Data Science Notes

Andrea Mansi UniUD

last update: 23/10/2020 !!!!!!!!!!!!!!

Info about the notes

In these notes you will find the main contents related to the advanced data science course. Bibliografy:

- Lessons, materials and slides from Prof. Massimo Franceschet (Università degli Studi di Udine) - http://users.dimi.uniud.it/~massimo.franceschet/ns/plugandplay/ns.html
- Online R Documentation

This PDF is generated with Rstudio from RMarkdown sourcecode.

Those notes are organized into 4 sections:

1. The Tidyverse
2. Blockchain
3. Network Science
4. Text Mining

Useful contents (click for open in browser):

- R cheat sheet
- RStudio IDE cheat sheet
- R Markdown cheat sheet
- R Cookbook
- readr and tidyr cheat sheet
- Tibbles
- Data import with readr
- Tidy data with tidyr
- dplyr cheat sheet
- Data transformation with dplyr
- Joins with dplyr
- ggplot2 cheat sheet
- R Cookbook, Chapter 10
- Data visualization with ggplot
- Exploratory Data Analysis with dplyr and ggplot
- Graphics for communication
- Fronkonstin
- Notes on Linear Algebra and Matrix Theory
- Graph Theory

The Tidyverse

1. Introduction to Data Science

> "Data science is an exciting discipline that allows you to turn raw data into understanding, insight, and knowledge." **Hadley Wickham**
>
> "Instead of using data just to become more efficient, we can use data to become more humane and to connect with ourselves and others at a deeper level." **Giorgia Lupi**

Data Science main activities:

Import

- first you must import your data into R
- this typically means that you take data stored in a file, database, or web API, and load it into a data frame in R
- if you can't get your data into R, you can't do data science on it

Tidy

- once you've imported your data, it is a good idea to **tidy** it
- tidying your data means storing it in a consistent form that matches the semantics of the dataset with the way it is stored
- in brief, when your data is tidy, each column is a variable and each row is an observation
- tidy data is important because the consistent structure lets you focus on questions about the data, not fighting to get the data into the right form to answer your questions

Transform

- once you have tidy data, a common first step is to **transform** (or **query**) it
- transformation includes:
  - narrowing in on observations of interest (like all people in one city, or all data from the last year)
  - creating new variables that are functions of existing variables (like computing velocity from speed and time)
  - calculating a set of summary statistics (like counts or means)
- together, tidying and transforming are called **wrangling**, because getting your data in a form that's natural to work with often feels like a fight!

Visualize and model

- once you have tidy data with the variables you need, there are two main engines of knowledge generation: **visualisation** and **modelling**
- these have complementary strengths and weaknesses so any real analysis will iterate between them many times

Visualize

- **visualisation** is a fundamentally human activity
- good visualisation will show you things that you did not expect, or raise new questions about the data
- a good visualisation might also hint that you're asking the wrong question, or you need to collect different data
- visualisations can surprise you, but **don't scale** particularly well because they require a human to interpret them

Model

- **models** are complementary tools to visualisation
- the goal of a model is to provide a simple low-dimensional summary of a dataset
- ideally, the model will capture true **signals** (i.e. patterns generated by the phenomenon of interest) and ignore **noise** (i.e. random variation that you're not interested in)
- models are a fundamentally mathematical or computational tool, so they generally scale well
- but "*the map is not the territory*": every model makes assumptions, and these make a difference between reality and a model of reality

Communicate

- the last step of data science is **communication**, an absolutely critical part of any data analysis project
- it doesn't matter how well your models and visualisation have led you to understand the data unless you can also communicate your results to others, including the future you

2. A hasty tour inside R

Let's start with an introduction of R language:

R language

R is a free software environment for statistical computing and graphics. R is a needful language for the data scientist. Its strengths include:

1. **capability**: it offers a gargantuan set of functionalities
2. **community**: it has an elephantine, ever growing community of users
3. **performance**: it is lightning fast (when running in main memory)

RStudio

RStudio is an integrated development environment (IDE) for R. It includes:

- a console
- syntax-highlighting editor that supports direct code execution
- tools for plotting, history, debugging and workspace management

Help and packages

To read a specific command documentation

```
# help on log function
?log
```

R comes with a number of **packages**, some of them are loaded by default

```
# install or update a package (only once!)
install.packages("igraph")
# load a package (when you need it)
library(igraph)
# list all packages where an update is available
old.packages()
# update all available packages
update.packages()
```

Basic arithmetic and logic operators

- **arithmetic**: sum (`+`), minus (`-`), product (`*`), division (`/`), integer division (`%/%`), modulus (`%%`), exponent (`^`)
- **comparison**: equal (`==`), different (`!=`), less than (`<`), greater than (`>`), less than or equal to (`<=`), greater than or equal to (`>=`)
- **logic operators**: conjunction (`&`), disjunction (`|`), negation (`!`), exclusive disjunction (`xor`)

Special values

- the value `NA` (not available) is used to represent missing values;
- the value `NULL` is the null object (not to be confused with NULL in databases);
- the value `Inf` stands for positive infinity;
- the value `NaN` (not a number) is the result of a computation that makes no sense.

Special values: some calculus examples

```
NA & TRUE
```

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

```
NA & FALSE
```

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

```
NA | TRUE
```

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

```
NA | FALSE
```

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

```
!NA
```

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

```
2^1024
```

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

```
1/0
```

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

```
0 / 0
```

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

```
Inf - Inf
```

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

Variables

Of course, you may use **variables** to store values. There are 3 equivalent ways to assign a value to a variable:

```
x = 42  # the standard in many programming languages
x <- 42 # this is the politically correct one! (R standard)
42 -> x # used rarely

# print x
x
```

```
## [1] 42
```

```
# print structure of x (with type)
str(x)
```

```
##  num 42
```

Atomic types

R has four main **atomic types**:

```
# double (double-precision number)
x = 108.801

# integer (integer number)
x = 108L
```

```r
# character (a string of characters)
x = "108L"

# logical (a Boolean, either TRUE or FALSE)
x = TRUE

# typeof(x) for getting its type
typeof(x)
```

```
## [1] "logical"
```

Data structures

The main data structures used in R include:

- atomic vector
- list
- matrix
- data frame

Atomic vectors

A **vector** is a sequence of elements with the **same type**. Vector indexes start at 1 (not 0).

```r
# create a vector with c() function
c(1, 3, 5, 7)
```

```
## [1] 1 3 5 7
```

```r
# concatenate vectors
c(c(1, 3), c(5, 7))
```

```
## [1] 1 3 5 7
```

```r
# element-wise sum
c(1, 2, 3, 4) + c(10, 20, 30, 40)
```

```
## [1] 11 22 33 44
```

```r
# recyclying
10 + c(1, 2, 3, 4)
```

```
## [1] 11 12 13 14
```

```r
# element-wise product
c(1, 2, 3, 4) * c(10, 20, 30, 40)
```

```
## [1]  10  40  90 160
```

```r
# recyclying
10 * c(1, 2, 3, 4)
```

```
## [1] 10 20 30 40
```

```r
# scalar product (the result is a 1x1 matrix)
c(1, 2, 3, 4) %*% c(10, 20, 30, 40)
```

```
##      [,1]
## [1,]  300
```

```r
x = c(TRUE, FALSE, TRUE, FALSE)
(y = !x) # also prints result
```

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

```r
x & y
```

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

```r
x | y
```

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

```r
xor(x, y)
```

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

Vector Indexing

You may refer to members of a vector in several ways:

```r
primes = c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
primes[5]
```

```
## [1] 11
```

```r
primes[c(1, 5, 10)]
```

```
## [1]  2 11 29
```

```r
# all except element in pos. 1
primes[-1]
```

```
## [1]  3  5  7 11 13 17 19 23 29
```

```r
# all except elements in pos. 1,5,10
primes[-c(1, 5, 10)]
```

```
## [1]  3  5  7 13 17 19 23
```

```r
# conditional selection
primes > 15
```

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

```r
# conditional check on values
primes[primes > 15]
```

```
## [1] 17 19 23 29
```

```r
# modify the vector with condition on values to update
primes[primes > 15] = Inf
primes
```

```
##  [1]   2   3   5   7  11  13 Inf Inf Inf Inf
```

Coercion

All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be automatically casted bye R to the most flexible type (*coercion*). Types from least to most flexible are: logical, integer, double, and character.

```
x = c(TRUE, TRUE, FALSE, FALSE)
# how many TRUE?
sum(x)
```

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

```
# how many TRUE on average
mean(x)
```

```
## [1] 0.5
```

Named vectors

Vector elements can have names:

```
x = c(a = 1, b = 2, c = 3)
# or
x = c(1, 2, 3)
names(x) = c("a", "b", "c")
x["a"]
```

```
## a
## 1
```

```
x[c("a", "b")]
```

```
## a b
## 1 2
```

Factors

A **factor** is a vector that can contain only predefined values, and is used to store **categorical variables** (for instance sex or religion).

Factors are built on top of integer vectors using the `levels` attribute, which defines the set of allowed values.

```
x = factor(c("male", "female", "female", "male", "male"))
x
```

```
## [1] male   female female male   male
## Levels: female male
```

```
typeof(x)
```

```
## [1] "integer"
```

```
levels(x)
```

```
## [1] "female" "male"
```

```
# if you use values that are not levels
# a warning is issued and a NA is generated
x[1] = "unknown"
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "unknown"): invalid factor level, NA
## generated
```

```
x
```

```
## [1] <NA>   female female male   male
## Levels: female male
```

Lists

A **list** is a sequence of elements that might have **different types**.

```
# create a list
l = list(thing = "hat", size = 8.25, female = TRUE)

# print the list
l
```

```
## $thing
## [1] "hat"
##
## $size
## [1] 8.25
##
## $female
## [1] TRUE
```

```
str(l)
```

```
## List of 3
##  $ thing : chr "hat"
##  $ size  : num 8.25
##  $ female: logi TRUE
```

```
# an element
l$thing
```

```
## [1] "hat"
```

```
l[[1]]
```

```
## [1] "hat"
```

```
# a sublist
l[c("thing", "size")]
```

```
## $thing
## [1] "hat"
##
## $size
## [1] 8.25
```

```
l[c(1, 2)]
```

```
## $thing
## [1] "hat"
##
## $size
## [1] 8.25
```

Abstract view of lists indexing: "If list x is a train carrying objects, then x[[5]] is the object in car 5; x[5] is car number 5."

```
# a sublist containing the first element of the list
l[1]
```

```
## $thing
## [1] "hat"
```

```
typeof(l[1])
```

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

```
# the first element of the list
l[[1]]
```

```
## [1] "hat"
```

```
typeof(l[[1]])
```

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

List elements can have any atomic or complex type. Hence a list can contain other lists, making it a **nested list**.

```
l = list(1, list(1, 2, 3), list("a", 1, list("TRUE", "FALSE")))
str(l)
```

```
## List of 3
##  $ : num 1
##  $ :List of 3
##   ..$ : num 1
##   ..$ : num 2
##   ..$ : num 3
##  $ :List of 3
##   ..$ : chr "a"
##   ..$ : num 1
##   ..$ :List of 2
##   .. ..$ : chr "TRUE"
##   .. ..$ : chr "FALSE"
```

Example: Consider the list:

```
l = list(1, list(1, 2, 3), list("a", 1, list("TRUE", "FALSE")))
```

Find:

- the list `list(1, 2, 3)`
- the element `1` of list `list(1, 2, 3)`
- the element `TRUE` of list `list("TRUE", "FALSE")`

Solutions:

```
l[[2]]
l[[2]][[1]]
l[[3]][[3]][[1]]
```

Matrices

A **matrix** is a 2-dimensional vector, that is a vector of vectors of the same type and length.

```
# create by row
M = matrix(data = 1:9, nrow = 3, byrow = TRUE)
M
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
# create by column (the default)
N = matrix(data = 1:9, ncol = 3)
N
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
nrow(M)
```

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

```
ncol(M)
```

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

```
dim(M)
```

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

Matrices Indexing

```
M
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
# element in row 1 and column 2
M[1, 2]
```

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

```
# first row
M[1, ]
```

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

```
# first column
M[ ,1]
```

```
## [1] 1 4 7
```

```
# sub-matrix
M[1:2, 1:2]
```

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

```
## [2,]    4    5
```

```
M[-3, -3]
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    4    5
```

```
# diagonal
diag(M)
```

```
## [1] 1 5 9
```

Add rows and columns

```
P = matrix(data = runif(9), nrow = 3, byrow = TRUE)
P
```

```
##            [,1]      [,2]      [,3]
## [1,] 0.02406879 0.5666112 0.2428733
## [2,] 0.96836627 0.9504228 0.4948267
## [3,] 0.39288986 0.1989929 0.5098307
```

```
# add column
cbind(P, c(0, 0, 0))
```

```
##            [,1]      [,2]      [,3] [,4]
## [1,] 0.02406879 0.5666112 0.2428733    0
## [2,] 0.96836627 0.9504228 0.4948267    0
## [3,] 0.39288986 0.1989929 0.5098307    0
```

```
# modify matrix -> needs assignement
P = cbind(P, c(0, 0, 0))
P
```

```
##            [,1]      [,2]      [,3] [,4]
## [1,] 0.02406879 0.5666112 0.2428733    0
## [2,] 0.96836627 0.9504228 0.4948267    0
## [3,] 0.39288986 0.1989929 0.5098307    0
```

```
# add row
P = rbind(P, c(0, 0, 0, 0))
```

Operations on matrices

```
# element-wise sum
M + N
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    6   10   14
## [3,]   10   14   18
```

```
# element-wise product
M * N
```

```
##      [,1] [,2] [,3]
## [1,]    1    8   21
## [2,]    8   25   48
```

```
## [3,]   21   48   81
```

```
# matrix product
M %*% N
```

```
##      [,1] [,2] [,3]
## [1,]   14   32   50
## [2,]   32   77  122
## [3,]   50  122  194
```

```
# matrix transpose
t(M)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
# matrix inverse
C = matrix(c(1,0,1, 1,1,1, 1,1,0), nrow=3, byrow=TRUE)
D = solve(C)
D
```

```
##      [,1] [,2] [,3]
## [1,]    1   -1    1
## [2,]   -1    1    0
## [3,]    0    1   -1
```

```
D %*% C
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
C %*% D
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
# linear systems C x = b
C
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    1
## [2,]    1    1    1
## [3,]    1    1    0
```

```
b = c(2, 1, 3)
# the system is:
# x1      + x3 = 2
# x1 + x2 + x3 = 1
# x1 + x2      = 3
x = solve(C,b)
x
```

```
## [1]  4 -1 -2
```

```
C %*% x
```

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

```
# matrix spectrum
spectrum = eigen(C)

# columns are the eigenvectors
spectrum$vectors
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.4151581 -0.4743098 -0.6026918
## [2,] -0.7480890 -0.2110877  0.7515444
## [3,] -0.5176936  0.8546767  0.2682231
```

```
# eigenvalues
spectrum$values
```

```
## [1]  2.2469796 -0.8019377  0.5549581
```

```
# check
x = spectrum$vectors[, 1]
lambda = spectrum$values[1]
lambda * x
```

```
## [1] -0.9328517 -1.6809406 -1.1632470
```

```
C %*% x
```

```
##            [,1]
## [1,] -0.9328517
## [2,] -1.6809406
## [3,] -1.1632470
```

Data frames

A **data frame** is a **list** of vectors (called columns). A data frame is like a **database table**:

- each column has a name and contain elements of the same type
- different columns have the same length and may have different types

```
name = c("John", "Samuel", "Uma", "Bruce", "Tim")
age = c(23, 31, 17, 41, 25)
married = c(TRUE, FALSE, FALSE, TRUE, TRUE)

# string columns are loaded as character vectors
# (not as factors, the default!)
pulp = data.frame(name, age, married, stringsAsFactors = FALSE)
pulp
```

```
##     name age married
## 1   John  23    TRUE
## 2 Samuel  31   FALSE
## 3    Uma  17   FALSE
## 4  Bruce  41    TRUE
## 5    Tim  25    TRUE
```

Data Frames Indexing

```
# first row
pulp[1, ]
```

```
##   name age married
## 1 John  23    TRUE
```

```
# first column
# matrix style
pulp[ ,1]
```

```
## [1] "John"   "Samuel" "Uma"    "Bruce"  "Tim"
```

```
pulp[, "name"]
```

```
## [1] "John"   "Samuel" "Uma"    "Bruce"  "Tim"
```

```
# list style (remember a data frame is a list)
pulp$name
```

```
## [1] "John"   "Samuel" "Uma"    "Bruce"  "Tim"
```

```
pulp[[1]]
```

```
## [1] "John"   "Samuel" "Uma"    "Bruce"  "Tim"
```

```
# filtering
pulp[pulp$name == "Uma", ]
```

```
##   name age married
## 3  Uma  17   FALSE
```

```
pulp[pulp$age < 18, ]
```

```
##   name age married
## 3  Uma  17   FALSE
```

```
pulp[married == TRUE, "name"]
```

```
## [1] "John"  "Bruce" "Tim"
```

Example: Extract from the `pulp` data frame the names of adult people that are not married.

```
pulp[married == FALSE & age >= 18, "name"]
```

```
## [1] "Samuel"
```

Nested data frames

Since a data frame is a list, and lists can contain other lists as elements, you can create **nested data frames**, that is data frames whose elements are data frames.

```
# a data frame
Venus = data.frame(
  x = c(17, 19),
  y = c("Hello", "Venus"),
  stringsAsFactors = FALSE
)

# a data frame
Jupiter = data.frame(
```

```
  x = c(21, 23),
  y = c("Hello", "Jupiter"),
  stringsAsFactors = FALSE
)

# a nested data frame
# I() treats the object 'as is'
worlds = data.frame(
  x = I(list(Venus, Jupiter)),
  y = c("Hello", "Worlds"),
  stringsAsFactors = FALSE
)
```

Programming in R

Conditional and repetition

R is an Turing-complete (functional) programming language.

It includes **conditional statements**:

```
x = 49
if (x %% 7 == 0) x else -x
```

```
## [1] 49
```

And **loops**:

```
x = 108
i = 2
while (i <= x/2) {
 if (x %% i == 0) print(i)
 i = i + 1;
}
```

```
## [1] 2
## [1] 3
## [1] 4
## [1] 6
## [1] 9
## [1] 12
## [1] 18
## [1] 27
## [1] 36
## [1] 54
```

```
for (i in 2:(x/2)) {
  if (x %% i == 0) print(i)
}
```

```
## [1] 2
## [1] 3
## [1] 4
## [1] 6
## [1] 9
## [1] 12
## [1] 18
## [1] 27
```

```
## [1] 36
## [1] 54
```

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

# we know the sequence and output lengths
# create a vector of a given size
output <- vector("double", ncol(df))
for (i in 1:ncol(df)) {
  output[i] <- mean(df[[i]])
}
```

```
# we know the sequence length but we do NOT know the output length (slow solution)
means <- c(0, 1, 2)
# a vector of doubles of length 0
output <- double()
for (i in 1:length(means)) {
  n <- sample(1:100, 1)
  # dynamically increase the vector (slow)
  output <- c(output, rnorm(n, means[i]))
}

# create a list with length(means) elements (faster solution)
output <- vector("list", length(means))
for (i in 1:length(means)) {
  n <- sample(1:100, 1)
  output[[i]] <- rnorm(n, means[i])
}
# unlist the list into a vector
output <- unlist(output)
```

```
# we do not know the sequence length
# iterate until a sequence of Heads of length difficulty is found
flips <- 0
nheads <- 0
difficulty <- 10

while (nheads < difficulty) {
  if (sample(c("T", "H"), 1) == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}
flips
```

```
## [1] 1741
```

Avoid loops (if possible)

Most of the times you can perform your task by applying functions, avoiding loops. This is typically faster. R is a functional programming language, imperative loops are allowed but not reccomended for performance reasons.

```
x = 1:100

# compute the sum (bad)
s = 0
for (i in 1:length(x)) {
  s = s + x[i]
}
s
```

```
## [1] 5050
```

```
# compute the sum (good)
sum(x)
```

```
## [1] 5050
```

```
# even faster
n = length(x)
n * (n+1) / 2
```

```
## [1] 5050
```

Functions

You may use **built-in functions**:

```
log
```

```
## function (x, base = exp(1))  .Primitive("log")
```

```
args(log)
```

```
## function (x, base = exp(1))
## NULL
```

```
log(x = 128, base = 2)
```

```
## [1] 7
```

```
log(128, 2)
```

```
## [1] 7
```

```
log(2, 128)
```

```
## [1] 0.1428571
```

```
log(128)
```

```
## [1] 4.85203
```

Or define your our functions:

```
euclidean = function(x=0, y=0) {sqrt(x^2 + y^2)}

euclidean(1, 1)
```

```
## [1] 1.414214
```

```
euclidean(1)
```

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

Or define your own binary operators using functions:

```
# xor
'%()%' = function(x, y) {(x | y) & !(x & y)}

TRUE %()% TRUE
```

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

```
TRUE %()% FALSE
```

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

```
FALSE %()% TRUE
```

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

```
FALSE %()% FALSE
```

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

Functionals

Functions may be **recursive**:

```
factorial = function(x) {
 if (x == 0) 1 else x * factorial(x-1)
}
factorial(5)
```

```
## [1] 120
```

You may write **functionals**, that are functions whose arguments are other functions:

```
# compute the sum of applications of f up to n
g = function(f, n) {
 sum = 0
 for (i in 1:n) sum = sum + f(i)
 return(sum)
}

g(factorial, 5)
```

```
## [1] 153
```

Apply-like functionals

An application of functionals and iteration is the set of **apply-like functionals**:

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

```r
# apply mean to each column of data frame, returns a list
lapply(df, mean)
```

```
## $a
## [1] -0.5033904
##
## $b
## [1] 0.01174957
##
## $c
## [1] 0.1874371
##
## $d
## [1] -0.05214255
```

```r
# apply mean to each column of data frame, returns an atomic vector
sapply(df, mean)
```

```
##           a           b           c           d
## -0.50339045  0.01174957  0.18743707 -0.05214255
```

```r
mtx <- cbind(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)


# apply mean to each column of matrix, returns an atomic vector
apply(mtx, 2, mean)
```

```
##           a           b           c           d
##  0.09151795  0.24743168 -0.39112420 -0.20356782
```

```r
# apply mean to each row of matrix, returns an atomic vector
apply(mtx, 1, mean)
```

```
##  [1]  0.46442348 -0.48415848  0.63655598 -0.04500374  0.11537436  0.07985593
##  [7]  0.02307620 -1.19135747 -0.18396188 -0.05416034
```

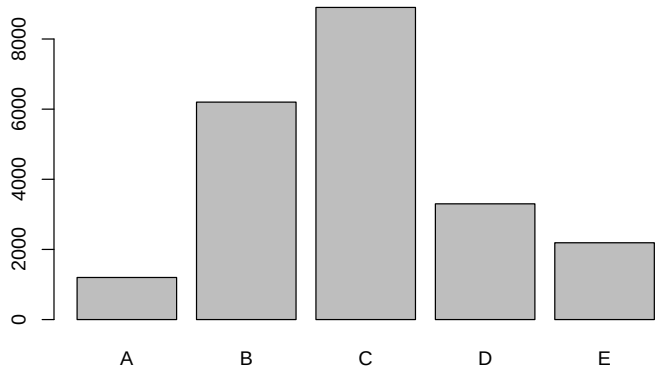
Plotting in R

Barplot

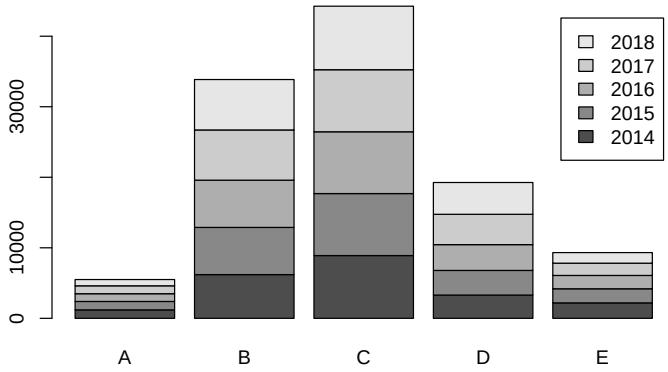
```
# a data matrix
M = matrix(c(
   c(1200, 1190, 1100, 1120, 890),
   c(6200, 6690, 6700, 7120, 7150),
   c(8900, 8790, 8760, 8800, 9010),
   c(3300, 3490, 3660, 4300, 4510),
   c(2190, 2000, 1890, 1740, 1500)), ncol = 5
)

# give names to rows
rownames(M) = 2014:2018
# give names to columns
colnames(M) = LETTERS[1:5]
M
```

```
##         A    B    C    D    E
## 2014 1200 6200 8900 3300 2190
## 2015 1190 6690 8790 3490 2000
## 2016 1100 6700 8760 3660 1890
## 2017 1120 7120 8800 4300 1740
## 2018  890 7150 9010 4510 1500
```

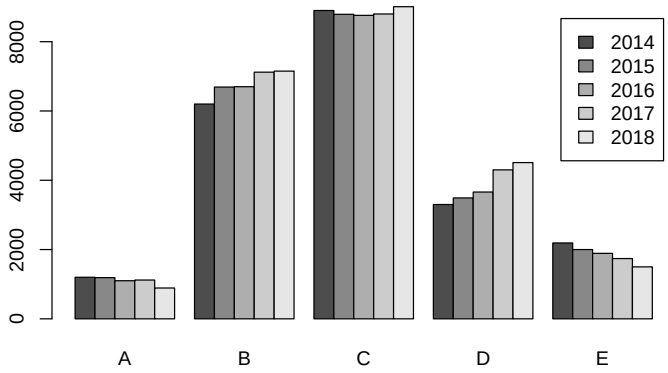
```
# barplot
barplot(M[1,])
```




```
# stacked barplot
barplot(M, legend=TRUE)
```

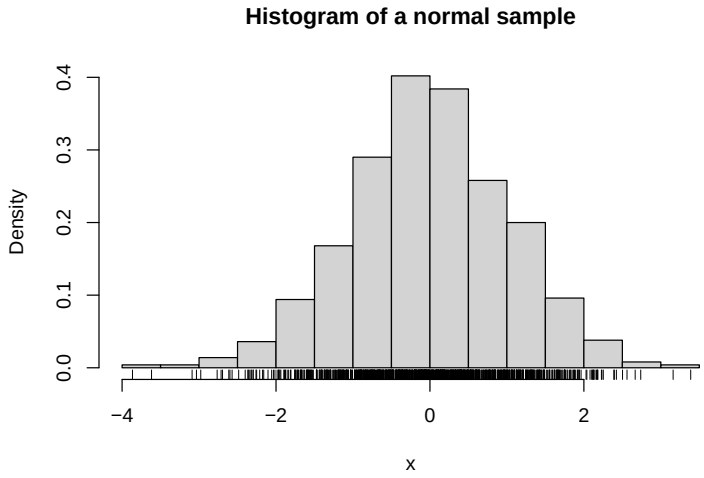



```
#  juxtaposed barplot
barplot(M, beside=TRUE, legend=TRUE)
```

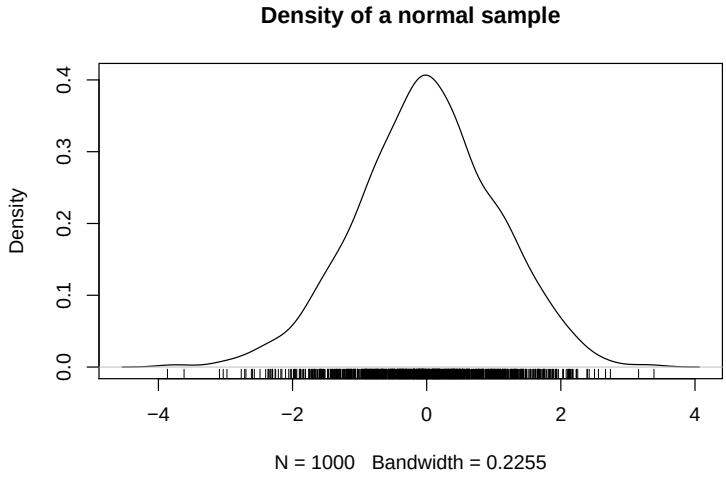


Histogram

```
# histogram
x = rnorm(1000) # random values with norm (gaussian) distribution
hist(x, probability=TRUE, main="Histogram of a normal sample")
## add distribution
rug(x)
```

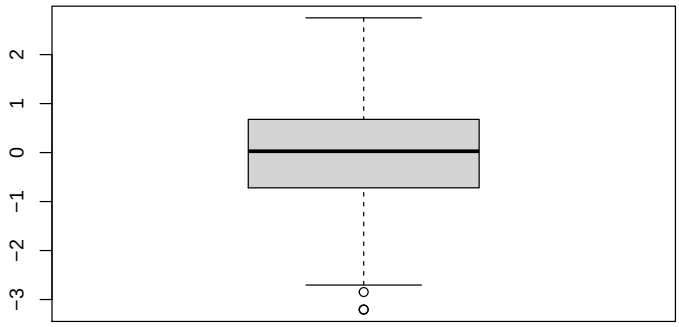



```
# density plot
plot(density(x), main="Density of a normal sample")
rug(x)
```

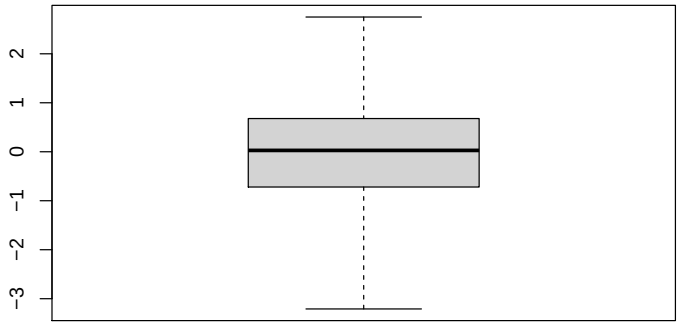


Boxplot

```
# boxplot
# If range is positive, the whiskers extend to the most extreme
# data point  which is no more than range times the interquartile
# range from the box.  A value of zero causes the whiskers to extend
# to the data extremes.
x = rnorm(1000)
boxplot(x, range = 1.5)
```

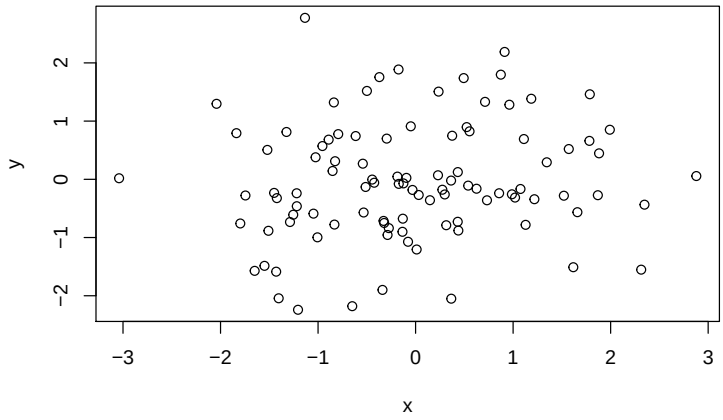



```
boxplot(x, range = 0)
```

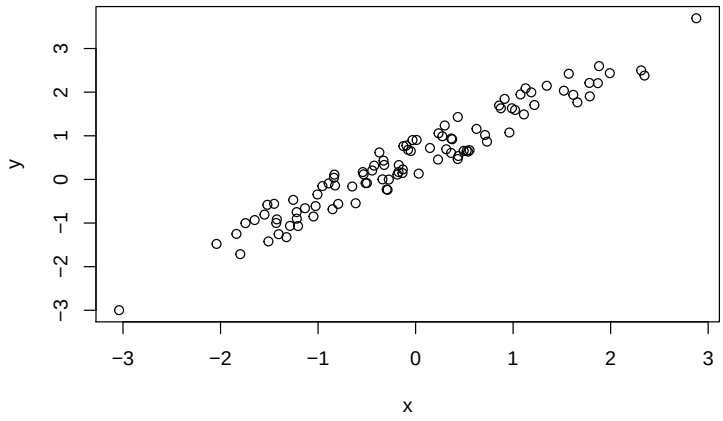


Scatter plot

```
# scatter plot
x = rnorm(100)
y = rnorm(100)
plot(x, y)
```




```r
#runif -> random values from 0 to 1
y = x + runif(100)
plot(x, y)
```



R Markdown

- **R Markdown** provides an unified authoring framework for data science, combining your **code**, its **results**, and your **prose** commentary
- R Markdown documents are fully **reproducible** and support many output formats, like HTML, PDF, and slideshows

R Markdown files are designed to be used in three ways:

1. for **communicating** to decision makers, who want to focus on the conclusions, not the code behind the analysis
2. for **collaborating** with other data scientists (including future you!), who are interested in both your conclusions, and how you reached them (the code)
3. as an **environment** in which to do data science, as a modern day **lab notebook** where you can capture not only what you did, but also what you were thinking

3. Read data - readr

Tibbles

Tibbles are data frames, but they tweak some older behaviours to make life a little easier. They are defined in the `tibble` package.

Most R packages use regular data frames, so you might want to coerce a data frame to a tibble:

```
library(tibble)
as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

You can create a new tibble from individual vectors with `tibble()`; it will automatically recycle inputs of length 1, and allows you to refer to variables that you just created:

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
```

```
## # A tibble: 5 x 3
##       x     y     z
##   <int> <dbl> <dbl>
## 1     1     1     2
## 2     2     1     5
## 3     3     1    10
## 4     4     1    17
## 5     5     1    26
```

If you're already familiar with data frames, note that tibbles do much less:

1. it never changes the names of variables
2. they never do partial matching when subsetting

```
# it never changes the names of variables
tibble(a = 1:10, a = 11:20)
data.frame(a = 1:10, a = 11:20)

names(tibble(`crazy name` = 1))
names(data.frame(`crazy name` = 1))

# they never do partial matching when subsetting
tb = tibble(abc = 1:10)
```

```
df = data.frame(abc = 1:10)
tb$a
df$a
```

Read and write CSV

Here you'll learn how to read plain-text rectangular files (Comma Separated Value, or CSV file) into R using the `readr` package.

```
library(readr)
```

Read a CSV file with `read_csv()`; it prints out a column specification that gives the name and type of each column:

```
heights = read_csv("http://users.dimi.uniud.it/~massimo.franceschet/ns/plugandplay/import/heights.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   earn = col_double(),
##   height = col_double(),
##   sex = col_character(),
##   ed = col_double(),
##   age = col_double(),
##   race = col_character()
## )
```

```
heights
```

```
## # A tibble: 1,192 x 6
##     earn height sex       ed   age race
##    <dbl>  <dbl> <chr>  <dbl> <dbl> <chr>
##  1 50000   74.4 male      16    45 white
##  2 60000   65.5 female    16    58 white
##  3 30000   63.6 female    16    29 white
##  4 50000   63.1 female    16    91 other
##  5 51000   63.4 female    17    39 white
##  6  9000   64.4 female    15    26 white
##  7 29000   61.7 female    12    49 white
##  8 32000   72.7 male      17    46 white
##  9  2000   72.0 male      15    21 hispanic
## 10 27000   72.2 male      12    26 white
## # ... with 1,182 more rows
```

You can also view the data frame with the `View()` function.

`read_csv()` uses the first line of the data (the header) for the column names, which is a very common convention. There are cases where you might want to tweak this behaviour:

```
# skip lines
read_csv("The first line of metadata
          The second line of metadata
          x,y,z
          1,2,3",
         skip = 2)
```

```
## # A tibble: 1 x 3
##       x     y     z
```

```
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

```
# skip comments
read_csv("# A comment I want to skip
          x,y,z
          1,2,3",
         comment = "#")
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

```
# no header -> header automatically created -> X1,X2,...,Xn
read_csv("1,2,3\n4,5,6", col_names = FALSE)
```

```
## # A tibble: 2 x 3
##      X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

```
# no header + custom header
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
```

```
## # A tibble: 2 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

You can specify which data to interpret as NA values as follows:

```
# NA as NA
read_csv("a,b,c\n1,NA,.")
```

```
## # A tibble: 1 x 3
##       a b     c
##   <dbl> <lgl> <chr>
## 1     1 NA    .
```

```
# . as NA
read_csv("a,b,c\n1,NA,.", na = ".")
```

```
## # A tibble: 1 x 3
##       a b     c
##   <dbl> <chr> <lgl>
## 1     1 NA    NA
```

Package `readr` also comes with a useful function for writing data back to disk: `write_csv()`.

```
write_csv(heights, "heights.csv")
```

However the type information of columns is lost when you save to CSV. This makes CSVs a little unreliable for caching interim results. One alternative is using **RDS** format (R's custom binary format):

```
write_rds(heights, "heights.rds")
read_rds("heights.rds")
```

```
## # A tibble: 1,192 x 6
```

```
##     earn height sex       ed   age race
##    <dbl>  <dbl> <chr>  <dbl> <dbl> <chr>
##  1 50000   74.4 male      16    45 white
##  2 60000   65.5 female    16    58 white
##  3 30000   63.6 female    16    29 white
##  4 50000   63.1 female    16    91 other
##  5 51000   63.4 female    17    39 white
##  6  9000   64.4 female    15    26 white
##  7 29000   61.7 female    12    49 white
##  8 32000   72.7 male      17    46 white
##  9  2000   72.0 male      15    21 hispanic
## 10 27000   72.2 male      12    26 white
## # ... with 1,182 more rows
```

4. Give a good shape to your data - tidyr

Tidy data

The data scientist manages variables, observations and values:

1. A **variable** is a quantity or quality that you can measure
2. A **value** is the state of a variable when you measure it
3. An **observation** is a set of measurements of variables made under similar conditions

There are three interrelated rules which make a dataset **tidy**:

1. Put each variable in a column
2. Put each observation in a row
3. Put each value in a cell

Why ensure that your data is tidy? There are two main advantages:

1. if you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying **uniformity**
2. there's a specific advantage to placing variables in columns because most built-in R functions work with **vectors of values**
3. however, there are lots of useful and well-founded data structures that are not tidy data. An important example is the **adjacency matrix** to represent a network.

Tidy and messy data

Each dataset below shows the same values of four variables *country*, *year*, *population*, and *cases*, but each dataset organises the values in a different way.

```
library(tidyr)

table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
```

```
## 12 China        2000 population 1280428583
```

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```
# Spread across two tibbles
table4a  # cases
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
table4b  # population
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

table1 is the best. table2 and table3 does not respect tidy rules.

table4 is even worse, split observations into two tibbles.

Let's see how to fix those tables:

Gathering

Apply gathering when variables are in fact values and a single row gathers many observations

This is the case of `table4a` and `table4b` above, that we can gather as follows:

```
gather(table4a, `1999`, `2000`, key = "year", value = "cases")
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
```

```
## 2 Brazil      1999   37737
## 3 China       1999  212258
## 4 Afghanistan 2000    2666
## 5 Brazil      2000   80488
## 6 China       2000  213766
```

```
# Development on gather() is complete, and for new code we recommend switching to pivot_longer()
pivot_longer(table4a, c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

```
gather(table4b, `1999`, `2000`, key = "year", value = "population")
```

```
## # A tibble: 6 x 3
##   country     year  population
##   <chr>       <chr>      <int>
## 1 Afghanistan 1999    19987071
## 2 Brazil      1999   172006362
## 3 China       1999  1272915272
## 4 Afghanistan 2000    20595360
## 5 Brazil      2000   174504898
## 6 China       2000  1280428583
```

```
# Development on gather() is complete, and for new code we recommend switching to pivot_longer()
pivot_longer(table4b, c(`1999`, `2000`), names_to = "year", values_to = "population")
```

```
## # A tibble: 6 x 3
##   country     year  population
##   <chr>       <chr>      <int>
## 1 Afghanistan 1999    19987071
## 2 Afghanistan 2000    20595360
## 3 Brazil      1999   172006362
## 4 Brazil      2000   174504898
## 5 China       1999  1272915272
## 6 China       2000  1280428583
```

Finally, we can join the gathered tables to obtain the original tidy data frame stored in `table1` above:

```
library(dplyr)

left_join(gather(table4a, `1999`, `2000`, key = "year", value = "cases"),
          gather(table4b, `1999`, `2000`, key = "year", value = "population"))
```

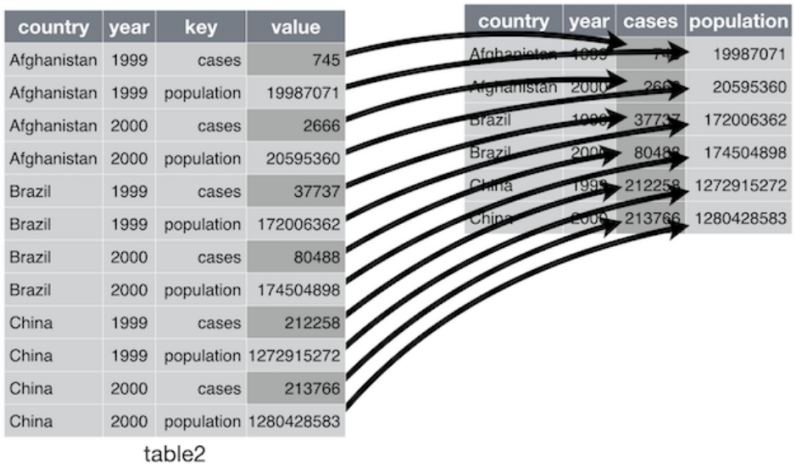
```
## # A tibble: 6 x 4
##   country     year   cases population
##   <chr>       <chr>  <int>      <int>
## 1 Afghanistan 1999     745   19987071
## 2 Brazil      1999   37737  172006362
## 3 China       1999  212258 1272915272
```

```
## 4 Afghanistan 2000    2666   20595360
## 5 Brazil      2000   80488  174504898
## 6 China       2000  213766 1280428583
```

Gathering makes wide tables narrower and longer.

Spreading

Apply spreading when values are in fact variables and a single observation is spread across many rows.



country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

table2

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

This is the case of `table2` above, that we can spread as follows:

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```
spread(table2, key = type, value = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

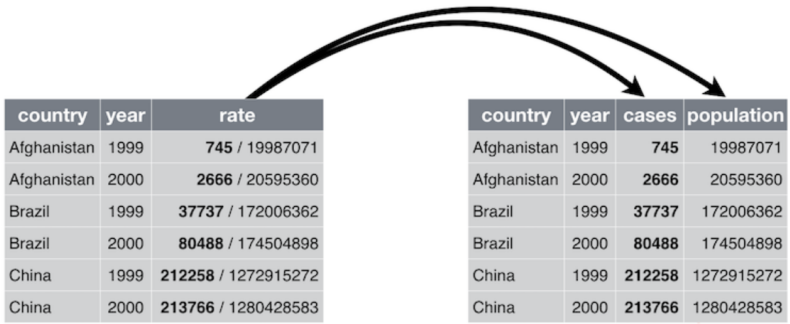
```
# Development on spread() is complete, and for new code we recommend switching to pivot_wider()
pivot_wider(table2, names_from = type, values_from = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Spreading makes long tables shorter and wider.

Separating

Apply separating when values are composite (not atomic): a column gathers many variables.



country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

table3

This is the case of `table3` above, that we can separate as follows:

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```
separate(table3, rate, into = c("cases", "population"),
         sep = "/", convert = TRUE)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

You can also pass a vector of integers to the parameter `sep`. Command `separate()` will interpret the integers as positions to split at:

```
table5 = separate(table3, year, into = c("century", "year"), sep = 2)
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
##   <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

Uniting

Apply uniting when values are partial: a variable is spread across many columns.

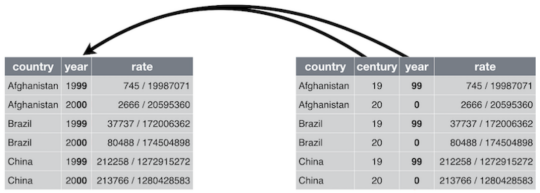


table6

This is the case of `table5` below, that we can unite as follows:

```
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
##   <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

```
unite(table5, new, century, year, sep = "")
```

```
## # A tibble: 6 x 3
##   country     new   rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 1999  745/19987071
## 2 Afghanistan 2000  2666/20595360
## 3 Brazil      1999  37737/172006362
## 4 Brazil      2000  80488/174504898
## 5 China       1999  212258/1272915272
## 6 China       2000  213766/1280428583
```

5. A grammar for data manipulation - dplyr

Overview - Unary verbs

dplyr is a **grammar for data manipulation**, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

- `select()` picks variables based on their names (it's a filter on columns of the data frame)
- `filter()` picks cases based on their values (it's a filter on rows of the data frame)
- `mutate()` adds new variables that are typically functions of existing variables
- `arrange()` changes the ordering of the rows
- `group_by()` partitions rows of data into groups defined by the values of some variables
- `summarise()` reduces multiple values down to a single summary, used in combination with `group_by()`

Binary verbs

In practice, you'll normally have many tables that contribute to an analysis, and you need flexible tools to combine them.

In dplyr, there are three families of verbs that work with two tables at a time:

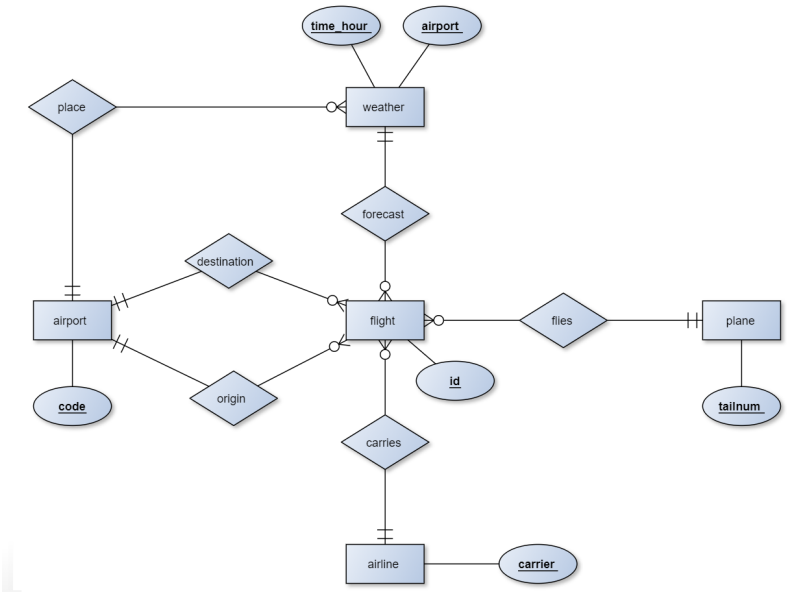
- **mutating joins**, which add new variables to one table from matching rows in another
- **filtering joins**, which filter observations from one table based on whether or not they match an observation in the other table
- **set operations**, which combine the observations in the data sets as if they were set elements

Example: dataset: New York flights

The nycflights13 dataset contains information about all flights that departed from New York City in 2013 and related metadata (planes, airports, airlines and weather conditions).

The graph notation below is called Entity-Relationship (ER) model; it is a conceptual model of data: read more here.

```
library("nycflights13")
```



Example: dataset: Star Wars

The Star Wars API, or SWAPI, is the world's first quantified and programmatically-accessible data source for all the data from the Star Wars canon universe!

A fraction of the data source is contained in the dataset starwars in the dplyr package:

```
library("dplyr")
```

Pipes Mechanism

Pipes are a powerful tool for clearly expressing a sequence of multiple operations.

> Little bunny Foo Foo
> Went hopping through the forest
> Scooping up the field mice
> And bopping them on the head

Intermediate steps:

```
foo_foo_1 <- hop(foo_foo, through = forest)
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
foo_foo_3 <- bop(foo_foo_2, on = head)
```

Too many unimportant names!

We can overwrite the original...

```
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)
```

...but debugging will be painful! Another way: function composition:

```
bop(
  scoop(
    hop(foo_foo, through = forest),
    up = field_mice
  ),
  on = head
)
```

Hard for a human to consume!

Solution: pipes.

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

This is our favorite form, because it focuses on verbs, not nouns!

Unary verbs

Now we will se all the unary verbs in detail.

filter

Verb `filter()` picks cases based on their values (it's a filter on rows of the data frame).

```
starwars %>%
  filter(species == "Droid")
```

```
## # A tibble: 6 x 14
##   name  height  mass hair_color skin_color eye_color birth_year sex   gender
```

```
##   <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 C-3PO    167    75 <NA>       gold       yellow           112 none  mascu~
## 2 R2-D2     96    32 <NA>       white, bl~ red               33 none  mascu~
## 3 R5-D4     97    32 <NA>       white, red red               NA none  mascu~
## 4 IG-88    200   140 none       metal      red               15 none  mascu~
## 5 R4-P~     96    NA none       silver, r~ red, blue         NA none  femin~
## 6 BB8       NA    NA none       none       black             NA none  mascu~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
starwars %>%
  filter(species == "Droid" & eye_color == "red")
```

```
## # A tibble: 3 x 14
##   name  height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 R2-D2     96    32 <NA>       white, bl~ red               33 none  mascu~
## 2 R5-D4     97    32 <NA>       white, red red               NA none  mascu~
## 3 IG-88    200   140 none       metal      red               15 none  mascu~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
# or
starwars %>%
  filter(species == "Droid", eye_color == "red")
```

```
## # A tibble: 3 x 14
##   name  height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 R2-D2     96    32 <NA>       white, bl~ red               33 none  mascu~
## 2 R5-D4     97    32 <NA>       white, red red               NA none  mascu~
## 3 IG-88    200   140 none       metal      red               15 none  mascu~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
starwars %>%
  filter(species == "Droid" | species == "Wookiee")
```

```
## # A tibble: 8 x 14
##   name  height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 C-3PO    167    75 <NA>       gold       yellow           112 none  mascu~
## 2 R2-D2     96    32 <NA>       white, bl~ red               33 none  mascu~
## 3 R5-D4     97    32 <NA>       white, red red               NA none  mascu~
## 4 Chew~    228   112 brown      unknown    blue             200 male  mascu~
## 5 IG-88    200   140 none       metal      red               15 none  mascu~
## 6 R4-P~     96    NA none       silver, r~ red, blue         NA none  femin~
## 7 Tarf~    234   136 brown      brown      blue              NA male  mascu~
## 8 BB8       NA    NA none       none       black             NA none  mascu~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

```
starwars %>%
  filter(species == "Droid", !is.na(birth_year))
```

```
## # A tibble: 3 x 14
##   name  height  mass hair_color skin_color eye_color birth_year sex   gender
```

```
##   <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 C-3PO    167    75 <NA>       gold       yellow           112 none  mascu~
## 2 R2-D2     96    32 <NA>       white, bl~ red               33 none  mascu~
## 3 IG-88    200   140 none       metal      red               15 none  mascu~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

select

Verb `select()` picks variables based on their names (it's a filter on columns of the data frame).

```
starwars %>%
  select(name, gender, birth_year)
```

```
## # A tibble: 87 x 3
##    name               gender    birth_year
##    <chr>              <chr>          <dbl>
##  1 Luke Skywalker     masculine       19
##  2 C-3PO              masculine      112
##  3 R2-D2              masculine       33
##  4 Darth Vader        masculine       41.9
##  5 Leia Organa        feminine        19
##  6 Owen Lars          masculine       52
##  7 Beru Whitesun lars feminine        47
##  8 R5-D4              masculine       NA
##  9 Biggs Darklighter  masculine       24
## 10 Obi-Wan Kenobi     masculine       57
## # ... with 77 more rows
```

```
starwars %>%
  select(name, ends_with("color"))
```

```
## # A tibble: 87 x 4
##    name               hair_color    skin_color  eye_color
##    <chr>              <chr>         <chr>       <chr>
##  1 Luke Skywalker     blond         fair        blue
##  2 C-3PO              <NA>          gold        yellow
##  3 R2-D2              <NA>          white, blue red
##  4 Darth Vader        none          white       yellow
##  5 Leia Organa        brown         light       brown
##  6 Owen Lars          brown, grey   light       blue
##  7 Beru Whitesun lars brown         light       blue
##  8 R5-D4              <NA>          white, red  red
##  9 Biggs Darklighter  black         light       brown
## 10 Obi-Wan Kenobi     auburn, white fair        blue-gray
## # ... with 77 more rows
```

```
starwars %>%
  select(-contains("color"))
```

```
## # A tibble: 87 x 11
##    name  height  mass birth_year sex   gender homeworld species films vehicles
##    <chr>  <int> <dbl>      <dbl> <chr> <chr>  <chr>     <chr>   <lis> <list>
##  1 Luke~    172    77       19   male  mascu~ Tatooine  Human   <chr~ <chr [2~
##  2 C-3PO    167    75      112   none  mascu~ Tatooine  Droid   <chr~ <chr [0~
##  3 R2-D2     96    32       33   none  mascu~ Naboo     Droid   <chr~ <chr [0~
##  4 Dart~    202   136       41.9 male  mascu~ Tatooine  Human   <chr~ <chr [0~
```

```
##  5 Leia~    150    49       19   fema~ femin~ Alderaan  Human   <chr~ <chr [1~
##  6 Owen~    178   120       52   male  mascu~ Tatooine  Human   <chr~ <chr [0~
##  7 Beru~    165    75       47   fema~ femin~ Tatooine  Human   <chr~ <chr [0~
##  8 R5-D4     97    32       NA   none  mascu~ Tatooine  Droid   <chr~ <chr [0~
##  9 Bigg~    183    84       24   male  mascu~ Tatooine  Human   <chr~ <chr [0~
## 10 Obi-~    182    77       57   male  mascu~ Stewjon   Human   <chr~ <chr [1~
## # ... with 77 more rows, and 1 more variable: starships <list>
```

Examples:

The name of all characters with blue eyes and blond hair

```
starwars %>%
  filter(hair_color == "blond" & eye_color == "blue") %>%
  select(name)
```

```
## # A tibble: 3 x 1
##   name
##   <chr>
## 1 Luke Skywalker
## 2 Anakin Skywalker
## 3 Finis Valorum
```

The film starred by Luke Skywalker

```
starwars %>%
  filter(name == "Luke Skywalker") %>%
  select(films)
```

```
## # A tibble: 1 x 1
##   films
##   <list>
## 1 <chr [5]>
```

```
starwars %>%
  filter(name == "Luke Skywalker") %>%
  pull(films) %>%
  unlist()
```

```
## [1] "The Empire Strikes Back" "Revenge of the Sith"
## [3] "Return of the Jedi"      "A New Hope"
## [5] "The Force Awakens"
```

1. flights on Christmas
2. flights that have a valid (not null) delay either on departure or on arrival
3. flights that have a valid (not null) delay both on departure and on arrival

```
# flights on Christmas
filter(flights, month == 12 & day == 25)

# flights that have a valid (not null) delay
# either on departure or on arrival
filter(flights, !is.na(dep_delay) | !is.na(arr_delay))

# flights that have a valid (not null) delay
# both on departure and on arrival
filter(flights, !is.na(dep_time) & !is.na(arr_time))
```

mutate

Verb `mutate()` adds new variables that are typically functions of existing variables.

```
starwars %>%
  mutate(bmi = mass / ((height / 100)  ^ 2)) %>%
  select(name:mass, bmi)
```

```
## # A tibble: 87 x 4
##    name               height  mass   bmi
##    <chr>               <int> <dbl> <dbl>
##  1 Luke Skywalker        172    77  26.0
##  2 C-3PO                 167    75  26.9
##  3 R2-D2                  96    32  34.7
##  4 Darth Vader           202   136  33.3
##  5 Leia Organa           150    49  21.8
##  6 Owen Lars             178   120  37.9
##  7 Beru Whitesun lars    165    75  27.5
##  8 R5-D4                  97    32  34.0
##  9 Biggs Darklighter     183    84  25.1
## 10 Obi-Wan Kenobi        182    77  23.2
## # ... with 77 more rows
```

Missing key: adding surrogate key

If a table lacks a natural primary key (a set of attributes that identify observations), it's sometimes useful to add one that simply contains the row number. This is called a **surrogate key**.

This is the case of flights table. Add an unique attribute id to the flights table.

```
flights =
  flights %>%
  arrange(year, month, day, sched_dep_time) %>%
  mutate(id = 1:nrow(flights)) %>%
  select(id, everything())

flights
```

```
## # A tibble: 336,776 x 20
##       id  year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1     1  2013     1     1      517            515         2      830
##  2     2  2013     1     1      533            529         4      850
##  3     3  2013     1     1      542            540         2      923
##  4     4  2013     1     1      544            545        -1     1004
##  5     5  2013     1     1      554            558        -4      740
##  6     6  2013     1     1      559            559         0      702
##  7     7  2013     1     1      554            600        -6      812
##  8     8  2013     1     1      555            600        -5      913
##  9     9  2013     1     1      557            600        -3      709
## 10    10  2013     1     1      557            600        -3      838
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
## #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

Example:

Add a catchup variable that contains the catch up time of the flight and select id and delays of flights that catched up during the flight.

```
# flights that catched up during the flight sorted by catch up time
flights %>%
  mutate(catchup = dep_delay - arr_delay) %>%
  select(id, dep_delay, arr_delay, catchup) %>%
  filter(catchup > 0)
```

```
## # A tibble: 221,565 x 4
##       id dep_delay arr_delay catchup
##    <int>     <dbl>     <dbl>   <dbl>
##  1     4        -1       -18      17
##  2     6         0        -4       4
##  3     7        -6       -25      19
##  4     9        -3       -14      11
##  5    10        -3        -8       5
##  6    13        -2        -3       1
##  7    15        -2       -14      12
##  8    17        -1        -8       7
##  9    18         0        -7       7
## 10    20         1        -6       7
## # ... with 221,555 more rows
```

arrange

Verb `arrange()` changes the ordering of the rows.

```
starwars %>%
  arrange(mass) %>%
  select(name, mass, height)
```

```
## # A tibble: 87 x 3
##    name                  mass height
##    <chr>                <dbl>  <int>
##  1 Ratts Tyerell           15     79
##  2 Yoda                    17     66
##  3 Wicket Systri Warrick   20     88
##  4 R2-D2                   32     96
##  5 R5-D4                   32     97
##  6 Sebulba                 40    112
##  7 Dud Bolt                45     94
##  8 Padmé Amidala           45    165
##  9 Wat Tambor              48    193
## 10 Sly Moore               48    178
## # ... with 77 more rows
```

```
starwars %>%
  arrange(desc(mass)) %>%
  select(name, mass, height)
```

```
## # A tibble: 87 x 3
##    name                  mass height
##    <chr>                <dbl>  <int>
##  1 Jabba Desilijic Tiure  1358    175
```

```
##  2 Grievous                159    216
##  3 IG-88                   140    200
##  4 Darth Vader             136    202
##  5 Tarfful                 136    234
##  6 Owen Lars               120    178
##  7 Bossk                   113    190
##  8 Chewbacca               112    228
##  9 Jek Tono Porkins        110    180
## 10 Dexter Jettster         102    198
## # ... with 77 more rows
```

```
starwars %>%
  arrange(-mass, -height) %>%
  select(name, mass, height)
```

```
## # A tibble: 87 x 3
##    name                  mass height
##    <chr>                <dbl>  <int>
##  1 Jabba Desilijic Tiure  1358    175
##  2 Grievous                159    216
##  3 IG-88                   140    200
##  4 Tarfful                 136    234
##  5 Darth Vader             136    202
##  6 Owen Lars               120    178
##  7 Bossk                   113    190
##  8 Chewbacca               112    228
##  9 Jek Tono Porkins        110    180
## 10 Dexter Jettster         102    198
## # ... with 77 more rows
```

Examples:

The name and birth year of all human characters living in Tatooine sorted by birth year

```
starwars %>%
  filter(species == "Human" & homeworld == "Tatooine") %>%
  arrange(birth_year) %>%
  select(name, birth_year)
```

```
## # A tibble: 8 x 2
##   name               birth_year
##   <chr>                   <dbl>
## 1 Luke Skywalker           19
## 2 Biggs Darklighter        24
## 3 Darth Vader              41.9
## 4 Anakin Skywalker         41.9
## 5 Beru Whitesun lars       47
## 6 Owen Lars                52
## 7 Shmi Skywalker           72
## 8 Cliegg Lars              82
```

The characters sorted in decreasing order of popularity (the popularity is the number of films starred by a characted)

Hint: use `sapply` function which apply a function over a List or Vector.

```
starwars %>%
  mutate(popularity = sapply(films, length)) %>%
```

```
  arrange(desc(popularity)) %>%
  select(name, popularity)
```

```
## # A tibble: 87 x 2
##    name           popularity
##    <chr>               <int>
##  1 R2-D2                   7
##  2 C-3PO                   6
##  3 Obi-Wan Kenobi          6
##  4 Luke Skywalker          5
##  5 Leia Organa             5
##  6 Chewbacca               5
##  7 Yoda                    5
##  8 Palpatine               5
##  9 Darth Vader             4
## 10 Han Solo                4
## # ... with 77 more rows
```

group by and summarise

- verb `group_by()` partitions rows of data into groups defined by the values of some variables
- verb `summarise()` reduces multiple values down to a single summary
- they are typically used in combination.

```
starwars %>%
  group_by(species) %>%
  summarise(n = n())
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 38 x 2
##    species       n
##    <chr>     <int>
##  1 Aleena        1
##  2 Besalisk      1
##  3 Cerean        1
##  4 Chagrian      1
##  5 Clawdite      1
##  6 Droid         6
##  7 Dug           1
##  8 Ewok          1
##  9 Geonosian     1
## 10 Gungan        3
## # ... with 28 more rows
```

```
starwars %>%
  count(species, sort = TRUE)
```

```
## # A tibble: 38 x 2
##    species      n
##    <chr>    <int>
##  1 Human       35
##  2 Droid        6
##  3 <NA>         4
##  4 Gungan       3
```

```
##  5 Kaminoan     2
##  6 Mirialan     2
##  7 Twi'lek      2
##  8 Wookiee      2
##  9 Zabrak       2
## 10 Aleena       1
## # ... with 28 more rows
```

```
starwars %>%
  group_by(species) %>%
  summarise(
    n = n(),
    mass = mean(mass, na.rm = TRUE)
  ) %>%
  filter(n > 1, mass > 50) %>%
  arrange(desc(n))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 8 x 3
##   species      n  mass
##   <chr>    <int> <dbl>
## 1 Human       35  82.8
## 2 Droid        6  69.8
## 3 Gungan       3  74
## 4 Kaminoan     2  88
## 5 Mirialan     2  53.1
## 6 Twi'lek      2  55
## 7 Wookiee      2 124
## 8 Zabrak       2  80
```

Example:

The number of characters with a given eye and hair color sorted in decreasing order

```
count(starwars, hair_color, eye_color, sort=TRUE)
```

```
## # A tibble: 35 x 3
##    hair_color eye_color     n
##    <chr>      <chr>     <int>
##  1 black      brown         9
##  2 brown      brown         9
##  3 none       black         9
##  4 brown      blue          7
##  5 none       orange        7
##  6 none       yellow        6
##  7 blond      blue          3
##  8 none       blue          3
##  9 none       red           3
## 10 black      blue          2
## # ... with 25 more rows
```

1. the number of flights per day
2. the busy days (with more than 1000 flights)

```
# the number of flights per day
group_by(flights, month, day) %>%
  summarise(count = n())
```

```
## `summarise()` regrouping output by 'month' (override with `.groups` argument)
```

```
## # A tibble: 365 x 3
## # Groups:   month [12]
##    month   day count
##    <int> <int> <int>
##  1     1     1   842
##  2     1     2   943
##  3     1     3   914
##  4     1     4   915
##  5     1     5   720
##  6     1     6   832
##  7     1     7   933
##  8     1     8   899
##  9     1     9   902
## 10     1    10   932
## # ... with 355 more rows
```

```
# or
count(flights, month, day)
```

```
## # A tibble: 365 x 3
##    month   day     n
##    <int> <int> <int>
##  1     1     1   842
##  2     1     2   943
##  3     1     3   914
##  4     1     4   915
##  5     1     5   720
##  6     1     6   832
##  7     1     7   933
##  8     1     8   899
##  9     1     9   902
## 10     1    10   932
## # ... with 355 more rows
```

```
# the busy days (with more than 1000 flights)
count(flights, month, day) %>%
  filter(n > 1000)
```

```
## # A tibble: 14 x 3
##    month   day     n
##    <int> <int> <int>
##  1     7     8  1004
##  2     7     9  1001
##  3     7    10  1004
##  4     7    11  1006
##  5     7    12  1002
##  6     7    17  1001
##  7     7    18  1003
##  8     7    25  1003
##  9     7    31  1001
## 10     8     7  1001
## 11     8     8  1001
## 12     8    12  1001
## 13    11    27  1014
```

```
## 14    12     2  1004
```

The mean departure delay per day sorted in decreasing order of all flights on busy days of July

```
# the mean departure delay per day sorted in
# decreasing order of all flights on busy days of July
filter(flights, month == 7) %>%
  group_by(month, day) %>%
  summarise(n = n(), avg_delay = mean(dep_delay, na.rm = TRUE)) %>%
  filter(n > 1000) %>%
  arrange(desc(avg_delay))
```

```
## `summarise()` regrouping output by 'month' (override with `.groups` argument)
```

```
## # A tibble: 9 x 4
## # Groups:   month [1]
##   month   day     n avg_delay
##   <int> <int> <int>     <dbl>
## 1     7    10  1004     52.9
## 2     7     8  1004     37.3
## 3     7     9  1001     30.7
## 4     7    12  1002     25.1
## 5     7    11  1006     23.6
## 6     7    18  1003     20.6
## 7     7    25  1003     19.7
## 8     7    17  1001     13.7
## 9     7    31  1001      6.28
```

Binary verbs

- It's rare that a data analysis involves only a single table of data
- In practice, you'll normally have many tables that contribute to an analysis, and you need flexible tools to combine them
- All two-table verbs work similarly: the first two arguments are tables to combine and the output is always a new table

Set operations

These expect the input tables x and y to have the same variables, and treat the observations like sets, hence the resulting table have unique observations:

- `intersect(x, y)`: return observations in both x and y
- `union(x, y)`: return observations in either x or y
- `setdiff(x, y)`: return observations in x, but not in y

```
df1 = tibble(x = c(1, 2), y = c("a", "a"))
df2 = tibble(x = c(1, 1), y = c("a", "b"))

df1
```

```
## # A tibble: 2 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 a
```

```
df2
```

```
## # A tibble: 2 x 2
```

```
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     1 b
```

```
intersect(df1, df2)
```

```
## # A tibble: 1 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
```

```
union(df1, df2)
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 a
## 3     1 b
```

```
setdiff(df1, df2)
```

```
## # A tibble: 1 x 2
##       x y
##   <dbl> <chr>
## 1     2 a
```

```
setdiff(df2, df1)
```

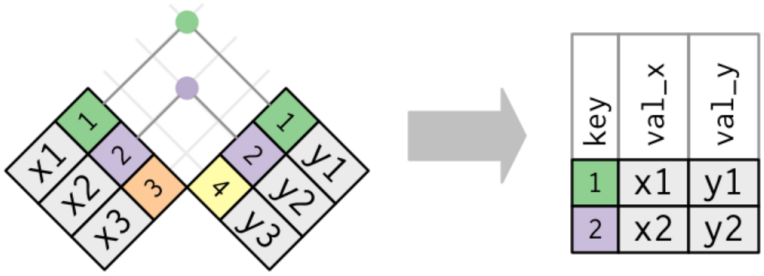
```
## # A tibble: 1 x 2
##       x y
##   <dbl> <chr>
## 1     1 b
```

Joins

- **mutating joins** add new variables to one table from matching rows in another
  - **inner join** includes observations that match in both tables
  - **outer join** (left, right, full) includes also observations that do not match in one of the tables
- **filtering joins** filter observations from one table from matching rows in another
  - **semi-join** filter observations from one table based on whether they match an observation in the other table
  - **anti-join** filter observations from one table based on whether they do not match an observation in the other table

Inner join

Inner join only includes observations in which key match in both tables.



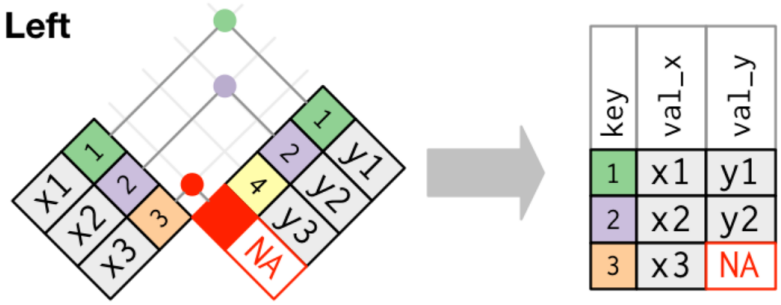

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)

inner_join(x, y)
```

```
## Joining, by = "key"
```

```
## # A tibble: 2 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
```

Left join

Left join includes observations that match in both tables (like inner join) plus the observations of the left table that find no match in the right table.




```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
```

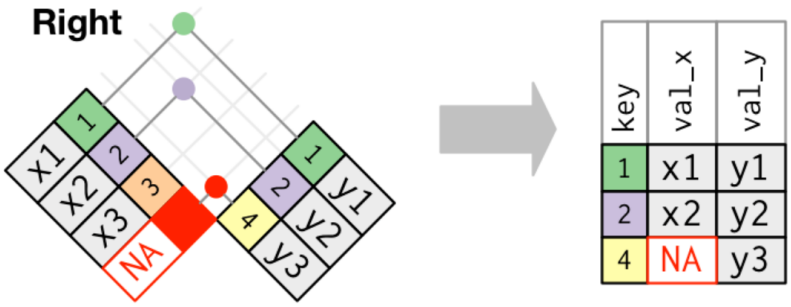
```
    2, "x2",
    3, "x3"
)
y <- tribble(
  ~key, ~val_y,
    1, "y1",
    2, "y2",
    4, "y3"
)

left_join(x, y)
```

```
## Joining, by = "key"
```

```
## # A tibble: 3 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
```

Right join

Right join includes observations that match in both tables (like inner join) plus the observations of the right table that find no match in the left table.




```
x <- tribble(
  ~key, ~val_x,
    1, "x1",
    2, "x2",
    3, "x3"
)
y <- tribble(
  ~key, ~val_y,
    1, "y1",
    2, "y2",
    4, "y3"
)

right_join(x, y)
```

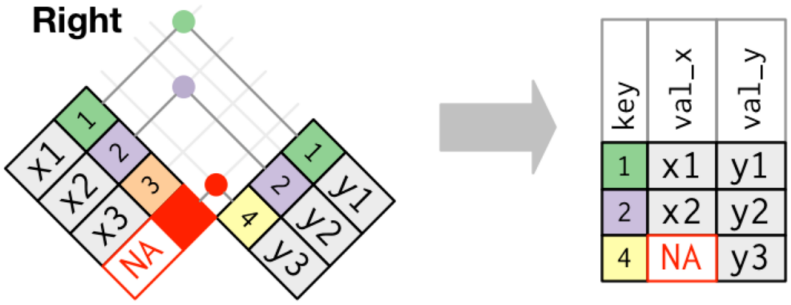
```
## Joining, by = "key"
```

```
## # A tibble: 3 x 3
```

```
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     4 <NA>  y3
```

Full join

Full join includes observations that match in both tables (like inner join) plus the observations of the left table that find no match in the right table and those in the right table that find no match in the left table.




```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)

full_join(x, y)
```

```
## Joining, by = "key"
```

```
## # A tibble: 4 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
## 4     4 <NA>  y3
```

Semi-join

Semi-join keeps all observations in the first table that have a match in the second table.

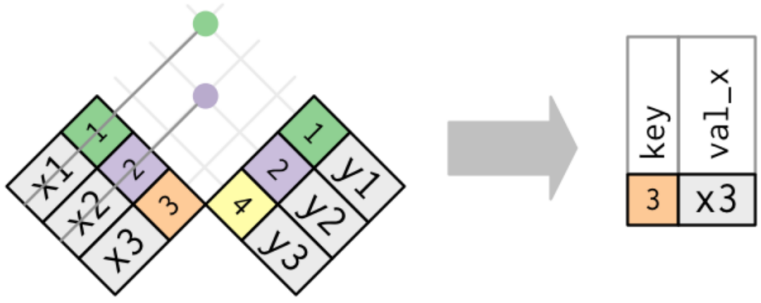

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)

semi_join(x, y)
```

```
## Joining, by = "key"
```

```
## # A tibble: 2 x 2
##     key val_x
##   <dbl> <chr>
## 1     1 x1
## 2     2 x2
```

Anti-join

Semi-join keeps all observations in the first table that have no match in the second table.




```
x <- tribble(
  ~key, ~val_x,
```

```
    1, "x1",
    2, "x2",
    3, "x3"
)
y <- tribble(
  ~key, ~val_y,
    1, "y1",
    2, "y2",
    4, "y3"
)

anti_join(x, y)
```

```
## Joining, by = "key"
```

```
## # A tibble: 1 x 2
##     key val_x
##   <dbl> <chr>
## 1     3 x3
```

Joins examples on Joining NYC flights dataset

A **natural join** is a join on common attributes of tables.

```
# Drop unimportant variables so it's easier
# to understand the join results
flights2 = flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)

flights2
```

```
## # A tibble: 336,776 x 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA
##  2  2013     1     1     5 LGA    IAH   N24211  UA
##  3  2013     1     1     5 JFK    MIA   N619AA  AA
##  4  2013     1     1     5 JFK    BQN   N804JB  B6
##  5  2013     1     1     5 EWR    ORD   N39463  UA
##  6  2013     1     1     5 JFK    BOS   N708JB  B6
##  7  2013     1     1     6 LGA    ATL   N668DN  DL
##  8  2013     1     1     6 EWR    FLL   N516JB  B6
##  9  2013     1     1     6 LGA    IAD   N829AS  EV
## 10  2013     1     1     6 JFK    MCO   N593JB  B6
## # ... with 336,766 more rows
```

```
airlines
```

```
## # A tibble: 16 x 2
##    carrier name
##    <chr>   <chr>
##  1 9E      Endeavor Air Inc.
##  2 AA      American Airlines Inc.
##  3 AS      Alaska Airlines Inc.
##  4 B6      JetBlue Airways
##  5 DL      Delta Air Lines Inc.
```

```
##  6 EV      ExpressJet Airlines Inc.
##  7 F9      Frontier Airlines Inc.
##  8 FL      AirTran Airways Corporation
##  9 HA      Hawaiian Airlines Inc.
## 10 MQ      Envoy Air
## 11 OO      SkyWest Airlines Inc.
## 12 UA      United Air Lines Inc.
## 13 US      US Airways Inc.
## 14 VX      Virgin America
## 15 WN      Southwest Airlines Co.
## 16 YV      Mesa Airlines Inc.
```

```
flights2 %>%
  left_join(airlines)
```

```
## Joining, by = "carrier"
```

```
## # A tibble: 336,776 x 9
##     year month   day  hour origin dest  tailnum carrier name
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA      United Air Lines Inc.
##  2  2013     1     1     5 LGA    IAH   N24211  UA      United Air Lines Inc.
##  3  2013     1     1     5 JFK    MIA   N619AA  AA      American Airlines Inc.
##  4  2013     1     1     5 JFK    BQN   N804JB  B6      JetBlue Airways
##  5  2013     1     1     5 EWR    ORD   N39463  UA      United Air Lines Inc.
##  6  2013     1     1     5 JFK    BOS   N708JB  B6      JetBlue Airways
##  7  2013     1     1     6 LGA    ATL   N668DN  DL      Delta Air Lines Inc.
##  8  2013     1     1     6 EWR    FLL   N516JB  B6      JetBlue Airways
##  9  2013     1     1     6 LGA    IAD   N829AS  EV      ExpressJet Airlines Inc.
## 10  2013     1     1     6 JFK    MCO   N593JB  B6      JetBlue Airways
## # ... with 336,766 more rows
```

Both tables flights and planes have year columns, but they mean different things, so we don't want to join them.

```
flights2
```

```
## # A tibble: 336,776 x 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA
##  2  2013     1     1     5 LGA    IAH   N24211  UA
##  3  2013     1     1     5 JFK    MIA   N619AA  AA
##  4  2013     1     1     5 JFK    BQN   N804JB  B6
##  5  2013     1     1     5 EWR    ORD   N39463  UA
##  6  2013     1     1     5 JFK    BOS   N708JB  B6
##  7  2013     1     1     6 LGA    ATL   N668DN  DL
##  8  2013     1     1     6 EWR    FLL   N516JB  B6
##  9  2013     1     1     6 LGA    IAD   N829AS  EV
## 10  2013     1     1     6 JFK    MCO   N593JB  B6
## # ... with 336,766 more rows
```

```
planes
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type          manufacturer   model  engines seats speed engine
##    <chr>   <int> <chr>         <chr>          <chr>    <int> <int> <int> <chr>
```

```
##  1 N10156   2004 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  2 N102UW   1998 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  3 N103US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  4 N104UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  5 N10575   2002 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  6 N105UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  7 N107US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  8 N108UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  9 N109UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 10 N110UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## # ... with 3,312 more rows
```

```
flights2 %>%
  left_join(planes, by = "tailnum")
```

```
## # A tibble: 336,776 x 16
##    year.x month   day  hour origin dest  tailnum carrier year.y type
##     <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>    <int> <chr>
##  1   2013     1     1     5 EWR    IAH   N14228  UA        1999 Fixe~
##  2   2013     1     1     5 LGA    IAH   N24211  UA        1998 Fixe~
##  3   2013     1     1     5 JFK    MIA   N619AA  AA        1990 Fixe~
##  4   2013     1     1     5 JFK    BQN   N804JB  B6        2012 Fixe~
##  5   2013     1     1     5 EWR    ORD   N39463  UA        2012 Fixe~
##  6   2013     1     1     5 JFK    BOS   N708JB  B6        2008 Fixe~
##  7   2013     1     1     6 LGA    ATL   N668DN  DL        1991 Fixe~
##  8   2013     1     1     6 EWR    FLL   N516JB  B6        2000 Fixe~
##  9   2013     1     1     6 LGA    IAD   N829AS  EV        1998 Fixe~
## 10   2013     1     1     6 JFK    MCO   N593JB  B6        2004 Fixe~
## # ... with 336,766 more rows, and 6 more variables: manufacturer <chr>,
## #   model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

Note that the year columns in the output are disambiguated with a suffix.

Each flight has an origin and destination airport, so we need to specify which one we want to join to:

```
flights2
```

```
## # A tibble: 336,776 x 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA
##  2  2013     1     1     5 LGA    IAH   N24211  UA
##  3  2013     1     1     5 JFK    MIA   N619AA  AA
##  4  2013     1     1     5 JFK    BQN   N804JB  B6
##  5  2013     1     1     5 EWR    ORD   N39463  UA
##  6  2013     1     1     5 JFK    BOS   N708JB  B6
##  7  2013     1     1     6 LGA    ATL   N668DN  DL
##  8  2013     1     1     6 EWR    FLL   N516JB  B6
##  9  2013     1     1     6 LGA    IAD   N829AS  EV
## 10  2013     1     1     6 JFK    MCO   N593JB  B6
## # ... with 336,766 more rows
```

```
airports
```

```
## # A tibble: 1,458 x 8
##    faa   name                      lat    lon   alt    tz dst   tzone
##    <chr> <chr>                   <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
```

```
##  1 04G   Lansdowne Airport         41.1  -80.6  1044    -5 A     America/New_Yo~
##  2 06A   Moton Field Municipal A~  32.5  -85.7   264    -6 A     America/Chicago
##  3 06C   Schaumburg Regional       42.0  -88.1   801    -6 A     America/Chicago
##  4 06N   Randall Airport           41.4  -74.4   523    -5 A     America/New_Yo~
##  5 09J   Jekyll Island Airport     31.1  -81.4    11    -5 A     America/New_Yo~
##  6 0A9   Elizabethton Municipal ~  36.4  -82.2  1593    -5 A     America/New_Yo~
##  7 0G6   Williams County Airport   41.5  -84.5   730    -5 A     America/New_Yo~
##  8 0G7   Finger Lakes Regional A~  42.9  -76.8   492    -5 A     America/New_Yo~
##  9 0P2   Shoestring Aviation Air~  39.8  -76.6  1000    -5 U     America/New_Yo~
## 10 0S9   Jefferson County Intl     48.1 -123.    108    -8 A     America/Los_An~
## # ... with 1,448 more rows
```

```
flights2 %>% left_join(airports, c("dest" = "faa"))
```

```
## # A tibble: 336,776 x 15
##     year month   day  hour origin dest  tailnum carrier name    lat   lon   alt
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr> <dbl> <dbl> <dbl>
##  1  2013     1     1     5 EWR    IAH   N14228  UA      Geor~  30.0 -95.3    97
##  2  2013     1     1     5 LGA    IAH   N24211  UA      Geor~  30.0 -95.3    97
##  3  2013     1     1     5 JFK    MIA   N619AA  AA      Miam~  25.8 -80.3     8
##  4  2013     1     1     5 JFK    BQN   N804JB  B6      <NA>   NA    NA      NA
##  5  2013     1     1     5 EWR    ORD   N39463  UA      Chic~  42.0 -87.9   668
##  6  2013     1     1     5 JFK    BOS   N708JB  B6      Gene~  42.4 -71.0    19
##  7  2013     1     1     6 LGA    ATL   N668DN  DL      Hart~  33.6 -84.4  1026
##  8  2013     1     1     6 EWR    FLL   N516JB  B6      Fort~  26.1 -80.2     9
##  9  2013     1     1     6 LGA    IAD   N829AS  EV      Wash~  38.9 -77.5   313
## 10  2013     1     1     6 JFK    MCO   N593JB  B6      Orla~  28.4 -81.3    96
## # ... with 336,766 more rows, and 3 more variables: tz <dbl>, dst <chr>,
## #   tzone <chr>
```

```
flights2 %>% left_join(airports, c("origin" = "faa"))
```

```
## # A tibble: 336,776 x 15
##     year month   day  hour origin dest  tailnum carrier name    lat   lon   alt
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr> <dbl> <dbl> <dbl>
##  1  2013     1     1     5 EWR    IAH   N14228  UA      Newa~  40.7 -74.2    18
##  2  2013     1     1     5 LGA    IAH   N24211  UA      La G~  40.8 -73.9    22
##  3  2013     1     1     5 JFK    MIA   N619AA  AA      John~  40.6 -73.8    13
##  4  2013     1     1     5 JFK    BQN   N804JB  B6      John~  40.6 -73.8    13
##  5  2013     1     1     5 EWR    ORD   N39463  UA      Newa~  40.7 -74.2    18
##  6  2013     1     1     5 JFK    BOS   N708JB  B6      John~  40.6 -73.8    13
##  7  2013     1     1     6 LGA    ATL   N668DN  DL      La G~  40.8 -73.9    22
##  8  2013     1     1     6 EWR    FLL   N516JB  B6      Newa~  40.7 -74.2    18
##  9  2013     1     1     6 LGA    IAD   N829AS  EV      La G~  40.8 -73.9    22
## 10  2013     1     1     6 JFK    MCO   N593JB  B6      John~  40.6 -73.8    13
## # ... with 336,766 more rows, and 3 more variables: tz <dbl>, dst <chr>,
## #   tzone <chr>
```

Use **semi-join** to filter observations of the first table that match in the second table. For instance:

Which are the flights to the top-10 popular destinations?

```
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10) # select first 10 rows

top_dest
```

```
## # A tibble: 10 x 2
##    dest      n
##    <chr> <int>
##  1 ORD   17283
##  2 ATL   17215
##  3 LAX   16174
##  4 BOS   15508
##  5 MCO   14082
##  6 CLT   14064
##  7 SFO   13331
##  8 FLL   12055
##  9 MIA   11728
## 10 DCA    9705
```

```
semi_join(flights2, top_dest)
```

```
## Joining, by = "dest"
```

```
## # A tibble: 141,145 x 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 JFK    MIA   N619AA  AA
##  2  2013     1     1     5 EWR    ORD   N39463  UA
##  3  2013     1     1     5 JFK    BOS   N708JB  B6
##  4  2013     1     1     6 LGA    ATL   N668DN  DL
##  5  2013     1     1     6 EWR    FLL   N516JB  B6
##  6  2013     1     1     6 JFK    MCO   N593JB  B6
##  7  2013     1     1     6 LGA    ORD   N3ALAA  AA
##  8  2013     1     1     6 JFK    LAX   N29129  UA
##  9  2013     1     1     6 EWR    SFO   N53441  UA
## 10  2013     1     1     6 LGA    FLL   N595JB  B6
## # ... with 141,135 more rows
```

Use **anti-join** to check foreign key integrity.

For example, the attribute tailnum in flights table refers to (is a foreign key of) the same attribute in table planes.

The foreign key constraint claims that:

Each not null foreign key value (tailnum in flights) must correspond to a value in the referenced table (tailnum in planes).

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
```

```
## # A tibble: 722 x 2
##    tailnum     n
##    <chr>   <int>
##  1 <NA>     2512
##  2 N725MQ    575
##  3 N722MQ    513
##  4 N723MQ    507
##  5 N713MQ    483
##  6 N735MQ    396
##  7 NOEGMQ    371
##  8 N534MQ    364
```

```
##  9 N542MQ    363
## 10 N531MQ    349
## # ... with 712 more rows
```

Mind that in case foreign key integrity is violated, left join and inner join produce different outputs:

```
flights2 %>%
  left_join(planes, by = "tailnum")
```

```
## # A tibble: 336,776 x 16
##    year.x month   day  hour origin dest  tailnum carrier year.y type
##     <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>    <int> <chr>
##  1   2013     1     1     5 EWR    IAH   N14228  UA        1999 Fixe~
##  2   2013     1     1     5 LGA    IAH   N24211  UA        1998 Fixe~
##  3   2013     1     1     5 JFK    MIA   N619AA  AA        1990 Fixe~
##  4   2013     1     1     5 JFK    BQN   N804JB  B6        2012 Fixe~
##  5   2013     1     1     5 EWR    ORD   N39463  UA        2012 Fixe~
##  6   2013     1     1     5 JFK    BOS   N708JB  B6        2008 Fixe~
##  7   2013     1     1     6 LGA    ATL   N668DN  DL        1991 Fixe~
##  8   2013     1     1     6 EWR    FLL   N516JB  B6        2000 Fixe~
##  9   2013     1     1     6 LGA    IAD   N829AS  EV        1998 Fixe~
## 10   2013     1     1     6 JFK    MCO   N593JB  B6        2004 Fixe~
## # ... with 336,766 more rows, and 6 more variables: manufacturer <chr>,
## #   model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

```
flights2 %>%
  inner_join(planes, by = "tailnum")
```

```
## # A tibble: 284,170 x 16
##    year.x month   day  hour origin dest  tailnum carrier year.y type
##     <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>    <int> <chr>
##  1   2013     1     1     5 EWR    IAH   N14228  UA        1999 Fixe~
##  2   2013     1     1     5 LGA    IAH   N24211  UA        1998 Fixe~
##  3   2013     1     1     5 JFK    MIA   N619AA  AA        1990 Fixe~
##  4   2013     1     1     5 JFK    BQN   N804JB  B6        2012 Fixe~
##  5   2013     1     1     5 EWR    ORD   N39463  UA        2012 Fixe~
##  6   2013     1     1     5 JFK    BOS   N708JB  B6        2008 Fixe~
##  7   2013     1     1     6 LGA    ATL   N668DN  DL        1991 Fixe~
##  8   2013     1     1     6 EWR    FLL   N516JB  B6        2000 Fixe~
##  9   2013     1     1     6 LGA    IAD   N829AS  EV        1998 Fixe~
## 10   2013     1     1     6 JFK    MCO   N593JB  B6        2004 Fixe~
## # ... with 284,160 more rows, and 6 more variables: manufacturer <chr>,
## #   model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

Flights that flew with a plane manufactured by BOEING

```
# flights that flew with a plane manufactured by BOEING
inner_join(flights, planes, by="tailnum") %>%
  select(id, tailnum, manufacturer) %>%
  filter(manufacturer == "BOEING")
```

```
## # A tibble: 82,912 x 3
##       id tailnum manufacturer
##    <int> <chr>   <chr>
##  1     1 N14228  BOEING
##  2     2 N24211  BOEING
##  3     3 N619AA  BOEING
```

```
##  4     5 N39463  BOEING
##  5     7 N668DN  BOEING
##  6    14 N29129  BOEING
##  7    15 N53441  BOEING
##  8    17 N76515  BOEING
##  9    25 N53442  BOEING
## 10    28 N633AA  BOEING
## # ... with 82,902 more rows
```

```
# dplyr (more efficient)
inner_join(flights,
           filter(planes, manufacturer == "BOEING"),
           by="tailnum") %>%
  select(id, tailnum, manufacturer)
```

```
## # A tibble: 82,912 x 3
##       id tailnum manufacturer
##    <int> <chr>   <chr>
##  1     1 N14228  BOEING
##  2     2 N24211  BOEING
##  3     3 N619AA  BOEING
##  4     5 N39463  BOEING
##  5     7 N668DN  BOEING
##  6    14 N29129  BOEING
##  7    15 N53441  BOEING
##  8    17 N76515  BOEING
##  9    25 N53442  BOEING
## 10    28 N633AA  BOEING
## # ... with 82,902 more rows
```

Flights that flew to a destination with an altitude greater than 6000 feet sorted by altitude

```
# flights that flew to a destination with an altitude
# greater than 6000 feet sorted by altitude
flights %>%
  inner_join(filter(airports, alt > 6000), by=c("dest" = "faa")) %>%
  select(id, dest, name, alt) %>%
  arrange(alt)
```

```
## # A tibble: 253 x 4
##       id dest  name                    alt
##    <int> <chr> <chr>                 <dbl>
##  1   159 JAC   Jackson Hole Airport  6451
##  2  1068 JAC   Jackson Hole Airport  6451
##  3 34189 JAC   Jackson Hole Airport  6451
##  4 40361 JAC   Jackson Hole Airport  6451
##  5 46706 JAC   Jackson Hole Airport  6451
##  6 53110 JAC   Jackson Hole Airport  6451
##  7 59662 JAC   Jackson Hole Airport  6451
##  8 66214 JAC   Jackson Hole Airport  6451
##  9 72763 JAC   Jackson Hole Airport  6451
## 10 79319 JAC   Jackson Hole Airport  6451
## # ... with 243 more rows
```

Flights that took off with a plane with 4 engines and a visibility lower than 3 miles

```r
# flights that took off with a plane with
# 4 engines and a visibility lower than 3 miles
flights %>%
  inner_join(filter(weather, visib < 3)) %>%
  inner_join(filter(planes, engines == 4), by = "tailnum") %>%
  select(id, engines, visib)
```

```
## Joining, by = c("year", "month", "day", "origin", "hour", "time_hour")
```

```
## # A tibble: 11 x 3
##        id engines visib
##     <int>   <int> <dbl>
##  1  10073       4  2.5
##  2  11255       4  0.25
##  3  25300       4  0
##  4  35913       4  0.06
##  5  35931       4  0.06
##  6  36437       4  0.12
##  7  98137       4  0.12
##  8 126433       4  1.25
##  9 128881       4  0.25
## 10 143690       4  1
## 11 143845       4  2
```

Flights with destination and origin airports with an altitude difference of more than 6000 feet

```r
# flights with destination and origin airports with
# an altitude difference of more than 6000 feet
inner_join(flights, airports, by = c("origin" = "faa")) %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
  select(id, alt.x, alt.y) %>%
  mutate(altdelta = alt.y - alt.x) %>%
  filter(altdelta > 6000)
```

```
## # A tibble: 253 x 4
##       id alt.x alt.y altdelta
##    <int> <dbl> <dbl>    <dbl>
##  1   159    18  6451     6433
##  2   199    18  6540     6522
##  3   590    13  6540     6527
##  4  1068    18  6451     6433
##  5  1101    18  6540     6522
##  6  1519    13  6540     6527
##  7  2067    18  6540     6522
##  8  2438    13  6540     6527
##  9  2969    18  6540     6522
## 10  3355    13  6540     6527
## # ... with 243 more rows
```

```r
# Alternative: simulate non equi-join in dplyr
# using dummy variables (stack overflow)
inner_join(mutate(airports, dummy = TRUE),
           mutate(airports, dummy = TRUE),
           by = "dummy") %>%
  mutate(altdelta = alt.y - alt.x) %>%
  filter(altdelta > 6000) %>%
```

```
  inner_join(flights, by = c("faa.x" = "origin", "faa.y" = "dest"))  %>%
  select(id, alt.x, alt.y, altdelta)
```

```
## # A tibble: 253 x 4
##       id alt.x alt.y altdelta
##    <int> <dbl> <dbl>    <dbl>
##  1   199    18  6540     6522
##  2  1101    18  6540     6522
##  3  2067    18  6540     6522
##  4  2969    18  6540     6522
##  5  3841    18  6540     6522
##  6  4543    18  6540     6522
##  7  5439    18  6540     6522
##  8  6362    18  6540     6522
##  9  7266    18  6540     6522
## 10  8173    18  6540     6522
## # ... with 243 more rows
```

Check that attribute tailnum is a key for table planes, that is it identifies the table observations.

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: tailnum <chr>, n <int>
```

- check that the foreign key constraint from attribute dest of table flights to attribute faa of table airports does not hold
- set to NA the attribute dest of the rows of flights that do not match an airport
- do not remove the rows of flights that do not match an airport since they contain other useful information
- then check the foreign key constraint again

```
flights %>%
  anti_join(airports, by = c("dest" = "faa")) %>%
  count(dest, sort = TRUE)
```

```
## # A tibble: 4 x 2
##   dest      n
##   <chr> <int>
## 1 SJU    5819
## 2 BQN     896
## 3 STT     522
## 4 PSE     365
```

```
id_set =
  flights %>%
  anti_join(airports, by = c("dest" = "faa")) %>%
  pull(id)

flightsValid =
  flights %>%
  mutate(dest = ifelse(id %in% id_set, NA, dest))

flightsValid %>%
  anti_join(airports, by = c("dest" = "faa")) %>%
  count(dest, sort = TRUE)
```

```
## # A tibble: 1 x 2
##   dest      n
##   <chr> <int>
## 1 <NA>   7602
```

6. A grammar for data visualization - ggplot2

ggplot2 is a system for declaratively creating graphics. You provide:

- the data,
- what graphical primitives to use,
- tell how to map variables to aesthetics,

and it takes care of the details.

The following is a reusable template for making graphs with ggplot2:

```
ggplot(data = <DATA FRAME>) +
  <GEOMETRIC OBJECT>(mapping = aes(<MAPPINGS>))
```

1. the **data frame** contains the variables we want to display
2. the **geometric object** is the geometry used by the plot to represent data (scatterplot, barplot, boxplot, and more)
3. the **aesthetic mapping** associates data frame variables to aesthetics of the plot.
4. An aesthetics is a visual property of the objects in your plot (size, shape, color, x and y positions)

The mpg dataset

We will work with the mpg dataset that contains fuel economy data and use the following variables:

- **displ**: engine displacement, in litres
- **hwy**: highway miles per gallon
- **drv** drive type: f = front-wheel drive, r = rear wheel drive, 4 = 4wd

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.0.3
```

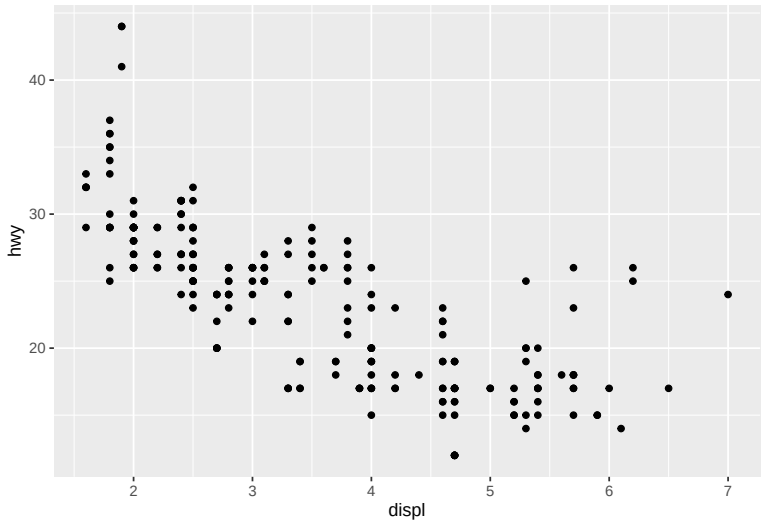
```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model    displ  year   cyl trans   drv     cty   hwy fl    class
##    <chr>        <chr>    <dbl> <int> <int> <chr>   <chr> <int> <int> <chr> <chr>
##  1 audi         a4         1.8  1999     4 auto(l~ f        18    29 p     comp~
##  2 audi         a4         1.8  1999     4 manual~ f        21    29 p     comp~
##  3 audi         a4         2    2008     4 manual~ f        20    31 p     comp~
##  4 audi         a4         2    2008     4 auto(a~ f        21    30 p     comp~
##  5 audi         a4         2.8  1999     6 auto(l~ f        16    26 p     comp~
##  6 audi         a4         2.8  1999     6 manual~ f        18    26 p     comp~
##  7 audi         a4         3.1  2008     6 auto(a~ f        18    27 p     comp~
##  8 audi         a4 quat~   1.8  1999     4 manual~ 4        18    26 p     comp~
##  9 audi         a4 quat~   1.8  1999     4 auto(l~ 4        16    25 p     comp~
## 10 audi         a4 quat~   2    2008     4 manual~ 4        20    28 p     comp~
## # ... with 224 more rows
```

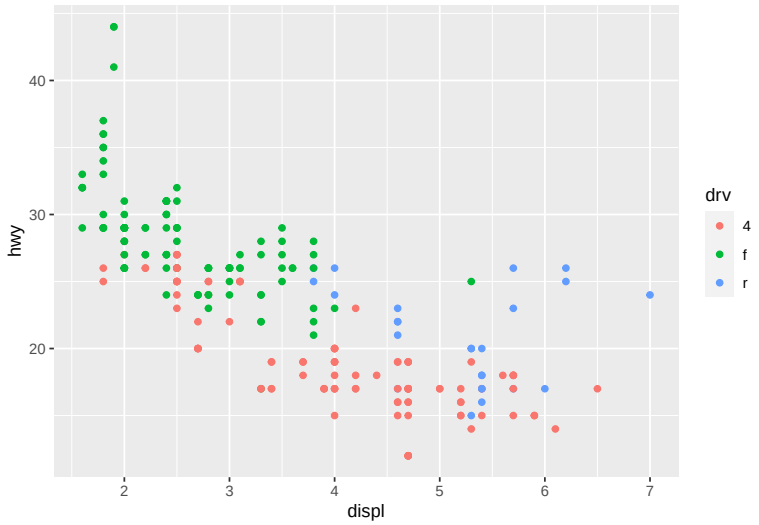
Aesthetics mapping

Let us stick to the well-known scatterplot geometric object `geom_point()` for the moment.

```
library(ggplot2)

# scatteplot of displ and hwy
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```




```
# add drv as color
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = drv))
```



Geometric objects

We will discover different geometric objects by exploring variation and covariation of qualitative and quantitative variables:

- **variation** is the tendency of the values of a variable to change from measurement to measurement
- **covariation** is the tendency for the values of two or more variables to vary together in a related way
- if variation describes the behavior *within* a variable, covariation describes the behavior *between* variables

The diamonds dataset

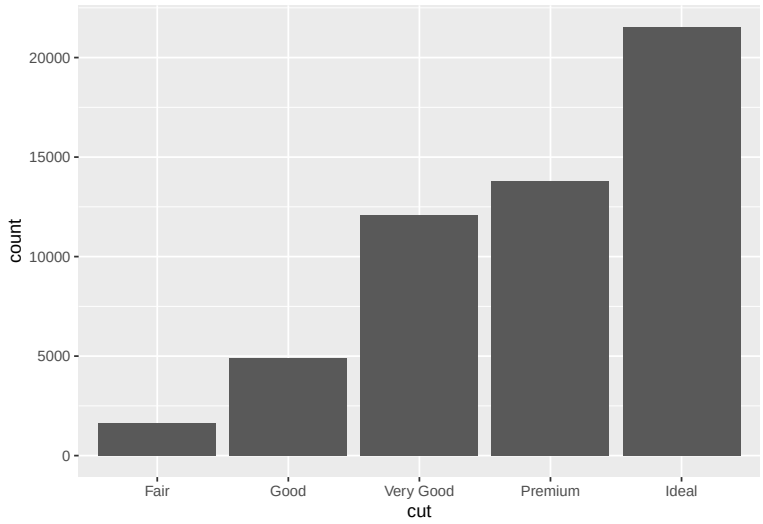
A dataset diamonds contains the prices and other attributes of almost 54,000 diamonds. The main attributes are:

- **price**: price in US dollars
- **carat**: weight of the diamond
- **cut**: quality of the cut (Fair, Good, Very Good, Premium, Ideal)

- **color**: diamond colour, from D (best) to J (worst)
- **clarity**: a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))

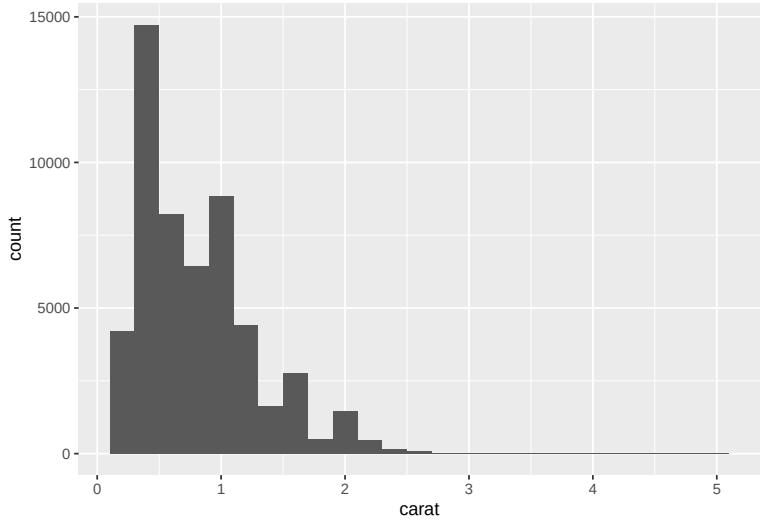
Qualitative variation: barplot

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```

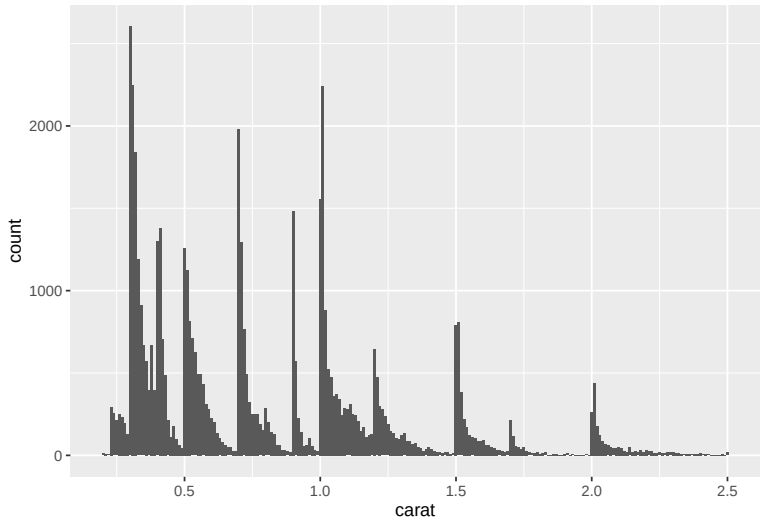


Quantitative variation: histogram

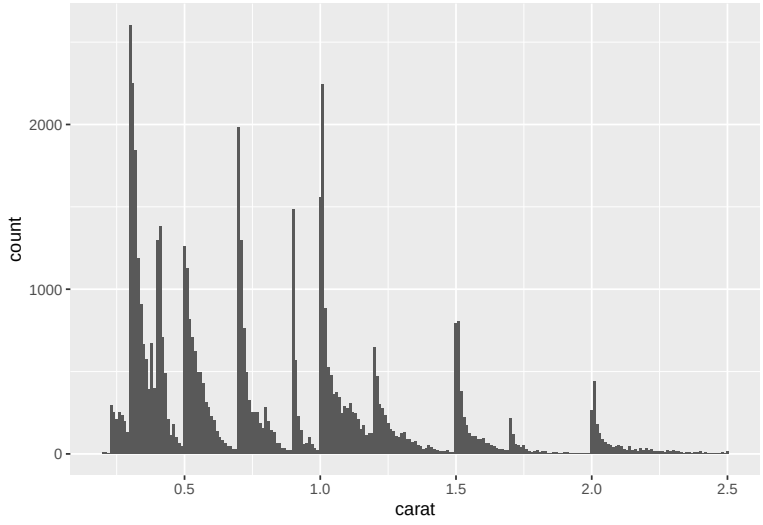
```
library(dplyr)

ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.2)
```



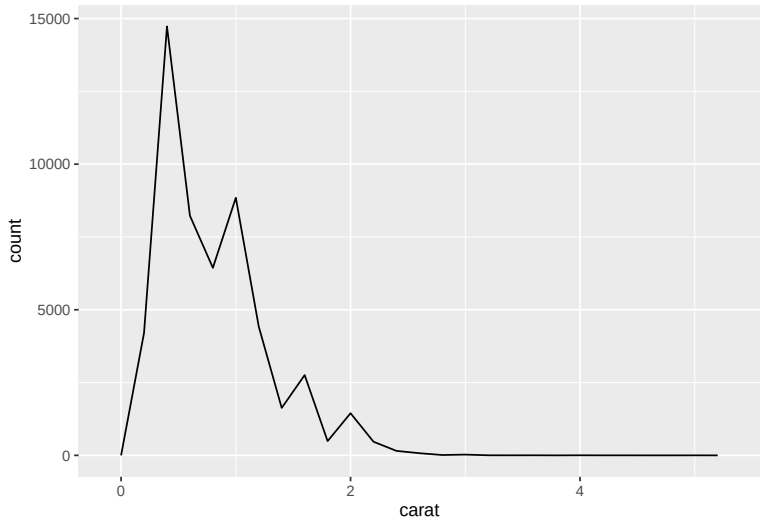

```
ggplot(data = filter(diamonds, carat <= 2.5)) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.01)
```




```
diamonds %>%
  filter(carat <= 2.5) %>%
  ggplot() +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.01)
```

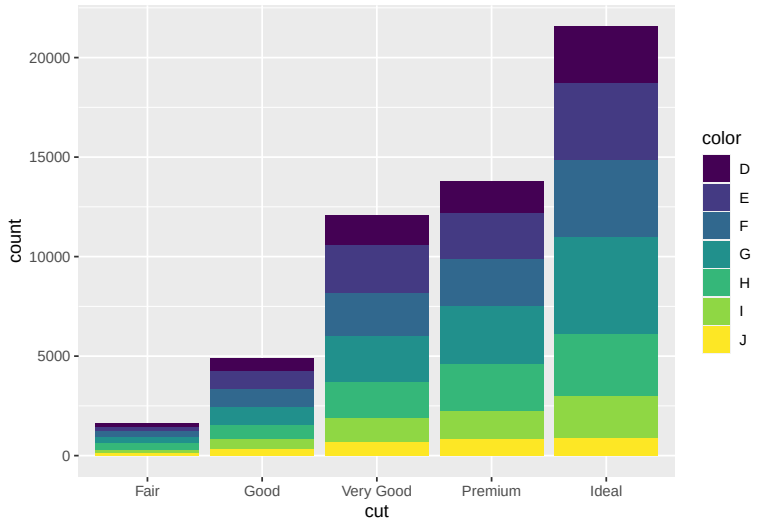



```
ggplot(data = diamonds) +
  geom_freqpoly(mapping = aes(x = carat), binwidth = 0.2)
```

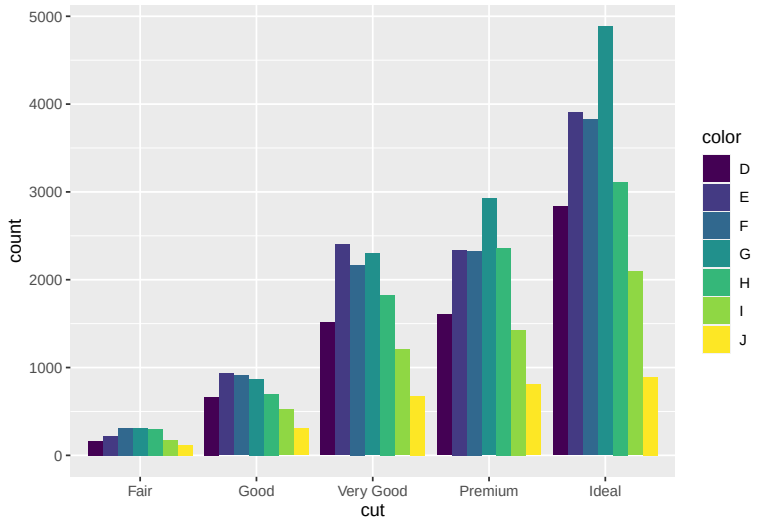


Qualitative versus qualitative covariation

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = color))
```

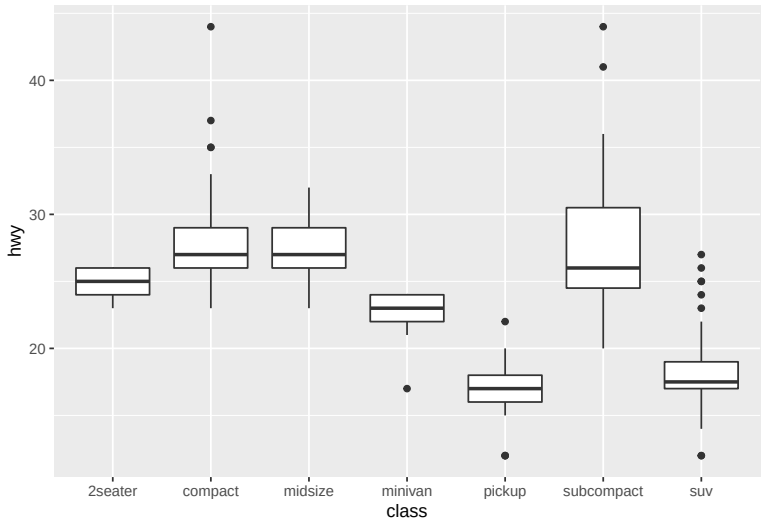



```
# putting the bars beside one another
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = color), position = "dodge")
```

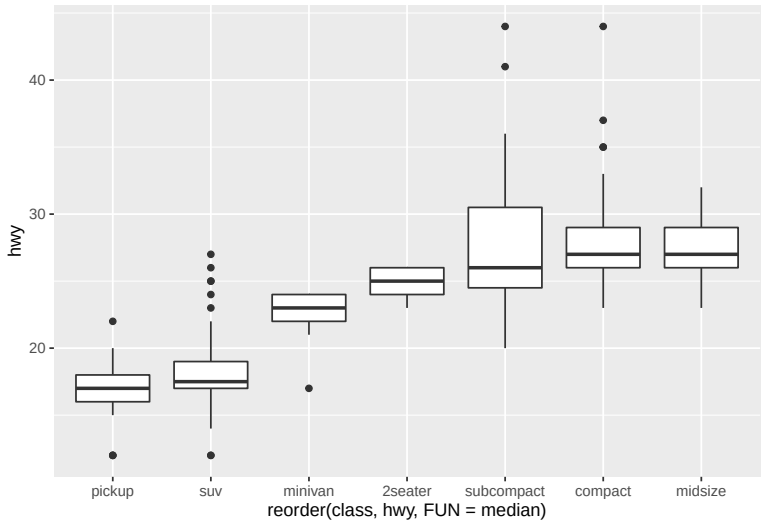


Qualitative versus quantitative covariation

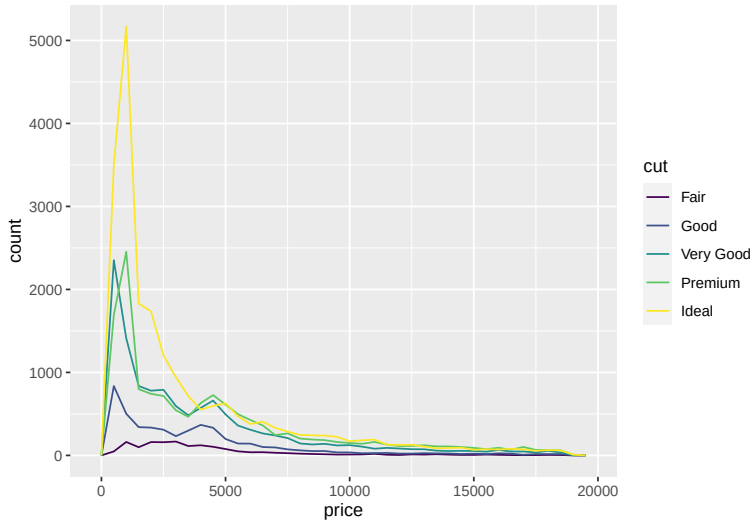
```
ggplot(data = mpg) +
  geom_boxplot(mapping = aes(x = class, y = hwy))
```




```
# reaorder qualitative variable class
ggplot(data = mpg) +
  geom_boxplot(mapping = aes(x = reorder(class, hwy, FUN = median),
                             y = hwy))
```

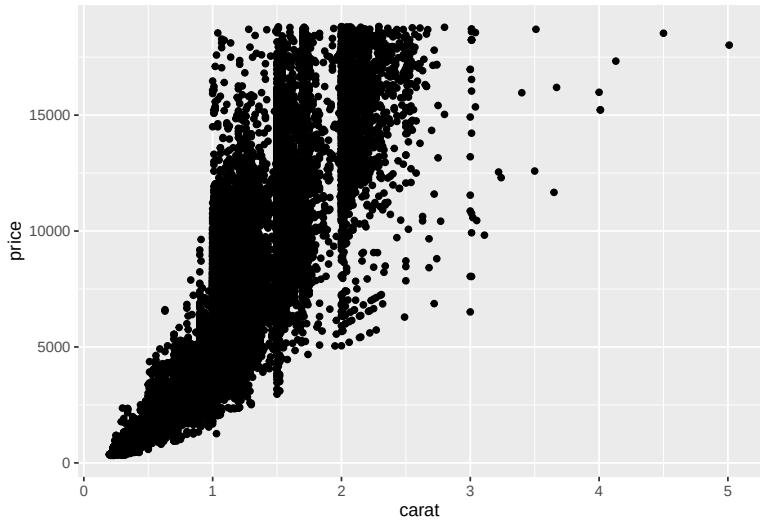



```
ggplot(data = diamonds) +
  geom_freqpoly(mapping = aes(x = price, colour = cut), binwidth = 500)
```

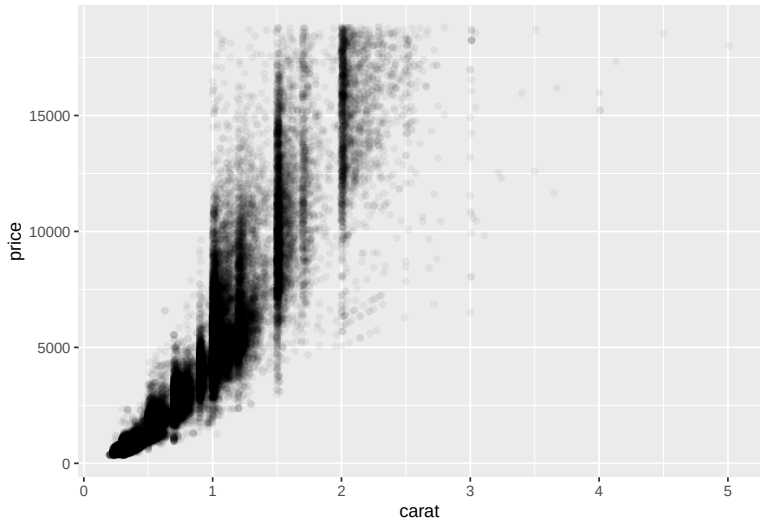


Quantitative versus quantitative covariation

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price))
```




```
# use transparency to avoid overlapping of points
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price), alpha = 0.05)
```



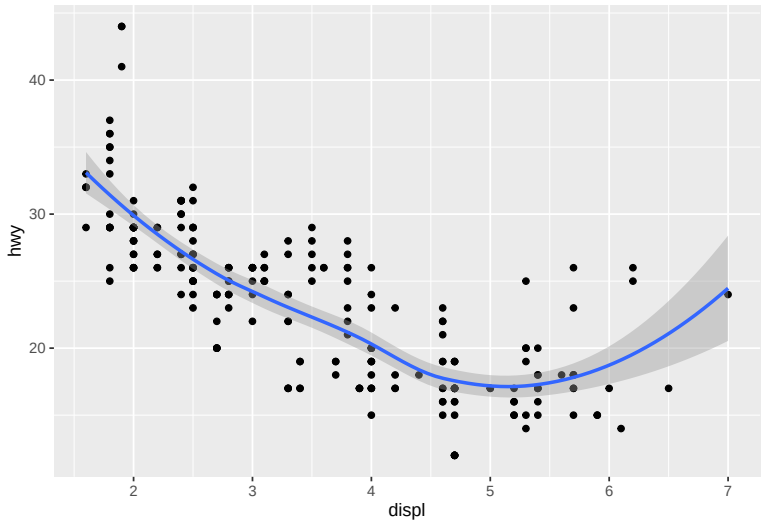
Multiple geometric objects

We overlay geom_point and geom_smooth geometries.

geom_smooth uses **local polynomial regression**, also known as moving regression, a generalization of moving average and polynomial regression.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

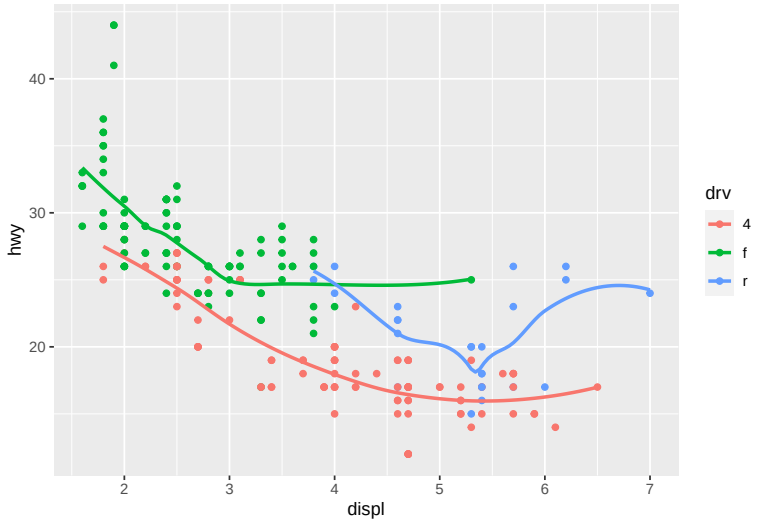


Global and local mappings

- **global mappings** of variables hold for all geometric objects in the plot
- **local mappings** of variables hold only for the geometric object containing the mapping and can overwrite global mappings

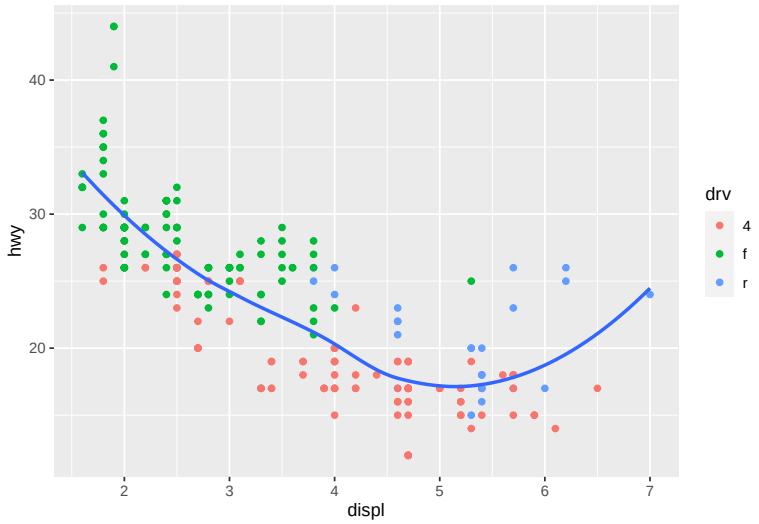
```
# color is global
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  geom_smooth(se = FALSE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



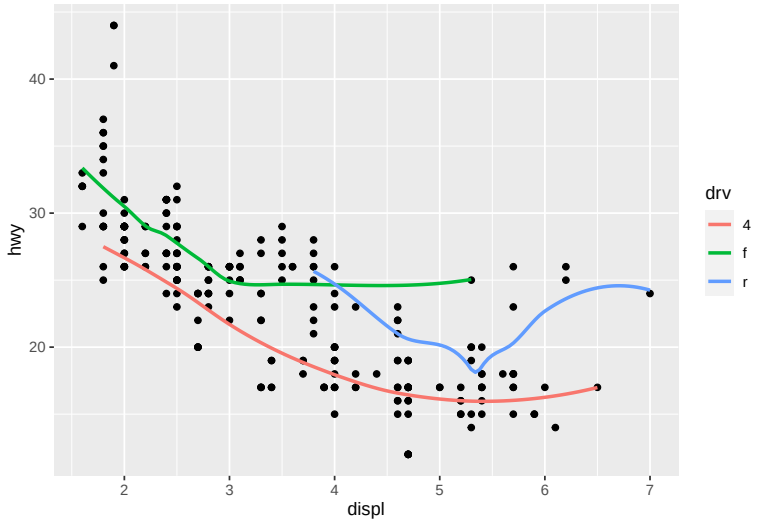

```
# color is local to geom_point
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = drv)) +
  geom_smooth(se = FALSE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
# color is local to geom_smooth
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(mapping = aes(color = drv), se = FALSE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



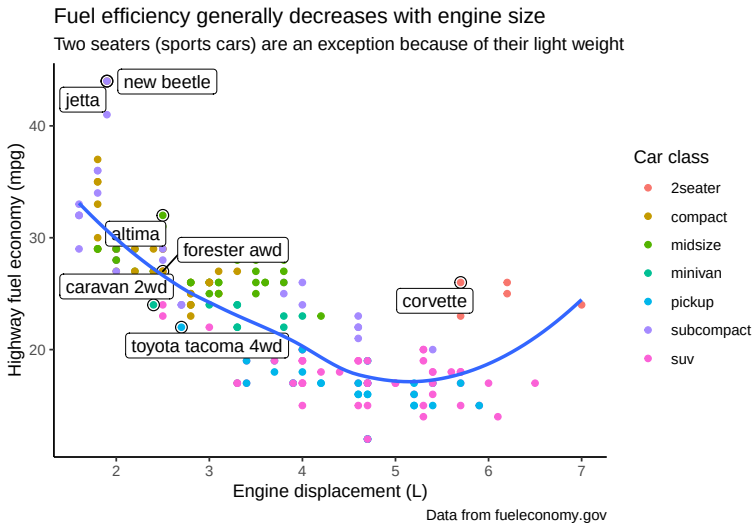
From exploratory to expository graphic

ggplot2 contains much more, that you need to exploit when turning an **exploratory** graphic into an **expository** graphic:

1. facets
2. labels
3. annotations
4. scales
5. zooming
6. themes

A final example

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
# filter best model in each class
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)

ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_point(data = best_in_class, size = 3, shape = 1) +
  ggrepel::geom_label_repel(data = best_in_class,
                            mapping = aes(label = model)) +
  geom_smooth(se = FALSE) +
  labs(
    title = "Fuel efficiency generally decreases
    with engine size",
    subtitle = "Two seaters (sports cars) are an exception
    because of their light weight",
    caption = "Data from fueleconomy.gov",
    x = "Engine displacement (L)",
    y = "Highway fuel economy (mpg)",
    colour = "Car class"
  ) +
  theme_classic()
```

Example:

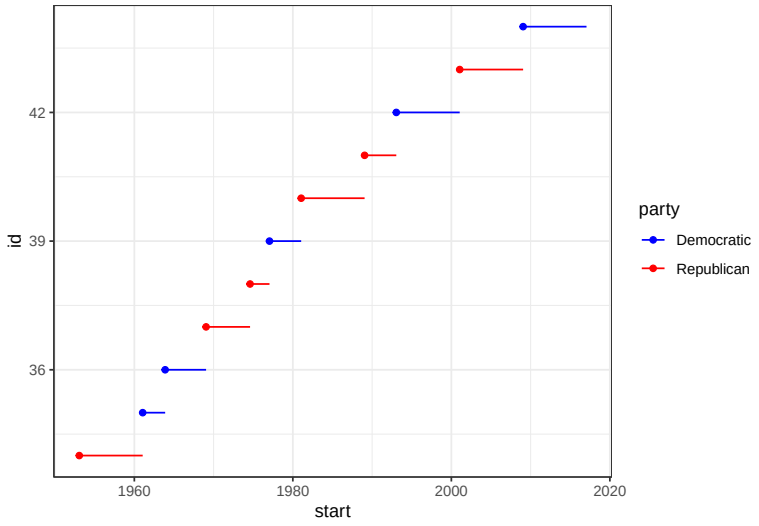
The `presidential` data set in ggplot2 package contains the names of each president, the start and end date of their term, and their party of 11 US presidents from Eisenhower to Obama.

```
presidential
```

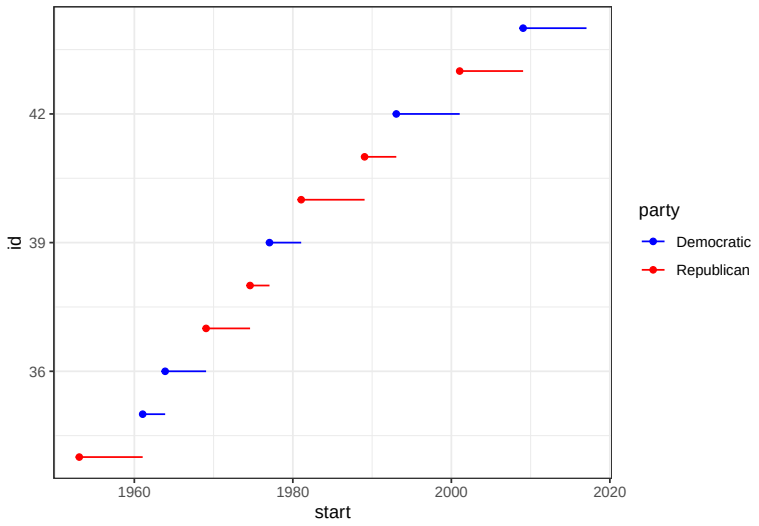
```
## # A tibble: 11 x 4
##    name       start      end        party
##    <chr>      <date>     <date>     <chr>
##  1 Eisenhower 1953-01-20 1961-01-20 Republican
##  2 Kennedy    1961-01-20 1963-11-22 Democratic
```

```
##  3 Johnson    1963-11-22 1969-01-20 Democratic
##  4 Nixon      1969-01-20 1974-08-09 Republican
##  5 Ford       1974-08-09 1977-01-20 Republican
##  6 Carter     1977-01-20 1981-01-20 Democratic
##  7 Reagan     1981-01-20 1989-01-20 Republican
##  8 Bush       1989-01-20 1993-01-20 Republican
##  9 Clinton    1993-01-20 2001-01-20 Democratic
## 10 Bush       2001-01-20 2009-01-20 Republican
## 11 Obama      2009-01-20 2017-01-20 Democratic
```

Draw the following plot:




```
presidential %>%
  mutate(id = 33 + 1:nrow(presidential)) %>%
  ggplot(aes(start, id, colour = party)) +
  geom_point() +
  geom_segment(aes(xend = end, yend = id)) +
  scale_colour_manual(values = c(Republican = "red", Democratic = "blue")) +
  theme_bw()
```



The gapminder dataset

The gapminder dataset is an excerpt of the Gapminder data on life expectancy, GDP per capita, and population by country.

```
library(gapminder)
```

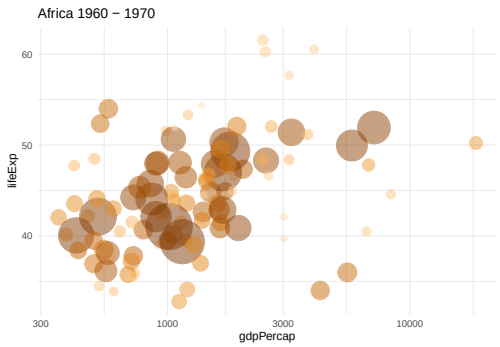
```
## Warning: package 'gapminder' was built under R version 4.0.3
```

```
gapminder
```

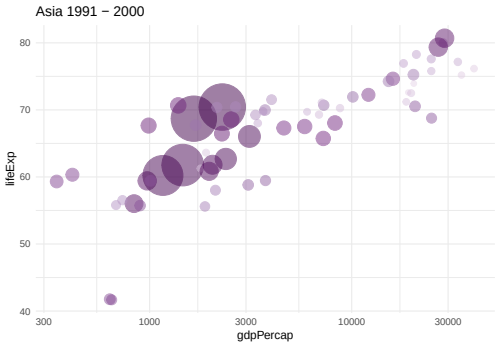
```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

1. using the gapminder dataset write a function that plots the scatterplot of GDP and life expectancy using population as size for a given continent and temporal span
2. plot the scatteplot for the same continent increasing the temporal span and for different continents with the same temporal span

```
gaplot = function(cont, year1, year2) {
  gapminder %>%
  filter(continent == cont, year >= year1, year <= year2) %>%
  ggplot(aes(gdpPercap, lifeExp, size = pop, colour = country)) +
  geom_point(alpha = 0.5, show.legend = FALSE) +
  scale_colour_manual(values = country_colors) +
  scale_size(range = c(2, 20)) +
  scale_x_log10() +
  labs(title = paste(cont, year1, "-", year2)) +
  theme_minimal()
}

gaplot("Africa", 1960, 1970)
```



```
gaplot("Asia", 1991, 2000)
```



7. Interactive graphics - shiny

- an **animated graphics** is dynamic but not interactive
- on the other hand, in **interactive graphics** the user can directly interact with the plots, for instance by changing the number of bins of an histogram in real-time
- interactive graphics is realized in R in two complementary ways: **HTML widgets** and **Shiny**

HTML widgets

HTML is an interactive format, and you can take advantage of that interactivity with HTML widgets, R functions that produce interactive HTML visualisations

For example, take the leaflet map below:

```
library(leaflet)
leaflet() %>%
  setView(174.764, -36.877, zoom = 16) %>%
  addTiles() %>%
  addMarkers(174.764, -36.877, popup = "Maungawhau")
```

- the great thing about HTML widgets is that you don't need to know anything about HTML or JavaScript to use them. All the details are wrapped inside the package, so you don't need to worry about it
- HTML widgets provide **client-side** interactivity — all the interactivity happens in the browser, independently of R
- that's great because you can distribute the HTML file without any connection to R
- however, that fundamentally limits what you can do to things that have been implemented in HTML and JavaScript

Shiny

- an alternative approach is to use shiny, a package that allows you to create interactivity using R code, not JavaScript
- shiny interactions occur on the **server-side**. This means that you need a server to run them on. T
- when you run shiny apps on your own computer, shiny automatically sets up a shiny server for you, but you need a public facing shiny server if you want to publish this sort of interactivity online
- that's the fundamental **trade-off** of shiny: you can do anything in a shiny document that you can do in R, but it requires someone to be running R
- you can share online your shiny apps using shinyapps.io

Let's reveal the basics of shiny:

- helloShiny just defines the user interface and the server logic functions
- helloWidgets showcases the possible widgets that allow interaction with the app
- helloReactivity shows how to create reative outputs reading the input of widgets

Dashboards

- Dashboards use R Markdown to publish a group of related data visualizations as a dashboard
- they are flexible and easy to specify row and column-based layouts
- components are intelligently re-sized to fill the browser and adapted for display on mobile devices
- they support both HTML widgets as well as shiny

8. Modelling

- the goal of a model is to provide a simple low-dimensional summary of a dataset
- ideally, the model will capture true **signals** (i.e. patterns generated by the phenomenon of interest), and ignore **noise** (i.e. random variation that you're not interested in)
- we are going to use models as a tool for exploration (**hypothesis generations**), and not for confirming that an hypothesis is true (**hypothesis confirmation**)

Model = patterns + residuals

We're going to use models to partition data into **patterns** (signal) and **residuals** (noise).

$$observed = pattern + residual$$

There are two parts to a model:

1. First, you define a **family of models** that express a precise, but generic, pattern that you want to capture. For example, the pattern might be a straight line, or a quadatric curve. You will express the model family as an equation like

   $$y = a_1 \cdot x + a_2$$

   or

   $$y = a_1 \cdot x^{a_2}$$

   Here, x and y are known variables from your data, and a_1 and a_2 are parameters that can vary to capture different patterns.

2. Next, you generate a **fitted model** by finding the model from the family that is the closest to your data. This takes the generic model family and makes it specific, like

   $$y = 3 \cdot x + 7$$

   or

   $$y = 9 \cdot x^2$$

The map is not the territory

It's important to understand that a fitted model is just the closest model from a family of models.

That implies that you have the *best* model (according to some criteria); it doesn't imply that you have a *good* model and it certainly doesn't imply that the model is *true*.

> The map is not the territory
>
> All models are wrong, but some are useful

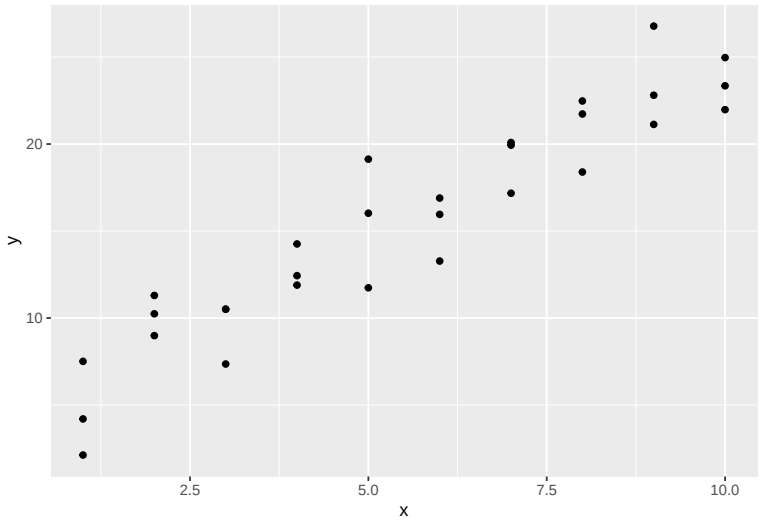
Code about models

```
library(dplyr)
library(ggplot2)
library(modelr)

# dataset
sim1
```

```
## # A tibble: 30 x 2
##        x     y
##    <int> <dbl>
##  1     1  4.20
```

```
##  2     1  7.51
##  3     1  2.13
##  4     2  8.99
##  5     2 10.2
##  6     2 11.3
##  7     3  7.36
##  8     3 10.5
##  9     3 10.5
## 10     4 12.4
## # ... with 20 more rows
```

```
# scatter plot
ggplot(sim1, aes(x,y)) + geom_point()
```




```
# correlation coefficient
cor(sim1$x, sim1$y)
```

```
## [1] 0.9405384
```

```
# linear model
mod1 <- lm(y ~ x, data = sim1)

# model (all information)
summary(mod1)
```

```
##
## Call:
## lm(formula = y ~ x, data = sim1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.1469 -1.5197  0.1331  1.4670  4.6516
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.2208     0.8688   4.858 4.09e-05 ***
## x             2.0515     0.1400  14.651 1.17e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 2.203 on 28 degrees of freedom
## Multiple R-squared:  0.8846, Adjusted R-squared:  0.8805
## F-statistic: 214.7 on 1 and 28 DF,  p-value: 1.173e-14
```

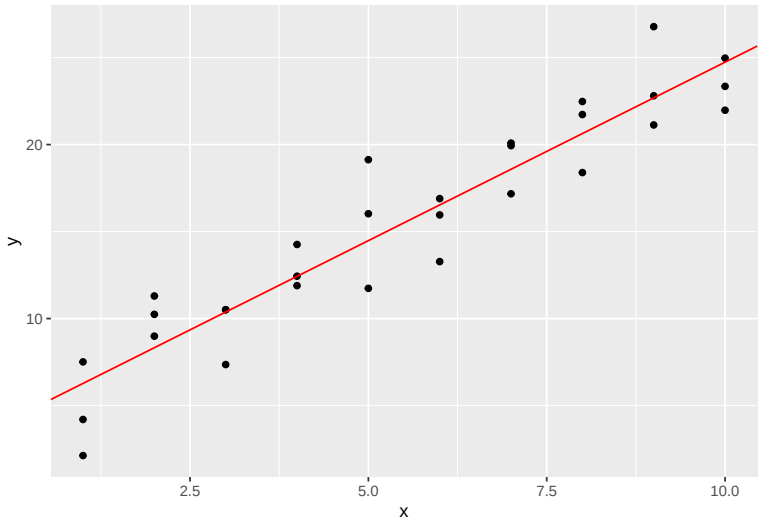
```
# coefficients
mod1$coefficients
```

```
## (Intercept)           x
##    4.220822    2.051533
```

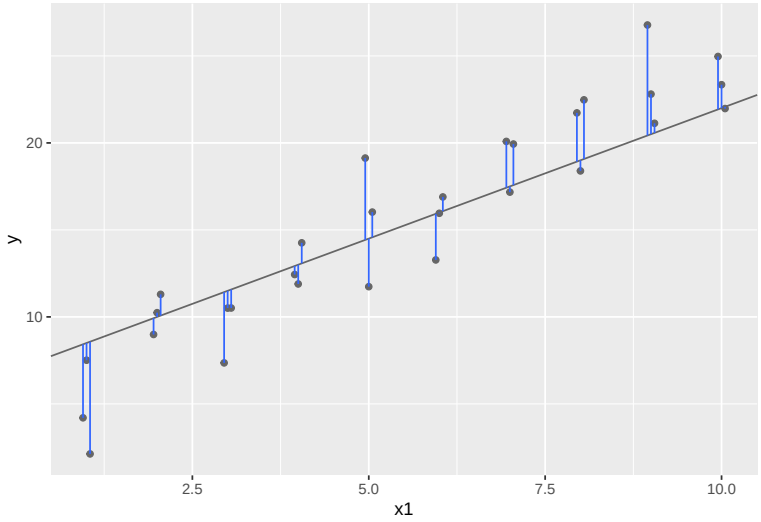
The model predics y in terms of x using the following linear relationship:

$$y = 4.220822 + 2.051533 \cdot x$$

```
# visualizing the model
ggplot(sim1, aes(x,y)) +
  geom_point() +
  geom_abline(intercept = mod1$coefficients[1],
              slope = mod1$coefficients[2],
              color = "red")
```



Response, prediction and residual



We have three interesting values in this plot for each value of variable x:

1. the value of variable y observed in the dataset (the **response**);
2. the value of variable y predicted by the model (the **prediction**)
3. the difference between observed and predicted values for variable y (the **residual**)

- the **predictions** tells you the pattern that the model has captured
- the **residuals** tell you what the model has missed
- residuals are powerful because they allow us to use models to remove striking patterns so we can study the subtler trends that remain

A general method

```
# To visualise a model, it is very useful to be able to generate
# an evenly spaced grid of points from the data
(grid <- data_grid(sim1, x))
```

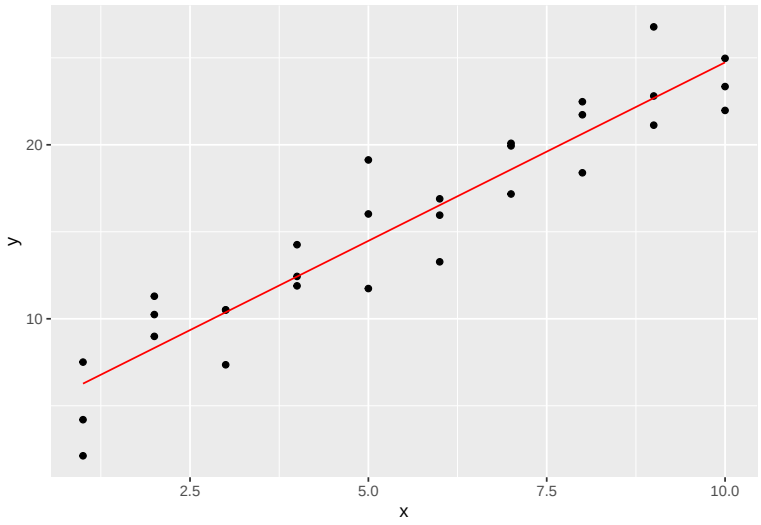
```
## # A tibble: 10 x 1
##        x
##    <int>
##  1     1
##  2     2
##  3     3
##  4     4
##  5     5
##  6     6
##  7     7
##  8     8
##  9     9
## 10    10
```

```
# add values predicted by the model over the grid
(grid <- add_predictions(grid, mod1))
```

```
## # A tibble: 10 x 2
##        x  pred
##    <int> <dbl>
```

```
##  1     1  6.27
##  2     2  8.32
##  3     3 10.4
##  4     4 12.4
##  5     5 14.5
##  6     6 16.5
##  7     7 18.6
##  8     8 20.6
##  9     9 22.7
## 10    10 24.7
```

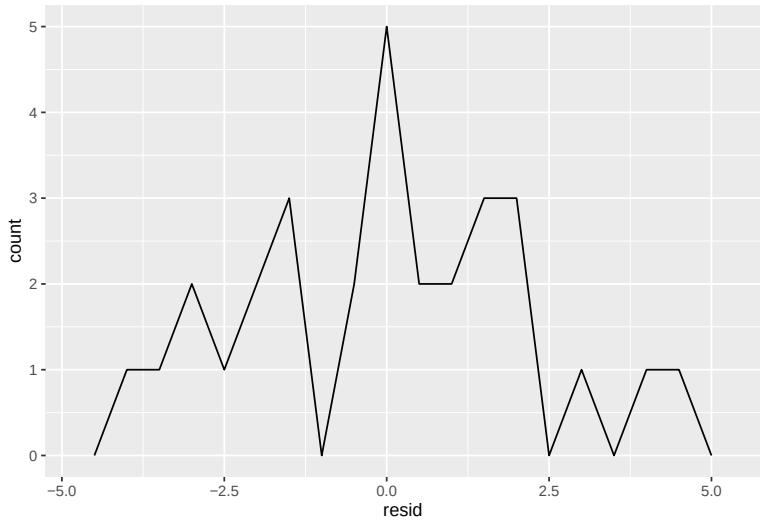
```
# plot both observed and predicted values
ggplot(sim1, aes(x = x)) +
  geom_point(aes(y = y)) + # observed values
  geom_line(data = grid, aes(y = pred), colour = "red") # predicted values
```



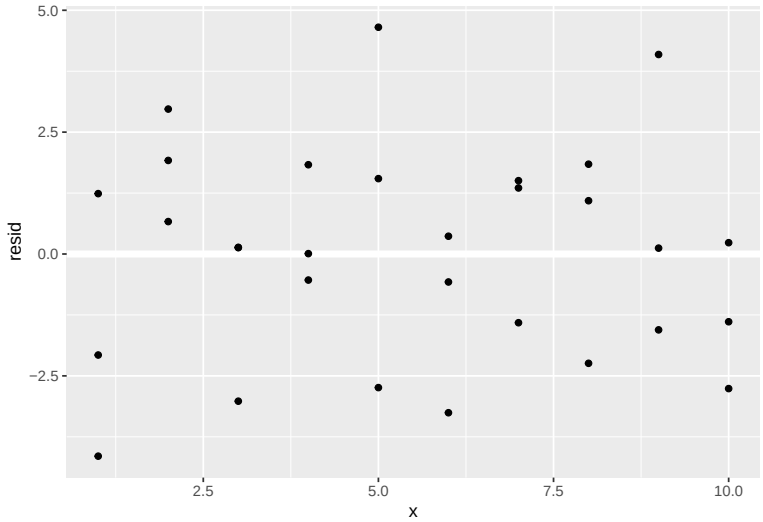

```
# add predictions to the model
sim1 <- add_predictions(sim1, mod1)

# add residuals to the model
(sim1 <- add_residuals(sim1, mod1))
```

```
## # A tibble: 30 x 4
##        x     y  pred    resid
##    <int> <dbl> <dbl>    <dbl>
##  1     1  4.20  6.27 -2.07
##  2     1  7.51  6.27  1.24
##  3     1  2.13  6.27 -4.15
##  4     2  8.99  8.32  0.665
##  5     2 10.2   8.32  1.92
##  6     2 11.3   8.32  2.97
##  7     3  7.36 10.4  -3.02
##  8     3 10.5  10.4   0.130
##  9     3 10.5  10.4   0.136
## 10     4 12.4  12.4   0.00763
## # ... with 20 more rows
```

```r
# histograph of residuals (mean is always 0)
ggplot(sim1, aes(resid)) +
  geom_freqpoly(binwidth = 0.5)
```




```r
# scatterplot of residuals (plot residuals as outcomes)
ggplot(sim1, aes(x, resid)) +
  geom_ref_line(h = 0) +
  geom_point()
```



9. Communication

The art of communication:

1. Be **honest**, don't even try to cheat
2. Be **effective**, maximize information entropy
3. Be **artistic**, make it beautiful

Principles of graphical excellence

After The Visual Display of Quantitative Information by Edward Tufte:

1. Graphical excellence is the well-designed presentation of interesting data – a matter of **substance**, of **statistics**, and of **design**
2. Graphical excellence consists of complex ideas communicated with **clarity**, **precision**, and **efficiency**
3. Graphical excellence is that which gives to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space
4. Graphical excellence is nearly always **multivariate**
5. Graphical excellence requires telling the **truth** about the data

Data Humanism

After Data Humanism, the Revolution will be Visualized by Giorgia Lupi:

1. Embrace complexity
2. Move beyond standards
3. Sneak context in
4. Remember that data is imperfect (as we are)

Embrace complexity

> Complexity is an inherent feature of our existence — the world is rich in information that can be combined in endless ways. Creating new points of view or uncovering something new typically cannot happen at a mere glance; this process of revelation often needs and requires an in-depth investigation of the context.

Move beyond standards

> Sketching with data — so, in a way, removing technology from the equation before bringing it back to finalize the design with digital tools — introduces novel ways of thinking, and leads to designs that are uniquely customized for the specific type of data problems we are working with.

Sneak context in

> As semiologists have theorized for centuries, language is only a part of the communication process — context is equally important.

Remember that data is imperfect (as we are)

> Data visualization should embrace imperfection and approximation, allowing us to envision ways to use data to feel more empathic, to connect with ourselves and others at a deeper level.

Blockchain

1. Introduction

Blockchains are interdisciplinary Imagine yourself to be a University Dean in the position of creating a new course on blockchains. To which department would you assign the course?

- **computer science** or **mathematics**?
- **economics** or **political science**?
- **sociology** or **philosophy**?

Any department

In fact, any of these departments would be appropriate.

- blockchains today are still **cutting edge and mysterious**, but one day they will be as ubiquitous as Internet and Web
- one day many academic departments will offer courses on them, each with their own particular **viewpoint**

A blockchain is:

- a **distributed system**
- using **cryptography**
- to secure an evolving **consensus**
- about a **token** with economic value

Blockchains brings together:

- **mathematics** (cryptography)
- **computer science** (distributed systems)
- **economics** (exchange of tokens with economic value)
- **politics** (mechanisms for reaching consensus)

Technical and social backgrounds Blockchains are both technological and social movements, yet very few people have both backgrounds.

- those who come from **technical backgrounds** sometimes fall in love with the novel technology inside blockchains and ignore the social aspects entirely
- this leads to projects that solve **useless** problems that no one actually has
- those who come from **social backgrounds** are sometimes unable (or unwilling) to understand the technological aspects of blockchains
- this leads to projects that are fundamentally **unsound**

More than a technology Blockchain is much more than a technology, it is also a **culture** and **community** that is passionate about creating a more equitable world through **decentralization**.

We are now entering a radical evolution of how we interact and trade because, for the first time, we can **lower uncertainty** not just with political and economic institutions but **with technology alone**.

Tokens A **cryptographic token** is a quantified and tradable unit of value recorded on the blockchain.

- **fungible** tokens are cryptocurrencies (Bitcoin and alternative coins); they are interchangeable and can be split in smaller pieces whose sum makes the whole
- **non-fungible** tokens (or NFTs, or nifties) represent something unique (for instance, a digital work of art); think of them like rare, one-of-a-kind collectibles. They are not interchangeable and cannot be divided

Types of tokens There are then different forms and functions of tokens:

- **asset tokens**. These certify that you are an owner of a good or a portion of it; transferring this token is a bit like transferring the certificate of ownership of a car. Asset tokens are typically non-fungible tokens
- **utility token**. If you spend them, you have access to a function (for example, access to a network). Utility tokens are generally fungible tokens
- **equity tokens**, which represent tokenized equity shares in a corporation or organization issues through initial coin offerings (ICOs)

Bitcoin, Ethereum and altcoins

- **Bitcoin** (BTC), first released in 2009, is generally considered the first decentralized cryptocurrency, a fungible token running on the Bitcoin blockchain
- a major alternative to Bitcoin is **Ether** (ETH), launched in 2015 and running on blockchain Ethereum
- according to CoinMarketCap, the major price-tracking website for cryptoassets, as of today (19 October 2020), the global crypto market capitalization is $358.78B and the website lists more than 1000 cryptocurrencies

Gaming: Sandbox

- the Sandbox is a virtual world built on the Ethereum blockchain, where players can build, own, and monetize their gaming experiences
- the SAND token is an utility fungible token that is used for value transfers as well as staking and governance
- the Sandbox offers an asset marketplace where virtual assets (published as non-fungible asset tokens) are bought and sold for SAND

Virtual land: Cryptovoxels and Decentraland

- Decentraland is a decentralized virtual reality world where players can own and exchange pieces of virtual land and other in-game NFT items
- Cryptovoxels is a similar game where players can build, develop, and exchange virtual property
- virtual land is associated with a non-fungible asset token that can be traded or even put on rent
- on a virtual land you can, for instance, open a digital art gallery that displays digital artworks (which are themselves non-fungible asset tokens)

Crypto art

- **crypto art** is a rising art movement
- it associates digital artworks with non-fungible asset tokens; these codes are the equivalent of the artist's signature
- early examples of crypto art include CryptoKitties, CryptoPunks, Autoglyphs, and Rare Pepe
- the most well-liked crypto art galleries today are SuperRare, KnownOrigin, MakersPlace as well as async art

Origins of blockchain

Crypto-anarchism and Cypherpunk

- the origins of blockchain go back to the crypto-anarchism and cypherpunk movements of the late 80s
- crypto-anarchists and cypherpunks advocate widespread use of **strong cryptography** in an effort to protect their privacy, their political freedom, and their economic freedom
- these movements in turn take the roots from anarcho-capitalism, a political philosophy and economic theory that advocates the elimination of **centralized states** in favor of self-ownership, private property and free markets

- which in turns refers to Laissez-faire (listen), an economic system in which **transactions between private parties** are absent any form of government intervention

Crypto Anarchist Manifesto The Crypto Anarchist Manifesto by Timothy C. May dates back to mid-1988 and was distributed to some like-minded techno-anarchists at the *Crypto '88* conference:

> Combined with emerging information markets, crypto anarchy will create a liquid market for any and all material which can be put into words and pictures. *Timothy C. May, Crypto Anarchist Manifesto*

Cypherpunk Manifesto The following excerpt from the Cypherpunk Manifesto by Eric Hughes is particularly telling since it contains, 30 years before, all ingredients of modern blockchain technology:

> We the Cypherpunks are dedicated to building anonymous systems. We are defending our privacy with cryptography, with anonymous mail forwarding systems, with digital signatures, and with electronic money. [...] Cypherpunks write code. We know that software can't be destroyed and that a widely dispersed system can't be shut down. *Eric Hughes, Cypherpunk Manifesto*

Haber and Stornetta

- the technical specification of blockchain was proposed in 1991 by **Stuart Haber**, a cryptographer, and **Scott Stornetta**, a physicist
- they published their work in *The Journal of Cryptography* in 1991 under the title How to Time-Stamp a Digital Document and one year later they registered it with a US patent

Haber and Stornetta were trying to deal with the epistemiological problem of **truth in the digital age**:

> The prospect of a world in which all text, audio, picture and video documents are in digital form on easily modifiable media raises the issue of how to certify when a document was created or last changed. The problem is to time-tamp the data, not the medium. *Haber and Stornetta, How to Time-Stamp a Digital Document*

Blockchain components

The numerous components of blockchain technology can make it challenging to understand.

However, each component can be described simply and used as a building block to understand the larger complex system.

1. blocks
2. hash
3. chain
4. proof of work
5. transactions
6. digital signature
7. peer-to-peer

Blocks The building blocks of a blockchain are... blocks.

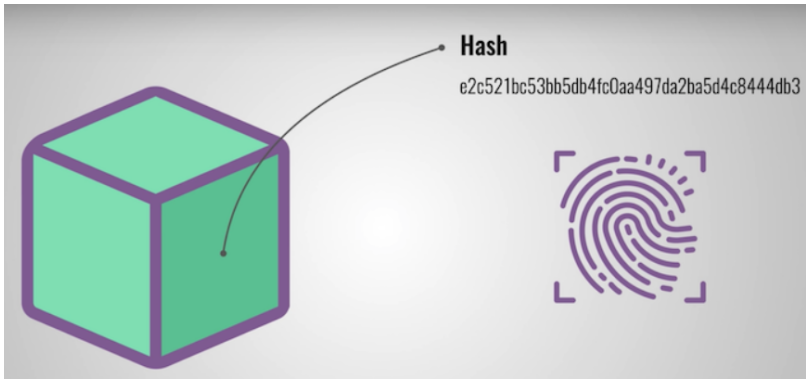
A block is a container for data

In its simplest form it contains:

- an identification number
- a timestamp of block creation
- a bunch of data (usually, transactions)

```
genesis_block = list(number = 0,
                     timestamp = "2009-01-03 18:15:05",
                     data = "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks")
```

Hash



- each block has a fingerprint called **hash** that is used to certify the information content of the block
- hashes of blocks are created using **cryptographic hash functions**, that are mathematical algorithms that maps data of arbitrary size to a bit string of a fixed size
- a popular hash algorithm is SHA-256, designed by the United States National Security Agency (NSA)
- it uses a hash of 256 bits (32 bytes), represented by an hexadecimal string of 64 figures
- $2^{256} \approx 10^{77}$ is huge (more or less the estimated number of atoms of our universe), an infinite number for any practical purposes

The ideal cryptographic hash function has five main properties:

- it is deterministic so the same message always results in the same hash
- it is quick to compute the hash value for any given message
- a small change to a message should change the hash value extensively
- it is infeasible (but not impossible) to generate a message from its hash value
- it is infeasible (but not impossible) to find two different messages with the same hash value

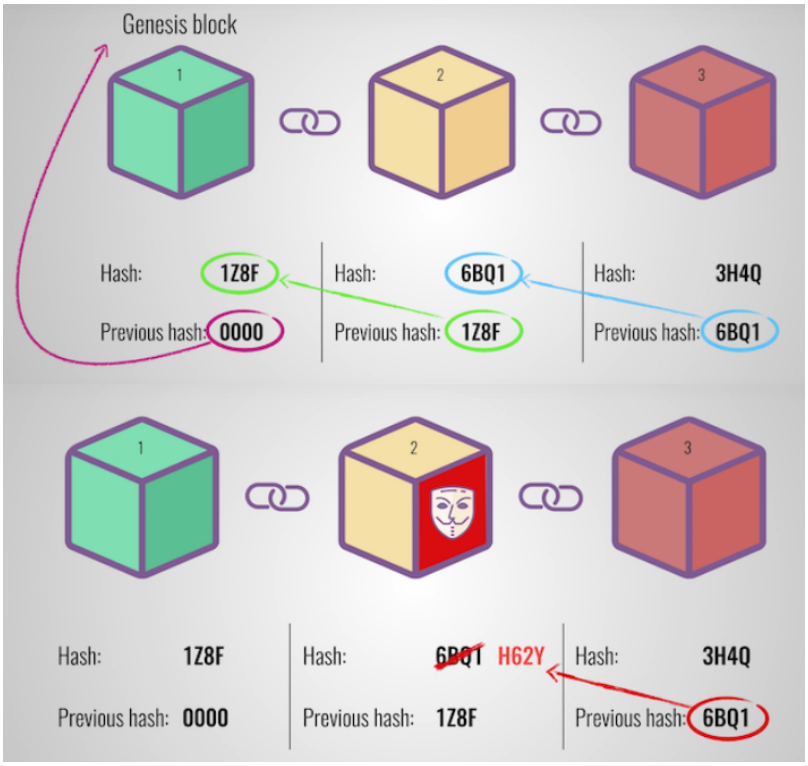
```
# load library
library(digest)

# hash a string
digest("Così tra questa immensità s'annega il pensier mio:
       e il naufragar m'è dolce in questo mare", "sha256")
```

```
## [1] "6adbb5533c24952dc70055cb22a8b07c81c34b57d6bbf580b7469737a2221f07"
```

```
# hash a slightly different string
digest("Così tra questa infinità s'annega il pensier mio:
       e il naufragar m'è dolce in questo mare", "sha256")
```

```
## [1] "063a12b2b1940bebc1b61cca3fa10834a30dd94dd8bf740c0f67f07cde179fef"
```

Chain



- blocks are chronologically concatenated into a chain by adding to the block a field with the hash of the previous block in the chain
- it follows that the hash of each block is computed using also the hash of the previous block
- this means if you alter one block you need to modify not only the hash of it but that of all following blocks for the chain to be valid
- the first block of the chain is called the **genesis block** and represents the initial state of the system (Bitcoin genesis block, Ethereum genesis block)

```
mine <- function(previous_block, genesis = FALSE){
  if (genesis) {
    # define genesis block
    new_block <- list(number = 0,
                      timestamp = Sys.time(),
                      data = "I'm genesis block",
                      parent_hash = "0")
  } else {
    # create new block
    current_number = previous_block$number + 1
    new_block <- list(number = current_number,
                      timestamp = Sys.time(),
                      data = paste0("I'm block ", current_number),
                      parent_hash = previous_block$hash)
  }
  # add hash
  new_block$hash <- digest(new_block, "sha256")
  return(new_block)
}
```

```r
chain = function(nblocks) {
  # mine genesis block
  block_genesis <- mine(NULL, TRUE)

  # first block is the genesis block
  blockchain <- list(block_genesis)

  if (nblocks >= 2) {
    # add new blocks to the chain
    for (i in 2:nblocks){
      blockchain[[i]] <- mine(blockchain[[i-1]], FALSE)
    }
  }

  return(blockchain)
}
```

```r
chain(nblocks = 3)
```

```
## [[1]]
## [[1]]$number
## [1] 0
##
## [[1]]$timestamp
## [1] "2021-01-29 15:20:05 CET"
##
## [[1]]$data
## [1] "I'm genesis block"
##
## [[1]]$parent_hash
## [1] "0"
##
## [[1]]$hash
## [1] "302b07a5fb28c7fcd91556b6772a1b16ffbfb2e9291c10cd0f1f806625d1db7d"
##
##
## [[2]]
## [[2]]$number
## [1] 1
##
## [[2]]$timestamp
## [1] "2021-01-29 15:20:05 CET"
##
## [[2]]$data
## [1] "I'm block 1"
##
## [[2]]$parent_hash
## [1] "302b07a5fb28c7fcd91556b6772a1b16ffbfb2e9291c10cd0f1f806625d1db7d"
##
## [[2]]$hash
## [1] "c1e56bd49ce1aa7b5d356cb7311738d57be0425203f885c84908890e10e6d2b5"
##
##
## [[3]]
```

```
## [[3]]$number
## [1] 2
##
## [[3]]$timestamp
## [1] "2021-01-29 15:20:05 CET"
##
## [[3]]$data
## [1] "I'm block 2"
##
## [[3]]$parent_hash
## [1] "c1e56bd49ce1aa7b5d356cb7311738d57be0425203f885c84908890e10e6d2b5"
##
## [[3]]$hash
## [1] "bdf07b81e7f8c8ce37f7f16fb0cd71ae526be633027d6ba08506ea8bae2554ce"
```

Proof of work

Byzantine Generals Problem

- the Byzantine Generals Problem is a computer-related problem consisting in finding an agreement by communicating through messages between the different components of the network
- this is a problem that was theorised by the mathematicians Leslie Lamport, Marshall Pease and Robert Shostak in 1982, who created the metaphor of the generals
- several generals are on the verge of attacking an enemy city during a siege. They are located in different strategic areas and can only communicate via messengers in order to coordinate the decisive attack
- however, among these messengers, it is highly probable that there are traitors. The traitors carry messages that contradict the army's strategy
- the problem, therefore, lies in the ability to carry out the attack effectively despite the risk of treason. This is known as **decentralised consensus**
- the problem faced by the Byzantine generals is the same as that faced by distributed computing systems. How to reach a consensus on a distributed network where some nodes may be faulty or voluntarily corrupted?
- **proof of work** is the solution proposed by Satoshi Nakamoto for Bitcoin
- hash alone is not enough to prevent tampering, since hash values can be computed fast by computers
- a **proof of work** method is needed to control the difficulty of creating a new block
- to mine (create) a new block you have to find a solution to a computational problem that is **hard to solve and easy to verify**
- this is a **cryptographic puzzle** that can be attacked only with a brute-force approach (trying many possibilities), so that only computational power counts
- **one CPU one vote** is blockchain democracy
- typically, the proof of work problem involves finding a number (called **nonce**) that once added to the block is such that the corresponding block hash starts with a string of leading zeros of a given length called **difficulty**
- the average work that a miner needs to perform in order to find a valid nonce is **exponential** in the difficulty, while one can verify the validity of the block by executing a single hash function
- miners tend to organize themselves into **pools** whereby they work together in parallel and split the reward
- available computing power increases over time, as does the number of miners, so the puzzle difficulty is generally increasing, so that the mining frequency is approximately constant

- the **amount of energy** used for the mining process is not trivial: it is estimated that Bitcoin mining consumes as much electricity as Denmark

```
proof_of_work = function(block, difficulty) {
  block$nonce <- 0
  hash = digest(block, "sha256")
  zero = paste(rep("0", difficulty), collapse="")
  while(substr(hash, 1, difficulty) != zero) {
      block$nonce = block$nonce + 1
      hash = digest(block, "sha256")
  }
  return(list(hash = hash, nonce = block$nonce))
}
```

```
proof_of_work(genesis_block, 1)
```

```
## $hash
## [1] "02500ee1e6e3a92cbb7b48516d9ef385e2395d90f5ebe52e932deec76c6b56c2"
##
## $nonce
## [1] 2
```

```
proof_of_work(genesis_block, 2)
```

```
## $hash
## [1] "00b620609d303b088c546b6abb3f8b460e0860494c17134a7b0a6b0594793c87"
##
## $nonce
## [1] 177
```

```
proof_of_work(genesis_block, 3)
```

```
## $hash
## [1] "0003041d0e9ff0cb92ec36b551eb31b87cffadf6b6704a2b4ad757dbba054432"
##
## $nonce
## [1] 4934
```

```
proof_of_work(genesis_block, 4)
```

```
## $hash
## [1] "0000ef819f1ac0522a9027d57217202a094cd1f31ad5ab5fd268857f85823689"
##
## $nonce
## [1] 83525
```

```
mine <- function(previous_block, difficulty = 3, genesis = FALSE){

  if (genesis) {
    # define genesis block
    new_block <-  list(number = 0,
                       timestamp = Sys.time(),
                       data = "I'm genesis block",
                       parent_hash = "0")
  } else {
    # create new block
    current_number <- previous_block$number + 1
    new_block <- list(number = current_number,
```

```
                      timestamp = Sys.time(),
                      data = paste0("I'm block ", current_number),
                      parent_hash = previous_block$hash)
  }

  # add nonce with proof of work
  new_block$nonce <- proof_of_work(new_block, difficulty)$nonce

  # add hash
  new_block$hash <- digest(new_block, "sha256")
  return(new_block)
}
```

```
chain = function(nblocks, difficulty = 3) {
  # mine genesis block
  block_genesis = mine(NULL, difficulty, TRUE)

  # first block is the genesis block
  blockchain <- list(block_genesis)

  if (nblocks >= 2) {
    # add new blocks to the chain
    for (i in 2:nblocks){
      blockchain[[i]] <- mine(blockchain[[i-1]], difficulty)
    }

  }

  return(blockchain)
}
```

```
chain(nblocks = 3)
```

```
## [[1]]
## [[1]]$number
## [1] 0
##
## [[1]]$timestamp
## [1] "2021-01-29 15:20:08 CET"
##
## [[1]]$data
## [1] "I'm genesis block"
##
## [[1]]$parent_hash
## [1] "0"
##
## [[1]]$nonce
## [1] 593
##
## [[1]]$hash
## [1] "000323ed561f1356984ec6cbcb1c1de00539188793caedfa38f8dff0be42147f"
##
##
## [[2]]
```

```
## [[2]]$number
## [1] 1
##
## [[2]]$timestamp
## [1] "2021-01-29 15:20:08 CET"
##
## [[2]]$data
## [1] "I'm block 1"
##
## [[2]]$parent_hash
## [1] "000323ed561f1356984ec6cbcb1c1de00539188793caedfa38f8dff0be42147f"
##
## [[2]]$nonce
## [1] 4590
##
## [[2]]$hash
## [1] "000fc716de6b8ab2a80537d53c13fcb769dccc30e1791a5edfc6ba7a04e5d359"
##
##
## [[3]]
## [[3]]$number
## [1] 2
##
## [[3]]$timestamp
## [1] "2021-01-29 15:20:08 CET"
##
## [[3]]$data
## [1] "I'm block 2"
##
## [[3]]$parent_hash
## [1] "000fc716de6b8ab2a80537d53c13fcb769dccc30e1791a5edfc6ba7a04e5d359"
##
## [[3]]$nonce
## [1] 2560
##
## [[3]]$hash
## [1] "000b81d95f2ea53e150e0792e337d8e87a094c4d91ca3a01a37d8caf40b4dd99"
```

Transactions

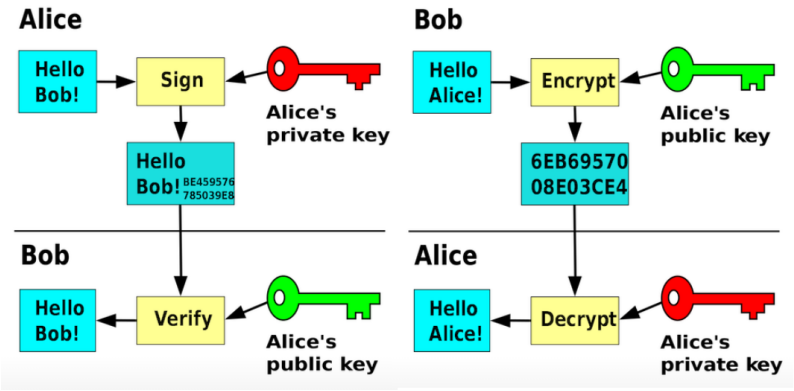
- a block contains a header with metadata (like block number and timestamp) and a data field with a certain number of **transactions**
- a transaction represents an **interaction between parties**, typically a transfer from sender to receiver of cryptocurrencies or of any other token
- each transaction has a **fee** that must be payed by the sender
- each potential miner includes in its block a subset of **pending transactions**
- the miner of the block gets the fees of all blocked transactions plus a fixed, newly minted amount of crypto coins (this is how new coins are introduced in the blockchain economy)

Digital signature

- blockchain uses **asymmetric cryptography** (also known as public-key cryptography) to implement **digital signatures** of transactions
- asymmetric cryptography uses a pair of keys: a **public key** and a **private key**
- the public key is made **public**, but the private key must remain **secret**

- even though there is a mathematical relationship between the two keys, the private key cannot efficiently be determined from the public key

Left: Signature – Right: Ecryption



Asymmetric cryptography

- asymmetric cryptography enables a trust relationship between users who do not trust one another
- it provides a mechanism to verify the integrity and authenticity of transactions while at the same time allowing transactions to remain public
- each transaction is **signed** with the sender's private key and anyone can verify the authenticity of the transaction using the sender's public key
- alternately, one can **encrypt** data with a user's public key such that only users with access to the private key can decrypt it
- this contrasts with **symmetric cryptography** in which a single pre-shared secret key is used to both encrypt and decrypt
- with symmetric cryptography users must already have a trust relationship

RSA

- RSA (Rivest–Shamir–Adleman) is one of the first asymmetric cryptography algorithms and is widely used for secure data transmission
- in RSA, the asymmetry between private and public keys is based on the practical difficulty of the factorization of the product of two large prime numbers, the **factoring problem**
- there are currently no published methods to defeat the system if a large enough key is used

```
## Digital signature

# load library
library(openssl)
```

```
## Warning: package 'openssl' was built under R version 4.0.3
```

```
## Linking to: OpenSSL 1.1.1g  21 Apr 2020
```

```
##
## Attaching package: 'openssl'
```

```
## The following object is masked from 'package:digest':
##
##     sha1
```

```
# generate a private key (key) and a public key (pubkey)
key <- rsa_keygen()
pubkey <- key$pubkey
```

```r
# build a transaction
trans = list(sender = "A", receiver = "B", amount = "100")

# serialize data
data <- serialize(trans, NULL)

# sign (a hash of) the transaction with private key
sig <- signature_create(data, sha256, key = key)

# verify the message with public key
signature_verify(data, sig, sha256, pubkey = pubkey)
```

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

```r
## Encryption

# load library
library(openssl)

# generate a private key (key) and a public key (pubkey)
key <- rsa_keygen(512)
pubkey <- key$pubkey

# message
msg <- charToRaw("Blockchain is magic!")

# cipher the message with public key
ciphermsg <- rsa_encrypt(msg, pubkey)

# decrypt the message with private key
rawToChar(rsa_decrypt(ciphermsg, key))
```

```
## [1] "Blockchain is magic!"
```

The impact of quantum computing on blockchain

- the cryptographic algorithms utilized within most blockchain technologies for asymmetric pairs (digital signature) will need to be replaced
- the hashing algorithms used by blockchain networks are much less susceptible but are still weakened

Peer-to-peer network

Finally, the blockchain ledger is distributed over a **peer-to-peer network**.

The steps to run the network are as follows:

1. new transactions are broadcast to all nodes
2. each node collects some transactions into a block
3. each node works on finding a difficult proof of work for its block
4. when a node finds a proof of work, it broadcasts the block to all nodes
5. nodes accept the block only if all transactions in it are authentic and not already spent
6. nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash (hence notice that uncompleted proof of work of miners is lost and all miners need to restart a new proof of work)

An essential glossary about blockchain

Conflicts and resolutions

- it is possible that multiple blocks will be published at approximately the same time
- this can cause differing versions of a blockchain to exist at any given moment
- these must be resolved quickly to have consistency in the blockchain network
- blockchain nodes will wait until the next block is published and use the **longer blockchain** as the official blockchain

51% attack

- an attacker might garner enough resources (more than half) to outpace the block creation rate of rest of the blockchain network
- she can now play with her rules, for instance defraud people by stealing back her payments
- why is this attack deemed to fail?

The incentive of rewards may help encourage nodes to stay honest.

If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins.

He ought to find it more profitable to play by the rules (generate new coins), such rules that favor him with more new coins than everyone else combined, than to undermine the system and the **validity of his own wealth**. Satoshi Nakamoto (Bitcoin white paper)

Hard forks

- a **hard fork** is a change to a blockchain implementation that is not backwards compatible
- at a given point in time (usually at a specific block number), all nodes will need to switch to using the updated protocol
- nodes that have not updated will reject the newly formatted blocks and only accept blocks with the old format
- this results in two incompatible versions of the blockchain existing simultaneously
- a popular hard fork separated Ethereum blockchain from Ethereum Classic after The DAO scam

Proof of stake

- an alternative, less energy-consuming consensus mechanism is **proof of stake**
- the proof of stake model is based on the idea that the more **stake** a user has invested into the system, the more likely they will want the system to succeed, and the less likely they will want to subvert it
- stake is an amount of cryptocurrency that once staked is no longer available to be spent
- the likelihood of a user mining a new block is tied to the ratio of their stake to the overall staked cryptocurrency
- with this consensus model, there is no need to perform resource intensive computations
- however, **the rich gets richer** phenomenon may arise

Digital wallets

- blockchain users must manage and securely store their own private keys
- instead of recording them manually, they often use software, called **wallet**, to securely store them
- if a user loses a private key, then any digital asset associated with that key is lost, because it is computationally infeasible to generate the private key from the public one
- if a private key is stolen, the attacker will have full access to all digital assets controlled by that private key
- on blockchain there is no central authority to restore a lost password

Seed phrases, addresses, and identity

- a seed phrase is a list of words which store all the information needed to access to a wallet
- an example of a seed phrase is: *witch collapse practice feed shame open despair creek road again ice least*
- the seed phrase can be converted to a number which is used as the seed integer to a deterministic wallet that generates all the key pairs (accounts) used in the wallet
- an address is a short, alphanumeric string of characters, derived from the user's public key using a cryptographic hash function, that identify an account of a wallet
- most blockchain implementations make use of addresses as the *to* and *from* endpoints in a transaction
- a person can possess many seed phrases (wallets) and a wallet can generate many pairs of public/private keys (accounts), this allows for a varying degree of **anonymity**

Privacy

- the traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party
- the necessity to announce all transactions publicly precludes this method
- but privacy can still be maintained by keeping **public keys anonymous**
- this is similar to the level of information released by stock exchanges, where the time and size of individual trades, the *tape*, is made public, but without telling who the parties were

Blockchain and GDPR

- how blockchain will comply with General Data Protection Regulation (GDPR)?
- any company storing personal data of EU citizens should follow the regulation
- mind that your digital wallet address might be personal data because it can link to your identity if for instance you bought crypto currency using a credit card or a digital exchange using the Know Your Customer (KYC) process
- there are three articles in the GDPR that conflicts with blockchain:
  1. Article 16: right to rectification
  2. Article 17: right to be forgotten
  3. Article 18: right to restrict processing
- but who is the data controller (the company that stores the personal data and is responsible for it) on the blockchain?

Possible solutions:

1. encrypt the personal data before you store it on blockchain
   - however encryption is in theory reversible
2. store the personal data in a permissioned blockchain, where access is restricted to only few trusted parties
   - we can comply with Article 18 (right to restrict processing)
   - but a permissioned blockchain is still immutable, hence we can't comply with Articles 16 and 17
3. store the personal data somewhere else, on a secure server, and put on the blockchain the hash of the personal data
   - you partially centralize the blockchain
4. use zero-knowledge proofs (ZKP), which allows you to prove that something is true without revealing the actual data, in case of blockchain you can prove that the transaction has happened without disclosing the data (parties and amount)

Blockchain applications

Some blockchain applications across finance, business, and government are listed below:

1. **Banking & Finance**
   - International Payments

- Capital Markets
- Trade Finance
- Regulatory Compliance and Audit
- Money Laundering Protection
- Insurance
- Peer-to-Peer Transactions

2. **Business**
   - Supply Chain Management
   - Healthcare
   - Real Estate
   - Media
   - Energy
3. **Government**
   - Record Management
   - Identity Management
   - Voting
   - Taxes
   - Non-Profit Agencies
   - Compliance/Regulatory Oversight
4. **Art**
   - Provenance and authenticity registries
   - Fractional ownership
   - Digital scarcity

2. Cryptocurrency

Fiat

Fiat money is a currency without intrinsic value that has been established as money, often by government regulation.

The term fiat derives from the Latin *fiat* ("let it be done") used in the sense of an order, decree or resolution.

Cryptocurrency

A **cryptocurrency** (or digital currency) is a digital asset designed to work as a medium of exchange that uses strong cryptography to secure financial transactions, control the creation of additional units, and verify the transfer of assets.

- cryptocurrencies use **decentralized control** as opposed to centralized digital currency and central banking systems
- they work through a **blockchain** that serves as a public financial transaction database
- **Bitcoin**, first released in 2009, is generally considered the first decentralized cryptocurrency (The rise and fall of Bitcoin)
- since the release of bitcoin many **altcoins** (alternative crypto coins) have been created

Bitcoin

- B-money was an early proposal, dated 1998 and circulated in the cypherpunk mailing lists, created by Wei Dai for an anonymous, distributed electronic cash system
- Satoshi Nakamoto (a pseudonymous) referenced B-money when proposing Bitcoin in 2009 in his white paper Bitcoin: A Peer-to-Peer Electronic Cash System
- the message of Satoshi Nakamoto embedded into the very first Bitcoin block:

  The Times 03/Jan/2009 Chancellor on brink of second bailout for banks
- considering the context in which they appear – during the bank-driven financial crisis of 2009, the worst after the economic recession of 1929 – these words are calling for economic revolution
- here is the original post of Satoshi Nakamoto proposing Bitcoin
- F2Pool added the following message, a clear reference to Satoshi's genesis block message, to block 629999, the last block before bitcoin halving:

  NYTimes 9/Apr/2020 With $2.3T Injection, Fed's Plan Far Exceeds 2008 Rescue

Bitcoin halving

- in all their infinite wisdom, bitcoin's anonymous inventor Satoshi Nakamoto decided that only **21 million** BTC would ever exist. They wanted new coins to be released gradually into the market
- new BTC are given to bitcoin miners as their bitcoin block reward when they verify blocks of transactions. To begin with, the reward stood at **50 BTC** per block
- rewards stay fixed for **210,000 blocks** (about 4 years), and then are cut by 50%. The **halving** is the moment (block creation) when this cut happens
- there will only ever be **32 bitcoin halving events**. Once the 32nd halving is completed, there will be no more new BTC created, as its maximum supply of 21 million will have been reached. Why?
- each BTC is divisible to the 8th decimal place, so each BTC can be split into 100,000,000 units. Each unit of BTC, or 0.00000001 BTC, is called a **satoshi**. A Satoshi is the smallest unit of Bitcoin. And we have that
$$50 \cdot 2^{-32} = 1.16 \cdot 10^{-8}$$
- on the other hand, there is no halving for ETH, hence the number of ETH introduced in the market will grow indefinitely

Ethereum

- a major altcoin is Ether (ETH), the currency of blockchain Ethereum, launched in 2015
- unlike other blockchains, Ethereum is **programmable**, which means that developers can use it to build new kinds of decentralized applications (**dapps**) using **smart contracts**

Smart contracts

Ethereum blockchain is enriched with smart contracts. The term **smart contract** dates to 1994, defined by Nick Szabo as:

A computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries.

The user issuing a transaction to a smart contract will have to pay a **fee** proportional to the complexity of the code executed.

Tokens on Ethereum

- a **token standard** defines a set of rules that apply to all these tokens that allow them to interact seamlessly with one another
- these rules are formalised as a minimum interface a smart contract belonging to the standard must implement to allow unique tokens to be managed, owned, and traded
- the most common standard for **fungible tokens** on Ethereum is ERC-20
- the most common standard for **non-fungible tokens** on Ethereum is ERC-721

Gas and gas price in Ethereum

- Ethereum is a **large virtual machine** that executes functions from programs called smart contracts in exchange for some fees in ether
- the machine is used by many users at the same time
- the miners compete to become the executors of the transactions, and hence to receive the fees
- all executed transactions are registered on a blockchain

Gas and gas price in Ethereum

- **gas** is a unit that measures the amount of computational effort that it will take to execute certain operations
- every single operation that takes part in Ethereum, be it a transaction or smart contract execution requires some amount of gas
- miners in theory should specify a **gas price** that reflects their cost of inclusion of the transaction in the block
- the price of gas that a user offers should generally reflect how fast they want a transaction mined (the higher, the faster)
- the gas price is expressed in gwei. The smallest unit of ether is the **wei**: 10^{18} wei make 1 ether. A **gwei** is 10^9 wei, hence 10^9 gwei make 1 ether
- for instance, if your transaction uses 20000 units of gas and the gas price is 50 gwei, then the transaction fees are:
$$20000 \cdot 50 = 10^6 \text{gwei} = 0.001 \text{ether}$$
- see ether gas station for statistics and a calculator
- if the price of gas is determined by miners, and I don't know in advance who is going to mine the block of my transaction, the fee of my transaction is not known in advance. However, you can specify a **gas limit**, the maximum amount of gas units you are allowed to pay

Cryptocurrency exchange

- a cryptocurrency exchange is a business that allows customers to trade cryptocurrencies for other assets, such as conventional fiat money or other digital currencies
- as of 2020, cryptocurrency and digital exchange regulations in many countries remain unclear as regulators are still considering how to deal with these types of businesses

Centralized exchanges

- a centralized cryptocurrency exchange is a platform owned by a private company where you can buy or sell digital assets
- on a centralized exchange you have to trust a third party to monitor the transaction and secure the assets
- such exchanges require you to submit your personal information for verification (**Know Your Coustomer** or KYC process)
- the KYC process is also a legal requirement intended as an Anti-Money Laundering (AML) measure
- **Anti-Money Laundering** (AML) describes the legal controls that require financial institutions and other regulated entities to prevent, detect and report money laundering activities

Decentralized exchanges

- A DEX or a decentralized cryptocurrency exchange is similar to a centralized one, except it doesn't have a third party on which you can rely
- all of the funds in this exchange remain stored on the blockchain
- here is a ranking of decentralized cryptocurrency exchanges by trade volume

Uniswap

- one of the top DEX for trade volume is Uniswap
- Uniswap is a protocol for exchanging ERC-20 tokens on Ethereum that eliminates trusted intermediaries (in particular there is no order book)
- the original idea comes from Vitalik Buterin, the mind behind Ethereum blockchain
- each liquidity pool on Uniswap is a pair of tokens on Ethereum
- a pool is defined by a smart contract that includes a few functions to enable **swapping** the tokens (trades) and **adding liquidity** to the pool
- the price of the token is defined by the ratio of the balances of the two tokens of the pool

Uniswap: the invariant formula

- at its core each pool uses the invariant function

$$x \cdot y = k$$

  to maintain a curve along which trades can happen, where x is the balance of the first token and y is the balance of the second token in the pool, while k is a constant
- for each trade (swap) a certain amount of tokens are removed from the pool for an amount of the other token
- this changes the balances held by the smart contract, therefore changing the price (the balance ratio)
- notice you can't empty the pool on either side since if either x or y is 0, the invariant equation doesn't hold anymore

```
# Uniswap: swap tokens
# balance of token X
x = 100

# balance of token Y
y = 10
```

```
# price of X wrt Y
y / x
```

```
## [1] 0.1
```

```
# price of Y wrt X
x / y
```

```
## [1] 10
```

```
# constant k
(k = x * y)
```

```
## [1] 1000
```

```
# swap x1 = 10 tokens of X for a number y1 of tokens of y
# use invariant formula (x + x1) * (y - y1) = k
# to compute y1 = y - k/(x + x1)
x1 = 10
y1 = y - k/(x + x1)

# new balances of X and Y
xn =  x + x1
yn = y - y1

# check invariant
xn * yn
```

```
## [1] 1000
```

```
# new price of X wrt Y
# (lower than before since there is more X in the pool)
yn / xn
```

```
## [1] 0.08264463
```

```
# new price of Y wrt X
# (higher than before since there is less Y in the pool)
xn / yn
```

```
## [1] 12.1
```

Uniswap: liquidity providing

- anyone can become a liquidity provider (LP) for a pool by depositing an equivalent value of each underlying token in return for pool tokens
- these tokens track pro-rata LP shares of the total reserves, and can be redeemed for the underlying assets at any time
- Uniswap applies a 0.30% fee to trades, which is added to reserves
- as a result, each trade actually increases k. This functions as a payout to LPs, which is realized when they burn their pool tokens to withdraw their portion of total reserves
- because the relative price of the two pair assets can only be changed through trading, divergences between the Uniswap price and external prices (for tokens listed elsewhere) create arbitrage opportunities
- this mechanism ensures that Uniswap prices always trend toward the market-clearing price (the equilibrium price at which quantity supplied is equal to quantity demanded)
- however, mind that LPs might incur in the so-called divergence loss (or impermanent loss). This loss is based on the divergence in price between deposit and withdrawal is only realised when the liquidity

provider withdraws their liquidity. This is due to the price change of the tokens in the pool: the higher the price spread, the higher the potential loss
- so the actual return for liquidity providers is a balance between the divergence loss caused by the price differential and the accumulated fees from trades on the exchange. You can track the return of your Uniswap investment using the website UniswapROI

Investing in crypto

Fidelity, one of the largest asset management funds in the world, claims in a report prepared by its research department that:

> Bitcoin is a unique investable asset with compelling differences relative to traditional asset classes as well as conventional alternative investments that could make it a beneficial addition to a portfolio.

The reasons given are:

- Bitcoin's lack of correlation with more traditional assets (stocks and commodities) that may be partially explained by the retail-driven market and the lack of overlap between institutional participants in traditional and bitcoin markets
- Bitcoin's fundamentals are relatively shielded from the economic impact of the COVID pandemic as its functionality is not predicated on profitability or production and bitcoin is natively digital

Investing in crypto: holding and trading

- **holding**: this means that you buy and keep the crypto assets in your digital wallet. This is a long-term investment strategy motivated by a bullish sentiment with respect to the held asset (you essentially believe in its fundamentals and foresee a mass adoption of the digital asset)
- **trading**: it means that you trade (buy and sell) your crypto assets against other crypto assets trying to make a revenue. The golden rule in trading is: *buy low and sell high*
- the three premises on which trading (using technical analysis) is based are:
  - Market action (price and volume) discounts everything
  - Prices move in trends
  - History repeats itself
- mind that trading is a **zero-sum game**: if you gain something, someone else loses the same. In trading you are competing with the entire market: all traders fall in one big category, from the most skillful and powerful ones to the newbies. Would you play your money at a poker table with all professional players? Yes, but only if you're a skilled professional player

Investing in crypto: decentralized finance

- DeFi stands for **decentralized finance** and refers to the ecosystem comprised of financial applications that are being developed on top of blockchain systems
- thanks to technologies like the internet, cryptography, and blockchain, DeFi aims to create a financial system that's open to everyone and minimizes one's need to trust and rely on central authorities
- these are a couple of sayings in the blockchain space that well describe the DeFi movement:
  - **Not your keys, not your money**
  - **Don't trust, verify**

Investing in crypto: decentralized finance DeFi is a way to generate rewards with cryptocurrency holdings. In simple terms, it means locking up cryptocurrencies and getting rewards.

Hence you still hold your crypto (and hence you are exposed to the price volatility of the assets) but at the same time you farm a yield.

- **lend** or **borrow** crypto for an interest; lending might be a locked or flexible saving account

- **provide liquidity** to a pool of crypto (market making) to get the fees of the trades (market taking) as well as LP tokens
- **stake** your crypto, for instance on a blockchain using the Proof of Stake consensus mechanism

DeFi protocols are permissionless and can seamlessly integrate with each other. In means that DeFi applications are **composable** – they can be easily joint together.

For instance: you can borrow a token (paying an interest), use it to provide liquidity in a pool (earning the trade fees and LP tokens), stake the LP tokens (getting a reward) and finally lend the accumulated crypto in a saving account for a yield or use it to compensate the payed interest on the initial borrow.

3. CryptoArt

Blockchain and art

Of course digital time-stamping is not limited to text. Any string of bits can be time-stamped, including digital audio recordings, photographs, and full-motion videos. [...] time-stamping can help to distinguish original photograph from a retouched one. *Haber and Stornetta, How to Time-Stamp a Digital Document, 1991*

Combined with emerging information markets, crypto anarchy will create a liquid market for any and all material which can be put into words and pictures. *Timothy C. May, Crypto Anarchist Manifesto, 1988*

Blockchain and art

- blockchain technology, while commonly associated with cryptocurrencies, has a potential to bring radical structural change to the arts and creative industries
- blockchain has core use cases in the arts including:
  - **provenance**
  - **fractional ownership**
  - **digital scarcity**

Notable examples of blockchain art

There are already several notable examples of art using blockchain technology including the following:

- in 2018, the amount collected from the sale of the Barney A. Ebsworth collection at Christie's was 318M$; the auction was held in partnership with the technology provider Artory using the blockchain to record information about the auction
- in 2018, the company Maecenas bought Andy Warhol's *14 Electric Chairs* and divided it up into shares sold as so-called ART tokens. The company raised 1.7M$ for 31.5% of the artwork at a valuation of 5.6M$
- Robert Alice's Block 21 has been the first non-fungible token sold at Christie's in October 2020. Starting from an estimate fork of 12,000$ - 18,000$, the artwork realized a price of 131,250$

Crypto art

- **crypto art** is a rising art movement that associates digital artworks with non-fungible asset tokens
- these codes are the equivalent of the artist's signature
- crypto artworks can be still or animated (gif) images, and short videos possibly with sound
- crypto art draws its origins from **conceptual art**, sharing the immaterial and distributive nature of artworks, the tight blending of artworks with currency, and the rejection of conventional art market and institutions
- early examples of crypto art include CryptoKitties, CryptoPunks, Autoglyphs, and Rare Pepe
- the most well-liked crypto art galleries today are SuperRare, KnownOrigin, MakersPlace as well as async art

We illustrate the typical workflow of crypto art with a real example from the digital gallery SuperRare.

1. an artist creates a digital artwork (an image or animation) and uploads it to the gallery
2. the smart contract of the gallery creates a non-fungible token on the Ethereum blockchain associated with the artwork and transfers the token to the artist's wallet
3. also, the gallery distributes the artwork file over the IPFS peer-to-peer network; hence nor the token nor the artwork are on any central server
4. collectors can place valued bids on the artwork by transferring a bidden amount to the smart contract of the gallery (the collector can withdraw bids at anytime)
5. eventually the artist accepts one of the bids: the smart contract of the gallery transfers the artwork's token to the collector's wallet and the agreed cryptocurrency to the artist's wallet
6. the artwork remains tradable on the market

IPFS

- the InterPlanetary File System (IPFS) is a protocol and peer-to-peer network for storing and sharing hypermedia in a distributed file system
- IPFS uses **content-addressing storage** to uniquely identify each file, a way to store information so it can be retrieved based on its content, not its location
- each file is identified by the **hash** of its content; when you look up a file to view or download, you're asking the network to find the nodes that are storing the content behind that file's hash
- each network node stores only content it is interested in, plus some indexing information that helps figure out which node is storing what
- IPFS lets you address large amounts of data and place the immutable, permanent links into blockchain transactions

What are the rights of the artist and those of the buyer?

"Ownership does not include intellectual property rights such as copyright claims, ability to produce commercially, and create merchandise therefrom, etc. The intellectual property remains the possession of the creator"

- the **corpus mysticum** of a protectable artwork is the intellectual and immaterial creation of the work
- to be perceivable the corpus mysticum must materialize in some material medium, namely the canvas of a painting, the stone for a sculpture, a CD for a musical work, the paper of the book for a novel, or a file for a digital work. This is the **corpus mechanicum**
- the artist transfers to the buyer the corpus mechanicum and not the corpus mysticum
- in particular, the possibility of commercial exploitation of any reproduction of the artwork remains with its author

Open problems

- crypto art poses a number of interesting new problems for the data scientist
- the data set of crypto art contains both **structured data** like mint, bid and sale transactions
- as well as **unstructured data** like the text that accompanies the artworks and the artwork itself as an (animated) image
- this data streams **quickly**, with events like mint, bid and sale happening every minute
- moreover, the dataset is **openly accessible** on the blockchain

Rating and Ranking

- the **rating problem** is to assign an artwork with a given score indicating its **extrinsic** value (like success of the artwork on the market in terms of bids and sales) as well as its **intrinsic** value (like the estimation given by an art expert or art curator)
- given a rating score for each artwork in a gallery, we have a **ranking** of the gallery artworks
- moreover, one can rate and rank artists and collectors by considering them as bags of artworks and extending the score from a single object to a set of objects in some suitable manner
- the rating problem is typically solved using **centrality measures** on networks that can be extracted from the dataset (like the sell or bid networks)
- possible use cases of the rating problem include:
  - a gallery wishes to acquire a piece of art for its collection and is looking for a fair estimation of the artwork
  - an auction house needs an estimation of a piece of art in view of an incoming auction
  - a collector wants to insure their art collection or make a will: in both cases they need a rating of the collection
  - an investor wants to diversify investments in the field of art and hopes to identify a set of art pieces with potential optimal return on investment (ROI)
  - a notable collector or an important artist yearn for a scrupulous evaluation of their art collection in order to mint a new crypto currency backed by their collection. Notable examples are the token

WHALE for collector WhaleShark and the token MORK for artist Hackatao

Price prediction

- the **price prediction problem** is to predict the price of an artwork at a given time in the future given a set of features of the artwork that are known at the present time
- the features for the prediction can be related to the bid and sale histories of the artwork (extrinsic features) or associated with the artwork itself (intrinsic features) such as the digital image or the more complex digital object (gif, video, sound, 3D object) representing the art
- in the simplest case, a multiple **linear regression** model is solved, with price as response (dependent) variable and the features as explanatory (independent) variables
- the price prediction can be used by an artist to set a reserve price for a new artwork or by a collector to have an estimate of how much to offer during an auction for an artwork
- it can be used to estimate how the value of a collection will evolve in the near future

Art discovery

- the **art discovery problem** it to recommend art items to users that they might not have found otherwise
- users can be artists, interested in discovering similar artists for collaboration, or collectors (including art investors), willing to find new art to purchase that is somewhat overlapping with that already present in their collection
- **recommender systems** usually make use of either or both collaborative filtering and content-based filtering
- **collaborative filtering** approaches build a model from a user's past behavior as well as similar decisions made by other users. For instance, if A and B are similar collectors in some meaningful sense and A purchased an art piece X that B does not have in their collection, then the system might recommend art X (or a similar one) to B as well
- **content-based filtering** approaches utilize a series of pre-tagged characteristics of an item in order to recommend additional items with similar properties. For instance, if a collector bought several artworks tagged as glitch art, then the system might suggest other glitch artworks not already in the in collector's gallery
- a particular case of art discovery is the **art radio** feature: starting from a seed artwork, the system generates a compilation of related artworks that the user can linearly visualize and eventually stop if something is of interest (possibly starting a second art radio from the new piece)

Network Science

1. Introduction

- networks are pervasive in the real world: **nature**, **society**, **information**, and **technology** are supported by as many networks
- these networks are ostensibly different but in fact share an amazing number of interesting structural properties
- the main common feature shared by these networks is that they are **webs without a spider**: there exist no central authority that regulates their growth, but they evolve in a self-organized and decentralized way
- **network science** is a subfield of complexity science devoted to the holistic analysis of complex systems through the study of the structure of networks that wire their components

From trees...

Trees are **hierarchical** structures typically associated with the following principles:

- **Centralism**. The tree structure has a unique root node – the only node without a parent node higher in the hierarchy – from which all other nodes descend. This expresses a concentration of power and authority in a central actor of the system
- **Finalism**. It describes the unidirectional, linear courses of trees: any point in a tree is reachable from the root by a unique, linear, top-down path of intermediate nodes. Trees therefore embody an organization devoid of multilinearity or feedback loops

...to networks.

Networks are **rhizomatic** structures associated with the following principles:

- **Decentralization**. Networks are webs without a spider: there exist no central authority (root) that regulates their growth, but they evolve in a self-organized way
- **Multilinearity**. Networks allows feedback loops, that are paths that come back on their feet. This means that actors of the system can be reached through multiple paths

Example: The Internet

- Opte Project, The Internet
- the vertices in this representation of the Internet are class C subnets - groups of computers with similar Internet addresses that are usually under the management of a single organization
- the connections between them represent the routes taken by Internet data packets as they hop between subnets
- the geometric positions of the vertices in the picture have no special meaning; they are chosen simply to give a pleasing layout and are not related, for instance, to geographic position of the vertices
- computers from different regions of the world are mapped out by color, allowing viewers to see how regions like Latin America have experienced explosive growth in Internet connectivity (white nodes are anonymous routers)

2. Network analysis: igraph

The main goals of the igraph library is to provide a set of data types and functions for

1. pain-free implementation of graph algorithms,
2. fast handling of large graphs, with millions of vertices and edges,
3. allowing rapid prototyping via high level languages like R.

Overview

- igraph is a collection of network analysis tools with the emphasis on efficiency, portability and ease of use
- igraph is open source and free
- igraph can be programmed in R, Python, Mathematica and C/C++
- the igraph manual page is a good place to start
- the igraph documentation is a good place to end

Read a graph from data frames

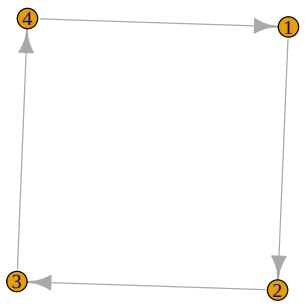
You can read a graph from a data frame representation. It consists of two data frames:

1. one data frame contains **nodes** and related attributes
2. one data frame contains **edges** (paris of nodes) and related attributes

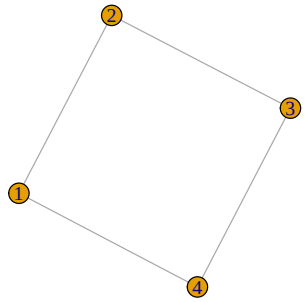
```
# read graph from data frame
actors <- data.frame(
  name = c("Alice", "Bob", "Cecil", "David", "Esmeralda"),
  age = c(48, 33, 45, 34, 21),
  gender = c("F", "M", "F", "M", "F")
)

relations <- data.frame(
  from = c("Bob", "Cecil", "Cecil", "David", "David", "Esmeralda"),
  to = c("Alice", "Bob", "Alice", "Alice", "Bob", "Alice"),
  sameDept = c(FALSE, FALSE, TRUE, FALSE, FALSE, TRUE),
  friendship = c(4, 5, 5, 2, 1, 1),
  advice = c(4, 5, 5, 4, 2, 3)
)

g = graph_from_data_frame(relations, directed = TRUE, vertices = actors)
```

Basic, notable and popular graphs

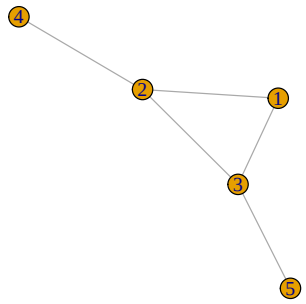
```
# make a basic graph (without attributes)
plot(make_graph(c(1,2, 2,3, 3,4, 4,1), directed = TRUE))
```



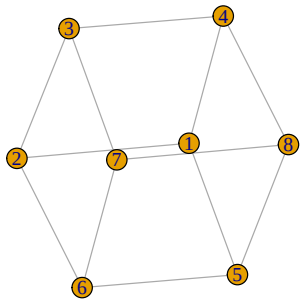
```r
plot(make_graph(c(1,2, 2,3, 3,4, 4,1), directed = FALSE))
```



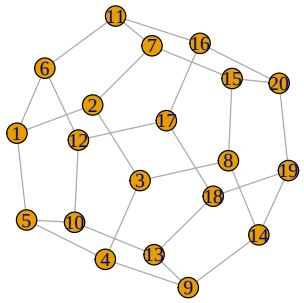

```r
# make a notable graph
plot(make_graph("Bull"))
```



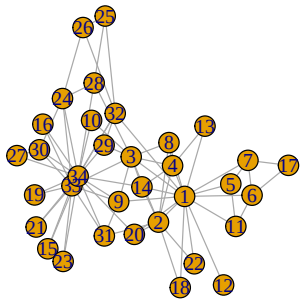

```r
plot(make_graph("Cubical"))
```



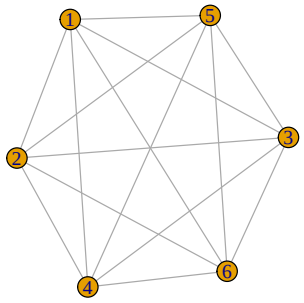

```
plot(make_graph("Dodecahedron"))
```



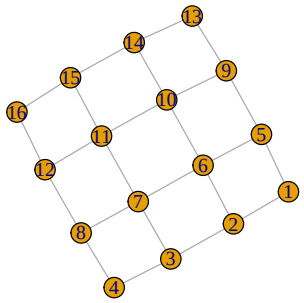

```
plot(make_graph("Zachary"))
```



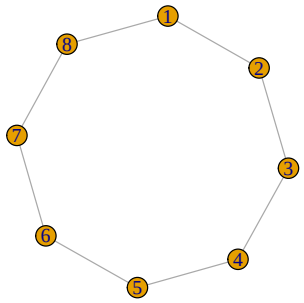

```
# make a popular graph
plot(make_full_graph(6))
```



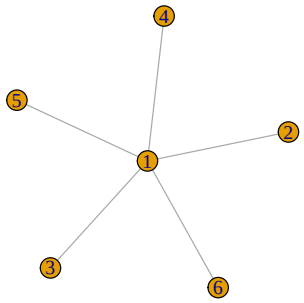

```
plot(make_lattice(c(4,4)))
```



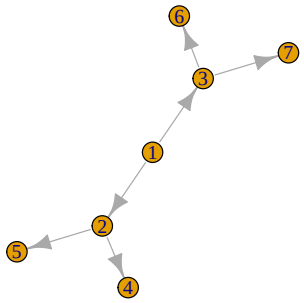

```
plot(make_ring(8))
```



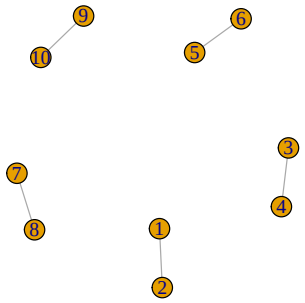

```
plot(make_star(6, mode = "undirected"))
```




```
plot(make_tree(7, children = 2))
```




```
plot(make_bipartite_graph(types = rep(0:1,length=10), edges = c(1:10)))
```



Explore the graph

```
# print graph
g
```

```
## IGRAPH 176a623 DN-- 5 6 --
## + attr: name (v/c), age (v/n), gender (v/c), sameDept (e/l), friendship
## | (e/n), advice (e/n)
## + edges from 176a623 (vertex names):
## [1] Bob      ->Alice Cecil    ->Bob   Cecil    ->Alice David    ->Alice
## [5] David    ->Bob   Esmeralda->Alice
```

```
# nodes and node count
V(g)
```

```
## + 5/5 vertices, named, from 176a623:
## [1] Alice     Bob       Cecil     David     Esmeralda
```

```
vcount(g)
```

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

```
# edges and edge count
E(g)
```

```
## + 6/6 edges from 176a623 (vertex names):
## [1] Bob      ->Alice Cecil    ->Bob   Cecil    ->Alice David    ->Alice
## [5] David    ->Bob   Esmeralda->Alice
```

```
ecount(g)
```

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

Graph attributes

A graph can have attributes for the entire graph, for nodes and for edges.

```
# add graph attribute
g$name = "Alice & friends"
graph_attr(g)
```

```
## $name
## [1] "Alice & friends"
```

```
# add node attribute
V(g)$height = c(122, 155, 178, 167, 198)
vertex_attr(g)
```

```
## $name
## [1] "Alice"     "Bob"       "Cecil"     "David"     "Esmeralda"
##
## $age
## [1] 48 33 45 34 21
##
## $gender
## [1] "F" "M" "F" "M" "F"
##
## $height
## [1] 122 155 178 167 198
```

```
# add edge attribute
E(g)$weight = c(1, 2, 2, 5, 5, 4)
edge_attr(g)
```

```
## $sameDept
## [1] FALSE FALSE  TRUE FALSE FALSE  TRUE
##
## $friendship
## [1] 4 5 5 2 1 1
##
## $advice
## [1] 4 5 5 4 2 3
##
## $weight
## [1] 1 2 2 5 5 4
```

```
# delete attribute
g <- delete_graph_attr(g, "name")
g <- delete_vertex_attr(g, "height")
g <- delete_edge_attr(g, "weight")
```

Graph iterators

Iterators allow to access groups of nodes and edges in a smart way.

```
# access nodes
V(g)[1]
```

```
## + 1/5 vertex, named, from 176a623:
## [1] Alice
```

```
V(g)["Alice"]
```

```
## + 1/5 vertex, named, from 176a623:
## [1] Alice
```

```
V(g)[1:3]
```

```
## + 3/5 vertices, named, from 176a623:
```

```
## [1] Alice Bob   Cecil
```

```
V(g)[age <= 30]
```

```
## + 1/5 vertex, named, from 176a623:
## [1] Esmeralda
```

```
V(g)[gender == "M"]
```

```
## + 2/5 vertices, named, from 176a623:
## [1] Bob   David
```

```
# access edges
E(g)[1]
```

```
## + 1/6 edge from 176a623 (vertex names):
## [1] Bob->Alice
```

```
E(g)["Bob|Alice"]
```

```
## + 1/6 edge from 176a623 (vertex names):
## [1] Bob->Alice
```

```
E(g)[1:3]
```

```
## + 3/6 edges from 176a623 (vertex names):
## [1] Bob  ->Alice Cecil->Bob   Cecil->Alice
```

```
E(g)[sameDept == TRUE]
```

```
## + 2/6 edges from 176a623 (vertex names):
## [1] Cecil    ->Alice Esmeralda->Alice
```

Switch between graph representations

We have 3 main graph representations:

1. the igraph internal representation
2. the adjcency matrix representation
3. the data frame representation

Use each representation for the right task, for instance:

1. compute a node centrality measure: use igraph representation
2. find the dominant eigenvector using the power method: use adjcency matrix representation
3. filter nodes or edges according to some condition: use data frame representation

```
# read graph from data frame
actors <- data.frame(
  name = c("Alice", "Bob", "Cecil", "David", "Esmeralda"),
  age = c(48, 33, 45, 34, 21),
  gender = c("F", "M", "F", "M", "F")
)

relations <- data.frame(
  from = c("Bob", "Cecil", "Cecil", "David", "David", "Esmeralda"),
  to = c("Alice", "Bob", "Alice", "Alice", "Bob", "Alice"),
  sameDept = c(FALSE, FALSE, TRUE, FALSE, FALSE, TRUE),
  friendship = c(4, 5, 5, 2, 1, 1),
```

```
  advice = c(4, 5, 5, 4, 2, 3)
)

g = graph_from_data_frame(relations, directed = TRUE, vertices = actors)

as_data_frame(g, what="vertices")
```

From data frame to igraph and back

```
##                name age gender
## Alice         Alice  48      F
## Bob             Bob  33      M
## Cecil         Cecil  45      F
## David         David  34      M
## Esmeralda Esmeralda  21      F
```

```
as_data_frame(g, what="edges")
```

```
##        from    to sameDept friendship advice
## 1       Bob Alice    FALSE          4      4
## 2     Cecil   Bob    FALSE          5      5
## 3     Cecil Alice     TRUE          5      5
## 4     David Alice    FALSE          2      4
## 5     David   Bob    FALSE          1      2
## 6 Esmeralda Alice     TRUE          1      3
```

```
as_data_frame(g, what="both")
```

```
## $vertices
##                name age gender
## Alice         Alice  48      F
## Bob             Bob  33      M
## Cecil         Cecil  45      F
## David         David  34      M
## Esmeralda Esmeralda  21      F
##
## $edges
##        from    to sameDept friendship advice
## 1       Bob Alice    FALSE          4      4
## 2     Cecil   Bob    FALSE          5      5
## 3     Cecil Alice     TRUE          5      5
## 4     David Alice    FALSE          2      4
## 5     David   Bob    FALSE          1      2
## 6 Esmeralda Alice     TRUE          1      3
```

```
# from igraph to adjacency matrix
# sparse matrix
as_adjacency_matrix(g)
```

From igraph to adjacency matrix and back

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##           Alice Bob Cecil David Esmeralda
## Alice         .   .     .     .         .
## Bob           1   .     .     .         .
```

```
## Cecil         1   1     .     .         .
## David         1   1     .     .         .
## Esmeralda     1   .     .     .         .
```

```
# non-sparse matrix
as_adjacency_matrix(g, sparse = FALSE)
```

```
##           Alice Bob Cecil David Esmeralda
## Alice         0   0     0     0         0
## Bob           1   0     0     0         0
## Cecil         1   1     0     0         0
## David         1   1     0     0         0
## Esmeralda     1   0     0     0         0
```

```
# weighted matrix
as_adjacency_matrix(g, attr = "friendship")
```

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##           Alice Bob Cecil David Esmeralda
## Alice         .   .     .     .         .
## Bob           4   .     .     .         .
## Cecil         5   5     .     .         .
## David         2   1     .     .         .
## Esmeralda     1   .     .     .         .
```

```
# from adjacency matrix to igraph
# non-weighted graph
(A = as_adjacency_matrix(g))
```

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##           Alice Bob Cecil David Esmeralda
## Alice         .   .     .     .         .
## Bob           1   .     .     .         .
## Cecil         1   1     .     .         .
## David         1   1     .     .         .
## Esmeralda     1   .     .     .         .
```

```
graph_from_adjacency_matrix(A)
```

```
## IGRAPH 178e4c5 DN-- 5 6 --
## + attr: name (v/c)
## + edges from 178e4c5 (vertex names):
## [1] Bob      ->Alice Cecil    ->Alice David    ->Alice Esmeralda->Alice
## [5] Cecil    ->Bob   David    ->Bob
```

```
# undirected graph
graph_from_adjacency_matrix(A, mode = "undirected")
```

```
## IGRAPH 178ec13 UN-- 5 6 --
## + attr: name (v/c)
## + edges from 178ec13 (vertex names):
## [1] Alice--Bob       Alice--Cecil     Alice--David     Alice--Esmeralda
## [5] Bob  --Cecil     Bob  --David
```

```
(A = as_adjacency_matrix(g, attr = "friendship"))
```

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##           Alice Bob Cecil David Esmeralda
## Alice         .   .     .     .         .
```

```
## Bob           4   .     .     .         .
## Cecil         5   5     .     .         .
## David         2   1     .     .         .
## Esmeralda     1   .     .     .         .
```

```
# multi-graph
graph_from_adjacency_matrix(A)
```

```
## IGRAPH 178f840 DN-- 5 18 --
## + attr: name (v/c)
## + edges from 178f840 (vertex names):
##  [1] Bob      ->Alice Bob      ->Alice Bob      ->Alice Bob      ->Alice
##  [5] Cecil    ->Alice Cecil    ->Alice Cecil    ->Alice Cecil    ->Alice
##  [9] Cecil    ->Alice David    ->Alice David    ->Alice Esmeralda->Alice
## [13] Cecil    ->Bob   Cecil    ->Bob   Cecil    ->Bob   Cecil    ->Bob
## [17] Cecil    ->Bob   David    ->Bob
```

```
# weighted graph on weight attribute
graph_from_adjacency_matrix(A, weighted = TRUE)
```

```
## IGRAPH 178ff8e DNW- 5 6 --
## + attr: name (v/c), weight (e/n)
## + edges from 178ff8e (vertex names):
## [1] Bob      ->Alice Cecil    ->Alice David    ->Alice Esmeralda->Alice
## [5] Cecil    ->Bob   David    ->Bob
```

```
# weighted graph on friendship attribute
graph_from_adjacency_matrix(A, weighted = "friendship")
```

```
## IGRAPH 179046d DN-- 5 6 --
## + attr: name (v/c), friendship (e/n)
## + edges from 179046d (vertex names):
## [1] Bob      ->Alice Cecil    ->Alice David    ->Alice Esmeralda->Alice
## [5] Cecil    ->Bob   David    ->Bob
```

Read and write graph on disk

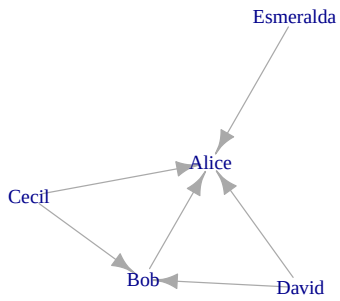
```
# write to graphml format
write_graph(g, file="Alice.xml", format="graphml")

# read from graphml format (notice the new id vertex attribute)
read_graph(file="Alice.xml", format="graphml")
```

```
## IGRAPH 1792683 DN-- 5 6 --
## + attr: name (v/c), age (v/n), gender (v/c), id (v/c), sameDept (e/l),
## | friendship (e/n), advice (e/n)
## + edges from 1792683 (vertex names):
## [1] Bob      ->Alice Cecil    ->Bob   Cecil    ->Alice David    ->Alice
## [5] David    ->Bob   Esmeralda->Alice
```

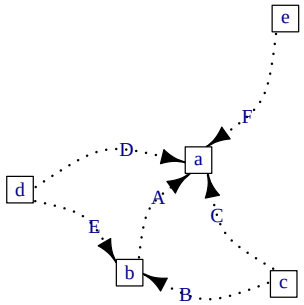
Visualize a graph

igraph have some **basic** visualization skills. Read the igraph plot manual for more.

```
# plot graph
plot(g, vertex.shape = "none")
```




```
# visualization parameters for graph, nodes and edges
plot(g,
     vertex.size = 20,
     vertex.color = "white",
     vertex.shape = "square",
     vertex.label = letters[1:vcount(g)],
     edge.width = 2,
     edge.color = "black",
     edge.lty = 3,
     edge.label = LETTERS[1:ecount(g)],
     edge.curved = TRUE)
```

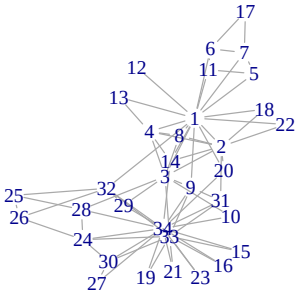


Layouts

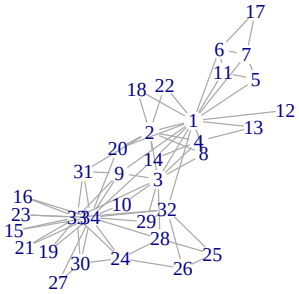
You can visualize the same graph with different **layouts**.

```
z = make_graph("Zachary")


# tries to choose an appropriate layout algorithm for the graph
```

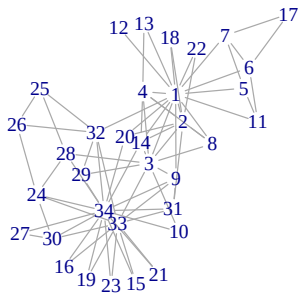
```
plot(z, layout = layout_nicely(z), vertex.shape = "none")
```



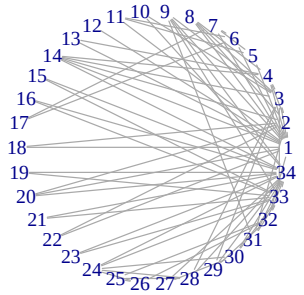

```
# Fruchterman and Reingold
plot(z, layout = layout_with_fr(z), vertex.shape = "none")
```



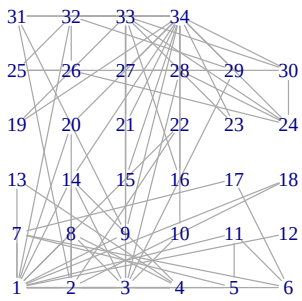

```
# Kamada-Kawai
plot(z, layout = layout_with_kk(z), vertex.shape = "none")
```



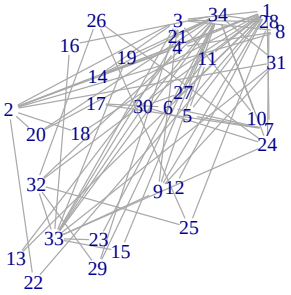

```
# In circle
plot(z, layout = layout_in_circle(z), vertex.shape = "none")
```




```
# On grid
plot(z, layout = layout_on_grid(z), vertex.shape = "none")
```



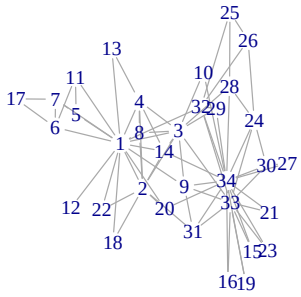

```
# Random
plot(z, layout = layout_randomly(z), vertex.shape = "none")
```




```
# save the layout of the graph in a variable
coords = layout_nicely(z)
coords[1:5, ]
```

```
##              [,1]     [,2]
## [1,] -0.797577504 2.018424
## [2,]  0.156512906 1.138030
## [3,]  1.689728340 2.262961
## [4,]  0.007833575 2.845172
## [5,] -2.695939555 2.592323
```

```
plot(z, layout = coords, vertex.shape = "none")
```



3. Tidy Network analysis and visualization: tidygraph and ggraph

Tidy network data?

- there's a discrepancy between network data and the tidy data idea, in that network data cannot in any meaningful way be encoded as a single tidy data frame
- on the other hand, both node and edge data by itself fits very well within the tidy concept as each node and edge is, in a sense, a single observation
- thus, a close approximation of tidyness for network data is two tidy data frames, one describing the node data and one describing the edge data

tidygraph

- tidygraph is an entry into the tidyverse that provides a tidy framework for network (graph) data
- tidygraph provides an approach to manipulate node and edge data frames using the interface defined in the **dplyr** package
- moreover it provides tidy interfaces to a lot of common graph algorithms, including **igraph** network analysis toolkit
- it is developed by Thomas Lin Pedersen

ggraph

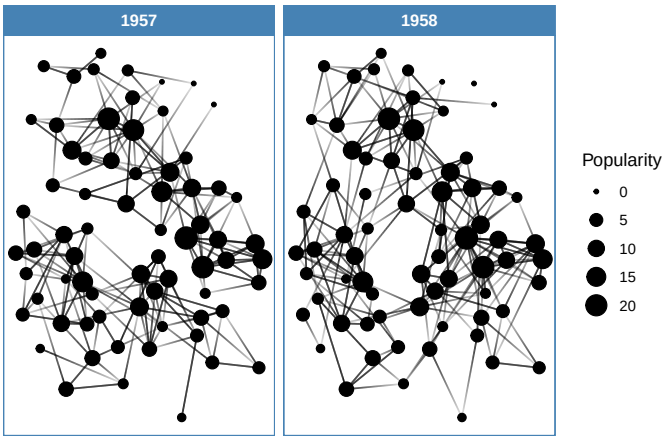
- ggraph is an extension of ggplot2 that implements a visualization grammar for network data
- it provides a huge variety of geoms for drawing nodes and edges, along with an assortment of layouts making it possible to produce a very wide range of network visualization types
- while **tidygraph** provides a manipulation and analysis grammar for network data (like **dplyr** for tabular data), **ggraph** offers a visualization grammar (like **ggplot** for tabular data)
- it is developed by Thomas Lin Pedersen

A full example: dplyr, tidygraph and ggraph

```
# setting theme_graph
set_graph_style()

# a graph of highschool friendships
head(highschool)
```

```
##   from to year
## 1    1 14 1957
## 2    1 15 1957
## 3    1 21 1957
## 4    1 54 1957
## 5    1 55 1957
## 6    2 21 1957
```

```
# create the graph and add popularity
graph <- as_tbl_graph(highschool) %>%
    mutate(Popularity = centrality_degree(mode = "in"))

# plot using ggraph
ggraph(graph, layout = "kk") +
    geom_edge_link(aes(alpha = stat(index)), show.legend = FALSE) +
    geom_node_point(aes(size = Popularity)) +
    facet_edges(~year) +
    theme_graph(foreground = "steelblue", fg_text_colour = "white", base_family="sans")
```



Read the graph with tidygraph

Let's read a dolphin network:

1. a set of nodes representing dolphins (dolphin_nodes.csv)

2. a set of edges representing ties among dolphins (dolphin_edges.csv)

Package tidygraph represents the graph as a pair of data frames:

- a data frame for nodes containing information about the nodes in the graph
- a data frame for edges containing information about the edges in the graph. The terminal nodes of each edge must either be encoded in a `to` and `from` column, or in the two first columns, as integers. These integers refer to nodes index.

```
library(readr)

nodes = read_csv("dolphin_nodes.csv")
```

```
##
```

```
## -- Column specification --------------------------------------------------------
## cols(
##   name = col_character(),
##   sex = col_character()
## )
```

```
edges = read_csv("dolphin_edges.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   x = col_double(),
##   y = col_double()
## )
```

```
nodes
```

```
## # A tibble: 62 x 2
##    name       sex
##    <chr>      <chr>
##  1 Beak       M
##  2 Beescratch M
##  3 Bumper     M
##  4 CCL        F
##  5 Cross      M
##  6 DN16       F
##  7 DN21       M
##  8 DN63       M
##  9 Double     F
## 10 Feather    M
## # ... with 52 more rows
```

```
edges
```

```
## # A tibble: 159 x 2
##        x     y
##    <dbl> <dbl>
##  1     4     9
##  2     6    10
##  3     7    10
##  4     1    11
##  5     3    11
##  6     6    14
##  7     7    14
##  8    10    14
##  9     1    15
## 10     4    15
## # ... with 149 more rows
```

```
# add edge type
edges =
  edges %>%
  mutate(type = sample(c("love", "friendship"),
                       nrow(edges),
                       replace = TRUE) )

# make a tidy graph
```

```
dolphin = tbl_graph(nodes = nodes, edges = edges, directed = FALSE)
dolphin
```

```
## # A tbl_graph: 62 nodes and 159 edges
## #
## # An undirected simple graph with 1 component
## #
## # Node Data: 62 x 2 (active)
##   name       sex
##   <chr>      <chr>
## 1 Beak       M
## 2 Beescratch M
## 3 Bumper     M
## 4 CCL        F
## 5 Cross      M
## 6 DN16       F
## # ... with 56 more rows
## #
## # Edge Data: 159 x 3
##    from    to type
##   <int> <int> <chr>
## 1     4     9 friendship
## 2     6    10 love
## 3     7    10 love
## # ... with 156 more rows
```

```
# extract node and edge data frames from the graph
as.list(dolphin)
```

```
## $nodes
## # A tibble: 62 x 2
##    name       sex
##    <chr>      <chr>
##  1 Beak       M
##  2 Beescratch M
##  3 Bumper     M
##  4 CCL        F
##  5 Cross      M
##  6 DN16       F
##  7 DN21       M
##  8 DN63       M
##  9 Double     F
## 10 Feather    M
## # ... with 52 more rows
##
## $edges
## # A tibble: 159 x 3
##     from    to type
##    <int> <int> <chr>
##  1     4     9 friendship
##  2     6    10 love
##  3     7    10 love
##  4     1    11 love
##  5     3    11 love
```

```
##  6     6    14 love
##  7     7    14 love
##  8    10    14 love
##  9     1    15 friendship
## 10     4    15 love
## # ... with 149 more rows
```

```
# extract node data frame from the graph
as.list(dolphin)$nodes
```

```
## # A tibble: 62 x 2
##    name       sex
##    <chr>      <chr>
##  1 Beak       M
##  2 Beescratch M
##  3 Bumper     M
##  4 CCL        F
##  5 Cross      M
##  6 DN16       F
##  7 DN21       M
##  8 DN63       M
##  9 Double     F
## 10 Feather    M
## # ... with 52 more rows
```

```
# extract edge data frame from the graph
as.list(dolphin)$edges
```

```
## # A tibble: 159 x 3
##     from    to type
##    <int> <int> <chr>
##  1     4     9 friendship
##  2     6    10 love
##  3     7    10 love
##  4     1    11 love
##  5     3    11 love
##  6     6    14 love
##  7     7    14 love
##  8    10    14 love
##  9     1    15 friendship
## 10     4    15 love
## # ... with 149 more rows
```

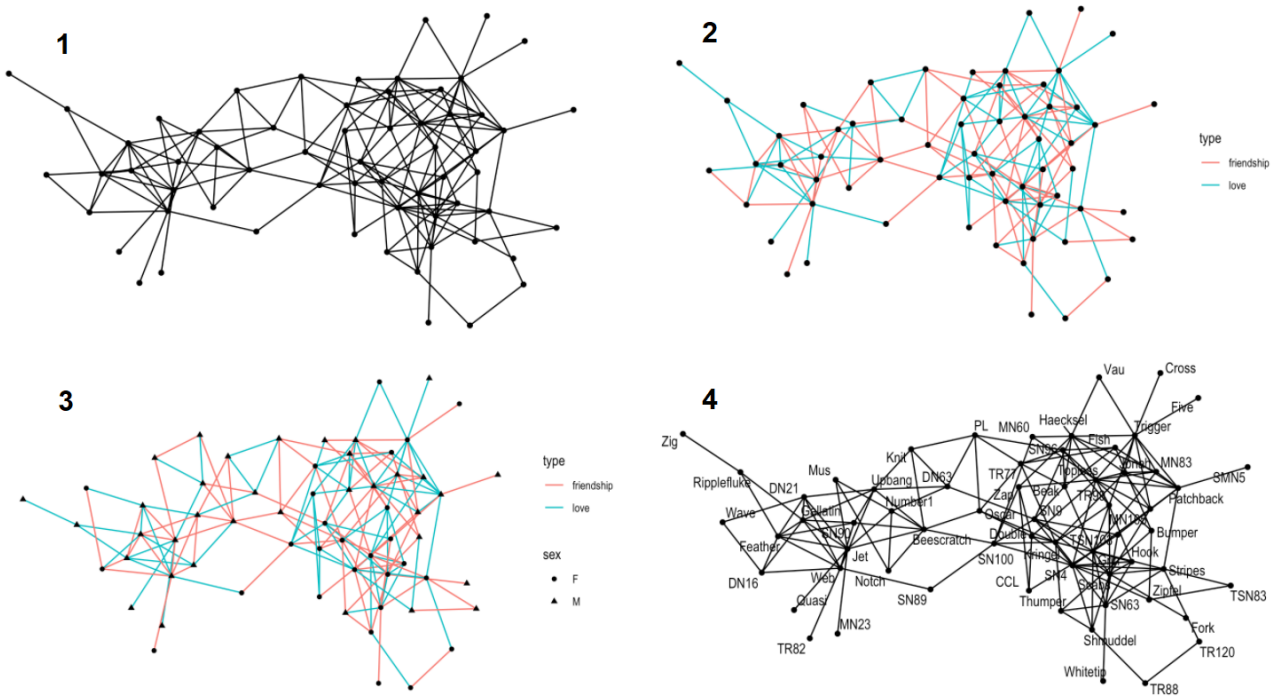
ggraph components

ggraph builds upon three core concepts that are quite easy to understand:

- the **layout** defines how nodes are placed on the plot. ggraph has access to all layout functions available in igraph and much more
- the **nodes** are the connected entities in the graph structure. These can be plotted using the `geom_node_*()` family of geoms
- the **edges** are the connections between the entities in the graph structure. These can be visualized using the `geom_edge_*()` family of geoms

ggraph basics

```
# setting theme_graph
set_graph_style()

# basic plot
ggraph(dolphin) +
  geom_edge_link() +
  geom_node_point() +
  theme_graph(base_family="sans")

# plot edge type
ggraph(dolphin) +
  geom_edge_link(aes(color = type)) +
  geom_node_point() +
  theme_graph(base_family="sans")

# plot node sex
ggraph(dolphin) +
  geom_edge_link(aes(color = type)) +
  geom_node_point(aes(shape = sex)) +
  theme_graph(base_family="sans")

# plot node name
ggraph(dolphin) +
  geom_edge_link() +
  geom_node_point() +
  geom_node_text(aes(label = name), repel=TRUE) +
  theme_graph(base_family="sans")
```

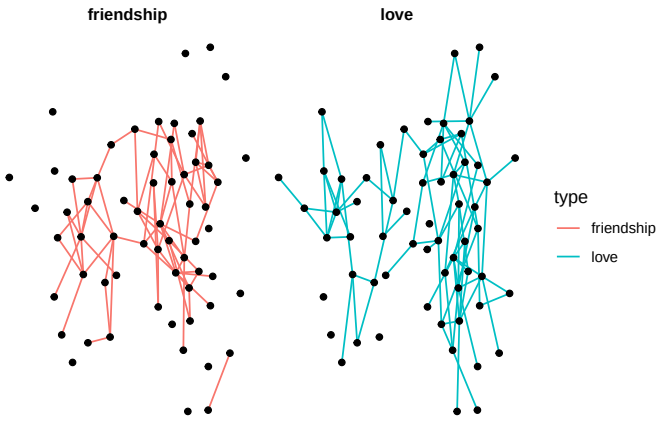


Faceting

Faceting allows to create sub-plots according to the values of a qualitative attribute on nodes or edges.

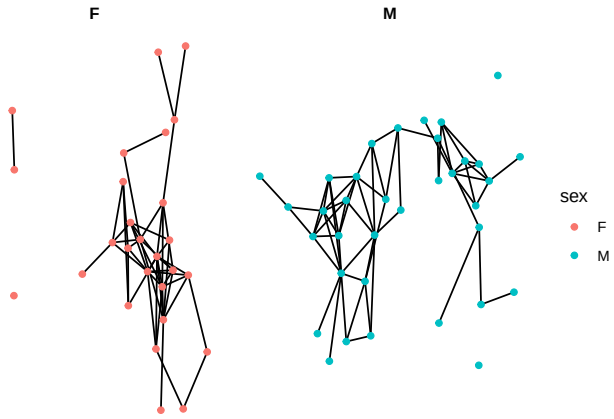
```
# setting theme_graph
set_graph_style()


# facet edges by type
ggraph(dolphin) +
  geom_edge_link(aes(color = type)) +
  geom_node_point() +
  facet_edges(~type) +
  theme_graph(base_family="sans")
```

```
## Using `stress` as default layout
```



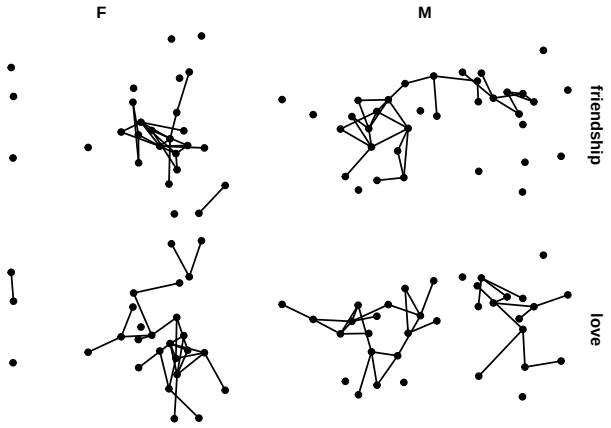

```
# facet nodes by sex
ggraph(dolphin) +
  geom_edge_link() +
  geom_node_point(aes(color = sex)) +
  facet_nodes(~sex) +
  theme_graph(base_family="sans")
```

```
## Using `stress` as default layout
```




```
# facet both nodes and edges
ggraph(dolphin) +
  geom_edge_link() +
  geom_node_point() +
  facet_graph(type~sex) +
  th_foreground(border = TRUE) +
  theme_graph(base_family="sans")
```

```
## Using `stress` as default layout
```



Directed graphs

```
# setting theme_graph
set_graph_style()

# direcred graphs
package = tibble(
  name = c("igraph", "ggraph", "dplyr", "ggplot", "tidygraph")
)

tie = tibble(
  from = c("igraph", "ggplot", "igraph", "dplyr", "ggraph"),
  to =   c("tidygraph", "ggraph", "tidygraph", "tidygraph", "tidygraph")
)

tidy = tbl_graph(nodes = package, edges = tie, directed = TRUE)


# use arrows for directions
ggraph(tidy, layout = "graphopt") +
    geom_edge_link(aes(start_cap = label_rect(node1.name),
                       end_cap = label_rect(node2.name)),
                   arrow = arrow(type = "closed",
                                 length = unit(3, "mm"))) +
    geom_node_text(aes(label = name))

# use edge alpha to indicate direction,
# direction is from lighter to darker node
ggraph(tidy, layout = 'graphopt') +
    geom_edge_link(aes(start_cap = label_rect(node1.name),
                       end_cap = label_rect(node2.name),
                       alpha = stat(index)),
                   show.legend = FALSE) +
    geom_node_text(aes(label = name))
```

Hierarchical layouts

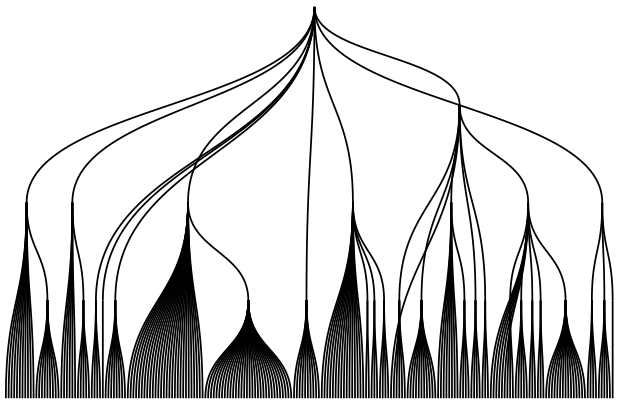
```
# setting theme_graph
set_graph_style()

# This dataset contains the graph that describes the class
# hierarchy for the Flare visualization library.
head(flare$vertices)
```

```
##                                          name size             shortName
## 1 flare.analytics.cluster.AgglomerativeCluster 3938  AgglomerativeCluster
## 2   flare.analytics.cluster.CommunityStructure 3812    CommunityStructure
## 3  flare.analytics.cluster.HierarchicalCluster 6714   HierarchicalCluster
## 4            flare.analytics.cluster.MergeEdge  743             MergeEdge
## 5  flare.analytics.graph.BetweennessCentrality 3534 BetweennessCentrality
## 6          flare.analytics.graph.LinkDistance 5731          LinkDistance
```

```
head(flare$edges)
```

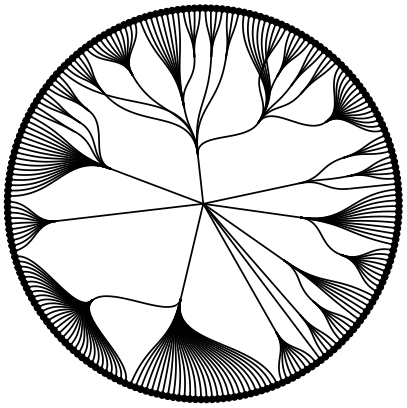
```
##                      from                                          to
## 1 flare.analytics.cluster flare.analytics.cluster.AgglomerativeCluster
## 2 flare.analytics.cluster   flare.analytics.cluster.CommunityStructure
## 3 flare.analytics.cluster  flare.analytics.cluster.HierarchicalCluster
## 4 flare.analytics.cluster            flare.analytics.cluster.MergeEdge
## 5   flare.analytics.graph  flare.analytics.graph.BetweennessCentrality
## 6   flare.analytics.graph           flare.analytics.graph.LinkDistance
```

```
# flare class hierarchy
graph = tbl_graph(edges = flare$edges, nodes = flare$vertices)

# dendrogram
ggraph(graph, layout = "dendrogram") +
  geom_edge_diagonal() +
  theme_graph(base_family="sans")
```

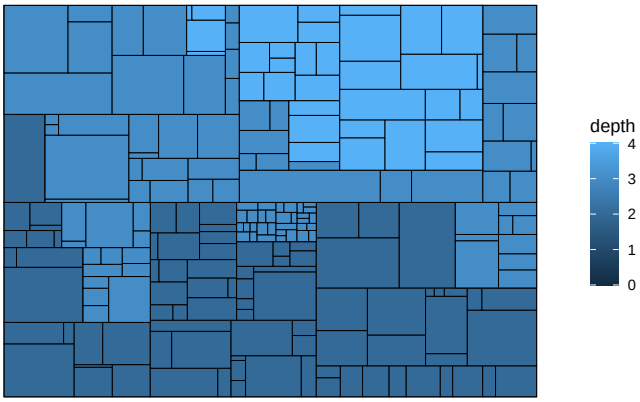


```
# circular dendrogram
ggraph(graph, layout = "dendrogram", circular = TRUE) +
  geom_edge_diagonal() +
  geom_node_point(aes(filter = leaf)) +
```

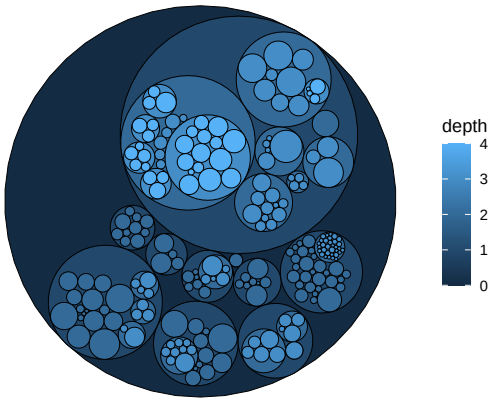
```
  coord_fixed() +
  theme_graph(base_family="sans")
```



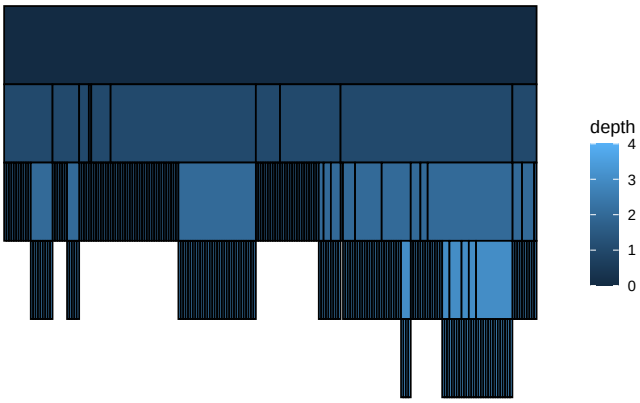
```
# rectangular tree map
ggraph(graph, layout = "treemap", weight = size) +
  geom_node_tile(aes(fill = depth), size = 0.25) +
  theme_graph(base_family="sans")
```



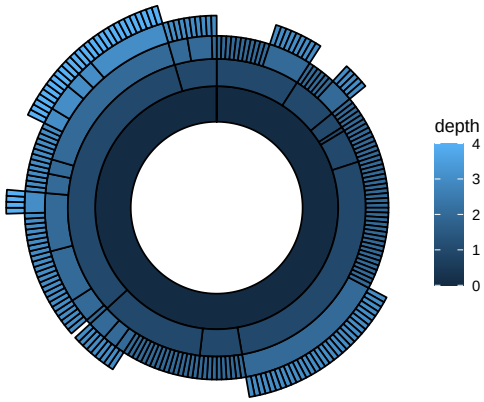

```
# circular tree map
ggraph(graph, layout = "circlepack", weight = size) +
  geom_node_circle(aes(fill = depth), size = 0.25, n = 50) +
  coord_fixed() +
  theme_graph(base_family="sans")
```




```
# icicle
ggraph(graph, layout = "partition") +
  geom_node_tile(aes(y = -y, fill = depth)) +
  theme_graph(base_family="sans")
```




```
# sunburst (circular icicle)
ggraph(graph, layout = "partition", circular = TRUE) +
  geom_node_arc_bar(aes(fill = depth)) +
  coord_fixed() +
  theme_graph(base_family="sans")
```



Network analysis with tidygraph

- the data frame graph representation can be easily augmented with metrics computed on the graph
- before computing a metric on nodes or edges use the `activate()` function to activate either node or edge data frames
- use dplyr verbs filter, arrange and mutate to manipulate the graph

```
dolphin =
  dolphin %>%
  activate(nodes) %>%
  mutate(degree = centrality_degree()) %>%
  filter(degree > 0) %>%
  arrange(-degree) %>%
  activate(edges) %>%
  mutate(betweenness = centrality_edge_betweenness(),
         # .N() gets the nodes data from edge you're accessing
         homo = (.N()$sex[from] == .N()$sex[to])) %>%
  arrange(-betweenness)

dolphin
```

```
## # A tbl_graph: 62 nodes and 159 edges
## #
## # An undirected simple graph with 1 component
## #
## # Edge Data: 159 x 5 (active)
##    from    to type       betweenness homo
##   <int> <int> <chr>            <dbl> <lgl>
## 1    10    17 friendship        283. FALSE
## 2    13    29 friendship        219. FALSE
## 3     6    10 friendship        184. TRUE
## 4     2    17 friendship        181. TRUE
## 5    17    48 love              173. TRUE
## 6     9    48 friendship        146. FALSE
## # ... with 153 more rows
## #
## # Node Data: 62 x 3
```

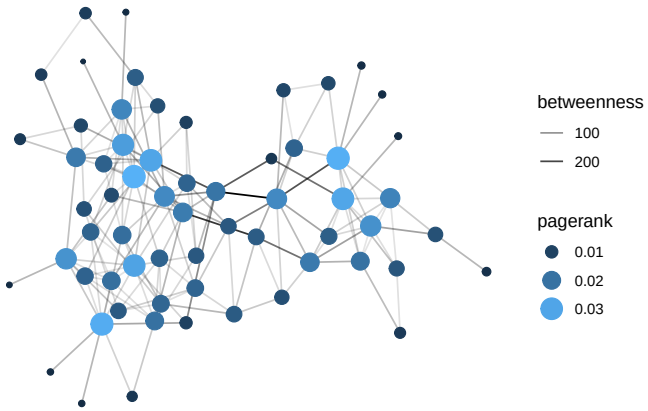
```
##   name    sex   degree
##   <chr>   <chr>  <dbl>
## 1 Grin    F         12
## 2 SN4     F         11
## 3 Topless M         11
## # ... with 59 more rows
```

Analyse and visualize network: centrality

Packages tidygraph and ggraph can be pipelined to perform analysis and visualization tasks in one go.

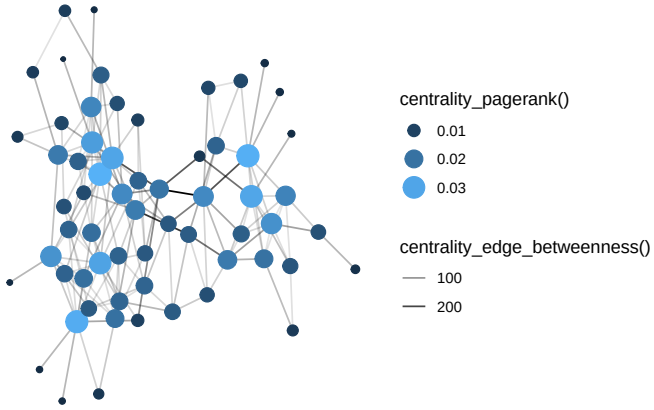
```
# setting theme_graph
set_graph_style()


dolphin %>%
  activate(nodes) %>%
  mutate(pagerank = centrality_pagerank()) %>%
  activate(edges) %>%
  mutate(betweenness = centrality_edge_betweenness()) %>%
  ggraph() +
  geom_edge_link(aes(alpha = betweenness)) +
  geom_node_point(aes(size = pagerank, colour = pagerank)) +
  # discrete colour legend
  scale_color_gradient(guide = "legend") +
  theme_graph(base_family="sans")
```

```
## Using `stress` as default layout
```




```
# or even less typing
ggraph(dolphin) +
  geom_edge_link(aes(alpha = centrality_edge_betweenness())) +
  geom_node_point(aes(size = centrality_pagerank(),
                      colour = centrality_pagerank())) +
  scale_color_gradient(guide = "legend") +
  theme_graph(base_family="sans")
```

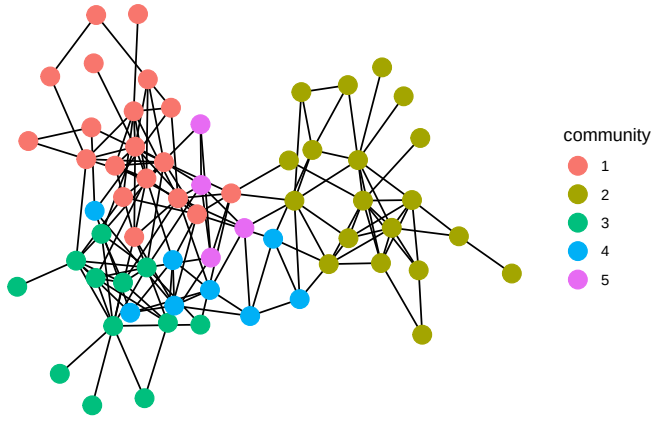
```
## Using `stress` as default layout
```



Analyse and visualize network: communities

```
# setting theme_graph
set_graph_style()


# visualize communities of nodes
dolphin %>%
  activate(nodes) %>%
  mutate(community = as.factor(group_louvain())) %>%
  ggraph() +
  geom_edge_link() +
  geom_node_point(aes(colour = community), size = 5) +
  theme_graph(base_family="sans")
```

```
## Using `stress` as default layout
```



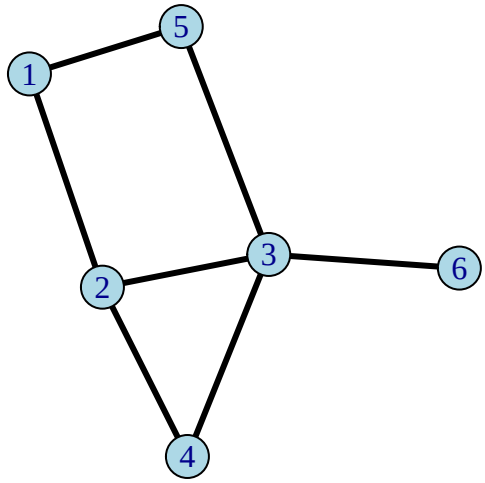
4. Graph basics

- a **network** is a collection of vertices joined by edges
- vertices and edges are also called nodes and links in computer science and actors and ties in sociology
- in mathematics, a network is called **graph** and it is typically represented as a square matrix
- given a graph G with n nodes numbered from 1 to n, the **adjacency matrix** $A = (a_{i,j})$ of G is a square $n \times n$ matrix such that $a_{i,j} = 1$ if there exists an edge joining nodes i and j, and $a_{i,j} = 0$ otherwise

Undirected graphs

- a graph is **undirected** if edges have no direction: if there is an edge from i to j , then there is also an edge from j to i
- this means that the adjacency matrix of an undirected graph is **symmetric**: $a_{i,j} = a_{j,i}$ for every pair i, j, or $A = A^T$, where A^T is the transpose of A
- if there is and edge between i and j, then i and j are said to be **neighbors**
- the neighbors of node i are the 1s of row (or column) i of A

```
library(igraph)
edges = c(1,2, 1,5, 2,3, 2,4, 3,4, 3,5, 3,6)
ug = graph(edges, directed=FALSE)
coords = layout_with_fr(ug)
plot(ug, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black")
```




```
(A = as_adjacency_matrix(ug, sparse=FALSE))
```

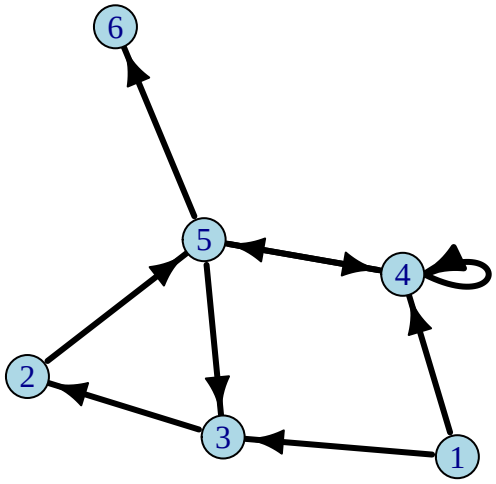
```
##      [,1] [,2] [,3] [,4] [,5] [,6]
```

```
## [1,]    0    1    0    0    1    0
## [2,]    1    0    1    1    0    0
## [3,]    0    1    0    1    1    1
## [4,]    0    1    1    0    0    0
## [5,]    1    0    1    0    0    0
## [6,]    0    0    1    0    0    0
```

Directed graphs

- a graph is **directed** if edges have a direction: if there is an edge from i to j, then there might be or not the inverse edge
- this means that the adjacency matrix of an directed graph is not necessarily symmetric
- if there is and edge from i to j, then i is a **predecessor** of j and j is a **successor** of i
- the predecessors of node i are the 1s on column i of A and the successors of node i are the 1s on row i of A
- **self-loops** or self-edges are edges (i, i) from a node i to itself: they correspond to diagonal entries in the adjacency matrix

```
edges = c(1,3, 1,4, 2,5, 3,2, 4,4, 4,5, 5,3, 5,6, 5,4)
dg = graph(edges, directed=TRUE)
coords = layout_with_fr(dg)
plot(dg, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black")
```




```
(A = as_adjacency_matrix(dg, sparse=FALSE))
```

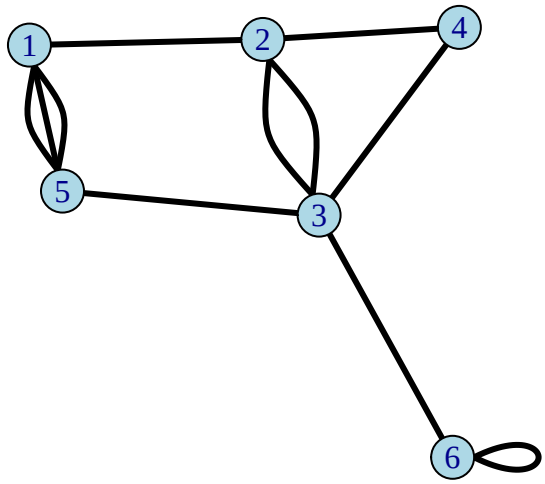
```
##      [,1] [,2] [,3] [,4] [,5] [,6]
```

```
## [1,]    0    0    1    1    0    0
## [2,]    0    0    0    0    1    0
## [3,]    0    1    0    0    0    0
## [4,]    0    0    0    1    1    0
## [5,]    0    0    1    1    0    1
## [6,]    0    0    0    0    0    0
```

Simple and multigraphs

- In some cases there can be more than one edge between the same pair of vertices; we refer to those edges collectively as a **multiedge**
- a network that has no multiedges is called a **simple graph**, otherwise a **multigraph**

```
edges = c(1,2, 1,5, 1,5, 1,5, 2,3, 2,3, 2,4, 3,4, 3,5, 3,6, 6,6)
mug = graph(edges, directed=FALSE)
coords = layout_with_fr(mug)
plot(mug, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black")
```




```
(A = as_adjacency_matrix(mug, sparse=FALSE))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    1    0    0    3    0
## [2,]    1    0    2    1    0    0
## [3,]    0    2    0    1    1    1
## [4,]    0    1    1    0    0    0
## [5,]    3    0    1    0    0    0
```

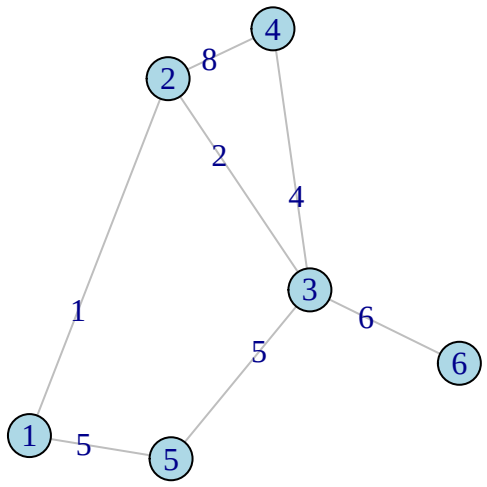
```
## [6,]    0    0    1    0    0    1
```

Unweighted and weighted graphs

In some situations it is useful to represent edges as having a strength, weight, or value to them:

1. in the Internet edges might have weights representing the amount of data flowing along them or their bandwidth
2. in a food web predator-prey interactions might have weights measuring total energy flow between prey and predator
3. in a social network connections might have weights representing the sign and intensity of the relationship: positive weights denote friendship and negative ones represent animosity

```
edges = c(1,2, 1,5, 2,3, 2,4, 3,4, 3,5, 3,6)
wg = graph(edges, directed=FALSE)
E(wg)$weight = c(1, 5, 2, 8, 4, 5, 6)
coords = layout_with_fr(wg)
plot(wg, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 1, edge.color = "grey", edge.label = E(wg)$weight)
```




```
(A = as_adjacency_matrix(wg, sparse=FALSE, attr = "weight"))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    1    0    0    5    0
## [2,]    1    0    2    8    0    0
## [3,]    0    2    0    4    5    6
## [4,]    0    8    4    0    0    0
```

```
## [5,]    5    0    5    0    0    0
## [6,]    0    0    6    0    0    0
```

Example:

1. a **path** in a network is any sequence of nodes such that every consecutive pair of vertices in the sequence is connected by an edge in the network
2. a simple path is a path with no repetitions of nodes and edges allowed
3. the **length** of a path is the number of edges of the path
4. write the number of (not necessarily simple) paths of a given length on a simple graph in terms of its adjacency matrix
5. write some code to test it.

Notice that:

$$N_{i,j}^{(1)} = [A]_{i,j}$$

is the number of paths of length 1 (edges) from i to j. Moreover, the product $a_{i,k}a_{k,j}$ is 1 if and only if there is a path of length 2 from i to j (that goes through k). Hence, the number of paths of length 2 from i to j is:

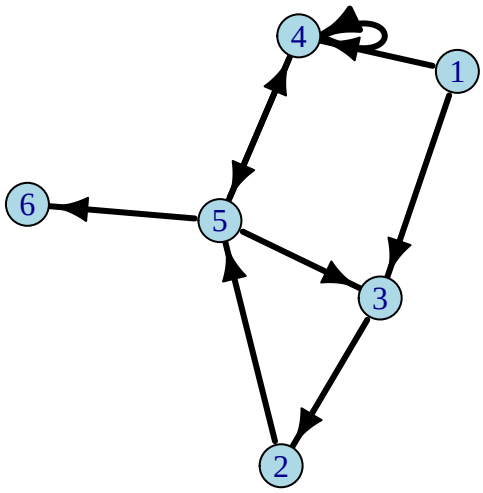
$$N_{i,j}^{(2)} = \sum_{k=1}^{n} a_{i,k}a_{k,j} = [A^2]_{i,j}$$

where $[A^2]_{i,j}$ is the (i,j) entry of matrix $A^2 = AA$. More generally, the number of paths of length r from i to j is:

$$N_{i,j}^{(r)} = [A^r]_{i,j}$$

Notice that $[A^r]_{i,i}$ is the number of (not necessarily simple) cycles of length r that start and end at the same node i.

```
edges = c(1,3, 1,4, 2,5, 3,2, 4,4, 4,5, 5,3, 5,6, 5,4)
dg = graph(edges, directed=TRUE)
coords = layout_with_fr(dg)
plot(dg, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black")
```




```
A = as_adjacency_matrix(dg, sparse=FALSE)

# length 1
A
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    1    1    0    0
## [2,]    0    0    0    0    1    0
## [3,]    0    1    0    0    0    0
## [4,]    0    0    0    1    1    0
## [5,]    0    0    1    1    0    1
## [6,]    0    0    0    0    0    0
```

```
# length 2
(A2 = A %*% A)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    1    0    1    1    0
## [2,]    0    0    1    1    0    1
## [3,]    0    0    0    0    1    0
## [4,]    0    0    1    2    1    1
## [5,]    0    1    0    1    1    0
## [6,]    0    0    0    0    0    0
```

```
# length 3
(A3 = A %*% A2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
```

```
## [1,]    0    0    1    2    2    1
## [2,]    0    1    0    1    1    0
## [3,]    0    0    1    1    0    1
## [4,]    0    1    1    3    2    1
## [5,]    0    0    1    2    2    1
## [6,]    0    0    0    0    0    0
```

```
# length 4
(A4 = A %*% A3)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    1    2    4    2    2
## [2,]    0    0    1    2    2    1
## [3,]    0    1    0    1    1    0
## [4,]    0    1    2    5    4    2
## [5,]    0    1    2    4    2    2
## [6,]    0    0    0    0    0    0
```

Example: Given and adjacency matrix A for a graph, matrices AA^T and A^TA are called **projection matrices**. Give a meaning to these projections on a directed graph. Write some code to check it.

In directed networks, we have that:

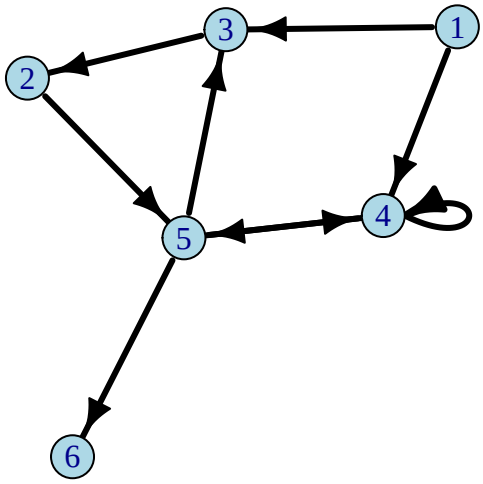
$$[AA^T]_{i,j} = \sum_{k=1}^{n} a_{i,k}a_{j,k}$$

is the number of **common successors** of i and j with $[AA^T]_{i,i}$ is the number of successors of i and

$$[A^TA]_{i,j} = \sum_{k=1}^{n} a_{k,i}a_{k,j}$$

is the number of **common predecessors** of i and j with $[A^TA]_{i,i}$ is the number of predecessors of i.

```
# projections of a directed graph
edges = c(1,3, 1,4, 2,5, 3,2, 4,4, 4,5, 5,3, 5,6, 5,4)
dg = graph(edges, directed=TRUE)

coords1 = layout_with_fr(dg)
plot(dg, layout=coords1,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black")
```




```
A = as_adjacency_matrix(dg, sparse=FALSE)
(P = A %*% t(A))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2    0    0    1    2    0
## [2,]    0    1    0    1    0    0
## [3,]    0    0    1    0    0    0
## [4,]    1    1    0    2    1    0
## [5,]    2    0    0    1    3    0
## [6,]    0    0    0    0    0    0
```

```
(Q = t(A) %*% A)
```

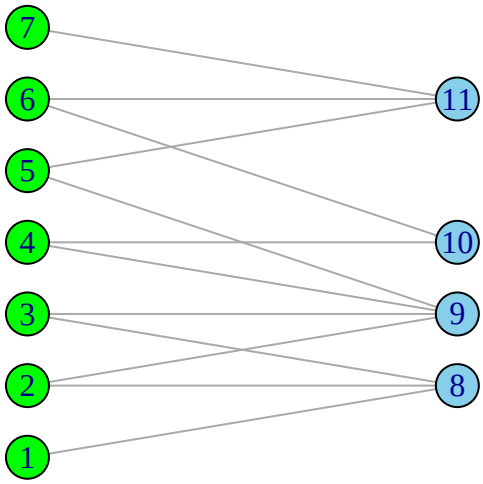
```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    0    0    0    0
## [2,]    0    1    0    0    0    0
## [3,]    0    0    2    2    0    1
## [4,]    0    0    2    3    1    1
## [5,]    0    0    0    1    2    0
## [6,]    0    0    1    1    0    1
```

Bipartite graphs

- a **bipartite graph** is a graph where there are two kinds of vertices, and edges run from nodes of different types only
- any network in which the vertices are connected together by common membership of groups of some kind can be represented in this way

- in sociology such networks are called **affiliation networks**: for instance, scientists coauthoring papers or film actors appearing together in films
- a bipartite network is typically represented with an **incidence matrix**: if n is the number of actors in the network and g is the number of groups, then the incidence matrix B is a $g \times n$ matrix having elements $B_{i,j} = 1$ if group i contains participant j and $B_{i,j} = 0$ otherwise

```
types = c(rep(TRUE,7), rep(FALSE,4))
edges = c(8,1, 8,2, 8,3, 9,2, 9,3, 9,4, 9,5, 10,4, 10,6, 11,5, 11,6, 11,7)
bg = make_bipartite_graph(types, edges, directed=FALSE)
lay = layout.bipartite(bg)
plot(bg, layout=lay[,2:1],
     vertex.color=c("skyblue","green")[V(bg)$type + 1],
     vertex.size = 20)
```




```
(B = as_incidence_matrix(bg))
```

```
##    1 2 3 4 5 6 7
## 8  1 1 1 0 0 0 0
## 9  0 1 1 1 1 0 0
## 10 0 0 0 1 0 1 0
## 11 0 0 0 0 1 1 1
```

Example: A bipartite graph is a special case of undirected graph, hence it can be represented with a symmetric adjacency matrix as well.

- write the adjacency matrix A of a bipartite graph in terms of the incidence matrix B of the graph
- test the solution with some code

$$A = \begin{pmatrix} 0 & B^T \\ B & 0 \end{pmatrix}$$

where 0 are square matrices of size $n \times n$ (upper) and $g \times g$ (lower).

```
# incidence matrix
(B = as_incidence_matrix(bg))
```

```
##    1 2 3 4 5 6 7
## 8  1 1 1 0 0 0 0
## 9  0 1 1 1 1 0 0
## 10 0 0 0 1 0 1 0
## 11 0 0 0 0 1 1 1
```

```
# adjacency matrix
(A = as_adjacency_matrix(bg, sparse = FALSE))
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
##  [1,]    0    0    0    0    0    0    0    1    0     0     0
##  [2,]    0    0    0    0    0    0    0    1    1     0     0
##  [3,]    0    0    0    0    0    0    0    1    1     0     0
##  [4,]    0    0    0    0    0    0    0    0    1     1     0
##  [5,]    0    0    0    0    0    0    0    0    1     0     1
##  [6,]    0    0    0    0    0    0    0    0    0     1     1
##  [7,]    0    0    0    0    0    0    0    0    0     0     1
##  [8,]    1    1    1    0    0    0    0    0    0     0     0
##  [9,]    0    1    1    1    1    0    0    0    0     0     0
## [10,]    0    0    0    1    0    1    0    0    0     0     0
## [11,]    0    0    0    0    1    1    1    0    0     0     0
```

Example: A **projection** of a bipartite graph is a weighted undirected graph such that:

- the nodes are the vertices of the bipartite graph of one type (hence, there are two projections)
- there is an edge between two nodes of the same type if they have a neighbor in common in the bipartite graph
- the weight of the edge is the number of common neighbors

Write the adjacency matrices of the two projections of a bipartite graph in terms of its incidence matrix B and code one example. What is the value on the diagonal of the projections?

If B is the incidence matrix of a bipartite graph with n actors and g groups, then the one-mode projection graph on actors is the $n \times n$ symmetric matrix

$$B^T B$$

and the one-mode projection graph on groups is the $g \times g$ symmetric matrix

$$BB^T$$

```
# incidence matrix
(B = as_incidence_matrix(bg))
```

```
##    1 2 3 4 5 6 7
## 8  1 1 1 0 0 0 0
## 9  0 1 1 1 1 0 0
## 10 0 0 0 1 0 1 0
## 11 0 0 0 0 1 1 1
```

```r
# first projection
t(B) %*% B
```

```
##   1 2 3 4 5 6 7
## 1 1 1 1 0 0 0 0
## 2 1 2 2 1 1 0 0
## 3 1 2 2 1 1 0 0
## 4 0 1 1 2 1 1 0
## 5 0 1 1 1 2 1 1
## 6 0 0 0 1 1 2 1
## 7 0 0 0 0 1 1 1
```

```r
# second projection
B %*% t(B)
```

```
##    8 9 10 11
## 8  3 2  0  0
## 9  2 4  1  1
## 10 0 1  2  1
## 11 0 1  1  3
```

Regular graphs

A **regular** graph is defined as follows:

1. An unweighted undirected graph is regular if there is an integer $r > 0$ such that all nodes have degree equal to r.
2. An unweighted directed graph is regular if there is an integer $r > 0$ such that all nodes have out-degree and in-degree equal to r.

Weighted degree

- in a weighted undirected graph, the **weighted degree** of a node is the sum of weights of edges incident in the node
- in a weighted directed graph, we distinguish between weighted out-degree (the sum of weights of edges leaving the node) and weighted in-degree (the sum of weights of edges entering the node)

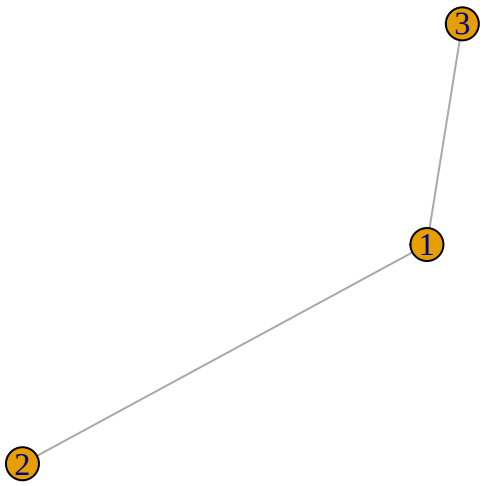
Regularizable graphs

A **regularizable** graph is defined as follows:

1. An unweighted undirected graph is regularizable if the edges of the graph can be weighted with positive integers and in the resulting weighted graph all nodes have weighted degree equal to some $r > 0$.
2. An unweighted directed graph is regularizable if the edges of the graph can be weighted with positive integers and in the resulting weighted graph all nodes have weighted out-degree and weighted in-degree equal to some $r > 0$.

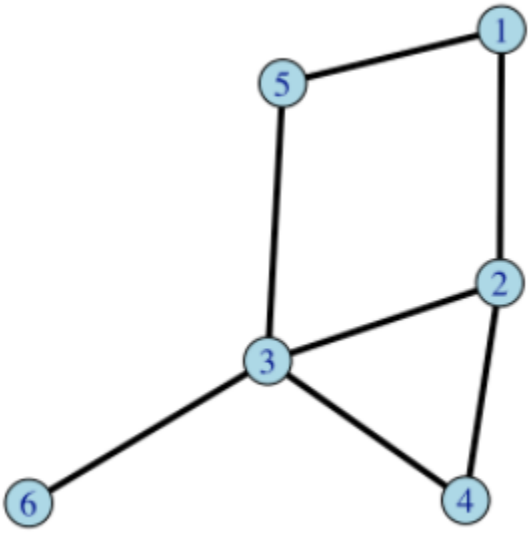
- a regular graph is regularizable but there are regularizable graphs that are not regular
- not all graphs are regularizable: it is easy to see that a star graph with at least 3 nodes is not regularizable

```r
library(igraph)
plot(make_star(3, mode = "undirected"))
```



Example:

Consider the following two undirected graphs. The left graph is not regularizable: can you tell why? The graph on the right is regularizable: can you find weights for the edges to prove it?



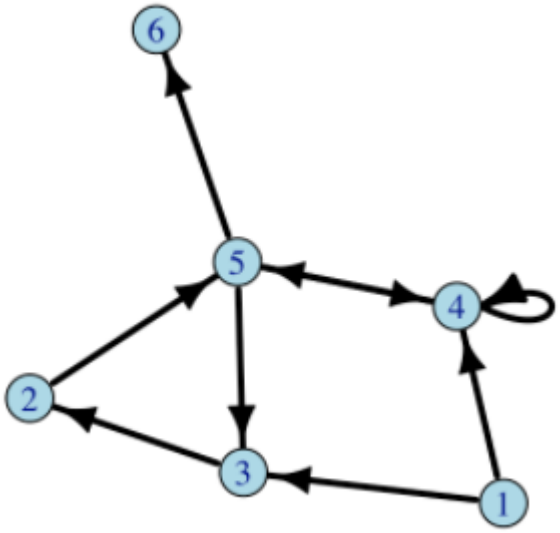
- on the left, suppose you put a weight of $\alpha > 0$ on edge $(1, 2)$ and a weight of $\beta > 0$ on edge $(1, 3)$. Then

node 2 has degree α and node 1 has degree $\alpha + \beta > \alpha$.

- the graph on the right is regularizable: if we label the outer edges with 3 and the inner edges with 2 we have a positive regularizability solution with regularization degree 8.

Example:

Consider the following two directed graphs. Find regularizable solution for graph on the right. Is graph on the left regularizable?



- a regularizable solution for the right graph labels all edges with 1 and edge from 4 to 1 with 2
- there is no regularizable solution for the graph on the left since nodes 2 and 4 (or 1) will necessarily have different degrees.

5. Centrality

A large volume of research on networks has been devoted to the concept of centrality. This research addresses the question:

Which are the most important nodes or edges in a network?

- what are the important **Web pages** about a certain topic to be delivered by a search engine?
- which are the popular **actors** in a social networks?
- what are the influential **academic papers** in a discipline?
- what are the crucial **proteins** in an living being?
- which are the vital **species** in an ecosystem?
- which are the indispensable **routers** in a computer network?

Degree centrality

- **degree** is a simple centrality measure that counts how many neighbors a node has
- if the network is directed, we have two versions of the measure: **in-degree** is the number of in-coming links; **out-degree** is the number of out-going links
- typically, we are interested in **in-degree**, since in-links are assigned by other nodes in the network, while out-links are determined by the node itself.

The thesis of degree centrality reads as follows:

A node is important if it has many neighbors, or, in the directed case, if there are many other nodes that link to it.

Math: Let $A = (a_{i,j})$ be the adjacency matrix of a directed graph and e is a vector with all components equal to unity.

The in-degree centrality x_i of node i is given by:

$$x_i = \sum_k a_{k,i}$$

or in matrix form:

$$x = eA$$

The out-degree centrality y_i of node i is given by:

$$y_i = \sum_k a_{i,k}$$

or in matrix form:

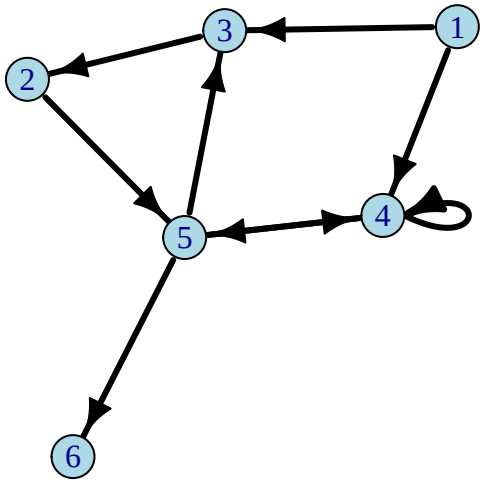
$$y = eA^T$$

- Function `degree` computes unweighted degree centrality

- Function `strength` computes weighted degree centrality

```
library(igraph)
edges = c(1,3, 1,4, 2,5, 3,2, 4,4, 4,5, 5,3, 5,6, 5,4)
dg = graph(edges, directed=TRUE)
coords = layout_with_fr(dg)
plot(dg, layout=coords,
```

```
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black")
```




```
degree(dg, mode = "in")
```

```
## [1] 0 1 2 3 2 1
```

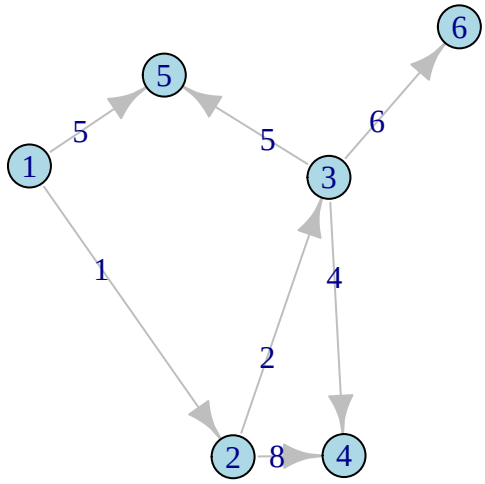
```
degree(dg, mode = "out")
```

```
## [1] 2 1 1 2 3 0
```

```
degree(dg, mode = "total")
```

```
## [1] 2 2 3 5 5 1
```

```
edges = c(1,2, 1,5, 2,3, 2,4, 3,4, 3,5, 3,6)
wg = graph(edges, directed=TRUE)
E(wg)$weight = c(1, 5, 2, 8, 4, 5, 6)
coords = layout_with_fr(wg)
plot(wg, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 1, edge.color = "grey", edge.label = E(wg)$weight)
```




```r
strength(wg, mode = "in")
```

```
## [1]  0  1  2 12 10  6
```

```r
strength(wg, mode = "out")
```

```
## [1]  6 10 15  0  0  0
```

```r
strength(wg, mode = "total")
```

```
## [1]  6 11 17 12 10  6
```

Example: Write a function `degreeCentrality` that, given an adjacency matrix A, computes the in, out and total degree of the corresponding graph (without using the `degree` function).

```r
# Degree centrality
# INPUT
# g = graph
# mode = degree mode
degreeCentrality = function(A, mode) {
  n = nrow(A)
  e = rep(1, n);
  if (mode == "in") x = e %*% A
  if (mode == "out") x = e %*% t(A)
  if (mode == "total") x = e %*% (A + t(A));
  return(as.vector(x));
}
```

```
(A = as_adjacency_matrix(dg, sparse=FALSE))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    1    1    0    0
## [2,]    0    0    0    0    1    0
## [3,]    0    1    0    0    0    0
## [4,]    0    0    0    1    1    0
## [5,]    0    0    1    1    0    1
## [6,]    0    0    0    0    0    0
```

```
degreeCentrality(A, mode = "in")
```

```
## [1] 0 1 2 3 2 1
```

```
degreeCentrality(A, mode = "out")
```

```
## [1] 2 1 1 2 3 0
```

```
degreeCentrality(A, mode = "total")
```

```
## [1] 2 2 3 5 5 1
```

Example: The friendship paradox is a phenomenon observed by sociologist Scott L. Feld. It claims that, on average,

> I have fewer friends than my friends have :-(

On a social network, it formally says that the mean node degree:

$$\mu_1 = \frac{\sum_i k_i}{n}$$

is less than or equal to the mean neighbors degree:

$$\mu_2 = \frac{\sum_{i,j} a_{i,j} k_j}{\sum_{i,j} a_{i,j}}$$

1. Prove the paradox.
2. When is $\mu_1 = \mu_2$?

Notice that:

$$\mu_2 = \frac{\sum_{i,j} a_{i,j} k_j}{\sum_{i,j} a_{i,j}} = \frac{\sum_j \sum_i a_{i,j} k_j}{\sum_j \sum_i a_{i,j}} = \frac{\sum_j k_j \sum_i a_{i,j}}{\sum_j k_j} = \frac{\sum_j k_j \cdot k_j}{\sum_i k_i} = \frac{\langle k^2 \rangle}{\langle k \rangle}$$

where $\langle k^2 \rangle$ is the quadratic mean of degrees and $\langle k \rangle$ is the mean of degrees.

Hence show that:

$$\mu_2 - \mu_1 \geq 0$$

$$\mu_2 - \mu_1 = \frac{\langle k^2 \rangle}{\langle k \rangle} - \langle k \rangle = \frac{\langle k^2 \rangle - \langle k \rangle^2}{\langle k \rangle} = \frac{\sigma^2}{\langle k \rangle} \geq 0$$

Since both the variance σ^2 and the mean $\langle k \rangle$ of the degree distribution are positive quantities, we have that the inequality holds.

In particular we have that $\mu_1 = \mu_2$ if and only if $\sigma^2 = 0$ if and only if all nodes have the same degree (the graph is regular). On the other hand, when the distribution of degrees is skewed (the typical case), the inequality is proper.

This because hub nodes, which are nodes with a high degree, are counted once in μ_1, but are counted many times in μ_2, because they are, by definition, friends on many others.

```
library(igraph)
# scale-free network
g = sample_pa(100, m=2)
d = degree(g)
sd(d)
```

```
## [1] 4.90067
```

```
(mu1 = mean(d))
```

```
## [1] 3.94
```

```
(mu2 = mean(d*d) / mean(d))
```

```
## [1] 9.974619
```

```
#random network
g = sample_gnm(100, 200)
d = degree(g)
sd(d)
```

```
## [1] 1.927997
```

```
(mu1 = mean(d))
```

```
## [1] 4
```

```
(mu2 = mean(d*d) / mean(d))
```

```
## [1] 4.92
```

Closeness centrality

Closeness centralityv measures the mean distance from a vertex to other vertices.

Given nodes i and j, the distance $d_{i,j}$ between them is the length of a **shortest path** from i to j.

The mean distance for vertex i to others is:

$$l_i = \frac{1}{n-1} \sum_{j \neq i} d_{i,j}$$

and the closeness centrality for i is:

$$C_i = \frac{1}{l_i} = \frac{n-1}{\sum_{j \neq i} d_{i,j}}$$

Example:

- when does the mean distance take maximum value and what is this value?
- when does the mean distance take minimum value and what is this value?

The maximum value for mean distance, hence the minimum for closeness, occurs for a root of a **chain graph**, a graph where the n nodes form a linear sequence or chain, where the roots are the two ends of the chain:

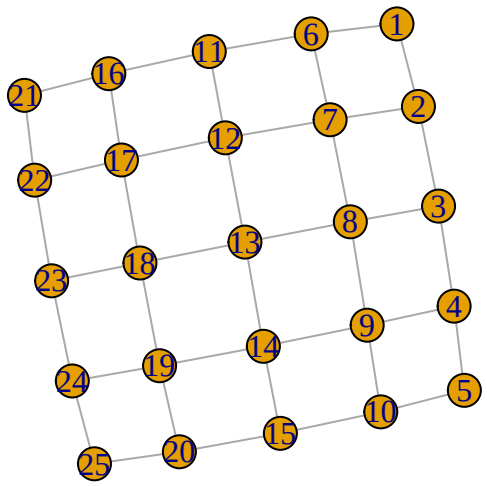
$$\frac{1}{n-1}\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2(n-1)} = \frac{n}{2}$$

The minimum value for mean distance, hence the maximum for closeness, occurs for the central node of a **star graph**, a network composed of a vertex attached to $n-1$ other vertices, whose only connection is with the central node:

$$\frac{1}{n-1}\sum_{i=1}^{n-1} 1 = \frac{n-1}{n-1} = 1$$

Function `closeness` computes closeness centrality (this function assigns a distance equal to the number of nodes of the graph to pairs of nodes that are not reachable).

```
g = make_lattice(c(5,5))
plot(g)
```




```
x = closeness(g)
names(x) = 1:vcount(g)
round(x, 3)
```

```
##     1     2     3     4     5     6     7     8     9    10    11    12    13
## 0.010 0.012 0.012 0.012 0.010 0.012 0.014 0.015 0.014 0.012 0.012 0.015 0.017
##    14    15    16    17    18    19    20    21    22    23    24    25
## 0.015 0.012 0.012 0.014 0.015 0.014 0.012 0.010 0.012 0.012 0.012 0.010
```

Betweenness centrality

- **betweenness centrality** measures the extent to which a vertex lies on paths between other vertices
- vertices with high betweenness may have considerable influence within a network by virtue of their control over information passing between others
- they are also the ones whose removal from the network will most disrupt communications between other vertices because they lie on the largest number of paths taken by messages

Let $n^i_{s,t}$ be the number of **shortest paths** from s to t that pass through i and let $n_{s,t}$ be the total number of geodesic paths from s to t.

Then the betweenness centrality of vertex i is:

$$b_i = \sum_{\substack{s \neq t \\ s \neq i \\ t \neq i}} w^i_{s,t} = \sum_{\substack{s \neq t \\ s \neq i \\ t \neq i}} \frac{n^i_{s,t}}{n_{s,t}}$$

where the formula counts undirected paths in only one direction and, by convention, the ratio $w^i_{s,t} = 0$ if $n_{s,t} = 0$.

Notice that each pair of vertex s, t contribute to the sum for i with a weight $w^i_{s,t}$ between 0 and 1 expressing the betweenness of i with respect to the pair s, t.

Example:

- when does the mean betweenness take maximum value and what is this value?
- when does the mean betweenness take minimum value and what is this value?

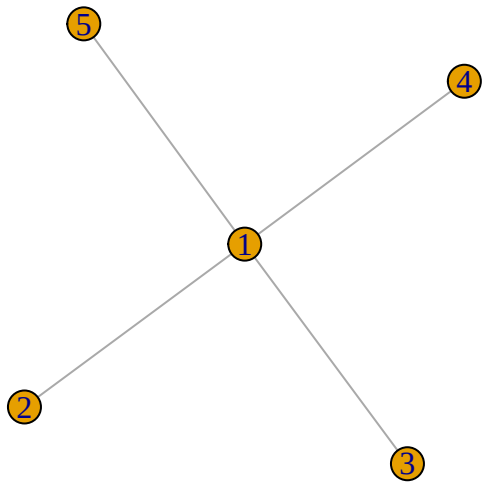
The maximum possible value for betweenness occurs for the central node of a **star graph**. All paths go through the central vertex, hence its betweenness is

$$\sum_{i=1}^{n-2} i = (n-2)(n-1)/2$$

At the other end of the scale, the smallest possible value for betweenness occurs for a **leaf node** (a node of degree 1). There are no shortest paths containing the leaf node, hence its betweenness is 0.

Function `betweenness` computes betweenness centrality.

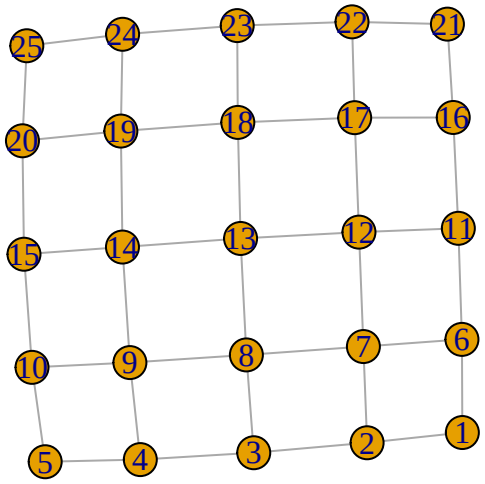
```
g = make_star(5, mode = "undirected")
plot(g)
```




```
x = betweenness(g)
names(x) = 1:vcount(g)
round(x, 1)
```

```
## 1 2 3 4 5
## 6 0 0 0 0
```

```
g = make_lattice(c(5,5))
plot(g)
```




```
x = betweenness(g)
names(x) = 1:vcount(g)
round(x, 0)
```

```
##  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
##  3 17 22 17  3 17 45 55 45 17 22 55 66 55 22 17 45 55 45 17  3 17 22 17  3
```

Dig deeper

Typically, shortest paths are considered in the definition of both closeness and betweenness. There are two drawbacks of this approach:

1. all paths (even slightly) longer than the shortest ones are not considered at all
2. the actual number of shortest paths that lie among the two vertices is irrelevant

In many applications, however, it is reasonable to consider the abundance and the length of all paths of the network, since communication on the network is enhanced as soon as more routes are possible, in particular if these pathways are short.

Current flow centrality addresses these issues, you can read more in this article: Resistance distance, closeness, and betweenness.

6. Centrality: recursive methods

Characteristic polynomial

- the eigenvalues of an $n \times n$ matrix A are the roots of the **characteristic polynomial** of A:

$$p(\lambda) = \det(A - \lambda I)$$

- since $p(\lambda)$ has degree n, there are n eigenvalues, but some of them can be complex, even if A is real, and some can be repeated (however all the eigenvalues of a symmetric matrix are real)

Multiplicity

- the **algebraic multiplicity** μ_λ of an eigenvalue λ is the number of times it is repeated as root of the characteristic polynomial
- the **geometric multiplicity** γ_λ of an eigenvalue λ is the number of linear independent eigenvectors associated with λ
- it holds that $1 \leq \gamma_\lambda \leq \mu_\lambda \leq n$
- if $\mu_\lambda = \gamma_\lambda = 1$ then λ is said to be **simple**

Eigenvalues and eigenvectors

- a vector $x \neq 0$ is a **right eigenvector** of a square matrix A if $Ax = \lambda x$ where λ is a scalar defined as eigenvalue associated with x
- a vector $x \neq 0$ is a **left eigenvector** of a square matrix A if $xA = \lambda x$ where λ is a scalar defined as eigenvalue associated with x
- unless A is symmetric, left eigenvectors are different from right eigenvectors
- however, the set of the eigenvalues associated to right and left eigenvectors is exactly the same
- the definition of eigenvector has a very intuitive geometrical meaning: If x is a (right) eigenvector of A, then Ax has the same direction of x. The action of A on x is, in some sense, quite polite

Spectrum and spectral radius

- the set of the distinct eigenvalues of A is defined as **spectrum** of A and denoted with $\sigma(A)$
- the **spectral radius** of A is the nonnegative number

$$\rho(A) = \max_{\lambda \in \sigma(A)} |\lambda|$$

Irreducibility

A square matrix A is irreducible provided that the graph whose A is the adjacency matrix is strongly connected, that is, there is a directed path between any ordered couple of arbitrarily chosen vertexes

Perron-Frobenius Theorem

If a nonnegative matrix A is irreducible then:

1. the spectral radius $r = \rho(A) > 0$ and $r \in \sigma(A)$
2. r is a simple eigenvalue of A, hence its associated eigenvector is unique (up to a multiplicative constant)
3. the unique right eigenvector x such that $Ax = rx$ is positive
4. the unique left eigenvector y such that $yA = ry$ is positive

Eigenvector centrality

- a natural extension of degree centrality is **eigenvector centrality**
- in-degree centrality awards one centrality point for every link a node receives
- but not all vertices are equivalent: some are more relevant than others, and, reasonably, endorsements from important nodes count more

The eigenvector centrality thesis reads:

> A node is important if it is linked to by other important nodes

Math Let $A = (a_{i,j})$ be the adjacency matrix of a graph. The eigenvector centrality x_i of node i is given by:

$$x_i = \frac{1}{\lambda} \sum_k a_{k,i}\, x_k$$

where we assume the unknown $\lambda \neq 0$. In matrix form we have:

$$\lambda x = xA$$

The centrality vector x is the left-hand **eigenvector** of the adjacency matrix A associated with the eigenvalue λ.

Notice that $\lambda x = xA$ is not a linear system since both λ and x are variables.

It is wise to choose λ as the largest eigenvalue in absolute value of matrix A.

By virtue of **Perron-Frobenius theorem** for non-negative matrices, this choice guarantees the following desirable property:

> If the adjacency matrix is irreducible, or equivalently if the graph is (strongly) connected, then the eigenvector solution is both unique and positive.

Power method

The **power method** can be used to solve the eigenvector centrality problem.

Let $m(v)$ denote the signed component of maximal magnitude of vector v. If there is more than one maximal component, let $m(v)$ be the first one. For instance, $m(-3, 3, 2) = -3$.

Let $x^{(0)} = e$, where e is the vector of all 1's. For $k \geq 1$:

1. repeatedly compute $x^{(k)} = x^{(k-1)}A$;
2. normalize $x^{(k)} = x^{(k)}/m(x^{(k)})$;

until the desired precision is achieved.

It holds that:

- $x^{(k)}$ converges to the **dominant eigenvector** of A
- $m(x^{(k)})$ converges to the **dominant eigenvalue** of A

Let us sort the eigenvalues of A in order of decreasing magnitude:

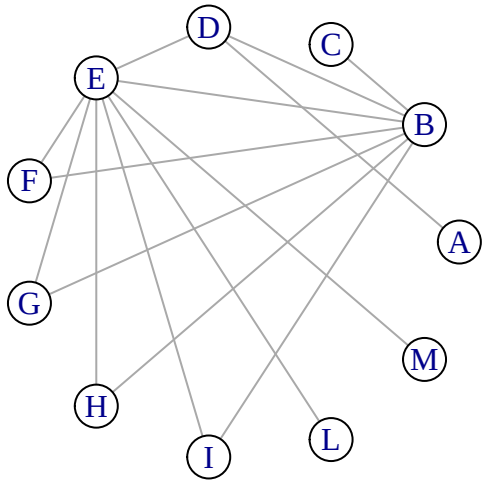
$$|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$$

Then:

- the power method converges when $|\lambda_1| > |\lambda_2|$
- the rate of convergence is the rate at which $(\lambda_2/\lambda_1)^k$ goes to 0
- in particular if $|\lambda_2/\lambda_1|$ is close to 1, the convergence is very slow

Function `eigen_centrality` computes eigenvector centrality for a graph using the ARPACK package (it works only on symmetric matrices):

```
library(igraph)
g = read_graph(file = "EVGraphUndirected.gml", format = "gml")
coords = layout_in_circle(g)
plot(g, layout = coords,
     vertex.size = 20,
     vertex.color = "white")
```



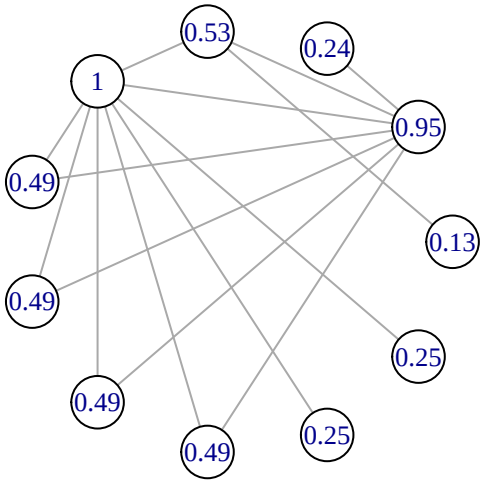

```
ev = eigen_centrality(g)
(x = ev$vector)
```

```
##         A         B         C         D         E         F         G         H
## 0.1332886 0.9460927 0.2395318 0.5264579 1.0000000 0.4927119 0.4927119 0.4927119
##         I         L         M
## 0.4927119 0.2531801 0.2531801
```

```
(lambda = ev$value)
```

```
## [1] 3.949758
```

```
plot(g, layout = coords,
     vertex.size = 25,
     vertex.color = "white",
     vertex.label = round(x, 2),
     vertex.label.cex = 0.80)
```




```
# check
A = as_adjacency_matrix(g, sparse = FALSE)
as.vector(x %*% A)
```

```
##  [1] 0.5264579 3.7368371 0.9460927 2.0793813 3.9497581 1.9460927 1.9460927
##  [8] 1.9460927 1.9460927 1.0000000 1.0000000
```

```
lambda * x
```

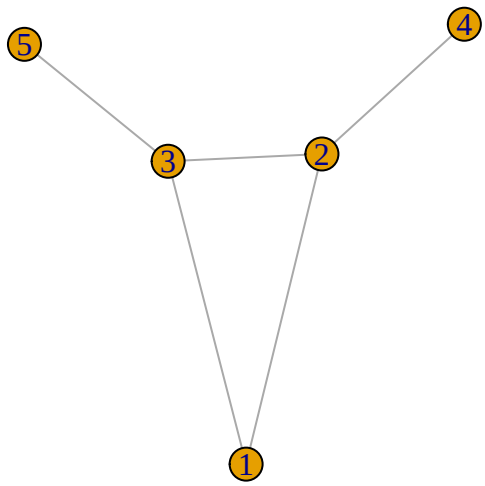
```
##         A         B         C         D         E         F         G         H
## 0.5264579 3.7368371 0.9460927 2.0793813 3.9497581 1.9460927 1.9460927 1.9460927
##         I         L         M
## 1.9460927 1.0000000 1.0000000
```

Example: Code in R the power method and test it on the Bull graph.

```
# Eigenvector centrality (power method)
#INPUT
# g = graph
# t = precision
# OUTPUT
# A list with:
# vector = centrality vector
# value = eigenvalue
# iter = number of iterations

eigenvectorCentrality = function(g, t) {
  A = as_adjacency_matrix(g);
```

```
  n = vcount(g);
  x0 = rep(0, n);
  x1 = rep(1/n, n);
  eps = 1/10^t;
  iter = 0;
  while (sum(abs(x0 - x1)) > eps) {
    x0 = x1;
    x1 = as.vector(x1 %*% A);
    m = x1[which.max(abs(x1))];
    x1 = x1 / m;
    iter = iter + 1;
  }
  return(list(vector = x1, value = m, iter = iter))
}
```

```
g = make_graph("Bull")
plot(g)
```




```
eigenvectorCentrality(g, 6)
```

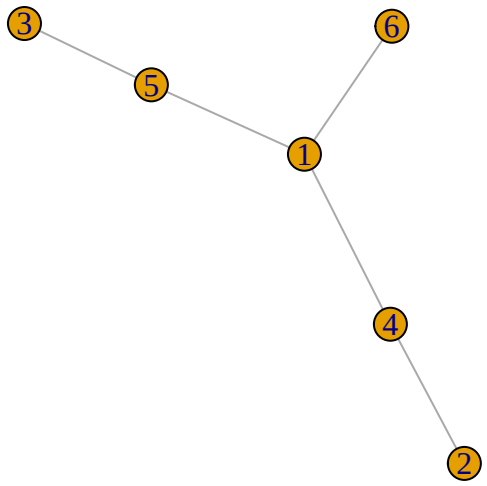
```
## $vector
## [1] 0.8685172 1.0000000 1.0000000 0.4342586 0.4342586
##
## $value
## [1] 2.302775
##
## $iter
```

```
## [1] 26
```

Example:

1. Build a bipartite graph
2. Run the power method on the graph. Does it converge?
3. Explain in theory the absence of convergence

```
g = make_bipartite_graph(types = c(0,0,0, 1,1,1),
                         edges = c(1,4, 1,5, 1,6, 2,4, 3,5))

plot(g)
```




```
# INFINITE COMPUTATION
# eigenvectorCentrality(g, 6)

A = as_adjacency_matrix(g, sparse = FALSE)
eigen(A)$values
```

```
## [1]  1.9318517  1.0000000  0.5176381 -0.5176381 -1.0000000 -1.9318517
```

```
eigen_centrality(g)$vector
```

```
## [1] 1.0000000 0.3660254 0.3660254 0.7071068 0.7071068 0.5176381
```

Katz centrality

- a practical problem with eigenvector centrality is that it works well only if the graph is (strongly) connected

- real undirected networks typically have a large connected component, of size proportional to the network size. However, real directed networks do not.
- if a directed network is not strongly connected, then only vertices that are in strongly connected components of at least two nodes or in the out-component of such components can have non-zero eigenvector centrality
- for instance, in a directed acyclic graph, all nodes get null eigenvector centrality
- a way to work around this problem is to give each node a small amount of centrality for free, regardless of the position of the vertex in the network, that it can transfer to other nodes
- it follows that highly linked nodes have high centrality, regardless of the centrality of the linkers
- however, nodes that receive few links may still have high centrality if the linkers have large centrality

The Katz centrality thesis is then:

> A node is important if it is linked from other important nodes or if it is highly linked.

This method has been proposed by **Leo Katz** (A new status index derived from sociometric analysis. Psychometrika, 1953) and later refined by **Charles H. Hubbell** (An input-output approach to clique identification. Sociometry, 1965).

Let $A = (a_{i,j})$ be the adjacency matrix of a directed graph.

The Katz centrality x_i of node i is given by:

$$x_i = \alpha \sum_k a_{k,i}\, x_k + \beta$$

where $\alpha > 0$ and $\beta > 0$ are positive constants. In matrix form we have:

$$x = \alpha x A + \beta$$

where β is now a vector whose elements are all equal a given positive constant.

This is no more an eigenvector problem, but instead a linear system, since α and β are constants.

Notice that the centrality vector x is defined by two components:

- an **endogenous component** that takes into consideration the network topology
- an **exogenous component** that is independent of the network structure

Notice that $x = \alpha x A + \beta$ is equivalent to $x(I - \alpha A) = \beta$.

If matrix $(I - \alpha A)$ is invertible then:

$$x = \beta (I - \alpha A)^{-1}$$

If we wish to make use of the Katz centrality we must first choose a value for constant α and β.

- if we let $\alpha \to 0$, then only the constant term survives and all vertices have the same centrality β
- as we increase α from zero the centralities increase and eventually there comes a point at which they diverge
- this happens at the point where $(I - \alpha A)$ is singular, i.e., when $\det(I - \alpha A) = 0$
- rewriting this condition as:
  $$\det(A - \alpha^{-1} I) = 0$$
  we see that it is simply the characteristic equation whose roots α^{-1} are equal to the eigenvalues of the adjacency matrix
- as α increases, the determinant first crosses zero when $\alpha^{-1} = \lambda_1$, the largest eigenvalue of A, or alternatively when $\alpha = 1/\lambda_1$
- thus, we should choose a value of $0 < \alpha < 1/\rho(A)$, where $\rho(A)$ the the spectral radius of A

Recall that

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

whenever $|x| < 1$. This result holds also for matrices.

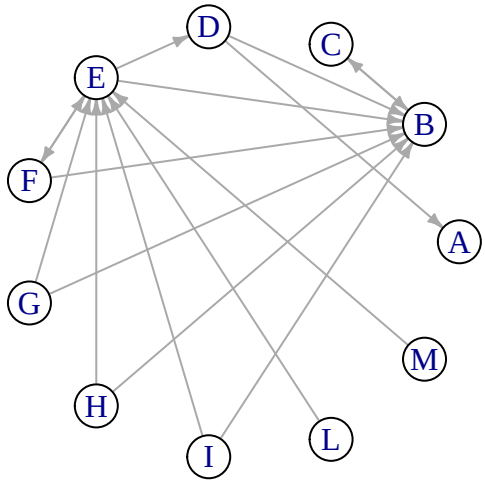
Hence, if $\rho(\alpha A) = \alpha\rho(A) < 1$, that is, $\alpha < 1/\rho(A)$, we can express the Katz centrality as follows:

$$x = \beta(I - \alpha A)^{-1} = \beta \sum_{i=0}^{\infty} (\alpha A)^i = \beta + \beta\alpha A + \beta\alpha^2 A^2 + \dots$$

- recall that A^i contains all paths of length i between nodes of the graph
- the Katz centrality of a node is hence the number of *weighted* paths reaching the node in the network plus an exogenous factor, a generalization of the in-degree measure which counts only paths of length one
- long paths are weighted less than short ones by exploiting the attenuation factor α; this is reasonable since endorsements devalue over long chains of links
- for small (close to 0) values of α the contribution given by paths longer than one rapidly declines, and thus Katz scores are mainly influenced by short paths (mostly in-degrees) and by the exogenous factor of the system
- when the damping factor is large, long paths are devalued smoothly, and Katz scores are more influenced the by endogenous topological part of the system
- as for the value of constant β, it is possible to assign to each node i a different minimum amount of status β_i
- nodes with higher exogenous status are, for some reason, privileged from the start
- for instance, in the setting of the Web, we might bias the result towards topics, like contemporary art or ballet, that are of particular interest to the user to which the Web page ranking is served

Function `alpha_centrality` computes Katz centrality for a graph:

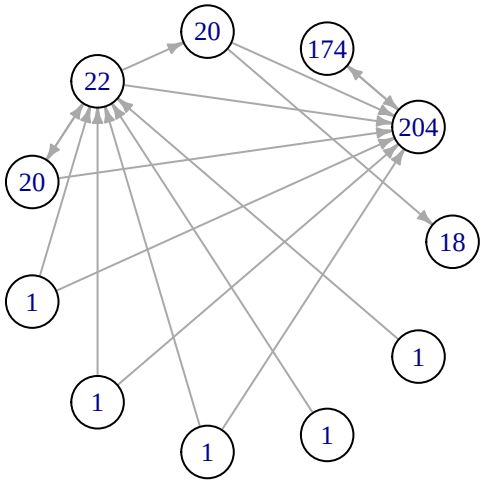
```
g = read_graph(file = "EVGraphDirected.gml", format = "gml")
coords = layout_in_circle(g)

plot(g,
     layout = coords,
     vertex.size = 20,
     vertex.color = "white",
     edge.arrow.size = 0.4)
```



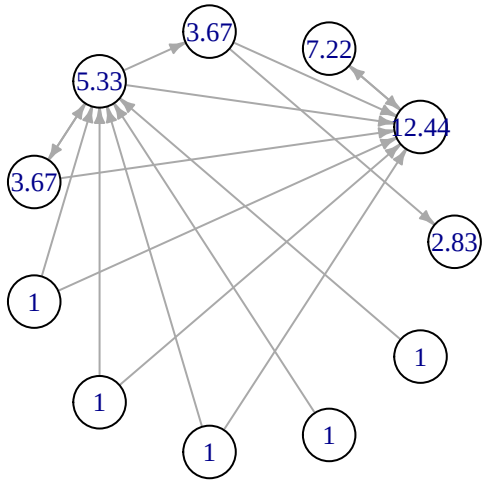

```
A = as_adjacency_matrix(g)
eig = eigen(A)$values
r = max(abs(eig))
alpha = 0.85 / r
(x =  alpha_centrality(g, alpha = alpha, exo = 1))
```

```
##         A         B         C         D         E         F         G         H
##  17.73198 203.77891 174.21208  19.68468  21.98198  19.68468   1.00000   1.00000
##         I         L         M
##   1.00000   1.00000   1.00000
```

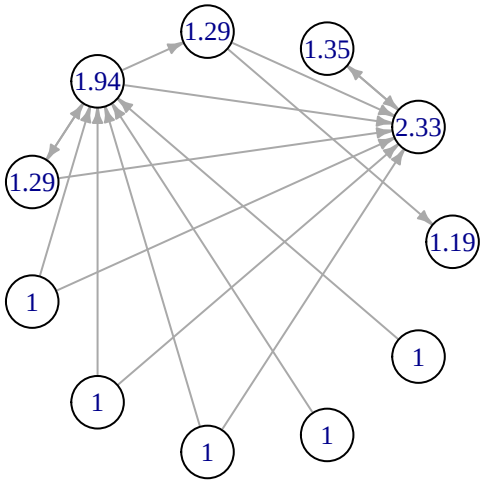
```
plot(g,
     layout = coords,
     vertex.size = 25,
     vertex.color = "white",
     vertex.label = round(x, 0),
     vertex.label.cex = 0.80,
     edge.arrow.size = 0.4)
```




```
alpha = 0.5 / r
x =  alpha_centrality(g, alpha = alpha, exo = 1)

plot(g,
     layout = coords,
     vertex.size = 25,
     vertex.color = "white",
     vertex.label = round(x, 2),
     vertex.label.cex = 0.80,
     edge.arrow.size = 0.4)
```




```
alpha = 0.15 / r
x =  alpha_centrality(g, alpha = alpha, exo = 1)

plot(g,
     layout = coords,
     vertex.size = 25,
     vertex.color = "white",
     vertex.label = round(x, 2),
     vertex.label.cex = 0.80,
     edge.arrow.size = 0.4)
```



PageRank

- a potential problem with Katz centrality is the following: if a node with high centrality links many others then all those others get high centrality
- in many cases, however, it means less if a node is only one among many to be linked
- the centrality gained by virtue of receiving a link from an important node should be diluted if the important vertex is very magnanimous with endorsements
- PageRank is an adjustment of Katz centrality that takes into consideration this issue
- it was proposed (and patented) by Sergey Brin and Larry Page (The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems, 1998).

There are three distinct factors that determine the PageRank of a node:

1. the number of links it receives
2. the centrality of the linkers
3. the link propensity of the linkers

The PageRank thesis might be summarized as follows:

> A node is important if it linked from other important and link parsimonious nodes or if it is highly linked.

Let $A = (a_{i,j})$ be the adjacency matrix of a directed graph.

The PageRank centrality x_i of node i is given by:

$$x_i = \alpha \sum_k \frac{a_{k,i}}{d_k} x_k + \beta$$

where α and β are constants and d_k is the out-degree of node k if such degree is positive, or $d_k = 1$ if the out-degree of k is null.

In matrix form we have:

$$x = \alpha x D^{-1} A + \beta$$

where β is now a vector whose elements are all equal a given positive constant and D^{-1} is a diagonal matrix with i-th diagonal element equal to $1/d_i$.

It follows that x can be computed as:

$$x = \beta(I - \alpha D^{-1} A)^{-1}$$

The damping factor α and the personalization vector β have the same role seen for Katz centrality. In particular, α should be chosen between 0 and $1/\rho(D^{-1}A)$.

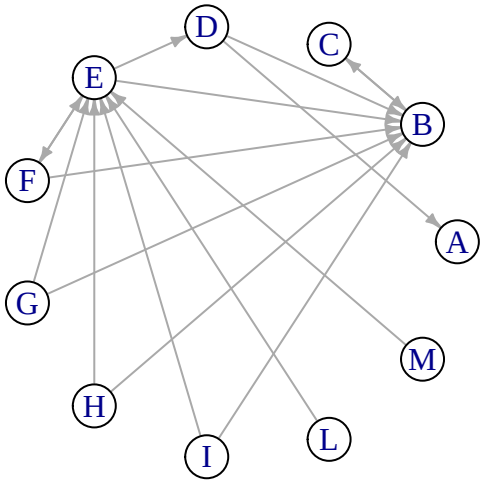
Notice that matrix $P = D^{-1}A$ is **row-stochastic**, that is each row sums to 1. The element

$$p_{i,j} = \frac{a_{i,j}}{d_i} = \frac{a_{i,j}}{\sum_k a_{i,k}}$$

is the probability of moving from i to j during a random walk on the graph of A.

Function `page_rank` computes PageRank centrality for a graph:

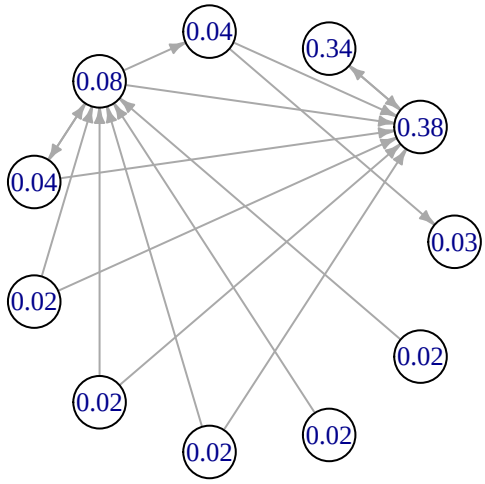
```
g = read_graph(file = "EVGraphDirected.gml", format = "gml")
coords = layout_in_circle(g)

plot(g,
     layout = coords,
     vertex.size = 20,
     vertex.color = "white",
     edge.arrow.size = 0.4)
```




```
pr =  page_rank(g)
# equivalent to:
pr =  page_rank(g,
                damping = 0.85,
                personalized = rep(1, vcount(g)))

(x = pr$vector)
```

```
##          A          B          C          D          E          F          G
## 0.03278149 0.38440095 0.34291029 0.03908709 0.08088569 0.03908709 0.01616948
##          H          I          L          M
## 0.01616948 0.01616948 0.01616948 0.01616948
```

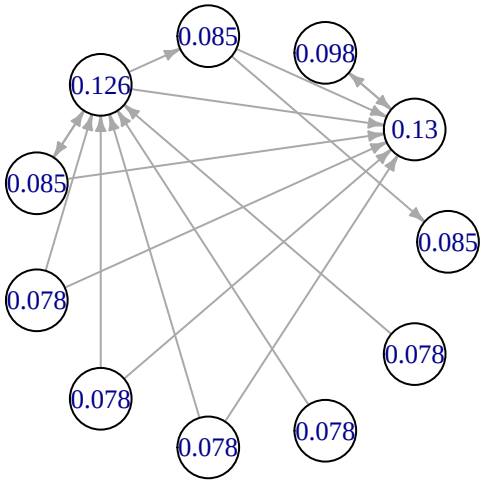
```
plot(g,
     layout = coords,
     vertex.size = 25,
     vertex.color = "white",
     vertex.label = round(x, 2),
     vertex.label.cex = 0.80,
     edge.arrow.size = 0.4)
```




```
pr =  page_rank(g, damping = 0.15)
(x = pr$vector)
```

```
##          A          B          C          D          E          F          G
## 0.08478337 0.12976638 0.09789382 0.08472679 0.12595853 0.08472679 0.07842886
##          H          I          L          M
## 0.07842886 0.07842886 0.07842886 0.07842886
```

```
plot(g,
     layout = coords,
     vertex.size = 30,
     vertex.color = "white",
     vertex.label = round(x, 3),
     vertex.label.cex = 0.80,
     edge.arrow.size = 0.4)
```



Dig deeper

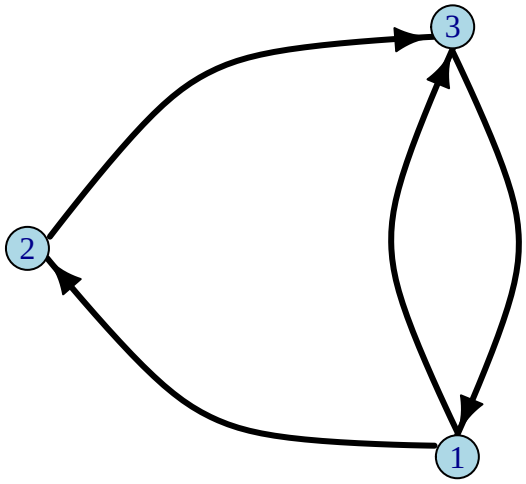
PageRank centrality, regarded as a ranking measure, is a remarkably old method. Early pioneers of this technique are:

- Wassily W. Leontief (The Structure of American Economy, 1919-1929. Harvard University Press, 1941)
- John R. Seeley (The net of reciprocal influence: A problem in treating sociometric data. The Canadian Journal of Psychology, 1949).

Read the full story.

Example: Consider the following network:

```
edges = c(1,2, 2,3, 3,1, 1,3)
g = graph(edges, directed=TRUE)
coords = layout_with_fr(g)
plot(g, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black",
     edge.curved = TRUE)
```



1. without running the code, figure out the ranking of nodes according to PageRank centrality
2. test you hypothesis using the code
3. Node 3 receives from both 1 and 2, hence it should be the leader
4. Node 1 receives from node 3, the leader, which reaches out only 1, hence it should be the second ranked
5. Node 2 receives from 1, which also reaches out 3, hence it should be the last in the ranking

```
page_rank(g)$vector
```

```
## [1] 0.3877897 0.2148106 0.3973997
```

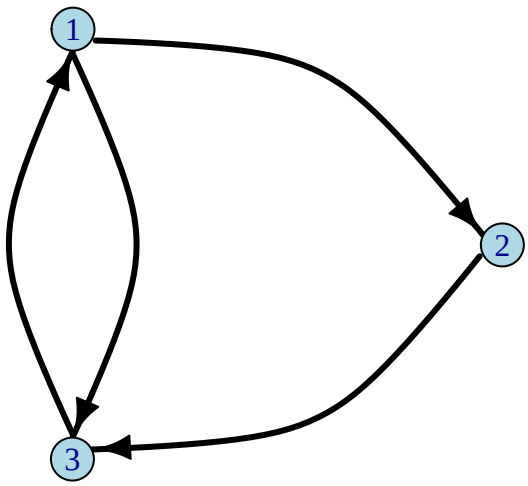
Notice the large difference between nodes 1 and 2.

Example: Write a power method for the PageRank and test it.

```
# PageRank centrality (power method)
# g = graph
# alpha = damping factor
# beta = personalized vector
# t = precision
# OUTPUT
# A list with:
# vector = centrality vector
# iter = number of iterations

pagerankCentrality = function(g, alpha, beta, t) {
  n = vcount(g)
```

```
  A = as_adjacency_matrix(g)

  d = degree(g, mode="out")
  d[d==0] = 1
  D = diag(1/d)
  A = D %*% A

  x0 = rep(0, n)
  x1 = rep(1/n, n)
  eps = 1/10^t
  iter = 0
  while (sum(abs(x0 - x1)) > eps) {
    x0 = x1
    x1 = alpha * x1 %*% A + beta
    iter = iter + 1
  }
  return(list(vector = as.vector(x1), iter = iter))
}
```

```
edges = c(1,2, 2,3, 3,1, 1,3)
g = graph(edges, directed=TRUE)
coords = layout_with_fr(g)
plot(g, layout=coords,
     vertex.size = 20, vertex.color = "lightblue",
     edge.width = 3, edge.color = "black",
     edge.curved = TRUE)
```



```r
## Notice the difference of centralities between 1 and 2
x = pagerankCentrality(g, alpha = 0.85, beta = 1, 6)$vector
x / sum(x)
```

```
## [1] 0.3877897 0.2148106 0.3973997
```

```r
# check
page_rank(g)$vector
```

```
## [1] 0.3877897 0.2148106 0.3973997
```

```r
# Notice that:
x = pagerankCentrality(g, alpha = 0.85, beta = 0, 6)$vector
x / sum(x)
```

```
## [1] 0.4 0.2 0.4
```

HITS

- so far, a node is important if it contains valuable content and hence receives many links from other important sources
- nodes with no incoming links cumulate, in the best case, only a minimum amount of centrality, regardless of how many other useful information sources they reference
- one can argue that a node is important also because it links to other important vertices
- for instance, a review paper may refer to other authoritative sources: it is important because it tells us where to find trustworthy information
- or an important art collector can tell us what is good art

Thus, there are now two types of central nodes:

- **authorities**, that contain reliable information on the topic of interest
- **hubs**, that tell us where to find authoritative information.
- a node may be both an authority and a hub: for instance, a review paper may be highly cited because it contains useful content and it may as well cite other useful sources
- or an important art collector may be also a successful artist

The Kleinberg centrality thesis reads:

> A node is an authority if it is linked to by hubs; it is a hub if it links to authorities.

This method has been conceived by Jon M. Kleinberg (Authoritative sources in a hyperlinked environment. In ACM-SIAM Symposium on Discrete Algorithms, 1998).

Let $A = (a_{i,j})$ be the adjacency matrix of a directed graph. The authority centrality x_i of node i is given by:

$$x_i = \alpha \sum_k a_{k,i}\, y_k$$

and hub centrality y_i of node i is given by:

$$y_i = \beta \sum_k a_{i,k}\, x_k$$

where α and β are constants. In matrix form we have:

$$\begin{aligned} x &= \alpha y A \\ y &= \beta x A^T \end{aligned}$$

or, combining the two:

$$\begin{aligned} \lambda x &= x A^T A \\ \lambda y &= y A A^T \end{aligned}$$

where $\lambda = (\alpha\beta)^{-1}$.

Hence the authority vector is an eigenvector of the **authority matrix** $A^T A$ and the hub vector is an eigenvector of the **hub matrix** AA^T (the matrices have the same eigenvalues).

Notice moreover that

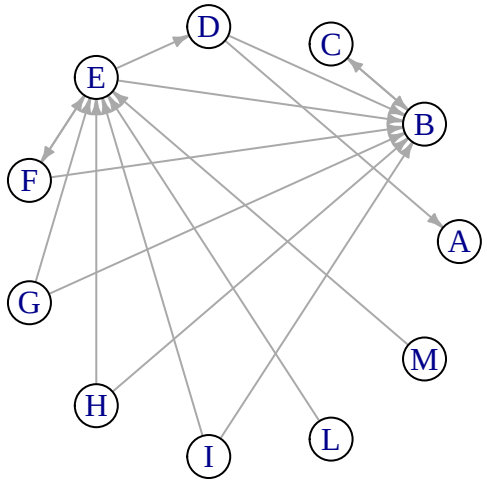
$$\lambda x A^T = x A^T A A^T$$

and hence xA^T is an eigenvector of the hub matrix AA^T with the same eigenvalue λ.

Hence, once computed the authority vector x, we can get the hub vector as $y = xA^T$, or, equivalently, the hub score of i is the sum of authority scores of nodes linked to by i.

Functions `authority_score` and `hub_score` computes authority and hub centralities:

```
g = read_graph(file = "EVGraphDirected.gml", format = "gml")
coords = layout_in_circle(g)

plot(g,
     layout = coords,
     vertex.size = 20,
     vertex.color = "white",
     edge.arrow.size = 0.4)
```



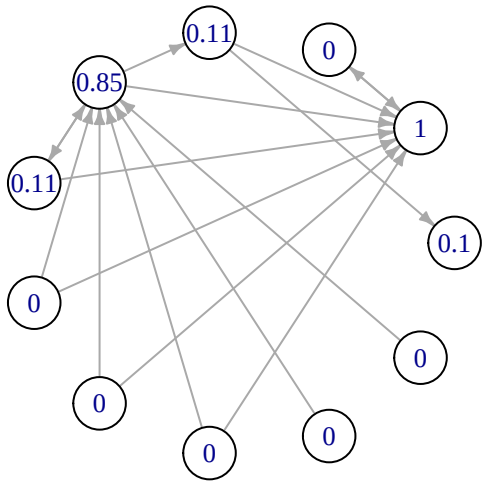
```
authority =  authority_score(g)$vector
hub =  hub_score(g)$vector

round(authority, 2)
```

```
##    A    B    C    D    E    F    G    H    I    L    M
## 0.10 1.00 0.00 0.11 0.85 0.11 0.00 0.00 0.00 0.00 0.00
```

```
round(hub, 2)
```

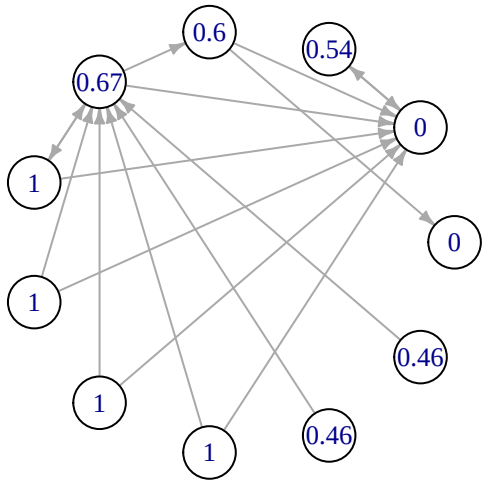
```
##    A    B    C    D    E    F    G    H    I    L    M
## 0.00 0.00 0.54 0.60 0.67 1.00 1.00 1.00 1.00 0.46 0.46
```

```
plot(g,
     layout = coords,
     vertex.size = 25,
     vertex.color = "white",
     vertex.label = round(authority, 2),
     vertex.label.cex = 0.80,
     edge.arrow.size = 0.4)
```




```
plot(g,
     layout = coords,
     vertex.size = 25,
     vertex.color = "white",
     vertex.label = round(hub, 2),
     vertex.label.cex = 0.80,
```

```
    edge.arrow.size = 0.4)
```



Dig deeper - Signed networks

- in our discussion of networks thus far, we have generally viewed the relationships contained in these networks as having positive connotations
- the terminology of on-line social networks reflects a largely similar view, through its emphasis on the connections one forms with friends, fans, followers, and so forth
- but in most network settings, there are also negative effects at work. Some relations are friendly, but others are antagonistic or hostile
- how should we reason about the mix of **positive and negative relationships** that take place within a network?
- Read more in Chapter 12 (Positive and Negative Relationships) of book Networks, crowds and markets

7. asd

7. Power Measures

- in some circumstances centrality - the quality of being connected to central ones - has limited utility in predicting the locus of power in networks
- consider exchange networks, where the relationship in the network involves the transfer of valued items (i.e., information, time, money, energy)
- a set of exchange relations is positive if exchange in one relation promotes exchange in others and negative if exchange in one relation inhibits exchange in others
- in **negative exchange networks**, power comes from being connected to those who have few options. Being connected to those who have many possibilities reduces one's power
- think, for instance, to a social network in which time is the exchanged value
- imagine that every actor has a limited time to listen to others and that each actor divides its time between its neighbors.
- clearly, exchange of time in one relation precludes the exchange of the same time in other relations
- what are the actors that receive most attention?
- these are the nodes that are connected to many neighbors with few options, since they receive almost full attention from all their neighbors
- on the other hand, actors connected to few neighbors with a lot of possibilities receive little consideration, since their neighbors are mostly busy with others

Power

The power thesis is as follows:

> A node is powerful if it is connected with powerless nodes.

Power x_i of node i is given by:

$$x_i = \sum_k \frac{a_{k,i}}{x_k}$$

that is:

$$x = x^{\div} A$$

The above states two properties of power:

1. the larger the degree of a node, the larger its power. Hence, the more ties an actor has, the more powerful the actor is
2. the smaller the power of neighbors of a node, the larger its power. For equal number of ties, actors that are linked to powerless others are powerful; on the other hand, actors that are tied to powerful others are powerless.

Power can be computed with the following **iterative process**.

Let $x_0 = e$, where e is the vector of all 1's. For $k \geq 1$:

$$x_k = x_{k-1}^{\div} A$$

For $k \geq 0$, let $r_k = x_{2k}$ and $c_k = x_{2k+1}$.

- If r_k converges, then it does to a vector r such that $\alpha r = r^{\div} A$, with $\alpha > 0$. It follows that $\sqrt{\alpha} r = (\sqrt{\alpha} r)^{\div} A$, hence $\sqrt{\alpha} r$ is a solution of the power equation $x = x^{\div} A$.
- Similarly, if c_k converges, it does to a vector c such that $\beta c = c^{\div} A$ and $\sqrt{\beta} c$ is a again solution of the power equation $x = x^{\div} A$.

It holds that solutions r and c are proportional: $r = \gamma c$, for some $\gamma > 0$.

- the 1st iteration
$$x_1 = eA$$
is, for a given node i, the degree of i
- the 2nd iteration
$$x_2 = (eA)^{\div} A$$
is, for a given node i, the sum of reciprocals of degrees of neighbors of i
- in general, power of node i is positively influenced by the degrees of nodes that are reachable from i with an even number of edges

- it is negatively influenced by the degrees of nodes that are reachable from i an odd number of edges
- nodes that are reachable from i both with a path of even length as well as with a path of odd length play a dual role with respect to i
- notice, however, that such an influence devalues with the length of the path: it is stronger when the path is shorter.

By **Sinkhorn-Knopp theorem** (on symmetric matrices):

> Power exists, and the corresponding iterative method converges, if and only if the graph G_A associated with matrix A is regularizable.

The regularizability problem is equivalent to the following linear programming feasibility problem:

$$\begin{aligned} Bw &= re \\ w &\geq 1 \end{aligned}$$

where:

- B is the incidence matrix of G_A, that is, an $n \times m$ matrix such that $b_{i,l} = 1$ if i belongs to edge l and $b_{i,l} = 0$ otherwise, with n and m the number of nodes and edges of the graph
- w is a vector of length m with edge weight variables

- r is a variable for the regularization degree.

If matrix A is not regularizable, it can be fully perturbed as follows:

$$A_\epsilon^F = A + \epsilon E.$$

A less intrusive diagonal perturbation is sufficient:

$$A_\epsilon^D = A + \epsilon I$$

where I is the diagonal matrix. This corresponds to add a loop with weight ϵ to each node in the graph.

Example: Prove that the graph associated to the perturbation matrices $A_\epsilon^F = A + \epsilon E$ and $A_\epsilon^D = A + \epsilon I$ are regularizable.

The graph of A_ϵ^F is complete, hence regular, hence regularizable.

The graph of A_ϵ^D has one self-loop for every node. This can be used to adjust the weighting of the edges to obtain a solution:

- Suppose, for instance, that the degrees of nodes A, B and C of the graph without self-loops are 1, 3, and 5.
- Then, take a number larger than the maximum degree, for instance 6. This will be the regular degree.
- Finally, set the weights of the self-loops associated with A, B, and C to $6 - 1 = 5$, $6 - 3 = 3$, and $6 - 5 = 1$.
- Now all nodes have degree 6.

The following user-defined function `regularify` checks is a graph is regularizable and in case gives the regularization solution.

It uses the lpSolve and lpSolveAPI packages for solving linear, integer and mixed integer programs.

In order to find a positive solution $w > 0$, the program solves the linear problem

$$\hat{B}(\hat{w}, r) = -d$$

with $\hat{w} \geq 0$, where:

- $\hat{B}$ is the incidence matrix B with one additional column $-e$,
- d is a vector with node degrees, and
- $(\hat{w}, r)$ is a vector with weight variables $\hat{w}$ and an additional variable r for the regularization degree.

Setting $w = \hat{w} + 1$, the problem corresponds to $Bw = re$, with the constrain that $w \geq 1$.

```
library(lpSolve)
```

```
## Warning: package 'lpSolve' was built under R version 4.0.3
```

```
library(lpSolveAPI)
```

```
## Warning: package 'lpSolveAPI' was built under R version 4.0.3
```

```
regularify = function (g) {
  n = vcount(g)
  m = ecount(g)
  E = get.edges(g, E(g))
  B = matrix(0, nrow = n, ncol = m)
  # build incidence matrix
  for (i in 1:m) {
    B[E[i,1], i] = 1
    B[E[i,2], i] = 1
  }
  # objective function
  obj = rep(0, m + 1)
  # constraint matrix
  con = cbind(B, rep(-1, n))
  # direction of constraints
  dir = rep("=", n)
  # right hand side terms
  rhs = -degree(g)
  # solve the LP problem
  sol = lp("max", obj, con, dir, rhs)
  # get solution
  if (sol$status == 0) {
    s = sol$solution
    # weights
    w = s[1:m] + 1
    # weighted degree
    d = s[m+1]
  }
  # return the solution
  if (sol$status == 0) {
    return(list(weights = w, degree = d))
  }
  else {
```

```
    return(NULL)
  }
}
```

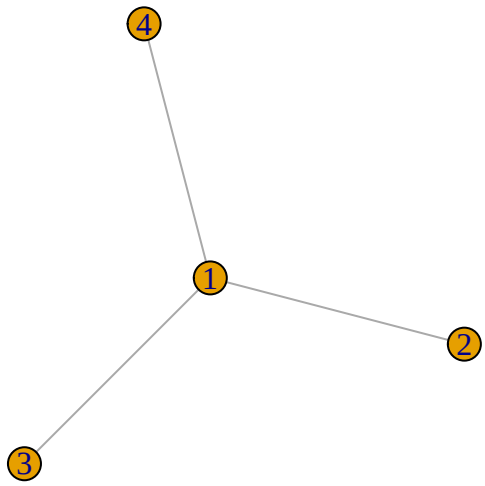
The following user-defined function **power** computes power using a direct computation on a regularizable adjacency matrix A:

```
# Compute power x = (1/x) A
#INPUT
# A = graph adjacency matrix
# t = precision
# OUTPUT
# A list with:
# vector = power vector
# iter = number of iterations

power = function(A, t) {
  n = dim(A)[1];
  # x_2k
  x0 = rep(0, n);
  # x_2k+1
  x1 = rep(1, n);
  # x_2k+2
  x2 = rep(1, n);
  diff = 1
  eps = 1/10^t;
  iter = 0;
  while (diff > eps) {
    x0 = x1;
    x1 = x2;
    x2 = (1/x2) %*% A;
    diff = sum(abs(x2 - x0));
    iter = iter + 1;
  }
  # it holds now: alpha x2 = (1/x2) A
  alpha = ((1/x2) %*% A[,1]) / x2[1];
  # hence sqrt(alpha) * x2 = (1/(sqrt(alpha) * x2)) A
  x2 = sqrt(alpha) %*% x2;
  return(list(vector = as.vector(x2), iter = iter))
}
```

Example:

1. make a star graph and check that it is not regularizable
2. perturb the graph matrix and compute power of the perturbed matrix

```
library(igraph)
g = make_star(4, mode = "undirected")
plot(g)
```




```
# The graph is not regularizable
regularify(g)
```

```
## NULL
```

```
# Use diagonal perturbation
A = as_adjacency_matrix(g)
I = diag(0.15, vcount(g))
(AI = A + I)
```

```
## 4 x 4 sparse Matrix of class "dgCMatrix"
##
## [1,] 0.15 1.00 1.00 1.00
## [2,] 1.00 0.15 .    .
## [3,] 1.00 .    0.15 .
## [4,] 1.00 .    .    0.15
```

```
# compute power
power(AI, 6)
```

```
## $vector
## [1] 6.3540340 0.4739017 0.4739017 0.4739017
##
## $iter
## [1] 18
```

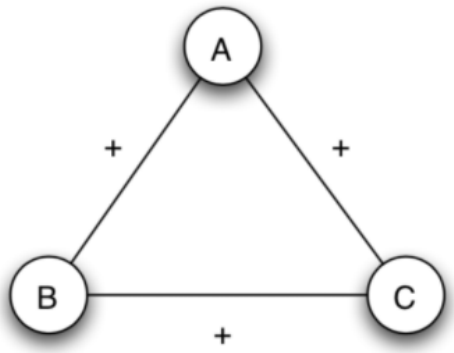
8. Signed networks

- in our discussion of networks thus far, we have generally viewed the relationships contained in these networks as having positive connotations
- the terminology of on-line social networks reflects a largely similar view, through its emphasis on the connections one forms with friends, fans, followers, and so forth
- but in most network settings, there are also negative effects at work. Some relations are friendly, but others are antagonistic or hostile
- how should we reason about the mix of **positive and negative relationships** that take place within a network?
- this part is taken from Chapter 12 (Positive and Negative Relationships) of book Networks, crowds and markets

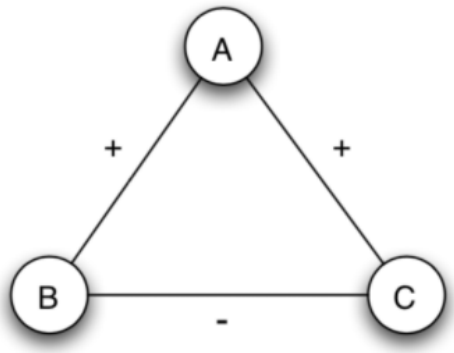
Structural balance

- suppose we have a social network on a set of people, in which everyone knows everyone else, no two people are indifferent to one another, or unaware of each other: a clique, or a **complete graph**
- we then label each edge with either + or -; a + label indicates that its two endpoints are friends, while a - label indicates that its two endpoints are enemies
- the model we're considering makes the most sense for a group of people small enough to have this level of mutual awareness, or for a setting such as **international relations**, in which the nodes are countries and every country has an official diplomatic position toward every other

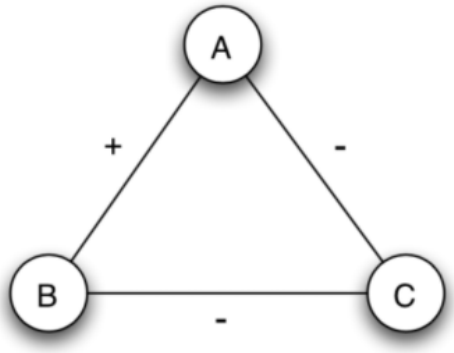
1. A friend of a friend will be a friend
2. A friend of an enemy will be an enemy
3. An enemy of a friend will be an enemy
4. An enemy of an enemy will be a friend



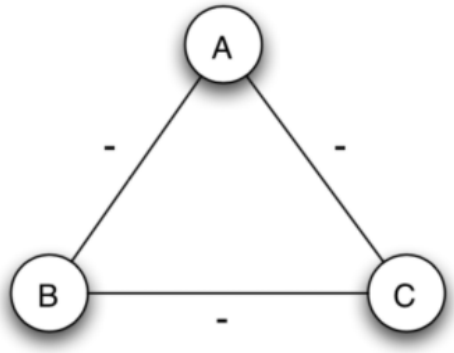
(a) *A, B, and C are mutual friends: balanced.*



(b) *A is friends with B and C, but they don't get along with each other: not balanced.*



(c) *A and B are friends with C as a mutual enemy: balanced.*



(d) *A, B, and C are mutual enemies: not balanced.*

Figure 5.1: Structural balance: Each labeled triangle must have 1 or 3 positive edges.

- the principles underlying structural balance are based on theories in social psychology dating back to the work of Heider in the 1940s, and generalized and extended to the language of graphs beginning with the work of Cartwright and Harary in the 1950s
- when we look at sets of three people at a time, certain configurations of +'s and -'s are **socially and psychologically more plausible** than others
- **balanced (or positive) triangles** have an odd number of positive signs (1 or 3), or the multiplication of the edge signs is positive
- **unbalanced (or negative) triangles** have an even number of positive signs (0 or 2), or the multiplication of the edge signs is negative
- a **network is balanced** if all triangles in it are balanced
- the argument of structural balance theorists is that because unbalanced triangles are sources of stress or psychological dissonance, people strive to minimize them in their personal relationships, and hence they will be less abundant in real social settings

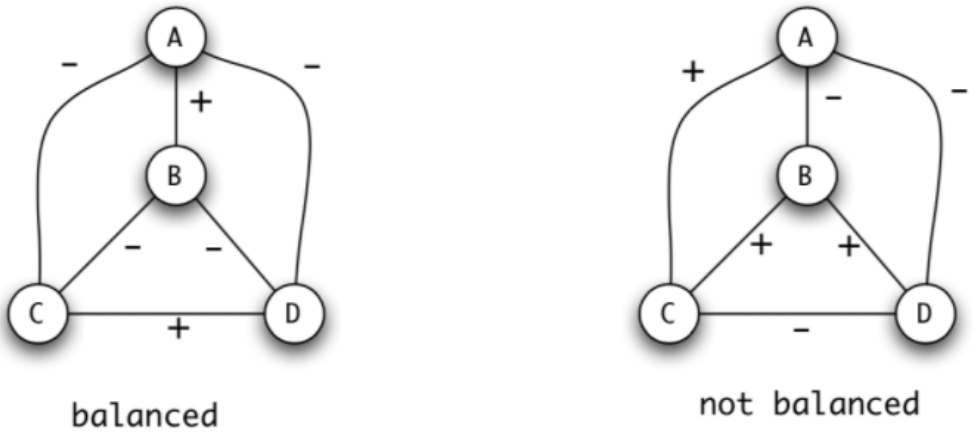


Figure 5.2: The labeled four-node complete graph on the left is balanced; the one on the right is not.

Characterizing the structure of balanced networks

We say that a signed complete network has the **mutual antagonism property** if:

> all pairs of nodes are friends, or else the nodes can be divided into two groups, X and Y, such that every pair of nodes in X like each other, every pair of nodes in Y like each other, and everyone in X is the enemy of everyone in Y.
>
> **Balance Theorem**: A signed complete graph is balanced if and only if it has the mutual antagonism property [Frank Harary, 1953]

- the Balance Theorem is not at all an obvious fact, nor should it be initially clear why it is true
- essentially, we're taking a purely **local** property, namely that all triangles are balanced, which applies to only three nodes at a time, and showing that it implies a strong **global** property: either everyone gets along, or the world is divided into two battling factions

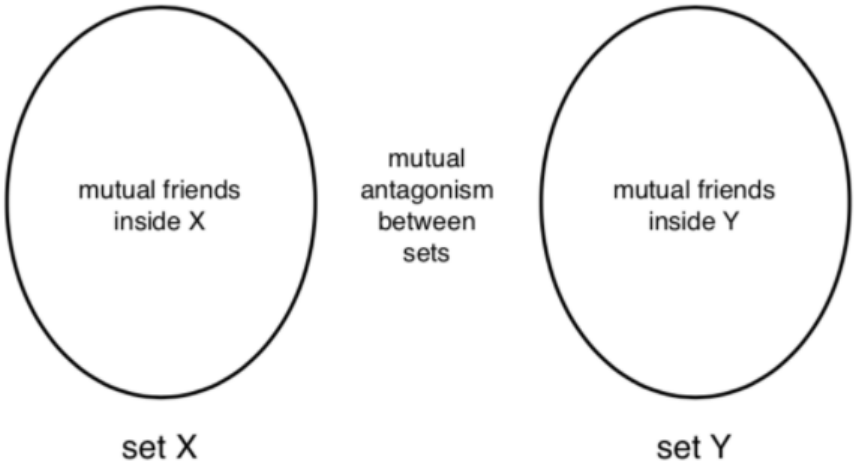


Figure 5.3: If a complete graph can be divided into two sets of mutual friends, with complete mutual antagonism between the two sets, then it is balanced. Furthermore, this is the only way for a complete graph to be balanced.

Proving the Balance Theorem

If a network has the mutual antagonism property then it is balanced

You can check that such a network is balanced: a triangle contained entirely in one group or the other has three + labels, and a triangle with two people in one group and one in the other has exactly one + label.

If a network is balanced then it has the mutual antagonism property

- let's pick any node in the network - we'll call it A - and consider things from A's perspective
- every other node is either a friend of A or an enemy of A
- thus, natural candidates to try for the sets X and Y would be to define X to be **A and all its friends**, and define Y to be **all the enemies of A**
- this is indeed a division of all the nodes, since every node is either a friend or an enemy of A

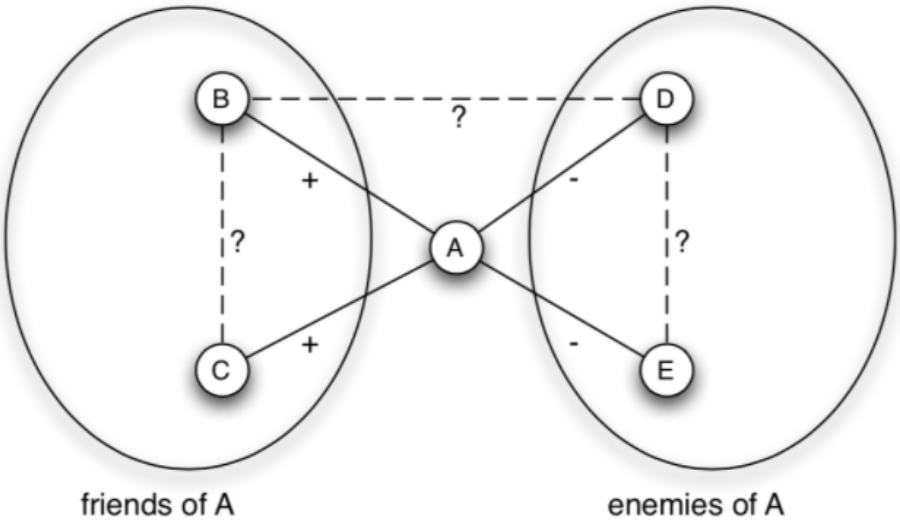


Figure 5.4: A schematic illustration of our analysis of balanced networks. (There may be other nodes not illustrated here.)

Approximately balanced networks

What if we only know that most triangles are balanced?

It turns out that the conditions of the theorem can be relaxed in a very natural way, allowing us to prove the following statement:

Let ϵ be any number such that $0 \leq \epsilon \leq 1$, and define $\delta = \sqrt[3]{\epsilon}$. If at least $1 - \epsilon$ of all triangles in a labeled complete graph are balanced, then either

1. there is a set consisting of at least $1 - \delta$ of the nodes in which at least $1 - \delta$ of all pairs are friends, or else
2. the nodes can be divided into two groups, X and Y, such that
   - at least $1 - \delta$ of the pairs in X like each other,
   - at least $1 - \delta$ of the pairs in Y like each other, and
   - at least $1 - \delta$ of the pairs with one end in X and the other end in Y are enemies.

For instance:

- if $\epsilon = 0$ then $\delta = 0$ and we get the original balance theorem
- if $\epsilon = 10^{-n}$ then $\delta = 10^{-n/3}$: for instance, if $\delta = 0.1$ (90% of the mutual antagonism property) the request is $\epsilon = 0.001$ (99.9% of the balance share)
- it is possible to show that this relationship between ϵ and δ is in fact essentially the best one can do

Structural balance in arbitrary networks

- let's consider the case of a social network that is not necessarily complete — that is, there are only edges between certain pairs of nodes, but each of these edges is still labeled with + or -
- so now there are three possible relations between each pair of nodes: a positive edge, indicating friendship; a negative edge, indicating enmity; or the absence of an edge, indicating that the two endpoints do not know each other

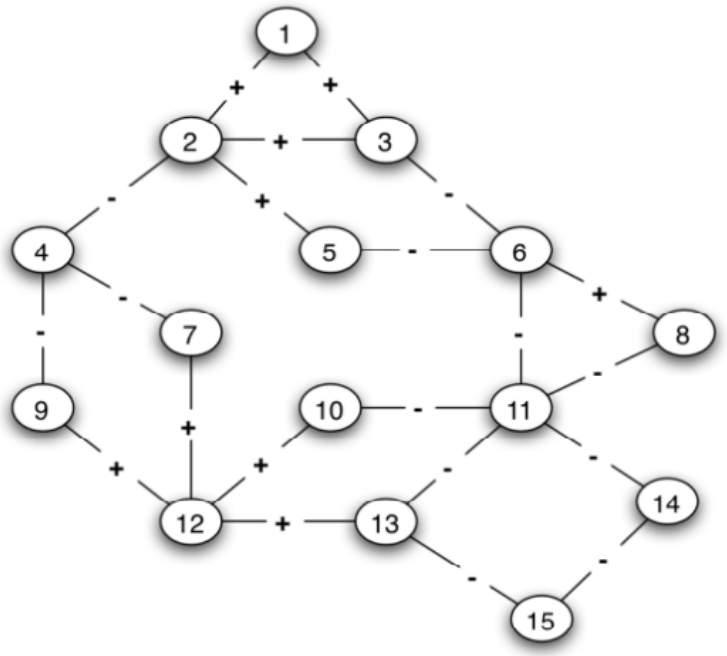
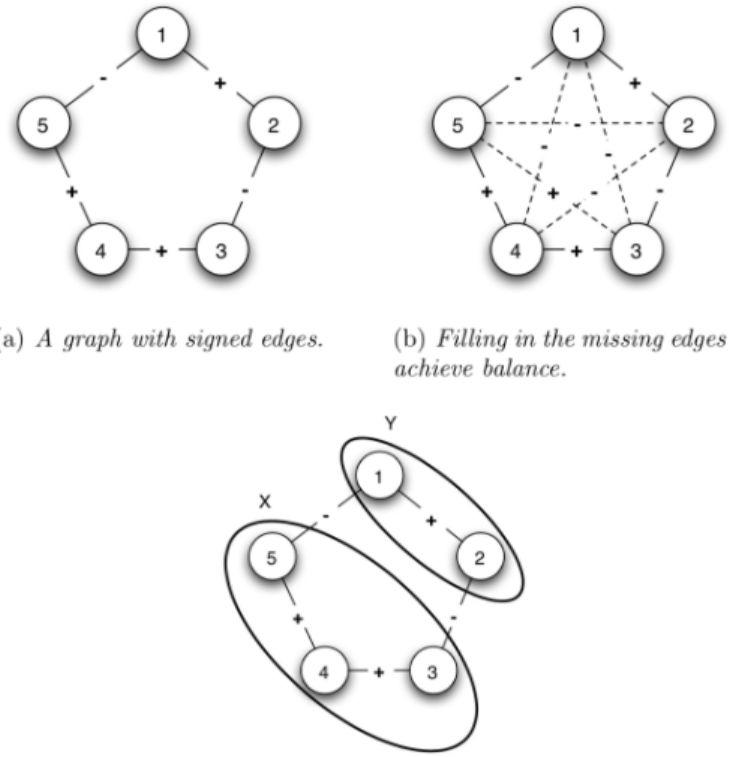


Figure 5.8: In graphs that are not complete, we can still define notions of structural balance when the edges that are present have positive or negative signs indicating friend or enemy relations.

Defining balance for general networks

- **a local definition**: we could then say that the graph is balanced if it possible to fill in all the missing labeled edges in such a way that the resulting signed complete graph is balanced
- **a global definition**: alternately, we could take a more global view, viewing structural balance as implying a division of the network into two mutually opposed sets of friends (to the extent that they know each other)



(a) *A graph with signed edges.*

(b) *Filling in the missing edges to achieve balance.*

(c) *Dividing the graph into two sets.*

Figure 5.9: There are two equivalent ways to define structural balance for general (non-complete) graphs. One definition asks whether it is possible to fill in the remaining edges so as to produce a signed complete graph that is balanced. The other definition asks whether it is possible to divide the nodes into two sets X and Y so that all edges inside X and inside Y are positive, and all edges between X and Y are negative.

These two ways of defining balance are equivalent: an arbitrary signed graph is balanced under the first definition if and only if it is balanced under the second definition.

- if a signed graph is balanced under the first definition, then after filling in all the missing edges appropriately, we have a signed complete graph to which we can apply the Balance Theorem. This gives us a division of the network into two sets X and Y that satisfies the properties of the second definition
- on the other hand, if a signed graph is balanced under the second definition, then after finding a division of the nodes into sets X and Y, we can fill in positive edges inside X and inside Y, and fill in negative edges between X and Y , and then we can check that all triangles will be balanced

Characterizing the structure of balanced general networks

A signed graph is balanced if and only if it contains no cycle with an odd number of negative edges

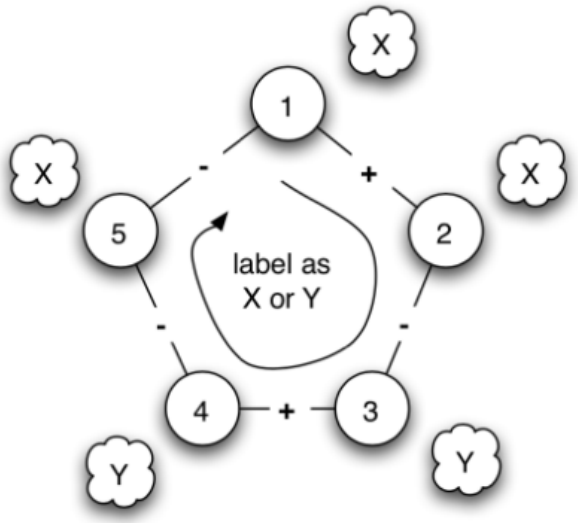


Figure 5.10: If a signed graph contains a cycle with an odd number of negative edges, then it is not balanced. Indeed, if we pick one of the nodes and try to place it in X, then following the set of friend/enemy relations around the cycle will produce a conflict by the time we get to the starting node.

(c) *Dividing the graph into two sets.*

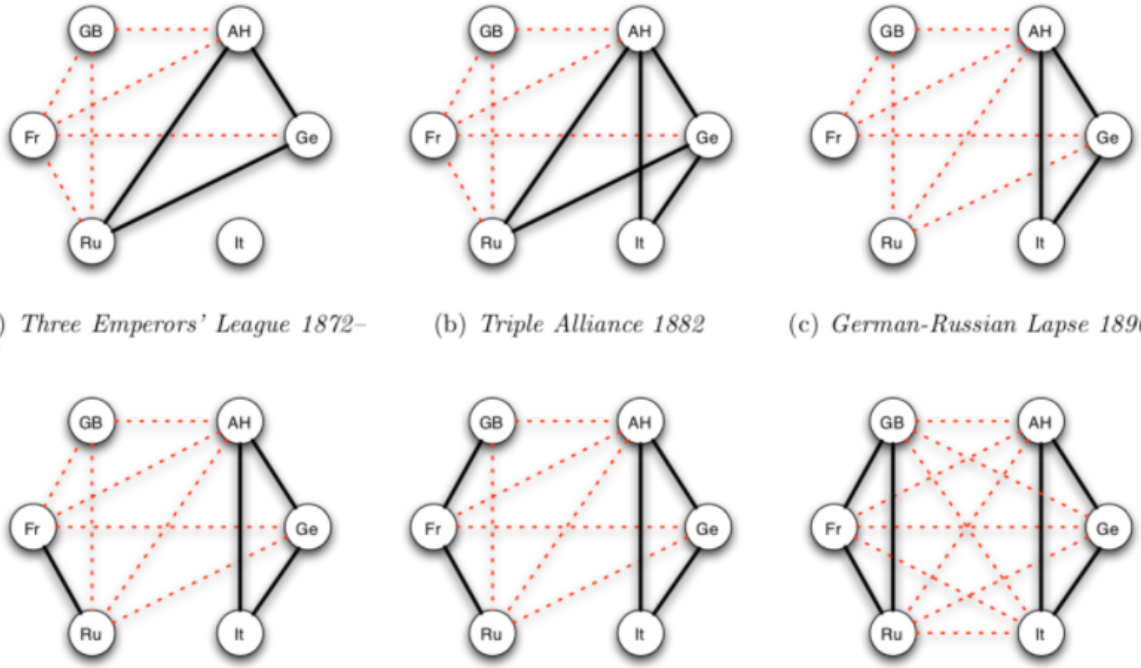
Approximately balanced general networks

There exists no shared notion of approximately balance in the case of general, non-complete networks, but different proposals exist including:

- **triangles**: returns the fraction of balanced triangles over all triangles
- **walk**: it quantifies the ratio of signed walks to unsigned walks of arbitrary length. Check the paper by Estrada for details
- **frustration**: A set E of edges is called minimum deletion set if deleting all edges in E results in a balanced graph. The frustration index equals the cardinality of a minimum deletion set. Note that the problem is NP-complete. See the work of Aref for details

Applications of structural balance - international politics

- **international politics** represents a setting in which it is natural to assume that a collection of nodes all have opinions (positive or negative) about one another
- here the nodes are nations, and + and - labels indicate alliances or animosity
- research in political science has shown that structural balance can sometimes provide an effective explanation for the behavior of nations during various international crises
- this also reinforces the fact that structural balance is **not necessarily a good thing**: since its global outcome is often two implacably opposed alliances, the search for balance in a system can sometimes be seen as a slide into a hard-to-resolve opposition between two sides



(a) *Three Emperors' League 1872–81*

(b) *Triple Alliance 1882*

(c) *German-Russian Lapse 1890*

(d) *French-Russian Alliance 1891–94*

(e) *Entente Cordiale 1904*

(f) *British Russian Alliance 1907*

Figure 5.5: The evolution of alliances in Europe, 1872-1907 (the nations GB, Fr, Ru, It, Ge, and AH are Great Britain, France, Russia, Italy, Germany, and Austria-Hungary respectively). Solid dark edges indicate friendship while dotted red edges indicate enmity. Note how the network slides into a balanced labeling — and into World War I. This figure and example are from Antal, Krapivsky, and Redner [20].

- the terminology of on-line social networks emphasis on the positive connections one forms with friends, fans, followers, and so forth
- but in most network settings, there are also negative effects at work
- a growing source for network data with both positive and negative edges comes from user communities on the Web where people can express positive or negative sentiments about each other
- for instance in the technology news site Slashdot the users can designate each other as a friend or a foe
- on on-line e-commerce sites such as eBay or vacantial rental marketplaces like airbnb, users can express trust or distrust of other users
- structural balance theory can be used to predict new links among users (and recommend products to users) not only based of positive relations but also on negative ones (for instance, why not link to the enemy of my enemy?)

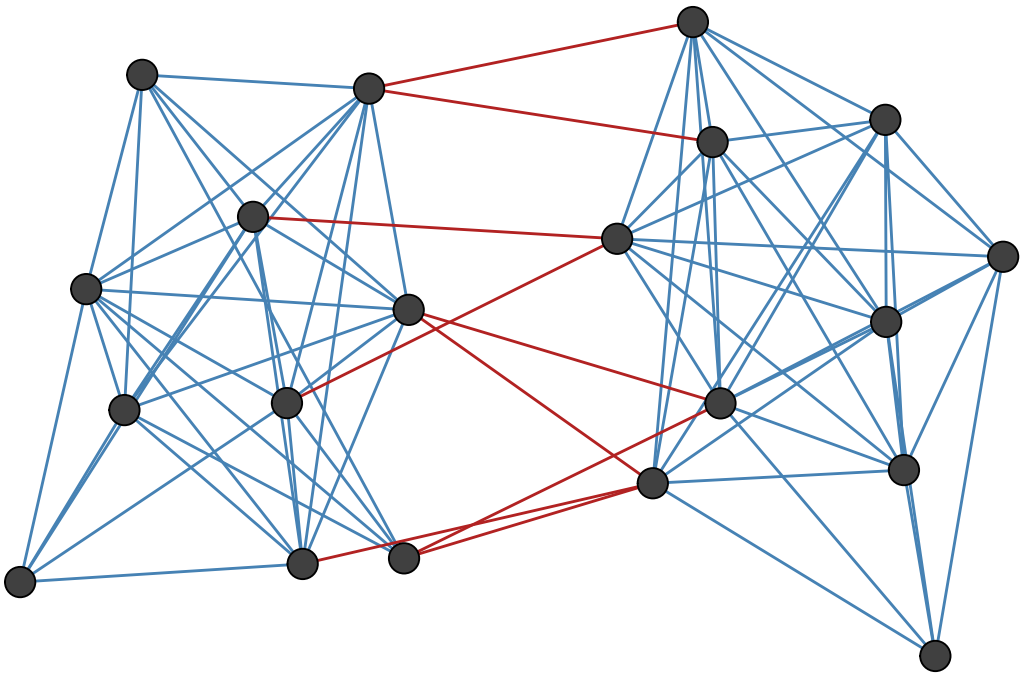
Example: Write code in dplyr to compute the relative frequency of balanced triangles in a signed network represented as a pair of data frames (nodes and edges).

Test the code on the signed graph with the followinge edges.

```
library(igraph)
library(signnet)
```

```
## Warning: package 'signnet' was built under R version 4.0.3
```

```
library(ggplot2)

# generate signed network with two sets of 10 nodes
g = sample_islands_signed(islands.n = 2, islands.size = 10,
                          islands.pin = 0.8,n.inter = 5)

# visualize network
ggsigned(g)
```



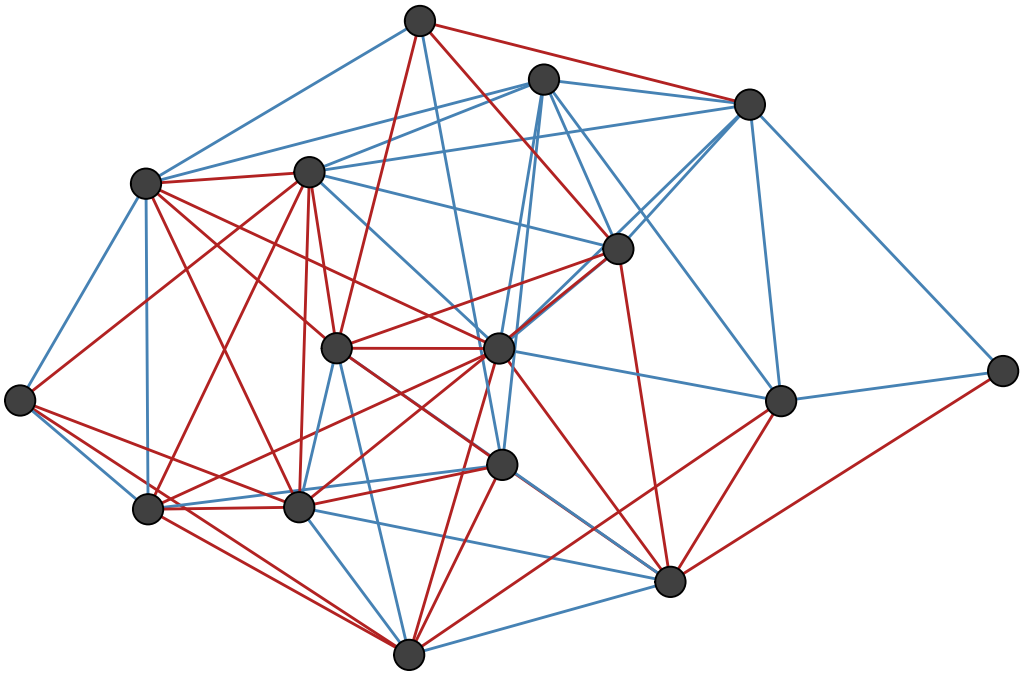
```
# count signed triangles
count_signed_triangles(g)
```

```
## +++ ++- +-- ---
## 114   0   3   0
```

```
# print signed tringles (P is the number of positive signs)
head(signed_triangles(g))
```

```
##      V1 V2 V3 P
## [1,]  7  1 13 3
## [2,]  7  2 19 3
## [3,]  7  3 30 3
## [4,]  7  4 33 3
## [5,]  7  5 35 3
## [6,]  7  6 36 3
```

```r
# use tribes dataset
data("tribes")
ggsigned(tribes)
```



```r
# not balanced
count_signed_triangles(tribes)
```

```
## +++ ++- +-- ---
##  19   2  40   7
```

```r
# degree of balance
balance_score(tribes, method = "triangles")
```

```
## [1] 0.8676471
```

```r
balance_score(tribes, method = "walk")
```

```
## [1] 0.3575761
```

```r
balance_score(tribes, method = "frustration")
```

```
## [1] 0.7586207
```

```r
balance_score(g, method = "triangles")
```

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

```r
balance_score(g, method = "walk")
```

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

```r
balance_score(g, method = "frustration")
```

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

9. Recommender Systems

"Many receive advice, only the wise profit from it." – Harper Lee

- the **recommendation problem** is to recommend items to users that they might not have found otherwise
- for example, consider a scenario of a content provider such as Netflix or Spotify. In such cases, users are able to easily provide feedback in the form of ratings (like/dislike or using a five-star rating system) or indirectly by using the content (viewing a movie or listening to a song)
- as another example, on online markets such as Amazon.com, the simple act of a user buying or browsing an item may be viewed as an endorsement for that item
- the basic idea of recommender systems is to utilize these various sources of data to infer customer interests and to suggest new items that correlate with the user profile
- the entity to which the recommendation is provided is referred to as the **user**, and the product being recommended is also referred to as an **item**
- therefore, recommendation analysis is often based on the previous interaction between users and items, because past interests and proclivities are often good indicators of future choices

Prediction version of problem

- the first approach to recommender systems is to predict the rating value for a user-item combination
- it is assumed that training data is available, indicating user preferences for items
- for m users and n items, this corresponds to an incomplete $m \times n$ matrix, where the specified (or **observed**) values are used for training
- the missing (or **unobserved**) values are predicted using this training model
- this problem is also referred to as the **matrix completion problem** because we have an incompletely specified matrix of values, and the remaining values are predicted by the learning algorithm
- this is a generalization of a prediction or classification problem; is these problems only one response variable is fully unknown and a number of independent variables are fully known

Ranking version of problem

- in practice, it is not necessary to predict the ratings of users for specific items in order to make recommendations to users
- rather, a merchant may wish to recommend the top-k items for a particular user, or determine the top-k users to target for a particular item

Goals of recommender systems

1. **Relevance**: The most obvious operational goal of a recommender system is to recommend items that are relevant to the user at hand. Users are more likely to consume items they find interesting. Although relevance is the primary operational goal of a recommender system, it is not sufficient in isolation.
2. **Novelty**: Recommender systems are truly helpful when the recommended item is something that the user has not seen in the past. For example, popular movies of a preferred genre would rarely be novel to the user. Repeated recommendation of popular items can also lead to reduction in sales diversity
3. **Serendipity**:
   - a related notion is that of serendipity, wherein the items recommended are somewhat unexpected, and therefore there is a modest element of lucky discovery, as opposed to obvious recommendations
   - serendipity is different from novelty in that the recommendations are truly surprising to the user, rather than simply something they did not know about before
   - increasing serendipity often has long-term and strategic benefits to the merchant because of the possibility of discovering entirely new areas of interest. On the other hand, algorithms that provide serendipitous recommendations often tend to recommend irrelevant items
4. **Diversity**: Recommender systems typically suggest a list of top-k items. When all these recommended items are very similar, it increases the risk that the user might not like any of these items. On the other

hand, when the recommended list contains items of different types, there is a greater chance that the user might like at least one of these items

5. **Explanation**: Finally, providing the user an explanation for why a particular item is recommended is often useful. For example, in the case of Netflix, recommendations are provided along with previously watched movies

Models of recommender systems

- **collaborative filtering models** leverage information on the user-item interactions, such as ratings or buying behavior
- **content-based models** leverage the attribute information about the users and items such as textual profiles or relevant keywords
- **knowledge-based models** leverage external knowledge bases and constraints
- some recommender systems combine these different aspects to create **hybrid systems**. Hybrid systems can combine the strengths of various types of recommender systems to create techniques that can perform more robustly in a wide variety of settings

Collaborative filtering models

- the basic idea of collaborative filtering methods is that these unspecified ratings can be imputed because the observed ratings are often highly correlated across various users and items
- for example, consider two users named Alice and Bob, who have very similar tastes, which means that the specified ratings by both users are very similar. In such cases, it is very likely that the ratings in which only one of them has specified a value (hence unspecified for the other), are also likely to be similar

Neighborhood-based collaborative filtering methods

- these were among the earliest collaborative filtering algorithms, in which the ratings of user-item combinations are predicted on the basis of their neighborhoods
- it can be based on users or based on items
- in **user-based** collaborative filtering, a similarity relation is computed among pairs of users, for instance using the cosine similarity among pairs of rows of the user-item matrix
- given a target user A, the k most similar users (neighbors) to A can be used to make rating predictions for A
- if A has not specified a rating for an item T, but the neighbors of A have specified this preference, then the unspecified rating of A for T is likely to be similar to the weighted mean of the rating of neighbors for T
- in **item-based** collaborative filtering, a similarity relation is computed among pairs of items, for instance using the cosine similarity among pairs of columns of the user-item matrix
- given a target item T, the k most similar items (neighbors) to T can be used to make rating predictions for the rating of a user A for T
- if A has not specified a rating for the item T, but it has specified the rating for the neighbors items of T, we can leverage these ratings to infer the unspecified rating of A for T

Model-based collaborative filtering methods

- the advantages of memory-based techniques are that they are simple to implement and the resulting recommendations are often easy to explain
- on the other hand, memory-based algorithms do not work very well with sparse ratings matrices
- in **model-based methods**, machine learning and data mining methods are used in the context of predictive models
- in cases where the model is parameterized, the parameters of this model are learned within the context of an optimization framework

- some examples of such model-based methods include decision trees, rule-based models, Bayesian methods and latent factor models
- many of these methods, such as latent factor models, have a high level of coverage even for sparse ratings matrices

Content-based models

- in **content-based recommender systems**, the descriptive attributes of items are used to make recommendations: the term "content" refers to these descriptions
- for example, consider a situation where we want to infer the rating of a user A for item T but we do not have access to the ratings of other users for T, maybe because T is new in the system; therefore, collaborative filtering methods are ruled out
- we can use the specified ratings of A for items similar to T, that is that share the same descriptive features (for instance, a movie of the same genre) to infer the rating of A for T (if I hate horror movies don't recommend me Dario Argento's movies)
- in content-based methods, the item descriptions, which are labeled with ratings, are used as training data to create a user-specific classification or regression modeling problem
- for each user, the training documents correspond to the descriptions of the items they have bought or rated
- the class (or dependent) variable corresponds to the specified ratings or buying behavior
- these training documents are used to create a classification or regression model, which is specific to the user at hand (or active user)
- this user-specific model is used to predict whether the corresponding individual will like an item for which their rating or buying behavior is unknown.
- content-based methods have some advantages in making recommendations for **new items**, when sufficient rating data are not available for that item. This is because other items with similar attributes might have been rated by the active user.
- in many cases, content-based methods provide **obvious recommendations**. For example, if a user has never consumed an item with a particular set of keywords, such an item has no chance of being recommended. This is because the constructed model is specific to the user at hand, and the community knowledge from similar users is not leveraged
- even though content-based methods are effective at providing recommendations for new items, they are not effective at providing recommendations for **new users**. This is because the training model for the target user needs to use the history of the user ratings.
- knowledge-based systems are unique in that they allow the users to explicitly specify what they want and then use a domain-specific knowledge base to make item recommendations to users
- they can be based on constraints or based on cases
- in **constraint-based systems**, users typically specify requirements or constraints, for instance I want to buy a house with 2 baths and at least 100 square meters large
- domain-specific rules are used to match the user requirements to item attributes
- in **case-based systems**, specific cases are specified by the user as targets or anchor points, for instance I want to buy a house similar to the one at the following address
- similarity metrics are defined on the item attributes to retrieve similar items to these cases
- the similarity metrics are often carefully defined in a domain-specific way. Therefore, the similarity metrics form the domain knowledge that is used in such systems
- knowledge-based recommender systems are particularly useful in the context of items that are not purchased very often, hence few rating information is available. Examples include items such as real estate or expensive luxury goods
- this problem is also encountered in the context of the **cold-start problem**, when the system is new and not enough rating information is available
- knowledge-based recommender systems are also useful when a user is new to the system, hence both collaborative filtering and content-based models are not applicable
- it is noteworthy that both knowledge-based and content-based systems depend significantly on the attributes of the items

- the main difference is that content-based systems learn from past user behavior, whereas knowledge-based recommendation systems recommend based on active user specification of their needs and interests
- because of their use of content-attributes, knowledge-based systems inherit some of the same disadvantages as content-based systems, in particular the recommendations given by these systems can be sometimes obvious.

10. Similarity and heterogeneity

Another central concept in social network analysis is that of similarity between vertices.

In what ways can two actors in a network be similar, and how can we quantify this similarity?

Similarity is relevant in at least two important applications:

- **recommender systems**, where the goal is to suggest to a user new products to buy according to what similar users bought or liked, and
- **link prediction**, where the task is to predict new connections for a user on social networks according to the connections of similar users.

Similarity

Similarity agrees with the following thesis:

Two nodes are similar to each other if they share many neighbors.

For instance, two customers are similar if they bought the same products (Amazon), or view the same movies (Netflix), or listen to the same music (Spotify).

- let $A = (a_{i,j})$ be the adjacency matrix of a possibly weighted graph
- the general idea is to associate each node i with a ith row (or column) of the adjacency matrix (a vector of size the number of nodes of the graph)
- the similarity (or dissimilarity) of two nodes i and j is then measured using a similarity (or distance) function among the associated vectors

Let us focus on unweighted undirected graphs with n nodes.

We denote with

$$k_i = \sum_k A_{i,k}$$

the degree of node i and with

$$n_{i,j} = \sum_k A_{i,k} A_{j,k}$$

the number of neighbors that nodes i and j have in common.

Moreover, A_i is the i-th row of A, a boolean (0-1) vector of length n.

Cosine similarity

We are going to describe a pool of similarity measures.

The first one is **cosine similarity**:

The similarity $\sigma_{i,j}$ of nodes i and j is the cosine of the angle between vectors A_i and A_j.

$$\sigma_{i,j} = \cos(A_i, A_j) = \frac{A_i \cdot A_j}{\|A_i\|\|A_j\|} = \frac{\sum_k A_{i,k} A_{j,k}}{\sqrt{\sum_k A_{i,k}^2}\sqrt{\sum_k A_{j,k}^2}}$$

The measure runs from 0 (orthogonal vectors or maximum dissimilarity) to 1 (parallel vectors or maximum similarity).

Since the involved vectors are 0-1 vectors, we have that:

$$\sigma_{i,j} = \frac{n_{i,j}}{\sqrt{k_i k_j}}$$

That is, cosine similarity between i and j is the number of neighbors shared by i and j divided by the geometric mean of their degrees.

Pearson similarity

An alternative similarity measure is **Pearson correlation coefficient**:

> The similarity $\sigma_{i,j}$ of nodes i and j is the correlation coefficient between vectors A_i and A_j

$$\sigma_{i,j} = \frac{cov(A_i, A_j)}{sd(A_i)\, sd(A_j)} = \frac{\sum_k (A_{i,k} - \langle A_i \rangle)(A_{j,k} - \langle A_j \rangle)}{\sqrt{\sum_k (A_{i,k} - \langle A_i \rangle)^2} \sqrt{\sum_k (A_{j,k} - \langle A_j \rangle)^2}}$$

The measure runs from -1 (maximum negative correlation or maximum dissimilarity) to 1 (maximum positive correlation or maximum similarity). Notice that values close to 0 indicate no correlation, hence neither similarity nor dissimilarity.

Again, since the involved vectors are 0-1 vectors, it is not difficult to see that the numerator of the correlation coefficient, that is the co-variance between vectors A_i and A_j is:

$$cov(A_i, A_j) = n_{i,j} - \frac{k_i k_j}{n}$$

Notice that $k_i k_j / n$ is the expected number of common neighbors between i and j if they would choose their neighbors at random: the probability that a random neighbor of i is also a neighbor of j is k_j/n, hence the expected number of common neighbors between i and j is $k_i k_j / n$.

Hence a positive covariance, or a similarity between i and j, holds when i and j share more neighbors than we would expect by change, while a negative covariance, or a dissimilarity between i and j happens when i and j have less neighbors in common than we would expect by change.

The user defined function `similarity` computes both cosine and Pearson similarity on rows or columns of a matrix:

```
similarity = function(g, type = "cosine", mode = "col" ) {
  A = as_adjacency_matrix(g, sparse = FALSE)
  if (mode == "row") {A = t(A)}
  if (type == "cosine") {
    euclidean = function(x) {sqrt(x %*% x)}
    d = apply(A, 2, euclidean)
    D = diag(1/d)
    S = D %*% t(A) %*% A %*% D
  }
  if (type == "pearson") {
    S = cor(A)
  }
  return(S)
}
```

Global similarity

- all similarity measures so far are local: they count the number of common neighbors, or number of paths of length two, between two nodes
- however, two nodes might be similar in a more general way, using longer paths, or indirect similarity
- two nodes may have few or none neighbors in common, but they still may be similar in an indirect, global way
- paths between nodes, of any length, are hints of similarity; nevertheless, the shorter the path, the stronger the contribution to similarity.

The recursive thesis of global similarity is the following:

> Two nodes are similar if one has a neighbor that is similar to the other.

Mathematically, the recursive thesis of global similarity can be written as follows:

$$\sigma_{i,j} = \alpha \sum_k A_{i,k}\sigma_{k,j} + \delta_{i,j}$$

or in matrix form as:

$$\sigma = \alpha A\sigma + I$$

Evaluating the expression by iterating starting from $\sigma^0 = 0$ we get:

$$\begin{array}{rcl} \sigma^{(1)} & = & I \\ \sigma^{(2)} & = & \alpha A + I \\ \sigma^{(3)} & = & \alpha^2 A^2 + \alpha A + I \\ & \cdots & \end{array}$$

Hence, $\sigma^{(k)}$ counts the contribution of paths of length less than k and the contribution of a path of length k is weighted by the attenuation factor α^k, as soon as $\alpha < 1$.

Notice that the contribution of the identity matrix I is that each node is similar to itself; this is necessary to propagate similarity through network using the network topology represented by the adjacency matrix A.

If follows that the similarity matrix σ is the following:

$$\sigma = \sum_{k=0}^{\infty} (\alpha A)^k = (I - \alpha A)^{-1}$$

Variants of global similarity

As defined, the metric tends to give high similarity to nodes with high degree. However, who is to say that two hermits are similar in an interesting way?

If we wish, we can remove the bias in favor of hubs by dividing by node degree, hence:

$$\sigma_{i,j} = \frac{\alpha}{k_i} \sum_k A_{i,k}\sigma_{k,j} + \delta_{i,j}$$

and hence

$$\sigma = (D - \alpha A)^{-1}$$

- another variant allows for cases where the term $\delta_{i,j}$ is not simply diagonal, but includes off-diagonal items
- this would allow us to specify explicitly that particular pairs of nodes are similar based on some external (non-network) information we have at disposal

Global similarity and Katz centrality

- this solution is reminiscent of **Katz centrality**: $x = \beta(I - \alpha A)^{-1}$
- indeed, Katz centrality of node i is exactly the sum of similarities of i with all other nodes
- hence, a node is central in Katz view if it is similar to many other vertices
- as with Katz centrality, the parameter α must be chosen less then $1/\rho(A)$, with $\rho(A)$ the spectral radius of A
- in the same way, the degree-controlled version of the similarity is reminiscent the **PageRank centrality**

Example: Compute and visualize the global similarity on the Zachary graph

```
library(tidyverse)
library(igraph)
library(ggraph)

g = make_graph("Zachary")

A = as_adjacency_matrix(g)
alpha = 0.85 / max(abs(eigen(A)$values))
I = diag(1, vcount(g))
S = solve(I - alpha * A)
S = S - diag(diag(S))

s = graph_from_adjacency_matrix(S, mode = "undirected", weighted = TRUE)
V(s)$name = 1:vcount(s)
sim = as_tibble(igraph::as_data_frame(s, what = "edges"))

ggraph(sim, layout = "circle") +
  geom_edge_link(aes(edge_alpha = weight, filter = (weight > quantile(weight, 0.9)))) +
  geom_node_point() +
  geom_node_text(aes(label = name, x = x * 1.05, y = y * 1.05))
```

Heterogeneity

We have learnt how to exploit the network topology to gauge the similarity (or dissimilarity) of pairs of networks.

This opens up the possibility of defining a measure of **heterogeneity** for a node in terms of the dissimilarity of its neighbors:

> A node is heterogeneous if it has dissimilar neighbors.

For instance:

- an individual is heterogeneous if her friends are ill-matched
- a scholar is heterogeneous if his papers are interdisciplinary

We will work on weighted undirected graphs. Let $A = (a_{i,j})$ be the adjacency matrix of a weighted undirected graph. Let i be a node with positive degree d_i. We set:

$$p_{i,j} = \frac{a_{i,j}}{\sum_k a_{i,k}} = \frac{a_{i,j}}{d_i}$$

Notice that $0 \leq p_{i,j} \leq 1$ and $\sum_j p_{i,j} = 1$, hence $(p_{i,1}, \ldots, p_{i,n})$ is a probability distribution for every node i.

Shannon entropy

We will work out the math for a general probability distribution $p = (p_1, \ldots, p_n)$, with $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$.

A measure of heterogeneity of p is **Shannon entropy**:

$$H(p) = -\sum_i p_i \log p_i$$

where $p_i \log p_i = 0$ if $p_i = 0$.

Simpson diversity

Another measure of heterogeneity of p is **Simpson diversity**:

$$S(p) = \sum_{i \neq j} p_i p_j = \sum_{i,j} p_i p_j - \sum_i p_i^2 = 1 - \sum_i p_i^2$$

where in the latter we have used the fact that

$$\sum_{i,j} p_i p_j = \sum_i p_i \sum_j p_j = \sum_i p_i 1 = 1$$

Minimum and maximum

- Both measures are **minimized** when p is fully concentrated on some element k, that is $p_k = 1$ and $p_i = 0$ for every $i \neq k$. In such case
$$H(p) = S(p) = 0$$
- Both measures are **maximized** when p is evenly distributed among the elements, that is $p_i = 1/n$ for every i. In such case
$$H(p) = \log n$$
and
$$S(p) = 1 - 1/n$$
Notice that in this case both measures grow with n, although Simpson diversity is limited by 1.

Rao quadratic entropy

If we have information about pairwise distance (dissimilarity) $d_{i,j}$ of elements i and j, then another measure of heterogeneity is **Rao quadratic entropy**:

$$R(p) = \sum_{i,j} p_i \, p_j \, d_{i,j}$$

There are two components in this definition of heterogeneity:

1. the evenness of the distribution p
2. the distances d among elements

To isolate both components of the definition, let us first assume that the distribution p is uniform, that is $p_i = 1/n$ for every i. Hence:

$$R(p) = \sum_{i,j} p_i \, p_j \, d_{i,j} = \frac{1}{n^2} \sum_{i,j} d_{i,j}$$

is the arithmetic mean distance among elements in p. Hence, the more distant are the elements in p among each other, the more heterogeneous is p.

Let us now assume the simplest distance among nodes: $d_{i,j} = 1$ for $i \neq j$ and $d_{i,i} = 0$. In this case:

$$R(p) = \sum_{i,j} p_i \, p_j \, d_{i,j} = \sum_{i \neq j} p_i \, p_j = S(p)$$

Hence in this case the more uniform is p, the more heterogeneous is p.

It follows that, in general:

- $R(p)$ is large, hence p is heterogeneous, when p evenly distributes its probability among dissimilar elements
- on the contrary, p is homogeneous when it concentrates its probability on similar elements
- If we go back to heterogeneity of nodes of a graph and apply $R(p)$ as a measure, we have that a node is heterogeneous when it evenly distributes its strength among dissimilar neighbors
- a node is homogeneous when it concentrates its strength on similar neighbors

```
shannon = function(p) {
  x = p * log2(p)
  x = replace(x, is.nan(x), 0)
  return(-sum(x))
}

simpson = function(p) {
  x = 1 - sum(p * p)
  return(x)
}

rao = function(p, D) {
  x = diag(p) %*% D %*% diag(p)
  return(sum(c(x)))
}


heterogeneity = function(g, D, mode = "col") {
  A = as_adjacency_matrix(g, attr = "weight", sparse = FALSE)
  if (mode == "col") {
    A = A %*% diag(1/colSums(A))
    dim = 2
  } else {
    A = diag(1/rowSums(A)) %*% A
    dim = 1
  }
  return(list(shannon = apply(A, dim, shannon),
              simpson = apply(A, dim, simpson),
              rao = apply(A, dim, rao, D)))
}
```

11. Community detection and clustering

> Loosely stated, community detection is the problem of finding the natural divisions of a network into groups of vertices, called communities, such that there are many edges within groups and few edges between groups.

For instance:

- in social networks, communities represent groups of individuals that are socially tight
- in a citation network among academic journals, communities might correspond to topics or disciplines
- communities of nodes in a web graph might indicate groups of related web pages
- communities of nodes in a metabolic network might indicate functional units within the network

Modularity

- our goal is to find a measure that quantifies how many edges lie **within** groups in our network relative to the number of such edges expected on the basis of chance
- a good division of nodes into communities is one that maximizes such a measure
- equivalently, we want a measure that quantifies how many edges lie **between** groups in our network relative to the expected number of such links
- a good division of nodes into communities is one that minimizes such a measure
- we will concentrate on the former measure of **modularity** of a network

Let us focus on undirected multi-graphs, that is, graphs that allow self-edges (edges involving the same node) and multi-edges (more than one simple edge between two vertices).

> A measure of modularity of a network is the number of edges that run between vertices of the same community minus the number of such edges we would expect to find if edges were positioned at random while preserving the vertex degrees (**configuration model**).

Let us denote by c_i the community of vertex i and $\delta(c_i, c_j) = 1$ if $c_i = c_j$ and $\delta(c_i, c_j) = 0$ otherwise.

Hence, the **actual** number of edges that run between vertices of the same group is:

$$\sum_{(i,j)\in E} \delta(c_i, c_j) = \frac{1}{2} \sum_{i,j} A_{i,j} \delta(c_i, c_j)$$

where E is the set of edges of the graph and $A_{i,j}$ is the actual number of edges between i and j, which is zero or more (notice that each undirected edge is represented by two pairs in the second sum, hence the factor one-half).

The **expected number** of edges that run between vertices of the same group is:

$$\frac{1}{2} \sum_{i,j} \frac{k_i k_j}{2m} \delta(c_i, c_j)$$

where k_i and k_j are the (weighted) degrees of i and j, while m is the number of edges of the graph.

Notice that $k_i k_j / 2m$ is the expected number of edges between vertices i and j in the configuration model assumption.

Indeed, consider a particular edge attached to vertex i:

- the probability that this edge goes to node j is $k_j/2m$, since the number of edges attached to j is k_j and the total number of edge ends in the network is $2m$ (the sum of all node degrees)
- since node i has k_i edges attached to it, the expected number of edges between i and j is $k_i k_j / 2m$

The difference between the actual and expected number of edges connecting nodes of the same group, expressed as a fraction with respect to the total number of edges m, is called **modularity**, and given by:

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

where:

$$B_{i,j} = A_{i,j} - \frac{k_i k_j}{2m}$$

and B is called the **modularity matrix**.

The modularity Q takes positive values if there are more edges between same-group vertices than expected, and negative values if there are less.

Our goal is to find the partition of network nodes into communities such that the modularity of the division is maximum.

Unfortunately, this is a **computationally hard problem**.

Indeed, the number of ways a network of n nodes can be divided into two groups of n_1 and n_2 nodes, with $n_1 + n_2 = n$ is:

$$\binom{n}{n_1} = \binom{n}{n_2} = \frac{n!}{n_1!\, n_2!} \sim \frac{2^{n+1}}{\sqrt{n}}$$

when $n_1 = n_2 = n/2$, which is exponential in n.

Instead, therefore, we turn to **heuristic algorithms**, algorithms that attempt to maximize the modularity in an intelligent way that gives reasonably good results in a quick time.

Spectral modularity maximization

This method tries to maximize the modularity of a partition by exploiting the spectral properties of the modularity matrix.

Recall that modularity is defined as:

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

Notice that the modularity matrix B is symmetric and all rows and all columns of B sum to 0. Indeed:

$$\sum_j B_{i,j} = \sum_j (A_{i,j} - \frac{k_i k_j}{2m}) = k_i - \frac{k_i}{2m} 2m = 0$$

Let us consider the division of a network into just two parts, namely group 1 and group 2 (the more general case can be treated with repeated bisection).

We represent the assignment of node i to a group with the variable s_i such that $s_i = 1$ if i belongs to group 1 and $s_i = -1$ if i belongs to group 2. It follows that:

$$\delta(c_i, c_j) = \frac{1}{2}(s_i s_j + 1)$$

Substituting this expression in the modularity formula we have:

$$Q = \frac{1}{4m} \sum_{i,j} B_{i,j}(s_i s_j + 1) = \frac{1}{4m} \sum_{i,j} B_{i,j} s_i s_j$$

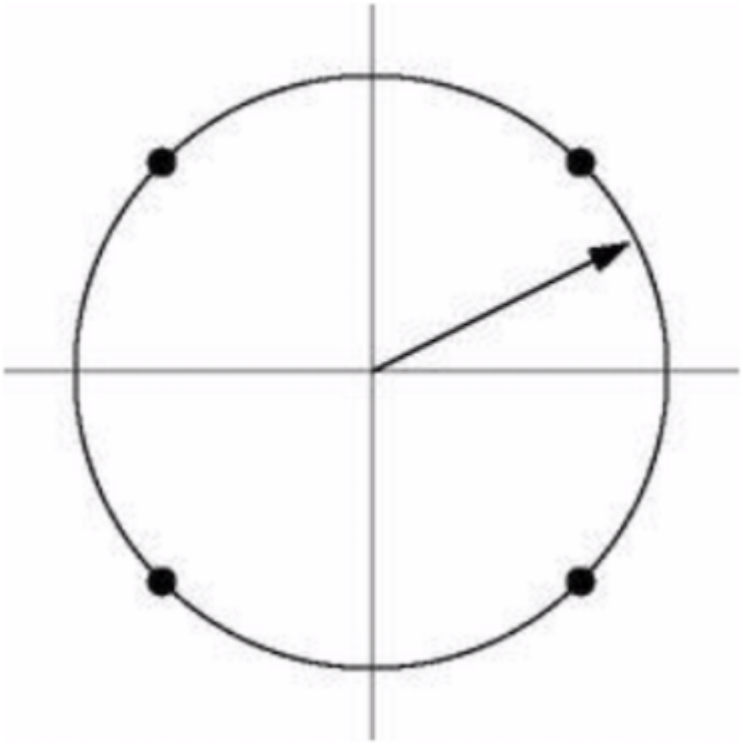
where we have used the fact that $\sum_{i,j} B_{i,j} = 0$.

In matrix form we have:

$$Q = \frac{1}{4m} sBs$$

- we wish to find the division of a given network, that is the value of s, that maximizes the modularity Q
- the elements of s are constrained to take integer values 1 or -1, so that the vector always points to one of the corners of an n-dimensional hypercube
- unfortunately, this optimization problem is a hard one, but it can be tackled approximately by a relaxation method
- we relax the constraint that s must point to a corner of the hypercube and allow it to point in any direction, though keeping its length the same:

$$ss = \sum_i s_i^2 = n$$



We maximize the modularity equation by differentiating, imposing the constraint with a single Lagrange multiplier β:

$$\frac{\partial}{\partial s_i} \left[\sum_{j,k} B_{j,k} s_j s_k + \beta(n - \sum_j s_j^2) \right] = 0$$

That is:

$$\sum_j B_{i,j}s_j + \sum_j B_{j,i}s_j - 2\beta s_i = 2\sum_j B_{i,j}s_j - 2\beta s_i = 0$$

which is:

$$\sum_j B_{i,j}s_j = \beta s_i$$

or in matrix notation:

$$Bs = \beta s$$

Hence the solution s is an eigenvector of the modularity matrix.

Therefore the modularity is given by:

$$Q = \frac{1}{4m}sBs = \frac{1}{4m}\beta ss = \frac{n}{4m}\beta$$

which leads us to the conclusion that for maximum modularity we have to choose s as the eigenvector u corresponding to the largest (positive) eigenvalue of the modularity matrix.

We typically cannot in fact choose $s = u$, since the elements of s are subject to the constraint $s_i \in \{+1, -1\}$. But we do the best we can and choose it as close to u as possible, hence:

- $s_i = +1$ if $u_i > 0$
- $s_i = -1$ if $u_i < 0$
- either value of s_i is good if $u_i = 0$

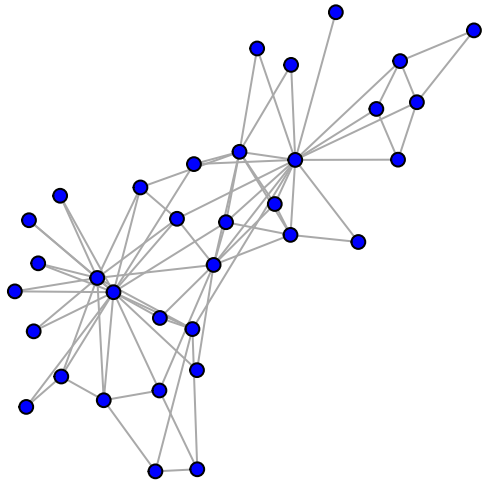
So we are led the following algorithm:

1. we calculate the eigenvector of the modularity matrix corresponding to the largest (most positive) eigenvalue
2. then we assign vertices to communities according to the signs of the vector elements, positive signs in one group and negative signs in the other
3. if all elements in the eigenvector are of the same sign that means that the network has no underlying community structure

Code

- let's work with the Zachary network represents the pattern of friendships between members of a karate club at a North American university
- the network is determined by direct observation of the club's members by the experimenter over a period of about two years
- the network is interesting because during the period of observation a dispute arose among the members of the club over whether to raise the club's fees
- as a result the club eventually split into two parts, of 18 and 16 members respectively, the latter departing to form their own club

```
library(igraph)
g = make_graph("Zachary")
coords = layout_with_fr(g)
plot(g, layout=coords,
     vertex.label=NA, vertex.size=6, vertex.color = "blue")
```



```
# spectral community detection
c = cluster_leading_eigen(g)

# modularity measure
modularity(c)
```

```
## [1] 0.3934089
```

```
# number of communities
length(c)
```

```
## [1] 4
```

```
# size of communities
sizes(c)
```

```
## Community sizes
##  1  2  3  4
##  7 12  9  6
```

```
# memberships of nodes
membership(c)
```

```
##  [1] 1 3 3 3 1 1 1 3 2 2 1 1 3 3 2 2 1 3 2 3 2 3 2 4 4 4 2 4 4 2 2 4 2 2
```

```
# inter-community crossing edges
crossing(c, g)
```

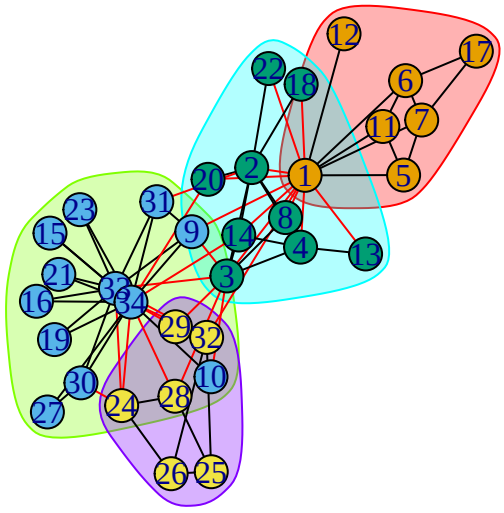
```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
## [25] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [61]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
## [73] FALSE FALSE FALSE  TRUE  TRUE FALSE
```

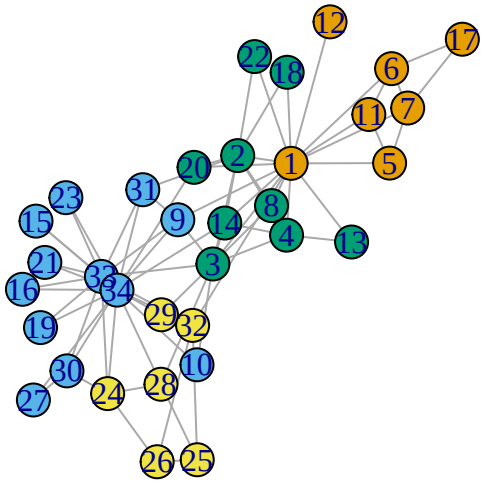
```
# modularity matrix
B = modularity_matrix(g, membership(c))
# first row
round(B[1,],2)
```

```
##  [1] -1.64  0.08 -0.03  0.38  0.69  0.59  0.59  0.59  0.49 -0.21  0.69  0.90
## [13]  0.79  0.49 -0.21 -0.21 -0.21  0.79 -0.21  0.69 -0.21  0.79 -0.21 -0.51
## [25] -0.31 -0.31 -0.21 -0.41 -0.31 -0.41 -0.41  0.38 -1.23 -1.74
```

```
# plot communities with shaded regions
plot(c, g, layout=coords)
```




```
# plot communities without shaded regions
plot(g, vertex.color=membership(c), layout=coords)
```



Greedy optimization of modularity

- `cluster_fast_greedy` starts out with each vertex in our network in a one-vertex group of its own
- then it successively amalgamate groups in pairs, choosing at each step the pair whose amalgamation gives the biggest increase in modularity, or the smallest decrease if no choice gives an increase
- eventually all vertices are amalgamated into a single large community and the algorithm ends
- then we go back over the states through which the network passed during the course of the algorithm and select the one with the highest value of the modularity

Edge betweenness

- `cluster_edge_betweenness` looks for the edges that lie between communities
- if we can find and remove these edges, we will be left with just the isolated communities
- to identify edges between communities one common approach is to use edge betweenness centrality, which counts the number of geodesic paths that run along edges
- a less expensive alternative would be to look for edges that belong to an unusually small number of short loops

Random walks

- `cluster_walktrap` is an approach based on random walks
- the general idea is that if you perform random walks on the graph, then the walks are more likely to stay within the same community because there are only a few edges that lead outside a given community
- this method runs short random walks of 3-4-5 steps (depending on one of its parameters) and uses the results of these random walks to merge separate communities in a bottom-up manner
- tou can use the modularity score to select where to cut the dendrogram

Statistical meachanics

- `cluster_spinglass` is an approach from statistical physics, based on the so-called Potts model
- in this model, each particle (i.e. vertex) can be in one of k spin states
- the interactions between the particles (i.e. the edges of the graph) specify which pairs of vertices would prefer to stay in the same spin state and which ones prefer to have different spin states
- the model is then simulated for a given number of steps, and the spin states of the particles in the end define the communities

Label propagation

- `cluster_label_prop` is a simple approach in which every node is assigned one of k labels.
- the method then proceeds iteratively and re-assigns labels to nodes in a way that each node takes the most frequent label of its neighbors in a synchronous manner
- the method stops when the label of each node is one of the most frequent labels in its neighborhood

Infomap community finding

- `cluster_infomap` is based on information theoretic principles
- it tries to build a grouping which provides the shortest description length for a random walk on the graph, where the description length is measured by the expected number of bits per vertex required to encode the path of a random walk
- when we describe a network as a set of interconnected communities, we are highlighting certain regularities of the network's structure while filtering out the relatively unimportant details
- a modular description of a network can be viewed as a lossy compression of that network's topology

Multi-level optimization of modularity

- `cluster_louvain` is based on the modularity measure and a hierarchical approach
- initially, each vertex is assigned to a community on its own
- in every step, vertices are re-assigned to communities in a local, greedy way: each vertex is moved to the community with which it achieves the highest contribution to modularity
- when no vertices can be reassigned, each community is considered a vertex on its own, and the process starts again with the merged communities
- the process stops when there is only a single vertex left or when the modularity cannot be increased any more in a step

Optimal community structure

- `cluster_optimal` calculates the optimal community structure for a graph, in terms of maximal modularity score
- the calculation is done by transforming the modularity maximization into an integer programming problem, and then calling the GLPK library to solve that

Example: Run at least 3 community detection algorithms on the Zachary network and find the best one according to modularity.

```
library(igraph)
g = make_graph("Zachary")
modularity(cluster_leading_eigen(g))
```

```
## [1] 0.3934089
```

```
modularity(cluster_fast_greedy(g))
```

```
## [1] 0.3806706
```

```
modularity(cluster_edge_betweenness(g))
```

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

```
modularity(cluster_walktrap(g))
```

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

```
modularity(cluster_spinglass(g))
```

```
## [1] 0.4197896
```

```
modularity(cluster_label_prop(g))
```

```
## [1] 0.3717949
```

```
modularity(cluster_infomap(g))
```

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

```
modularity(cluster_louvain(g))
```

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

```
modularity(cluster_optimal(g))
```

```
## [1] 0.4197896
```

Hierarchical clustering

- the basic idea behind hierarchical clustering is to define a measure of similarity or connection strength between vertices, based on the network structure
- then start by joining together those pairs of vertices with the highest similarities, forming a group or groups of size two
- then we further join together the groups that are most similar to form larger groups, and so on, until a unique group is formed that contains all nodes
- this produces a hierarchical decomposition of a network into a set of **nested** (non-overlapping) communities, that can be visualized in the form of a **dendrogram**

But what we actually have is a measure of similarity between individual vertices, so we need to combine these vertex similarities somehow to create similarities for the groups.

There are three common ways to do so:

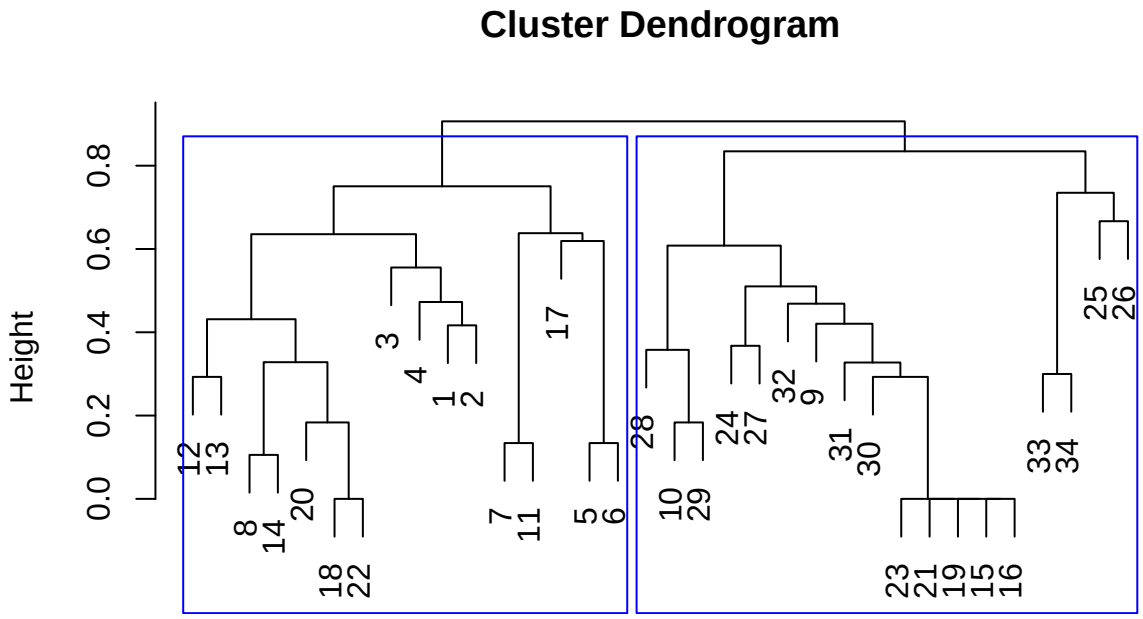
- **single-linkage**: the similarity between two groups is defined to be the maximum of the similarities of pairs of vertices belonging to different groups
- **complete-linkage**: the similarity between two groups is defined to be the minimum of the similarities of pairs of vertices belonging to different groups
- **average-linkage**: the similarity between two groups is defined to be the average of the similarities of pairs of vertices belonging to different groups

Given a similarity measure for individual nodes and a clustering method to create similarities for groups of nodes, the hierarchical clustering method is as follows:

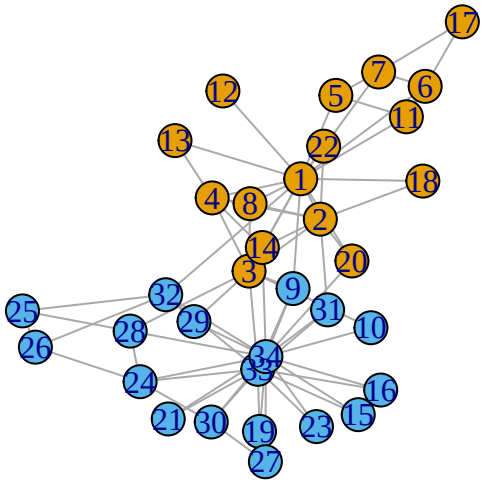
1. evaluate the similarity measures for all vertex pairs
2. assign each vertex to a group of its own, consisting of just that one vertex
3. find the pair of groups with the highest similarity and join them together into a single group
4. calculate the similarity between the new composite group and all others
5. repeat from step 3 until all vertices have been joined into a single group

Function `hclust` computes hierarchical clustering:

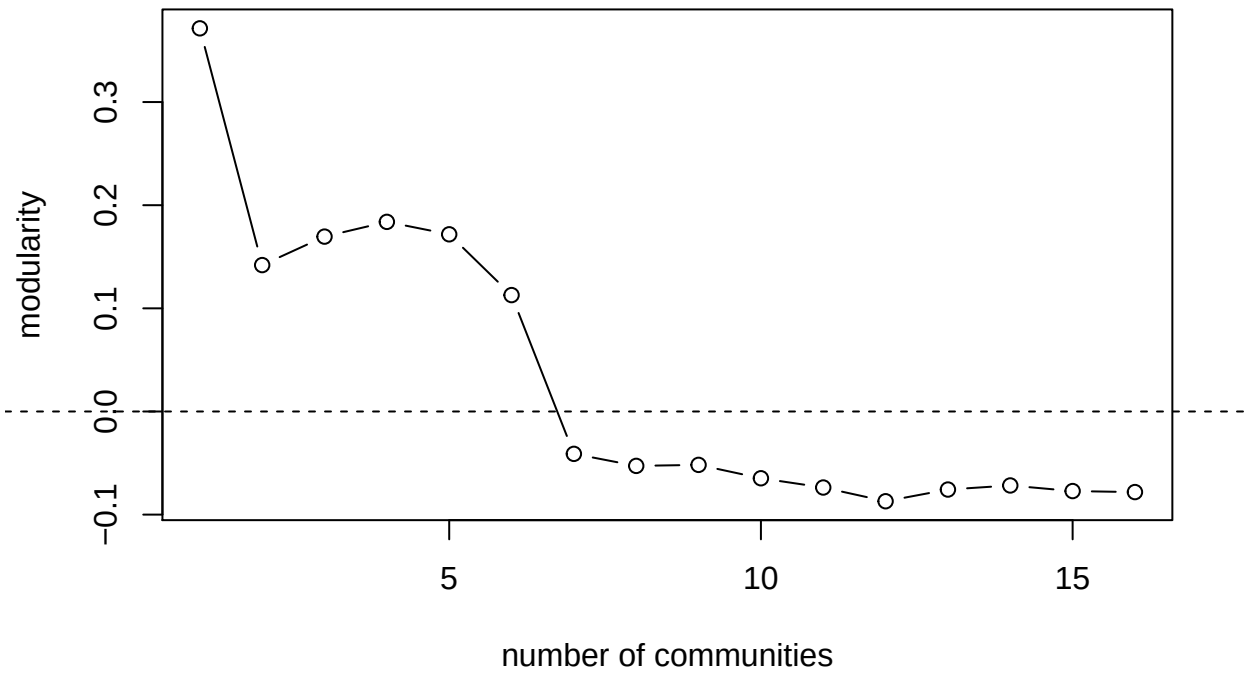
```r
library(igraph)

# make graph
g = make_graph("Zachary")

# hierarchical clustering
A = as_adjacency_matrix(g, sparse=FALSE)

# cosine similarity
euclidean = function(x) {sqrt(x %*% x)}
d = apply(A, 2, euclidean)
D = diag(1/d)
S = D %*% t(A) %*% A %*% D

# distance matrix
D = 1-S

# distance object
d = as.dist(D)

# average-linkage clustering method
cc = hclust(d, method = "average")

# plot dendrogram
plot(cc)

# draw blue borders around clusters
clusters.list = rect.hclust(cc, k = 2, border="blue")
```




```
# cut dendrogram at 2 clusters
clusters = cutree(cc, k = 2)

# plot graph with clusters
coords = layout_with_fr(g)
plot(g, vertex.color=clusters, layout=coords)
```




```
# cut dendrogram at different heights
clusters = cutree(cc, k = 2:(vcount(g)/2))

# compute modularity for each partition
m = apply(clusters, 2, FUN = function(x) {modularity(g, x)})

# plot modularities
plot(m, xlab = "number of communities", ylab = "modularity", type = "b")
abline(h = 0, lty = 2)
```



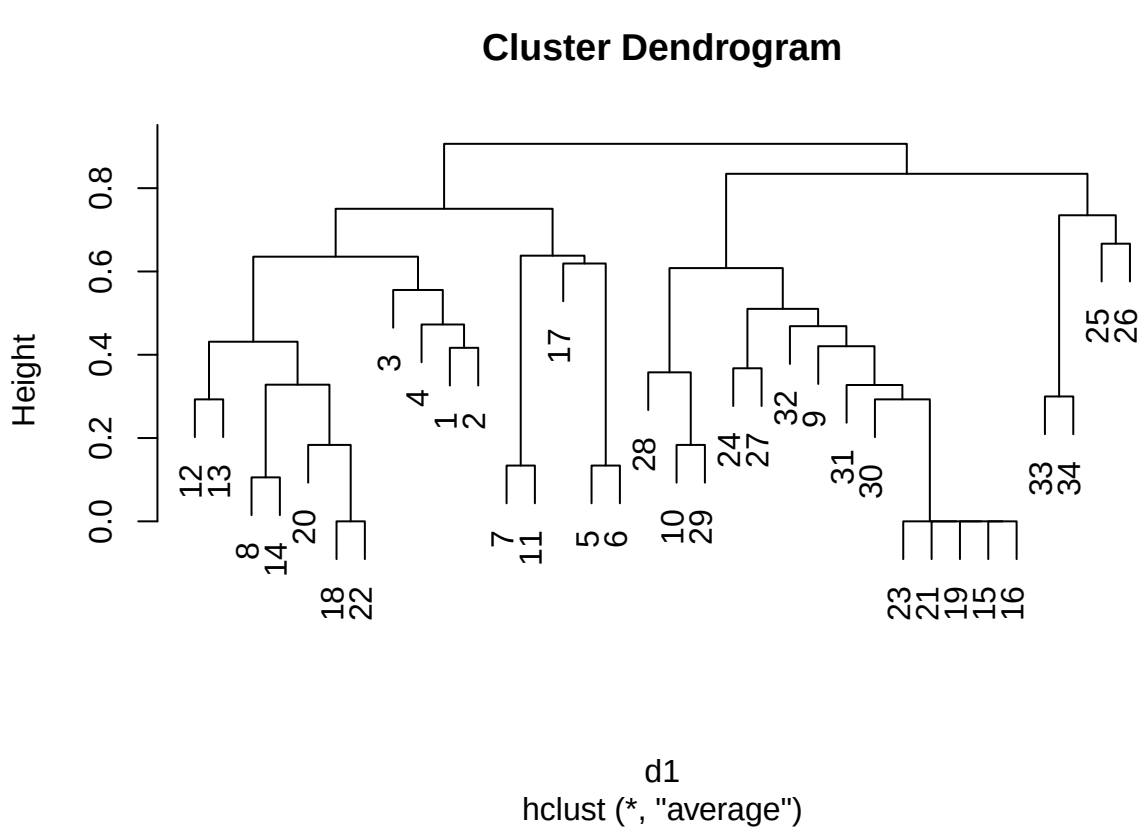

```
# find partition with maximum modularity
clusters[ ,which.max(m)]
```

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

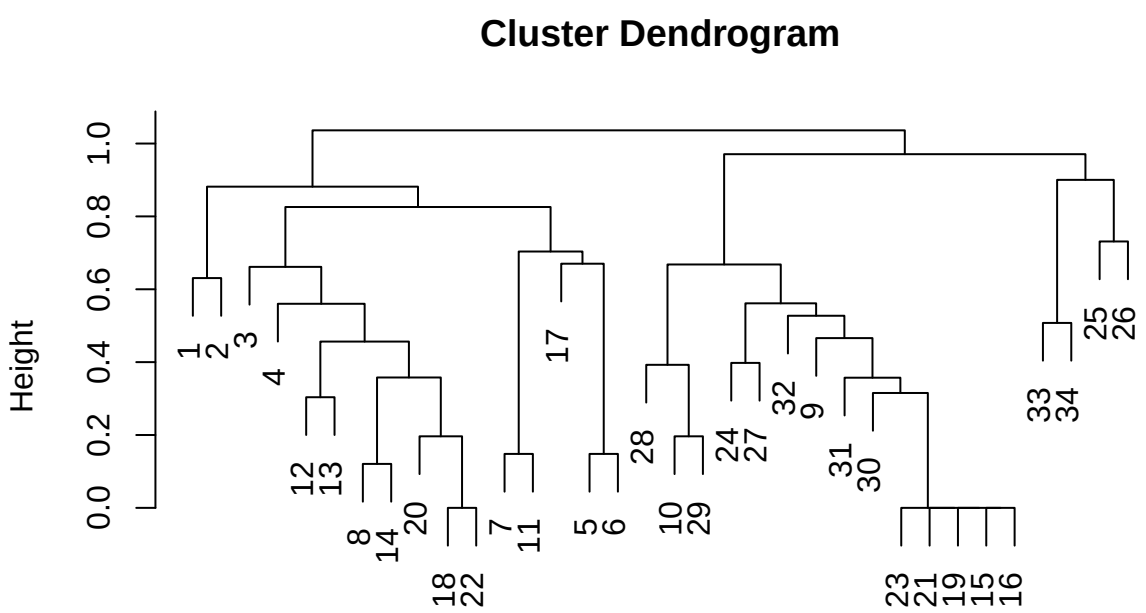
Example: Compute hierarchical clustering on Zachary graph using Pearson similarity and compare with cosine similarity.

```
library(igraph)

# make graph
g = make_graph("Zachary")

# hierarchical clustering
A = as_adjacency_matrix(g, sparse=FALSE)

# cosine similarity
euclidean = function(x) {sqrt(x %*% x)}
d = apply(A, 2, euclidean)
D = diag(1/d)
S1 = D %*% t(A) %*% A %*% D

# distance matrix
d1 = as.dist(1-S1)

# average-linkage clustering method
cc1 = hclust(d1, method = "average")
```

```r
# Pearson similarity
S2 = cor(A)

# distance matrix
d2 = as.dist(1-S2)

# average-linkage clustering method
cc2 = hclust(d2, method = "average")


# plot dendrogram
plot(cc1)
```




```r
plot(cc2)
```




```
clusters1 = cutree(cc1, k = 2)
clusters2 = cutree(cc2, k = 2)
clusters1 - clusters2
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

12. Network models

Network structure

There are a number of common **recurring patterns** seen in network structures, patterns that have a profound effect on the way networked systems work:

- connected components and **giant components**
- path lengths and the **small-world effect**
- degree distributions, power laws and **scale-free networks**
- motifs

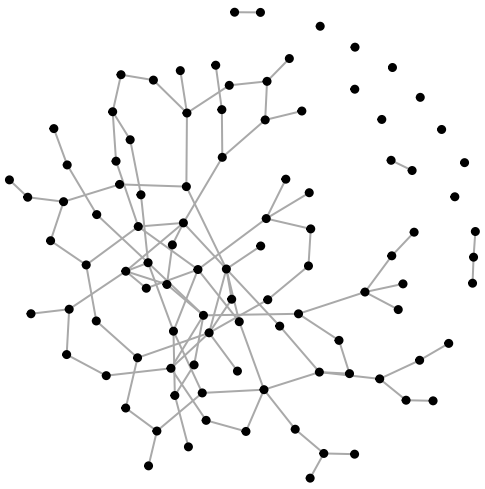
- assortative mixing
- a **network model** is a method to generate artificial networks
- when implemented in software, a network model can be used in simulations and experiments
- there are two main network models: the **random model** and the **preferential attachment model**

Random model

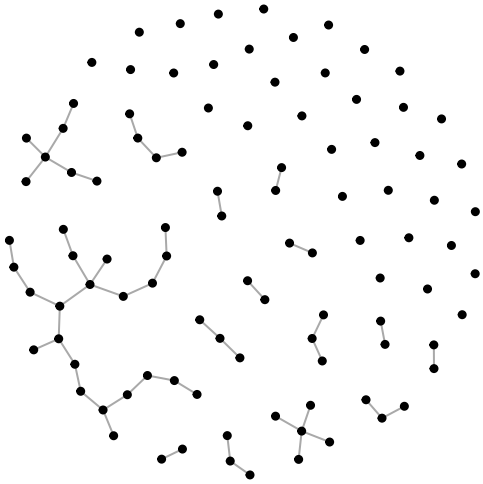
- the random model, also known as **Erdős-Rényi model**, is simplest and oldest network model
- according to this model, a network is generated by:
  a. laying down a number n of nodes

  b. adding edges between them with independent probability p for each node pair

```
library(igraph)
plot(sample_gnp(n = 100, p = 0.02),
     vertex.label = NA, vertex.size = 3, vertex.color = "black")
```



```r
plot(sample_gnp(n = 100, p = 0.01),
     vertex.label = NA, vertex.size = 3, vertex.color = "black")
```



Preferential attachment model

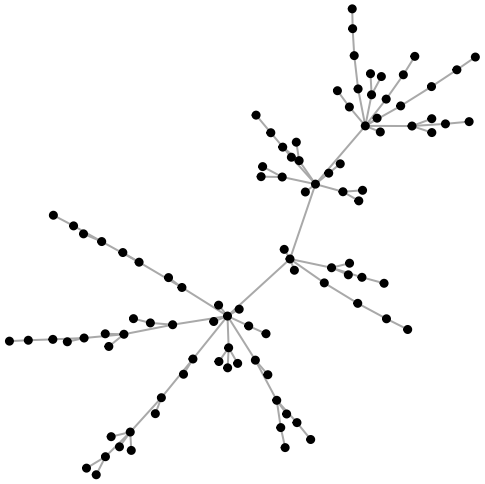
An alternative is the **preferential attachment model**, also known as Barabasi-Albert model:

1. the n nodes are added to the network one at a time
2. each node connects to the existing nodes with a fixed number m of links. The probability that it will choose a given node is proportional to the degree of the chosen node

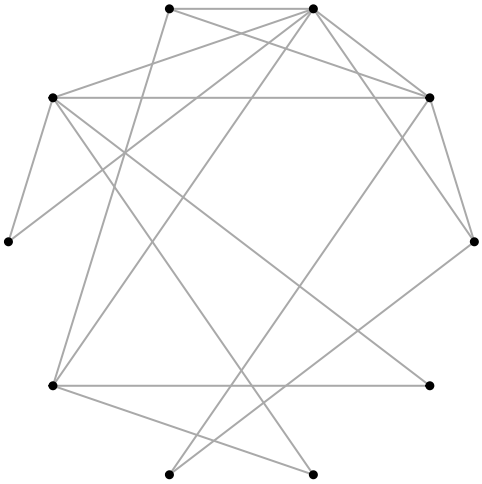
As we all experienced in our lives, **success breeds success**, the rich get richer, popularity is attractive.

> "To him that hath, more shall be given; and from him that hath not, the little that he hath shall be taken away." (Matthew 25:29, The parable of the talents)

```r
library(igraph)
plot(sample_pa(n = 100, m = 1, directed = FALSE),
     vertex.label = NA, vertex.size = 3, vertex.color = "black")
```



```r
plot(sample_pa(n = 10, m = 2, directed = FALSE), layout = layout_in_circle,
     vertex.label = NA, vertex.size = 3, vertex.color = "black")
```



13. Connectivity and resilience

Connected components

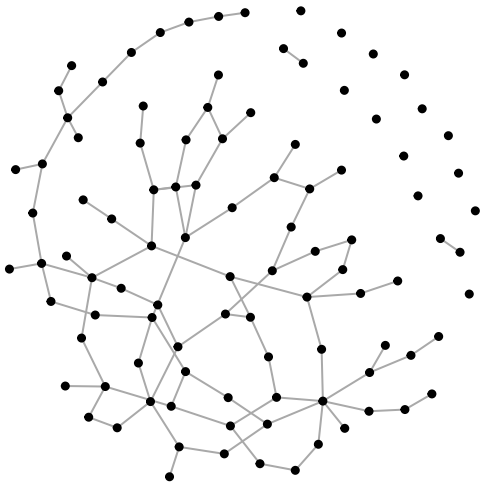
A **connected component** of an undirected graph is a maximal set of nodes such that each pair of nodes is connected by a path.

- connected components form a **partition** of the set of graph vertices, meaning that connected components are:
  1. non-empty
  2. pairwise disjoints
  3. the union of connected components forms the set of all vertices
- equivalently, we can say that the relation that associates two nodes if and only if they belong to the same connected component is an **equivalence relation**, that is it is:
  1. reflexive
  2. symmetric
  3. transitive

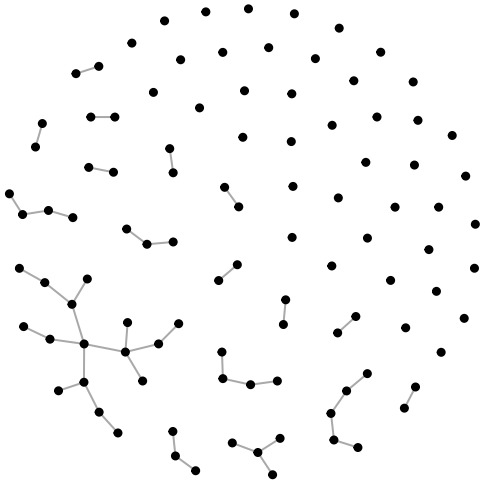
The giant component

- in real undirected networks, we typically find that there is a large component (the **giant component**) that fills most of the network
- both the random model and the preferential attachment model generate networks with giant components
- in particular, random graphs with n nodes and edge probability p show a giant component as soon as $p > 1/n$
- it follows that if the mean node degree is at least one, $p(n-1) \geq 1$, then $p \geq 1/(n-1) > 1/n$, and we have the giant component

```
library(igraph)
n = 100
p = 2/n
g = sample_gnp(n, p)
plot(g, vertex.size = 3, vertex.color = "black", vertex.label=NA)
```



```
n = 100
p = 1/n
g = sample_gnp(n, p)
plot(g, vertex.size = 3, vertex.color = "black", vertex.label=NA)
```



Example: Name at least three networks in which the giant component tipically covers the entire graph.

The Internet is a good example: there would be no point in being part of the Internet if you are not part of its largest component, since that would mean that you are disconnected from and unable to communicate with almost everyone else.

Other examples include power grids, train routes, and electronic circuits.

Biconnected components and more

> k-component is a maximal set of vertices such that each is reachable from each of the others by at least k node-independent paths (paths that do not share any node unless the source and the target ones).

- a 1-component is just an ordinary component; a 2-component is called bicomponent; a 3-component is called tricomponent
- for $k \geq 2$, a k-component is nested within a $k-1$ component: every pair of nodes connected by k independent paths is also connected by $k-1$ independent paths
- for $k \geq 2$, k-components may overlap and they do not define a partition of (or equivalence relation on) the set of vertices
- for $k \geq 3$, the k-components in a network can be non-contiguous

Example:

- find a graph with two overlapping bicomponents
- find a graph with a non-contiguous tricomponent

IMG x2

Connectivity

- the number of node-independent paths between two nodes is also known as the **connectivity** of the two nodes
- the minimum connectivity of any pair of nodes in a graph is the connectivity (or cohesion) of the graph
- connectivity of two nodes can be thought as a measure of how strongly connected the two nodes are
- the smallest set of vertices that would have to be removed in order to disconnect two nodes not linked by an edge is called the **minimum cut set** for the two vertices
- it holds that the size of the minimum cut set for two nodes (not linked by an edge) is exactly the connectivity of the two nodes
- it follows that a k-component remains connected as soon as less than k nodes (and the incident edges) are removed
- for instance, a bicomponent is still connected if we remove any of its nodes
- for these reasons, the idea of k-component is associated with the idea of network **robustness**
- for instance, one would hope that most of the Internet network backbone is a k-component with high k, so that it would be difficult for routers on the backbone to lose connection with one another

Strongly connected components

> Two vertices are in the same **strongly connected component** if each can reach and is reachable from the other along a directed path

- strongly connected components form a partition of the vertex set and define an equivalence relation among nodes
- not all directed networks have a large strongly connected component
- in particular, any acyclic network has no strongly connected components of size greater than one
- real-life networks that are (almost) acyclic are article citation networks and food webs
- associated with each strongly connected component is an **out-component**: the set of vertices that can be reached from the strongly connected component but that cannot reach it
- and an **in-component**: the set of vertices that can reach the strongly connected component but that are note reachable from it

The bow-tie structure of the Web

IMG

Code implementations

```
n = 100
p = 2/n
g = sample_gnp(n, p)

# connected?
is_connected(g)
```

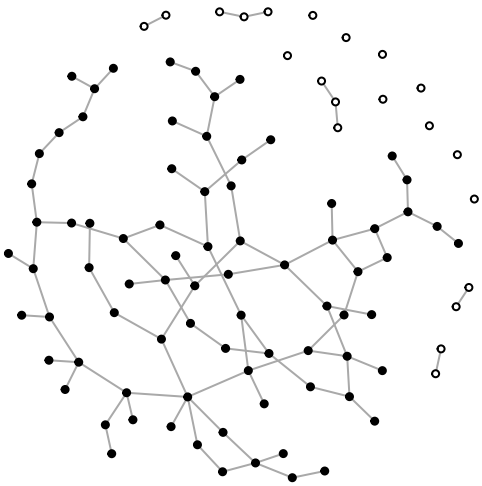
Code – Connected components

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

```
# get the components
(c = components(g))
```

```
## $membership
##   [1]  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1
##  [26]  1  1  1  1  1  3  1  1  1  4  4  1  1  1  1  1  5  1  6  1  1  1  1  7  1
##  [51]  1  1  5  1  8  1  1  1  1  1  1  1  2  1  1  8  9  1  1  1  1 10  1  1  1
##  [76]  1  7 11  1  1  1  1  1  1 12  1  1  1  1  1  1 13  1 14 15  1  1  1  1  7
##
## $csize
##  [1] 79  3  1  2  2  1  3  2  1  1  1  1  1  1  1
##
## $no
## [1] 15
```

```
# get the giant component
nodes = which(c$membership == which.max(c$csize))

# color in red the nodes in the giant component
V(g)$color = "white"
V(g)[nodes]$color = "black"
plot(g, vertex.size = 3, vertex.label=NA)
```



```
n = 100
p = 3/n
g = sample_gnp(n, p)

bc = biconnected_components(g)
```

```
# number of bicomponents
bc$no
```

Code – Bicomponents

```
## [1] 21
```

```
# print nodes of biconnected components
bcn = bc$components
bcn[[1]]
```

```
## + 2/100 vertices, from 1c21b10:
## [1] 73 46
```

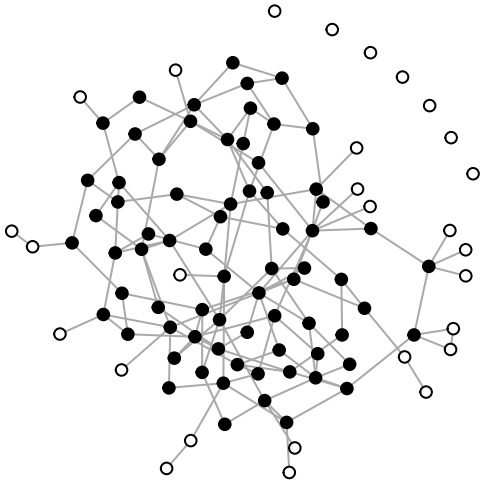
```
# print edges of biconnected components
bce = bc$component_edges
bce[[1]]
```

```
## + 1/142 edge from 1c21b10:
## [1] 46--73
```

```
# size of bicomponents
size = sapply(bcn, length)
table(size)
```

```
## size
##  2  3 72
## 19  1  1
```

```
# highlight the largest bi-component
giant = bcn[[which.max(size)]]
V(g)$color  = "white"
V(g)[giant]$color = "black"
plot(g, vertex.label=NA, vertex.size=5)
```



```
g = make_graph("Zachary")

# connectivity of the graph
vertex_connectivity(g)
```

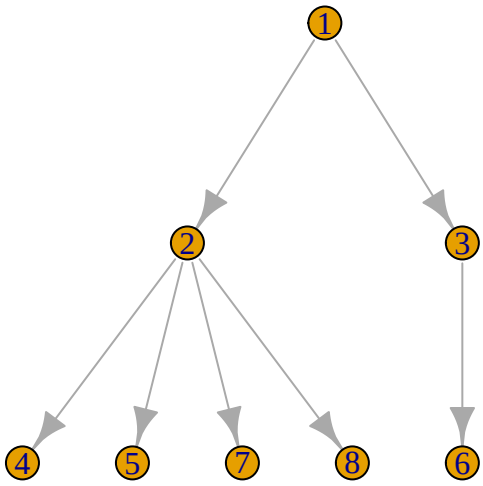
k-connected components

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

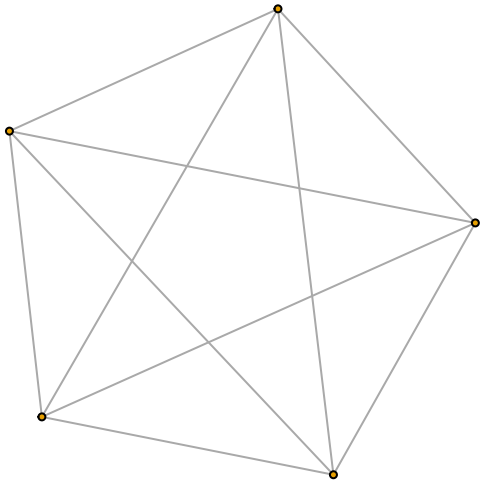
```
# this is computationally **heavy**
b = cohesive_blocks(g)

# print hiearchy of blocks
print(b)
```

```
## Cohesive block structure:
## B-1         c 1, n 34
## '- B-2      c 2, n 28   oooo...ooo ..oooo.ooo ooooooooo oooo
##    '- B-4   c 4, n  5   oooo...o.. .......... .......... ....
##    '- B-5   c 3, n  7   ooo.....o. .......... .......... o.oo
##    '- B-7   c 4, n  5   oooo..... ...o..... .......... ....
##    '- B-8   c 3, n 10   ..o..... .......... ...ooo.ooo .ooo
## '- B-3      c 2, n  6   o...ooo... o.....o... .......... ....
##    '- B-6   c 3, n  5   o...ooo... o......... .......... ....
```

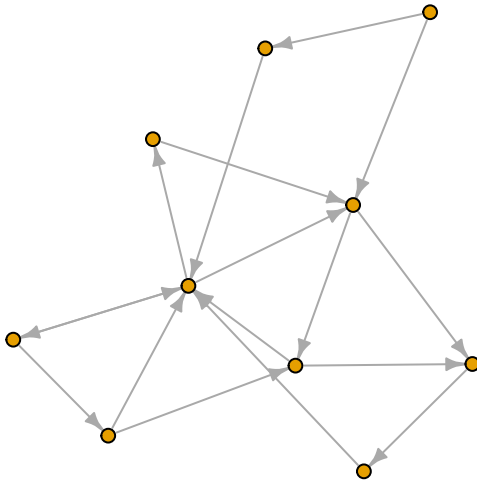
```
# plot hiearchy of blocks
plot_hierarchy(b)
```




```
# nodes from blocks
bn = blocks(b)

# graphs from blocks
bg = graphs_from_cohesive_blocks(b, g)

# plot most cohesive graph
c = cohesion(b)
h = bg[[which.max(c)]]
plot(h, vertex.size = 3, vertex.label=NA)
```

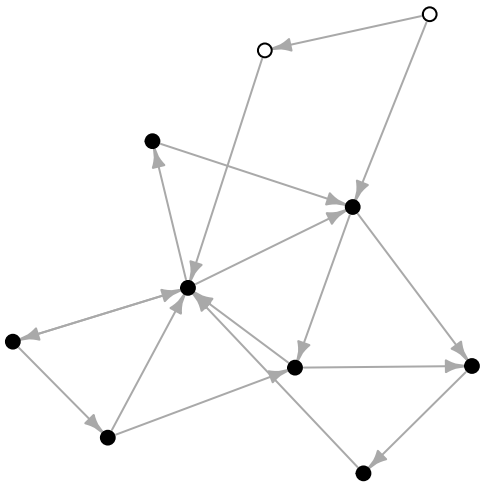


```
n = 10
p = 2/n
g = sample_gnp(n, p, directed=TRUE)
coords = layout_with_fr(g)
plot(g, layout=coords,
     vertex.size = 6, vertex.label=NA,
     edge.arrow.size = 0.5)
```



Code – Strongly connected components

```
# strong components
c = components(g, mode="strong")
nodes = which(c$membership == which.max(c$csize))

# color in red the nodes in the giant component
V(g)$color = "white"
V(g)[nodes]$color = "black"
plot(g, layout=coords,
     vertex.size = 6, vertex.label=NA,
     edge.arrow.size = 0.5)
```



Percolation

- related to network connectivity and robustness is **percolation**, one of the simplest of processes taking place on networks
- imagine taking a network and removing some fraction of its vertices, along with the edges connected to those vertices: this process is called percolation
- one of the goals of percolation theory on networks is to understand how the knock-on effects of vertex removal affect the network as a whole

Failure of Internet routers

- the failure of routers on the Internet, for instance, can be formally represented by removing the corresponding vertices and their attached edges from a network representation of the Internet.
- in fact, about 3% of the routers on the Internet are non-functional for one reason or another at any one time, and it is a question of some practical interest what effect this will have on the performance of the network

Vaccination

- another example of a percolation phenomenon is the vaccination or immunization of individuals against the spread of disease
- diseases spread through populations over the networks of contacts between individuals, but if an individual is vaccinated against a disease and therefore cannot catch it, then that individual does not contribute to the spread of the disease
- of course, the individual is still present in the network, but, from the point of view of the spread of the disease, might as well be absent, and hence the vaccination process can again be formally represented by removing vertices

Percolation strategies

- there is more than one way in which vertices can be removed from a network
- in the simplest case they could be removed purely at random
- one popular alternative removal scheme is to remove vertices according to their degree in decreasing order
- this approach turns out to make an effective vaccination strategy for the control of disease
- the vaccination of an individual in a population not only prevents that individual from becoming infected but also prevents them from infecting others
- any other node centrality measures can be used as an alternative and might be more efficient

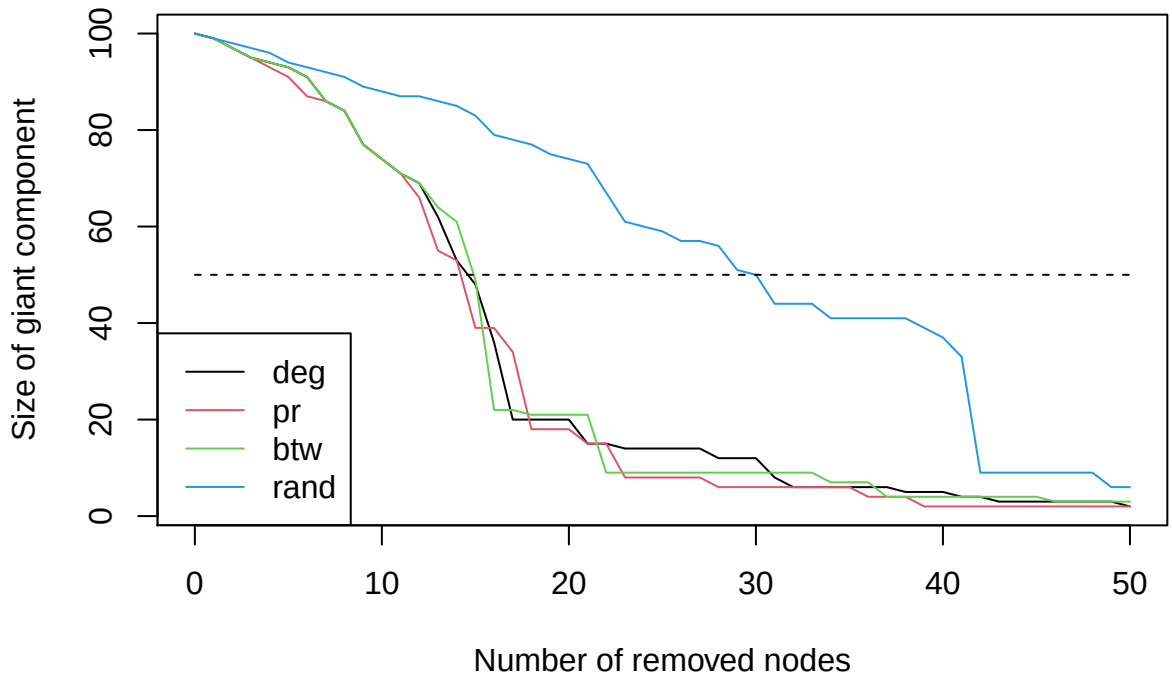
Example: Consider the following code for the percolation process on a network. It increamentally remove from the graph a given number of nodes from a given vector of nodes and compute the size of the giant component after each removal

```
# percolation removes nodes from a graph and computes
# the size of the giant connected component
# INPUT
# g: graph to percolate
# size: number of nodes to remove
# d: removal vector
# OUTPUT
# giant: a vector with sizes of giant components when nodes are removed
percolate = function(g, size, d) {

  giant = vector()

  # initial size of giant component
  c = components(g)
  giant[1] = max(c$csize)

  # find vital nodes
  names(d) = 1:length(d)
  d = sort(d, decreasing=TRUE)
  vital = as.integer(names(d[1:size]))

  # compoute size of giant component after incremental removal
  for (i in 1:size) {
    c = components(delete_vertices(g, vital[1:i]))
    giant[i+1] = max(c$csize)
  }

  return(giant)

}
```

Test the percolation process on random and preferential attachment networks: do you see any difference? Why?

```
# Preferential attachment graph
g = sample_pa(n = 100, m = 2, directed=FALSE)

# resilience
size = vcount(g)/2
# random
rand = percolate(g, size, d = sample(V(g), size))
```

```
# degree
deg = percolate(g, size, d = degree(g))
# pagerank
pr = percolate(g, size, d=page_rank(g)$vector)
# betweenness
bet = percolate(g, size, d = betweenness(g))

plot(0:size, deg, type = "l", col=1,
     xlab="Number of removed nodes",
     ylab="Size of giant component")
lines(0:size, pr, col=2)
lines(0:size, bet, col=3)
lines(0:size, rand, col=4)
lines(0:size, rep(vcount(g)/2, size+1), lty=2)
legend(x = "bottomleft",
       legend = c("deg", "pr", "btw", "rand"), lty = 1, col = 1:4)
```




```
# random model
g = sample_gnp(n=100, p=5/100)

# resilience
size = vcount(g)/2
# random
rand = percolate(g, size, d = sample(V(g), size))
# degree
deg = percolate(g, size, d = degree(g))
```

```r
# pagerank
pr = percolate(g, size, d=page_rank(g)$vector)
# betweenness
bet = percolate(g, size, d = betweenness(g))

plot(0:size, deg, type = "l", col=1,
     xlab="Number of removed nodes",
     ylab="Size of giant component")
lines(0:size, pr, col=2)
lines(0:size, bet, col=3)
lines(0:size, rand, col=4)
lines(0:size, rep(vcount(g)/2, size+1), lty=2)
legend(x = "bottomleft",
       legend = c("deg", "pr", "btw", "rand"), lty = 1, col = 1:4)
```