SNA R1: Introduction to R for Social Network Analytics

Overview

J. Alberto Espinosa

2025-07-01

This Quarto file contains a basic introduction to R and Quarto for Social Network Analytics for Managers (SNA4M). It contains an overview of packages for SNA and an overview of R scripting for SNA. If you have no prior experience with R, I recommend reviewing my introduction to R from available in my PAML4M book R companion PAML4M_1R_R_Intro.Qmd available in my GitHub site https://github.com/jibe4fun/paml4m/tree/R-Code. If you are already familiar with R, but not with R for SNA, this script is a good place to start. If you are interested in Predictive Analytics and Machine Learning for Managers, please refer to my PAML4M book Predictive Analytics and Machine Learning for Managers or chat with my chatbot on this topic PAML4M ChatBot. You can also query all the lectures, R scripts and other material for this course on the [SNA4M ChatBot] (https://poe.com/SNA4M).

Table of contents

Note: The parameter `#| eval: false` inside some R code sections marked with `{r}` are there to prevent them from running when you are rendering the script into a document to avoid cluttering the output with data displays, help information, etc. To explore these commands, simply go to the respective script line an run it with `Ctrl-Enter`.

R Overview

Introduction

This R Overview covers very basic aspects of R, necessary for the class. Students who wish to gain deeper understanding of R can enroll in Canvas KSB-999 R Workshop Overview for Business Analytics, and review the video lectures, R scripts and other materials posted there. In subsequent scripts I go into more detail about the various SNA methods covered in this class.

If you have taken ITEC 621 Predictive Analytics, are familiar with the R code in the PAML4M GitHub site noted above, or have already gone through KSB-999 R Workshop for Business Analytics, you can skip this, or skim it to refresh your memory. However, there are a few sections below that are specific to SNA and it would help to review those sections.

To enroll in the KSB-999 course, go to:

https://american.instructure.com/enroll/9DJDYC

Setup a Project

I strongly recommend creating a project environment for every R project you need to or have worked on. You need to create a project environment this class where you can keep all your R and R Markdown scripts and data files. When you open that project, your working directory will be automatically set to the file folder where your project resides and you will have easy access to these files. Also, any objects that were open the last time you used R will be restored. When you exit R, it will ask you if you want to save your project environment. Always answer "yes", if you want to see everything next time you open your project.

To create a project for this class, first, create a dedicated working folder in your computer for this class (e.g., I use `C:/AU/Courses/ITEC623/R`). Then select: `File -> New Project` (or click on the scrollable list in the upper-right corner of R Studio and select **New Project**; then select **Existing Directory**, point to your working folder, and click on "Create Project". That easy.

Also, download all the data sets provided on Canvas into your project folder. Some of the scripts provided assume that certain **.CSV** and other data sets are present in your project folder.

Required Packages

A lot of the material in this script follows Luke's book: (UGNA) A User's Guide to Network Analysis with R, by Douglas Luke. All material labeled **"UGNA"** comes from that textbook. See the book's web site:

http://www.springer.com/us/book/9783319238821

Please install the following textbook packages for this course:

```
install.packages("devtools") # UGNA Textbook resource
library(devtools) # Need to load this library for github installation next
install_github("DougLuke/UserNetR") # Textbook data sets
```

ASIDE, BUT IMPORTANT – FOR WINDOWS USERS ONLY

If you installed **devtools** and **DougLuke/UserNetR** above successfully, ignore this ASIDE. If not, please follow these instructions:

Note: the `install_github()` function requires that you have a current version of the `{Rtools}` package running in your system. If you are a Windows user, you may get an error when you install the `UserNetR` library as instructed above. If so, you will need to install the current version of `{Rtools}`, which requires a few steps, outlined in:

https://cran.r-project.org/bin/windows/Rtools/

Packages installed with `install.packages()` from the CRAN repository install fine and don't need further action. Once installed, they are ready to go. However, there are other packages written in C, C++ or other languages, stored in other repositories (e.g. GitHub), which need to be compiled before you can use them. Two ways to do that are:

1. With the `{devtools}` package, which needs to be installed and then loaded with the library() function. `{devtools}` makes it a lot easier to install binary packages, but it relies on `{Rtools}` to work, and in some versions of R, so `{Rtools}` is installed as a dependent package to `{devtools}`. But `{Rtools}` does not install directly in some versions of R and Rtools. If that is the case, follow the instructions below.
2. The `{Rtools}` package will take care of the compilation of GitHub packages for you. If you install `{Rtools}`, your packages will be compiled upon installation

One option to install Rtools is to first recover the latest package from the CRAN archive

```
install_url("https://cran.r-project.org/src/contrib/Archive/perturb/perturb_2.10.tar.gz")
```

Then install the package

```
install.packages("Rtools")
```

Again, if `{Rtools}` does not install (it does not install directly from CRAN in Windows machines with some versions of R). If you cannot install `{Rtools}` with the `install.packages()` function, this is the workaround, as outlined in:

https://cran.r-project.org/bin/windows/Rtools/

As that web site indicates, `{Rtools}` is only needed for packages built with C, C++ and Fortran. Packages installed directly from CRAN don't need Rtools.

The first step is to download `{Rtools}`. Note that there are various versions of Rtools:

- RTools 4.5 for R 4.5
- RTools 4.4 for R 4.4

- RTools 4.3 for R 4.3
- RTools 4.2 for R 4.2
- RTools 4.0 for R 4.0 to 4.1

I recommend updating R to 4.5 or the latest version available. The installation of `{Rtools}` is much simpler.

If you are running **R version 4.0 (only)**, after you install `{Rtools}` 4.0, follow these instructions. Put `{Rtools}` in you R environment PATH by running this script line (ONLY do this after you successfully installed Rtools 4.0)

```
writeLines('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"', con = "~/.Renviron")
```

If you are running R version 4.2 to 4.5 install `{RTools}` the PATH is automatically read by R, so no need to set the PATH.

After installing either version of `{Rtools}`, exit R Studio and restart it, so that it reads the updated path. To verify if the installation worked out, try this:

```
Sys.which("make")
```

If the function above returns one of the following, corresponding to your R version, then you are all set:

`"C:\\rtools40\\usr\\bin\\make.exe"`, you are all set with Rtools 4.0 `"C:\\rtools42\\usr\\bin\\make.exe"`, you are all set with Rtools 4.2 `"C:\\rtools43\\usr\\bin\\make.exe"`, you are all set with Rtools 4.3 `"C:\\rtools44\\usr\\bin\\make.exe"`, you are all set with Rtools 4.4 `"C:\\rtools45\\usr\\bin\\make.exe"`, you are all set with Rtools 4.5

If the function above returns `""`, either `{Rtools}` did not install or the PATH was not set correctly and it is important to correct this.

End of ASIDE

Main Package Installations

Most of the work in this class will be done with two packages: `{statnet}` and `{igraph}`. Actually, `{statnet}` is not a single package but a compilation of SNA packages (e.g., `{sna}`, `{network}`, etc.) written by a number of expert SNA professors. Installing `{statnet}` conveniently installs all the associated SNA packages and, conveniently, loading the `{statnet}` library loads all the libraries contained in it.

The `{igraph}` package does most of the same things that `{statnet}` does and more. It was created by Csárdi et. al. and there is a similar version of this package for Python. More information on this below. In my opinion, `{statnet}` is more complete because it contains many specialized packages like `{sna}`, `{ergm}`, `{network}` and so one. However, `{igraph}` is easier to use and has a cleaner syntax. In addition there are many web sites and books about `{igraph}` and its documentation is more complete and clearer.

```
install.packages("statnet", dependencies = T)
install.packages("igraph")
```

We will be using both packages in this class, but `{igraph}` should be sufficient for most calculations and applications illustrated in this course. Also, the `{sand}` package contains useful data sets and libraries used in the book *Statistical Analysis of Network Data* by Kolaczyk and Csárdi, who developed `{igraph}`.

```
install.packages("sand")
```

In addition, we will be using several other useful packages occasionally. Please install the following packages:

```
install.packages("influenceR") # To quantify structural importance
install.packages("intergraph") # To work with network data objects
```

```
install.packages("igraphdata") # {igraph} data sets
install.packages("Matrix") # To work with matrices
install.packages("RColorBrewer") # To manipulate color palettes
install.packages("networkD3") # A package for interactive network graphs
install.packages("rgl", type = "binary") # For 3D network graphs
install.packages("ergm") # For ERGM regressions (will only cover lightly)
install.packages("amap") # For hierarchical clustering and k-means
```

Other packages we will be using (if you completed the **KSB 999 R Workshop**, you should already have these packages installed):

```
install.packages("ggplot2") # For attractive stats graphics
install.packages("Hmisc") # Misc statistical functions
install.packages("GGally") # Some additional stats visuals
install.packages("corrplot") # Nice correlation plots
```

Detaching Packages

As I'm sure you know already, you need to load any libraries you need during your R session, such as:

```
library(statnet)
library(igraph)
```

However, loading two packages that contain functions with the same names is problematic, because one package's functions will **mask** functions from the other package, and you will get errors.

There are 2 ways to overcome this problem:

1. *(Recommended)* Because `{statnet}` and `{igraph}` do very similar things, you will often work with one or the other, whichever you like the most. But sometimes you reach a point where you need to switch from one package to the other to access functions from one of these packages. For example, if you need to use {statnet} first and then need to switch to {igraph} you need to follow this sequence:

```
library(statnet) # To load {statnet} and do what you have to do in {statnet}
detach(package:statnet) # To unload {statnet}
library(igraph) # To then load {igraph}
```

If you need to go from {igraph} to {statnet} do the opposite:

```
library(igraph) # To load {igraph} \# Do what you have to do in {statnet}
detach(package:igraph) # To unload {igraph}
library(statnet) # Then load {statnet}
```

Or,

2. If you are in one library, e.g., `{statnet}`, and you need just need to invoke a function from `{igraph}` to do some light work and go back to `{statnet}`, it is probably quicker to force a function reference with the double colon `::` operator:

```
library(igraph) # To load {igraph}
library(statnet) # To load {statnet}
igraph::add.edges(xxx) # For example, to access the add.edges() igraph function
```

This second way can get confusing some times if you don't know the masking order of the functions, so I recommend the first method above.

Help with Packages

Help for R and SNA in general is abundant. Here are a few help resources

The {devtools} contains several useful functions for R:

```
help(package = "devtools")
browseURL("https://github.com/hadley/devtools")
```

{UserNetR} is a package by the textbook author, Luke, with data sets. You can find out more here:

```
help(package="UserNetR") # For statistical modeling of network data
browseURL("https://github.com/DougLuke/UserNetR")
```

Resources for {statnet}:

```
help(package = "statnet")
```

https://cran.r-project.org/web/packages/statnet/statnet.pdf

https://www.rdocumentation.org/packages/statnet/versions/2016.9/topics/statnet-package

https://statnet.csde.washington.edu/trac/raw-attachment/wiki/Sunbelt2014/introToSNAinR-handout.pdf

Resources for {igraph}

```
help(package = "igraph") # A comprehensive SNA library
```

{igraph} web site (for R)

http://igraph.org/r/

There is a web site for Python too, if you are interested:

https://igraph.org/python/

{igraph} reference manual:

https://cran.r-project.org/web/packages/igraph/igraph.pdf

{igraph} documentation:

https://igraph.org/r/doc/

Comprehensive {igraph} reference manual by its author and colleagues

https://igraph.org/c/doc/igraph-docs.pdf

{intergraph} It is often necessary to convert {statnet} data objects into {igraph} and the other way around. This package has functions to do that.

As mentioned above, R Studio comes with several data sets already available in your working environment in the {datasets} package.

```
help(package = "datasets")
```

Other libraries, like {ggplot2}, {StatNet} also come with datasets you can use. Take a look:

```
library(ggplot2)
library(UserNetR)
library(igraphdata)

data() # To display all data sets loaded in the working environment
data(package = "ggplot2") # View all datasets in {ggplot2}
data(package = "UserNetR") # View all datasets in {UserNetR}
```

Reading and Writing External Data

Reading Data Into R

For sociomatrices and incidence matrices, it is often a lot easier to create the matrices in Excel, save them as **.CSV** files, and then read them into R.

CAUTION: because we often store row names in network data in the first column and column names in the first row, sometimes we need to read the row names or column headers, and sometimes we don't.

To read the full table, simply enter:

```
mydata <- read.table("GlobalTeamsTZ.csv", sep = ",")
```

Notice that `read.table()` creates a data frame:

```
class(mydata)
```

```
[1] "data.frame"
```

Also, notice how the column headers were read into the data frame:

```
View(mydata)
```

This is not useful because the column names are randomly assigned by R as V1 (Variable 1), V2, etc.

To use the first row as the column headed and use them as variable names, use `header = T`:

```
mydata <- read.table("GlobalTeamsTZ.csv",
                     header = T,
                     sep = ",")
```

```
View(mydata) # Check it out
```

But notice that the row names appear as data in column 1, when they should be just row names (equal to the column names). To read the first column as row names use `row.names = 1`:

```
mydata <- read.table("GlobalTeamsTZ.csv",
                     header = T,
                     row.names = 1,
                     sep = ",")
```

```
 View(mydata) # Check it out
```

Writing R Data into External Files

You can save R data back to .csv files as follows:

```
write.table(mydata, file = "mydata.csv", sep = ",")
```

The `"sep ="` option specifies the delimiter between values, with `","` being the most common and one the Excel will read.

R Data Structures

R data objects can be stored in a variety of data structures. Each data structure below is unique, and it is critical that you understand the difference between these data structures to use R proficiently. This is even more important when working with network data. There are many data structures in R, but vectors, data frames, matrices, and lists are some of the most important ones. Many of you already know what vectors, matrices and data frames are. For the sake of continuity, I review these concepts briefly below.

Vectors

A **vector** is simply a list of data elements of the same type (e.g., all numbers or all characters). Think of it as a column in a database table.

The `c()` function is the **vector creator** function, and one you will be using frequently. For example, to create a vector named **e** with 7 numbers, enter:

```
e <- c(1, 2, 3, 2, 6, 3, 5)
e # Notice the 7 numeric values are numeric
```

```
[1] 1 2 3 2 6 3 5
```

R is said to be a **vectorized** language, meaning that many values are stored in vectors and that R has many convenient features to manipulate data contained in these vectors. Vectorized operations in R make it really easy to make massive computations. For example, suppose that **e** is a vector containing regression residuals (i.e., errors) and we want to compute the **sum of errors squared**. To do that, we would simply square **e** and **sum** the resulting elements:

```
e2 <- e^2 # To create a vector with the squared residuals
e2 # Check it out
```

```
[1]  1  4  9  4 36  9 25
```

```
cat("\n")
```

```
cat("The sum of squared errors is", sum(e2))
```

```
The sum of squared errors is 88
```

```
cat("\n")
```

```r
cat("The mean squared error is", round(mean(e2), 2))
```

```
The mean squared error is 12.57
```

Again, all values in a vector must be of the same type. For example:

```r
x <- c(1, "al") # creates a vector with "1" (not 1) and "al"
x # Notice that "1" is converted to character to be compatible with "al"
```

```
[1] "1"  "al"
```

Vector indices – used to refer to a specific value in the vector

```r
x <- c(1, 2, 3, 2, 6, 3, 5)
x
```

```
[1] 1 2 3 2 6 3 5
```

```r
x[4] # Indexing: e.g., 4th element of the vector
```

```
[1] 2
```

The negative sign is used to remove vector elements. This command removes the 4th element from the vector

```r
x <- x[-4]
x # Check it out
```

```
[1] 1 2 3 6 3 5
```

Naming vector elements

```r
prof <- c(First = "Alberto", Last = "Espinosa", Title = "Professor")
prof # Check it out
```

```
    First      Last     Title
"Alberto" "Espinosa" "Professor"
```

Or, you can also name them like this:

```r
prof <- c("Alberto", "Espinosa", "Professor")
names(prof) <- c("First", "Last", "Title")
prof # Check it out
```

```
    First      Last     Title
"Alberto" "Espinosa" "Professor"
```

Data Frames

Think of **data frames** as structured database tables. Essentially, a data frame is a collection of vector columns. That is, different columns can be of different data types, but all values contained in each column of a data frame contains data of the same type. Conveniently, vectors can be used to assemble data frames very easily by binding columns together. Conversely, you can decompose data frames into separate vectors using the $ attribute designator. For example, let's create a few vectors:

```
courses <- c("ITEC 610", "ITEC 616", "ITEC 620", "ITEC 621",
             "ITEC 660", "ITEC 670", "KSB 620")

courses # Check it out
```

```
[1] "ITEC 610" "ITEC 616" "ITEC 620" "ITEC 621" "ITEC 660" "ITEC 670" "KSB 620"
```

```
cat("\n")
```

```
instructors <- c("Wasil", "Carmel", "Simon", "Espinosa",
                 "Lee", "Clark", "Janzen")

instructors # Check it out
```

```
[1] "Wasil"    "Carmel"   "Simon"    "Espinosa" "Lee"      "Clark"    "Janzen"
```

Let's create a vector with course pre-requisites

```
prereqs <- c("None", "None", "ITEC 610", "ITEC 620",
             "ITEC 610", "ITEC 616", "ITEC 620")

prereqs # Check it out
```

```
[1] "None"     "None"     "ITEC 610" "ITEC 620" "ITEC 610" "ITEC 616" "ITEC 620"
```

Now let's create the data frame containing these 2 vectors:

```
MyDataFrame <- data.frame(courses, instructors, prereqs)
MyDataFrame # Check it out
```

```
   courses instructors  prereqs
1 ITEC 610       Wasil     None
2 ITEC 616      Carmel     None
3 ITEC 620       Simon ITEC 610
4 ITEC 621    Espinosa ITEC 620
5 ITEC 660         Lee ITEC 610
6 ITEC 670       Clark ITEC 616
7  KSB 620      Janzen ITEC 620
```

```
cat("\n")
```

```
class(MyDataFrame) # Check it out
```

```
[1] "data.frame"
```

Now let's extract columns as vectors with the `$` operator:

```
MyCourses <- MyDataFrame$courses
MyCourses # Check it out
```

```
[1] "ITEC 610" "ITEC 616" "ITEC 620" "ITEC 621" "ITEC 660" "ITEC 670" "KSB 620"
```

Data Frame Indices

Vectors only need one index (e.g., `x[3]`) to refer to a particular element. But data frames need row and column indices to address an element:

```
MyDataFrame[1, 2] # Element in 1st row and 2nd column
```

```
[1] "Wasil"
```

```
MyDataFrame[3:6, 2:3] # Rows 3 through 6 from columns 2 throgh 3
```

```
  instructors  prereqs
3       Simon ITEC 610
4    Espinosa ITEC 620
5         Lee ITEC 610
6       Clark ITEC 616
```

```
MyDataFrame[2, ] # Second row (all columns)
```

```
   courses instructors prereqs
2 ITEC 616      Carmel    None
```

```
MyDataFrame[, 1] # First column (all rows)
```

```
[1] "ITEC 610" "ITEC 616" "ITEC 620" "ITEC 621" "ITEC 660" "ITEC 670" "KSB 620"
```

Let's look at just part of the data frame:

```
head(MyDataFrame) # Display the first 6 rows (6 is the default)
```

```
   courses instructors  prereqs
1 ITEC 610       Wasil     None
2 ITEC 616      Carmel     None
3 ITEC 620       Simon ITEC 610
4 ITEC 621    Espinosa ITEC 620
5 ITEC 660         Lee ITEC 610
6 ITEC 670       Clark ITEC 616
```

```
head(MyDataFrame, 7) # Display the first 7 rows
```

```
   courses instructors  prereqs
1 ITEC 610       Wasil     None
2 ITEC 616      Carmel     None
3 ITEC 620       Simon ITEC 610
4 ITEC 621    Espinosa ITEC 620
5 ITEC 660         Lee ITEC 610
6 ITEC 670       Clark ITEC 616
7  KSB 620      Janzen ITEC 620
```

```
tail(MyDataFrame) # Display the last 6 rows
```

```
   courses instructors  prereqs
2 ITEC 616      Carmel     None
3 ITEC 620       Simon ITEC 610
4 ITEC 621    Espinosa ITEC 620
5 ITEC 660         Lee ITEC 610
6 ITEC 670       Clark ITEC 616
7  KSB 620      Janzen ITEC 620
```

```
MyDataFrame[2, c("courses", "prereqs")] # List selected columns for row 2
```

```
   courses prereqs
2 ITEC 616    None
```

```
MyDataFrame[, c("courses", "prereqs")] # List selected columns for all rows
```

```
   courses  prereqs
1 ITEC 610     None
2 ITEC 616     None
3 ITEC 620 ITEC 610
4 ITEC 621 ITEC 620
5 ITEC 660 ITEC 610
6 ITEC 670 ITEC 616
7  KSB 620 ITEC 620
```

Matrices

The matrix is a **very important** data structure for SNA because a network is represented quantitatively as a sociomatrix, which a matrix of linked network actors. Once we begin to add attributes to nodes and edges, and further relationships, we can no longer represent a network as a single matrix, but as a collection of matrices conveniently stored in network objects. However, once we create a network object, we can extract matrices from it to render graphs and perform computations.

A matrix is similar to a data frame in many respects because it is a table with rows, columns, and cells with values. The difference is that the values in all the cells in the matrix must be of the same type for the entire matrix. Most commonly, matrices contain quantitative value, which can be easily manipulated with matrix algebra. For SNA, the matrices will either contain 0's and 1's for binary networks, or quantitative values for weighted networks.

Data frames can be converted into matrices and the other way around. This is important because certain SNA functions work only with data frames, others only with matrices, and some others with both. These conversions are straightforward:

Data Frame to Matrix:

```
MyMatrix <- as.matrix(MyDataFrame)
MyMatrix # Check it out
```

```
     courses    instructors prereqs
[1,] "ITEC 610" "Wasil"     "None"
[2,] "ITEC 616" "Carmel"    "None"
[3,] "ITEC 620" "Simon"     "ITEC 610"
[4,] "ITEC 621" "Espinosa"  "ITEC 620"
[5,] "ITEC 660" "Lee"       "ITEC 610"
[6,] "ITEC 670" "Clark"     "ITEC 616"
[7,] "KSB 620"  "Janzen"    "ITEC 620"
```

```
class(MyMatrix) # Check it out
```

```
[1] "matrix" "array"
```

Matrix to Data Frame

```
MyDataFrame <- as.data.frame(MyMatrix)
MyDataFrame
```

```
   courses instructors  prereqs
1 ITEC 610       Wasil     None
2 ITEC 616      Carmel     None
3 ITEC 620       Simon ITEC 610
4 ITEC 621    Espinosa ITEC 620
5 ITEC 660         Lee ITEC 610
6 ITEC 670       Clark ITEC 616
7  KSB 620      Janzen ITEC 620
```

```
class(MyDataFrame) # Check it out
```

```
[1] "data.frame"
```

Note, we can use the `cbind()` function to create matrices from vectors:

```
MyMatrix <- cbind(courses, instructors, prereqs)
MyMatrix # Check it out
```

```
     courses    instructors prereqs
[1,] "ITEC 610" "Wasil"     "None"
[2,] "ITEC 616" "Carmel"    "None"
[3,] "ITEC 620" "Simon"     "ITEC 610"
[4,] "ITEC 621" "Espinosa"  "ITEC 620"
[5,] "ITEC 660" "Lee"       "ITEC 610"
[6,] "ITEC 670" "Clark"     "ITEC 616"
[7,] "KSB 620"  "Janzen"    "ITEC 620"
```

```
cat("\n") # Blank line
```

```
  class(MyMatrix)
```

```
[1] "matrix" "array"
```

Note:

`cbind()` binds columns of the same type into a matrix `rbind()` binds rows of the same type and number of columns into a matrix and `data.frame()` binds columns of different types into data frames.

```
  MyDataFrame <- data.frame(courses, instructors, prereqs)
  MyDataFrame # Check it out
```

```
   courses instructors  prereqs
1 ITEC 610       Wasil     None
2 ITEC 616      Carmel     None
3 ITEC 620       Simon ITEC 610
4 ITEC 621    Espinosa ITEC 620
5 ITEC 660         Lee ITEC 610
6 ITEC 670       Clark ITEC 616
7  KSB 620      Janzen ITEC 620
```

```
  class(MyDataFrame)
```

```
[1] "data.frame"
```

If multiple vectors are of the same type and size, they can be `rbinded` or `cbinded` into matrices, which is very useful for network data manipulation. For example, let's create 5 vectors with `c()` and `rbind()` them into 5 matrix rows

```
  net.matrix.r <- rbind(c(0, 1, 0, 1, 0),
                        c(0, 0, 1, 1, 0),
                        c(0, 1, 0, 0, 0),
                        c(0, 0, 0, 0, 0),
                        c(0, 0, 1, 0, 0))

  class(net.matrix.r) # NOT a network object, just a matrix
```

```
[1] "matrix" "array"
```

```
  # Let's add row an column vertex names:
  rownames(net.matrix.r) <- # Let's add row an column vertex names:
    colnames(net.matrix.r) <-
    c("joe", "moe", "doe", "boe", "foe")

  net.matrix.r # Check it out
```

```
    joe moe doe boe foe
joe   0   1   0   1   0
moe   0   0   1   1   0
doe   0   1   0   0   0
boe   0   0   0   0   0
foe   0   0   1   0   0
```

Alternatively, let's `cbind()` 5 vectors into 5 matrix columns

```
net.matrix.c <- cbind(c(0, 1, 1, 0, 0),
                      c(0, 0, 1, 1, 0),
                      c(0, 1, 0, 0, 0),
                      c(0, 0, 0, 0, 0),
                      c(0, 0, 1, 0, 0))

class(net.matrix.c) # NOT a network object, just a matrix
```

```
[1] "matrix" "array"
```

```
# Let's add row an column vertex names:
rownames(net.matrix.c) <- # Let's add row an column vertex names:
  colnames(net.matrix.c) <-
  c("joe", "moe", "doe", "boe", "foe")

net.matrix.c # Check it out
```

```
    joe moe doe boe foe
joe   0   0   0   0   0
moe   1   0   1   0   0
doe   1   1   0   0   1
boe   0   1   0   0   0
foe   0   0   0   0   0
```

Matrix Transpose

`rbind()` will treat each vector as a row and `cbind()` will treat each vector as a column. You will notice that the two matrices we constructed above are the transpose of each other, which is very useful for network data operations. You could create the transpose of a matrix more easily using the `t()` function.

```
net.matrix.c <- t(net.matrix.r)
net.matrix.c # Check it out
```

```
    joe moe doe boe foe
joe   0   0   0   0   0
moe   1   0   1   0   0
doe   0   1   0   0   1
boe   1   1   0   0   0
foe   0   0   0   0   0
```

Lists

Lists are arguably the most important data structure for network objects. This is so because many complex network objects can be and are generally stored internally as lists. While matrices and data frames are important for manipulating sociomatrices and other simpler network data objects, lists are the most effective data structure to store and manipulate complex data, particularly when the network is **decorated**, that is, when vertices and edges have lots of attributes. This is so because a list element is not limited to a single value, but could also be a matrix, another list, a vector, a data frame, or any other type of data object. Therefore, many network objects in popular SNA packages are stored as lists. For example a network object stored in a list would typically contain: **(1) Vertices or Nodes** (e.g., who are the network actors), **(2) Edges or Dyad Ties** (e.g., which actor is connected

to which); **(3) Vertex or Node Attributes** (e.g., age, height, profession, department); and **(4) Edge or Dyad Tie Attributes** (e.g., communication frequency, friendship).

This is a simple list with three named elements with their values, very similar to a vector, except that we can store both text and data in the same list:

```
y <- list(name = "Alberto",
          title = "Professor",
          age = 16)

y # Check it out -- the element names and data types are preserved
```

```
$name
[1] "Alberto"

$title
[1] "Professor"

$age
[1] 16
```

A list can be quite complex because its elements can be anything, that is, single values, vectors, other lists. Lists can also be **named** or **un-named**. For example, the list below is named **"friends"** and it is more complex. Notice that we have a few lists inside the list, and all list elements are named, which makes it easier to reference.

Named Lists: The elements are named and the name chains are concatenated with the `$` symbol:

```
friends <- list(MyName = "Alberto",
                MyAge = 15,
                Charlie = list(age = 20, major = "Analytics"),
                John = list(age = 30, job = "Programmer"),
                Dan = list(age = 40, profession = "Lawyer"),
                Others = c("Larry", "Moe", "Curly"))

friends # Check it out
```

```
$MyName
[1] "Alberto"

$MyAge
[1] 15

$Charlie
$Charlie$age
[1] 20

$Charlie$major
[1] "Analytics"


$John
$John$age
[1] 30

$John$job
[1] "Programmer"
```

```
$Dan
$Dan$age
[1] 40

$Dan$profession
[1] "Lawyer"


$Others
[1] "Larry" "Moe"   "Curly"
```

You can extract distinct elements from a list with the `$` or the double brackets `[[ ]]`, as illustrated next:

```
  class(friends) # A list
```

```
[1] "list"
```

```
  summary(friends) # Very basic information about its elements
```

```
        Length Class  Mode
MyName  1      -none- character
MyAge   1      -none- numeric
Charlie 2      -none- list
John    2      -none- list
Dan     2      -none- list
Others  3      -none- character
```

```
  str(friends) # More detailed information
```

```
List of 6
 $ MyName : chr "Alberto"
 $ MyAge  : num 15
 $ Charlie:List of 2
  ..$ age  : num 20
  ..$ major: chr "Analytics"
 $ John   :List of 2
  ..$ age: num 30
  ..$ job: chr "Programmer"
 $ Dan    :List of 2
  ..$ age       : num 40
  ..$ profession: chr "Lawyer"
 $ Others : chr [1:3] "Larry" "Moe" "Curly"
```

Let's use the element names from the `str()` display to extract some data from the list:

```
  friends$MyName # A simple element
```

```
[1] "Alberto"
```

```
friends$John # The element "John" contains 2 sub-elements, $age and $job
```

```
$age
[1] 30

$job
[1] "Programmer"
```

```
friends$John$job # John's job
```

```
[1] "Programmer"
```

```
friends$Others[2] # The second element of Others is "Moe"
```

```
[1] "Moe"
```

Single and double brackets in lists retrieve the exact same values, but these values are returned with different classes. A single bracket index, e.g., `friends[1]` will retrieve list element 1 and return its value as a **list**, even if it contains a single value.

A double bracket index can be used to retrieve the same list element values, as with single brackets, but the results will be returned in its own data type, rather than a list:

First list element, which is a single character value, returned as a list:

```
friends[1]
```

```
$MyName
[1] "Alberto"
```

```
class(friends[1])
```

```
[1] "list"
```

First list element, which is a single character value, returned as a character:

```
friends[[1]]
```

```
[1] "Alberto"
```

```
class(friends[[1]])
```

```
[1] "character"
```

Similarly:

Third list element, which is a list:

```
  friends[3]
```

```
$Charlie
$Charlie$age
[1] 20

$Charlie$major
[1] "Analytics"
```

```
  class(friends[3])
```

```
[1] "list"
```

Third list element, which is a list:

```
  friends[[3]]
```

```
$age
[1] 20

$major
[1] "Analytics"
```

```
  class(friends[[3]])
```

```
[1] "list"
```

Sixth list element, returned as a list:

```
  friends[6]
```

```
$Others
[1] "Larry" "Moe"   "Curly"
```

```
  class(friends[6]) # Returned as a list
```

```
[1] "list"
```

Sixth list element, which is a vector, returned as a character vector

```
  friends[[6]]
```

```
[1] "Larry" "Moe"   "Curly"
```

```
  class(friends[[6]])
```

```
[1] "character"
```

Unnamed Lists:

These are less useful, but are used a lot with networks. The only way to reference values in an unnamed list is with numeric indices, which makes the resulting R code more cryptic:

```
  x <- c(1, 2, 3)
  y <- c("a", "b", "c")
  z <- rbind(c(0, 1, 0, 1, 0),
             c(0, 0, 1, 1, 0),
             c(0, 1, 0, 0, 0),
             c(0, 0, 0, 0, 0),
             c(0, 0, 1, 0, 0))

  u.list <- list(x, y, z)
  str(u.list) # Check out the list contents and structure
```

```
List of 3
 $ : num [1:3] 1 2 3
 $ : chr [1:3] "a" "b" "c"
 $ : num [1:5, 1:5] 0 0 0 0 0 1 0 1 0 0 ...
```

```
  u.list[2] # Retrieve second list element
```

```
[[1]]
[1] "a" "b" "c"
```

```
  class(u.list[2]) # As a list
```

```
[1] "list"
```

```
  u.list[[2]] # Retrieve second list element
```

```
[1] "a" "b" "c"
```

```
  class(u.list[[2]]) # In it's original form, a character vector
```

```
[1] "character"
```

```
  u.list[3] # Retrieve third list element (a matrix)
```

```
[[1]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    0    1    0
[2,]    0    0    1    1    0
[3,]    0    1    0    0    0
[4,]    0    0    0    0    0
[5,]    0    0    1    0    0
```

```
class(u.list[3]) # As a list
```

```
[1] "list"
```

```
u.list[[3]] # Retrieve third list element (a matrix)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    0    1    0
[2,]    0    0    1    1    0
[3,]    0    1    0    0    0
[4,]    0    0    0    0    0
[5,]    0    0    1    0    0
```

```
class(u.list[[3]]) # In its original form, a matrix
```

```
[1] "matrix" "array"
```

{statnet} Network Objects

To create and work with `{statnet}` network objects we first need to load the `{statnet}` library:

```
library(statnet)
```

Network data can be stored in a data frame, matrix, network or graph object. Some network functions in R will work fine with data frames and matrices, but most functions will require the data to be read into a network object (in `{statnet}`) or a graph object (in `{igraph}`).

In addition, data frames and matrix can only contain row and column data. Network and graph objects store the data as lists, so you can store all kinds of additional data to document the network (e.g., vertex labels, edge labels, vertex sizes and labels, centralities, network properties, etc.), making these objects excellent containers for data.

Let's create a `{statnet}` network object. First, note that we will use the `network()` function, which is contained in the `{statnet}` library. As we will discuss later, a network object is the same, regardless of the format of the matrix or data frame used to read the data. But we can read data into a network object in a variety of formats, which will come in handy when we retrieve data from large networks or some other complex data.

Note that we use the attribute `matrix.type = "adjacency"` to tell the `network()` object what data format the network raw data is in. An **adjacency matrix**, or **sociomatrix** is one in which each row represents a network node and each column also represents a network node. A cell in an adjacency matrix shows the relationship between the row node and the column node.

Let's re-create the matrices we created earlier for convenience. For example, the second element of the third row of **net.matrix.r** is 1. This means that members 3 (row) and 2 (column) share a tie, but members 3 (row) and 1 (column) do not because the corresponding value is 0. Also, the two matrices below contain network data, but as a matrix, not as a network object yet, and in sociomatrix or adjacency format.

```
net.matrix.r <- rbind(c(0, 1, 0, 1, 0),
                      c(0, 0, 1, 1, 0),
                      c(0, 1, 0, 0, 0),
                      c(0, 0, 0, 0, 0),
                      c(0, 0, 1, 0, 0))
```

```
  rownames(net.matrix.r) <- # Let's add row an column vertex names:
    colnames(net.matrix.r) <-
    c("joe", "moe", "doe", "boe", "foe")

  net.matrix.r # To double check
```

```
    joe moe doe boe foe
joe   0   1   0   1   0
moe   0   0   1   1   0
doe   0   1   0   0   0
boe   0   0   0   0   0
foe   0   0   1   0   0
```

```
  cat("\n")
```

```
  net.matrix.c <- cbind(c(0, 1, 1, 0, 0),
                        c(0, 0, 1, 1, 0),
                        c(0, 1, 0, 0, 0),
                        c(0, 0, 0, 0, 0),
                        c(0, 0, 1, 0, 0))

  rownames(net.matrix.c) <- # Let's add row an column vertex names:
    colnames(net.matrix.c) <-
    c("joe", "moe", "doe", "boe", "foe")

  net.matrix.c # To double check
```

```
    joe moe doe boe foe
joe   0   0   0   0   0
moe   1   0   1   0   0
doe   1   1   0   0   1
boe   0   1   0   0   0
foe   0   0   0   0   0
```

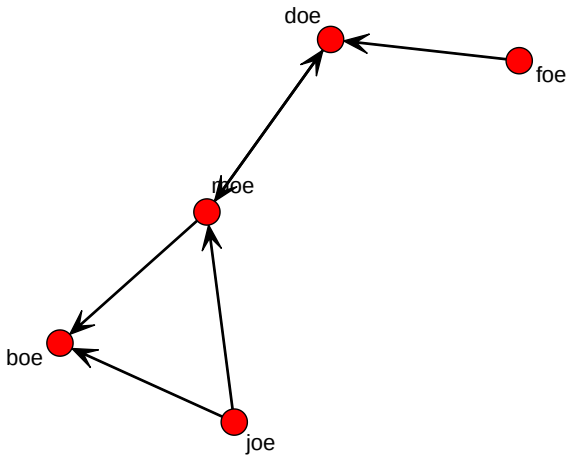
Now, let's create the respective **{statnet}** network objects from these matrices:

```
  net.object.r <- network(net.matrix.r, matrix.type = "adjacency")
  class(net.object.r) # Check it out
```

```
[1] "network"
```

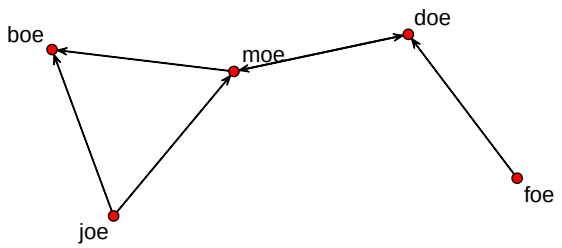
Now let's plot the network with **gplot()**, which is a function in the {statnet} library, which requires a network object to work. It works similar to plot() but it has its own attributes. Check ?gplot(). The vertex.col attribute is for node colors.

```
  gplot(net.object.r, vertex.col = "red", displaylabels = T)
```



The `plot()` function works similarly, but not as nice

```
plot(net.object.r, vertex.col = "red", displaylabels = T)
```

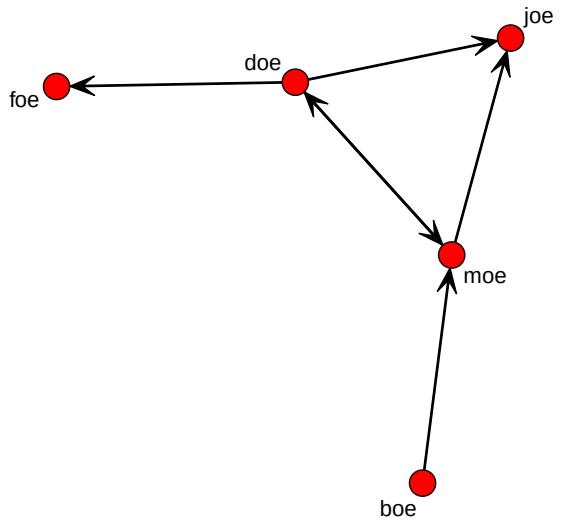


Asymmetric (i.e., Directed) vs. Symmetric (i.e., Undirected) Networks

Asymmetric matrices yield **directed** networks. This means that the relationship A to B is NOT the same as B to A. For example, John may like Judy, but Judy may not like John. In this case, the graph edges will contain arrows indicating the direction of the relationship. In an asymmetric adjacency matrix, the relationship is interpreted from the row member to the column member.

Notice how when we read the network object created column-wise, the direction of the arrows gets inverted.

```
net.object.c <- network(net.matrix.c, matrix.type = "adjacency")
gplot(net.object.c, vertex.col = "red", displaylabels = T)
```

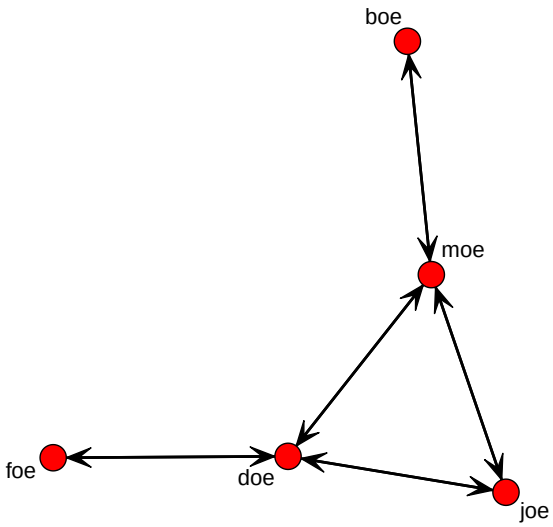


Notice how the arrows changed directions between the two network graphs above.

To create symmetric or undirected networks, use the attribute `directed = F`

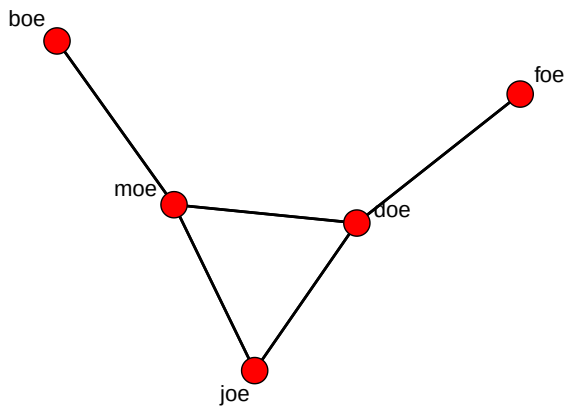
```
net.object.und <- network(net.matrix.c,
                          matrix.type = "adjacency",
                          directed = F)

gplot(net.object.und,
      vertex.col = "red",
      displaylabels = T)
```



You will notice that the edges now have arrows at both ends. But undirected networks should not have arrows. This happened because the network itself is undirected, but the graph has been specified as a directed graph or "digraph", which is the default. To change it to undirected, use gmode="graph").

```
gplot(net.object.und,
      vertex.col = "red", displaylabels = T, gmode = "graph")
```



You can also read the sociomatrix or data frame from a CSV file:

```
net.data.f <- read.table("NetMat.csv",
                         header = T, row.names = 1, sep = ",")

class(net.data.f) # Not a network object yet, just a data frame
```

```
[1] "data.frame"
```

Creating a Network Object

A sociomatrix defines a network, but a matrix object is not a network object yet. We need to create the network object, to which we will be able to add node and edge attributes later on. Since net.data.f is a dataframe, we need to first convert it into a matrix with the as.matrix() function.

```
s.net <- network(as.matrix(net.data.f, matrix.type = "adjacency"))
class(s.net) # A network object
```

```
[1] "network"
```

```
summary(s.net) # Display the adjacency matrix (or sociomatrix)
```

```
Network attributes:
  vertices = 5
  directed = TRUE
  hyper = FALSE
```

```
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
 total edges = 6
   missing edges = 0
   non-missing edges = 6
 density = 0.3

Vertex attributes:
  vertex.names:
   character valued attribute
   5 valid vertex names

No edge attributes

Network adjacency matrix:
        Alberto Jay Engin Joseph Mahsa
Alberto       0   1     1      0     0
Jay           0   0     1      1     0
Engin         0   1     0      0     0
Joseph        0   0     0      0     0
Mahsa         0   0     1      0     0
```

```
str(s.net) # Note that the network object is actually a list
```

```
List of 5
 $ mel:List of 6
  ..$ :List of 3
  .. ..$ inl : int 2
  .. ..$ outl: int 1
  .. ..$ atl :List of 1
  .. .. ..$ na: logi FALSE
  ..$ :List of 3
  .. ..$ inl : int 2
  .. ..$ outl: int 3
  .. ..$ atl :List of 1
  .. .. ..$ na: logi FALSE
  ..$ :List of 3
  .. ..$ inl : int 3
  .. ..$ outl: int 1
  .. ..$ atl :List of 1
  .. .. ..$ na: logi FALSE
  ..$ :List of 3
  .. ..$ inl : int 3
  .. ..$ outl: int 2
  .. ..$ atl :List of 1
  .. .. ..$ na: logi FALSE
  ..$ :List of 3
  .. ..$ inl : int 3
  .. ..$ outl: int 5
  .. ..$ atl :List of 1
  .. .. ..$ na: logi FALSE
  ..$ :List of 3
  .. ..$ inl : int 4
  .. ..$ outl: int 2
```

```
  .. ..$ atl :List of 1
  .. .. ..$ na: logi FALSE
 $ gal:List of 7
  ..$ n        : num 5
  ..$ mnext    : int 7
  ..$ directed : logi TRUE
  ..$ hyper    : logi FALSE
  ..$ loops    : logi FALSE
  ..$ multiple : logi FALSE
  ..$ bipartite: logi FALSE
 $ val:List of 5
  ..$ :List of 2
  .. ..$ na          : logi FALSE
  .. ..$ vertex.names: chr "Alberto"
  ..$ :List of 2
  .. ..$ na          : logi FALSE
  .. ..$ vertex.names: chr "Jay"
  ..$ :List of 2
  .. ..$ na          : logi FALSE
  .. ..$ vertex.names: chr "Engin"
  ..$ :List of 2
  .. ..$ na          : logi FALSE
  .. ..$ vertex.names: chr "Joseph"
  ..$ :List of 2
  .. ..$ na          : logi FALSE
  .. ..$ vertex.names: chr "Mahsa"
 $ iel:List of 5
  ..$ : int(0)
  ..$ : int [1:2] 2 1
  ..$ : int [1:3] 3 5 4
  ..$ : int 6
  ..$ : int(0)
 $ oel:List of 5
  ..$ : int [1:2] 3 1
  ..$ : int [1:2] 6 4
  ..$ : int 2
  ..$ : int(0)
  ..$ : int 5
 - attr(*, "class")= chr "network"
```

You can read list elements like this:

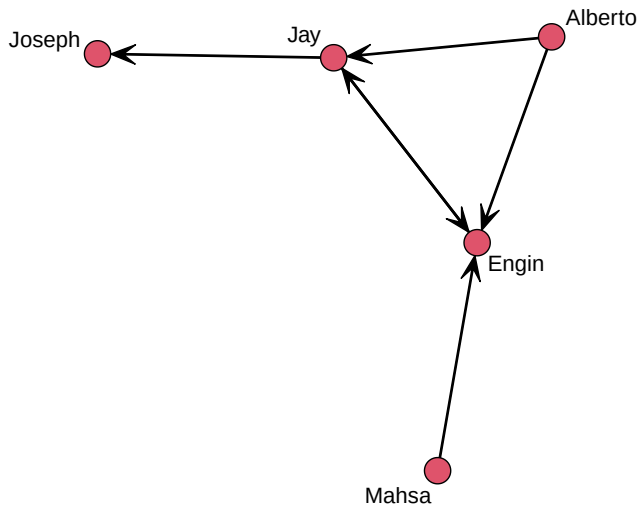
```
s.net$gal$directed # gal means "graph attribute list"
```

```
[1] TRUE
```

```
s.net$val[[2]]$vertex.names # 2nd element of $val list
```

```
[1] "Jay"
```

```
gplot(s.net, vertex.col = 2, displaylabels = T) # Plot the network
```



If for some reason the row names are not in the first column, you can change the row.names value to match the column where the names are. For example, if the names are in the 3rd row:

```
net.data.f2 <- read.table("NetMat2.csv",
                          header = T,
                          row.names = 3,
                          sep = ",")

net.data.f2 # Check it out
```

```
        Age Income Alberto Jay Engin Joseph Mahsa
Alberto  20  10000       0   1     1      0     0
Jay      25  32000       0   0     1      1     0
Engin    32   5000       0   1     0      0     0
Joseph   11 200000       0   0     0      0     0
Mahsa    79     45       0   0     1      0     0
```

As you can see, this data frame includes **Age** and **Income**, which are **vertex attributes**. These are useful, but don't belong in a sociomatrix. Let's read the data into appropriate objects:

```
net.Age <- net.data.f2[, 1] # Read age from all rows of first column
net.Age <- net.data.f2$Age # Alternatively, read the vector net.Age

net.Income <- net.data.f2[, 2] # Read income of second column
net.Income <- net.data.f2$Income # Alternatively net.Income
```

Then read the sociomatrix from the data frame, except columns 1-3

```
  net.data.f2.soc <- net.data.f2[, -1:-3]
  net.data.f2.soc # Check it out
```

```
        Jay Engin Joseph Mahsa
Alberto   1     1      0     0
Jay       0     1      1     0
Engin     1     0      0     0
Joseph    0     0      0     0
Mahsa     0     1      0     0
```

```
  class(net.data.f2.soc) # It's a data frame
```

```
[1] "data.frame"
```

```
  net.mat.2 <- as.matrix(net.data.f2.soc) # Do this if you need a matrix
  net.mat.2 # Check it out
```

```
        Jay Engin Joseph Mahsa
Alberto   1     1      0     0
Jay       0     1      1     0
Engin     1     0      0     0
Joseph    0     0      0     0
Mahsa     0     1      0     0
```

```
  class(net.mat.2)
```

```
[1] "matrix" "array"
```

IMPORTANT: At first glance, it would appear that matrix, data frame and network objects are just about the same thing. But network objects are much more complex. In this example, we have created a very simple network object, but network objects can be very rich in data with multiple node, edge and network attributes. More on this later.

KEEP IN MIND: some {statnet}, {igraph} and other SNA functions require matrices, other require data frames and others require network objects. So, you need to review the library documentation carefully and use the right type of object.

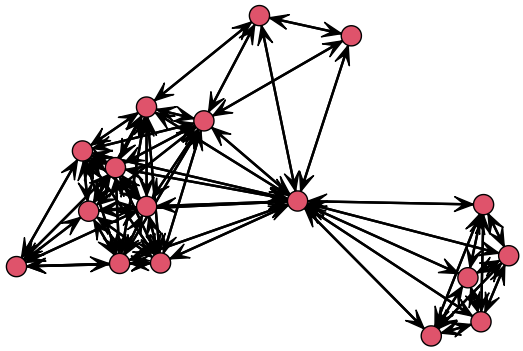
{statnet} Network Plots

Lets access the **Bali** terrorist network data

```
  library(UserNetR) # Contains the Bali terrorist network data
```

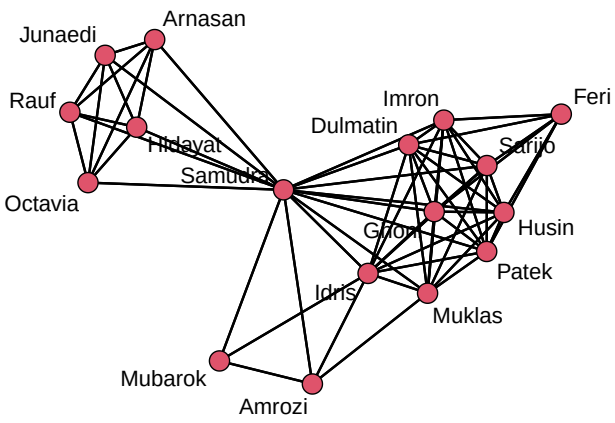
Let's plot the network:

```
  data(Bali) # Load the Bali Terrorist Network data set and plot it:
  gplot(Bali) # Not too friendly
```



Let's add vertex labels to the plot:

```
gplot(Bali,
      main = "Bali Terrorist Network",
      gmode = "graph",
      displaylabels = T)
```



There are some cool network graphics libraries. Let's look at the graph in 3D (click on any node in the popup graph and rotate it as you wish).

```
library(rgl) # Contains the gplot3d() function

gplot3d(Bali,
        gmode = "graph",
        mode = "kamadakawai",
        displaylabels = T,
        edge.lwd = 0.01)
```

Note: `mode = "kamadakawai"` is one of many graph layouts you can use to render a network. We will discuss the various graph layouts available in depth later in the semester.

{igraph} Network Objects

Detach `{statnet}` if loaded before loading `{igraph}`

```
detach(package:statnet)
library(igraph)
```

Creating an {igraph} object from a sociomatrix

First, create (or read) the sociomatrix:

```
net.mat <- rbind(c(0, 1, 1, 0, 0),
                 c(0, 0, 1, 1, 0),
                 c(0, 1, 0, 0, 0),
                 c(0, 0, 0, 0, 0),
                 c(0, 0, 1, 0, 0))

rownames(net.mat) <- # Let's add row an column vertex names:
  colnames(net.mat) <-
  c("joe", "moe", "doe", "boe", "foe")
```

Then, create the {igraph} network object with the graph_from_adjacency_matrix() function:

```
ig.net <- graph_from_adjacency_matrix(net.mat)
class(ig.net) # An {igraph} network object
```

```
[1] "igraph"
```

```
summary(ig.net) # Check out a summary of its contents
```

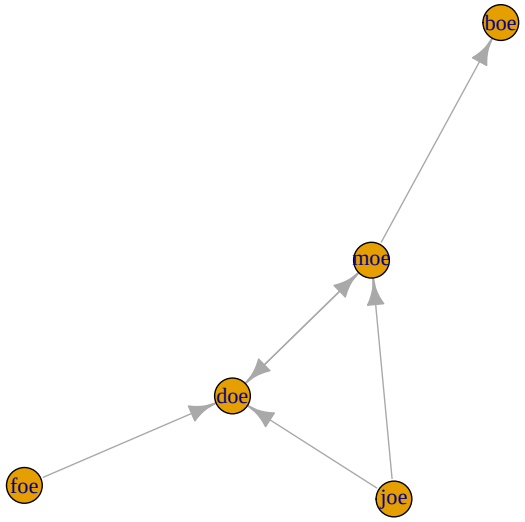
```
IGRAPH 6108287 DN-- 5 6 --
+ attr: name (v/c)
```

IGRAPH is an ID code, D is for directed, 5 nodes, 6 ties

```
ig.net # Check out the graph contents, who is connected to who
```

```
IGRAPH 6108287 DN-- 5 6 --
+ attr: name (v/c)
+ edges from 6108287 (vertex names):
[1] joe->moe joe->doe moe->doe moe->boe doe->moe foe->doe
```

```
plot(ig.net) # gplot() is not an {igraph} function, use plot() instead
```



Graph Formulas

You can also create `{igraph}` network objects directly with the `graph.formula()` as follows:

Undirected Graphs: Connect the nodes with a dash `-`

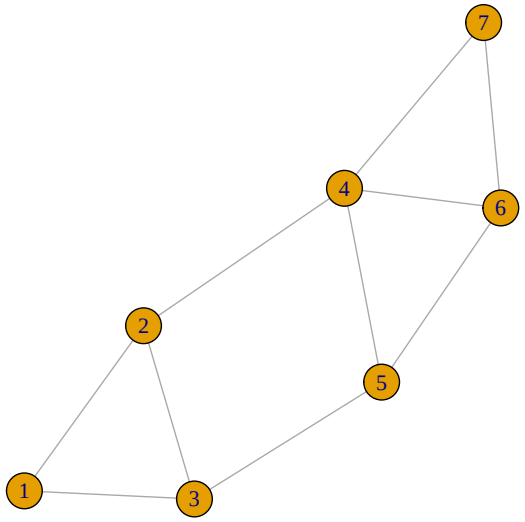
```
u.ig <- graph.formula(1-2, 1-3, 2-3, 2-4, 3-5, 4-5, 4-6, 4-7, 5-6, 6-7) # Ties
V(u.ig) # Vertices
```

```
+ 7/7 vertices, named, from 6125aa2:
[1] 1 2 3 4 5 6 7
```

```
E(u.ig) # Edges
```

```
+ 10/10 edges from 6125aa2 (vertex names):
 [1] 1--2 1--3 2--3 2--4 3--5 4--5 4--6 4--7 5--6 6--7
```

```
plot(u.ig)
```

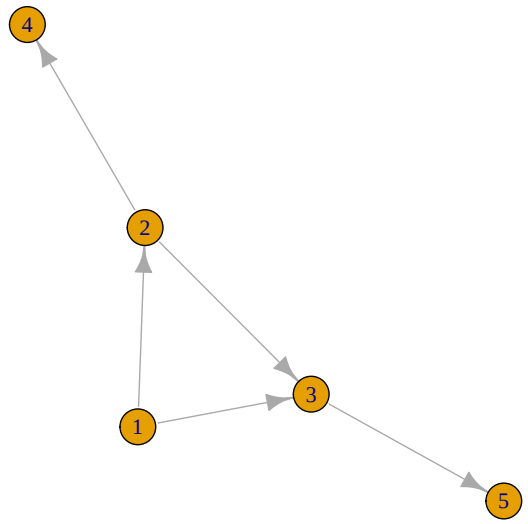


Directed Graphs: Connect nodes in the -+ direction

```
d.ig <- graph.formula(1-+2, 1-+3, 2-+3, 2-+4, 3-+5) # Directed graph, - to +
E(d.ig) # Edges
```

```
+ 5/5 edges from 6128ac0 (vertex names):
[1] 1->2 1->3 2->3 2->4 3->5
```

```
plot(d.ig)
```



Add names to the nodes or vertices using the `$name` attribute

```
V(d.ig)$name <- c("Alberto", "Mark", "Shawn", "Jay", "Engin")
get.edgelist(d.ig) # Edges in 2-column matrix formula
```

```
     [,1]      [,2]
[1,] "Alberto" "Mark"
[2,] "Alberto" "Shawn"
[3,] "Mark"    "Shawn"
[4,] "Mark"    "Jay"
[5,] "Shawn"   "Engin"
```

```
get.adjacency(d.ig) # Adjacency matrix -- notice that no ties are null
```

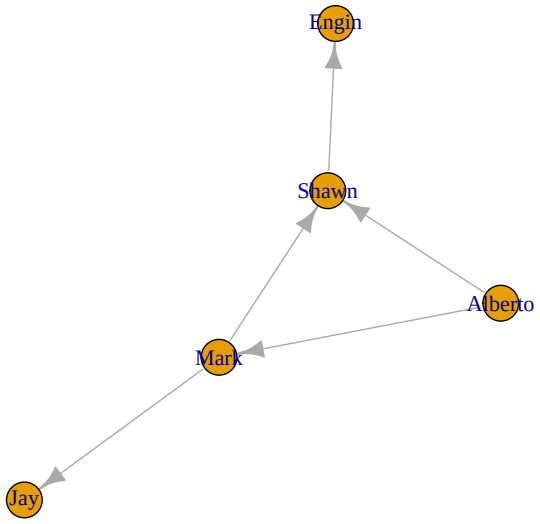
```
5 x 5 sparse Matrix of class "dgCMatrix"
        Alberto Mark Shawn Jay Engin
Alberto       .    1     1   .     .
Mark          .    .     1   1     .
Shawn         .    .     .   .     1
Jay           .    .     .   .     .
Engin         .    .     .   .     .
```

Also, the default is sparse = T, which yields a sparse matrix

```
get.adjacency(d.ig, sparse = F) # Adjacency matrix with no ties = 0's
```

```
        Alberto Mark Shawn Jay Engin
Alberto       0    1     1   0     0
Mark          0    0     1   1     0
Shawn         0    0     0   0     1
Jay           0    0     0   0     0
Engin         0    0     0   0     0
```

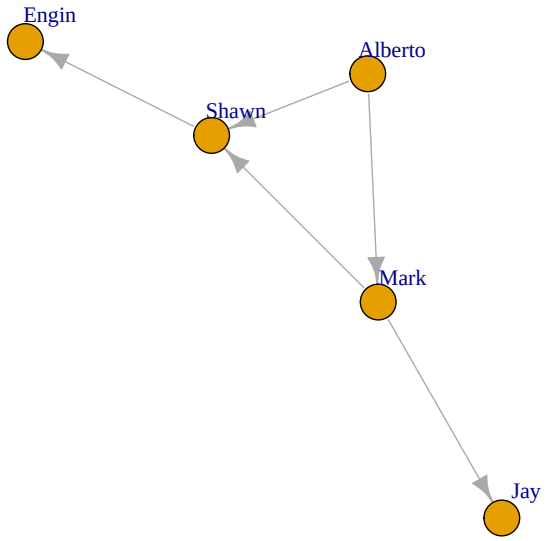
```
plot(d.ig)
```



{igraph} Plots

Simple plot:

```
plot(d.ig,
     vertex.label.dist = 2) # To position labels above the vertex
```



3D Plot:

```
rglplot(d.ig) # Interactive plot
```

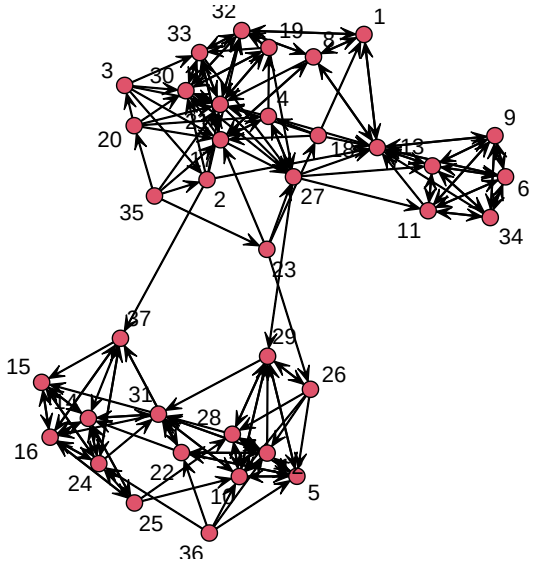
{statnet} Vertex Attributes

Quick Vertex Attributes

```
detach(package:igraph) # Only run if igraph was loaded last
library(statnet)
library(UserNetR) # Contains the middleschool data set
```

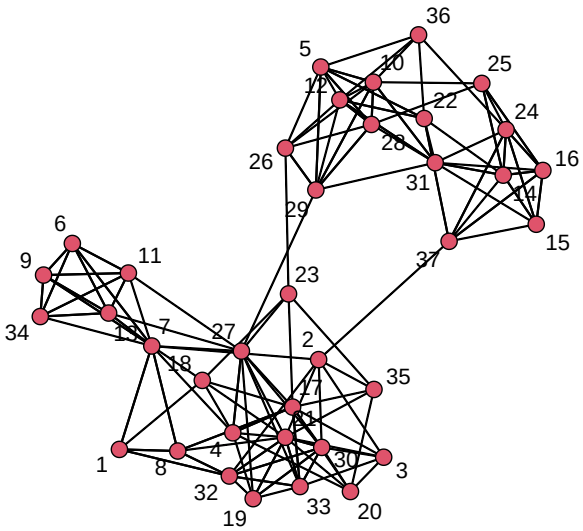
Let's plot the **middleschool** network of **directed** friendship ties in a California school. The direction of the ties indicates who a student (i.e., tie sender) views who (i.e., tie receiver) as a friend.

```
gplot(middleschool,
      displaylabels = T,
      vertex.cex = 1)
```



The graph default `gmode` parameter is `"digraph"` (i.e., directed graph), that is, with arrows, therefore, I did not included it in the `gplot()` function above. To render the graph without arrows, include the parameter to `"gmode = graph"` (i.e., undirected graph):

```
gplot(middleschool,
      displaylabels = T,
      gmode = "graph",
      vertex.cex = 1)
```



Centrality

There are various network attributes that we will cover in this course, which can provide useful information about individual vertices (or network actor nodes), edges (or network ties), sub-groups of vertices, or the whole network.

A network **actor prominence attribute** is a **vertex-level** attribute that provides a measure of the importance of the actor in the network, or how visible it is to others on the network. **Centrality** is a measure of an actor's **prominence** on the network, and it can be of various types (e.g., degree, betweenness, closeness, etc.). In this section I provide a brief introduction to and overview of a few measures. I will discuss these measures in more detail later in the course.

While centrality is a straightforward concept, its measurement and interpretation of its attributes can be a bit challenging because these measures change depending on whether the network is **binary** or **weighted**, and whether it is **directed** or **undirected**. I provide some quick definitions and calculations below.

Degree centrality

Degree centrality is one of the most widely used measures of network actor prominence. A network actor with more ties is said to be more prominent or visible to others in the network. I use the **middleschool** network to illustrate some of these measures.

If a network is **undirected** and **binary**, the degree centrality of a vertex is simply a count of the number of ties or edges connected to that vertex. If a network is **directed** and **binary**, the degree centrality of a vertex is the total number of outgoing plus incoming ties. For example, the **middleschool** network is directed, so outgoing ties are different than incoming ties. This is also called **"freeman"** centrality, which is the default attribute, so it can be included or omitted.

```
deg.un <- degree(middleschool, cmode = "freeman") # Or
deg.un <- degree(middleschool)
deg.un
```

```
 [1]  7  7  6  9 11 10 16  9 10 12 10 11 11 12  8 11 13  6  8  6 16  9  5 11  7
[26]  7 12 11 11 13 12 12 11  9  5  5  9
```

In directed networks, one can differentiate between **outgoing** or **incoming** ties. For example, in a friendship network, an actor with lots of incoming ties is an actor that many other actors consider a friend. An actor with lots of outgoing ties is an actor that tends to befriend other actors. For example:

```
deg.un <- degree(middleschool, gmode = "graph", cmode = "outdegree")
deg.un
```

```
 [1]  4  2  1  4  6  5 11  4  5  8  5  6  6  7  5  6  8  1  3  1 11  4  1  6  2
[26]  2  7  6  6  8  7  7  6  4  0  0  4
```

```
cat("\n")
```

```
deg.di <- degree(middleschool, gmode = "digraph")
deg.di
```

```
 [1]  7  7  6  9 11 10 16  9 10 12 10 11 11 12  8 11 13  6  8  6 16  9  5 11  7
[26]  7 12 11 11 13 12 12 11  9  5  5  9
```

If you want to identify the outgoing and incoming ties for an actors, you can observe which of its ties are 1's and which ar 0's, for example:

```
as.sociomatrix(middleschool)[3, ]   # Outgoing ties from actor 3, i.e., row 3
```

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0
27 28 29 30 31 32 33 34 35 36 37
 0  0  0  1  0  0  1  0  0  0  0
```

```
cat("\n")
```

```
as.sociomatrix(middleschool)[, 3]   # Incoming ties to actor 3, i.e., column 3
```

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0
27 28 29 30 31 32 33 34 35 36 37
 0  0  0  0  0  0  0  0  0  0  0
```

An important issue when analyzing degree centralities is that larger networks will generally have vertices with more ties, simply because they are larger, so there is a higher likelihood of having more ties. This makes it difficult to compare the centralities of members of networks of different sizes. A way to overcome this issue is to use the parameter `rescale = T`, which will **rescale** all the centrality values to add up to 1. So, the central actor of a network with a **"star"** configuration in which the central actor has ties to all other members as a pure **"structural hole"** (other members have no ties with each other), will have a re-scaled degree centrality of 1, regardless of network size. All the peripheral members will have a rescaled degree centrality of 0. In contrast, in a fully connected network in which all members are connected with each other, all members will have a re-scaled degree centrality of 1/n. Rescaled centralities can be used to compare actor prominence across multiple networks of different sizes.

```r
deg.un.r <- degree(middleschool, rescale = T)

round(deg.un.r, 3)
```

```
 [1] 0.020 0.020 0.017 0.025 0.031 0.028 0.045 0.025 0.028 0.034 0.028 0.031
[13] 0.031 0.034 0.022 0.031 0.036 0.017 0.022 0.017 0.045 0.025 0.014 0.031
[25] 0.020 0.020 0.034 0.031 0.031 0.036 0.034 0.034 0.031 0.025 0.014 0.014
[37] 0.025
```

```r
cat("\n")
```

```r
cat("Sum of all re-scaled centralities = ", sum(deg.un.r))
```

```
Sum of all re-scaled centralities =  1
```

When a network is **weighted**, the degree centrality computations are similar to those described above, except that rather than counting ties, the respective tie weights are added up. This can be confusing because one heavy tie can make an actor seem to be prominent, when other members may have many more ties but lighter. I defer the discussion of weighted centralities for later.

Eigenvector Centrality

There may be network actors with many connections to unimportant or non-prominent actors who themselves don't have a lot of ties. In contrast, one actor may have a few ties to actors with lots of ties to other members, and it may be considered prominent because of that (i.e., lots of ties to unimportant actors s. few ties to very powerful actors). In such cases, degree centrality can be deciving because if an actor has many ties to isolates or actors with very little connections, that actor will not have a lot of influence on the network. **Eigengvector** centrality overcomes this issue by weighting each tie by the degree centrality of its adjacent actor connections. This way, a tie to an actor that has 3 other ties will carry triple the weight than a tie to an actor that has only one other tie. Later on, I will discuss the related concept of Ron Burt's **structural holes**. A **structural hole** is a network actor that has ties to members that are not tied to each other, which minimizes tie redundancy. A structural hole actor has **efficient** non-redundant ties. More on this later.

The eigenvector calculations have the same differentiation for binary, weighted, directed, and undirected networks. Also, the eigenvectors are automatically re-scaled so that the sum of the squares of all eigenvector centralities is equal to 1. Eigenvectors and eigenvalues are central to many co-variance based statistics and one of the main properties of eigenvectors is that the sum of their corresponding eigenvalues squared equals 1, thus the rescaling of eigenvector centralities..

```r
options(scipen = 4) # To minimize the use of scientific notation
eig <- evcent(middleschool)
round(eig, 3)
```

```
 [1] 0.127 0.173 0.208 0.216 0.000 0.258 0.226 0.199 0.258 0.000 0.258 0.000
[13] 0.258 0.000 0.000 0.000 0.212 0.203 0.210 0.217 0.210 0.000 0.126 0.000
[25] 0.000 0.000 0.197 0.000 0.000 0.210 0.000 0.196 0.210 0.258 0.192 0.000
[37] 0.000
```

Betweenness Centrality

Betweenness centrality measures the extent to which actors are in between other actors. An actor with a high betweenness centrality has the ability to bridge different parts of the network. While high betweenness actors have strong power to bridge and broker connections, they are also viewed as points of vulnerability in the network because if they exit the network this will create connectivity problems. To understand betweeness centrality we need to

understand the concept of actors' **paths** and **distance** first. Unless a network is disconnected, any two actors have one or more paths to and from each other, going through other actors to reach each other. The distance between two actors is the shortest path (i.e., fewer connecting ties) between the two. This shortest path is also called the **geodesic** for the pair. The betweenness centrality of an actor is the number of times the actor or vertex is in the shortest path between all dyads in the network. If a dyad has more than one shortest path, the count is divided by the number of geodesics spanned by the dyad. As with other centrality measures, betweenness centrality can be binary, raw, normalized, valued, directed or undirected.

```
betw <- betweenness(middleschool)
round(betw, digits = 3)
```

```
 [1]   9.233  31.777   8.309  19.923  42.029   3.560 163.912 141.267   3.560
[10]  15.157   9.244  16.551  12.804  35.405   7.713  39.442  93.881   4.333
[19]   2.083   1.750 101.125  33.834   8.339  10.947  12.925  20.818 293.423
[28]  14.593 250.443  22.320 141.830  29.633  16.625   0.000   0.000   0.000
[37]  38.214
```

Closeness Centrality

This centrality value measures how close an actor is to all other actors in the network. It simply sums the distances from the actor to all other actors. Because closeness is the opposite of distance, this resulting closeness centrality is the inverse of this sum, so that large values indicate more closeness and the other way around. Another important issue with closeness centrality is that the distance between 2 disconnected actors is infinite, so closeness cannot be computed when there are disconnected components or isolates, and therefore, it is not very useful sometimes.

```
clos <- closeness(middleschool, gmode = "graph")
round(clos, 3)
```

```
 [1] 0.360 0.474 0.387 0.419 0.379 0.336 0.474 0.375 0.336 0.383 0.396 0.391
[13] 0.396 0.356 0.346 0.350 0.424 0.409 0.400 0.356 0.462 0.360 0.429 0.356
[25] 0.316 0.391 0.522 0.379 0.468 0.444 0.383 0.387 0.404 0.336 0.404 0.316
[37] 0.424
```

Take a look

```
round(cbind(deg.un, deg.di, deg.un.r, betw, eig, clos), 3)
```

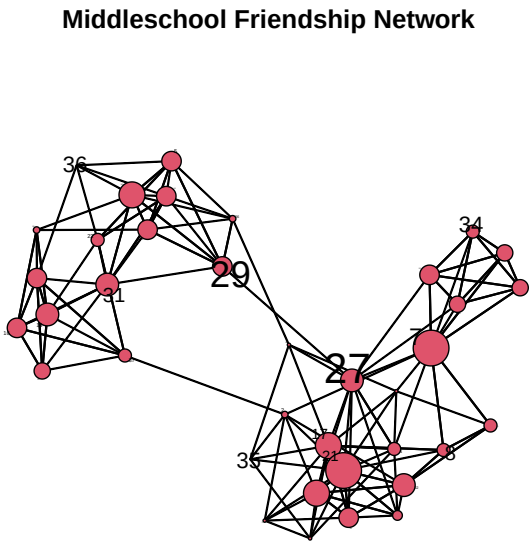
```
      deg.un deg.di deg.un.r    betw   eig  clos
 [1,]      4      7    0.020   9.233 0.127 0.360
 [2,]      2      7    0.020  31.777 0.173 0.474
 [3,]      1      6    0.017   8.309 0.208 0.387
 [4,]      4      9    0.025  19.923 0.216 0.419
 [5,]      6     11    0.031  42.029 0.000 0.379
 [6,]      5     10    0.028   3.560 0.258 0.336
 [7,]     11     16    0.045 163.912 0.226 0.474
 [8,]      4      9    0.025 141.267 0.199 0.375
 [9,]      5     10    0.028   3.560 0.258 0.336
[10,]      8     12    0.034  15.157 0.000 0.383
[11,]      5     10    0.028   9.244 0.258 0.396
[12,]      6     11    0.031  16.551 0.000 0.391
[13,]      6     11    0.031  12.804 0.258 0.396
[14,]      7     12    0.034  35.405 0.000 0.356
[15,]      5      8    0.022   7.713 0.000 0.346
[16,]      6     11    0.031  39.442 0.000 0.350
```

```
[17,]      8     13    0.036  93.881 0.212 0.424
[18,]      1      6    0.017   4.333 0.203 0.409
[19,]      3      8    0.022   2.083 0.210 0.400
[20,]      1      6    0.017   1.750 0.217 0.356
[21,]     11     16    0.045 101.125 0.210 0.462
[22,]      4      9    0.025  33.834 0.000 0.360
[23,]      1      5    0.014   8.339 0.126 0.429
[24,]      6     11    0.031  10.947 0.000 0.356
[25,]      2      7    0.020  12.925 0.000 0.316
[26,]      2      7    0.020  20.818 0.000 0.391
[27,]      7     12    0.034 293.423 0.197 0.522
[28,]      6     11    0.031  14.593 0.000 0.379
[29,]      6     11    0.031 250.443 0.000 0.468
[30,]      8     13    0.036  22.320 0.210 0.444
[31,]      7     12    0.034 141.830 0.000 0.383
[32,]      7     12    0.034  29.633 0.196 0.387
[33,]      6     11    0.031  16.625 0.210 0.404
[34,]      4      9    0.025   0.000 0.258 0.336
[35,]      0      5    0.014   0.000 0.192 0.404
[36,]      0      5    0.014   0.000 0.000 0.316
[37,]      4      9    0.025  38.214 0.000 0.424
```

Visualizing Centralities in a Graph

One of the great benefits of computing centralities for all the vertices in a network is that the results are returned as vectors, with one element per vertex. These vectors can be used as graph parameters to size vertices and/or labels based on their centralities. For example:

```
gplot(middleschool,
      main = "Middleschool Friendship Network",
      gmode = "graph",
      vertex.cex = deg.un/5,
      label.cex = betw/150,
      displaylabels = T)
```



In the example above, I used the attribute `vertex.cex =` to size the vertices. We can use a fixed value (e.g., `vertex.cex = 1.5`) if we just want to enlarge or reduce the size of all vertices. In the example above, I used `vertex.cex = deg.un/5` instead, so that the vertices are sized using their undirected degree centrality, although you can see that I scale the size back dividing by 5 because the vertices were too large for visualization. Similarly, I used the betweeness centrality divided by 150 to size the vertex labels with the `label.cex =` parameter.

{igraph} Vertex Attributes

Let's do the same work I did above, but using `{igraph}` this time.

Quick Vertex Attributes

We first need to detach `{statnet}` and load `{igraph}`.

```
detach(package:statnet)
library(igraph)
```

Let's specify a graph

```
g <- graph.formula(1-2, 1-3, 2-3, 2-4, 3-5, 4-5, 4-6, 4-7, 5-6, 6-7)
```

One great advantage of `{igraph}` network objects is that you can store vertex and edge attributes in the network itself, thus reducing the need to us parameters when rendering graphs. There are also few attributes with reserved names you can use to assign common attributes to vertices, such as `$name` (vertex labels), `$color`, and `$size`, whose purpose is self-explanatory. But you can add any other vertex or edge attributes as needed, as with I illustrate with `$gender` below.

Let's store names for the vertices in the internal attribute `$name`:

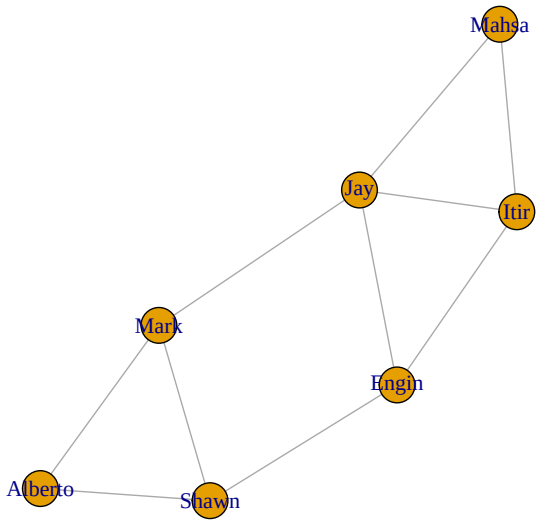
```
V(g)$name <- c("Alberto", "Mark", "Shawn", "Jay", "Engin", "Itir", "Mahsa")
```

Now, let's store gender attributes for of the vertices in a defined attribute we are naming `$gender`.

```
V(g)$gender <- c("M", "M", "M", "M", "M", "F", "F" )
```

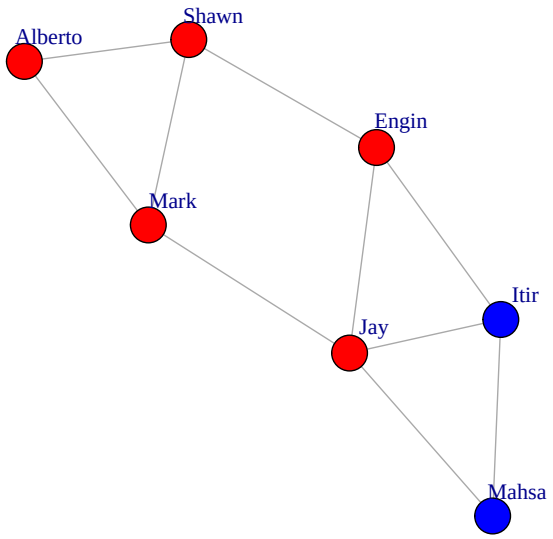
Let's first plot without colors

```
plot(g)
```



Let's now add color based on gender and store them in the $color attribute

```
V(g)$color <- ifelse(V(g)$gender == "M", "red", "blue")
plot(g, vertex.label.dist = 2)
```



Centrality

Centrality Functions

```
deg <- degree(g) # Degree centrality
betw <- betweenness(g) # Betweenness centrality
clos <- closeness(g) # Closeness centrality
eig <- eigen_centrality(g)$vector # Eigenvector centrality
# Note: the eigenvector centralities are stored in the $vector attribute
pgrk <- page_rank(g)$vector # PageRank centrality

round(cbind(deg, betw, clos, eig, pgrk), 3)
```

```
        deg  betw  clos   eig  pgrk
Alberto   2 0.000 0.083 0.468 0.107
Mark      3 3.333 0.111 0.719 0.150
Shawn     3 2.000 0.100 0.678 0.151
Jay       4 5.167 0.125 1.000 0.192
Engin     3 2.667 0.111 0.837 0.147
Itir      3 0.833 0.100 0.820 0.148
Mahsa     2 0.000 0.083 0.610 0.104
```

Note: The **PageRank** centrality computed above is similar to Eigenvector centality, except that the weights are based on incoming ties only. This method was created by Google co-founder **Larry Page** (thus the term "Page") to rank web pages based on their popularity associated with links pointing to a web page from other web pages. PageRank centrality is a good measure of widespread popularity in a network. You can find an actor's neighbors as follows:

```
neighbors(g, 3) # Member 3's neighbors
```

```
+ 3/7 vertices, named, from 61beb5f:
[1] Alberto Mark    Engin
```

```
neighbors(g, "Shawn") # Shawn's neighbors
```

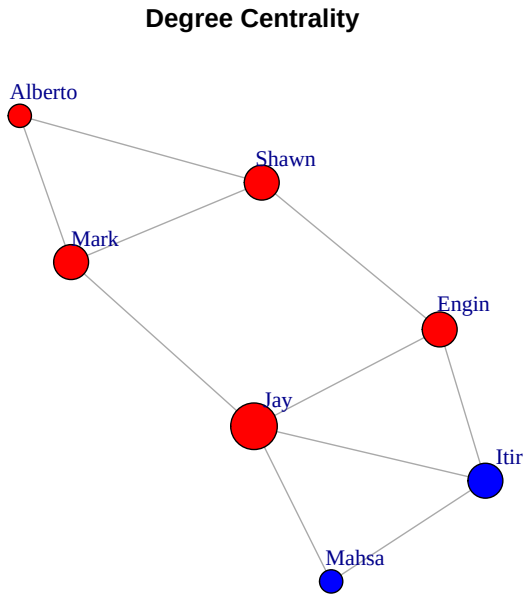
```
+ 3/7 vertices, named, from 61beb5f:
[1] Alberto Mark    Engin
```

Visualizing Centralities in a Graph

```
par(mfrow = c(2, 2)) # Let's divide the output into 2x2 first
```

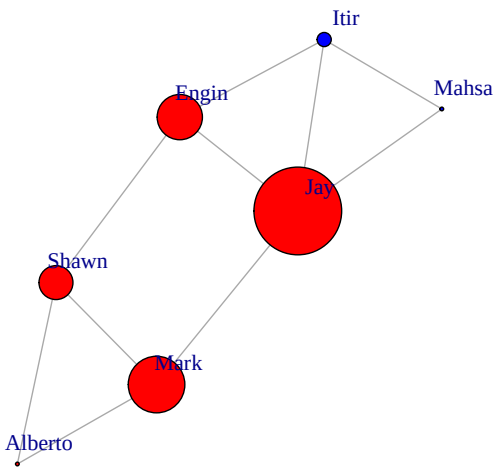
Sizing vertices at 5 times degree centrality:

```
V(g)$size <- 5 * deg
plot(g, vertex.label.dist = 2, main = "Degree Centrality")
```



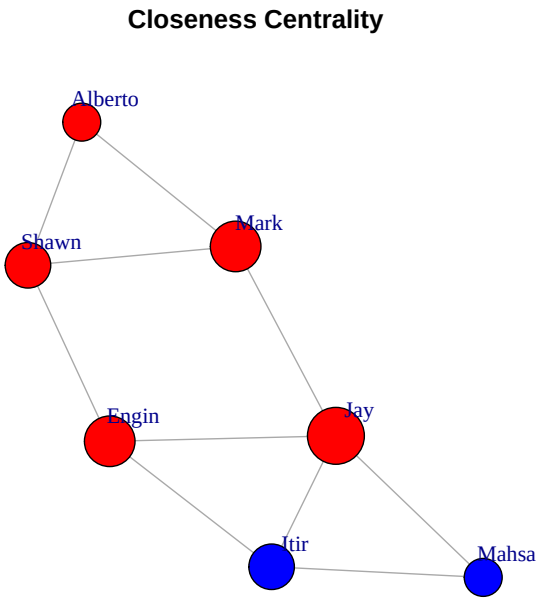
Sizing vertices at 8 times betweenness centrality:

```
V(g)$size <- 8 * betw
plot(g, vertex.label.dist = 2, main = "Betweeness Centrality")
```



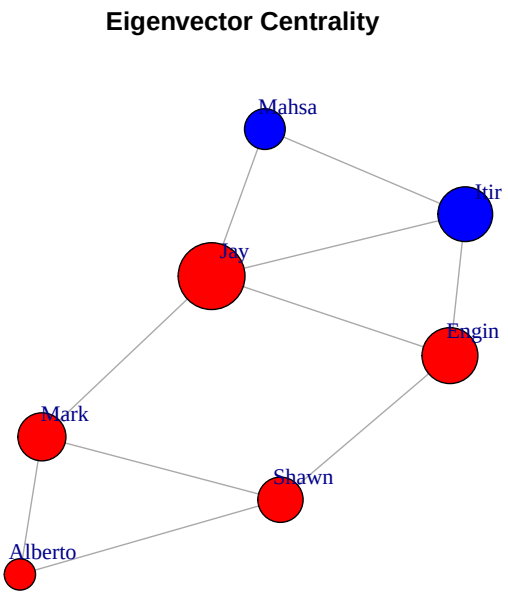
Sizing vertices at 200 times closeness centrality:

```
V(g)$size <- 200 * clos
plot(g, vertex.label.dist = 2, main = "Closeness Centrality")
```



Sizing vertices at 30 times eigenvector centrality:

```
V(g)$size <- 30 * eig
plot(g, vertex.label.dist = 2, main = "Eigenvector Centrality")
```



Let's reset the output back to to 1 row x 1 column

```
par(mfrow=c(1,1))
```

Converting {igraph} to/from {statnet} Objects

`{statnet}` functions don't work with `{igraph}` objects and the other way around. The `{intergraph}` package takes care of this issue and can convert data objects from one package to the other. If you need to create network objects with one package and work with them in another package, use the `{intergraph}` package to convert these objects as needed. We will see this in more detail later.

Let's start by creating a simple `{statnet}` network object. Let's load `{statnet}` first:

```
detach(package:igraph) # Detach only if {igraph} was loaded last
library(statnet)
```

Let's now create a simple binary matrix and then its corresponding `{statnet}` network object.

```
net.mat <- rbind(c(0,1,1,0,0), # Matrix object
                 c(0,0,1,1,0),
                 c(0,1,0,0,0),
                 c(0,0,0,0,0),
                 c(0,0,1,0,0))

s.net <- network(net.mat, matrix.type="adjacency") # Network object
class(s.net) # We just created a {statnet} network object
```

```
[1] "network"
```

```
  summary(s.net) # Display the adjacency matrix (or sociomatrix)
```

```
Network attributes:
  vertices = 5
  directed = TRUE
  hyper = FALSE
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
 total edges = 6
   missing edges = 0
   non-missing edges = 6
 density = 0.3

Vertex attributes:
  vertex.names:
   character valued attribute
   5 valid vertex names

No edge attributes

Network adjacency matrix:
  1 2 3 4 5
1 0 1 1 0 0
2 0 0 1 1 0
3 0 1 0 0 0
4 0 0 0 0 0
5 0 0 1 0 0
```

Let's now convert the {statnet} object to an {igraph} object:

```
  library(intergraph) # To manipulate network objects

  i.net <- asIgraph(s.net) # Converting the Network object to an {igraph} object
  class(i.net) # Check it out
```

```
[1] "igraph"
```

Let's convert the {igraph} object back to a {statnet} network object.

```
  s.net.new <- asNetwork(i.net)
  class(s.net.new) # Check it out
```

```
[1] "network"
```

Capturing Network Graphics in PDF or GIF Files

To generate a PDF file, first initialize the PDF file. This command opens a blank PDF file for you to add material to it:

```
pdf("Figure.pdf") # Or any file name you wish
# jpeg() instead of pdf() to create a JPG files

# From this point on, everything you graph will be captured in `Figure.pdf`:
plot(g, vertex.label.dist = 2, main = "Degree Centrality")

dev.off() # to close and generate the PDF file
shell.exec("Figure.pdf") # Check out the PDF file
```

Note: the `plot()` function did not display the graph because it was piped directly to the PDF file. You can render as many graphs and they will all be piped to the PDF (of JPG) file. Use `dev.off()` when done to close the PDF file.

Quick {statnet} Visualizations

```
library(UserNetR) # Contains the Moreno data set
```

Let's list Moreno's network vertex attribute names:

```
list.vertex.attributes(Moreno) # To list vertex attribute names
```

```
[1] "gender"       "na"           "vertex.names"
```

```
cat("\n")
```

```
get.vertex.attribute(Moreno, "gender") # To get specific node values
```

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

```
cat("\n")
```

```
get.vertex.attribute(Moreno, "vertex.names") # To get specific node values
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33
```

The `%e%` and `%v%` operators can also be used to retrieve edges and vertices from the network, respectively. For example:

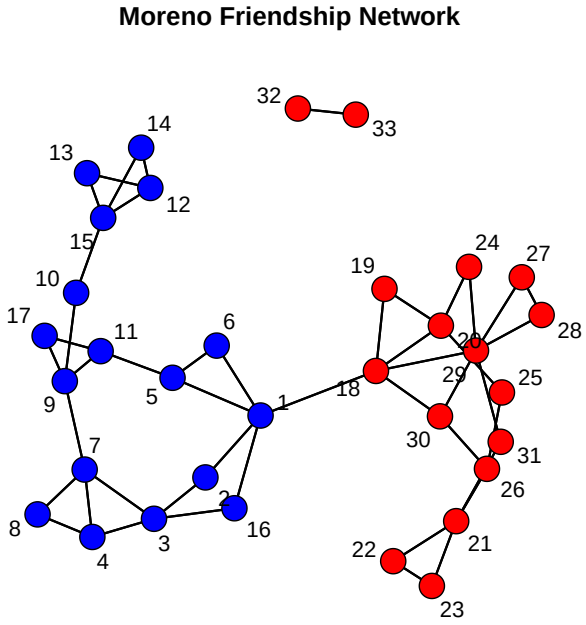
```
Moreno %v% "gender" # Will list genders of Moreno's actors
```

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

You can also store vertex (or edge) attributes in a vector and then plot the colorized network:

```r
gender.color <- ifelse(Moreno %v% "gender" == 1,
                       "red", "blue") # Color vector

gplot(Moreno,
      main = "Moreno Friendship Network",
      displaylabels = T,
      vertex.col = gender.color, # To colorize the net
      vertex.cex = 1.5,
      gmode = "graph")
```



Quick {igraph} Visualizations

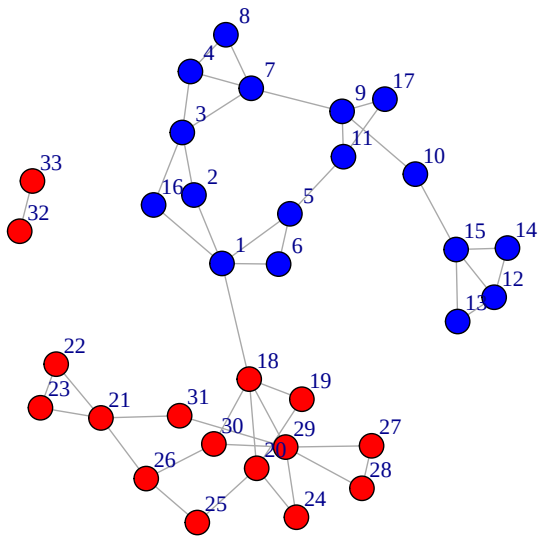
```r
detach(package:statnet) # Run only if `{statnet}` was loaded last
library(igraph)
library(intergraph) # To convert to {igraph} object
```

Let's now convert the network object to `{igraph}` and assign gender colors

```r
Moreno.ig <- asIgraph(Moreno)
V(Moreno.ig)$color <- gender.color # Assign node colors by gender
```

Most `{igraph}` plot vertex attributes names start with the word `vertex.` Notice also that, unlike `{statnet}`, we don't need to specify the vertex colors, because they are already stored in the network object's `$color` attribute

```
plot(Moreno.ig,
     vertex.size = 10,
     vertex.label.dist = 1.5,
     main = "Moreno Friendship Network")
```



We will discuss graph layouts in depth later, but notice below how the `layout =` parameter changes the graph layout. An important thing to note is that the layout only changes the arrangement of vertices and edges in a graph, but it does not alter the network in any way.

```
plot(Moreno.ig,
     vertex.size = 10,
     vertex.label.dist = 2,
     main = "Moreno Friendship Network (grid layout)",
     layout = layout_on_grid) # More on layouts later
```

Moreno Friendship Network (grid layout)