Gina
Adam      0     1      0      1      0      4      0
Betty      1     0      3      1      0      2      0
Charles    0     3      0      0      0      0      0
Daniel     1     1      0      0      1      1      0
Esther     0     0      0      1      0      1      0
Frank      4     2      0      1      1      0      2
Gina       0     0      0      0      0      2      0

```

Now we have our adjacency matrix, and we are ready to convert this into an igraph object! Note that this is a *weighted* adjacency matrix. Note that we are going to add an argument **weighted=T** to indicate that the edges have weights. The results from the code below should look the same as above.

```

g=graph_from_adjacency_matrix(am, mode="undirected", weighted=T)
g
plot(g, edge.width=E(g)$weight)

```

1.5 Network Plots: Layouts and Attributes

Researchers often use network analysis as a tool for displaying complex data. But you should also know that one must be very careful not to over-interpret network plots. This is because there is no single rule for how to display networks, and network plots can obscure interesting patterns or be misleading if the layout format does not match the purpose. With that said, network plots can be a useful and powerful way to get your message across if used appropriately.

The plotting function in *igraph* comes with a lot of different options. We have used several arguments already, but if you want to explore the ins and outs of plotting in R, look up `?igraph.plotting`. I also highly recommend my favorite web tutorial on static and dynamic network visualization by Katya Ognyanova (<http://kateto.net/network-visualization>).

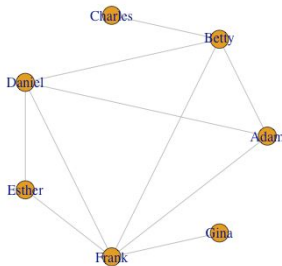
Here, we will deal with two main topics: (1) layout functions and how to merge network data and (2) attribute data to illustrate the network.

1.5.1 Layout

There are several graph layout functions in *igraph*. You can use these layout functions inside the `plot()` function, or you can specify the layout as a separate two-column matrix. For example, let's say you want to arrange the network as a circle:

```
#option 1
plot(g, layout=layout_in_circle(g))

#option 2:
l=layout_in_circle(g)
plot(g, layout=l)
```



So, the `layout_in_circle()` function can be used to generate an object (*l*) that can be used to determine the placements of vertices. Let's now see what the *l* object that we just created looks like:

```
class(l)
1

> class(l)
[1] "matrix"
> l
      [,1]      [,2]
[1,] 1.0000000 0.0000000
[2,] 0.6234898 0.7818315
[3,] -0.2225209 0.9749279
[4,] -0.9009689 0.4338837
[5,] -0.9009689 -0.4338837
[6,] -0.2225209 -0.9749279
[7,] 0.6234898 -0.7818315
```

You can see that the object *l* is a two-column matrix of x- and y-coordinates. So this is what the layout functions create, and the plotting functions use these coordinates to place the nodes. Now let's learn a little bit about the different types of layout functions.

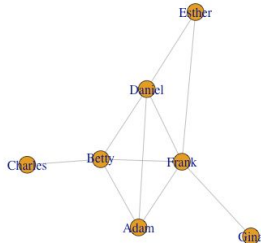
Force-directed layouts

Networks are generally plotted using a **force-directed algorithm**, which is a class of algorithms for drawing graphs. The default function in *igraph* is called the Fruchterman-Reingold (1991) algorithm. Typically, force-directed algorithms use a physical simulation where some kind of attractive force (imagine a spring) are used to attract nodes connected by edges together. So 'tightly' connected clusters of nodes will show up close to each other, and those that are 'loosely' connected will be repulsed towards the outside. However, the algorithm does not specify where any node has to be other than these constraints. Therefore, **every time you run the `plot()` command, you will get a slightly different configuration of the graph**. Let's try plotting the network with the default Fruchterman-Reingold algorithm. Run this line of code several times:

```
plot(g, layout=layout_with_fr(g))
```

You will observe that each time you plot the graph, you get a different configuration. They all follow the same algorithm, but they turn out differently because the algorithm is simulating a stochastic process of forces acting on nodes. In order to create reproducible layouts, you can use the `set.seed()` function when setting up the layout.

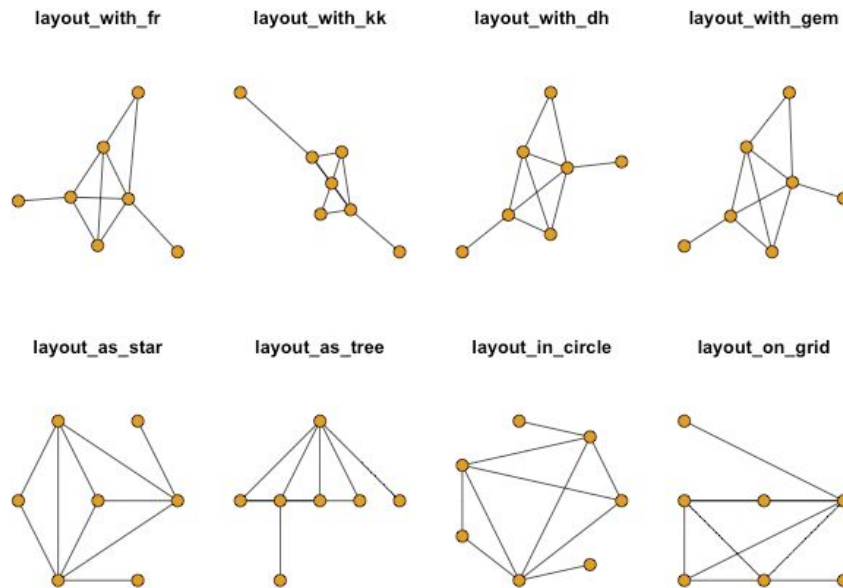
```
set.seed(10)
l=layout_with_fr(g)
plot(g, layout=l)
```



Now, your plot should be reproducible when you run the plotting function again.

Other popular force-directed algorithms include Kamada & Kawai (1989) and Davidson & Harel (1996), and “GEM” (Frick et al. 1995). They all have slightly different properties and there is no single ‘best method’. Below, we will plot the same network in 8 different ways. The top row will be force-directed methods, and the bottom row will be static methods:

```
set.seed(10)
layouts = c("layout_with_fr", "layout_with_kk", "layout_with_dh", "layout_with_gem",
"layout_as_star", "layout_as_tree", "layout_in_circle", "layout_on_grid")
par(mfrow=c(2,4), mar=c(1,1,1,1))
for(layout in layouts){
  l=do.call(layout, list(g))
  plot(g, layout=l, edge.color="black", vertex.label="", main=layout)
}
```

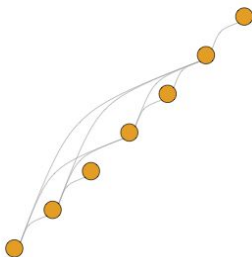


Before we move on, let's clear the plotting region:

```
dev.off()
```

You can also use custom layouts. I'll set up the vertices on a line and then used curved edges.

```
l=matrix(c(1,2,3,4,5,6,7, 1,2,3,4,5,6,7),ncol=2)
plot(g,layout=l,edge.curved=TRUE, vertex.label="")
```



The ability to control the layout is particularly useful when we want to plot spatial networks, as we will see later.

1.5.2 Adding vertex attributes

Now that we know a bit about how to change the looks of vertices and edges, we are ready to start adding information to our edges. For example, you might have other information about your vertices that you'd like to display in the graph plot--e.g., traits of individuals in a social network, or the attributes of a species in a network of species interactions. Typically, you would store these "node attributes" in a separate file. We will use a sample "attributes file" called `sample_attrib.csv`. Let's go ahead and import this file:

```
attrib=read.csv("https://dshizuka.github.io/NAOC2016/Sample_attrib.csv")
attrib
```

```
> attrib
      Name Sex Age
1    Adam   M  30
2    Gina   F   8
3 Charles   M  35
4  Daniel   M  15
5   Frank   M  10
6   Betty   F  28
7  Esther   F  12
```

Now, the goal is to incorporate these vertex attributes, Sex and Age, to the graph object. Note that I have made this data NOT alphabetized on purpose. Therefore, the vertex number and the row numbers of this attributes file do not line up correctly. So we have to first sort these attributes so that we assign the correct attribute to the correct vertex. We can do this by using the `match()` function.

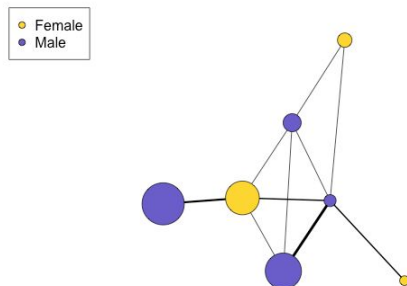
```
V(g)$sex=factor(attrib[match(V(g)$name, attrib$Name), "Sex"]) # factor() preserves data as M/F
V(g)$age=attrib[match(V(g)$name, attrib$Name), "Age"]
```

The trick now is to assign colors to each sex. Here, let's assign males and females to different colors. Here's one way to do it:

```
V(g)$color=c("gold", "slateblue")[as.numeric(V(g)$sex)]
```

This line of code converts the sex to numbers by alphabetical order, i.e., 1 = female and 2 = male. Now let's plot the network with node colors as sex and node size as age and add a legend for the node colors:

```
set.seed(10)
l=layout_with_fr(g)
plot(g, layout=l, vertex.label="", vertex.size=V(g)$age, edge.width=E(g)$weight,
     edge.color="black")
legend("topleft", legend=c("Female", "Male"), pch=21, pt.bg=c("gold", "slateblue"))
```



2. AN ECOLOGIST'S GUIDE TO DIFFERENT NETWORK TYPES

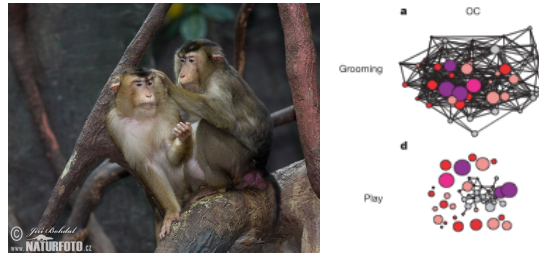
There are dozens of different types of biological networks. I will focus on a few main types of networks here. For each network, it is particularly important to think about how nodes and edges are defined.

We will present three general types of networks that have been of particular interest for ecologists, each with particular challenges. After a brief overview, I highlight a few particular topics for each type of network.

2.1 Overview of some different types of networks

2.1.1. Social networks

Social networks describe the dynamics of social interactions and associations (edges) between individuals (nodes). Examples of the types of interactions/associations involved include direct interactions (e.g., aggression, grooming), co-membership in groups such as herds and flocks, or co-use of spatial locations (e.g., roosts, refuges). The analysis of such social networks can be used to understand the structure of societies, the dynamics of social structure across time and space, and the flow of information/disease/etc through these connections.

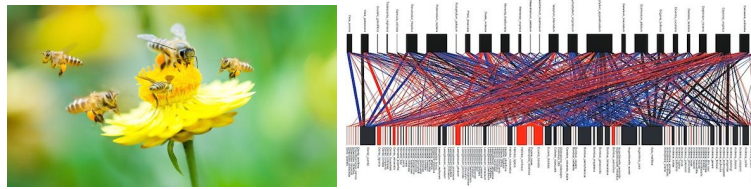


*Example of a social network: Grooming and play networks of pigtail macaques (*Macaca nemestrina*), from Flack et al. (2006).*

Social network analysis has a long history in fields such as sociology, where it dates back to at least the 1930s. Among ecologists, social networks have been used by primatologists for many decades. However, the study of animal social networks has undeniably exploded in the past decade. This is a field with many review articles (Krause et al. 2007, Wey et al. 2008, Sih et al. 2009, Pinter-Wollman et al. 2014) and books (Croft et al. 2008, Whitehead 2008, Krause et al. 2015) which are recommended for those who are interested in learning more.

2.1.2. Ecological networks

In many ecological networks, nodes represent species and edges represent some sort of species interaction. Two prominent examples include *food webs* and *mutualism networks*. In food webs, the edges represent trophic interactions. In mutualism networks, the edges may represent interactions such as plant-pollinator relations. In mutualism networks in particular, the network may be **bipartite**--that is, there are two types of nodes (e.g., plants and pollinators), and interactions only occur between two different types of nodes.

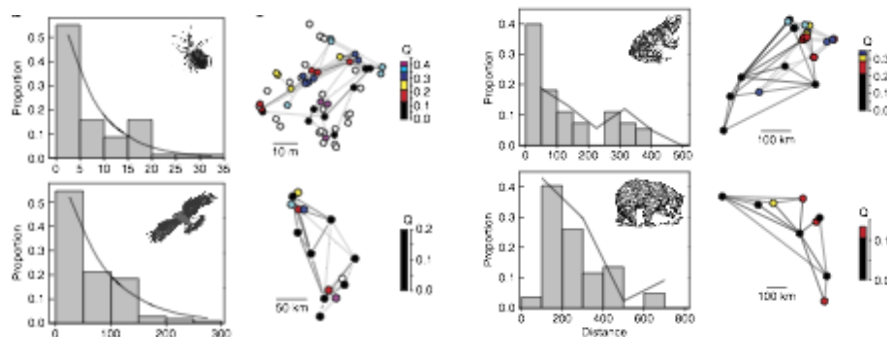


Bee-plant interaction network, from Burkle et al. (2013)

The network view of ecology has a long history (e.g., May 1973), and many great reviews/perspective pieces (Montoya et al. 2006; Bascompte & Jordano 2007; Ings et al. 2009) and books (e.g., Pacual & Dunne 2006) now exist. We will not even scratch the surface of the advances in this area over the past couple of decades!

2.1.3 Spatial networks

Spatial networks are distinguished by the fact that nodes represent some point in space, e.g., habitats within a landscape, or populations across a region. The edges may represent connectivity of these sites or populations, e.g., predicted or actual movements of individuals or measures of genetic connectivity. Defining edges in these type of networks tend to be a challenge because of the inherent difficulties in observing movements, especially since rare dispersal events can have important impacts in ecological patterns and evolutionary processes.



Examples of spatial networks for four different species, each displaying connectivity of populations using movement or genetic data. From Fletcher et al. (2013).

Using network analysis to assess landscape connectivity was first proposed by Urban and Keitt (2001), and has since exploded in the context of conservation (reviews and perspectives by Bunn et al. 2000; McRae et al. 2008; Urban et al. 2008). Dyer & Nelson (2004) and Dyer (2015) provide a good framework for using population genetics to define spatial networks of connectivity. Fall et al. (2007) and Dale & Fortin (2010) are also great references for general challenges in considering spatial networks. Some empirical examples include Fletcher et al. (2011; 2013), Ribeiro et al. (2009),

2.2 Social Networks From Group Associations

Association networks are created by connecting nodes that belong to the same ‘group’. For example, individuals that belong to the same club may be considered to be ‘associates’ and therefore connected in a network. Similarly, in animal social networks, individuals that are observed to be in the same flock or herd may be considered to be associated in some way. Keep in mind though that this is an *assumption*, and the inference of social networks from patterns of group association is often called the ‘gambit of the group’ (Franks et al. 2010).

Typically, association data would be first gathered as an individual-by-group matrix (or group-by-individual matrix). That is, individuals are listed in rows and groups (e.g., flocks/herds/schools) listed in columns (or vice-versa). The cell value is 1 if that individual is in that group, and 0 if not.

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

Let’s play with a sample dataset (*Sample_association.csv*), which is a snippet of real data from a study on Golden-crowned Sparrows. My colleagues and I collected some data on which individually-marked sparrow was observed together in flocks (typically 2-10 birds), defined as a collection of individuals within a 5m radius during a given observation (Shizuka et al. 2014). Flock membership changes within minutes—some birds leave and other birds join. Here, each individual is presented in the rows, and each flock is presented in columns (‘individual-by-group matrix’). The cell value is 1 if that individual was seen in that flock, and 0 if not.. Let’s take a look at a slice of the data (output not shown):

```
assoc=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016//Sample_association.csv",
header=T, row.names=1))
assoc
```

Using an association index

However, note that in this type of data, the probability that we draw a connection between any two nodes is partially dependent on *how often we saw each individual*. For example, if two individuals are simply observed more often, they are more likely to be observed together. Conversely, if an individual is rarely seen, it is likely to be seen only with a small subset of other individuals. We may like to take this into account. One popular solution to this is to define network edges based on an **association index** that accounts for the frequency of observation of each node. One simple association index is called the Simple Ratio Index (Cairns & Schwager 1987; this is the same as the Jaccard Index).

$$\text{Association Index} = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

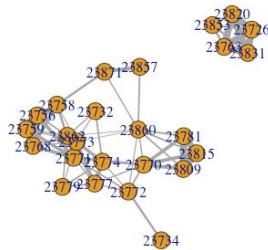
where

- $|A \cap B|$ is the number of times A and B were seen together
- $|A|$ is the total number of times A was seen (together or separate from B)
- $|B|$ is the total number of times B was seen (together or separate from A)

Construct social network from associations using the ‘asnipe’ package

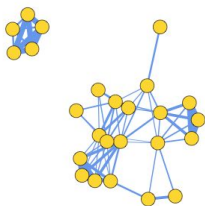
Damien Farine (<http://collectivebehaviour.com/farine-lab/>) has created a very useful R package called ‘asnipe’, which is tailored for the analysis of animal social network. We will use the `get_network()` function in this handy package to convert our association matrix into a social network.

```
library(asnipe) #load the asnipe package
gbi=t(assoc) #assoc is in individual-by-group format, but this package likes
group-by-individual, so we will transpose the matrix.
adj.m=get_network(t(assoc), association_index="SRI") #this function converts the association
matrix into an adjacency matrix using the Simple Ratio Index.
assoc.g=graph_from_adjacency_matrix(m, "undirected", weighted=T) #create a graph object
plot(g, edge.width=E(g)$weight*10) #Since the edge weights are small (values = 0 to 1),
let's multiply by 10 for edge widths.
```



For fun, let's make it a little bit prettier:

```
plot(g, edge.width=E(g)$weight*10, vertex.label="", vertex.color="gold",
edge.color="cornflowerblue", vertex.frame.color="black")
```



The Do-it-yourself version:

We can convert the association matrix into an adjacency matrix manually in a several steps:

#step 1: use matrix multiplication to convert affiliations into an adjacency matrix of the raw number of times a pair was seen together (note: this is actually just the bipartite projection).

```
m=assoc%*%t(assoc)
```

#step 2: use `rowSums()` to calculate the raw number of times each individual was seen total

```
seen=rowSums(assoc)
```

#step 3: use the `sapply()` function to add up the total number of times individual i and j were seen (together or separate). NOTE: This will "double-count" the number of times the two individuals were seen together. This is "A + B" in the above equation

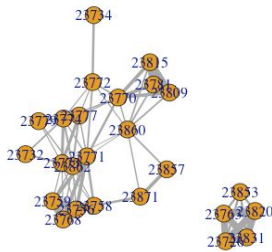
```
bothseen=sapply(seen, function(x) x+seen)
```

```
#step 4: calculate the association index. Remember, since "bothseen" double-counts the
number of times both individuals i and j were seen together, we will the number of times A
and B were seen together from the denominator
ai_adjacency=m/(bothseen-m)
```

```
#step 5: set diagonal to 0
diag(ai_adjacency)=0
```

Now use this weighted adjacency matrix to make the network of flock associations.

```
assoc_net=graph_from_adjacency_matrix(ai_adjacency, "undirected", weighted=T, diag=F)
plot(assoc_net, edge.width=E(assoc_net)$weight*10)
```



2.3 Working with Bipartite Networks

2.3.1 Intro to bipartite networks

Bipartite network, or two-mode network, is a special type of network in which there are two kinds of nodes, and only nodes of different type are connected. In this case, the data is often organized as a 'biadjacency matrix', in which rows and columns are two different types of nodes. For example, let's take a hypothetical dataset on plant-pollinator interactions. The following interaction matrix describes which animal pollinates which plant--cell value is 1 if there is an interaction and 0 if not.

	Hummingbird	Bat	Moth	Butterfly	Bee
Plant 1	1	1	0	1	0
Plant 2	1	0	0	1	0
Plant 3	0	1	1	1	0
Plant 4	0	0	0	1	1

You can see that this can be expressed as as bipartite network in which plant species are one type of node (black squares) and pollinator species are another type of node (white circles). These types of networks are called **mutualism networks**, and constitute one type of ecological network. A two-trophic **food web** can be considered a bipartite network, although food webs in general can take more complex forms (multiple trophic levels, intraguild predation, etc.). Intuitively **sexual networks**, i.e., a network of social or genetic mates, can also be considered a bipartite network consisting of male and female nodes.

2.3.2 Working with bipartite networks in R

Let's now try playing around with this type of data in R. We will start by using the igraph package. First, we will read the sample data into R. The file Sample_bipartite.csv is the plant-pollinator interaction matrix as presented above:

```

bmat=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016/Sample_bipartite.csv",
header=T, row.names=1))
bmat

> bmat
      hummingbird bat moth butterfly bee
Plant_1          1  1  0          1  0
Plant_2          1  0  0          1  0
Plant_3          0  1  1          1  0
Plant_4          0  0  0          1  1

```

To read this interaction matrix into an *igraph* object called *bg*, we will use the function `graph_from_incidence_matrix()`.

```

bg=graph_from_incidence_matrix(bmat)
bg

> bg
IGRAPH UN-B 9 10 --
+ attr: type (v/l), name (v/c)
+ edges (vertex names):
 [1] Plant_1--hummingbird Plant_1--bat          Plant_1--butterfly
Plant_2--hummingbird Plant_2--butterfly    Plant_3--bat          Plant_3--moth
 [8] Plant_3--butterfly    Plant_4--butterfly    Plant_4--bee

```

Here, the first line of information for object *bg* tells us that we have an Undirected, Named network that is Bipartite, and it has 9 nodes and 10 edges. The second line of information tells that there are two vertex attributes: vertex type and vertex name. The rest of the lines show us the edges that exist.

Now let's take a moment to learn how to deal with the vertex attribute called "type". You will see that this attribute is coded as TRUE/FALSE, but you can convert this into numbers (output not shown).

```

V(bg)$type #Display the vertex types. They are "FALSE" or "TRUE"
V(bg)$type+1 #If you add 1 to FALSE/TRUE vector in R, they convert to 1 and 2, respectively.

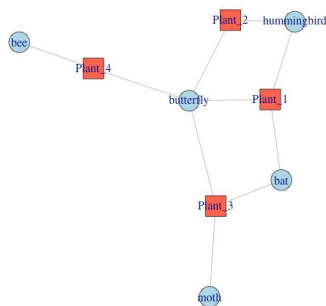
```

Then, you can use this to assign vertex shapes and colors.

```

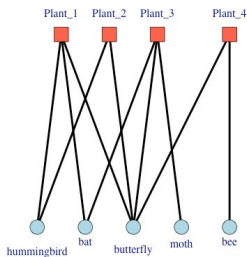
V(bg)$shape=c("square","circle")[V(bg)$type+1] # you can assign vertex shapes this way
V(bg)$color=c("tomato","lightblue")[V(bg)$type+1] # assign colors the same way
plot(bg) #now plot

```



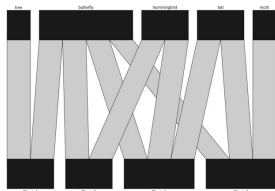
igraph also includes a 'bipartite layout', in which the two node types are arranged in two rows.

```
plot(bg, layout=layout_as_bipartite(bg), edge.color="black",
edge.width=4, vertex.label.dist=1, vertex.label.degree=c(rep(-pi/2, 4), rep(pi/2, 5)))
```



The ‘bipartite’ package also has plotting function that produces a prettier version of this:

```
library(bipartite)
plotweb(bmat) #the base function for bipartite network plot
```

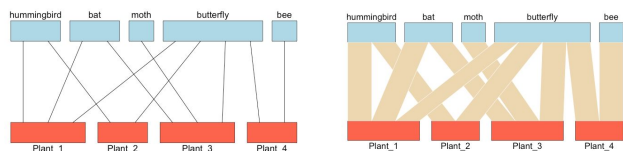


Let’s make this prettier:

```
plotweb(bmat, method="normal", ybig=0.1, col.interaction="gray", arrow="both",
col.high="lightblue", col.low="tomato", labsize=2, adj.low=c(0.5,0.8), adj.high=c(0.5,0))
```

or,

```
plotweb(bmat, method="normal", ybig=0.1, col.interaction="gray",
bor.col.interaction="wheat2", arrow="no", col.high="lightblue", col.low="tomato", labsize=2,
adj.low=c(0.5,0.8), adj.high=c(0.5,0))
```



2.3.3 Bipartite projections

Sometimes, you are only interested in the relationships between one type of node: e.g., Which pollinators compete over the same nectar sources? Which males compete over the same females? In such cases, we are interested in what is called a *one-mode projection of the bipartite network*.

For example, Hummingbird, Bat and Butterfly would be connected because they all pollinate Plant 1, whereas Bee and Butterfly would be connected because they are both pollinate Plant 4; so on and so forth. This is called a “single-mode projection” of the bipartite matrix.

Note that we can make bipartite projections in two directions: we can make: (1) one-mode network of plants by shared pollinators, or (2) one-mode network of pollinators by shared plants. We can make both of these using a single function in igraph called `bipartite_projection()`.

```
proj=bipartite_projection(bg, multiplicity = T)
proj

> proj
$proj1
IGRAPH UNW- 4 6 --
+ attr: name (v/c), shape (v/c), color (v/c), weight (e/n)
+ edges (vertex names):
[1] Plant_1--Plant_2 Plant_1--Plant_3 Plant_1--Plant_4 Plant_2--Plant_3
Plant_2--Plant_4 Plant_3--Plant_4

$proj2
IGRAPH UNW- 5 6 --
+ attr: name (v/c), shape (v/c), color (v/c), weight (e/n)
+ edges (vertex names):
[1] hummingbird--bat          hummingbird--butterfly bat          --butterfly bat
--moth          moth          --butterfly butterfly --bee
```

This new object, called `proj`, is in a format called *list* [you can see this by running `class(proj)`]. A list is an object that has multiple components—in this case, `proj` is a list with two distinct graphs: (1) `proj$proj1` (or alternatively, `proj[[1]]`) is a network with 4 nodes and 6 edges. (2) `proj$proj2` (or alternatively, `proj[[2]]`) is also a network with 5 nodes and 6 edges. You'll get a better sense of these projections if you plot them:

```
plot(proj$proj1, edge.color="black", edge.width=E(proj$proj1)$weight^3)
plot(proj$proj2, edge.color="black", edge.width=E(proj$proj2)$weight^3)
```



2.3.4 What is a bipartite projection anyway?

Although I've presented the bipartite projection using a canned function in igraph, so let's explain it briefly here. Mathematically, the adjacency matrices of the bipartite projection is *the matrix product of the interaction matrix and its transpose*. That is:

$$\text{Projection 1} = A \otimes A^T$$

$$\text{Projection 2} = A^T \otimes A$$

We can calculate this in R:

```
proj1=bmat%*%t(bmat) # %*% = matrix multiplication and t() = transpose of the matrix
proj2=t(bmat)%*%bmat
proj1
proj2
```



```
> proj1
      Plant_1 Plant_2 Plant_3 Plant_4
Plant_1      3      2      2      1
Plant_2      2      2      1      1
Plant_3      2      1      3      1
Plant_4      1      1      1      2

> proj2
      hummingbird bat moth butterfly bee
hummingbird      2   1   0         2   0
bat              1   2   1         2   0
moth             0   1   1         1   0
butterfly        2   2   1         4   1
bee              0   0   0         1   1
```

Here, one thing to note is that the diagonal of the matrix represents the number of times each plant or pollinator interacts. We may want to ignore this when plotting the networks. These codes should produce the same as the plots above (outputs not shown):

```
pg1=graph_from_adjacency_matrix(proj1, mode="undirected",diag=F, weighted=T) #diag=F ignores
the diagonal of the matrix
pg2=graph_from_adjacency_matrix(proj2, mode="undirected",diag=F, weighted=T)

plot(pg1, vertex.color="tomato", vertex.shape="square",
edge.color="black",edge.width=E(pg1)$weight^3)

plot(pg2, vertex.color="lightblue", vertex.shape="circle",
edge.color="black",edge.width=E(pg1)$weight^3)
```

2.4 Visualizing Spatial Networks

With spatial networks, the nodes represent spatial locations and the edges represent connections between those locations, e.g., *habitat connectivity* ('landscape connectivity networks': Urban & Keitt 2001, Urban et al. 2009), *movements of individuals* ('movement networks': Jacoby & Freeman 2016), *genetic covariance among populations* ('population graphs': Dyer & Nason 2004, Dyer 2015), *migration pathways* ('migratory flow networks': Taylor & Norris 2010). Such networks of connectivity between populations are important in a variety of fields including conservation biology, population genetics, evolution and metapopulation ecology.

To explore this type of data, we will be demonstrating the construction of space-based networks using a dataset on genetic connectivity ('population graph') of the cactus *Lophocereus schottii* from Dyer & Nason (2004). We will be using an R package called 'popgraph', written by Rodney Dyer. A nice tutorial is available online (<http://dyerlab.github.io/popgraph/>), and the codes here are mostly adapted from this site.



Lophocereus schottii

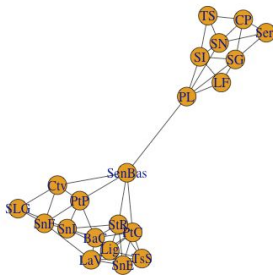
First, we will import an adjacency matrix of genetic connectivity among sites in *L. schottii* (Dyer & Nason 2004). Nason et al. (2002) assayed 29 polymorphic allozyme loci from 948 individual cacti in 21 populations. This data was converted to a 21 x 21 genetic distance matrix. Then the pairs of populations that were (using a method called edge exclusion deviance--see Dyer and Nason 2004 for details). In the resulting network (i.e., ‘population graph’), sites are connected by an edge if they are genetically ‘significantly similar’ (i.e., there is gene flow).

Let’s read the data and take a look (output not shown here):

```
lopnet=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016/lopho_network.csv", header=T,
row.names=1)) #From Dyer et al. 2004
lopnet
```

Now, let’s convert this into a graph object and plot it:

```
g=graph_from_adjacency_matrix(lopnet, mode="undirected")
plot(g)
```



Again, in this network, each node represents a population and the edges connect populations that are significantly similar genetically.

2.4.1 Using spatial data to plot networks

Now, let’s now display this network using the actual locations of the nodes. To do this, we will read in a second dataset that includes the latitude/longitude coordinates of each node as well as a “region” variable indicating whether the population is in Sonora or Baja state of Mexico. This data is from Table 1 in Nason et al. (2002).

```
locations=read.csv("https://dshizuka.github.io/NAOC2016/lopho_locations.csv")
head(locations)
```

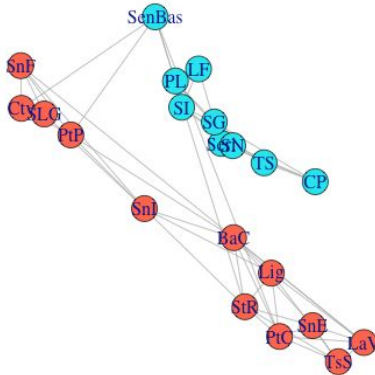
```
> head(locations)
  Population Region      Lat      Long
1          CP Sonora 27.9461 -110.6594
2          TS Sonora 28.4050 -111.3658
3          SN Sonora 28.8211 -111.7992
4        Seri Sonora 28.8761 -111.9550
5          SG Sonora 29.3950 -112.0536
6          SI Sonora 29.7539 -112.5050
```

We can use these coordinates to plot the spatial network. We won’t worry about specific projection methods for converting coordinates to 2D. We can do this by creating a 2-column matrix of x- and y-coordinates and using this as the ‘layout’, which we can specify when plotting the network. While we’re at it, we will also color-code the nodes based on region:

```
#create a two-column matrix of x- and y-coordinates
V(g)$x=locations[match(V(g)$name, locations$Population),"Long"]
V(g)$y=locations[match(V(g)$name, locations$Population),"Lat"]
l=matrix(c(V(g)$x, V(g)$y), ncol=2)

# color-code nodes by region
regions=locations[match(V(g)$name, locations$Population),"Region"]
V(g)$color=c("tomato", "turquoise2")[as.numeric(regions)]

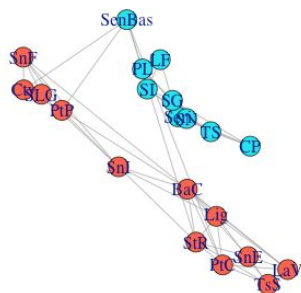
plot(g, layout=1)
```



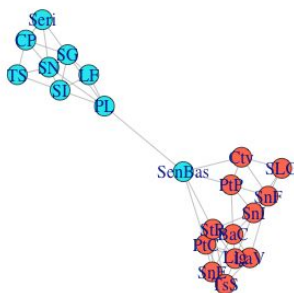
Now, let's compare what the networks look like when using this 'spatial layout', compared to a force-directed layout (Fruchterman-Reingold), which is the default layout in igraph.

```
par(mfrow=c(1,2), mar=c(2,2,2,2))
plot(g, layout=1, main="Spatial Layout")
plot(g, layout=layout_with_fr(g), main="Force-directed layout")
```

Spatial Layout



Force-directed layout



Comparing these network layouts, one thing pops out: one population ("SenBas") is located in Sonora, but it clusters genetically with the Baja California populations. Geographically, it doesn't seem to be particularly closer to the Baja populations, so what gives? It would be nice to be able to see the underlying geographic features here...

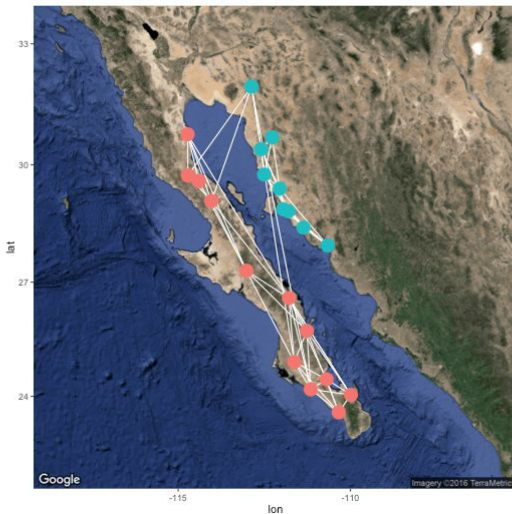
2.4.2 Making it a lot prettier using popgraph

Now, let's make this spatial network much nicer using the 'popgraph' package. This will allow us to make a figure that overlays this network on top of a satellite image. Beware: this package uses the package 'ggplot2' for graphics, so the syntax may be unfamiliar to some of you.

```
library(popgraph)
library(ggmap)

location = c( mean(V(g)$x), mean(V(g)$y))
map = get_map(location, maptype="satellite", zoom=6) #use ggmap to get a satellite image of
the study area.

p = ggmap( map ) #plot will include this image
p = p + geom_edgeset(aes(x=x, y=y), g, color="white") #plot will also include the network
edges, in white
p = p + geom_nodeset(aes(x=x, y=y, color=region), g, size=6) #plot will also include the
network nodes, color-coded by region
p #plot color
```



3. MEASURING NETWORKS

Now that we have a handle on visualizing a network and the basics of relevant types of networks, we will go about the task of quantitatively describing its characteristics.

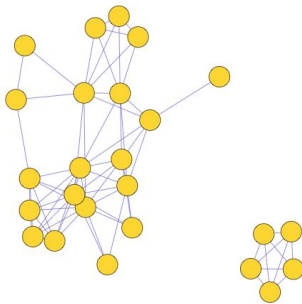
There are multiple levels at which we can measure and describe networks:

- Node-level
- Subcomponent-level
- Network-level

For this section, let's use the sample social network we used in section 2.1. We will first start by loading the required libraries, reading the sample association data, converting to an adjacency matrix, and then creating a graph to plot. We will just use the default Fruchterman-Reingold layout.

```
assoc=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016/Sample_association.csv",
header=T, row.names=1))
gbi=t(assoc)
mat=get_network(t(assoc), association_index="SRI")
g=graph_from_adjacency_matrix(mat, "undirected", weighted=T) #create a graph object

# plot the network with pretty colors
plot(g, vertex.label="", vertex.color="gold", edge.color="slateblue",
edge.width=E(g)$weight*10)
```



3.1 Centrality measures

Centrality is a general term that relates to measures of a node's position in the network. There are many such centrality measures, and it can be a daunting task to wade through all of the different ways to measure a node's importance in the network. Here, we will introduce just a few examples.

Degree and Strength

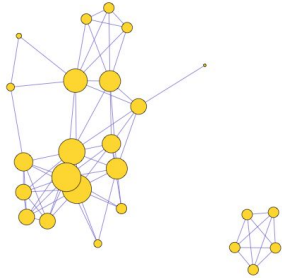
Let's start with the most straight-forward centrality metric: *degree centrality*. Degree centrality is simply the *number of edges connected to a given node*. In a social network, this might mean the number of friends an individual has. We can calculate degree centrality with a simple function:

```
degree(g)
```

```
> degree(g)
23820 23726 23831 23763 23772 23770 23771 23777 23774 23860 23779 23773 23862
    4     4     4     4     6     8    10     7     8     9     4    11    11
23857 23871 23853 23732 23734 23756 23759 23768 23758 23781 23815 23809
    2     3     4     3     1     6     6     6     7     4     4     4
```

Let's visualize what this means by varying the node sizes proportional to degree centrality.

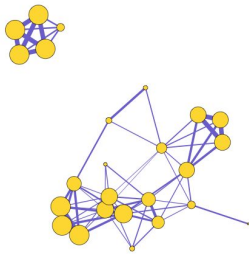
```
de=degree(g)
plot(g, vertex.label="", vertex.color="gold", edge.color="slateblue", vertex.size=de*2)
```



This helps us visualize which member of the network has many 'friends'.

In weighted networks, we can also 'node strength', which is the sum of the weights of edges connected to the node. Let's calculate node strength and plot the node sizes as proportional to these values.

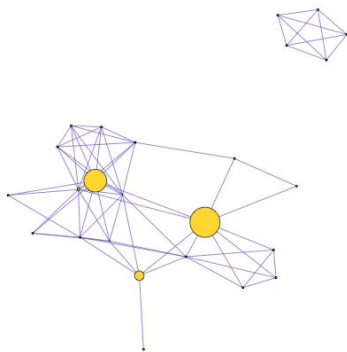
```
st=strength(g)
plot(g, vertex.label="", vertex.color="gold", edge.color="slateblue",
edge.width=E(g)$weight*10, vertex.size=st*5)
```



Betweenness

Let's now do the same for *betweenness centrality*, which is defined as *the number of geodesic paths (shortest paths) that go through a given node*. Nodes with high betweenness might be influential in a network if, for example, they capture the most amount of information flowing through the network because the information tends to flow through them. Here, we use the normalized version of betweenness.

```
be=betweenness(g, normalized=T)
plot(g, vertex.label="", vertex.color="gold", edge.color="slateblue", vertex.size=be*50)
```



You can see that there are three nodes that have qualitatively higher betweenness values than all other nodes in the network. One way to interpret this is that these are nodes that tend to act as “bridges” between different clusters of nodes in the network (but of course, this is only sample data).

What does this say about the importance of these nodes? Well, that depends on the network and the questions--in particular how you might quantify ‘importance’ in your network.

Here’s a short list of some commonly-used centrality measures:

Centrality measure	Function	Description
Degree	degree()	Number of edges connected to node
Strength	graph.strength()	Sum of edge weights connected to a node (aka <i>weighted degree</i>)
Betweenness	betweenness()	Number of geodesic paths that go through a given node
Closeness	closeness()	Number of steps required to access every other node from a given node
Eigenvector centrality	eigen_centrality()	Values of the first eigenvector of the graph adjacency matrix. The values are high for vertices that are connected to many other vertices that are, in turn, connected many others, etc.

Assembling a dataset of node-level measures

So now we know the basics of how to get centrality measures. For data analysis, we will likely want to measure and compare measures of node centrality with other traits. This will require putting together a dataframe that combines vertex attributes and centrality measures.

Let’s say we want to assemble a dataset of node centrality for the Karate Club network. Let’s use the three centrality measures we already familiar with: *degree*, *strength*, and *betweenness*.

```
names=V(g)$name
de=degree(g)
st=strength(g)
be=betweenness(g, normalized=T)

d=data.frame(node.name=names, degree=de, strength=st, betweenness=be) #assemble dataset
head(d) #display first 6 lines of data

> head(d) #display first 6 lines of data
  name degree strength betweenness
23820 23820     4 3.333333  0.000000
23726 23726     4 3.333333  0.000000
23831 23831     4 3.333333  0.000000
23763 23763     4 3.333333  0.000000
23772 23772     6 1.333333  0.1123188
23770 23770     8 2.683333  0.000000
```

In this case, there are rownames that are pre-set (because this is known automatically called in centrality function like degree()), but I like to keep the redundancy because this way I can easily tell that there are no discrepancies in my row orders.

Finally, we can write this dataset into a .csv file.

```
write.csv(d, "centrality.csv")
```

This command writes the specified object as a .csv file. This file will show up in the “working directory”, which you can set under Preferences > Startup. You can look up the current working directory this way:

```
getwd()
```

3.2 Network-level measurements

Size and density

Let's start by getting some basic information for the network, such as the number of nodes and edges. There are a couple of functions to help you extract this information without having to look it up in the "object summary" (e.g., `summary(g)`). Using these functions, you can store this information as separate objects, e.g., n for # nodes and m for # edges.

```
n=vcount(g)
m=ecount(g)
n
m
```

```
> n
[1] 25
> m
[1] 70
```

Since we now have the network size and the number of edges, we can calculate the *density* of the network. The definition of network density is:

density = [# edges that exist] / [# edges that are possible]

In an undirected network with no loops, the number of edges that are possible is exactly the number of dyads that exist in the network. In turn, the number of dyads is $\frac{n(n-1)}{2}$ where n = number of nodes. With this information, we can calculate the density with the following:

```
dyads=n*(n-1)/2
density=m/dyads
density
```

```
> density
[1] 0.2333333
```

Of course, there is a pre-packaged function for calculating density, called `edge_density()`:

```
edge_density(g)
```

```
> edge_density(g)
[1] 0.2333333
```


Components

When networks are ‘fully connected’, you can follow edges from any given vertex to all other vertices in the network. Alternatively, networks can be composed of multiple components that are not connected to each other, as with our sample network above. We can get this information with a simple function (output not shown).

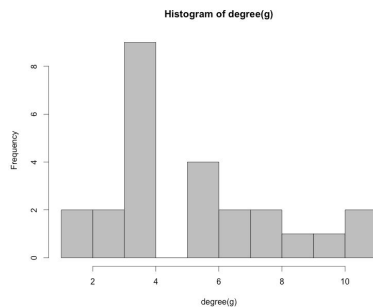
```
components(g) #number of components, sizes of components, and membership of components
```

Degree distributions

Degree distribution—i.e., the statistical distribution of node degrees in a network—is a common and often powerful way to describe a network. We will play around with this more when we talk in depth about ‘random graphs’, but the specific degree distribution can help distinguish networks with specific properties (e.g., ‘scale-free networks’).

We could simply look at the degree distribution as a histogram of degrees:

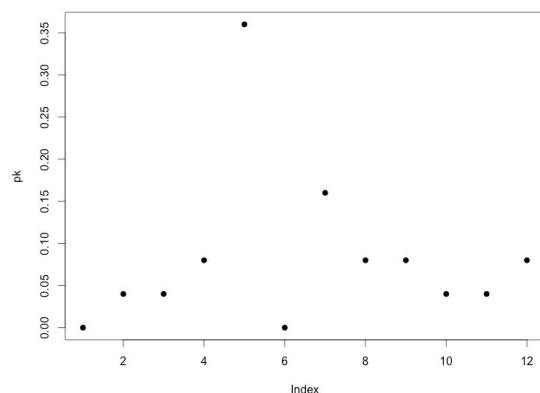
```
hist(degree(g), breaks=10, col="gray")
```



However, if we wanted to compare the degree distributions of different networks, it might be more useful to plot the probability densities of each degree: i.e., what proportion of nodes has degree = 1, degree = 2, etc.

We can do this by using a pre-packaged function called `degree.distribution()`. Try this out:

```
pk=degree.distribution(g)
plot(pk, pch=19)
```



Average path length & Diameter

In network jargon, a “path” is typically a shorthand for “geodesic path” or “shortest path”—the fewest number of edges that you would have to go on to get from one node to another. The average path length and the ‘diameter’ (maximum path length) can be useful measures of the network. The average path length can be considered the average “*degrees of separation*” between all pairs of nodes in the network, and the diameter is the maximum degree of separation that exists in the network.

You can calculate path lengths with or without the edge weights (if using edge weights, you often simply count up the weights as you go along the path). The igraph package includes a convenient function for finding the shortest paths between every dyad in a network. Here, makes sure you specify the algorithm = “unweighted” (output not shown):

```
paths=distances(g, algorithm="unweighted")
paths
```

This matrix gives us the geodesic path length between each pair of nodes in the network. We can describe the network using some characteristics of the paths that exist in that network. However, you will notice that this matrix contains a bunch of cells that are “Inf” (i.e., infinity). This is because the network is **not connected**, and you can’t calculate path lengths between nodes in different components.

How should we measure the average path length & diameter of a network with multiple components? There are two common solutions. First is to ignore pairs of nodes that are in different components and only measure average lengths of the paths that exist. This solution doesn’t really make sense for the diameter--the diameter of an unconnected network should be infinity. The second solution is to measure each component separately. Let’s do each of these in turn.

Option 1:

To calculate the average path length while ignoring pairs of nodes that are in different components, we can first replace the “Inf” with “NA” in the path length matrix. Next, we want just the “upper triangle” or “lower triangle” of this matrix, which lists all the geodesic paths without duplicates.

```
paths[paths=="Inf"]=NA
mean(paths[upper.tri(paths)], na.rm=T)

> mean(paths[upper.tri(paths)], na.rm=T)
[1] 1.865
```

You can see that this is what the canned function mean_distances() does for unconnected networks because you will get the same value:

```
mean_distance(g)

> mean_distance(g)
[1] 1.865
```

Option 2:

To calculate the average path lengths and diameter separately for each component, we will first ‘decompose’ the network into a list that contains each component as separate graph objects. We can then use the lapply() function to calculate separate path length matrices, and sapply() function to calculate the mean and max for each matrix.

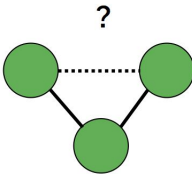
```
comps=decompose(g)
comps # a list object consisting of each component as graph object
path.list=lapply(comps, function(x) distances(x, algorithm="unweighted")) #make list object
with two path length matrices
avg.paths=sapply(path.list, mean) #average path length of each component
diams=sapply(path.list, max) #diameter of each component
```

```
avg.paths
diams
```

```
> avg.paths
[1] 0.800 1.815
> diams
[1] 1 3
```

Clustering coefficients

There are two formal definitions of the Clustering Coefficient (or Transitivity): “global clustering coefficient” and “local clustering coefficient”. Though they are slightly different, they both deal with the probability of two nodes that are connected to a common node being connected themselves (e.g., the probability of two of your friends knowing each other).



Here are the verbal definitions:

(1) *Global Clustering Coefficient* = “ratio of triangles to connected triples”

(2) *Local Clustering Coefficient* = for each node, the proportion of their neighbors that are connected to each other

(3) *Average Local Clustering Coefficient*: If C_i is the proportion of two nodes connected to node i that are also connected to each other (i.e., the Local Clustering Coefficient), then *Average Local Clustering Coefficient* = $\frac{1}{n} \sum_{i=1}^n C_i$

Try these:

```
g.cluster=transitivity(g, "global")
l.cluster=transitivity(g, "local")
av.l.cluster=transitivity(g, "localaverage")
```

```
g.cluster
l.cluster
av.l.cluster
```

```
> g.cluster
[1] 0.6219512
> l.cluster
[1] 1.0000000 1.0000000 1.0000000 1.0000000 0.4666667 0.4642857 0.5555556 0.6666667
[9] 0.5714286 0.2500000 1.0000000 0.4727273 0.4727273 1.0000000 0.3333333 1.0000000
[17] 1.0000000      NaN 1.0000000 1.0000000 1.0000000 0.7142857 1.0000000 1.0000000
[25] 1.0000000
> av.l.cluster
[1] 0.7903199
```

4. COMMUNITY DETECTION & ASSORTATIVITY

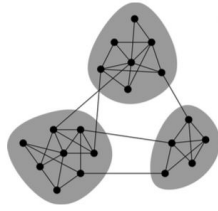


Figure from Newman, 2006

One of the ubiquitous properties of networks is that they exhibit *community structure*--the presence of discrete clusters of nodes that are densely connected, which themselves are only loosely connected to other clusters. In ecological contexts, these may be clusters of individuals that form social groups, groups of populations that form cohesive genetic or ecological units, or sets of species that have intimate ecological and evolutionary interactions. The problem is, how do we detect the presence of such *clusters* or *communities*, and how can we quantify the degree of community structure?

In a series of papers, Mark Newman and colleagues presented a quantitative measure called **modularity**, which quantify the degree to which such clusters are discrete (Girvan & Newman 2002; Neman & Girvan 2004; Newman 2006). The modularity index, Q , is a measure of *the proportion of edges that occur within communities, relative to the expected proportion if all edges were placed randomly*.

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

Where m is the total number of edges in a network, A_{ij} is the adjacency matrix, k_i and k_j are the degrees of node i and j , c_i and c_j refer to the communities to which i and j belong, and $\delta(c_i, c_j)$ is the Kronecker delta function, which equals 1 when $c_i = c_j$ and 0 otherwise. This value theoretically ranges from 0 to 1.

One class of methods for community detection (often called ‘modularity-optimization method’) to find the partitions in the network that assigns nodes into communities such that Q is maximized. The problem, however, is that there are too many possible partitions that can exist in a given network (ranging from 1 to n number of communities, with every possible combination of nodes), and so an exhaustive search is usually not feasible. Thus, modularity-optimization techniques rely on search algorithms that use different approaches (e.g., agglomerative *versus* divisive methods) with different strengths and weaknesses (we will hight a few below).

One should also be aware that modularity-based methods of community detection is not fool-proof. In particular, there is a well-studied resolution limit to modularity optimization--these methods tend to miss small, well-defined communities when there are other large communities. They also suffer from the basic assumption that each node belongs to just one community. While other community detection methods exist that can overcome some of these shortfalls, they too often have their weaknesses. In short, there is no perfect approach to community detection. Useful references to these debates include Palla et al. (2005), Fortunato (2010), Lancichinetti & Fortunato (2011), and countless others.

Community detection in igraph

There are several functions available for community detection in *igraph*, and all of them take the modularity-optimization approach.

Function	Basic idea	Reference
<code>edge.betweenness.community()</code>	One of the first in the class of “modularity optimization” algorithms. It is a “divisive” method. Cut the edge with highest edge betweenness, and recalculate. Eventually, you end up cutting the network into different groups. The best division is one that maximizes modularity.	Newman & Girvan 2004
<code>fastgreedy.community()</code>	Hierarchical agglomerative method that is designed to run well in large networks. It’s basically a hierarchical clustering method, but you create “multigraphs” where you lump groups of nodes together in the process of agglomeration in order to save time on sparse graphs.	Clauset et al. 2004
<code>walktrap.community()</code>	Uses random walks to calculate distances, and then use agglomerative method to optimize modularity	Pons & Latapy 2005
<code>spinglass.community()</code>	This method uses the analogy of the lowest-energy state of a collection of magnets (a so-called spin glass model). It is similar to a simulated annealing procedure.	Reichardt & Bornholdt 2006
<code>leading.eigenvector.community()</code>	This is a “spectral partitioning” method. You first define a ‘modularity matrix’, which sums to 0 when there is no community structure. The leading eigenvector of this matrix ends up being useful as a community membership vector. However, I think this is only really good when you have two communities...	Newman 2006
<code>label.propagation.community()</code>	I have not used this one...	Raghavan et al. 2007
<code>multilevel.community()</code>	I have not used this one...	Blondel et al. 2008
<code>rnetcarto::netcarto()</code>	Simulated Annealing method. Thought to be useful for smaller networks. Not available in <i>igraph</i> , but available through ‘ <i>rnetcarto</i> ’ package (Doulcier 2015).	Guimera & Amaral 2005

The Assortativity Coefficient

One major pattern common to many social networks (and other types of networks) is *homophily* or *assortativity*—the tendency for nodes that share a trait to be connected. The *assortativity coefficient* is a commonly used measurement of homophily. It is similar to the modularity index used in community detection, but the assortativity coefficient is used when we know *a priori* the ‘type’ or ‘value’ of nodes. For example, we can use the assortativity coefficient to examine whether discrete node types (e.g., gender, ethnicity, species, etc.) are more or less connected to each other. Assortativity coefficient can also be used with “scalar attributes” (i.e., continuously varying traits).

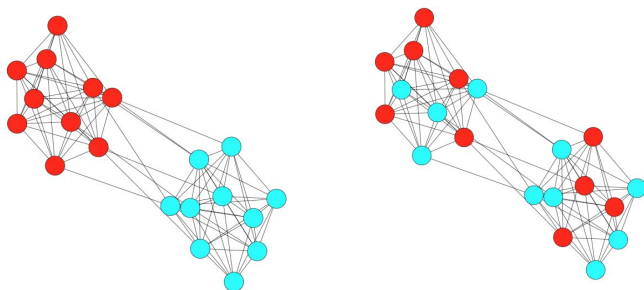


Figure: Two networks with different patterns of assortment. Node color represents some discrete node value (e.g., male vs. female). Left, an assortative, or homophilous, network. Right, a network with no assortativity.

The assortativity coefficient has a general form,

$$r = \frac{\sum_{ij} (A_{ij} - k_i k_j / 2m) x_i x_j}{\sum_{ij} (k_i \delta_{ij} - k_i k_j / 2m) x_i x_j}$$

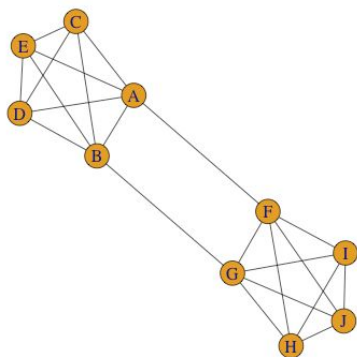
where $k_i \delta_{ij}$ is the degree of i excluding its connection with j . This is actually a form of the Pearson correlation coefficient. The coefficient is large (approaches 1) if nodes with similar values are more connected and small (approaches -1) when similar nodes are less connected. The value is 0 when edges are random with respect to node values.

For more on the assortativity coefficient, I recommend reading the original paper on this by Newman (2002), and its extension by Farine (2014).

Modularity: A simple example

To illustrate how to measure modularity and assortativity using igraph, let's start by creating a simple graph with clear community structure. First, we'll create a network consisting of two clusters that are completely connected, which themselves are loosely connected.

```
g=make_graph(~A:B:C:D:E--A:B:C:D:E, F:G:H:I:J--F:G:H:I:J, A--F, B--G)
set.seed(7)
l=layout_with_fr(g)
plot(g, layout=l, edge.color="black")
```



Let's now apply a community detection algorithm to figure out the optimal division of this network into communities. Because the community division in this example is very clear, we can choose any of the community detection method in the table above and we are likely to come up with the same answer. Let's use one called the edge betweenness method:

```
eb=edge.betweenness.community(g)
eb
> eb
IGRAPH clustering edge betweenness, groups: 2, mod: 0.41
+ groups:
  $`1`
  [1] "A" "B" "C" "D" "E"

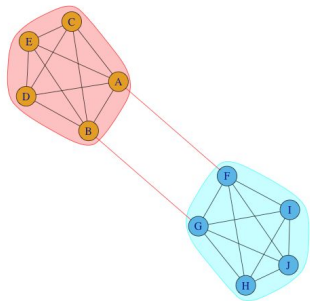
  $`2`
  [1] "F" "G" "H" "I" "J"
```

The resulting object is a 'communities object', which includes a few pieces of information--the number of communities (groups), the modularity value based on the node assignments, and the membership of nodes to each community. We can call each of these values separately (outputs not shown):

```
length(eb) #number of communities
modularity(eb) #modularity
membership(eb) #community membership of nodes
```

You can also use this communities object to produce a pretty network figure that shows the community structure by incorporating it into the plot() function.

```
plot(eb, g, layout=1)
```

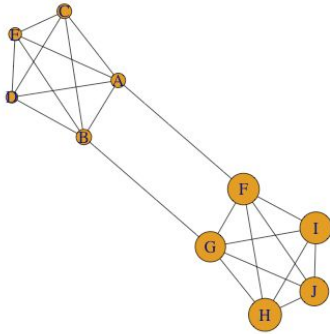


Assortment: a simple example

Let's use the same example network to demonstrate how to calculate assortativity, and to compare the difference between modularity and assortativity.

Let's start by assigning each node a value--let's say nodes vary in size.

```
V(g)$size=c(rnorm(5, mean=10), rnorm(5, mean=20)) #assign sizes to nodes using two normal
distributions with different means
plot(g, layout=1, edge.color="black")
```

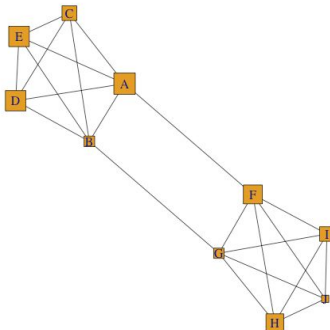


Now, we can measure the degree to which this network exhibits assortment by node size.

```
assortativity(g, V(g)$size, directed=F)
> assortativity(g, V(g)$size, directed=F)
[1] 0.8367375
```

As you can see, this network exhibits high levels of assortativity by node size ($r = 0.84$). As a comparison, let's create a node attribute that varies randomly across all nodes in the network, and then measure the assortativity coefficient based on this trait. We will plot the figure with square nodes, just to make it clear that we are plotting a different trait.

```
V(g)$random=rnorm(10, mean=10, sd=3) #create a node trait that varies randomly for all nodes
assortativity(g, V(g)$random, directed=F)
plot(g, layout=l, vertex.size=V(g)$random, vertex.shape="square")
> assortativity(g, V(g)$random, directed=F)
[1] 0.05002418
```



You can see that there is no assortment based on this trait.

Just to be extra clear, this network is still exhibits community structure (A through E and F through J), but the trait we are measuring does not exhibit assortativity.

*The igraph function for `assortativity()` can only handle unweighted networks. To measure assortativity on weighted networks, use the `assortment.discrete()` or `assortment.continuous()` function in the *assortnet* library (Farine 2014).

5. Network Permutations

Thus far, we have learned how to measure empirical networks in a variety of ways. However, we often want to compare these values against some baseline, i.e., a null hypothesis/model. Because most networks are complex and relational data do not conform to the requirement of non-independence of data, we often use randomizations/permutations to generate null models against which we can compare the empirical data, though new classes of statistical models to deal with relational data are becoming more widely adopted in the social sciences (e.g., Exponential Random Graph Models, or ERGM: Anderson et al. 1999; Robins et al. 2007) and hold great promise for applications of network analysis in ecology (Pinter-Wollman et al. 2014). Here, we focus narrowly on some classic forms of null model hypothesis testing using randomizations. In the context of animal social networks, there have been a couple of good review and tutorial of this method (Croft et al. 2011; Farine & Whitehead 2015). We will provide a quick overview here.

The main question we often want to address is this: *Is my network structure different from what is “expected by chance”?* To address these types of questions, we employ one of three different flavors of randomization/permutations: permuting nodes, permuting edges, and permuting group membership.

5.1 Node-label Permutations

Node-label permutations involve shuffling the node type or node values (e.g., sex, size, etc.) randomly across all nodes in a network, while keeping the edges the same. This ensures that the inherent structural pattern of the network remains the same, but the node values are randomized. This type of randomization scheme may be appropriate when you are interested in understanding the relations between types of nodes, e.g.:

- (a) testing correlations between node attribute and network position
- (b) using as null model for assortativity

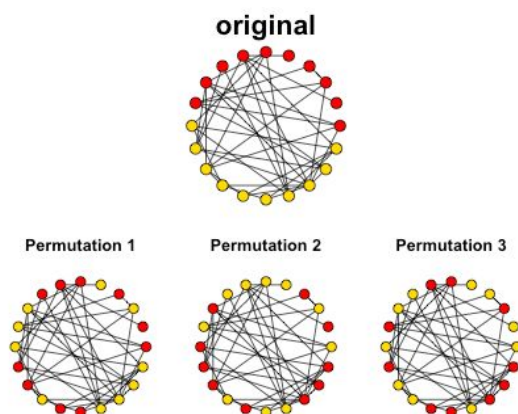


Figure: Illustration of node-label permutation method. Here, the edges stay the same, but the color of the nodes (representing node type or value) are shuffled.

You can find a worked example of the node-label permutation test in the worked example based on a study by Firth & Sheldon (2015) called “Testing assortment of RFID-tagged birds in Whytham Woods”

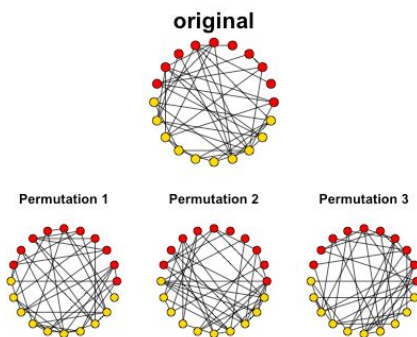
5.2 Edge Permutations

May be appropriate when:

- Not necessarily testing for the roles of particular node types
- Testing whether structure of network is non-random... but what is random?
- Must be careful about exactly how we permute edges--do we want to preserve any aspect of the connectivity of nodes?

5.2.1 Unconstrained edge permutations

One straight-forward way to randomize edges in a network is to generate 'random graphs' in which there are the same number of nodes and edges as the empirical network, but the edges are now distributed randomly. One can create such networks by generating a 'Erdős-Renyí random graph' (a random graph in which the probability that pairs of nodes are connected follows a uniform distribution) or by 'rewiring' the network randomly.



5.2.2. Edge rewiring while keeping the degree distribution constant

Alternatively, we may want to ask whether the degree distribution itself may affect the clustering coefficient of the network. That is, if the distribution of node degrees is non-random, that alone may cause the clustering coefficient to be larger than expected by Erdős-Renyí random graph. We can implement this with the `rewire()` function as well, with a different method.

I'll briefly explain how this edge rewiring method works. This method is sometimes called the "switching" algorithm because it works by identifying two edges that connect different pairs of edges and then switches the ends of these nodes (illustrated below). Milo et al. (2003) found that this method works well and produces a randomized graph after implementing this switching step m number of times, where m is the number of edges in the network.

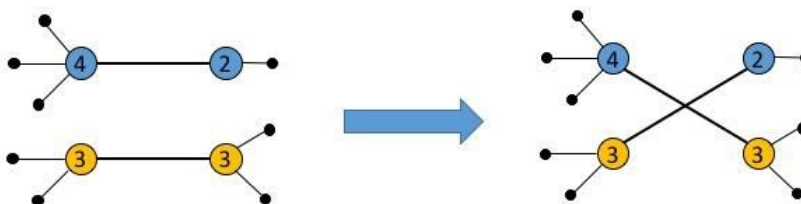


Figure illustrating the rewiring algorithm to preserve the degree distribution. The numbers in the nodes represent their degrees. In one step of the switching algorithm, we identify a pair of connected nodes that themselves are not connected to each other (represented in different colors), and we simply swap the ends of the two edges. You can see that this does not change the degree of the nodes.

5.3 Group membership permutation

The edge randomization with constant degree sequence as described above preserves much more of the network structure than a completely random graph. However, these edge randomization methods are both using association indices that are non-random based on the group membership of individuals. An alternative way to conduct a randomization is to permute the group membership data (i.e., the original group-by-individual matrix) such that we randomize which group (e.g., flock) that each individual is in. Now I will explain how the group membership swapping works. This method is sometimes called the “swap algorithm”, or “flipping procedure”. It is actually the same algorithm that community ecologists use to generate a null model for species co-occurrences (see Manly 1995; Gotelli 2000; Ulrich & Gotelli 2007). It has been adapted for generating a null model for social associations in animals by Bejder et al. (1998) and advocated for use in social network studies (Whitehead et al. 2005).

Step 1: Identify a 2x2 sub-matrix within the bipartite matrix that looks like: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Step 2: Swap the row/columns so that the 2x2 matrix looks like: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$


		Group						Group			
		A	B	C	D			A	B	C	D
Individual	1	0	1	1	0			1	0	1	0
	2	1	0	0	0			2	0	1	0
	3	0	0	1	0			3	0	1	0
	4	0	1	0	0			4	1	0	0
	5	1	0	0	1			5	1	0	1

Figure: One iteration of the sequential swap process. You can see that the row and column totals remain the same after each swap.

There are generally two ways to generate a P-value using the group membership swapping algorithm. First, one could repeat the swaps until the test statistic of interest (modularity in this case) stabilizes to a range of values corresponding to a randomized matrix, and then repeat this procedure a large number of times--say 1,000 times--to calculate a distribution of the test statistic under the null model (let's call it the 'global test'). Alternatively, one can run a large number of swaps from a single initial matrix, calculating a test statistic after each 'swap' of the matrix, and compare this distribution against the empirical test statistic ('serial test'). Manly (1995) discusses why the serial method is a valid method for testing whether the empirical matrix is non-random as long as we conduct a very large number of swaps. I will not get into the logic behind this--I highly recommend reading the Manly (1995) and references therein. The 'serial test' method is much more computationally efficient than the 'global test'.

You can find an example using the group membership swap method based on a study by Shizuka et al. (2014) in the worked example called “Comparing sparrow flock social networks across years”.

6. Network Regression (MRQAP): A Brief Overview

Perhaps the most common class of statistical models ecologists employ is the linear model. Linear models can be easily applied to predict the effect of one individual attribute on another (e.g., effect of node trait on node position in a network). However, the task is much more difficult when we want to predict the effect of some relational variable on network relations--e.g., how the similarity between two nodes affects the probability of an edge, or how network relations predict the transmission of pathogens. This is because relational data are not independent, so we require methods that can account for this non-independence.

The Mantel Test (Mantel 1967) has long been a useful approach for this type of data. The Mantel Test is a nonparametric matrix correlation test that uses permutations (node-permutations in this case) to get around the problem of non-independence. This test is widely available in several different R packages, for example those focused on community ecology (e.g., 'ecodist').

The Quadratic Assignment Procedure (QAP) is an extension of the Mantel Test in a regression framework, developed in the field of psychometrics. The Multiple Regression Quadratic Assignment Procedure (MRQAP) is an extension of this approach to allow for multiple covariate matrices (Krackhardt 1988). Essentially, MRQAP allows you to determine the influence of one matrix on another, controlling for the effects of one or more covariate matrices. There are several forms of MRQAP, but the most popular method is called the Double Semipartialling (DSP) method, developed by Dekker et al. (2007). This method permutes the matrix of residuals from the ordinary least regression of the dependent matrix on the independent matrices, to estimate error and calculate the effects.

MRQAP can be implemented in the R packages 'sna' (which is bundled inside 'statnet') and 'asnipe'. There are slight differences between the outputs for the two functions, but the results are nearly the same. Here is a simple code just to illustrate how to implement these:

```
library(sna)
library(asnipe)

#generate 3 random adjacency matrices using the rgraph() function within sna
m1=rgraph(10, m=1, tprob=0.5, mode="graph")
m2=rgraph(10, m=1, tprob=0.5, mode="graph")
m3=rgraph(10, m=1, tprob=0.5, mode="graph")

#test the effect of m2 on m1, controlling for m3. sna package function.
netlm(m1, m2+m3, mode="graph", nullhyp="qap", test.statistic="t-value")

#test the effect of m2 on m1 controlling for m3, and effect of m3 on m1, controlling for m2.
asnipe package function.
mrqap.dsp(m1~m2+m3, directed="undirected")
```

We will practice implementing this approach better in the worked example: “*Comparing sparrow flock social network across years*”

WORKED EXAMPLE:

Centrality, Community Structure and Resilience of US Air Transportation Network



This example is based on methods presented in: Guimerà, R., Mossa, S., Turtleschi, A., & Amaral, L. N. (2005). The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences of the United States of America*, 102(22), 7794–7799.

This classic study delves into the question of how we measure the importance of nodes in a network, and how different measures can tell us different things when networks are structured non-randomly. They also propose some alternative measures of position that tells a little bit more about the roles that different nodes play in a complex network. In addition, we will think a little bit about methods for testing the ‘resilience’ of networks to disturbance using virtual “knockout” experiments. Researchers in a wide variety of fields use these virtual knockouts to understand things like:

- the potential effects of node failure on power grids, transportation networks and information flow
- the effects of the loss of particular individuals in a social network
- the effects of habitat alteration on spatial networks of organisms

1. Getting and plotting the US air transportation network

The Guimerà et al. (2005) paper is based on the global air transportation network. Today, we will play with a smaller (but still plenty big) network of the US domestic air transportation network. The `igraphdata()` package contains a version of this network. In this network, the nodes represent airports and the edges represent flights that went between the airports in December 2010. You can import this data using the following set of codes:

```
library(igraph)
library(igraphdata)
data(USairports)
USairports

> USairports
IGRAPH DN-- 755 23473 -- US airports
+ attr: name (g/c), name (v/c), City (v/c), Position (v/c), Carrier (e/c), Departures (e/n), Seats (e/n),
| Passengers (e/n), Aircraft (e/n), Distance (e/n)
+ edges (vertex names):
[1] BGR->JFK BGR->JFK BOS->EWR ANC->JFK JFK->ANC LAS->LAX MIA->JFK EWR->ANC BJC->MIA
MIA->BJC TEB->ANC JFK->LAX LAX->JFK
[14] LAX->SFO AEX->LAS BFI->SBA ELM->PIT GEG->SUN ICT->PBI LAS->LAX LAS->PBI LAS->SFO
LAX->LAS PBI->AEX PBI->ICT PIT->VCT
```

This igraph object actually includes a vertex attribute called “Position” that encodes the latitude/longitude information. HOWEVER, I found that there are some errors in this data (due to slight errors in airport codes and coordinates). So, what we will do is to use an free, external dataset on airport locations, compiled by the Open Flights project (openflights.org). We can download the airport location data directly a url (output not shown):

```
airports=read.csv('https://raw.githubusercontent.com/jpatokal/openflights/master/data/airports.dat', header=F)
head(airports)
```

So, we can use this data and import the correct latitude (column 7) and longitude (column 8) for each airport as a node attribute:

```
V(USairports)$lat=airports[match(V(USairports)$name, airports[,5]), 7]
V(USairports)$long=airports[match(V(USairports)$name, airports[,5]), 8]
```

Now, we are going to do a little bit of “clean up” to the data. We are going to (1) ‘simplify’ the network into an undirected network with no loops, (2) remove nodes with airport codes that didn’t match the Open Flights database (probably data entry error in igraphdata), (3) remove US territories nodes that are in the Eastern and Southern Hemispheres to make it easier to see, and (4) keep only the airports that belong to the largest connected component (called the ‘giant component’), which will also make the network easier to interpret. Follow the codes below to end up with a new igraph object called usair:

```
#remove loops and make undirected
usair=as.undirected(simplify(USairports))

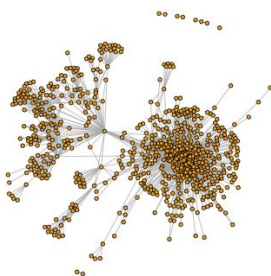
#remove airports whose codes didn't match the OpenFlights database (and hence returned "NA" for latitude)
usair=delete.vertices(usair, which(is.na(V(usair)$lat)==TRUE))

#remove nodes in the Eastern and Southern Hemispheres (US territories). This will make the plot easier to see.
usair=delete.vertices(usair, which(V(usair)$lat<0))
usair=delete.vertices(usair, which(V(usair)$long>0))

#keep only the largest connected component of the network ("giant component"). this also makes the network easier to see.
decomp=decompose.graph(usair)
usair=decomp[[1]]
```

Now, let’s take a look at this network:

```
set.seed(2)
l=layout_with_fr(usair)
plot(usair, layout=l, vertex.label="", vertex.size=3)
```

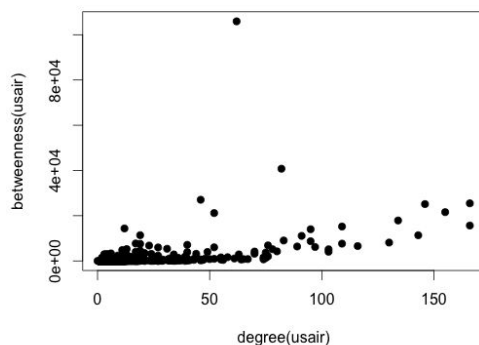


2. Relationship between degree and betweenness of U.S. airports

How should we define the ‘importance’ of an airport in this network? Degree and betweenness are the two most commonly used measure of node centrality (i.e., measures of node position in a network), and in this case, they represent two different measures of importance. Degree centrality tell us how many airports one fly to directly from a given airport. In contrast, betweenness centrality (which is the number of geodesic paths go through the node) gives us a sense for how important the airport is as a transfer location for people getting from one part of the country to another. Note that we are using a simplified network here that only counts unique flight routes, so we are not accounting for the number of replicated flights or the size of the planes.

Let’s explore the relationship between degree and betweenness by generating a simple scatterplot:

```
plot(degree(usair), betweenness(usair), pch=19) #pch=19 uses filled circles
```



Compare this to Figure 2a in Guimerà et al. (2005). We can see that in both cases, there is a strong relationship between degree and betweenness, but there are a few nodes that have high betweenness despite intermediate degree. When looking at the U.S. domestic flights only, there is one airport that stands out for having a huge betweenness value but moderate degree.

To see what some of these points are, let’s first take a look at the top 10 airports in terms of node degree:

```
V(usair)$name[order(degree(usair), decreasing=T)][1:10] #order() gives us the element number in order of ranking.
```

```
[1] "ATL" "DEN" "ORD" "MSP" "DFW" "DTW" "LAS" "IAH" "CLT" "LAX"
```

You can see that most of these are hubs in the interior of the lower 48 states--probably because we are not counting international flights in this network. Given that, there are few surprises here: we see a lot of familiar names here including Atlanta (ATL), Denver (DEN), O’Hare (ORD), Minneapolis (MSP), Dallas-Ft. Worth (DFW), etc.

Let’s now look at the top 10 airports in terms of node betweenness:

```
V(usair)$name[order(betweenness(usair), decreasing=T)][1:10] #order() gives us the element number in order of ranking.
```

```
[1] "ANC" "SEA" "DEN" "MSP" "ORD" "FAI" "BET" "DTW" "ATL" "LAX"
```

Many of the top 10 betweenness airports are familiar ones from looking at node degree: Denver (DEN), Minneapolis (MSP), Chicago O’Hare (ORD), Detroit (DTW), Atlanta (ATL) and LAX. However, there are some surprises in here: Anchorage (ANC) is the top airport in terms of betweenness by far. Second is Seattle (SEA), which is a big airport but did not appear in the top 10 for degree. There are two other

airports from Alaska in here: Fairbanks (FAI), the second largest city in Alaska, and perhaps most surprisingly, Bethel (BET), an airport based in a town of ~ 6,000 people!

Why this discrepancy between the top 10 nodes by degree and top 10 nodes by betweenness? This becomes clear when you look at the network plotted in space. We can also use the coordinates data that we downloaded from Open Flights project earlier to plot the network in space:

```
longlat=matrix(c(V(usair)$long, V(usair)$lat), ncol=2) #set up layout matrix
plot(usair, layout=longlat, vertex.label="", vertex.size=3)
```



You can see the outline of the US here. Now note how many airports there are in Alaska alone! This makes sense because there are so few roads in Alaska relative to landmass that flying is the main mode of transportation between remote towns. Thus, the airports in Alaska are not well connected in terms of the number of places you can get to by direct flight, but the hubs that connect Alaska to the rest of the U.S. (e.g., Anchorage and Seattle) are disproportionately important in terms of betweenness centrality.

The main point of this exercise is to consider what different indices of centrality are measuring and make sure you consider the complex structure of the network so that you can make sense of what that index is telling you.

3. Community structure in the U.S. air transportation network

One of the findings of the Guimerà et al. (2005) study was that the global airport network has significant community structure (see their Figure 3). That is, the globe divides up into discrete regions where the air transportation network are tightly knit, and the connections between them are relatively sparse.

We can apply community detection algorithms to figure out the cluster patterns of the U.S. domestic air transportation network. Guimerà et al. (2005) uses an algorithm called *simulated annealing* (Guimerà and Amaral 2005). This algorithm is available through a different R package called ‘*rnetcarto*’. However, since this algorithm runs a little bit slow for our purposes today, we will use a faster community detection algorithm called the fast greedy method (Clauset, Newman & Moore 2004), which yield similar results.

```
fg=fastgreedy.community(usair)
length(fg)
modularity(fg)
```

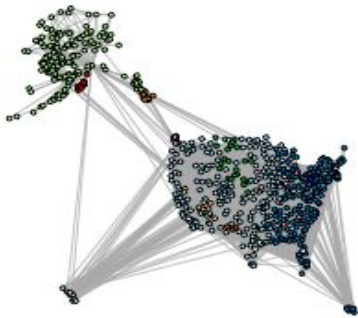
```
> length(fg)
[1] 14
> modularity(fg)
[1] 0.4085545
```

You can see that the algorithm found 14 communities with modularity value (Q) = 0.41. Now let's visualize this on the network. We will plot the network with nodes colored based on their community

membership. I will use the RColorBrewer to generate 12 colors (maximum distinct colors I can get), and make the last two communities white:

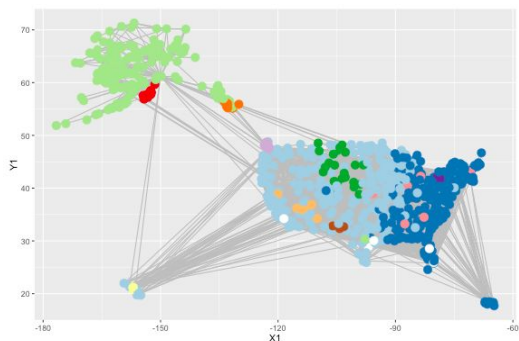
```
library(RColorBrewer)
fgmembership=membership(fg)
colors.fg=c(brewer.pal(12, name="Paired"), rep("white",2)) #first 12 colors from
RColorBrewer, last 2 communities are white

plot(usair, layout=longlat, vertex.label="", vertex.color=colors.fg[fgmembership],
vertex.size=3)
```



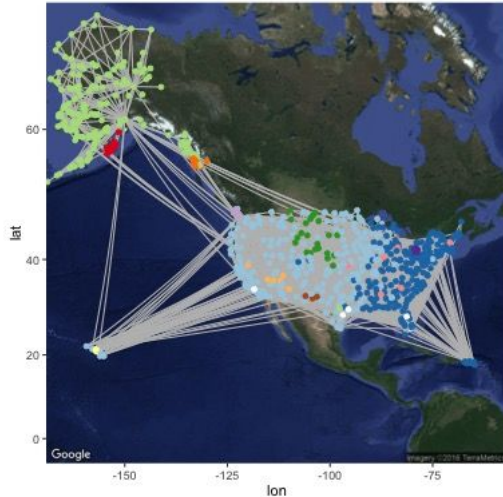
Let's make this figure prettier using the 'ggplot2' and 'popgraph' packages (see earlier Section 2.3).

```
library(ggplot2)
library(popgraph)
p=ggplot()
p=p + geom_edgeset(aes(x=long, y=lat), usair, color="gray")
p = p + geom_nodeset(aes(x=long, y=lat), usair, size=4, color=colors.fg[fgmembership])
p
```



Or make it even prettier by overlaying the plot on satellite image using 'ggmap':

```
library(ggmap)
location = c( mean(V(usair)$long), mean(V(usair)$lat))
map = get_map(location, maptype="satellite", zoom=3) #use ggmap to get a satellite image of
the study area.
p=ggmap(map)
p=p + geom_edgeset(aes(x=long, y=lat), usair, color="gray")
p = p + geom_nodeset(aes(x=long, y=lat), usair, color=colors.fg[fgmembership])
p
```



What you can see in these figures is that the U.S. transportation system is broken up into several communities that make some geographic sense: Alaska, West Coast/Hawaii, Eastern/Puerto Rico. There are also a few small, very local subnetworks like the Kenai Peninsula and Southern Alaska or the Northern Great Plains. If you look carefully, there are even a few small clusters that don't make much geographic sense--some of these are subnetworks of private/corporate airports or Air Force bases.

Guimerà et al. (2005) proposed that we can use this community structure to identify the roles that nodes play within and across their networks. They propose two measures in particular: Within-Community Degree (z) and Participation Coefficient (P). They define them as follows:

The Within-community Degree of node i is:

$$z_i = \frac{k_i - \bar{k}_{si}}{\sigma_{k_{si}}}$$

Where k_i is the number of links of node i to other nodes in its community, \bar{k}_{si} , then this value is the z-score of the node's degree within its community, standardized across communities.

The Participation Coefficient is defined as:

$$P_i = 1 - \sum_{s=1}^{N_M} \left(\frac{k_{is}}{k_i} \right)^2$$

Where k_{is} is the number of links of node i to nodes in community s , and k_i is the total degree of node i . N_M is the total number of communities. Thus, the participation coefficient is close to one if the links of a node are evenly distributed across all communities and zero if its links are exclusively with nodes of its own community.

We can manually calculate both of these scores:

```
##calculate within-module degree, z
z=list()
for (i in 1:max(fgmembership)){
  newg=delete.vertices(usair, which(fgmembership!=i))
  wideg=degree(newg)
  z[[i]]=(wideg-mean(wideg))/sd(wideg)
}
z=unlist(z)
## calculate participation coefficient
```

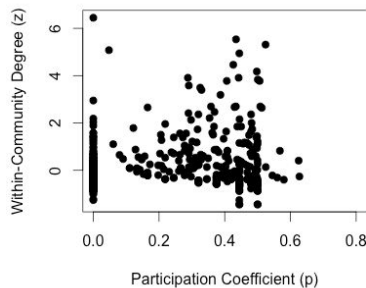
```

p=vector(length=vcount(usair))
for(i in 1:vcount(usair)){
  nei=neighbors(usair, i) #get all neighbors of node i
  tab=table(fgmembership[names=nei]) #get table of community membership for neighbors
  p[i]=1-sum((tab/sum(tab))^2)
}

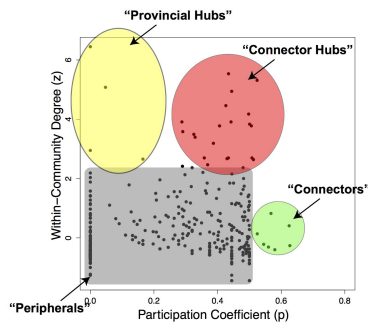
#create dataframe with airport name, z and p
dat=data.frame(name=V(usair)$name, z=z[match(V(usair)$name,names(z))], p=p)

```

Let's plot the z-P phase-space, as in Figure 4a of Guimerà et al. (2005):



In the paper, Guimerà and colleagues heuristically into several parts. Here is a simplified version of what they do:



Under this framework, we can identify several different types of roles.

- Most nodes are considered “Peripherals”, with relatively low z and low P .
- “Connector” nodes may have high P without having a high z : they don’t play a strong role in their own community but tend to bridge communities. These tend to be lesser-known airports (e.g., Topeka, Ketchikan).
- “Provincial Hubs” are nodes that have high degree within their community but low connections across communities. These include airports like Fairbanks and Bethel in Alaska.
- “Connector Hubs” are strongly connected within their community but also bridge their community with other communities. These include airports like Anchorage and Seattle.

The argument is that this type of description goes beyond simplistic interpretations of ‘degree’ and ‘betweenness’ and could be much more informative. Indeed, these indices are now widely used.

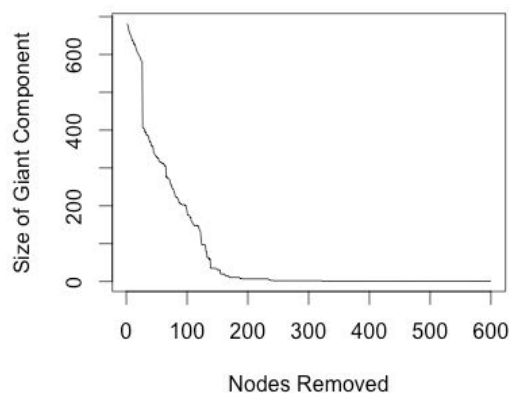
Testing the resilience of US air transport system to node failure

Thus far, we have removed one node at a time to ask questions about the importance of a particular node to the whole system. We can use a similar procedure to ask how vulnerable is the network to systematic failure of a set of nodes? For example, we can ask: how many nodes have to fail before the air transport system capacity breaks down? And importantly—is this dynamic different from random?

It is difficult to come up with a good criterion for “system breakdown”. Here, we will track one specific measure: the number of nodes in the *largest connected component of the system*. This is often called the ‘giant component’. In the case of a transportation network, this represents the largest part of the system where one can reach their destination via airplane. Recall that in the beginning of this exercise, we trimmed the original network to the giant component, so we are already starting with a connected component of the network.

Now, we want to remove different number of nodes and see how this affects the size of the biggest component. But in what order should we remove nodes? We could remove nodes in random order, or we could remove them in some systematic way. For this exercise, let's remove nodes in the order of their node degree in descending order. This simulates a scenario whereby the most “central” airports fail in sequence. We are going to remove nodes cumulatively—first 1 node, then 2 nodes, then 3 nodes... up to 600 nodes. Again, we are removing the nodes with highest degree first. After each iteration, we will measure the size of the giant component. We will then plot the result.

```
giant.size=vector(length=600)
g=usair
for (i in 1:600){
  g1=delete.vertices(g, which.max(degree(g)))
  giant.size[i]=max(components(g1)$csize)
  g=g1
}
plot(giant.size, type="l", xlab="Nodes Removed", ylab="Size of Giant Component")
```



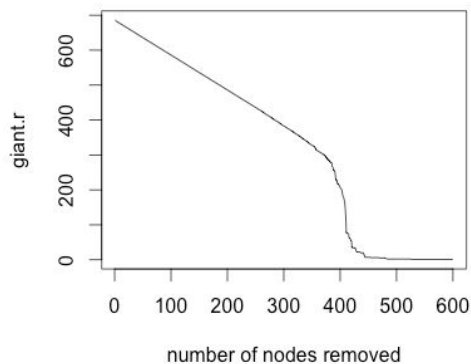
We see that the size of the giant component decreases surprisingly fast. But is this what we would generally expect from any kind of network? To ask this question, let's take a ‘random graph’ (officially called Erdős-Renyí random graph) of the same size and density as the US air transportation network and do this same routine again.

```

giant.r=vector(length=600)
r=erdos.renyi.game(n=vcount(usair), p=edge_density(usair))

for (i in 1:600){
  r1=delete_vertices(r, which.max(degree(r)))
  giant.r[i]=max(components(r1)$csize)
  r=r1
}
plot(1:600, giant.r, type="l", xlab="number of nodes removed")

```

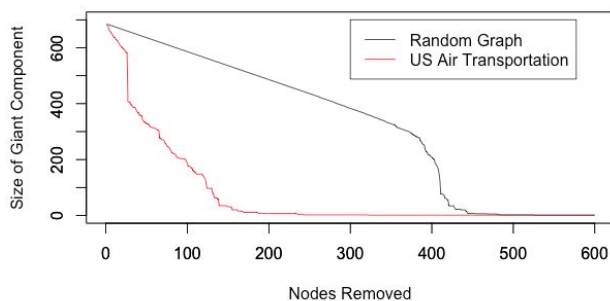


This looks quite different. Let's plot both of our results together in one figure. To do this, we will use `par(new=T)`, which allows us to overlay a plot on top of a previous plot. We have to make sure that we use the same axes for both plots. To make it look nicer, we will also remove the text from the axes for the second plot. We can also add a legend:

```

plot(giant.size, type="l", ylim=c(0,700), xlim=c(0,600), col="red", ylab="", xlab="")
par(new=T)
plot(giant.r, type="l", ylim=c(0,700), xlim=c(0,600), col="black", ylab="Size of Giant Component", xlab="Nodes Removed")
legend(300, 700, legend=c("Random Graph", "US Air Transportation"), lty=1, col=c("black", "red"))

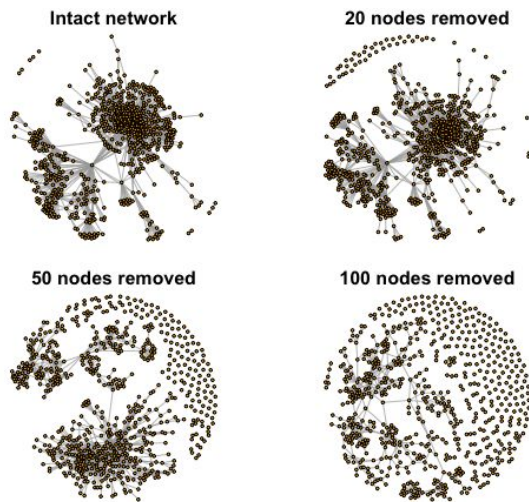
```



This plot reveals a very interesting dynamic—the real US air transportation network is highly vulnerable to the failure of a few key nodes. Once the first few dozen nodes are moved, the network quickly disintegrates into smaller components with no connections in between. Of course, be aware that this is a very simplified exercise that doesn't take into account the actual numbers of people traveling on each route.

We can visualize this dynamic by plotting the networks at a few different states—intact, 20 nodes removed, 50 nodes removed, and 100 nodes removed.

```
g20=delete_vertices(usair, order(degree(usair), decreasing=T)[1:10])
g50=delete_vertices(usair, order(degree(usair), decreasing=T)[1:50])
g100=delete_vertices(usair, order(degree(usair), decreasing=T)[1:100])
par(mfrow=c(2,2), mar=c(1,1,1,1))
plot(usair, vertex.label="", vertex.size=3, main="Intact network")
plot(g20, vertex.label="", vertex.size=3, main="20 nodes removed")
plot(g50, vertex.label="", vertex.size=3, main="50 nodes removed")
plot(g100, vertex.label="", vertex.size=3, main="100 nodes removed")
```



See if you can do this also with the random network of the same size and density and observe the difference in the dynamics.

Worked Example: Comparing plant-pollinator networks 120 years apart



(Photo: [szefei](#)/Shutterstock)

This example is based on the paper: Burkle, L. A., Marlin, J. C., & Knight, T. M. (2013). Plant-pollinator interactions over 120 years: loss of species, co-occurrence, and function. *Science*. <http://doi.org/10.1126/science.1230200>

In 2009-2010, Laura Burkle and colleagues resampled a plant-bee network that was originally sampled by Charles Robertson in the 1800s (Robertson 1929). This allowed them to compare the mutualism network at this site 120 years apart (Burkle et al. 2013)! What they found is pretty astounding:

- Half of the bee species were locally extinct, leading to massive loss of interactions.
- Many other interactions between plant species and bee species that were still surviving also disappeared. In total, only 24% of the original interactions were intact.
- However, there were also 121 new interactions that were not observed by Robertson.
- Across time, there can be massive rewiring of the mutualism network.
- This leads to big changes in network structure, such as nestedness.

Here, we will take the data presented in this paper to reconstruct two key figures and one analysis.

First, we will load the R packages and the network data collected in 2009-2010, which is available on Dryad (<http://datadryad.org/resource/doi:10.5061/dryad.rp321>). We can import the data directly using the correct link address for the .csv file:

```
library(igraph)
library(bipartite)
new.el=read.csv("http://datadryad.org/bitstream/handle/10255/dryad.48295/dryad%20-%20interac
tions%20now.csv")
head(new.el)
```

```
> head(new.el)
      plant      bee X.sum.ints
1 Dentaria_laciniata Andrena_arabis      2
2 Claytonia_virginica Andrena_carlini     64
3 Dentaria_laciniata Andrena_carlini     17
4 Dicentra_cucullaria Andrena_carlini      3
5 Erigenia_bulbosa Andrena_carlini    118
6 Erythronium_albidum Andrena_carlini     14
```

You can see that this data frame is an edge list--it has the two connected nodes in two columns, and edge weights in a third column. We will ignore the edge weights for this exercise because I don't have access to the edge weights for the Robertson's original network (that data was not deposited in the database).

I have, however, compiled the unweighted network from Robertson's original data using the published figure showing the interactions between plant and bee species (Figure S9A). You can download that data, as well as the ordering of the plant and bee species in that figure:

```
old.el=read.csv("https://dshizuka.github.io/NAOC2016//old_edgelist.csv")
old.order=read.csv("https://dshizuka.github.io/NAOC2016//old_species.csv")
```

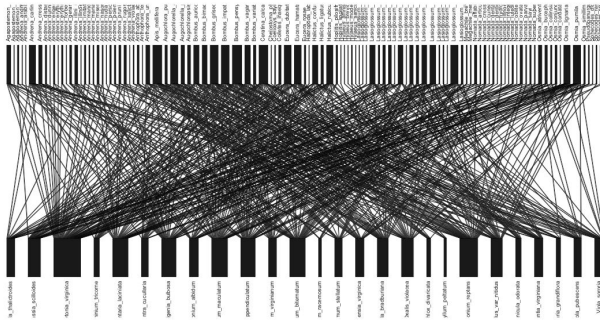
Let's now convert both the new network and old network from edge lists to interaction matrices, with plants on rows and bees on columns. We can do this simply using the `table()` function and converting it to a matrix (output not shown):

```
new.bip=as.matrix(table(new.el[,1:2]))
old.bip=as.matrix(table(old.el[,1:2]))
new.bip
```

Recreating Figure 1

We will now use the 'bipartite' package to visualize the "old" network from Robertson's data:

```
plotweb(old.bip, method="normal", text.rot=90) #method='normal' preserves the order of
species in matrix (currently alphabetical)
```



Now, let's try to recreate Figure 1 of the paper, which is this network colored by whether the bee species is now locally extinct (red boxes), whether the interaction is gone because the bee species is gone (red lines), or the interaction is gone for some other reason (blue lines).

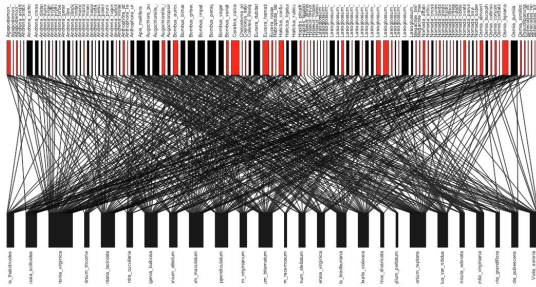
First, we can figure out if the bee species is gone by comparing the column names of the 'old' interaction matrix and the 'new' interaction matrix--if it is in the old but not the new, then it is gone (output not shown).

```
extinctbees=colnames(old.bip)[is.na(match(colnames(old.bip), colnames(new.bip)))] #the match
function in the bracket returns NA if the name is in old network but not the new network.
extinctbees
```

We can use this list of bee species to color these boxes red in the network plot:

```
bee.colors=rep("black", length(colnames(old.bip)))
bee.colors[match(extinctbees, colnames(old.bip))]='red'

plotweb(old.bip, method="normal", text.rot=90, col.high=bee.colors)
```

To color the lines, we have to first know that the `plotweb()` function allows you to color the interactions using the argument `col.interaction=`.

First, let's create an empty matrix for the line colors where we will assign the colors. These will have row and column names that correspond to the "old" interaction matrix.

```
linecols=matrix(nrow=nrow(old.bip), ncol=ncol(old.bip), dimnames=dimnames(old.bip))
```

We will now start assigning colors to interactions.

- (1) We will assign the color red to interactions that involve bee species that do not appear in the new dataset (locally extinct). We will do this by assigning the color 'red' to all interactions that correspond to columns of bees that go locally extinct.
- (2) We will then color all interactions that occur in both old and new datasets as black. To do this, we will construct a combined interaction matrix using both the old AND new edge lists. The interactions whose value = 2 in this combined matrix are interactions that exist in both datasets.
- (3) We will color the remaining interactions blue--these are ones that exist in the old network but do not exist in the new network, but do not correspond to columns with extinct bees.
- (4) Finally, let's set all of the interactions that actually do not exist in the network to NA.

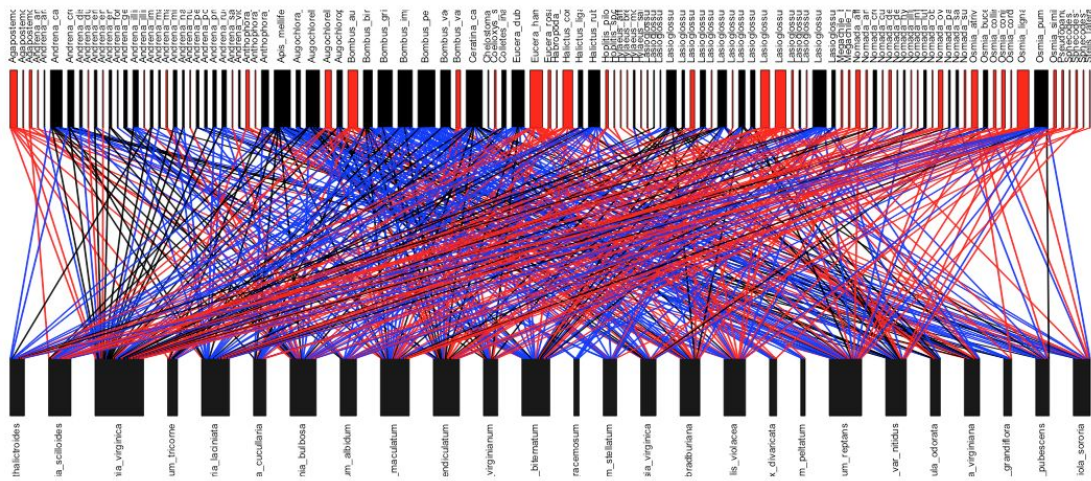
```
#color all columns with bees that will disappear in the new dataset = "red"
linecols[,match(extinctbees, colnames(linecols))]="red"

#make combined interaction matrix, and interactions that have 2 = exist in both datasets =
"black"
total.el=rbind(old.el[,1:2], new.el[,1:2])
total.bip=as.matrix(table(total.el$plant, total.el$bee))
linecols[which(total.bip==2)]="black"

#the rest of the interactions will be colored "blue"
linecols[is.na(linecols)]= "blue"

#set the interactions that do not exist in the old network to NA
linecols[which(total.bip==0)]=NA

#now plot
plotweb(old.bip, method="normal", col.interaction=t(linecols),
bor.col.interaction=t(linecols), text.rot=90, col.high=bee.colors)
```



This looks almost exactly like Figure 1 in the paper. The main differences come from the fact that the published figure includes edge weights, but ours do not (because I did not have access to the edge weight data from the ‘old network’).

Recreating Figure 2

Figure two is a colored matrix that displays similar information as Figure 1, but with addition of *new interactions* (in yellow) that did not exist in the old dataset but appear in the new dataset.

We can start this process using the ‘linecols’ matrix we created above. The only information that this matrix lacks is the new interactions--the interactions that exists in the new network but not the old network. We can figure out these interactions by using the combined interaction matrix we created above (total.bip). If we set all values >0 in this matrix to 1 (i.e., convert all 2 to 1), and then subtract it from the ‘old’ interaction matrix, then all the new interactions will have the cell value = -1:

```
total.bip[total.bip>0]=1
linecols[which(old.bip-total.bip==-1)]="yellow"
```

Now, we will convert this matrix of color names to numerics corresponding to their color assignments as determined in base R (white = 0, black = 1, red = 2, blue = 4, yellow = 7). Once we do that, we will have to convert all of these cell values from “character” to “numeric”.

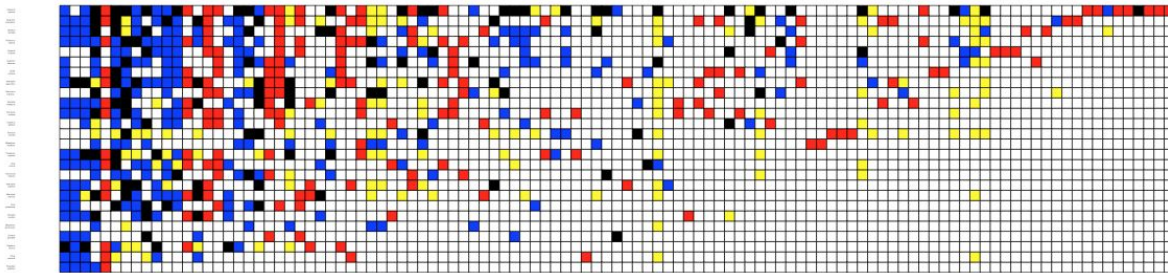
```
linecols[is.na(linecols)]=0
linecols[which(linecols=="black")]=1
linecols[which(linecols=="red")]=2
linecols[which(linecols=="blue")]=4
linecols[which(linecols=="yellow")]=7
linecols=matrix(as.numeric(linecols), nrow=nrow(old.bip), ncol=ncol(old.bip),
dimnames=dimnames(old.bip))
```

Finally, we want to re-order the rows and columns here to match the figure, which is based on their “nestedness position” (Figure 2 caption). I actually could not re-create this because they do not explain this ordering any further--so I will have to cheat and simply import the species order here as a separate .csv file:

```
fig.order=read.csv("https://dshizuka.github.io/NAOC2016/Fig2_order.csv")
row.order=fig.order[match(rownames(linecols),fig.order$species), "order"]
col.order=fig.order[match(colnames(linecols),fig.order$species), "order"]
linecols.ordered=linecols[order(row.order), order(col.order)] #matrix with the species
sorted as in Figure2
```

Now, we can use the `visweb()` function in `bipartite` package to generate Figure 2

```
colset=c(0, 1,2,4, 7) #sets the color according to cell values in increasing order
visweb(linecols.ordered, "none",square="defined", def.col=colset, clear=F) #def.cols=
argument means use the defined color set
```



Let's now calculate some specific numbers that tell us about the changes in the plant-bee network at this location over 120 years. The numbers here are very slightly different from what is reported in the paper--I think this is due to the fact that there are two bee species in Figure 2 of the paper that is not found in the dataset. I don't know where that discrepancy comes from, but it does not impact the result very much. Here are some examples exploring the differences in the networks (outputs not shown):

```
table(linecols) # table showing frequencies of cell types

table(linecols)[1]/sum(table(linecols)) #proportion of species pairs that never interact

table(linecols)[2]/sum(table(linecols)[c(2,3,4)]) #proportion of interactions existing in
old network that persisted to new network

table(linecols)[5] #number of new interactions

table(linecols)[5]/sum(table(linecols)[c(2,5)]) #proportion of new network links that were
new.
```

Comparing nestedness of old and new networks

In the paper, Burkle et al. (2013) found that the plant-bee network that existed 120 years ago was significantly nested, while the current plant-bee network is not. Given a series of previous work suggesting that nestedness can promote stability and biodiversity in mutualism networks (Bascompte & Jordano 2007; Bastolla et al. 2009), this could have important implications.

Below is a sample code that one might use to conduct this comparison. Note that the results are not going to be exactly the same as in the published paper because my networks do not include edge weights (due to lack of edge weight data for the old dataset in the supplemental materials).

```
##nestedness compared to null model
```

```

library(vegan)
ne.old=nestedtemp(old.bip)$statistic

nm=vegan::nullmodel(old.bip, method="swap")
nulls=simulate(nm, nsim=100)
null.nest=apply(nulls, 3, function(x) nestedtemp(x)$statistic)
hist(null.nest, xlim=c(min(null.nest)-0.5, max(null.nest)+0.5))
abline(v=ne.old, col="red", lty=2)

p.old=(length(which(null.nest<=ne.old))+1)/101
p.old

ne.new=nestedtemp(new.bip)$statistic

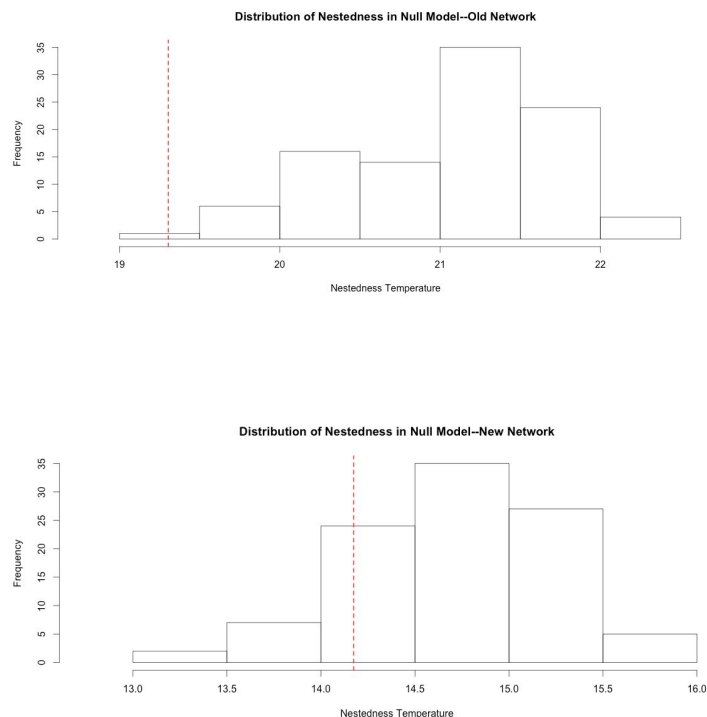
nm=vegan::nullmodel(new.bip, method="swap")
nulls=simulate(nm, nsim=100)
null.nest=apply(nulls, 3, function(x) nestedtemp(x)$statistic)
hist(null.nest)
abline(v=ne.new, col="red", lty=2)

p.new=(length(which(null.nest<=ne.new))+1)/101
p.new

```

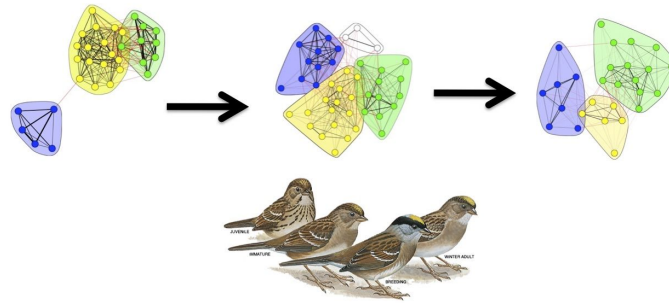
P-value for old network = 0.02

P-value for new network = 0.18



Figures above show the distribution of nestedness values for 100 null models (using the Manly ‘swap’ method), and the red line shows the nestedness value for the empirical network (Robertson’s dataset above, current dataset below). This is using unweighted interactions.

Worked example: Comparing sparrow flock social network across years¹



We will use data from our ongoing study on social networks of flock membership in a wintering population of golden-crowned sparrows, *Zonotrichia atricapilla* (Shizuka et al. 2014). Golden-crowned sparrows are long distance migrants that breed in western Canada and Alaska and winter along the west coast of mainland US. This work is based at a small (~7 ha) field site in California (UC Santa Cruz Arboretum) where golden-crowned sparrows are abundant during the non-breeding season (October-March). Since 2009, my colleagues and I have been collecting data on which individually-marked sparrows are observed together in flocks (typically 2-10 birds), defined as a collection of individuals within a 5m radius during a given observation. Flock membership changes within minutes—some birds leave and other birds join. Each year, we built a social network based on flock co-membership. In this paper, we showed that the social networks of wintering sparrows are highly clustered, with ‘social communities’ of individuals that flock together throughout the season. These social communities are somewhat fluid and spatially overlapping. Nevertheless, they seem to be quite stable across years--i.e., all birds leave for the breeding season, but those that come back tend to form the same clusters again. Moreover, we show using MRQAP analysis that past association (association index from the previous year) is a better predictor of current flock associations than a metric of home range overlap. That is, given that two individuals overlap in space, they are more likely to flock together if they had flocked together often in the previous year--a potential signature of social fidelity that bridges years.

This study was based on three seasons of data (side note: we are now analyzing 6 years of data--and new patterns are emerging!). Here, I will demonstrate some of the analysis techniques employed here using the data from Season 2 and Season 3.

First, make sure that we load the required packages and import the individual-by-group matrices for Season 2 and Season 3. Here, we will combine these matrices into a single list object.

```
library(igraph)
library(asnipe)
mat2=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016/GCSPflocks2.csv", header=T,
row.names=1))
mat3=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016/GCSPflocks3.csv", header=T,
row.names=1))
mats=list(mat2, mat3)
```

¹ This module is based on the data published in: Shizuka, D., Chaine, A. S., Anderson, J., Johnson, O., Laursen, I. M., & Lyon, B. E. (2014). Across-year social stability shapes network structure in wintering migrant sparrows. *Ecology Letters*, 17(8), 998–1007. <http://doi.org/10.1111/ele.12304>

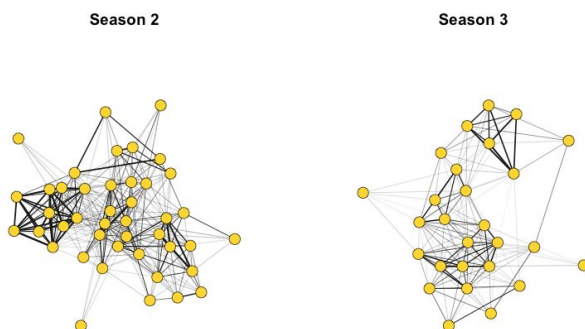
As you will see, having these matrices in a list object allows us to use `lapply()` and `sapply()` functions to run a routine on both datasets. This will save you some time and lines of code.

For example, let's do the following routine on both datasets at once: (1) Take out transient individuals that were seen fewer than three times during the course of the study each season. (2) Convert the individual-by-group matrix into an adjacency matrix using the `get_network()` function in `asnipe`. (3) Create `igraph` objects from these adjacency matrices.

```
newmats=lapply(mats, function(x) x[which(rowSums(x)>2), ]) #take out transients
assocs=lapply(newmats, function(x) get_network(t(x), "GBI")) #convert to adjacency matrices
#(transpose the association matrix first from individual-by-group to group-by-individual)
gs=lapply(assocs, function(x) graph_from_adjacency_matrix(x, "undirected", weighted=T))
#make weighted networks
gs
```

Now, let's plot these networks side-by-side. Here, we are going to use a trick to save the default graphical parameters before setting it to display two networks side-by-side. That way, we can reset the parameters after we're done:

```
seasons=c("Season 2", "Season 3") # save for plot title
default=par() #save default graphical parameters first
par(mfrow=c(1,2)) #set up to plot networks side-by-side
for(i in 1:2){
  plot(gs[[i]], edge.width=E(gs[[i]])$weight*10, vertex.label="", vertex.color="gold1",
  vertex.size=10,edge.color="gray10", main=paste(seasons[i]))
}
par(default)
```



Modularity of the network

One of the first things we did in this study is to determine whether there were discrete clusters in this network that would suggest that individual segregate out into social communities within this small study area.

```
coms=lapply(gs, function(x) cluster_fast_greedy(x)) #apply community detection function to
each network

mods=sapply(coms, modularity) #calculate modularity based on community assignments

com.colors=list(c("blue", "yellow", "green", "red"), c("green", "yellow", "blue")) # assign
colors to communities. Community colors are in different order for each year because
community ID number depends on the order of nodes that belong to them.

set.seed(10) #make plots reproducible
par(mfrow=c(1,2))
```

```

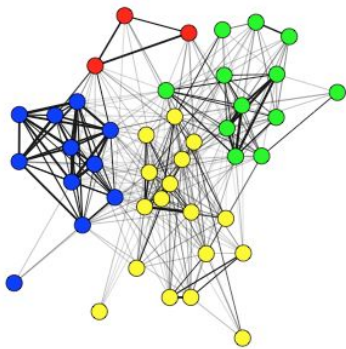
for(i in 1:2){
  l=layout_with_fr(gs[[i]])

  V(gs[[i]])$color=com.colors[[i]][membership(coms[[i]])]

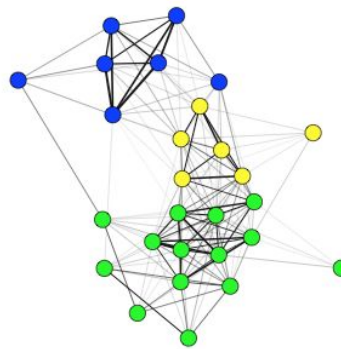
  plot(gs[[i]], layout=l, edge.width=E(gs[[i]])$weight*10, vertex.label="",
       vertex.size=10, edge.color="gray10", main=paste(seasons[i], ": Modularity =",
       round(mods[[i], 2)))
}
par(default)

```

Season 2 : Modularity = 0.49



Season 3 : Modularity = 0.43



In both years, the network appears to be highly clustered. The colors of the clusters are assigned based on the consistency of the home ranges of each community across years (see Shizuka et al. 2014 for details).

Note: Here and in the paper, we used a ‘fast and greedy’ community detection algorithm (Clauset, Newman & Moore 2004) because it performed best on our networks. More recently, I have confirmed that an alternative method called Simulated Annealing (which is reported to perform better on smaller networks: Guimerà and Amaral, 2005) yields identical results.

Testing modularity of empirical network against randomized networks

Randomized networks using group membership swaps

To see whether this pattern of network clustering is meaningful, we want to compare this against some kind of null model. For networks based on association data (e.g., individuals in groups), the basic null model should be constructed using a ‘group membership swap’ method first proposed by Manly (1995) and adapted to animal social data by Bejder (1998). The method is reviewed in detail in Whitehead et al. (2005). See the earlier section on group membership swaps in this handbook (Section 5.3).

This swapping algorithm is available in two different functions within the ‘asnipe’ package: `network_swap()` and `network_permutation()`. The two function are slightly different in how their outputs and are useful for different reasons.

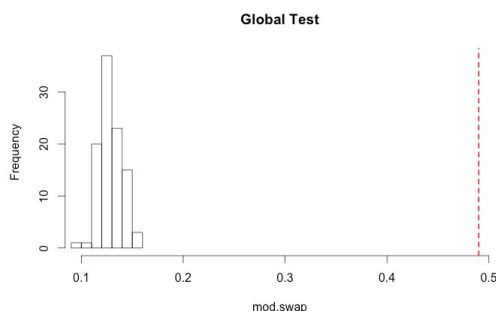
There are generally two ways to generate a P-value using the group membership swapping algorithm. First, one could repeat the swaps until the test statistic of interest (modularity in this case) stabilizes to a range of values corresponding to a randomized matrix, and then repeat this procedure a large number of

times--say 1,000 times--to calculate a distribution of the test statistic under the null model (let's call it the 'global test'). Alternatively, one can run a large number of swaps from a single initial matrix, calculating a test statistic after each 'swap' of the matrix, and compare this distribution against the empirical test statistic ('serial test'). Manly (1995) discusses why the serial method is a valid method for testing whether the empirical matrix is non-random as long as we conduct a very large number of swaps. I will not get into the logic behind this--I highly recommend reading the Manly (1995) and references therein. The 'serial test' method is much more computationally efficient than the 'global test'. In the paper, we used the 'global method' in which we conducted a large number of swaps and repeated this procedure 1,000 times. Below, I will show how to generate P-values using both the 'global' and 'serial' methods:

Global test: Let's do 100 runs of the group membership swapping, with 500 swaps per run (in the paper, we conducted 1,000 runs of ~5,000 swaps). We will calculate modularity at the end of each iteration and then generate a histogram.

```
gbi=t(newmats[[1]])
swap.m=list()
times=100
for (k in 1:times){
  swap.m[[k]]=network_swap(gbi, swaps=500)$Association_index
}
swap.g=lapply(swap.m, function(x) graph_from_adjacency_matrix(x, "undirected", weighted=T))
mod.swap=sapply(swap.g, function(x) modularity(cluster_fast_greedy(x)))
hist(mod.swap, xlim=c(min(mod.swap), mods[[1]]))
abline(v=mods[[1]], col="red", lty=2, lwd=2)
p=(length(which(mod.swap>=mods[[1]]))+1)/(times+1)
p
```

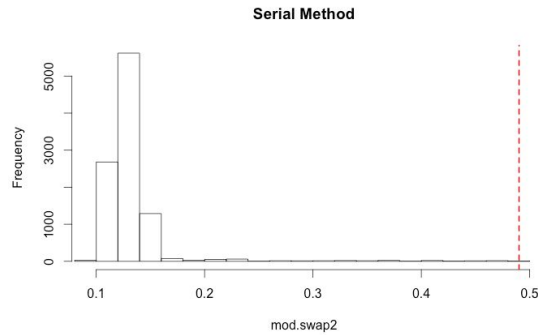
```
> p
[1] 0.00990099
```



Serial Method:

```
gbi2=t(newmats[[1]])
assoc2=get_network(gbi2)
net.perm=network_permutation(gbi2, permutations=10000, returns=1, association_matrix =
assoc2)
swap.g2=apply(net.perm, 1, function(x) graph_from_adjacency_matrix(x, "undirected",
weighted=T))
mod.swap2=sapply(swap.g2, function(x) modularity(cluster_fast_greedy(x)))
hist(mod.swap2,xlim=c(min(mod.swap2), mods[[1]]), main="Serial Method")
abline(v=mods[[1]], col="red", lty=2, lwd=2)
p=(length(which(mod.swap2>=mods[[1]]))+1)/100001
p
```

```
> p
[1] 9.9999e-06
```

In either case, the empirical modularity value is much larger than what we expect from random. However, the P-values are different because we are comparing the empirical test statistic against a much larger pool of randomized values in the ‘serial method’ test (10,000 vs. 100).

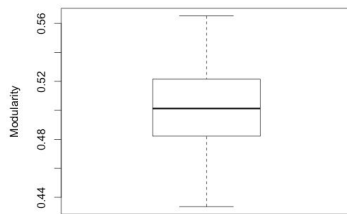
Bootstrapping to account for sampling error

Above, we compared the single empirical value of modularity against a distribution of modularity values generated from randomizing the network. But what about the error around the empirical modularity value itself? Given that we did not sample all flocks existing throughout the season, there must be error associated with this empirical modularity value as well.

One popular way to account for sampling error in all areas of ecology is *resampling*--i.e., randomly picking a subset of observations (e.g., jackknifing) or randomly resampling the data while holding sample size constant (e.g., bootstrapping). We can use a bootstrapping scheme to generate a distribution of modularity values associated with our empirical network. NOTE, however, that there are some inherent difficulties in using bootstrapping to estimate error in modularity--namely, the bootstrapped network may result in different numbers of communities, which affects the modularity value (discussed in Shizuka & Farine 2016). For now, I will simply demonstrate how to implement a bootstrapping procedure on the individual-by-group matrix and use this to generate a simple distribution of modularity values without getting into these confounding effects.

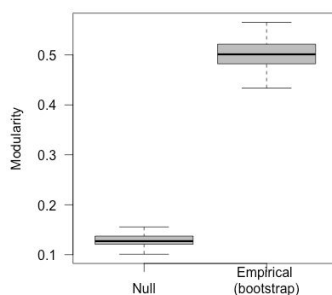
To implement the bootstrapping procedure, we will resample with replacement the observed flocks (columns of the individual-by-group matrix) and recalculate the modularity value using the same community detection algorithm as we used on the empirical (and the randomization scheme above). We will repeat this 100 times.

```
com.boot=vector(length=100) #set up empty vector
for (i in 1:100){
  s.col=sample(1:ncol(newmats[[1]]), ncol(newmats[[1]]), replace=T) #sample with replacement
  the columns
  nm1=newmats[[1]][,s.col] #create new matrix using resampled columns
  g.boot=graph_from_adjacency_matrix(get_network(t(nm1)), "undirected", weighted=T)
  #bootstrapped network
  com.boot[i]=modularity(cluster_fast_greedy(g.boot)) #calculate modularity using
  fast_greedy community detection
}
boxplot(com.boot)
```



We can compare the modularity values generated from this bootstrapping scheme against the modularity values generated from the randomization (i.e., group membership swap) presented above.

```
boxplot(mod.swap, com.boot, col="gray", las=1, names=c("Null", "Empirical \n(bootstrap)"),
ylab="Modularity")
```



Note: In the actual study, we used a second type of randomization scheme, which we call the “spatial null model”, which accounts for the home ranges of each individual and the location of each flock observation (not presented here). Figure 2 from the Shizuka et al. (2014) paper presents the resulting distributions of modularity values from the two randomization schemes (group membership swap & spatial flock model) and the bootstrapped empirical values.

Testing the consistency of association patterns between years

One of the goals of this study was to determine whether flock association patterns were consistent across years--i.e., do birds re-create the social networks year to year? To test for this pattern for each pair of years, we filtered the networks to those individuals that were seen in both years and then conducted a Mantel Test to see if association indices were correlated across years. Here is one way to conduct this analysis using the ‘ecodist’ package for its Mantel test function:

```
#Mantel Test of across-year consistency of associations
library(ecodist)
#restrict comparison to individuals that were seen in two sequential years
id12=rownames(assocs[[1]])[rownames(assocs[[1]])%in%rownames(assocs[[2]])] #get IDs of birds
that were present in both networks
```

```
ids.m1=match(id12,rownames(assocs[[1]])) #get row/columns of those individuals in matrix 1
ids.m2=match(id12,rownames(assocs[[2]])) #get row/columns of those individuals in matrix 2

m12=assocs[[1]][ids.m1,ids.m1] #matrix 1 of association indices of only returning
individuals
m21=assocs[[2]][ids.m2,ids.m2] #matrix 2 of association indices of only returning
individuals

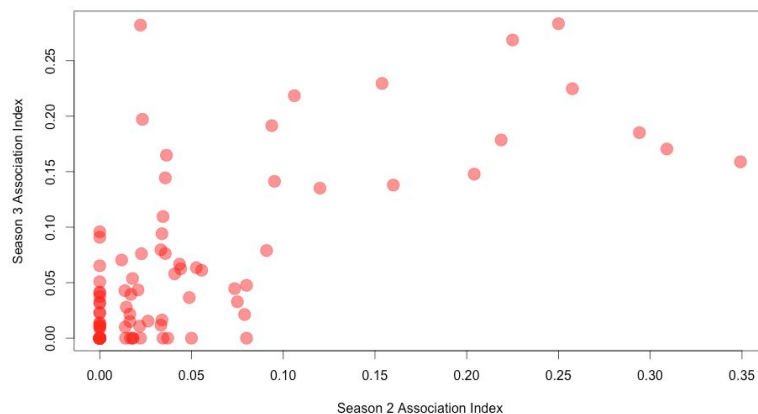
m12=m12[order(rownames(m12)),order(rownames(m12))] #re-order the rows/columns by
alphanumeric order
m21=m21[order(rownames(m21)),order(rownames(m21))] #re-order the rows/columns by
alphanumeric order

mantel12=mantel(as.dist(m12)~as.dist(m21))
mantel12

> mantel12
      mantelr      pval1      pval2      pval3  llim.2.5% ulim.97.5%
0.7281521  0.0010000  1.0000000  0.0010000  0.6412324  0.8379550
```

So the Mantel r coefficient = 0.73, a very high correlation of association indices across years. One way to visualize this is to simply plot the association indices from Season 2 against those of Season 3. Here, I'm only going to plot the values for the upper triangle of the adjacency matrix, since the matrix is symmetrical. I'm also going to use the `rgb()` function to make transparent red circles to plot so that we can see the overlapping points.

```
plot(m12[upper.tri(m12)], m21[upper.tri(m21)], pch=19, col=rgb(1,0,0,0.5), cex=2,
xlab="Season 2 Association Index", ylab="Season 3 Association Index")
```

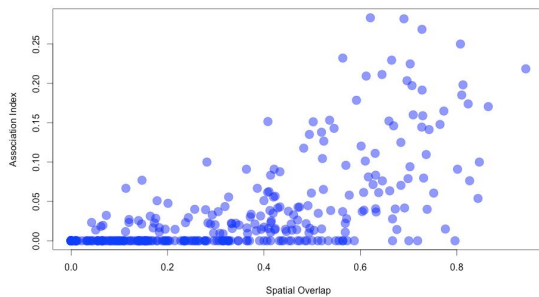


Using MRQAP to test for the effects of spatial overlap and previous year's association

The Mantel Test and the plot above shows us that there is significant correlation between the association indices across years--i.e., birds flock with the same individuals each year. However, this effect could simply be driven by the fact that these birds are faithful to their home ranges rather than their social associates. This is important because there is no doubt that spatial overlap influences social associations (it has to, since associations are measured as co-occurrence in time and space). For example, if we import a matrix of spatial overlap between each pair of each individual (measured as the proportion of joint home range that is shared) and plot the association index relative to this measure, we get this:

```
s3=as.matrix(read.csv("https://dshizuka.github.io/NAOC2016/GCSPspaceoverlap3.csv", header=T,
row.names=1))

plot(s3[upper.tri(s3)], assocs[[2]][upper.tri(assocs[[2]])], pch=19, cex=2, col=rgb(0, 0, 1,
0.5), xlab="Spatial Overlap", ylab="Association Index")
```



So how can we test for the effect of the previous year's association on the current year's association while taking into account the amount of spatial overlap in home ranges? This is where MRQAP (aka 'network regression') comes in. Here, we can use the 'spatial overlap matrix' and 'previous year's adjacency matrix' as covariates to test their effects on the 'current year adjacency matrix'. Briefly, the MRQAP with Double-Semipartialing (Krackhardt 1988; Dekker et al. 2007) is related to the Mantel Test, but permutes the residual matrix from each independent variable to calculate the effect of each variable on the response variable matrix. I recommend reading Dekker et al. (2007) for details on the method. We can implement MRQAP in 'asnipe' or 'sna' packages. Here, I will demonstrate its application with the 'asnipe' package.

First, let's make the spatial overlap matrix restricted to the individuals that are included in both seasons, and sorted in the same way as the adjacency matrices:

```
s3.match=s3[match(rownames(m21),rownames(s3)), match(rownames(m21),rownames(s3))] #sort rows
and columns to match the adjacency matrix
```

Now we have already made the relevant matrices: adjacency matrix of Season 3 (dependent variable), adjacency matrix of Season 2 (independent variable 1), spatial overlap matrix of Season 3 (independent variable 2), each with only birds that are in both seasons. We can now run the MRQAP model:

```
mrqap.dsp(m21~m12 + s3.match)
```

```
> mrqap.dsp(m21~m12 + s3.match)
MRQAP with Double-Semi-Partialing (DSP)
```

```
Formula: m21 ~ m12 + s3.match
```

```
Coefficients:
```

	Estimate	P($\beta > r$)	P($\beta \leq r$)	P(β <= r)
intercept	-0.01690381	0.05	0.95	0.051
m12	0.46892388	1.00	0.00	0.000
s3.match	0.12701098	1.00	0.00	0.000

```
Residual standard error: 0.04148 on 102 degrees of freedom
F-statistic: 106.9 on 2 and 102 degrees of freedom, p-value: 0
Multiple R-squared: 0.6771      Adjusted R-squared: 0.6708
AIC: -40.53804
```

You can see that both of the independent variables, spatial overlap (`s3.match`) and the adjacency matrix of the previous year (`m12`) are significant predictors of the current year's adjacency matrix. The overall fit of the model is very good (adjusted $R^2 = 0.67$), which suggests that these two variables explain a great deal of the variation in pairwise flock associations. Interestingly, the effect of the previous year's adjacency matrix is much stronger than that of the spatial overlap of home ranges. These results suggest that given a certain degree of spatial overlap, sparrows are more likely to flock together if they had flocked together in the previous year.

Worked Example: Experimentally inducing assortment in birds of Wytham Woods

This exercise will be based on a study by Firth & Sheldon (2015) using several species of birds in Wytham Woods, Oxford. This is a fantastic system in which great tits (*Parus major*), blue tits (*Cyanistes caeruleus*), marsh tits (*Poecile palustris*), coal tits (*Periparus ater*) and Eurasian nuthatches (*Sitta europaea*) have been tagged with RFID tags, and they are given access to feeders that are connected to RFID readers. In this study, Firth & Sheldon manipulated access to feeders based on an arbitrary criterion (odd vs. even-numbered RFID tags) and observed that this artificial segregation of co-feeding birds led to corresponding assortment of birds that feed at those sites. Moreover, this new segregation of co-feeding relationships bled over to other contexts ('ephemeral' open feeding sites and nestbox inspections). Here, we will take the opportunity to re-create a part of Figure 2 of this study, which plots the assortativity coefficient of the networks at different time periods and contexts and compares these values to the expected values based on permutation method called 'node-label permutation'.

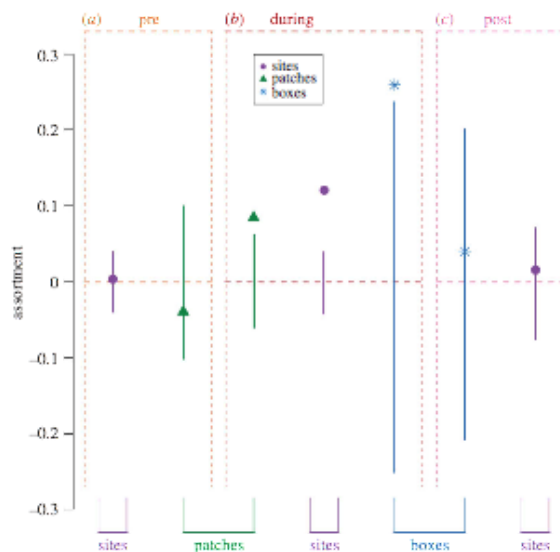


Figure 2. The observed level of assortment by PIT tag type in the system over the different periods and social contexts. Vertical lines show the 95% range of the assortment coefficients calculated from permuted data. Dots indicate the observed assortment coefficient. Colours of lines and point types illustrate data from different contexts (purple circle, selective feeder sites; green triangle, ephemeral patches; blue star, nest-boxes; base x axis). (a) 'Pre-manipulation' period. (b) 'During-manipulation' period. (c) 'Post-manipulation' period.

Figure 2 from Firth & Sheldon (2015)

Importing and using .RData file

To do this, we will first have to import the publicly-accessible dataset associated with the Firth & Sheldon (2015) study. This is available on the Dryad Data Repository ([direct link to dataset](https://doi.org/10.7554/dryad.71301)).

You can load R Data Files using the `load()` function. Here, we will just load it straight from the dryad website:

```
load(url("http://datadryad.org/bitstream/handle/10255/dryad.71301/figure2data_wholenetworks.RData"))
```

You should see that this loads several objects that the authors have compiled into a convenient packet:

- *description*: a simple written description of the data

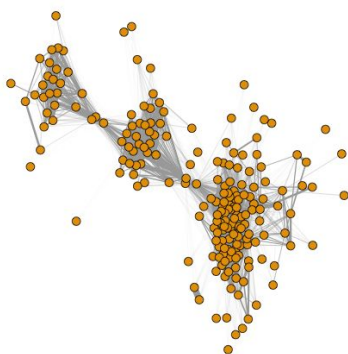
- *id.info*: a vector of 339 values, corresponding to each individual in the networks. The value is 1 if the individual has an odd-numbered tag and 0 if it has an even-numbered tag.
- *whole.networks*: a list object with 7 adjacency matrices. This corresponds to the 7 lines and dots in Figure 2 (though not in exact order).

The *whole.network* object is key here. Let's look at its structure using the `str()` function:

```
str(whole.networks)
> str(whole.networks)
List of 7
 $ pre.site.whole.network      : num [1:203, 1:203] 0 0.1869 0.0117 0.7084 0.4832 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:203] "260" "338" "115" "320" ...
  .. ..$ : chr [1:203] "260" "338" "115" "320" ...
 $ during.site.whole.network   : num [1:336, 1:336] 0 0.0899 0.0104 0.2398 0.2098 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:336] "260" "338" "115" "320" ...
  .. ..$ : chr [1:336] "260" "338" "115" "320" ...
 $ post.site.whole.network     : num [1:122, 1:122] 0 0 0 0.0791 0 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:122] "338" "289" "247" "326" ...
  .. ..$ : chr [1:122] "338" "289" "247" "326" ...
... [4 more matrices]...
```

So, there are 7 components and each component is a matrix of slightly different dimensions (e.g., 'pre.site.whole.network' is 203 x 203, while 'during.site.whole.network' is 336 x 336). Let's take a look at one of the networks. We'll take the first adjacency matrix (the pre-experimental network for the whole site), convert it into an igraph object, and then plot it:

```
library(igraph)
g.pre=graph_from_adjacency_matrix(whole.networks[[1]], mode="undirected", weighted=T)
plot(g.pre, edge.width=E(g.pre)$weight*10, vertex.label="", vertex.size=5)
```



Now, let's try to add whether the individual had an odd- or even-numbered tag. To do this, we have to link the *id.info* information with the node names. The node names in the adjacency matrices corresponds to the order of individuals in the *id.info* vector. What we will do is first create a separate data frame object that links the node name to the RFID tag type. We can then use this information to create a node attribute for the network by matching the node name with the id name in this data frame we created:

```
#create a data frame in which the first column is the "id number" and the second column is
1-odd or 0-even. **Note that we can name the columns within the data.frame() function
```

```
rfid.dat=data.frame(id=as.factor(1:339), oddeven=id.info)

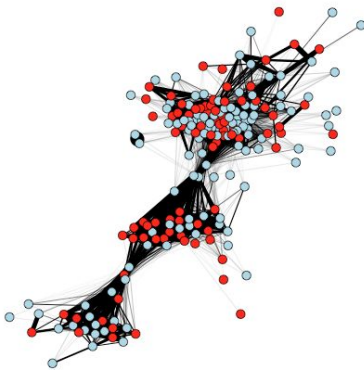
#look up the appropriate node name in the rfid.dat object and return the odd-or-even data.
V(g.pre)$oddeven=rfid.dat[match(V(g.pre)$name, rfid.dat$id), "oddeven"]
```

Take a look at the `V(g.pre)$oddeven` attribute now and see what this did.

Now, we can use this to create a separate “node color” attribute and use this to plot the network.

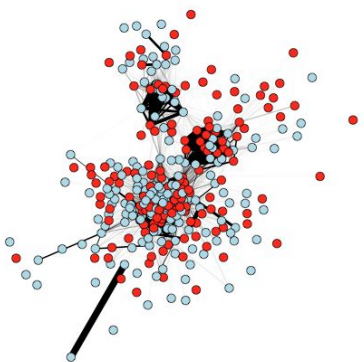
```
#create node color attribute based on odd-even. This code will make the node red for
even-numbered tags and light blue for odd-colored tags.
V(g.pre)$node.color=c("red", "lightblue")[V(g.pre)$oddeven+1]

#plot the network using the node colors
plot(g.pre, vertex.color=V(g.pre)$node.color, edge.width=E(g.pre)$weight*20,
vertex.label="", vertex.size=5, edge.color="black")
```



Try doing this with the other networks, such as the “during experiment” network (the second element in the *whole.networks* list object).

```
g.during=graph_from_adjacency_matrix(whole.networks[[2]], mode="undirected", weighted=T)
V(g.during)$oddeven=rfid.dat[match(V(g.during)$name, rfid.dat$id), "oddeven"]
V(g.during)$node.color=c("red", "lightblue")[V(g.during)$oddeven+1]
plot(g.during, vertex.color=V(g.during)$node.color, edge.width=E(g.during)$weight*20,
vertex.label="", vertex.size=5, edge.color="black")
```



Now, you might notice some visual differences in this network, but it’s actually quite difficult to get any sense of definitive differences here. We now want to directly measure the

Measuring Assortativity using the ‘assortnet’ package

Assortativity coefficient for both discrete and continuous traits (in weighted or unweighted networks) can be calculated using the R package ‘assortnet’ (Farine 2014).

First, let’s load the assortnet package and use the `assortment.discrete()` function to calculate the assortativity index based on odd vs. even RFID tag in the pre-treatment period and look at the result. Note that this function takes the adjacency matrix, not the igraph object:

```
library(assortnet)
#use the first adjacency matrix in the whole.networks list object--this is the pre-treatment
period network
r.pre=assortment.discrete(whole.networks[[1]], V(g.pre)$oddeven)
r.pre

> r.pre
$r
[1] 0.003459347

$mixing_matrix
      1      0      ai
1 0.3038673 0.2465973 0.5504646
0 0.2465973 0.2029381 0.4495354
bi 0.5504646 0.4495354 1.0000000
```

You can see that the `assortment.discrete()` function actually returns two elements: the *r* value (assortativity index) and the “mixing matrix”, which shows how the *r* value is calculated—we will actually not use the mixing matrix here.

If you want to just return the assortativity index:

```
r.pre$r

> r.pre$r
[1] 0.003459347
```

Now try this with the “during-treatment” network:

```
r.during=assortment.discrete(whole.networks[[2]], V(g.during)$oddeven)
r.during$r

> r.during$r
[1] 0.1211521
```

You can see that the assortativity coefficient is much higher during the experiment than the pre-experiment period. You should also see that these ‘pre-treatment’ and ‘during-treatment’ assortativity values correspond to the purple circles in Figure 2a and 2b, respectively.

Null Hypothesis Testing using node-label permutation

So far, we have calculated the assortativity coefficient of the bird social network in the pre- and during-treatment periods and showed that there is greater assortativity during the treatment. However, since there is no replication of the experiment, it is difficult to say whether or not this comparison is meaningful. What we would like to do is actually compare these values to the “null expectation”—the level of assortativity we would expect if the connections between node types were random. One potential way to ask whether the observed assortativity is non-random would be to compare the observed assortativity values with *random graphs* of the same size, density and frequency of node types. However, there are many reasons why comparing the observed network to random graphs may not be

very informative: since the pattern of connections in observed networks are almost always non-random to begin with, whatever difference between the observed and expected values may be due to the fact that patterns of connections are non-random with respect to factors that are unrelated to RFID tag numbers. A better way to ask whether the observed assortativity values are non-random would be to compare the empirical value against networks that have the exact same pattern of connections (i.e., network topology), but the node types are randomized across all nodes. This is what is called ‘node-label permutation’).

Implementing node-label permutation is not as difficult as you might think. The first thing you need to learn is how to “resample” a series of values. In our current case, we want to use a resampling (or *permuting*) procedure to randomly reassign each node to represent an odd- or even-numbered tag. You can do this by using the function `sample()`. You can see what this has done by comparing the actual RFID type of the first 6 nodes and compare it with their **permuted** values:

```
s=sample(V(g.pre)$oddeven, length(V(g.pre)), replace=F) #we 'resample without replacement'
the odd/even values for nodes.
head(V(g.pre)$oddeven) #first 6 values of odd/even types for the empirical data
head(s) #first 6 values of odd/even types for the permuted data

> head(V(g.pre)$oddeven) #first 6 values of odd/even types for the empirical data
[1] 1 0 0 1 0 0
> head(s) #first 6 values of odd/even types for the permuted data
[1] 1 1 1 0 0 0
```

Note that the resampled values `s` will be different each time you run the above `sample()` function. Now, lets measure the assortativity index of this “node-label permuted” network. We can do this simply by using the same adjacency matrix, but now using the permuted version of node RFID types:

```
assortment.discrete(whole.networks[[1]], s)$r

> assortment.discrete(whole.networks[[1]], s)$r
[1] 0.02117892
```

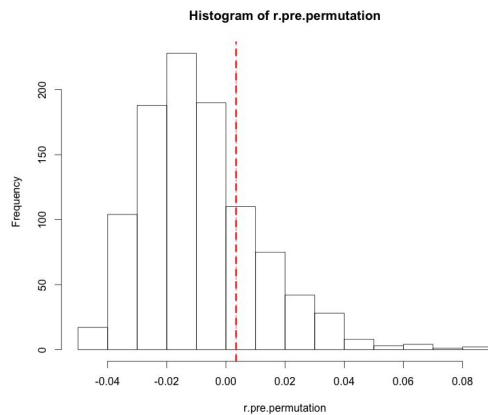
What we have done here is a single iteration of the permutation procedure. To do a proper statistical test, we would like to repeat this permutation a large number of iterations—say 10,000 times—and compare the distribution of assortativity values of these permuted networks with the observed assortativity value. Using this procedure, we can calculate a ***p-value of the observed assortativity index as the probability that the observed assortativity value falls within the distribution of assortativity values from the permuted networks.***

We can use a loop function to repeat this permutation procedure a large number of times. Here, we will just do 1000 permutations. We will do this permutation procedure for the “pre-treatment” network and store the resulting assortativity values into a vector called ***r.pre.permutation***:

```
t=1000
r.pre.permutation=vector(length=t)
for (i in 1:t){
  s=sample(V(g.pre)$oddeven, length(V(g.pre)), replace=F)
  r.pre.permutation[i]=assortment.discrete(whole.networks[[1]], s)$r
}
```

The resulting `r.pre.permutation` object will now contain 1000 values. You can plot this as a histogram to visualize the data. In addition, you can draw a vertical line corresponding to the observed assortativity value and see where it falls in the distribution of these values:

```
hist(r.pre.permutation)
abline(v=r.pre$r, lty=2, col="red", lwd=3) #arguments: lty = line type (2 = dashed line);
col = color; lwd = line width
```



You can visually tell that the observed assortativity value falls well within the distribution of the permuted assortativity values. The corresponding p-value based on a one-tailed test would be the number² of times the permuted values exceeded the observed value (`r.pre$r`), divided the number of permutations²:

```
p.pre=(length(which(r.pre.permutation>r.pre$r))+1)/(t+1)
p.pre
```

```
> p.pre
[1] 0.2067932
```

Ok, let's repeat this exercise for the “during-treatment” network:

```
t=1000
r.during.permutation=vector(length=t)
for (i in 1:t){
  s=sample(V(g.during)$oddeven, length(V(g.during)$oddeven), replace=F)
  r.during.permutation[i]=assortment.discrete(whole.networks[[2]], s)$r
}

hist(r.during.permutation, xlim=c(-0.05, 0.15)) #NOTE: you will need to use the xlim=
argument to adjust the x-axis limits so that the empirical assortativity value will show up!
abline(v=r.during$r, lty=2, col="red", lwd=3)
```

```
p.during=(length(which(r.during.permutation>r.during$r))+1)/(t+1)
p.during
```

```
> p.during
[1] 0.000999001
```

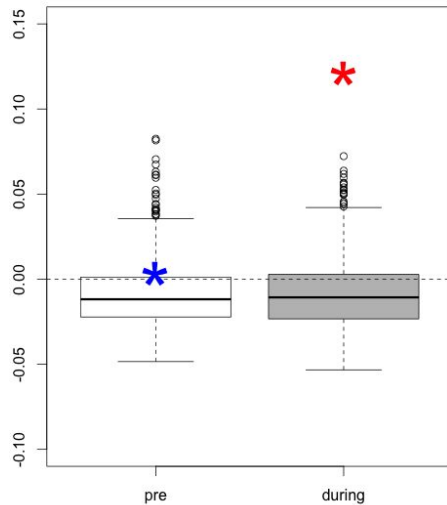
Finally, let's make the results look a bit more like the Figure 2 in the Firth & Sheldon (2015) study by using a boxplots to show the distribution of permuted values and colored asterisks to show the observed values:

² Note: some people prefer to calculate p-value from permutations by adding 1 to both numerator and denominator. There are some theoretical arguments for that. I will skip that here.

```

boxplot(r.pre.permutation, r.during.permutation, col=c("white", "gray"), ylim=c(-0.1, 0.15),
names=c("pre", "during"))
abline(h=0, lty=2)
points(1,r.pre$r, pch="*", cex=5, col="blue")
points(2,r.during$r, pch="*", cex=5, col="red")

```



Basically, this tells us that the distribution of the expected values of assortativity index is similar in the pre- and during-treatment networks, but the observed value (red asterisk) far exceeds the expected values during the treatment. This is evidence that the experimental treatment caused assortment by RFID tag type.

Bonus material: The following codes will repeat the analyses in the post-treatment period and use the package ‘ggplot2’ to generate a set of plots that look much more like the Figure 2 from the published paper. This will generate the plots just for the “selective feeder sites” (purple dots and lines) from the figure:

```

#create the post-treatment network and calculate assortativity
g.post=graph_from_adjacency_matrix(whole.networks[[3]], mode="undirected", weighted=T)
V(g.post)$oddeven=rfid.dat[match(V(g.post)$name, rfid.dat$id), "oddeven"]
r.post=assortment.discrete(whole.networks[[3]], V(g.post)$oddeven)
r.post$r

#node-label permutation of the post-treatment network
t=1000
r.post.permutation=vector(length=t)
for (i in 1:t){
  s=sample(V(g.post)$oddeven, length(V(g.post)$oddeven), replace=F)
  r.post.permutation[i]=assortment.discrete(whole.networks[[3]], s)$r
}

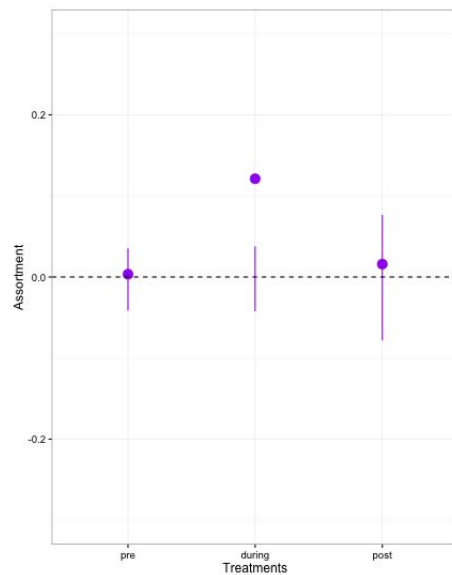
#calculate the lower and upper 95% confidence interval
pre.quant=quantile(r.pre.permutation, c(0.025, 0.975))
during.quant=quantile(r.during.permutation, c(0.025, 0.975))
post.quant=quantile(r.post.permutation, c(0.025, 0.975))

#load ggplot2, set up the dataframe and plot

```

```
library(ggplot2)
df=data.frame(Treatments=1:3, lower=c(pre.quant[1], during.quant[1], post.quant[1]),
upper=c(pre.quant[2], during.quant[2], post.quant[2]), Assortment=c(r.pre$r, r.during$r,
r.post$r), treatments=c("pre", "during", "post"))

ggplot(df, aes(x = Treatments, y = Assortment)) +
  geom_linerange(aes(ymax = upper, ymin = lower), colour="purple") +
  theme_bw() +
  geom_point(colour="purple", size=4) +
  scale_x_discrete(limits=c("pre", "during", "post")) +
  scale_y_continuous(limits=c(-0.3, 0.3)) +
  geom_hline(aes(yintercept=0), linetype="dashed")
```



Worked Example: Diffusion of Cultural Innovations in Networks³



Social networks describe the patterns of associations and interactions between individuals. Increasingly, people have used networks to understand how these associations between individuals facilitate the transmission of information, disease and socially learned behaviors within a population. These types of analyses have shown that social networks influence the spread of obesity (Christakis & Fowler 2007) and happiness (Fowler & Christakis 2008), and emotions (Hill et al. 2010) in human societies. In animal societies, recent studies have demonstrated social transmission of novel feeding modes in humpback whales (Allen et al. 2013) and chimpanzees (Hobaiter et al. 2014), and Aplin et al. (2015) showed the experimentally induced feeding innovations can be socially transmitted rapidly in a population of great tits. There are several different methods available to detect the spread of things in networks. Here, we will first show how to simulate social transmission on networks, briefly introduce one statistical method to detect social transmission (Network-Based Diffusion Analysis), and demonstrate how to visualize social transmission on a network using data from a study on humpback whales (Allen et al. 2013).

Simulating asocial and social spread of behavior on a network

Scenario 1: Asocial Model. Everyone adopts the innovation on their own. Nodes vary in probability to adopt.

We will first consider how a new behavior may spread in a population WITHOUT social transmission if each individual has an inherent probability to adopt the innovation at any given time. We can track the number of individuals that adopt the innovation across time. We start by using a random number generator to assign individual probabilities of adoption at any given time:

```
x=runif(100, min=0, max=1) #each individual's probability of adoption at any given time.  
(100 random numbers between 0 and 1)
```

³ This module is based on data and analysis from:

Franz, M. & Nunn, C. 2009. Network-based diffusion analysis: a new method for detecting social learning.

Hoppitt, W., Boogert, N. J. & Laland, K. N. 2010. Detecting social transmission in networks. *Journal of Theoretical Biology*, **263**, 544–555.

Allen, J., Weinrich, M., Hoppitt, W., & Rendell, L. (2013). Network-based diffusion analysis reveals cultural transmission of lobtail feeding in humpback whales. *Multiple Values Selected*, 340(6131), 485–488. <http://doi.org/10.1126/science.1231976#>

Now, we will set the initial ‘innovation adoption’ status for all individuals (all initially 0) and also create an empty list where we will store the identities of individuals that adopt at each time step. We then start a for loop that has several steps:

Step 1: Set the probability of adoption to 0 if the individual has already adopted (can’t re-adopt the innovation).

Step 2: Flip a coin based on the probabilities established above and determine if the individual adopts the innovation or not.

Step 3: Change the status of the individuals that did adopt.

Step 4: Save the identities of the new adopters.

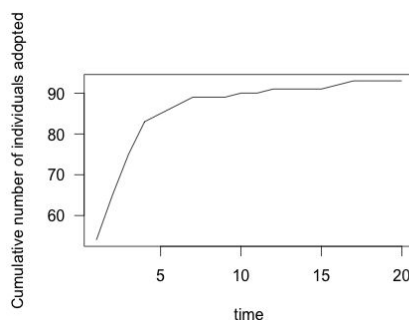
Repeat Steps 1 through 4 x20 times

```
status=rep(0,100) #the 'innovation adoption status' for each individual. All initially 0.
adopt.list=list() #empty list
t=20
for (j in 1:t){
  p=x*abs(status-1) #multiply the probability of adoption by 0 if already adopted, 1
  #otherwise.
  adopters=sapply(p, function(x) sample(c(1,0),1, prob=c(x, 1-x))) #based on the
  #probabilities, flip a coin and determine if the individual adopted or not.
  status[which(adopters==1)]=1 #change status of new adopters
  adopt.list[[j]]=which(adopters==1) #save the identities of adopters for that time
  step
}
```

The end result will be a list object that shows the individuals that adopted the innovation at each of t=20 time steps:

We can use the `sapply()` function to count the number of individuals adopted the innovation at each time step and plot a cumulative frequency of adopters across time.

```
n.adopt.asocial=sapply(adopt.list, length)
cumulative.asocial=cumsum(n.adopt.asocial)
plot(cumulative.asocial, type="l", las=1, ylab="Cumulative number of individuals adopted",
xlab="Time")
```



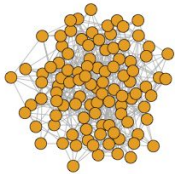
You can see that the result is a decelerating curve—the rate of adoption decreases as the number of adoptees increase.

Scenario 2: Social Model. Innovation spreads purely by social learning in a random network.

Let's consider a hypothetical case of diffusion of innovation on a network. It can be any innovation—a new feeding mode, a new technology, etc.

Let's start by creating a random graph:

```
library(igraph)
g=erdos.renyi.game(100, p=0.1) #create a Erdos-Renyi Random Graph with uniform edge
probability = 0.1
plot(g, vertex.label="")
```

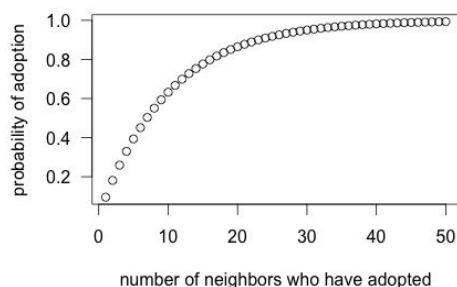


We are now going to work on simulating a case where 2 random individuals will adopt an innovation. For the rest of the nodes, the probability that a node will adopt this innovation in any given time step increases by a certain amount for each 'neighbor' that has adopted that innovation. We'll call this the 'social influence' parameter, s ($0 < s < 1$). We will define the probability that node i that has not yet adopted the innovation will adopt it as:

$$P_i = 1 - \exp(-s * n) \quad \text{equation (1)}$$

where n is the number of neighbors of node i that have already adopted the innovation. This results in an asymptotic increase in the probability of adoption as the number of adopter neighbors increases. For illustration, let's plot what this probability function looks like when $s = 0.1$:

```
nei.adopt=1:50
s=0.1
prob=(1-exp(-s*nei.adopt))
plot(nei.adopt, prob, type="b", las=1, ylab="probability of adoption", xlab="number of
neighbors who have adopted")
```



Try playing around with the value of s and replotting this curve to get a sense for how the social influence parameter works in our simulation.

Quick side note: How to count number of neighbors who have adopted the innovation

To move forward from here, we first need to learn a new function called `neighbors()`. This function shows us which nodes are connected to a particular node. So, if we wanted to find out which nodes are connected to node 1 in our random graph:

```
neighbors(g, 1)
+ 10/100 vertices:
[1] 10 12 17 27 44 67 71 77 83 84
```

Your output will look different than mine. We will use this trick to simulate social diffusion based on the number of neighbors that have adopted the innovation. Ok, now we are ready to set up our simulation.

To begin, we will do several things: We will (1) set the social influence parameter (*s*) value, (2) define the number of time steps we want to track, (3) create a vertex attribute to track the ‘adoption status’ of each node, and (4) assign two random nodes to be the first adopters of the innovation.

```
t=20 #time steps to run the simulation
s=0.1 #social influence parameter
V(g)$status=rep(0, vcount(g)) # Create a vertex attribute for adoption status. 1 if the node
has adopted the innovation. 0 if not.
seed=sample(1:vcount(g),2) #select 2 innovators
V(g)$status[seed]=1 #These 'seed' individuals get a status of 1 at the beginning.
```

We will now set up an empty list where the identity of adopters for each time step will go. Then, we will set up a loop function with several steps:

- Step 1: For each node, count the number of neighbors who have already adopted the innovation
 - Step 2: Calculate the probability that each node will adopt the innovation in this time step (only for nodes that have not yet adopted).
 - Step 3: Use the probability to flip a coin and determine if the node adopted the innovation or not
 - Step 4: For those that adopted the innovation, set their status as adopted
 - Step 5: Save the identities of the adopters for this time step.
- Repeat Steps 1-5 x20 times

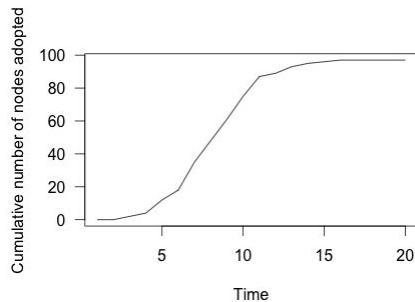
```
adopt.list=list(0) #empty list

for (j in 1:t){
  nei.adopt=sapply(V(g), function(x) sum(V(g)$status[neighbors(g,x)]))
  p=(1-exp(-s*nei.adopt))*abs(V(g)$status-1) ##here, we multiply the probabilities by 0 if
  node is already adopted, and 1 if not yet adopted
  adopters=sapply(p, function(x) sample(c(1,0), 1, prob=c(x, 1-x)))
  V(g)$status[which(adopters==1)]=1
  adopt.list[[j]]=which(adopters==1)
}
```

Now, you will end with a list object called `adopt.list`, which lists the nodes that adopted the innovation in each time step.

Let’s plot the cumulative number of individuals that adopt the innovation across time:

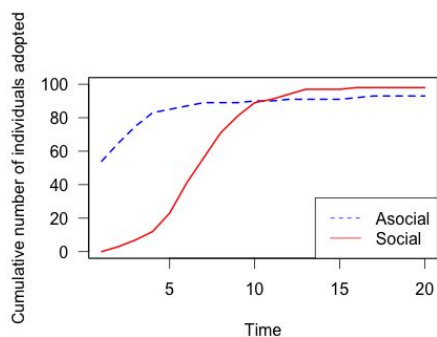
```
n.adopt.social=sapply(adopt.list, length) #for each time step, count the number of adopters.
cumulative.social=cumsum(n.adopt.social)
plot(cumulative.social, type="l", las=1, ylab="Cumulative number of nodes adopted",
xlab="Time")
```



Your figure will look a bit different, but it should generally show this S-shaped curve.

Now let's plot the asocial and social models together and make it pretty:

```
plot(cumulative.asocial, type="l", las=1, ylim=c(0,100),ylab="Cumulative number of
individuals adopted", xlab="Time", col="blue", lty=2, lwd=2)
par(new=T)
plot(cumulative.social, type="l", las=1, ylim=c(0,100),ylab="", xlab="", yaxt="n", xaxt="n",
col="red", lty=1, lwd=2)
legend("bottomright", lty=c(2,1), col=c("blue", "red"), legend=c("Asocial", "Social"))
```



This example nicely illustrates the classic idea behind social learning or social adoption of innovations: social learning/adoption generates an accelerating curve of adoption, while asocial learning/adoption leads to a non-accelerating (and often decelerating) curve (Reader 2004).

BUT WAIT!

Note that this entire exercise was based on a RANDOM NETWORK. But we know that no real social network follows an Erdos-Renyi random graph model. Real networks tend to be more heterogeneous (e.g., there is a wider range of node degrees), they tend to be clustered, etc. So does this general pattern of 'diffusion curves' hold up in real networks? As you might imagine, the answer is NO. What the diffusion curve will look like is highly dependent on network structure AND who the initial innovators are (among other important factors).

Scenario 3: Diffusion of innovation in a real network

To illustrate how the diffusion dynamics differ between random and real networks, we will repeat the social model from above on a social network that is available in the 'igraphdata' package.

BUT FIRST, to make things a little bit easier going forward, we are going to begin by converting the simulation codes we wrote above into a custom function. The goal is to write a function such that we just

need to specify several parameters (the social network, the ‘seed’ individuals, # times steps and the social influence parameter, s) and run the simulation. Here is the script:

```
run_simulation=function(graph, seed, time, s){
  V(graph)$status=rep(0, vcount(graph))
  V(graph)$status[seed]=1
  results=list(0)
  for (j in 1:time){
    nei.adopt=sapply(V(graph),function(x) sum(V(graph)$status[neighbors(graph,x)]))
    p=(1-exp(-s*nei.adopt))*abs(V(graph)$status-1)
    adopters=sapply(p, function(x) sample(c(1,0), 1, prob=c(x, 1-x)))
    V(graph)$status[which(adopters==1)]=1
    results[[j]]=which(adopters==1)
  }
  return(results)
}
```

The above script will establish a function called `run_simulation()`, and all we can apply the simulation to various networks and starting conditions. The function will return the list of individuals that adopted the innovation at each time step.

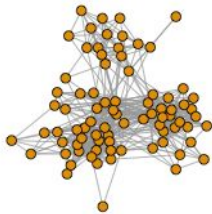
For example, this should more or less re-create the social diffusion model we tried above in three lines of code:

```
adopt.list=run_simulation(graph=g, seed=sample(1:100,2), time=20, s=0.1)
n.adopt=sapply(adopt.list, length)
plot(cumsum(n.adopt), type="l", las=1, ylab="cumulative nodes adopted", xlab="time")
```

Now that we have established this useful function, let’s run the simulation of social diffusion again, but this time using a real network. For this, we will use the “UKfaculty” network in the “igraphdata” package. This is a friendship network of a UK university. This network shows the classic characteristics of a social network such as community structure and high clustering coefficients. Since this network is directed, let’s make a simpler undirected version and plot it:

```
library(igraphdata)
data(UKfaculty)

uk=as.undirected(UKfaculty)
plot(uk, vertex.label="", vertex.size=10)
```



Now you can try running the diffusion simulation on this network. You can play around with the parameters too. The code below will run the simulation with two random nodes as the seed individuals and the s parameter set slightly lower at 0.05.

```
uk.adopt=run_simulation(graph=uk, seed=sample(1:vcount(uk),2), time=20, s=0.05)
n.adopt.uk=sapply(uk.adopt, length)
```

```
plot(cumsum(n.adopt.uk), type="l", las=1, ylim=c(0,81))
```

Try running this a few times. You should see that the accumulation curve will look slightly different each time. One of the reasons is because we are randomly choosing the ‘seed’ individuals--but who exactly starts the innovation actually matters a lot in clustered networks! We will now explore this dynamic.

Where innovation starts matters

We will now use this UK faculty network to illustrate how the starting point of innovations may matter, particularly when the network is heterogeneous and non-random. What we will do is run the simulation two times. In the first run, we will initiate the diffusion process from the two lowest-degree nodes. In the second run, we will initiate the diffusion from the two highest-degree nodes. We will then plot the diffusion curves on the same figure and see how different they look.

First, here is how you find the two lowest and two highest degree nodes:

```
low.degree=order(degree(uk), decreasing=F)[1:2]
high.degree=order(degree(uk), decreasing=T)[1:2]
```

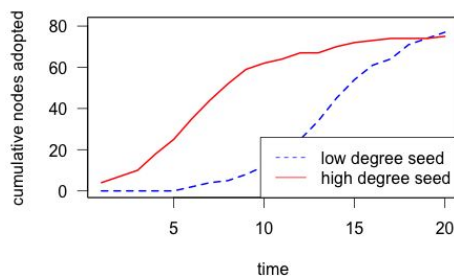
Now we can run the simulation two times, saving the list of adopters separately. We can then plot the results together:

```
adopt.list1=run_simulation(graph=uk, seed=low.degree, time=20, s=0.05)
n.adopt1=sapply(adopt.list1, length)

adopt.list2=run_simulation(graph=uk, seed=high.degree, time=20, s=0.05)
n.adopt2=sapply(adopt.list2, length)

plot(cumsum(n.adopt1), type="l", las=1, ylim=c(0,81),ylab="cumulative nodes adopted",
xlab="time", col="blue", lty=2, lwd=2)
par(new=T)
plot(cumsum(n.adopt2), type="l", las=1, ylim=c(0,81),ylab="", xlab="", col="red", xaxt="n",
yaxt="n", lty=1, lwd=2)
legend("bottomright", lty=c(2,1),col=c("blue","red"), legend=c("low degree seed", "high
degree seed"))
```

The exact result will vary each time you run this because it is a stochastic process. However, more often than not, you should get a result in which innovations spread faster when it is started by a high-degree node, like this:



Network-Based Diffusion Analysis (NBDA)

One of the take-home messages of the simulation exercises is that the social learning can lead to different patterns, and thus it can be difficult to tell between social and asocial learning from these diffusion curves alone. To address this problem, Franz & Nunn. (2009) proposed a Network-Based Diffusion Analysis (NBDA) method to detect social learning based on timing of adoption and network connections. Briefly, one takes the data on who adopts or does not adopt a new behavior (e.g., new feeding strategy) at each

event (each individual for each time step: time step is determined by the researcher), and fits this data to two models: an *asocial model* in which all naïve individuals have equal probability of adoption the behavior and a *social model* in which the probability of adoption depend on the number of network neighbors have adopted the innovation (following equation 1 above). The log-likelihoods of model fits are summed for all time steps and compared using an model selection (AIC) method. If the behavior spreads through social transmission, then the social model will be a better fit to the data (i.e., will have lower AIC value).

Hoppitt et al. (2010) extended the NBDA framework for using the order of adoption rather than the time step of adoption (referred to as OADA and TADA respectively, for Order of Acquisition Diffusion Analysis and Time of Acquisition Diffusion Analysis). Whalen & Franz (2016) presents a Bayesian framework for applying NBDA with random effects. Both Franz & Nunn (2009) and Hoppitt et al. (2010) both have extended supplementary materials that are highly recommended for any potential users.

Here, I will demonstrate the basic application of Franz & Nunn's (2009) R codes for implementing NBDA for time of acquisition data using the simulations presented above. What we will do is simulate diffusion of a behavior using the model, and then apply the NBDA codes to determine whether we can recover the specific social transmission parameter that we set in the simulation.

Note that the script here requires you to load a set of functions that were written by Franz & Nunn. The supplemental materials for the paper is available here:

<http://rspb.royalsocietypublishing.org/content/276/1663/1829.figures-only>

What I have done here is copied and pasted these codes into an R script and made it available to directly read using the function `source()`. If you want to look at that R script directly, you can simply copy and paste the url inside the `source()` function into a browser.

Here is the script to run the simulation and run NBDA a la Franz & Nunn (2009):

```
## NBDA using Franz method
##load functions from R code in Franz & Nunn supplemental materials
source(url("https://dshizuka.github.io/NAOC2016/Franz_functions.R"))
#simulate social diffusion with small network

g=erdos.renyi.game(30, p=0.1) #create a random network
V(g)$status=rep(0, vcount(g)) # Create a vertex attribute for adoption status.
seed=sample(1:vcount(g),2) #select 2 random innovators
V(g)$status[seed]=1 #These 'seed' individuals get a status of 1 at the beginning.

#set up & run simulation to get list of adoptees for each time step
t=20 #number of time steps
seed=sample(1:vcount(g),2) # two random innovators
s=0.5 # the social transmission parameter
adopt.list.sim=run_simulation(graph=g, seed=seed, time=t, s=0.5) #run simulation

#convert the simulation result to a time-of-acquisition vector
ta=vector(length=vcount(g))
for(i in 1:length(adopt.list.sim)){
  ta[adopt.list.sim[[i]]]=i
}
ta #result
ta[ta==0]=t+1 #individuals that did not adopt get max time + 1 as 'time of acquisition'
ta[seed]=0 #seed individual are set to 0

#set up data for NBDA
sn.sim=as_adjacency_matrix(g, sparse=F) #adjacency matrix
```

```
diff.sim=as.matrix(ta) #time of acquisition data as a vertical matrix format

#other parameters required to run the nbda() function
tau_max <- 1 # maximum value for parameter tau in the social learning model
time_max <- t # maximum time steps
#run model
res.sim <- nbda(social_network = sn.sim, diffusion_data = diff.sim, tau_max = tau_max,
time_max = time_max)

# display results
res.sim

> res.sim
```

	AIC	Akaike_weight	Parameter_estimate
social learning model	80.89454	1	0.46561495
asocial learning model	159.09891	0	0.08539234

The results will differ each time the simulation, but the social parameter estimate for the social learning model (in bold) should be somewhere around 0.5. Sometimes the parameter estimate will be off, but I believe this is due to lack of power from running the simulation on a small network.

Visualizing the social transmission of lobtail feeding

In 1980, researchers first observed a new feeding strategy called ‘lobtail feeding’ used by a single humpback whale in the Gulf of Maine. Allen et al. (2013) used observation data from this population over the subsequent 27 years to show that this feeding innovation has spread through social transmission, and now up to 40% of the population are known to use this tactic. Here, we will demonstrate how to visualize the spread of ‘lobtail feeding’ among humpback whales by creating a GIF animation.

You can get the data from Allent et al. (2013) as a .zip file here:

<http://science.sciencemag.org/content/suppl/2013/04/24/340.6131.485.DC1>

Save it somewhere to access the .Rdata file we will use. Or you can just follow the link in this line of code to access my downloaded copy:

```
load(url("https://dshizuka.github.io/NAOC2016/Allen2013_SM.RData"))
```

If you read the associated readme file (read it [here](#)), you will get a sense for the three main files we will use. The `assocDat20` object is a big adjacency matrix (653 x 653). The `supp20` object is a dataframe with information on individual ID, covariates like number of time sighted, etc. and the `tadaDat20` object is a dataframe with timing information on individuals that were seen lobtail feeding.

Let’s first plot the cumulative numbers of whales that are known to use lobtail feeding. What we first want to do is to convert the date each whale was first observed to use lobtail feeding (if at all) into a “date” format object. This makes it easier for us to plot the patterns at different timescales (e.g., by years or months, etc.). For simplicity, we will then convert this data to years, and then plot the cumulative number of whales that lobtail feed across years of observation. Try out the codes below (outputs not shown).

```
#Date as entered in data
tadaDat20$DateFirstFR

# Date as "Date object"
dates=as.Date(tadaDat20$DateFirstFR, "%d/%m/%Y")
```

```

dates

# dates as years only
years.lobtail=as.numeric(format(dates, "%Y"))
years.lobtail

```

Now, we can use this data to create a network plot with the nodes that are ‘lobtail feeders’ colored differently. The nodes that are colored in will be different for each year. Let’s say we want to create a network that shows the known lobtail feeders in the year 2000:

First, create the network:

```

g=graph_from_adjacency_matrix(assocDat20, "undirected", weighted=T, diag=F)

```

Now, let’s find the individuals that are known to lobtail feed by year 2000 (show only first 6 individuals):

```

year=2000
v.ids=tadaDat20[which(years.lobtail<=year),"ID"]
head(v.ids)

```

```

> head(v.ids)
[1] "BL1" "SA1" "SC1" "MO1" "NI1" "LA4"

```

We can match up these IDs with the vertex names. The match() function will return NA for the nodes that do not match up (output not shown):

```

match(as.character(supp20$ID), v.ids) #match function returns a number if find v.ids in the
list of node IDs, NA if not

```

We can use this to create a vertex color attribute, which will be “gray” if the above match function returns NA and “red” if otherwise:

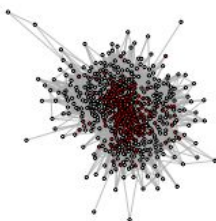
```

V(g)$color=ifelse(is.na(match(as.character(supp20$ID), v.ids)), "gray", "red") #converts
this into node colors we want

plot(g, layout=lo, vertex.label="", vertex.size=3, main="2000") #known lobtail feeders in
red.

```

2000



Now, we can use this method to create a GIF animation file in one of two ways: (1) create a series of .png files that can then be converted to a .gif file using an online GIF maker (e.g., <http://gifmaker.me/>), or (2) use the ‘animation’ package to directly create a .gif file. Note, however, that the ‘animation’ package requires that you also install an additional software called ImageMagick (<http://imagemagick.org/>). You

likely do not have this installed on your computer during the workshop, so you can try this at home if you are interested. I will present the scripts for both methods here:

Option (1): Create a series of .png files and save it to your working directory. You can then drag these to a GIF maker.

```
#print figures into a series of .png files.
setwd("~/Dropbox/Dai_Research/R Output") #make sure you know where the files are going
g=graph_from_adjacency_matrix(assocDat20, "undirected", weighted=T, diag=F)
years=seq(1979,2007,1) #set up the years
for(i in 1:length(years)){
  png(paste(years[i], ".png", sep="")) #sets up a file, i.e. "plotting device"
  v.ids=tadaDat20[which(years.lobtail<=years[i]),"ID"]
  V(g)$color=ifelse(is.na(match(as.character(supp20$ID), v.ids)), "gray", "red")
  plot(g, layout=lo, vertex.label="", vertex.size=3, main=years[i])
  dev.off()
}
```

Option (2): Create a .gif file directly using 'animation' package.

```
#using animation package. Your computer needs ImageMagick to run this
library(animation)
year.colors=list(0)
for (i in 1:length(years)){
  v.ids=tadaDat20[which(years.lobtail<years[i]),"ID"]
  year.colors[[i]]=ifelse(is.na(match(as.character(supp20$ID),v.ids))==FALSE, "red", "gray")
}

saveGIF( {for (i in 1:length(years)) plot(g, layout=lo, vertex.label="", vertex.size=3,
vertex.color=year.colors[[i]], main=paste(years[i]))},
movie.name="lobtail.networks.years.class.gif", interval=1, nmax=30, ani.width=600,
ani.height=600)
```

You can see the result here: <https://dshizuka.github.io/NAOC2016/lobtail.network.gif>

REFERENCES

- Allen, J., Weinrich, M., Hoppitt, W. & Rendell, L. 2013. Network-based diffusion analysis reveals cultural transmission of lobtail feeding in humpback whales. *Multiple values selected*, 340, 485–488.
- Anderson CJ, Wasserman S, Crouch B. 1999. A p* primer: logit models for social networks. *Soc Networks*. 21:37–66.
- Aplin, L. M., Farine, D. R., Morand-Ferron, J., Cockburn, A., Thornton, A. & Sheldon, B. C. 2015. Experimentally induced innovations lead to persistent culture via conformity in wild birds. *Nature*, 518, 538–541.
- Bascompte, J., & Jordano, P. (2007). Plant-Animal Mutualistic Networks: The Architecture of Biodiversity. *Annual Review of Ecology, Evolution, and Systematics*, 38(1), 567–593. <http://doi.org/10.1146/annurev.ecolsys.38.091206.095818>
- Bastolla, U., Fortuna, M. A., Pascual-García, A., Ferrera, A., Luque, B., & Bascompte, J. (2009). The architecture of mutualistic networks minimizes competition and increases biodiversity. *Nature*, 458(7241), 1018–1020. <http://doi.org/10.1038/nature07950>
- Bejder, L., Fletcher, D., & Bräger, S. (1998). A method for testing association patterns of social animals. *Animal Behaviour*, 56(3), 719–725. <http://doi.org/10.1006/anbe.1998.0802>
- Bunn, A. G., Urban, D. L., & Keitt, T. H. (2000). Landscape connectivity: a conservation application of graph theory. *Journal of environmental management*, 59(4), 265–278.
- Burkle, L. A., Marlin, J. C., & Knight, T. M. (2013). Plant-pollinator interactions over 120 years: loss of species, co-occurrence, and function. *Science*. <http://doi.org/10.1126/science.1230200>
- Cairns, S. J., & Schwager, S. J. (1987). A comparison of association indices. *Animal Behaviour*. 35, 1454–1469.
- Clauset, A., Newman, M. & Moore, C. 2004. Finding community structure in very large networks. *Physical Review E*, 70, 66111.
- Christakis, N. A. & Fowler, J. H. 2007. The spread of obesity in a large social network over 32 years. *New England Journal of Medicine*, 357, 370–379.
- Csardi, Gabor, and Tamas Nepusz. "The igraph software package for complex network research." *InterJournal, Complex Systems* 1695.5 (2006): 1-9.
- Croft, D. P., James, R., & Krause, J. (2008). *Exploring Animal Social Networks*. Princeton: Princeton University Press.
- Croft, D. P., Madden, J. R., Franks, D. W., & James, R. (2011). Hypothesis testing in animal social networks. *Trends in Ecology and Evolution*, 26(10), 502–507. <http://doi.org/10.1016/j.tree.2011.05.012>

Davidson, R. and Harel, D.: Drawing Graphs Nicely Using Simulated Annealing. *ACM Transactions on Graphics* 15(4), pp. 301–331, 1996.

Dekker, D., Krackhardt, D., & Snijders, T. A. B. (2007). Sensitivity of MRQAP Tests to Collinearity and Autocorrelation Conditions. *Psychometrika*, 72(4), 563–581. <http://doi.org/10.1007/s11336-007-9016-1>

Dunne, J. A., Williams, R. J., & Martinez, N. D. (2002). Food-web structure and network theory: the role of connectance and size. *Proceedings of the National Academy of Sciences of the United States of America*, 99(20), 12917–12922.

Dyer, R. J. (2015). Population Graphs and Landscape Genetics. *Annual Review of Ecology, Evolution, and Systematics*, 46(1), 327–342. <http://doi.org/10.1146/annurev-ecolsys-112414-054150>

Dyer, R. J., & Nason, J. D. (2004). Population Graphs: the graph theoretic shape of genetic structure. *Molecular Ecology*, 13(7), 1713–1727. <http://doi.org/10.1111/j.1365-294X.2004.02177.x>

Fall, A., FORTIN, M.-J., Manseau, M., & O'Brien, D. (2007). Spatial Graphs: Principles and Applications for Habitat Connectivity. *Ecosystems*, 10(3), 448–461. <http://doi.org/10.1007/s10021-007-9038-7>

Farine, D. R. (2014). Measuring phenotypic assortment in animal social networks: weighted associations are more robust than binary edges. *Animal Behaviour*, 89(C), 141–153.

Farine, D. R., & Whitehead, H. (2015). Constructing, conducting and interpreting animal social network analysis. *Journal of Animal Ecology*. 84, 1144–1163. <http://doi.org/10.1016/j.cub.2015.06.071>

Firth, J. A., & Sheldon, B. C. (2015). Experimental manipulation of avian social structure reveals segregation is carried over across contexts. *Proceedings of the Royal Society B-Biological Sciences*, 282(1802), 20142350–20142350. <http://doi.org/10.1098/rspb.2011.1539>

Fletcher, R. J., Acevedo, M. A., Reichert, B. E., Pias, K. E., & Kitchens, W. M. (2011). Social network models predict movement and connectivity in ecological landscapes. *Proceedings of the National Academy of Sciences*, 108(48), 19282–19287.

Fletcher, R. J., Revell, A., Reichert, B. E., Kitchens, W. M., Dixon, J. D., & Austin, J. D. (2013). Network modularity reveals critical scales for connectivity in ecology and evolution. *Nature Communications*, 4, 2572. <http://doi.org/10.1038/ncomms3572>

Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3-5), 75–174. <http://doi.org/10.1016/j.physrep.2009.11.002>

Fowler, J. H. & Christakis, N. A. 2008. Dynamic spread of happiness in a large social network: longitudinal analysis over 20 years in the Framingham Heart Study. *BMJ*, 337, a2338–a2338.

Franks, D. W., Ruxton, G. D., & James, R. (2009). Sampling animal association networks with the gambit of the group. *Behavioral Ecology and Sociobiology*, 64(3), 493–503. <http://doi.org/10.1007/s00265-009-0865-8>

- Franz, M. & Nunn, C. 2009. Network-based diffusion analysis: a new method for detecting social learning.
- Frick, A. Ludwig, A. and Mehldau, H.: A Fast Adaptive Layout Algorithm for Undirected Graphs, *Proc. Graph Drawing 1994*, LNCS 894, pp. 388-403, 1995.
- Fruchterman, T.M.J. and Reingold, E.M. (1991). Graph Drawing by Force-directed Placement. *Software - Practice and Experience*, 21(11):1129-1164.
- Girvan, M., & Newman, M. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12), 7821–7826.
- Hill, A., Rand, D., Nowak, M. & Christakis, N. 2010. Emotions as infectious diseases in a large social network: the SISa model. *Proceedings Of The Royal Society B-Biological Sciences*,
- Hobaiter, C., Poisot, T., Zuberbühler, K., Hoppitt, W. & Gruber, T. 2014. Social Network Analysis Shows Direct Evidence for Social Transmission of Tool Use in Wild Chimpanzees. *PLoS Biology*, 12, e1001960.
- Hoppitt, W., Boogert, N. J. & Laland, K. N. 2010. Detecting social transmission in networks. *Journal of Theoretical Biology*, 263, 544–555.
- Ings, Thomas C., et al. "Review: Ecological networks–beyond food webs." *Journal of Animal Ecology* 78.1 (2009): 253-269.
- Jacoby, D. M. P., & Freeman, R. (2016). Emerging Network-Based Tools in Movement Ecology. *Trends in Ecology and Evolution*, 31(4), 301–314. <http://doi.org/10.1016/j.tree.2016.01.011>
- Kamada, T. and Kawai, S.: An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31/1, 7–15, 1989.
- Krackhardt, D. (1988). Predicting with networks: Nonparametric multiple regression analysis of dyadic data. *Social Networks*, 10(4), 359–381.
- Krause, J., James, R., Franks, D. W., & Croft, D. P. (Eds.). (2015). *Animal Social Networks*. Oxford: Oxford University Press.
- Lancichinetti, A., & Fortunato, S. (2011). Limits of modularity maximization in community detection. *Physical Review E*, 84(6), 066122. <http://doi.org/10.1103/PhysRevE.84.066122>
- Manly, B. F. (1995). A note on the analysis of species co-occurrences. *Ecology*, 1109–1115.
- May, R. M. (1973). *Stability and Complexity in Model Ecosystems*. Princeton: Princeton University Press.

- McRae, B. H., Dickson, B. G., Keitt, T. H., & Shah, V. B. (2008). Using circuit theory to model connectivity in ecology, evolution, and conservation. *Ecology*, 89(10), 2712-2724.
- Montoya, J. M., Pimm, S. L., & SOLÉ, R. V. (2006). Ecological networks and their fragility. *Nature*, 442(7100), 259–264. <http://doi.org/10.1038/nature04927>
- Newman, M. E. J. (2002). Assortative mixing in networks. *Physical Review Letters*, 89(20), 208701.
- Newman, M. E. J. (2006). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23), 8577–8582.
- Newman, M. E. J., & Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E*, 69(2), 026113. <http://doi.org/10.1103/PhysRevE.69.026113>
- Palla, G., Derényi, I., Farkas, I., & Vicsek, T. (2005). Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043), 814–818. <http://doi.org/10.1038/nature03607>
- Pascual, M., & Dunne, J. A. (2006). *Ecological networks: linking structure to dynamics in food webs*. Oxford University Press.
- Pinter-Wollman, N., Hobson, E. A., Smith, J. E., Edelman, A. J., Shizuka, D., de Silva, S., et al. (2014). The dynamics of animal social networks: analytical, conceptual, and theoretical advances. *Behavioral Ecology*, 25(2), 242–255. <http://doi.org/10.1093/beheco/art047>
- Reader, S. M. 2004. Distinguishing social and asocial learning using diffusion dynamics. *Learning & Behavior*, 32, 90–104.
- Ribeiro, M. C., Metzger, J. P., Martensen, A. C., Ponzoni, F. J., & Hirota, M. M. (2009). The Brazilian Atlantic Forest: How much is left, and how is the remaining forest distributed? Implications for conservation. *Biological conservation*, 142(6), 1141-1153.
- Robins G, Pattison P, Kalish Y, Lusher D. 2007. An introduction to exponential random graph (p*) models for social networks. *Soc Networks*. 29:173–191.
- Robertson, C. (1929). *Flowers and Insects: Lists of Visitors to Four Hundred and Fifty-Three Flowers*. Science Press Printing Company, Lancaster, PA.
- Shizuka, D., & Farine, D. R. (2016). Measuring the robustness of network community structure using assortativity. *Animal Behaviour*, 112, 237–246. <http://doi.org/10.1016/j.anbehav.2015.12.007>
- Shizuka, D., Chaine, A. S., Anderson, J., Johnson, O., Laursen, I. M., & Lyon, B. E. (2014). Across-year social stability shapes network structure in wintering migrant sparrows. *Ecology Letters*, 17(8), 998–1007. <http://doi.org/10.1111/ele.12304>

Sih, A., Hanser, S. F., & Mchugh, K. A. (2009). Social network theory: new insights and issues for behavioral ecologists. *Behavioral Ecology and Sociobiology*, 63(7), 975–988.
<http://doi.org/10.1007/s00265-009-0725-6>

Taylor, C. M., & Norris, D. R. (2009). Population dynamics in migratory networks. *Theoretical Ecology*, 3(2), 65–73. <http://doi.org/10.1007/s12080-009-0054-4>

Urban, D., & Keitt, T. (2001). Landscape connectivity: A graph-theoretic perspective. *Ecology*, 82(5), 1205–1218.

Urban, D. L., Minor, E. S., Treml, E. A., & Schick, R. S. (2009). Graph models of habitat mosaics. *Ecology Letters*, 12(3), 260–273.

Wey, T., Blumstein, D. T., Shen, W., & Jordán, F. (2008). Social network analysis of animal behaviour: a promising tool for the study of sociality. *Animal Behaviour*, 75(2), 333–344.
<http://doi.org/10.1016/j.anbehav.2007.06.020>

Whitehead, H. (2008). *Analyzing Animal Societies: Quantitative Methods for Vertebrate Social Analysis*. Chicago: University of Chicago Press.