Computational Sociology

Social networks and data structures

Dr. Thomas Davidson

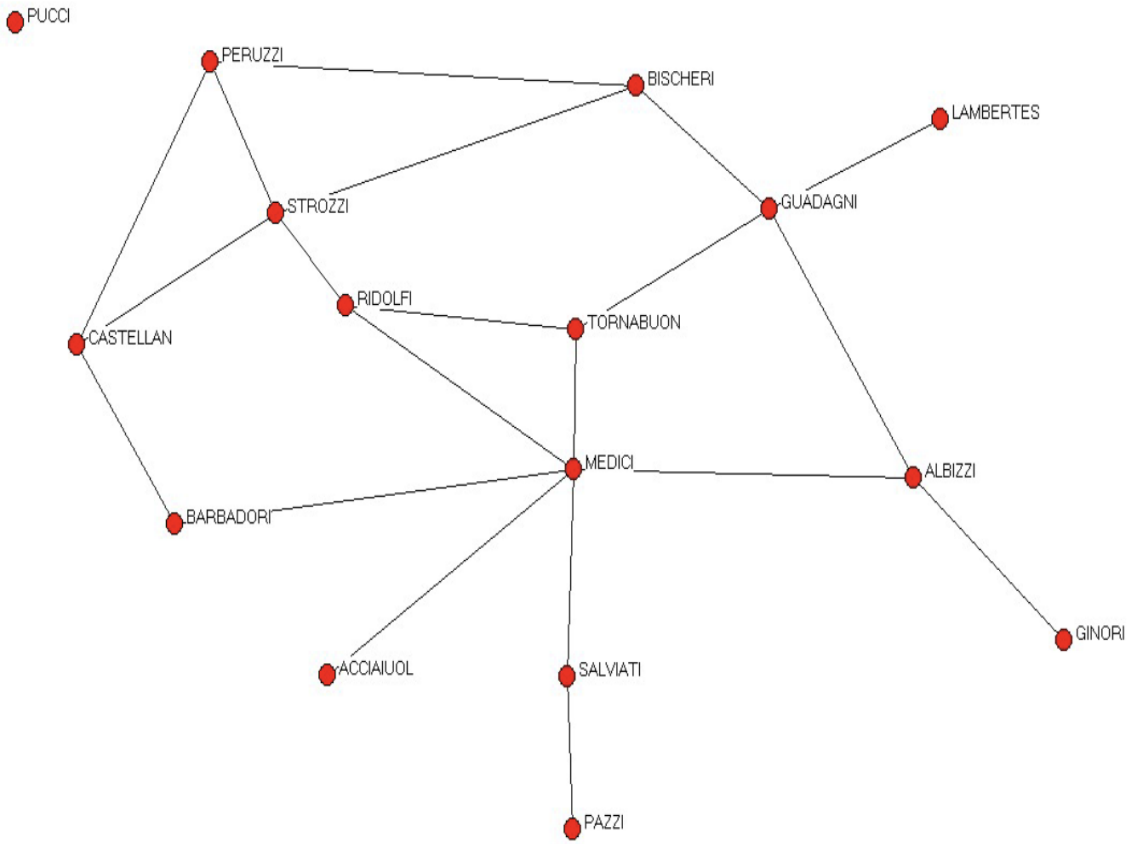
Rutgers University

January 25, 2024

Plan

- Part I : Social networks and social network analysis
  - Introduction to social networks
  - Big ideas
  - Two studies and implications for computational social science
- Part II : Data structures in R
  - Basic types
  - Vectors
  - Lists
  - Matrices
  - Data frames

Part I: Social networks and social network analysis



Social networks

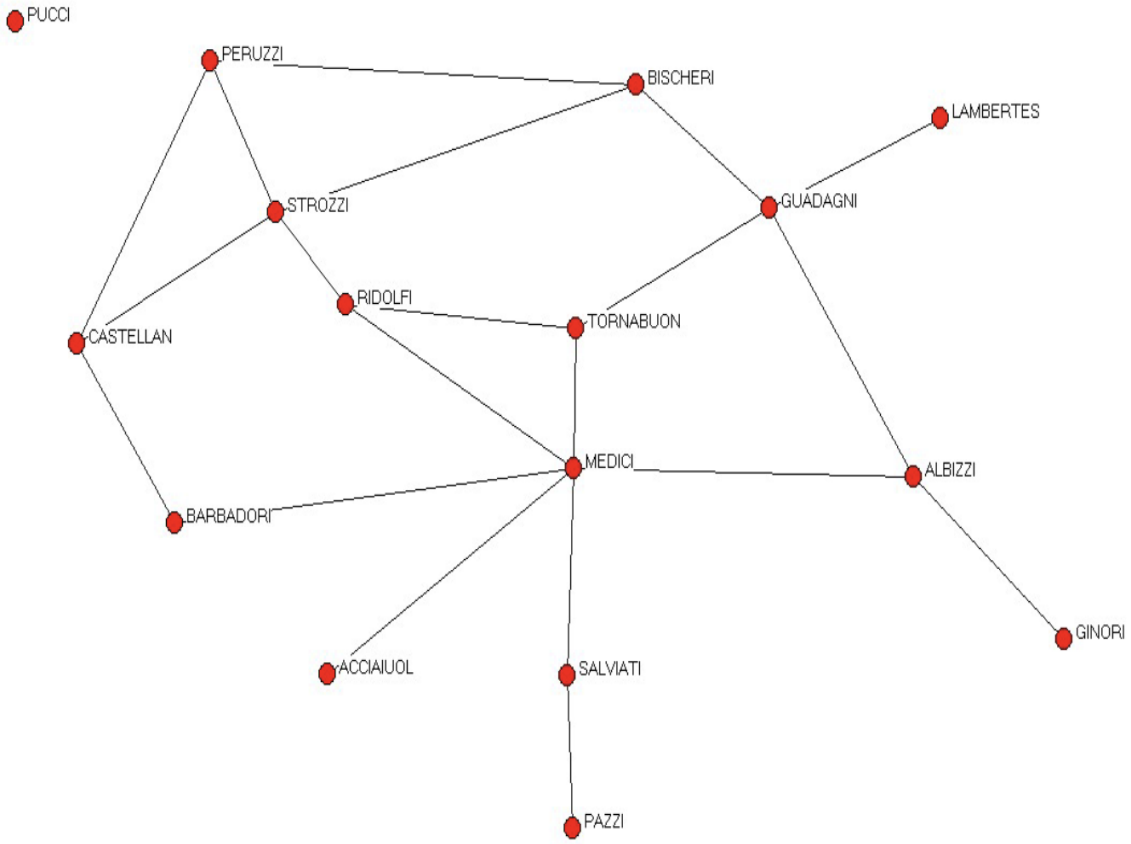
- A social network is a set of actors, or people, and the relationships between them
  - Family structures and kinship networks
  - Friendship and acquaintanceship networks
  - Organizations
  - Online social networks

Social network analysis

- Social network analysis is the study of social networks using mathematical and computational tools
- Networks can be represented using concepts from a branch of mathematics called graph theory
  - People or actors are represented as “nodes”
  - Relationships are represented as “edges”

Social network analysis

Visualizing networks



15th century Florentine marriage network (depicted in Jackson, Matthew O. 2010. *Social and Economic Networks*. Princeton University Press.)

Social network analysis

Visualizing networks

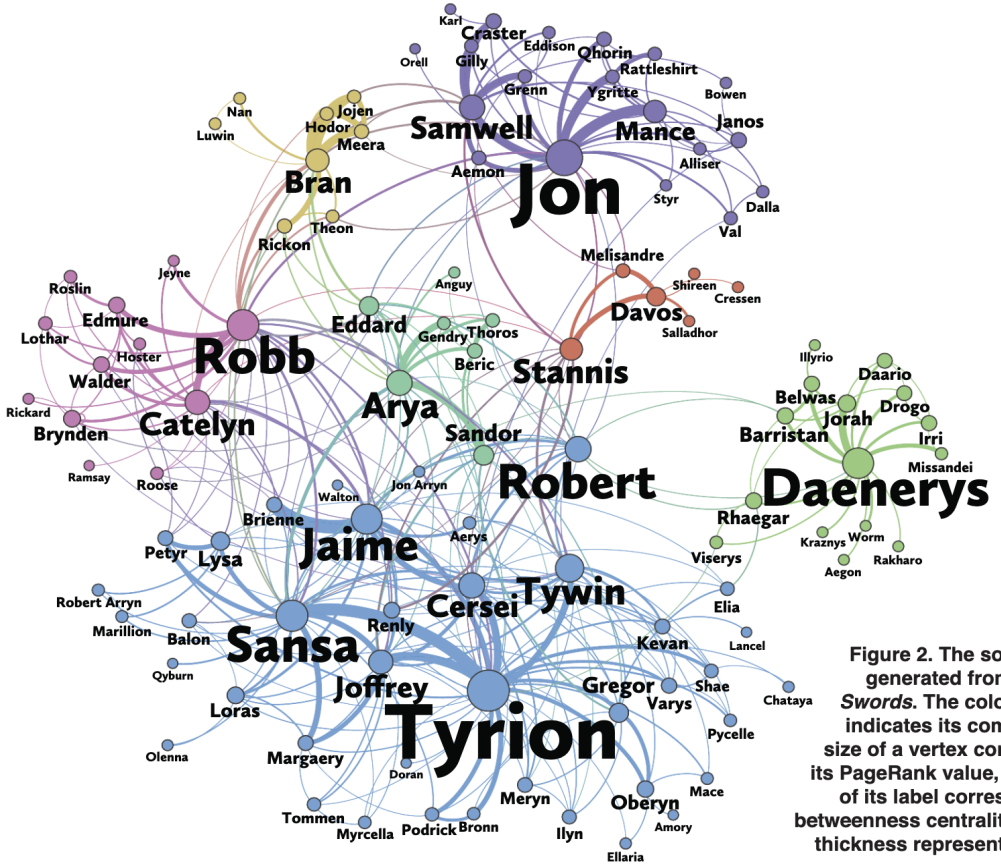


Figure 2. The social network generated from *A Storm of Swords*. The color of a vertex indicates its community. The size of a vertex corresponds to its PageRank value, and the size of its label corresponds to its betweenness centrality. An edge's thickness represents its weight.

Characters mentioned in *Game of Thrones* books, annotated with structural information. From Beveridge, Andrew, and Jie Shan. 2016. "Network of Thrones." *Math Horizons* 23(4): 18–22.

Social network analysis

Big ideas: Homophily

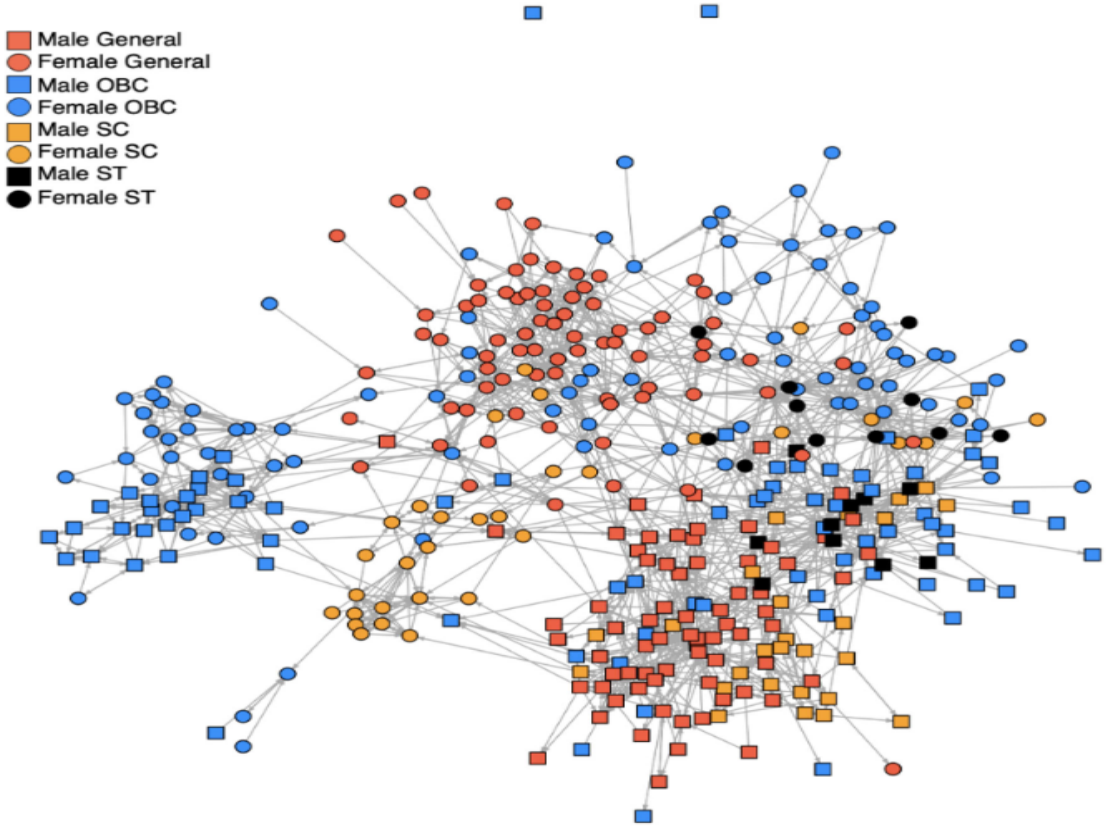
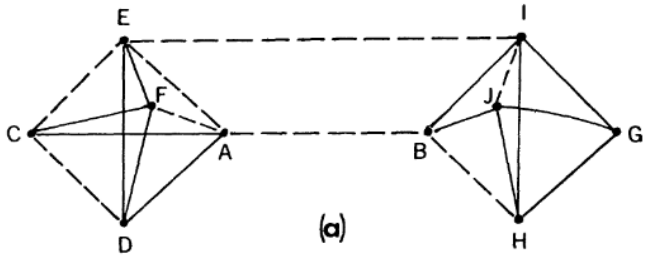


FIGURE 2 Network visualization of a selected village (Village #52).

Lee, Jaemin, and Mudit Kumar Singh. 2024. "Expansion, Cohesion and Diversity: The Network Advantages of Microfinance Groups in Indian Villages." *Journal of International Development* 36(1):559–86.

Social network analysis

Big ideas: The strength of weak ties



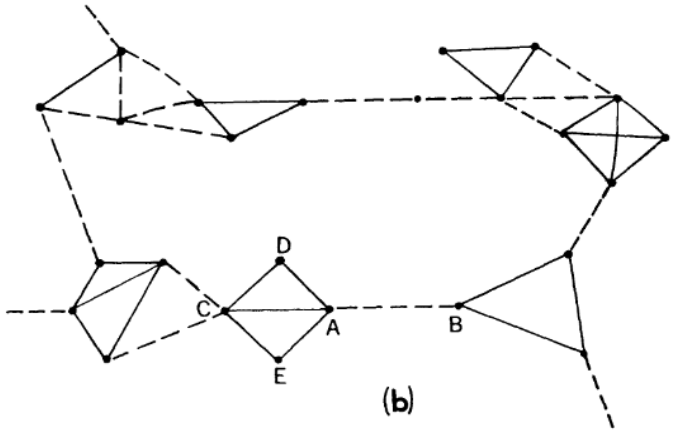


FIG. 2.—Local bridges. *a*, Degree 3; *b*, Degree 13. ——— = strong tie; — — — = weak tie.

Granovetter 1973.

Social network analysis

Big ideas: Structural holes

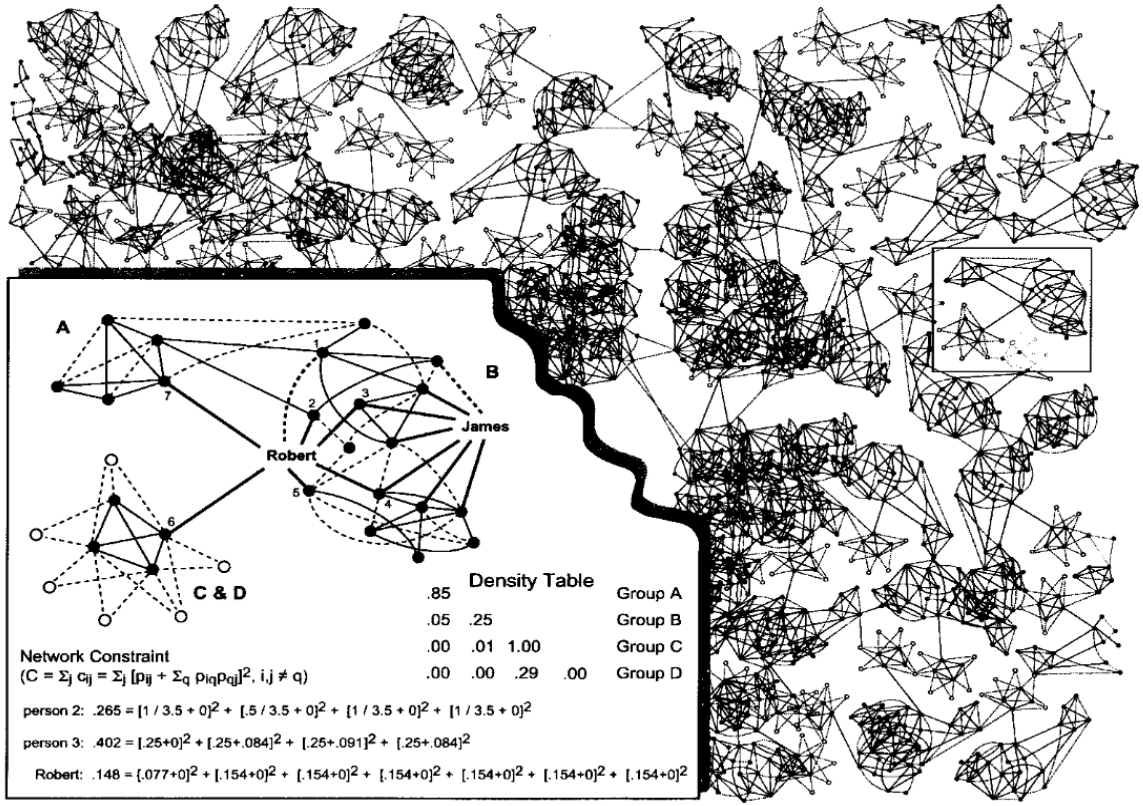
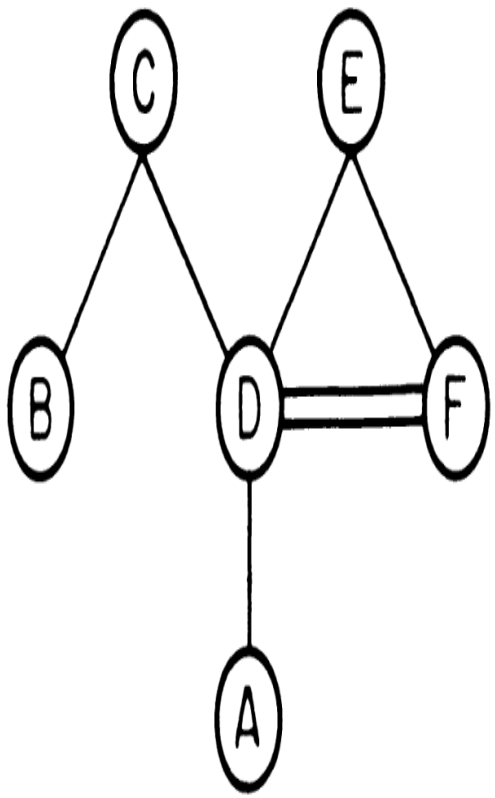


FIG. 1.—The small world of markets and organizations

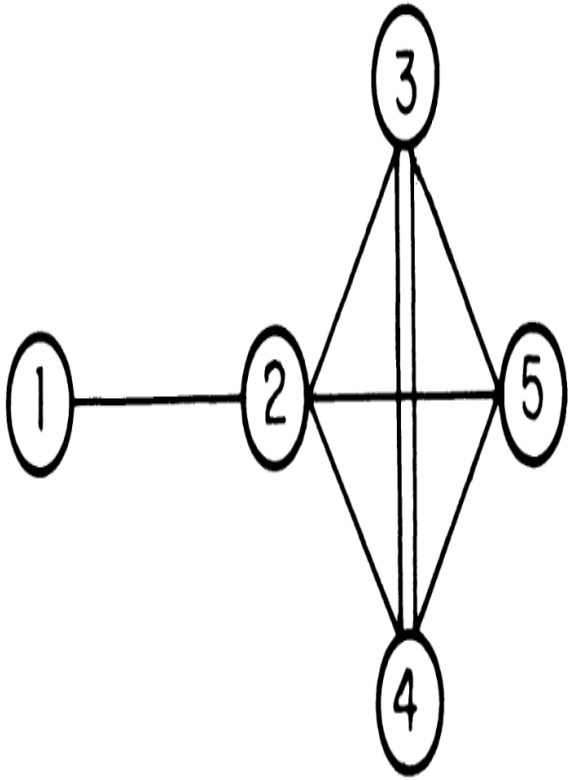
Burt 2004.

Social network analysis

Big ideas: Duality of persons and groups



A-1. Interpersonal network

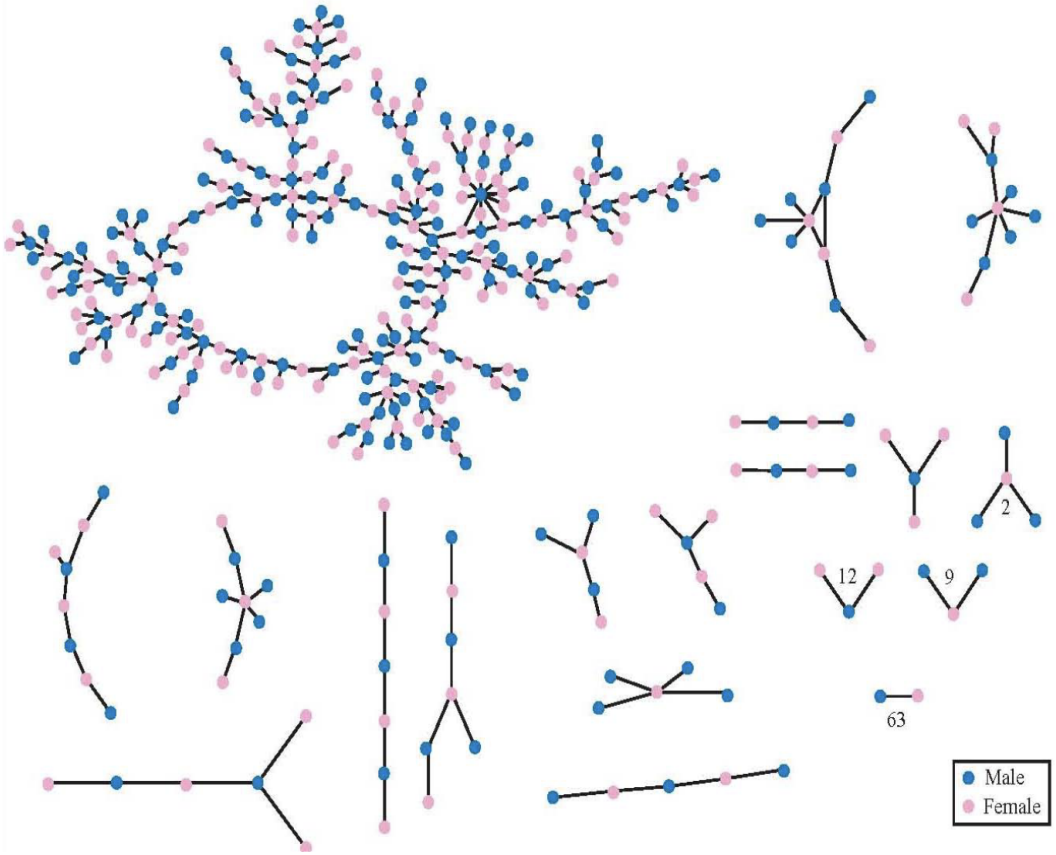


A-2. Intergroup network

Breiger 1974

Social network analysis

Spanning trees of sexual relationships in a high school



Bearman, Moody, and Stovel 2004

Social network analysis

Forbidden triads and triadic closure

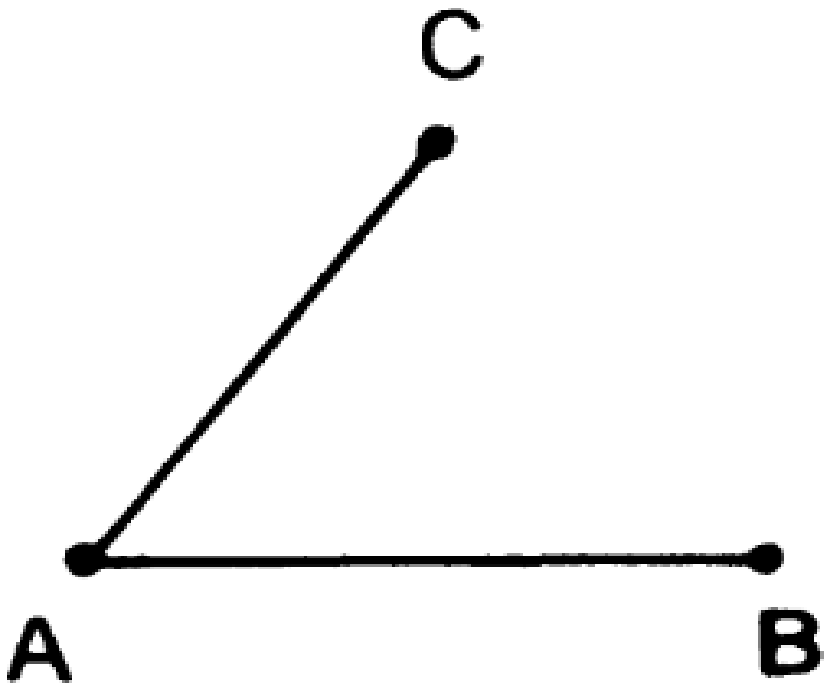


FIG. 1.—Forbidden triad

Granovetter 1973.

Social network analysis

Micro-mechanism: Norms against cycles and "seconds"

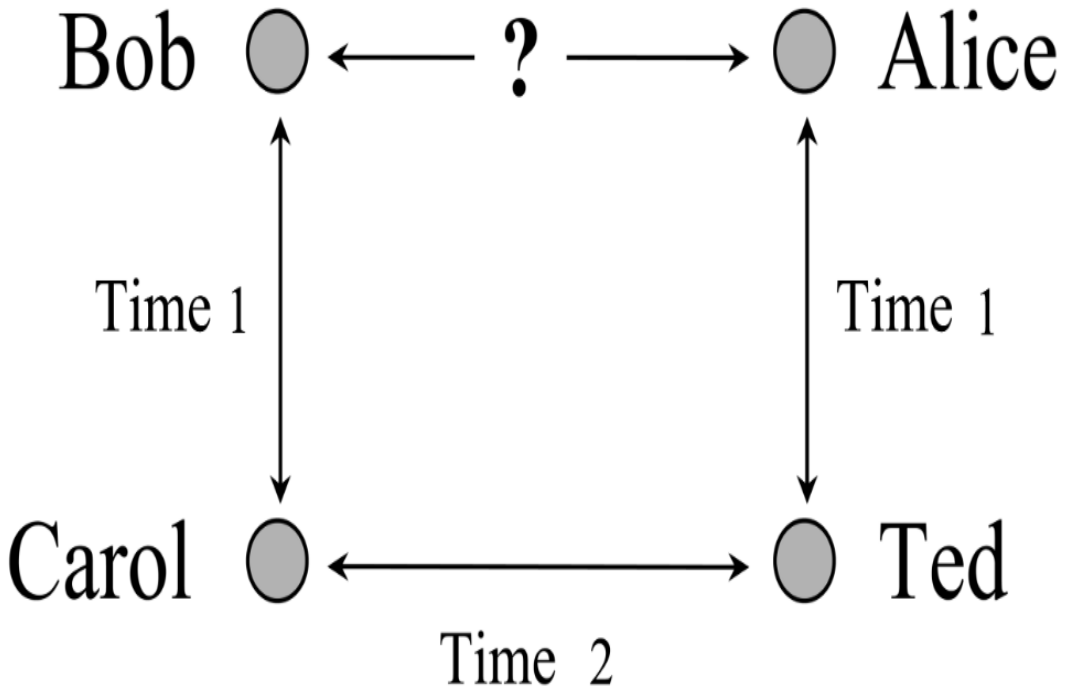
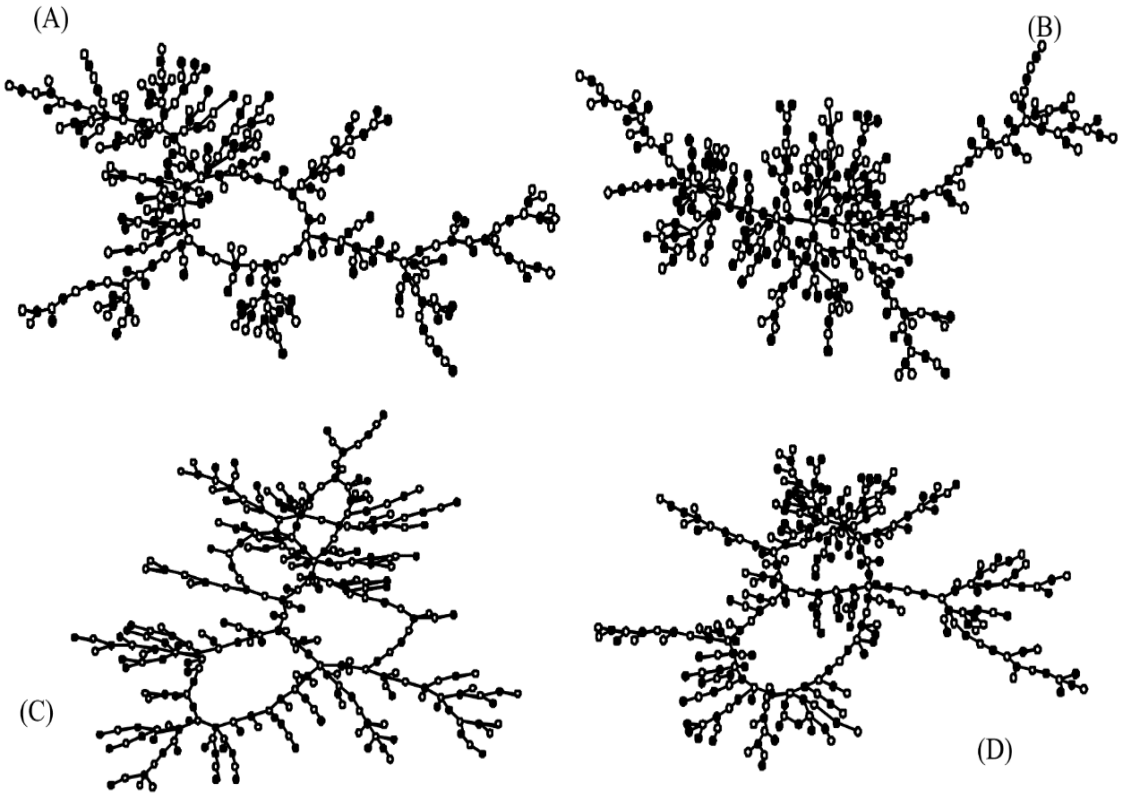


FIG. 8.—Hypothetical cycle of length 4

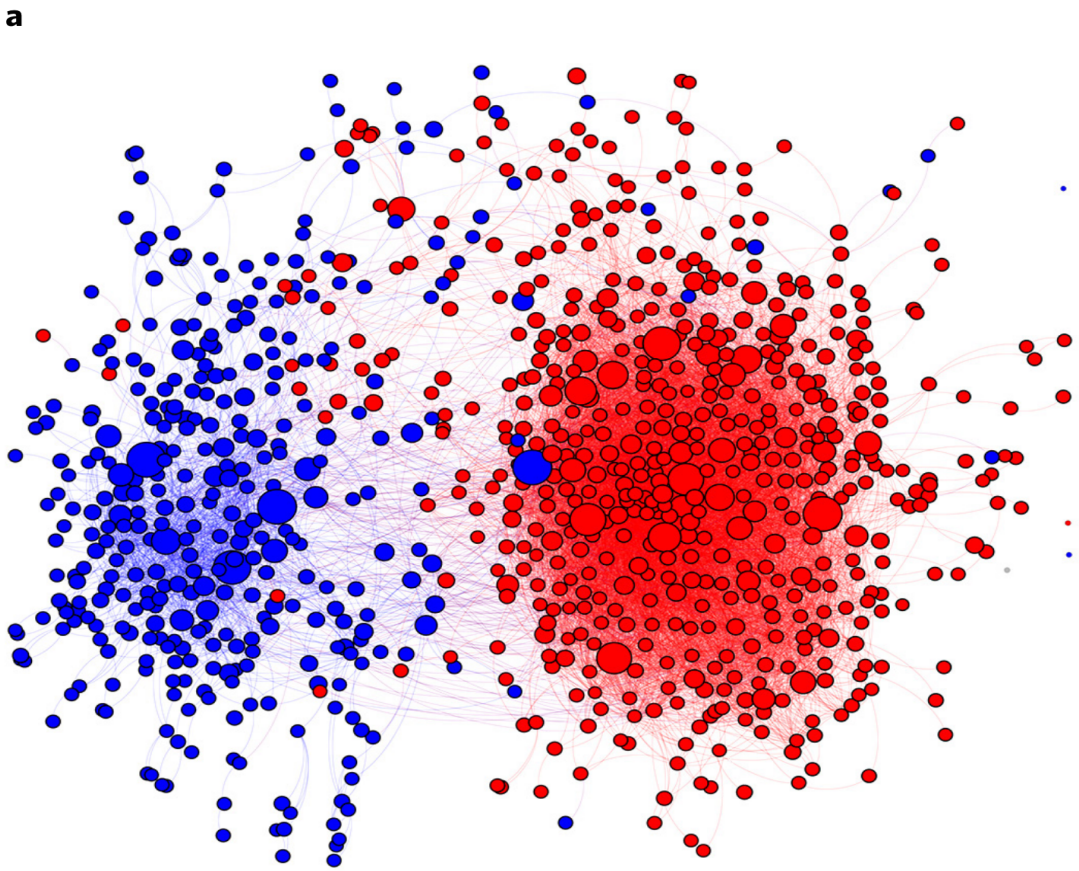
Social network analysis

Simulated networks



Social network analysis

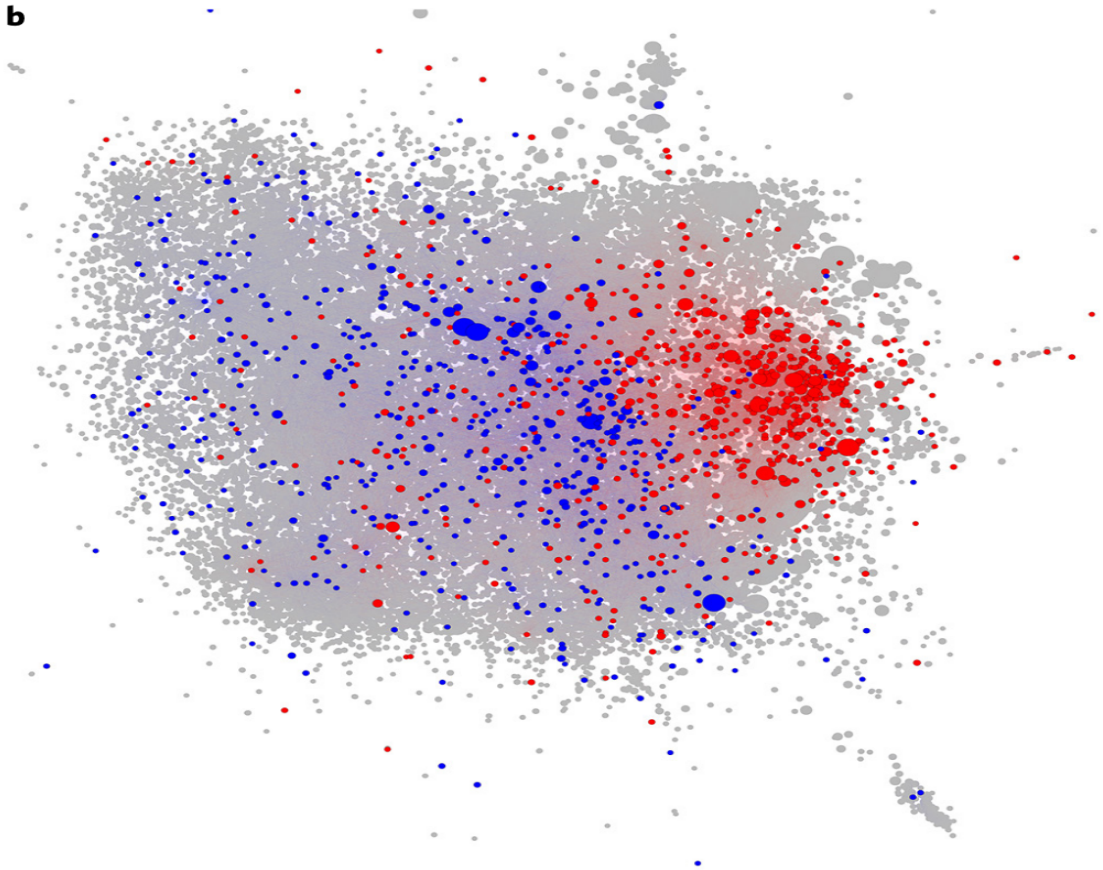
Polarization in book purchases



Shi et al. 2017

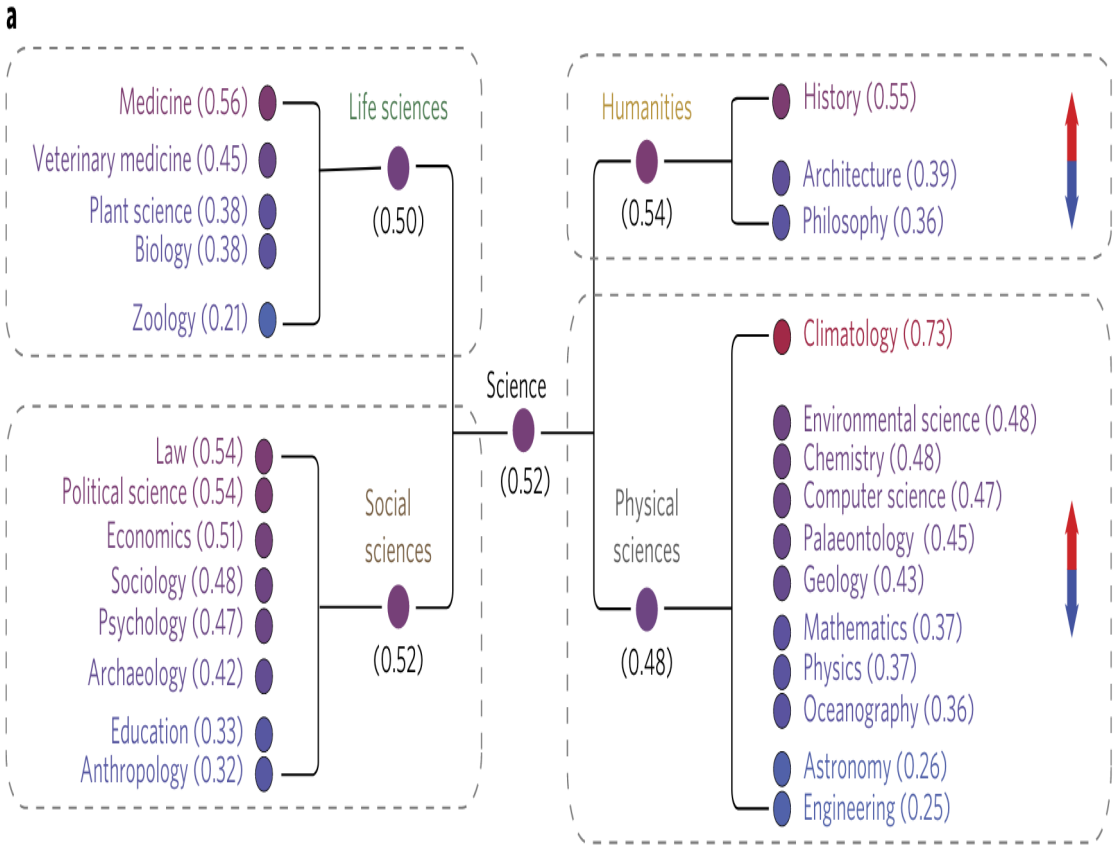
Social network analysis

Polarization in book purchases



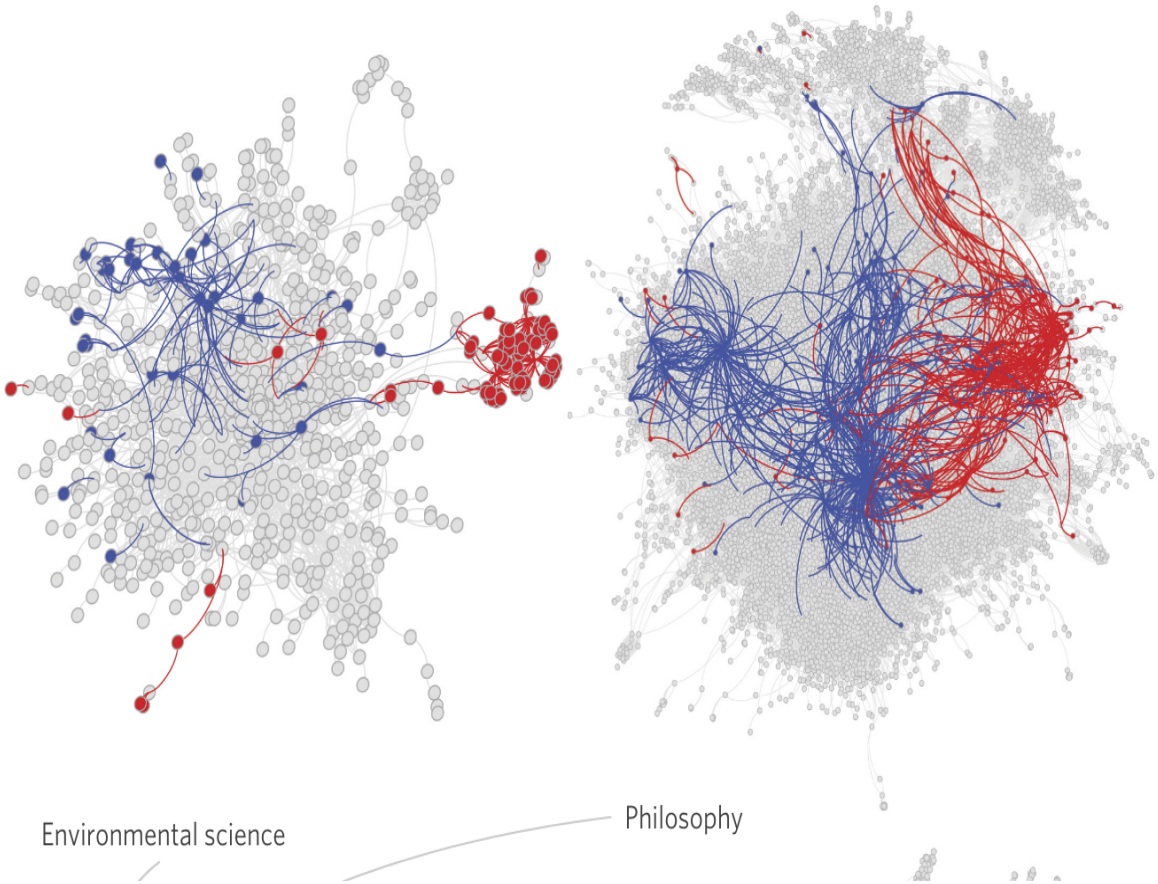
Social network analysis

Scientific fields and political ideology



Social network analysis

Polarized science?



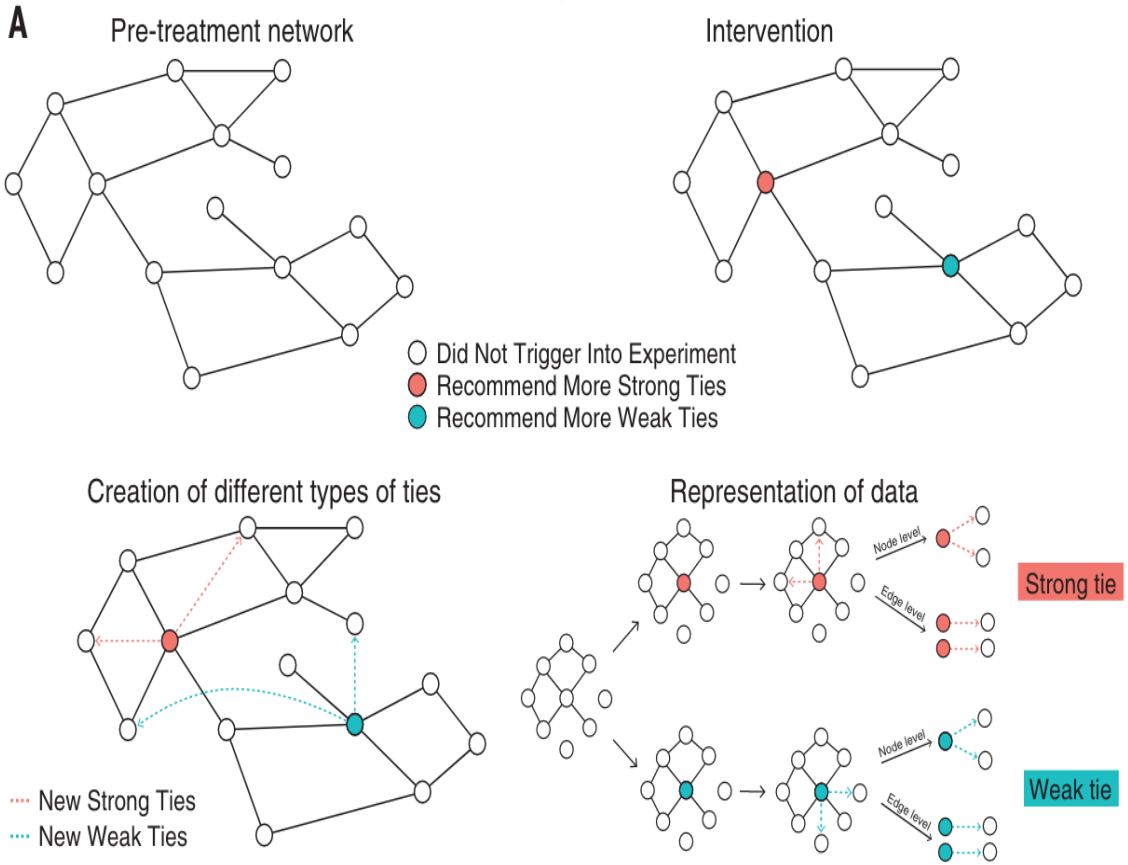
Social network analysis

Social networks and computational social science

- *Custommade vs. readymade data*:
  - Add Health data is custommade
  - Amazon book data is readymade
    - Ubiquity of large-scale, dynamic network datasets on the internet creates new opportunities
- *Computational approaches to network analysis*:
  - Bearman et al. 2004 demonstrate how simulations can reveal micro-mechanism underlying macro-structures
  - Advances in computational power enable new types of network simulation (see Block, Stadtfeld, and Snijders 2019)
    - Exponential random graph models (ERGMs)
    - Stochastic actor-oriented models (SOAMs)

Social networks and computational social science

Revisiting old theories with new data and methods



Rajkumar, Karthik, Guillaume Saint-Jacques, Iavor Bojinov, Erik Brynjolfsson, and Sinan Aral. 2022. "A Causal Test of the Strength of Weak Ties." *Science* 377:1304–10.

Social networks and computational social science

Advances in network modeling

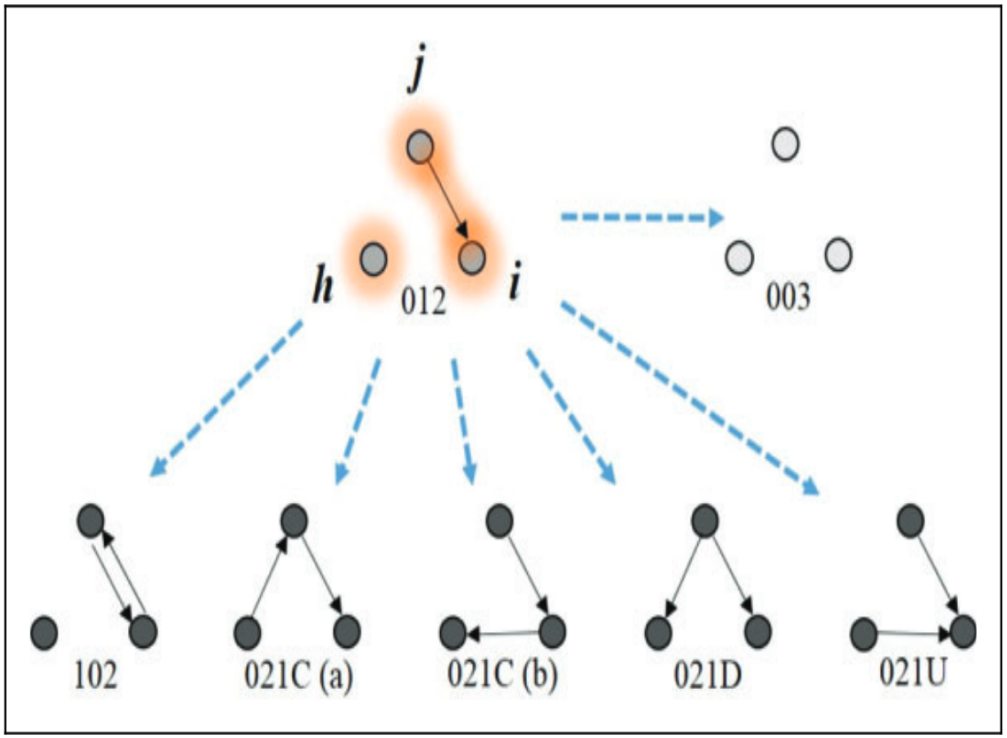


Figure 1. Possible transitions from triad 012.

Block, Stadtfeld, and Snijders 2019.

Part II: Data structures

Open RStudio and load `lecture2-data-structures.Rmd`, located in this week's Canvas Module. Scroll down to this line.

Object-oriented programming

- A paradigm of computer programming
  - We create *objects* of different *classes* such as numbers, strings, and data frames
  - These objects have *attributes*, properties such as data
    - e.g. The numeric object we call `A` has an attribute called `value` equal to `1`
  - Objects are associated with *methods* that allow us to manipulate them
    - e.g. a numeric object might have a method called `add`, such that `A + A` will return 2.

Basic types

There are four basic types we will be using throughout the class. Here I used them to record some information about one of my cats. In R, it is convention to use the <- operator to assign an object to a name.

```
# Character (also known as "strings")
name <- "Gary"
# Numeric
weight <- 13.2
# Integer ("int" for short)
age <- 5L
# Logical
human <- FALSE
```

The other two are called `complex` and `raw`. See documentation

Basic types

There are a few useful commands for inspecting objects.

```
print(name) # Prints value in console
```

```
## [1] "Gary"
```

```
class(name) # Shows class of object
```

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

```
typeof(name) # Shows type of object, not always equal to class
```

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

Basic types

```
print(weight) # Prints value in console
```

```
## [1] 13.2
```

```
class(weight) # Shows class of object
```

```
## [1] "numeric"
```

```
typeof(weight) # Shows type of object, not always equal to class
```

```
## [1] "double"
```

Basic types

We can use the == expression to verify the value of an object. We will discuss Boolean operations in more detail next lecture.

```
name == "Tabitha"
```

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

```
age == 3L
```

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

```
age >= 3L # is greater than
```

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

```
age != 3L # is not
```

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

Vectors

A vector is a collection of elements of the *same* type. We can define an empty vector with `N` elements of a type. Empty vectors assume certain default values depending on the type.

```
N <- 5
x <- logical(N)
print(x)
```

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

```
y <- numeric(N)
print(y)
```

```
## [1] 0 0 0 0 0
```

```
z <- character(N)
print(z)
```

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

Vectors

Let's take a closer look at numeric vectors. We can use the combine function `c()` to concatenate multiple values into a vector.

```
v1 <- c(1,2,3,4,5)
v2 <- c(1,1,1,1,1)
class(v1) # check the class of v1
```

```
## [1] "numeric"
```

Vectors

We can easily perform various mathematical operations on numeric vectors

```
v1 + v2 # addition
```

```
## [1] 2 3 4 5 6
```

```
v1 - v2 # subtraction
```

```
## [1] 0 1 2 3 4
```

```
v1 * v2 # multiplication
```

```
## [1] 1 2 3 4 5
```

```
sum(v1) # sum over v1
```

```
## [1] 15
```

Note how the different methods return different types of outputs. The arithmetic operations return vectors while `sum` returns a numeric value.

Vectors

What happens if we try to combine objects of different types using combine?

```
t <- c("cat", 5, TRUE)
typeof(t)
```

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

```
t
```

```
## [1] "cat"  "5"    "TRUE"
```

Vectors

There are lots of commands for generating special types of numeric vectors. Note how `N` has already been defined above.

```
seq(N) # generates a sequence from 1 to N
```

```
## [1] 1 2 3 4 5
```

```
rev(seq(N)) # reverses order
```

```
## [1] 5 4 3 2 1
```

```
rnorm(N) # samples N times from a normal distribution
```

```
## [1] 0.24968646 0.52157412 0.12241054 0.02406736 0.40940433
```

Vectors

We can use the help ? command to find information about each of these commands.

```
?seq
```

Vectors

We can use the `index` to access the specific elements of a vector. R uses square brackets for such indexing.

```
x <- rnorm(N)
print(x)
```

```
## [1] -1.007546736 -1.545895130  0.124763140 -0.192839704 -0.009919529
```

```
print(x[1]) # R indexing starts at 1; Python and some others start at 0
```

```
## [1] -1.007547
```

```
x[1] <- 9 # We can combine indexing with assignment to modify elements
print(x)
```

```
## [1]  9.000000000 -1.545895130  0.124763140 -0.192839704 -0.009919529
```

Vectors

The `head` and `tail` commands are useful when we're working with larger objects. Here we draw 10,000 observations from an normal distribution.

```
x <- rnorm(10000)
length(x)
```

```
## [1] 10000
```

```
head(x)
```

```
## [1]  0.1884328 -0.3262688  0.3019843  0.5101624 -1.4279502 -1.062462
```

```
tail(x)
```

```
## [1] -0.56416472  0.05099439  0.25569710 -0.37142275 -0.67521889 -0.7
```

Exercise: Vectors

Retrieve the final element from `x` using indexing.

Vectors

Vectors can also contain null elements to indicate missing values, represented by the NA symbol.

```
x <- c(1,2,3,4,NA)
is.na(x) # The is.na function indicates whether each value is missing.
```

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

```
!is.na(x) # Prepending ! denotes the inverse of a logical operation
```

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

NA is a logical type but can exist within numeric and character vectors. It is an exception to the rule discussed above regarding the presence of multiple types in the same vector.

Lists

A list is an object that can contain different types of elements, including basic types and vectors.

```
print(v1)
```

```
## [1] 1 2 3 4 5
```

```
l1 <- list(v1) # We can easily convert the vector v1 into a list.
print(l1)
```

```
## [[1]]
## [1] 1 2 3 4 5
```

Lists

Lists have a slightly different form of indexing. This can be one of the most confusing aspects of R for beginners!

```
l1[1] # The first element of the list contains the vector
```

```
## [[1]]
## [1] 1 2 3 4 5
```

```
l1[[1]] # Double brackets allows us to access the vector itself
```

```
## [1] 1 2 3 4 5
```

```
class(l1[1]) # first element is a list
```

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

```
class(l1[[1]]) # double indexing gives us the contents
```

```
## [1] "numeric"
```

Lists

We can access specific elements of a list by using standard indexing.

```
l1[[1]][1] # Followed by single brackets to access a specific element
```

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

```
l1[1][1] # If we're not careful, we will just get the entire sublist
```

```
## [[1]]
## [1] 1 2 3 4 5
```

Lists

We can easily combine multiple vectors into a list.

```
v.list <- list(v1,v2) # We could store both vectors in a list
print(v.list)
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] 1 1 1 1 1
```

Lists

We index sublists using double brackets, then specific elements with single brackets.

```
v.list[[2]][4] # We can use double brackets to get element 4 of list 2
```

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

Lists

We can make indexing easier if we start with an empty list and then add elements using a named index via the $ operator.

```
v <- list() # initialize empty list
v$v1 <- v1 # the $ sign is used for named indexing
v$v2 <- v2
print(v)
```

```
## $v1
## [1] 1 2 3 4 5
##
## $v2
## [1] 1 1 1 1 1
```

Excercise: Lists

Combine `$` and square bracket indexing to extract the 5th element of `v1` from the list `v`.

Lists

We can define lists more concisely by providing sublists as named arguments.

```
cats <- list(names = c("Gary", "Tabitha"), ages = c(5,2))
print(cats)
```

```
## $names
## [1] "Gary"    "Tabitha"
##
## $ages
## [1] 5 2
```

Matrices

A matrix is a two-dimensional data structure. Like vectors, matrices hold objects of a single type. Here we're defining a matrix using two arguments, the number of rows and columns.

```
matrix(nrow=5,ncol=5) # Here there is no content so the matrix is empty
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   NA   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA   NA
## [4,]   NA   NA   NA   NA   NA
## [5,]   NA   NA   NA   NA   NA
```

Matrices

We can also pass an argument to define the initial contents of a matrix.

```
M <- matrix(0L, nrow=5, ncol=5) # 5x5 matrix of zeros
M
```

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

Matrices

We can create a matrix by combining vectors using `cbind`.

```
M1 <- cbind(v1,v2) # Treat vectors a columns
print(M1)
```

```
##      v1 v2
## [1,]  1  1
## [2,]  2  1
## [3,]  3  1
## [4,]  4  1
## [5,]  5  1
```

Matrices

If we want to treat the vectors as rows, we alternatively use `rbind`. We could also get the same result by *transposing* `M1`.

```
M2 <- rbind(v1, v2) # Vectors as rows
print(M2)
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2    1    1    1    1    1
```

```
print(t(M1)) # t() is the transpose function
```

```
##    [,1] [,2] [,3] [,4] [,5]
## v1    1    2    3    4    5
## v2    1    1    1    1    1
```

Matrices

The `dim` function provides us with information about the dimensions of a given matrix. It returns the number of rows and columns.

```
dim(M1) # Shows the dimensions of the matrix
```

```
## [1] 5 2
```

```
dim(M2)
```

```
## [1] 2 5
```

Matrices

We can get particular values using two-dimensional indexing. By convention `i` denotes the row and `j` the column.

```
i <- 1 # row index
j <- 2 # column index
M1[i,j] # Returns element i,j
```

```
## v2
##  1
```

```
M1[i,] # Returns row i
```

```
## v1 v2
##  1  1
```

```
M1[,j] # Returns column i
```

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

Matrices

Like lists, we can also name rows and columns to help make indexing easier. The `colnames` and `rownames` functions show the names of each column and row.

```
colnames(M1)
```

```
## [1] "v1" "v2"
```

```
rownames(M1)
```

```
## NULL
```

Matrices

We can use these functions to assign new names.

```
colnames(M1) <- c("X", "Y")
rownames(M1) <- seq(1, nrow(M1))
print(M1)
```

```
##   X Y
## 1 1 1
## 2 2 1
## 3 3 1
## 4 4 1
## 5 5 1
```

Style

A note on style

- Not only do programming languages require a specific syntax to function, but there are also stylistic conventions
- There are packages you can use to automatically style your code (`styler` and `lintr`)
- See https://style.tidyverse.org/ for more info on R style

Style

Some style tips

- Naming
  - Use short, informative variable names
  - Use snake_case or CamelCase for multiple words
  - Maintain a consistent naming convention

Style

Some style tips

- Use appropriate spacing to make code readable
  - e.g. `a <- 1` is preferable to `a<-1`
- Try to avoid long expressions
  - Make complex functions modular (more next lecture)
  - Tidyverse uses the %>% operator to help with this (more next lecture)

Style

Some style tips

- Comment on your code for your future self and others
- In RMarkdown, write explanations outside of chunks
- ChatGPT and other AI can be helpful for explaining code and showing how to add useful comments

Use data structures to represent social networks

Storing names in vectors

Let's see how these types of objects can be used to create formal representations of social networks. We can start by defining a vector of names for a fictional friend group.

```
names <- c("Bob", "Alice", "Carol", "Ted")
```

Use data structures to represent social networks

Representing relationships in a matrix

Next, we can define a matrix to store relationship ties. Each person is referenced according to their index in the name vector

```
X <- matrix(0, nrow = length(names), ncol = length(names))
X[1,2] <- 1 # edge from Bob to Alice
X[1,3] <- 1
X[3,2] <- 1
X[1,4] <- 1
X[4,2] <- 1

print(X)
```

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

Use data structures to represent social networks

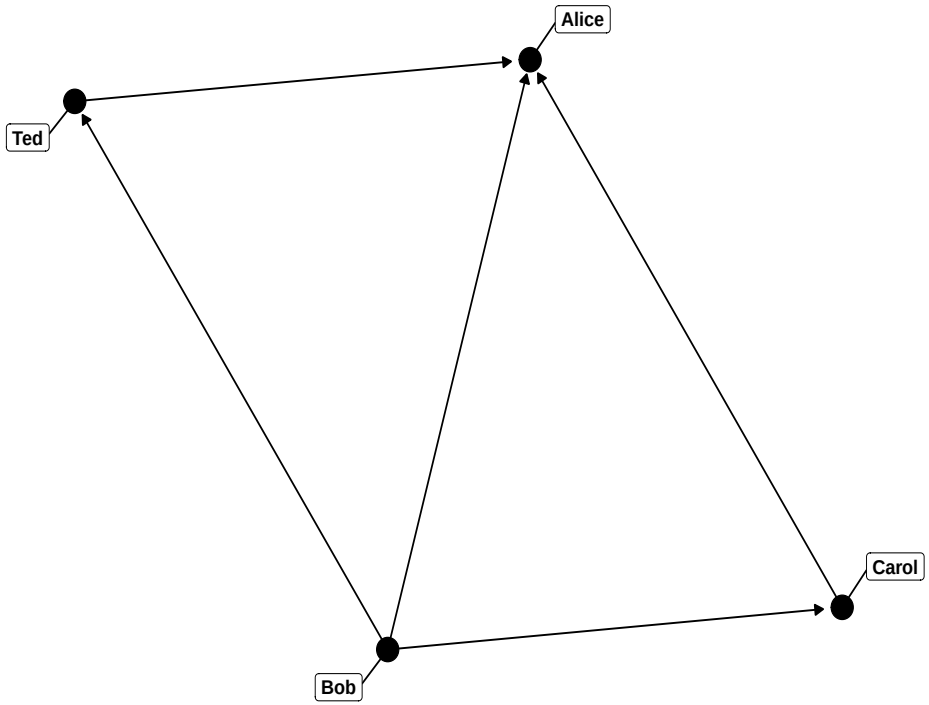
Storing information in a list

```
network_info <- list(names = names, network = X)
print(network_info)
```

```
## $names
## [1] "Bob"   "Alice" "Carol" "Ted"
##
## $network
##      [,1] [,2] [,3] [,4]
## [1,]    0    1    1    1
## [2,]    0    0    0    0
## [3,]    0    1    0    0
## [4,]    0    1    0    0
```

Putting it all together

Plotting the network



Next week

- Programming in R
  - Boolean logic
  - If-statements
  - Loops
  - Functions
- Agent-based modeling