Social Network Analysis Workshop

A Practical Introduction from Theory to Code

Dr Clemens Jarnach

2026-03-18

Table of contents

Introduction

This workshop material accompanies a two-day Grand Union DTP workshop on Social Network Analysis, taught by Dr Clemens Jarnach at the University of Oxford.

Networks are everywhere. The neurons firing as you read this sentence, the friends who shaped your career, the servers routing your emails, the supply chains that stocked your breakfast — all are networks. So too are the invisible webs of trust and obligation inside organisations, the chains of transmission that carry a virus from one city to another, and the recommendation algorithms that decide what you watch next.

Social Network Analysis (SNA) is the systematic study of these relational structures. Its core insight is deceptively simple: to understand the social world, we must look not only at *who* people are, but at *how they are connected*. Individual attributes — education, income, attitudes — take us only so far. The structure of relationships around a person shapes their opportunities, constraints, and behaviour in ways that attribute-based analysis cannot capture.

What makes SNA especially powerful is that connections and their *absence* both carry meaning. A gap between two otherwise dense clusters of contacts can be more consequential than any single tie. Structural holes, weak ties, brokerage positions — these concepts, which we will explore in depth, have proven explanatory across an extraordinary range of phenomena.

The field draws on an unusually broad intellectual inheritance: graph theory from mathematics, statistical mechanics from physics, computational methods from computer science, and relational theories of action from sociology. This interdisciplinary heritage is part of what makes it such a rich area of inquiry — and part of what makes it challenging to learn. This workshop is designed to give you a solid footing across all of it.

Over two days, we will move from foundations to application. We will work through the core theory — graph-theoretic concepts, network measures, structural principles — and then put it into practice using R, analysing real network data, building visualisations, and fitting statistical models. The examples throughout are drawn from published research: how mapping HIV transmission saves lives, why some TikTok videos go viral and others vanish, and yes, which Mafia soldier is statistically most likely to be murdered next.

By the end of the workshop, you will have the conceptual vocabulary and technical grounding to design and conduct your own network research — and to see the world a little differently in the process.

Objectives

This workshop introduces the core concepts and methods of social network analysis, from foundational graph theory and personal networks to contemporary network science approaches and statistical modelling. Participants will learn how to design and conduct network-based research, focusing on the analysis of social networks, i.e., sets of actors connected by relationships. By the end of the course, participants will have the conceptual and technical grounding to develop their own network research project and to think critically about the world through the lens of networks.

By the end of this workshop, participants will:

- Understand the theoretical foundations and key concepts of social network analysis
- Understand principles of research design and data collection for network studies
- Be able to apply computational methods to analyse network data in R
- Develop skills in visualising, modelling, and testing hypotheses using network data
- Engage critically with research in network science and its applications
- Able to formulate and theorise research questions using SNA approaches
- Design and conduct their own network-based research project

Why use R?

Conducting network analysis requires computational tools. Throughout this workshop we use **R**, which has become one of the richest environments for network analysis available. The core packages (e.g., `igraph`, `tidygraph`, `ggraph`, and `statnet`) support the full analytical pipeline: data construction and manipulation, descriptive analysis, visualisation, and statistical modelling. R's broader ecosystem (the tidyverse, spatial packages, multilevel modelling tools) also makes it straightforward to integrate network methods with the other quantitative approaches you likely already use in your research.

All workshop code, data, and exercises use R. If you have not yet installed the required packages, please do so before the first session:

```
install.packages(c(
  "tidyverse", "janitor", "readxl", "haven", "lubridate",
   ↪  "forcats",
  "igraph", "tidygraph", "ggraph", "statnet", "intergraph",
```

```
  "RColorBrewer", "viridis"
))
```

Key R Packages

R has a rich ecosystem for network analysis, spanning graph-theoretic computation, tidy data workflows, visualisation, and statistical modelling. The core packages for this workshop are:

- **Network data structures and algorithms:** igraph — the workhorse for network construction, manipulation, and computation of network measures
- **Tidy network analysis:** tidygraph — a tidy API for graph manipulation, fully compatible with the tidyverse
- **Network visualisation:** ggraph — `ggplot2`-based network visualisation built on top of `tidygraph`
- **Statistical network models:** statnet — a suite of packages for the statistical modelling of network data, including ERGMs
- **Network interoperability:** intergraph — converts network objects between `igraph` and `statnet`/`network` formats
- **Colour palettes:** RColorBrewer and viridis — perceptually uniform and accessible colour scales for network visualisation
- **General data wrangling:** tidyverse — for preparing and manipulating node and edge data prior to analysis

All packages can be installed in one go:

```
install.packages(c(
  "tidyverse", "janitor", "readxl", "haven", "lubridate",
  ↪  "forcats",
  "igraph", "tidygraph", "ggraph", "statnet", "intergraph",
  "RColorBrewer", "viridis"
))
```

Recommended Reading

Core Textbooks

Introductory and applied

- Borgatti, Stephen P., Martin G. Everett, Jeffrey C. Johnson, and Filip Agneessens. (2022) *Analyzing Social Networks Using R*. London: SAGE Publications. — The updated, R-focused companion to the 2013 original; covers research design through ERGMs. Recommended as the primary reference for this workshop.

- Luke, Douglas A. (2015). *A User's Guide to Network Analysis in R*. Cham: Springer. doi:10.1007/978-3-319-23883-8. — Accessible and practically oriented; good for getting up and running in R quickly.

- Borgatti, Stephen P., Martin G. Everett, and Jeffrey C. Johnson. (2013). *Analyzing Social Networks*. Los Angeles: SAGE. — The conceptual foundation for the 2022 volume; strong on theory and research design.

- Wasserman, Stanley, and Katherine Faust. (1997). *Social Network Analysis: Methods and Applications*. Cambridge: Cambridge University Press. — The canonical reference. Dense but authoritative on formal methods and notation.

- McLevey, John, John Scott, and Peter J. Carrington (eds.). (2024). *The Sage Handbook of Social Network Analysis*. Second edition. London ; Sage. — Comprehensive state-of-the-art overview across theory, methods, and applications; includes new chapters on computational social science, multilevel networks, and digital data.

Statistical modelling

- Avrachenkov, Konstantin, and Maximilien Dreveton. (2022). *Statistical Analysis of Networks. Now Publishers*. — Rigorous treatment of inferential approaches to network data; useful for the statistical modelling sessions.

- Lusher, Dean, Johan Koskinen, and Garry Robins. (2013). Exponential Random Graph Models for Social Networks : Theories, Methods, and Applications. Cambridge: Cambridge University Press.

- Rawlings, Craig M., Jeffrey A. Smith, James W. Moody, and Daniel McFarland. (2023) Network Analysis : Integrating Social Network Theory, Method, and Application with R. Cambridge: Cambridge University Press. https://inarwhal.github.io/NetworkAnalysisR-book/.

Visualisation

- Khokhar, Devangana. (2015). *Gephi Cookbook*. Birmingham: Packt Publishing. — Hands-on guide to network visualisation with Gephi; useful supplement for those who prefer a GUI workflow alongside R.

Online Tutorials and Resources

- **Ognyanova, Katya.** *Static and Dynamic Network Visualization with R.* kateto.net/network-visualization — Comprehensive and regularly updated tutorial covering `igraph`, `statnet`, and interactive visualisation. Highly recommended.
- **Ognyanova, Katya.** *Network Analysis with R and igraph* (NetSci X Tutorial). kateto.net/networks-r-igraph — A concise, hands-on introduction to `igraph`; good pre-workshop reading.
- **Rawlings, Craig, Jeffrey A. Smith, James Moody, and Daniel A. McFarland.** *Network Analysis: Integrating Social Network Theory, Method, and Application with R.* inarwhal.github.io/NetworkAnalysisR-book — A full open-access textbook integrating theory and R application.
- **INSNA — International Network for Social Network Analysis.** insna.org — International association for network researchers; resources, conference information, and links to the journal *Social Networks*.
- **Stanford Network Analysis Project (SNAP).** snap.stanford.edu — Large repository of network datasets and computational tools; useful for finding empirical data for your own projects.

1 What are Networks?

A brief history and definition of networks as objects of study

Networks are a way of thinking about the world that shifts our attention from individual entities to the relationships between them. Rather than asking *what* something is, network analysis asks *how it is connected* — and what those connections imply.

The nodes in a network can be almost anything: people, organisations, countries, genes, web pages, neurons. What unites them is the presence of ties — relationships, interactions, flows — that link one node to another. These ties interlink through shared nodes, creating chains and paths that connect parts of a system indirectly. This is part of what makes networks so powerful as an analytical concept: indirect connection provides a mechanism by which distant parts of a system can influence one another in ways that are invisible if we study actors in isolation.

When we talk about *social* networks specifically, nodes are typically active agents — individuals, teams, firms, cities — connected by social relationships: friendship, collaboration, communication, trade, trust, conflict. Networks of this kind operate at multiple levels simultaneously. At the **dyad** level, we study pairwise relationships: do actors who share business ties also develop affective ones? At the **node** level, we examine individual positions: do people with more connections tend to fare better in job markets? At the **network** level, we ask structural questions: do tightly connected networks diffuse information faster than sparse ones?

A central theoretical claim runs through nearly all of network science: an actor's position in a network shapes the constraints and opportunities they encounter. Structure is not merely descriptive — it is causally consequential. The same logic applies at the collective level. A sports team of talented individuals may still underperform if collaboration structures are poor. An organisation may fail to innovate not because its members lack creativity, but because information cannot flow across structural holes between teams.

Crucially, **absence** carries meaning too. Gaps between otherwise dense clusters — what Burt famously called structural holes — can be as consequential as ties themselves. The broker who spans a gap between two disconnected groups holds a position of informational and strategic advantage that no attribute-level analysis would reveal.

1.1 A Brief Intellectual History

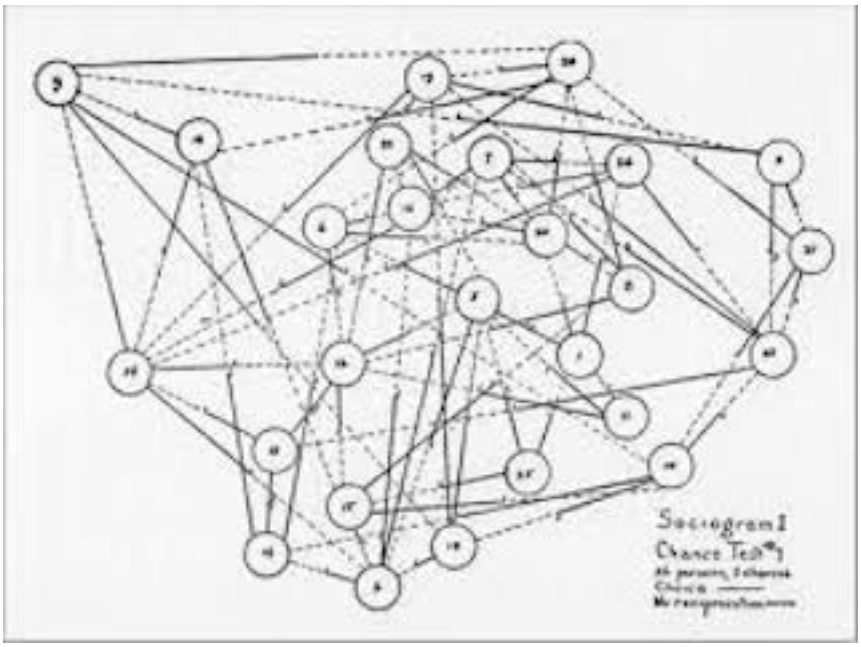
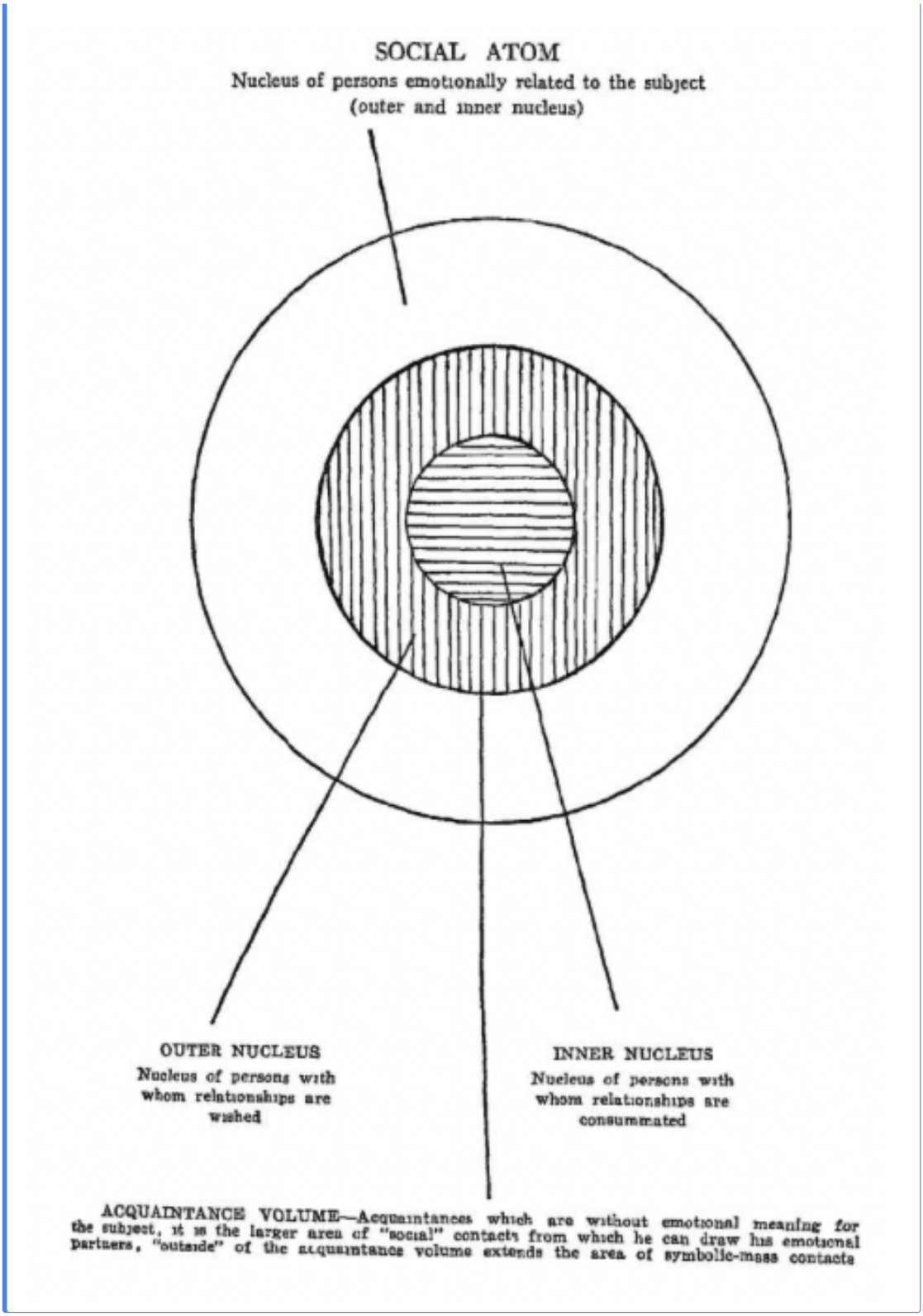


Figure 1.1: Moreno's Chance Sociogram (Moreno 1934)

Network analysis as a distinct scientific enterprise grew from contributions across multiple disciplines: graph theory and topology from mathematics, kinship systems from anthropology, and theories of social groups and process from sociology and psychology. Its modern form is often traced to Jacob Moreno's work in the first half of the 20th century (1934, 1951), who defined the study of social relations as *sociometry* and invented the *Sociogram* — a visual representation of relational structure, as well as the *Social Atom* model.

> **Moreno's Social Atom**
>
> Constructing a social atom involves using concentric circles to map varying degrees of relational proximity. The individual occupies the central nucleus, while the immediate surrounding layer contains intimate bonds like family and close friends. The outer ring encompasses the broader social network, including professional colleagues and acquaintances.



The field developed steadily across the twentieth century, with important contributions from sociology, political science, public health, and computer science. Interest exploded from the 1990s onwards, driven by three converging forces: influential new theories of network structure (small worlds, scale-free networks, preferential attachment); dramatic advances in computational power that made large-scale network analysis tractable; and the development of statistical models that moved the field beyond description toward formal inference.

2 Lecture Slides

Core concepts, fundamentals, and theoretical foundations

The lecture slides are distributed as a separate PDF deck. In this archived version, the full slides are available in the workshop repository at https://github.com/clemensjarnach/social_network_analysis_workshop (see the `files/` directory).

3 R Fundamentals

Essential programming skills for network analysis

Before we build networks, we need to ensure you're comfortable with R itself. This chapter introduces the R programming concepts you will need throughout the workshop. If you are already comfortable with R, treat it as a quick reference. If you are new to R, work through it carefully — the material here underpins everything that follows.

> **Running code**
>
> All code blocks in this workshop can be run interactively. Open the `.qmd` source file in RStudio, place your cursor inside a code chunk, and press **Ctrl+Enter** (Windows/Linux) or **Cmd+Enter** (Mac) to run a single line, or **Ctrl+Shift+Enter** to run the entire chunk.

3.1 Basic Operations and Assignment

R can be used as a calculator, but its real power lies in storing and manipulating values. The assignment operator `<-` binds a value to a name in the current environment.

```
1 + 3
```

```
[1] 4
```

```
# arithmetic evaluation
```

```
2^8             # exponentiation
```

```
[1] 256
```

```
17 %% 5          # modulo (remainder)
```

```
[1] 2
```

```
17 %/% 5         # integer division
```

```
[1] 3
```

```
a <- 9           # assignment: bind 9 to the name 'a'
```

```
sqrt(a)          # apply a function to an object
```

```
[1] 3
```

```
b <- sqrt(a)
```

```
a == b           # logical comparison: are they equal?
```

```
[1] FALSE
```

```
a != b           # not equal
```

```
[1] TRUE
```

```
a > 5            # greater than
```

```
[1] TRUE
```

```
ls()             # list all objects in the current environment
```

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

> **Note**
>
> R is case-sensitive: `A` and `a` are different objects. Object names can contain letters, digits, `.` and `_`, but must start with a letter or `.`.

3.2 Data Types

Every value in R has a type. The most common scalar types are:

```
  class(3.14)         # numeric (double)
```

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

```
  class(42L)          # integer (note: I used the L suffix to tell R
   ↪  this is an integer)
```

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

```
  class("hello")      # character
```

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

```
  class(TRUE)         # logical
```

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

```
  class(2 + 3i)       # complex
```

```
[1] "complex"
```

```
  # Type coercion
  as.numeric("3.14")
```

```
[1] 3.14
```

```
  as.integer(42)
```

```
[1] 42
```

```
  as.character(100)
```

```
[1] "100"
```

```
as.logical(0)       # 0 is FALSE; anything non-zero is TRUE
```

```
[1] FALSE
```

```
is.na(NA)           # test for missing value
```

```
[1] TRUE
```

3.3 Vectors

A vector is the fundamental data structure in R — an ordered collection of values of the **same type**. Almost all operations in R are vectorised, meaning they apply element-wise without explicit loops.

```
x <- c(1, 3, 5)          # combine values into a vector
y <- c("one", "three", "five")
```

```
# Indexing (1-based)
x[1]                     # first element
```

```
[1] 1
```

```
x[c(1, 3)]               # first and third elements
```

```
[1] 1 5
```

```
x[-2]                    # all except the second
```

```
[1] 1 5
```

```
# Sequences
a <- 1:10
b <- seq(from = 0, to = 1, by = 0.25)
c_rep <- rep(c(1, 2), times = 3)
c_rep
```

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

```
# Vectorised operations - applied element-wise
x * 2
```

```
[1]  2  6 10
```

```
x + c(10, 20, 30)
```

```
[1] 11 23 35
```

```
# Logical operations on vectors
x > 2
```

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

```
any(x > 2)                # is any element > 2?
```

```
[1] TRUE
```

```
all(x > 2)                # are all elements > 2?
```

```
[1] FALSE
```

```
which(x > 2)              # indices where condition is TRUE
```

```
[1] 2 3
```

```
# Common summary functions
length(x)
```

```
[1] 3
```

```
sum(x)
```

```
[1] 9
```

```
mean(x)
```

```
[1] 3
```

```
sd(x)
```

```
[1] 2
```

```
min(x); max(x)
```

```
[1] 1
```

```
[1] 5
```

3.4 Matrices

A matrix is a two-dimensional vector: all elements share the same type, and values are arranged in rows and columns. In network analysis, **adjacency matrices** are the canonical representation of graph structure — rows and columns correspond to nodes, and cell values indicate the presence (or weight) of an edge.

```
m <- matrix(1:25, nrow = 5, ncol = 5)
m
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

```
# Element access
m[1, 2]                    # row 1, column 2
```

```
[1] 6
```

```
m[1, ]                     # entire first row
```

```
[1]  1  6 11 16 21
```

```
  m[, 2]                    # entire second column
```

```
[1]  6  7  8  9 10
```

```
  m[2:3, 3:5]               # submatrix
```

```
     [,1] [,2] [,3]
[1,]   12   17   22
[2,]   13   18   23
```

```
  # Dimensions
  nrow(m); ncol(m)
```

```
[1] 5
```

```
[1] 5
```

```
  dim(m)
```

```
[1] 5 5
```

```
  # Matrix operations - essential for network mathematics
  t(m)                      # transpose
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]   16   17   18   19   20
[5,]   21   22   23   24   25
```

```
  m %*% t(m)                # matrix multiplication (not
   ↪  element-wise!)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]  855  910  965 1020 1075
[2,]  910  970 1030 1090 1150
[3,]  965 1030 1095 1160 1225
[4,] 1020 1090 1160 1230 1300
[5,] 1075 1150 1225 1300 1375
```

```
diag(m)                    # diagonal elements
```

```
[1]  1  7 13 19 25
```

```
# Build an adjacency matrix manually
adj <- matrix(0, nrow = 4, ncol = 4,
              dimnames = list(c("A","B","C","D"),
               ↪  c("A","B","C","D")))
adj["A", "B"] <- 1
adj["B", "C"] <- 1
adj["C", "D"] <- 1
adj["D", "A"] <- 1
adj                        # a 4-node directed cycle
```

```
  A B C D
A 0 1 0 0
B 0 0 1 0
C 0 0 0 1
D 1 0 0 0
```

3.5 Lists

Lists are R's most flexible container: each element can hold any object of any type or length. Many R functions return lists (model output, igraph objects, etc.), so you need to know how to navigate them.

```
person <- list(
  name    = "Alice",
  age     = 34,
  scores  = c(88, 92, 79),
  active  = TRUE
)
```

```
# Access by name
person$name
```

```
[1] "Alice"
```

```
  person[["age"]]
```

```
[1] 34
```

```
  # Access by position
  person[[3]]               # third element (the scores vector)
```

```
[1] 88 92 79
```

```
  person[[3]][2]            # second score
```

```
[1] 92
```

```
  # Inspect structure
  str(person)
```

```
List of 4
 $ name  : chr "Alice"
 $ age   : num 34
 $ scores: num [1:3] 88 92 79
 $ active: logi TRUE
```

```
  length(person)
```

```
[1] 4
```

```
  names(person)
```

```
[1] "name"   "age"    "scores" "active"
```

3.6 Data Frames

A data frame is the standard rectangular data structure in R: a list of vectors of equal length, where each vector is a column. Think of it as a spreadsheet or database table. Node attribute tables and edge lists are both naturally represented as data frames.

```r
df <- data.frame(
  id     = 1:5,
  name   = c("Alice", "Bob", "Clara", "David", "Eva"),
  degree = c(3, 7, 2, 5, 4),
  active = c(TRUE, TRUE, FALSE, TRUE, FALSE),
  stringsAsFactors = FALSE
)

df
```

```
  id  name degree active
1  1 Alice      3   TRUE
2  2   Bob      7   TRUE
3  3 Clara      2  FALSE
4  4 David      5   TRUE
5  5   Eva      4  FALSE
```

```r
# Column access
df$name
```

```
[1] "Alice" "Bob"   "Clara" "David" "Eva"
```

```r
df[["degree"]]
```

```
[1] 3 7 2 5 4
```

```r
# Row and column indexing
df[1, ]                  # first row
```

```
  id  name degree active
1  1 Alice      3   TRUE
```

```r
df[, 3]                  # third column
```

```
[1] 3 7 2 5 4
```

```r
df[df$active, ]          # rows where active == TRUE
```

```
  id  name degree active
1  1 Alice      3   TRUE
2  2   Bob      7   TRUE
4  4 David      5   TRUE
```

```
df[df$degree > 3, c("name", "degree")]
```

```
   name degree
2   Bob      7
4 David      5
5   Eva      4
```

```
# Useful inspection functions
nrow(df); ncol(df)
```

```
[1] 5
```

```
[1] 4
```

```
head(df, 3)
```

```
  id  name degree active
1  1 Alice      3   TRUE
2  2   Bob      7   TRUE
3  3 Clara      2  FALSE
```

```
str(df)
```

```
'data.frame':   5 obs. of  4 variables:
 $ id    : int  1 2 3 4 5
 $ name  : chr  "Alice" "Bob" "Clara" "David" ...
 $ degree: num  3 7 2 5 4
 $ active: logi  TRUE TRUE FALSE TRUE FALSE
```

```
summary(df)
```

```
       id        name               degree      active
 Min.   :1   Length:5           Min.   :2.0   Mode :logical
 1st Qu.:2   Class :character   1st Qu.:3.0   FALSE:2
 Median :3   Mode  :character   Median :4.0   TRUE :3
 Mean   :3                      Mean   :4.2
```

```
 3rd Qu.:4                       3rd Qu.:5.0
 Max.   :5                       Max.   :7.0
```

3.7 Control Flow

3.7.1 Conditionals

```r
x <- 15

if (x > 10) {
  cat("x is greater than 10\n")
} else if (x == 10) {
  cat("x is exactly 10\n")
} else {
  cat("x is less than 10\n")
}
```

```
x is greater than 10
```

```r
# ifelse() - vectorised conditional
scores <- c(55, 72, 48, 91, 60)
ifelse(scores >= 60, "pass", "fail")
```

```
[1] "fail" "pass" "fail" "pass" "pass"
```

3.7.2 Loops

In R, explicit loops are often avoidable thanks to vectorisation, but they are useful for iterative tasks and are worth knowing.

```r
# for loop
for (i in 1:5) {
  cat("Iteration:", i, "\n")
}
```

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

```
# while loop
count <- 0
while (count < 3) {
  count <- count + 1
  cat("count is", count, "\n")
}
```

```
count is 1
count is 2
count is 3
```

```
# break and next
for (i in 1:10) {
  if (i == 4) next       # skip this iteration
  if (i == 7) break      # exit the loop
  cat(i, "")
}
```

```
1 2 3 5 6
```

3.7.3 Apply Functions

The `apply` family offers a concise way to apply a function over a vector, list, or matrix — often replacing explicit loops.

```
m <- matrix(1:12, nrow = 3)
```

```
apply(m, 1, sum)          # row sums
```

```
[1] 22 26 30
```

```
apply(m, 2, mean)         # column means
```

```
[1]  2  5  8 11
```

```r
nums <- list(a = 1:4, b = 5:10, c = 11:15)
sapply(nums, mean)        # returns a named vector
```

```
   a    b    c
 2.5  7.5 13.0
```

```r
lapply(nums, length)      # returns a list
```

```
$a
[1] 4

$b
[1] 6

$c
[1] 5
```

3.8 Writing Functions

Functions are the building blocks of reusable code. When you find yourself repeating the same operations, write a function.

```r
# Basic function definition
greet <- function(name, greeting = "Hello") {
  paste(greeting, name)
}

greet("Alice")
```

```
[1] "Hello Alice"
```

```r
greet("Bob", greeting = "Welcome")
```

```
[1] "Welcome Bob"
```

```r
# A more practical example: normalise a vector to [0, 1]
normalise <- function(x) {
```

```
  (x - min(x)) / (max(x) - min(x))
}
```

```
centralityScores <- c(3, 7, 2, 9, 4)
normalise(centralityScores)
```

```
[1] 0.1428571 0.7142857 0.0000000 1.0000000 0.2857143
```

```
# Functions can return multiple values via a list
describe <- function(x) {
  list(n = length(x), mean = mean(x), sd = sd(x), range = range(x))
}

describe(centralityScores)
```

```
$n
[1] 5

$mean
[1] 5

$sd
[1] 2.915476

$range
[1] 2 9
```

3.9 Data Wrangling with the Tidyverse

The tidyverse is a collection of R packages designed around a consistent philosophy of tidy data and readable code. The core package for data manipulation is **dplyr**, which provides a small set of verbs that cover the vast majority of data-wrangling tasks.

```
library(tidyverse)
```

3.9.1 The Pipe Operator

The pipe `|>` (base R, since 4.1) or `%>%` (magrittr/tidyverse) passes the result of one expression as the first argument of the next. It allows you to write a sequence of transformations in the order they happen, which is much easier to read than nested function calls.

```
# Without pipe - read inside out
round(mean(c(1, 2, 3, 4, 5)), digits = 2)
```

```
[1] 3
```

```
# With pipe - read left to right
c(1, 2, 3, 4, 5) |> mean() |> round(digits = 2)
```

```
[1] 3
```

3.9.2 A Working Dataset

We'll use a small node-attribute table representing actors in a network.

```
nodes <- tibble(
  id         = 1:8,
  name       = c("Alice","Bob","Clara","David","Eva","Frank",
   ↪  "Grace","Hugo"),
  department = c("Research","Operations","Research","HR",
   ↪  "Operations","Research","HR","Operations"),
  tenure     = c(5, 2, 8, 3, 6, 1, 4, 7),
  score      = c(72, 85, 91, 60, 78, 55, 88, 74)
)
```

```
nodes
```

```
# A tibble: 8 x 5
     id name  department tenure score
  <int> <chr> <chr>       <dbl> <dbl>
1     1 Alice Research        5    72
2     2 Bob   Operations      2    85
3     3 Clara Research        8    91
4     4 David HR              3    60
5     5 Eva   Operations      6    78
```

```
6     6 Frank Research        1    55
7     7 Grace HR              4    88
8     8 Hugo  Operations      7    74
```

3.9.3 filter() — Keep Rows

```
# Keep only Research department members
nodes |> filter(department == "Research")
```

```
# A tibble: 3 x 5
     id name  department tenure score
  <int> <chr> <chr>       <dbl> <dbl>
1     1 Alice Research        5    72
2     3 Clara Research        8    91
3     6 Frank Research        1    55
```

```
# Multiple conditions
nodes |> filter(department == "Operations", tenure > 4)
```

```
# A tibble: 2 x 5
     id name  department tenure score
  <int> <chr> <chr>       <dbl> <dbl>
1     5 Eva   Operations      6    78
2     8 Hugo  Operations      7    74
```

```
# Using %in% for multiple values
nodes |> filter(department %in% c("HR", "Operations"))
```

```
# A tibble: 5 x 5
     id name  department tenure score
  <int> <chr> <chr>       <dbl> <dbl>
1     2 Bob   Operations      2    85
2     4 David HR              3    60
3     5 Eva   Operations      6    78
4     7 Grace HR              4    88
5     8 Hugo  Operations      7    74
```

3.9.4 select() — Choose Columns

```
nodes |> select(name, department, score)
```

```
# A tibble: 8 x 3
  name  department score
  <chr> <chr>      <dbl>
1 Alice Research      72
2 Bob   Operations    85
3 Clara Research      91
4 David HR            60
5 Eva   Operations    78
6 Frank Research      55
7 Grace HR            88
8 Hugo  Operations    74
```

```
# Drop a column with -
nodes |> select(-id)
```

```
# A tibble: 8 x 4
  name  department tenure score
  <chr> <chr>       <dbl> <dbl>
1 Alice Research        5    72
2 Bob   Operations      2    85
3 Clara Research        8    91
4 David HR              3    60
5 Eva   Operations      6    78
6 Frank Research        1    55
7 Grace HR              4    88
8 Hugo  Operations      7    74
```

```
# Rename while selecting
nodes |> select(name, dept = department, performance = score)
```

```
# A tibble: 8 x 3
  name  dept       performance
  <chr> <chr>            <dbl>
1 Alice Research            72
2 Bob   Operations          85
3 Clara Research            91
4 David HR                  60
5 Eva   Operations          78
```

```
6 Frank Research            55
7 Grace HR                  88
8 Hugo  Operations          74
```

3.9.5 mutate() — Create or Modify Columns

```
nodes |>
  mutate(
    senior   = tenure >= 5,
    score_z  = (score - mean(score)) / sd(score),   # standardise
    label    = paste0(name, " (", department, ")")
  )
```

```
# A tibble: 8 x 8
     id name  department tenure score senior score_z label
  <int> <chr> <chr>       <dbl> <dbl> <lgl>    <dbl> <chr>
1     1 Alice Research        5    72 TRUE    -0.261 Alice (Research)
2     2 Bob   Operations      2    85 FALSE    0.745 Bob (Operations)
3     3 Clara Research        8    91 TRUE     1.21  Clara (Research)
4     4 David HR              3    60 FALSE   -1.19  David (HR)
5     5 Eva   Operations      6    78 TRUE     0.203 Eva (Operations)
6     6 Frank Research        1    55 FALSE   -1.58  Frank (Research)
7     7 Grace HR              4    88 FALSE    0.977 Grace (HR)
8     8 Hugo  Operations      7    74 TRUE    -0.106 Hugo (Operations)
```

3.9.6 arrange() — Sort Rows

```
nodes |> arrange(desc(score))          # highest score first
```

```
# A tibble: 8 x 5
     id name  department tenure score
  <int> <chr> <chr>       <dbl> <dbl>
1     3 Clara Research        8    91
2     7 Grace HR              4    88
3     2 Bob   Operations      2    85
4     5 Eva   Operations      6    78
5     8 Hugo  Operations      7    74
6     1 Alice Research        5    72
7     4 David HR              3    60
8     6 Frank Research        1    55
```

```
nodes |> arrange(department, tenure)   # sort by two columns
```

```
# A tibble: 8 x 5
     id name  department tenure score
  <int> <chr> <chr>       <dbl> <dbl>
1     4 David HR              3    60
2     7 Grace HR              4    88
3     2 Bob   Operations      2    85
4     5 Eva   Operations      6    78
5     8 Hugo  Operations      7    74
6     6 Frank Research        1    55
7     1 Alice Research        5    72
8     3 Clara Research        8    91
```

3.9.7 summarise() and group_by() — Aggregation

```
# Overall summary
nodes |>
  summarise(
    n          = n(),
    mean_score = mean(score),
    sd_score   = sd(score),
    max_tenure = max(tenure)
  )
```

```
# A tibble: 1 x 4
      n mean_score sd_score max_tenure
  <int>      <dbl>    <dbl>      <dbl>
1     8       75.4     12.9          8
```

```
# Summary by group - critical for network attribute analysis
nodes |>
  group_by(department) |>
  summarise(
    n           = n(),
    mean_score = mean(score),
    mean_tenure = mean(tenure)
  ) |>
  arrange(desc(mean_score))
```

```
# A tibble: 3 x 4
```

```
  department     n mean_score mean_tenure
  <chr>      <int>      <dbl>       <dbl>
1 Operations     3       79          5
2 HR             2       74          3.5
3 Research       3       72.7        4.67
```

3.9.8 Joining Tables

Joins combine two data frames on a shared key column — essential when merging node attributes with an edge list.

```
# Edge list: pairs of node IDs representing connections
edges <- tibble(
  from = c(1, 1, 2, 3, 4, 5, 6),
  to   = c(2, 3, 4, 5, 6, 7, 8),
  weight = c(1.2, 0.8, 1.5, 0.5, 2.0, 1.1, 0.9)
)
```

```
# Attach sender attributes to edge list
edges |>
  left_join(nodes |> select(id, name, department),
            by = c("from" = "id")) |>
  rename(from_name = name, from_dept = department)
```

```
# A tibble: 7 x 5
   from    to weight from_name from_dept
  <dbl> <dbl>  <dbl> <chr>     <chr>
1     1     2    1.2 Alice     Research
2     1     3    0.8 Alice     Research
3     2     4    1.5 Bob       Operations
4     3     5    0.5 Clara     Research
5     4     6    2   David     HR
6     5     7    1.1 Eva       Operations
7     6     8    0.9 Frank     Research
```

3.9.9 Reshaping Data

```
# Wide to long (pivot_longer) - useful for comparing multiple
 ↪ metrics
nodes |>
```

```
    select(name, tenure, score) |>
    pivot_longer(cols = c(tenure, score),
                 names_to  = "metric",
                 values_to = "value")
```

```
# A tibble: 16 x 3
   name  metric value
   <chr> <chr>  <dbl>
 1 Alice tenure     5
 2 Alice score     72
 3 Bob   tenure     2
 4 Bob   score     85
 5 Clara tenure     8
 6 Clara score     91
 7 David tenure     3
 8 David score     60
 9 Eva   tenure     6
10 Eva   score     78
11 Frank tenure     1
12 Frank score     55
13 Grace tenure     4
14 Grace score     88
15 Hugo  tenure     7
16 Hugo  score     74
```

```
  # Long to wide (pivot_wider)
  nodes |>
    select(name, department) |>
    mutate(present = 1) |>
    pivot_wider(names_from = department, values_from = present,
     ↪  values_fill = 0)
```

```
# A tibble: 8 x 4
  name  Research Operations    HR
  <chr>    <dbl>      <dbl> <dbl>
1 Alice        1          0     0
2 Bob          0          1     0
3 Clara        1          0     0
4 David        0          0     1
5 Eva          0          1     0
6 Frank        1          0     0
7 Grace        0          0     1
8 Hugo         0          1     0
```

3.10 Reading and Writing Data

```r
# CSV files
df <- read_csv("data/nodes.csv")            # tidyverse
 ↪  (recommended)
df <- read.csv("data/nodes.csv")            # base R
```

```r
write_csv(df, "data/nodes_clean.csv")
```

```r
# Excel files
library(readxl)
df <- read_excel("data/data.xlsx", sheet = 1)
```

```r
# SPSS, Stata, SAS
library(haven)
df <- read_spss("data/survey.sav")
df <- read_dta("data/survey.dta")
```

```r
# R's native format
saveRDS(df, "data/df.rds")
df <- readRDS("data/df.rds")
```

3.11 Quick Reference

Task	Function
Assign a value	`x <- value`
Create a vector	`c(1, 2, 3)`
Create a sequence	`1:10`, `seq(0, 1, 0.1)`
Create a matrix	`matrix(data, nrow, ncol)`
Create a data frame	`data.frame()` / `tibble()`
Check an object's type	`class()`, `typeof()`

Task	Function
Check dimensions	`dim()`, `nrow()`, `ncol()`, `length()`
Inspect structure	`str()`, `summary()`, `head()`
Filter rows	`filter()`
Select columns	`select()`
Create columns	`mutate()`
Sort rows	`arrange()`
Aggregate	`group_by()` + `summarise()`
Join tables	`left_join()`, `inner_join()`
Reshape wide → long	`pivot_longer()`
Reshape long → wide	`pivot_wider()`

4 Creating Your First Network

From data to graph objects: construction and visualisation with igraph

4.1 Introduction

Networks live as graph objects in R. This tutorial walks through the fundamentals of creating them from scratch, visualising them in different ways, and incrementally adding complexity. Using the `igraph` package, we will build a network incrementally, starting with isolated clusters and progressively adding connections, colour coding, and degree-based sizing.

4.2 Setup

Load the required package:

```
library(igraph)
```

4.3 Building the Network

We'll construct a simple network with 8 nodes and two clusters: one tightly connected triad (nodes 1–3) and a loosely connected cluster (nodes 4–8) with an initial internal structure.

```
set.seed(42)
g <- graph.empty(n = 8, directed = FALSE)
```

```
edges <- matrix(
  c(1, 2,  1, 3,  2, 3,  4, 5,  5, 6,  6, 7,  5, 7,  5, 8),
```

```
  ncol = 2, byrow = TRUE
)
```

```
g <- add_edges(g, as.vector(t(edges)))
```

```
## Inspect structure
print(g)
```

```
IGRAPH 7a4e9ed U--- 8 8 --
+ edges from 7a4e9ed:
[1] 1--2 1--3 2--3 4--5 5--6 6--7 5--7 5--8
```

```
cat("\nEdge list:\n")
```

```
Edge list:
```

```
print(edges)
```

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

The network now has: - A complete triangle (nodes 1, 2, 3) - A second cluster (nodes 4–8) with internal ties but no external connections - 8 edges total

4.4 Fixed Layout for Consistency

For the first three visualisations, we'll compute the layout once and apply it to all subgraphs. This prevents nodes from "jumping" as we add or remove elements.

```
layout_fixed <- layout_with_fr(g)
```

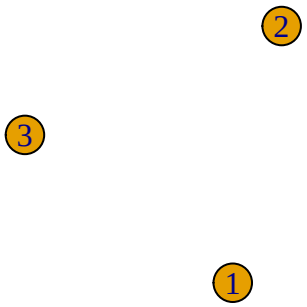
The Fruchterman–Reingold algorithm spaces nodes based on attractive and repulsive forces, producing an intuitive arrangement. By fixing this layout before subsetting, we ensure nodes remain in their original positions across plots.

4.5 Plot 1: Isolated Triangle (No Edges)

Visualise only nodes 1–3, with edge colour set to `NA` to suppress edge rendering:

```
g1 <- induced_subgraph(g, c(1, 2, 3))
layout_g1 <- layout_fixed[c(1, 2, 3), ]

plot(
  g1,
  main = "Plot 1: Three Nodes (No Edges)",
  vertex.size = 30,
  edge.color = NA,
  layout = layout_g1
)
```

Plot 1: Three Nodes (No Edges)



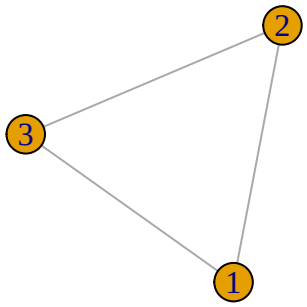
4.6 Plot 2: Triangle with Edges (Undirected)

Now reveal the edges connecting the three nodes:

```
plot(
  g1,
  main = "Plot 2: Three Nodes (With Edges)",
  vertex.size = 30,
```

```
  layout = layout_g1
)
```

Plot 2: Three Nodes (With Edges)

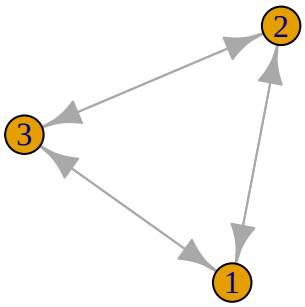


4.7 Plot 3: Triangle as Directed Graph

Convert the undirected graph to directed. By default, each undirected edge becomes a pair of directed edges:

```
g1_directed <- as.directed(g1)

plot(
  g1_directed,
  main = "Plot 3: Three Nodes (Directed)",
  vertex.size = 30,
  layout = layout_g1
)
```

Plot 3: Three Nodes (Directed)

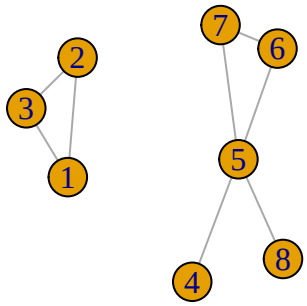


4.8 Plot 4: Two Clusters (Disconnected)

Return to the full graph with all 8 nodes. At this stage, we have two separate clusters:

```
g3 <- g

plot(
  g3,
  main = "Plot 4: Two Clusters (Disconnected)",
  vertex.size = 30,
  layout = layout_fixed
)
```

Plot 4: Two Clusters (Disconnected)



Nodes 1–3 form a tight triangle; nodes 4–8 form a larger, sparser cluster. There are no edges between the two groups.

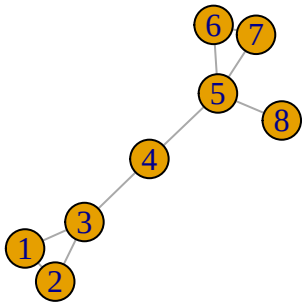
4.9 Plot 5: Bridging the Clusters

Add a single edge between node 3 (cluster 1) and node 4 (cluster 2) to bridge the two components:

```
g4 <- add_edges(g, c(3, 4))
layout_g4 <- layout_with_fr(g4)

plot(
  g4,
  main = "Plot 5: Clusters Connected",
  vertex.size = 30,
  layout = layout_g4
```

```
)
```

Plot 5: Clusters Connected



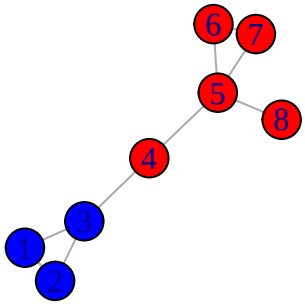
The network is now fully connected. We recompute the layout here because the new edge changes the structural relationships; earlier fixed layouts no longer reflect the updated topology.

4.10 Plot 6: Colour-Coded Clusters

Assign colours based on original cluster membership: blue for the first cluster (nodes 1–3), red for the second (nodes 4–8):

```
colors <- c("blue", "blue", "blue", "red", "red", "red", "red",
 ↪  "red")

plot(
  g4,
  main = "Plot 6: Colour-Coded Clusters",
  vertex.size = 30,
  vertex.color = colors,
  layout = layout_g4
)
```

Plot 6: Colour–Coded Clusters



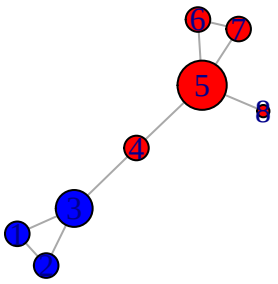
Colour is now a visual encoding of cluster membership, making the original structure immediately apparent despite the bridging edge.

4.11 Plot 7: Node Size by Degree Centrality

Resize each node proportionally to its degree (number of incident edges). This highlights which nodes are most connected:

```
deg <- degree(g4)

plot(
  g4,
  main = "Plot 7: Node Size by Degree",
  vertex.size = deg * 10,
  vertex.color = colors,
  layout = layout_g4
)
```

Plot 7: Node Size by Degree



Node 5 is now visibly larger (degree = 4), followed by nodes 3 and 4 (degree = 3 each). This size-encoding provides an immediate visual cue to network centrality.

4.12 Summary

This tutorial has introduced key concepts in network visualisation:

1. **Fixed layouts** preserve spatial consistency when subsetting or modifying subgraphs
2. **Colour coding** reveals structural properties such as cluster membership
3. **Node sizing** encodes quantitative attributes like centrality measures
4. **Edge presence/absence and directionality** control what relationships are displayed
5. **Progressive construction** builds intuition by incrementally adding complexity

These techniques are foundational for exploratory network analysis and communicating network structure to stakeholders.

5 Working with Network Data

Importing, cleaning, and managing network data from multiple sources

We will not always generate our own graph objects. As social scientists, we usually want to analyse social networks observed in the real world, so we need to learn how to deal with real network data, which rarely arrives in a tidy format. This chapter addresses the practical challenge of converting raw data — such as edge lists from APIs, adjacency matrices from surveys, and bipartite data from affiliations — into usable network objects in R. You will learn how to import, validate, and manipulate network data from multiple sources.

5.1 Setup

5.1.1 Loading Packages

```
library(igraph)
library(tidyverse)
library(tidygraph)
library(ggraph)
library(ergm)      # provides florentine dataset as network class
 ↪ objects
library(network)   # underlies ergm; provides as.matrix() for
 ↪ network objects
```

5.2 Data Formats: A Conceptual Overview

Network data arrives in three common representations, each with distinct structural properties and appropriate use cases:

Format	Structure	Typical Source
Adjacency Matrix	$n \times n$ matrix; cell $(i, j) = 1$ if an edge exists	Survey instruments, sociometric data
Edge List	Two-column data frame of sender–receiver pairs	API exports, relational databases, CSV files
Incidence Matrix	$n \times m$ matrix linking actors to events	Affiliation records, membership data

The choice of representation is rarely arbitrary: it reflects the data collection instrument, the storage system, and the analytical tradition of the field. Familiarity with all three formats — and the conversion pathways between them — is essential for applied network research.

5.3 Part 1: Adjacency Matrices — The Florentine Families

5.3.1 The Dataset

The Florentine families network is a canonical dataset in social network analysis, originally compiled by Padgett and Ansell (**padgett1993robust?**) from historical records of fifteenth-century Florence. It encodes marriage and business alliances among 16 elite Florentine families during the Medici's rise to political dominance. The dataset has become a benchmark for studying how social structure shapes political outcomes.

The `ergm` package provides this data as a pair of `network` class objects — the native format of the `statnet` software suite — giving us a realistic starting point for demonstrating data conversion.

```
# Load the florentine families data
data(florentine)

# Two network objects are now available:
# flomarriage  - marriage alliance network
# flobusiness  - business partnership network

class(flomarriage)
```

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

```
# Inspect the network object's summary
flomarriage
```

```
 Network attributes:
  vertices = 16
  directed = FALSE
  hyper = FALSE
  loops = FALSE
  multiple = FALSE
  bipartite = FALSE
  total edges= 20
    missing edges= 0
    non-missing edges= 20

 Vertex attribute names:
    priorates totalties vertex.names wealth

No edge attributes
```

The printed summary reveals key structural properties: 16 vertices, 20 edges, undirected, and unweighted. It also lists vertex attributes stored within the `network` object — including `wealth` and `priorates` (number of seats held on the city council) — which we will extract shortly.

5.3.2 Extracting the Adjacency Matrix

The `network` class is the native format of the `statnet` suite but is not directly compatible with `igraph`. The standard conversion pathway proceeds via the adjacency matrix: we extract it using `as.matrix()` and then pass it to `igraph`.

```
# Extract the adjacency matrix from the network object
flo_mat <- as.matrix(flomarriage)

# Inspect structure
dim(flo_mat)
```

```
[1] 16 16
```

```
flo_mat[1:6, 1:6]
```

```
           Acciaiuoli Albizzi Barbadori Bischeri Castellani Ginori
Acciaiuoli          0       0         0        0          0      0
```

```
Albizzi             0       0       0       0       0       1
Barbadori           0       0       0       0       1       0
Bischeri            0       0       0       0       0       0
Castellani          0       0       1       0       0       0
Ginori              0       1       0       0       0       0
```

The matrix is symmetric (confirming undirected ties), with 1s indicating marriage alliances and 0s indicating no recorded alliance. Row and column names are automatically preserved from the `network` object's vertex names.

5.3.3 Converting to igraph

`igraph`'s `graph_from_adjacency_matrix()` handles this conversion directly. The `mode` argument controls directionality: `"undirected"` collapses the symmetric matrix into a single edge per pair and ignores the diagonal.

```
g_flo <- graph_from_adjacency_matrix(
  flo_mat,
  mode     = "undirected",
  weighted = NULL,
  diag     = FALSE
)

g_flo
```

```
IGRAPH c3716bd UN-- 16 20 --
+ attr: name (v/c)
+ edges from c3716bd (vertex names):
 [1] Acciaiuoli--Medici       Albizzi   --Ginori       Albizzi
  ↪  --Guadagni
 [4] Albizzi   --Medici       Barbadori --Castellani   Barbadori
  ↪  --Medici
 [7] Bischeri  --Guadagni     Bischeri  --Peruzzi      Bischeri
  ↪  --Strozzi
[10] Castellani--Peruzzi      Castellani--Strozzi      Guadagni
 ↪  --Lamberteschi
[13] Guadagni  --Tornabuoni   Medici    --Ridolfi      Medici
 ↪  --Salviati
[16] Medici    --Tornabuoni   Pazzi     --Salviati     Peruzzi
 ↪  --Strozzi
[19] Ridolfi   --Strozzi      Ridolfi   --Tornabuoni
```

The printed summary confirms the correct structure: 16 vertices, 20 edges, undirected, unweighted. Node names are automatically imported from the matrix dimnames.

```
# Verify that vertex names are preserved
V(g_flo)$name
```

```
 [1] "Acciaiuoli"   "Albizzi"      "Barbadori"    "Bischeri"
  ↪  "Castellani"
 [6] "Ginori"       "Guadagni"     "Lamberteschi" "Medici"       "Pazzi"
[11] "Peruzzi"      "Pucci"        "Ridolfi"      "Salviati"
 ↪  "Strozzi"
[16] "Tornabuoni"
```

Direct Conversion with `intergraph`

An alternative one-step conversion from `network` to `igraph` is available via the `intergraph` package:

```
# install.packages("intergraph")
 g_flo <- intergraph::asIgraph(flomarriage)
```

5.3.4 Extracting Node Attributes from the Source Object

Before converting to `tbl_graph`, we extract the vertex-level attributes from the original `network` object. These are stored internally and accessible via `network::get.vertex.attribute()`.

```
# Extract vertex attributes from the network object into a tibble
family_meta <- tibble(
  name      = network::get.vertex.attribute(flomarriage,
   ↪  "vertex.names"),
  wealth    = network::get.vertex.attribute(flomarriage, "wealth"),
  priorates = network::get.vertex.attribute(flomarriage,
   ↪  "priorates")
) |>
  mutate(
    # Simplified political faction grouping for illustration
    faction = case_when(
      name == "Medici"
 ↪  ~ "Medici Alliance",
      name %in% c("Albizzi", "Pazzi", "Strozzi",
                  "Castellani", "Bischeri", "Peruzzi", "Guadagni")
                   ↪  ~ "Oligarch",
```

```
      TRUE
       ↪  ~ "Other"
    )
  )

family_meta
```

```
# A tibble: 16 x 4
   name         wealth priorates faction
   <chr>         <dbl>     <dbl> <chr>
 1 Acciaiuoli       10        53 Other
 2 Albizzi          36        65 Oligarch
 3 Barbadori        55         0 Other
 4 Bischeri         44        12 Oligarch
 5 Castellani       20        22 Oligarch
 6 Ginori           32         0 Other
 7 Guadagni          8        21 Oligarch
 8 Lamberteschi     42         0 Other
 9 Medici          103        53 Medici Alliance
10 Pazzi            48         0 Oligarch
11 Peruzzi          49        42 Oligarch
12 Pucci             3         0 Other
13 Ridolfi          27        38 Other
14 Salviati         10        35 Other
15 Strozzi         146        74 Oligarch
16 Tornabuoni       48         0 Other
```

5.3.5 Converting to tbl_graph and Joining Attributes

For tidyverse-compatible workflows, we convert the `igraph` object to a `tbl_graph` and join the metadata table using standard `dplyr` syntax. Within a `tbl_graph`, `activate(nodes)` or `activate(edges)` switches the active context for subsequent operations.

```
tg_flo <- as_tbl_graph(g_flo) |>
  activate(nodes) |>
  left_join(family_meta, by = "name")

tg_flo
```

```
# A tbl_graph: 16 nodes and 20 edges
#
# An undirected simple graph with 2 components
```

```
#
# A tibble: 16 x 4
  name       wealth priorates faction
  <chr>       <dbl>     <dbl> <chr>
1 Acciaiuoli     10        53 Other
2 Albizzi        36        65 Oligarch
3 Barbadori      55         0 Other
4 Bischeri       44        12 Oligarch
5 Castellani     20        22 Oligarch
6 Ginori         32         0 Other
# i 10 more rows
#
# A tibble: 20 x 2
   from    to
  <int> <int>
1     1     9
2     2     6
3     2     7
# i 17 more rows
```

```
# Confirm all attributes are present
tg_flo |> activate(nodes) |> as_tibble()
```

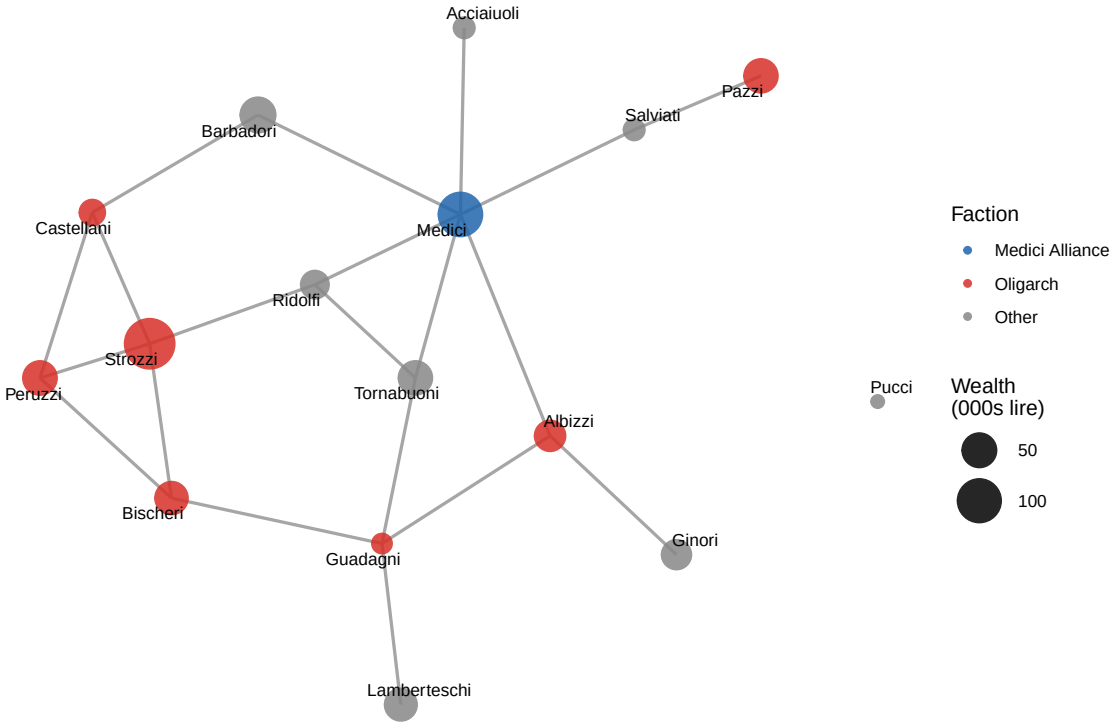
```
# A tibble: 16 x 4
   name         wealth priorates faction
   <chr>         <dbl>     <dbl> <chr>
 1 Acciaiuoli       10        53 Other
 2 Albizzi          36        65 Oligarch
 3 Barbadori        55         0 Other
 4 Bischeri         44        12 Oligarch
 5 Castellani       20        22 Oligarch
 6 Ginori           32         0 Other
 7 Guadagni          8        21 Oligarch
 8 Lamberteschi     42         0 Other
 9 Medici          103        53 Medici Alliance
10 Pazzi            48         0 Oligarch
11 Peruzzi          49        42 Oligarch
12 Pucci             3         0 Other
13 Ridolfi          27        38 Other
14 Salviati         10        35 Other
15 Strozzi         146        74 Oligarch
16 Tornabuoni       48         0 Other
```

5.3.6 Visualisation

```
ggraph(tg_flo, layout = "fr") +
  geom_edge_link(colour = "grey65", width = 0.8) +
  geom_node_point(aes(size = wealth, colour = faction), alpha =
   ↪  0.85) +
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +
  scale_size_continuous(range = c(3, 12), name = "Wealth\n(000s
   ↪  lire)") +
  scale_colour_manual(
    values = c(
      "Medici Alliance" = "#2166ac",
      "Oligarch"        = "#d73027",
      "Other"           = "#888888"
    ),
    name = "Faction"
  ) +
  theme_graph(base_family = "sans") +
  labs(
    title    = "Florentine Marriage Network",
    subtitle = "Node size = family wealth; colour = political
     ↪  faction (Padgett & Ansell, 1993)"
  )
```

Florentine Marriage Network

Node size = family wealth; colour = political faction (Padgett & Ansell, 1993)



The Medici's structural position — not merely their wealth — explains their political dominance. Despite the Strozzi being wealthier, the Medici occupied a more central brokerage position in the marriage network, connecting otherwise disconnected families.

5.4 Part 2: Edge Lists — Zachary's Karate Club

5.4.1 The Dataset

Zachary's Karate Club (**zachary1977information?**) is a network of 34 friendship ties among members of a university karate club, collected over two years of participant observation. During the study period, the club underwent a factional split between the instructor (node 1, "Mr. Hi") and the administrator (node 34, "John A."), eventually dividing into two independent clubs. This network has become one of the most widely used benchmarks for community detection algorithms, because the ground-truth group memberships are known from the ethnographic record.

```
# Load the built-in igraph implementation of the Zachary dataset
karate_raw <- make_graph("Zachary")

vcount(karate_raw)   # number of members
```

```
[1] 34
```

```
ecount(karate_raw)   # number of friendship ties
```

```
[1] 78
```

5.4.2 Extracting and Inspecting the Edge List

The edge list is the most common format in which network data arrives from external sources: relational databases, API responses, and CSV files all naturally produce rows of sender–receiver pairs. `igraph`'s `as_data_frame()` extracts the edge list as a tibble, giving us a realistic starting point.

```
# Extract edge list as a data frame
el_karate <- igraph::as_data_frame(karate_raw, what = "edges")
head(el_karate, 10)
```

```
   from to
1     1  2
2     1  3
3     1  4
4     1  5
5     1  6
6     1  7
7     1  8
8     1  9
9     1 11
10    1 12
```

```
glimpse(el_karate)
```

```
Rows: 78
Columns: 2
$ from <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
 ↪  2, 2,~
```

```
$ to   <dbl> 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 18, 20, 22, 32, 3,
 ↪  4, 8, ~
```

Each row represents one undirected friendship tie. Vertex identities are defined implicitly by their appearance in the `from` and `to` columns — there is no separate node table required for the basic conversion.

5.4.3 Converting a data.frame to igraph

We reconstruct the `igraph` object from the extracted edge list, demonstrating the standard import pathway for data arriving as a flat file. Any additional columns in the data frame are automatically treated as edge attributes.

```
# Reconstruct igraph from the edge list data frame
g_karate <- graph_from_data_frame(
  d        = el_karate,
  directed = FALSE
)

g_karate
```

```
IGRAPH 66b466e UN-- 34 78 --
+ attr: name (v/c)
+ edges from 66b466e (vertex names):
 [1] 1 --2  1 --3  1 --4  1 --5  1 --6  1 --7  1 --8  1 --9  1 --11 1
  ↪  --12
[11] 1 --13 1 --14 1 --18 1 --20 1 --22 1 --32 2 --3  2 --4  2 --8  2
 ↪  --14
[21] 2 --18 2 --20 2 --22 2 --31 3 --4  3 --8  3 --28 3 --29 3 --33 3
 ↪  --10
[31] 3 --9  3 --14 4 --8  4 --13 4 --14 5 --7  5 --11 6 --7  6 --11 6
 ↪  --17
[41] 7 --17 9 --31 9 --33 9 --34 10--34 14--34 15--33 15--34 16--33
 ↪  16--34
[51] 19--33 19--34 20--34 21--33 21--34 23--33 23--34 24--26 24--28
 ↪  24--33
[61] 24--34 24--30 25--26 25--28 25--32 26--32 27--30 27--34 28--34
 ↪  29--32
[71] 29--34 30--33 30--34 31--33 31--34 32--33 32--34 33--34
```

5.4.4 Joining Node Metadata

Vertex-level attributes — here, the faction membership from Zachary's original field observations — are typically stored in a separate table and must be joined to the graph after construction.

```
# Faction membership from Zachary (1977): 1 = Mr. Hi, 2 = John A.
faction_vec <- c(
  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
)

node_meta_karate <- tibble(
  name    = as.character(1:34),
  faction = ifelse(faction_vec == 1, "Mr. Hi", "John A."),
  is_key  = name %in% c("1", "34")   # the two principal actors
)

head(node_meta_karate, 10)
```

```
# A tibble: 10 x 3
   name  faction is_key
   <chr> <chr>   <lgl>
 1 1     Mr. Hi  TRUE
 2 2     Mr. Hi  FALSE
 3 3     Mr. Hi  FALSE
 4 4     Mr. Hi  FALSE
 5 5     Mr. Hi  FALSE
 6 6     Mr. Hi  FALSE
 7 7     Mr. Hi  FALSE
 8 8     Mr. Hi  FALSE
 9 9     John A. FALSE
10 10    John A. FALSE
```

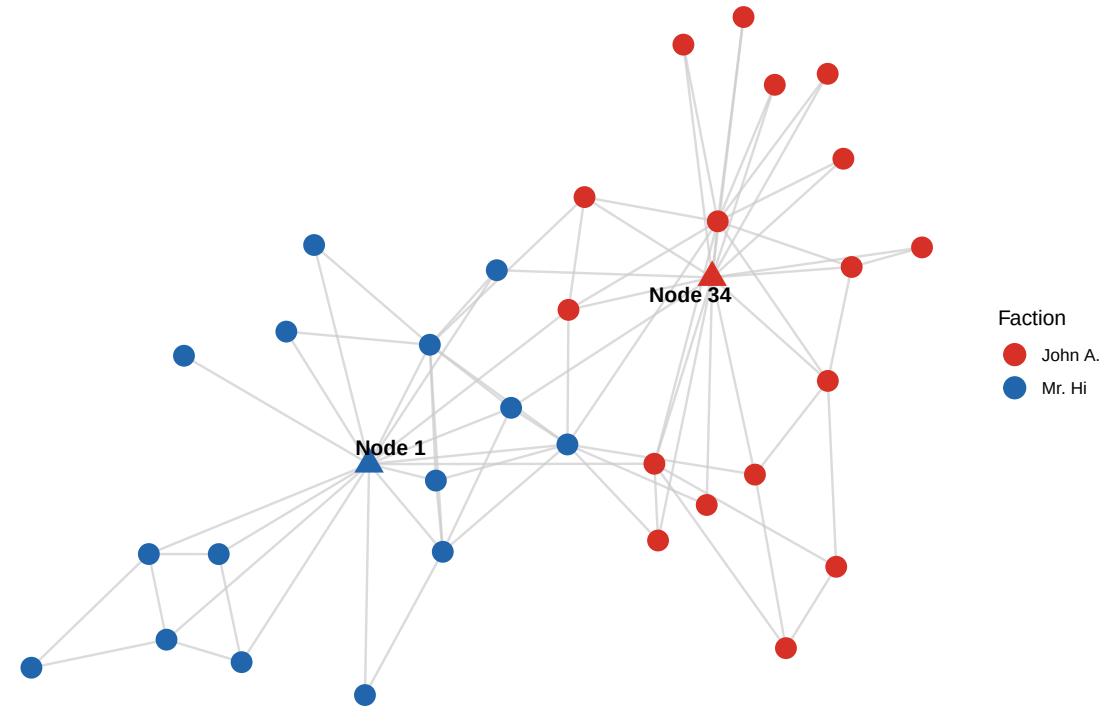
```
# Convert to tbl_graph and join metadata
tg_karate <- as_tbl_graph(g_karate) |>
  activate(nodes) |>
  left_join(node_meta_karate, by = "name")

tg_karate |> activate(nodes) |> as_tibble() |> head()
```

```
# A tibble: 6 x 3
  name  faction is_key
```

```
  <chr> <chr>   <lgl>
1 1     Mr. Hi  TRUE
2 2     Mr. Hi  FALSE
3 3     Mr. Hi  FALSE
4 4     Mr. Hi  FALSE
5 5     Mr. Hi  FALSE
6 6     Mr. Hi  FALSE
```

5.4.5 Visualisation

```
ggraph(tg_karate, layout = "fr") +
  geom_edge_link(colour = "grey80", width = 0.6, alpha = 0.7) +
  geom_node_point(aes(colour = faction, shape = is_key), size = 5)
   ↪  +
  geom_node_text(
    aes(label = ifelse(is_key, paste0("Node ", name), "")),
    repel = TRUE, size = 4, fontface = "bold"
  ) +
  scale_colour_manual(
    values = c("Mr. Hi" = "#2166ac", "John A." = "#d73027"),
    name = "Faction"
  ) +
  scale_shape_manual(
    values = c("TRUE" = 17, "FALSE" = 16),
    guide  = "none"
  ) +
  theme_graph(base_family = "sans") +
  labs(
    title    = "Zachary's Karate Club Network",
    subtitle = "Colour = observed faction split; triangles mark the
     ↪  two principal actors (nodes 1 and 34)"
  )
```



The two factions are visually separable even without the community detection algorithm, illustrating that the structural division in the network preceded the formal organisational split — a classic finding in the sociology of organisations.

5.5 Part 3: Bipartite Networks — Seminar Attendance

5.5.1 The Data Structure

A **bipartite** (or two-mode) network connects two distinct sets of nodes — typically actors and events — where edges run exclusively *between* sets, never *within* them. In social science research, this structure arises naturally from affiliation data: researchers attending seminars, legislators co-sponsoring bills, board members sharing directorships, or consumers purchasing products.

The typical representation is the **incidence matrix** (also called a membership matrix): rows are actors, columns are events, and cell $(i, j) = 1$ if actor i participated in event j.

5.5.2 Constructing the Seminar Attendance Matrix

We represent a cohort of seven researchers and their attendance at five institute events.

```
# Define node sets
researchers <- c(
  "Alice Smith", "Bob Jones",   "Carol McGregor",
  "David Park",  "Emma Wilson", "Frank Dow",  "Grace Kelly"
)

events <- c(
  "SNA Workshop", "AI Ethics Seminar", "Public Policy Forum",
  "DPhil Symposium", "Methods Masterclass"
)

# Incidence matrix: rows = researchers, columns = events
# Cell (i, j) = 1 if researcher i attended event j
attendance_mat <- matrix(
  c(1, 0, 1, 0, 1,
    0, 1, 1, 1, 0,
    1, 1, 0, 0, 1,
    0, 0, 1, 1, 1,
    1, 1, 0, 1, 0,
    0, 1, 1, 0, 1,
    1, 0, 0, 1, 1),
  nrow     = 7,
  byrow    = TRUE,
  dimnames = list(researchers, events)
)

attendance_mat
```

```
              SNA Workshop AI Ethics Seminar Public Policy Forum
Alice Smith              1                 0                    1
Bob Jones                0                 1                    1
Carol McGregor           1                 1                    0
David Park               0                 0                    1
Emma Wilson              1                 1                    0
Frank Dow                0                 1                    1
Grace Kelly              1                 0                    0
              DPhil Symposium Methods Masterclass
Alice Smith                 0                  1
Bob Jones                   1                  0
Carol McGregor              0                  1
```

```
David Park                  1                   1
Emma Wilson                 1                   0
Frank Dow                   0                   1
Grace Kelly                 1                   1
```

5.5.3 Converting to a Bipartite igraph

`igraph::graph_from_incidence_matrix()` converts an incidence matrix directly to a bipartite graph object. The function assigns a `type` vertex attribute — `FALSE` for row nodes (researchers) and `TRUE` for column nodes (events) — which is used by downstream layout and projection functions.

```
g_bip <- graph_from_incidence_matrix(attendance_mat, directed =
 ↪  FALSE)

# Confirm bipartite structure
is_bipartite(g_bip)
```

```
[1] TRUE
```

```
# Inspect node types
tibble(
  name = V(g_bip)$name,
  type = ifelse(V(g_bip)$type, "Event", "Researcher")
)
```

```
# A tibble: 12 x 2
   name                type
   <chr>               <chr>
 1 Alice Smith         Researcher
 2 Bob Jones           Researcher
 3 Carol McGregor      Researcher
 4 David Park          Researcher
 5 Emma Wilson         Researcher
 6 Frank Dow           Researcher
 7 Grace Kelly         Researcher
 8 SNA Workshop        Event
 9 AI Ethics Seminar   Event
10 Public Policy Forum Event
11 DPhil Symposium     Event
12 Methods Masterclass Event
```

```
vcount(g_bip)   # 7 researchers + 5 events = 12 nodes
```

```
[1] 12
```

```
ecount(g_bip)   # one edge per attendance record
```

```
[1] 21
```

5.5.4 Projecting to One-Mode Networks

Bipartite networks are commonly projected onto a single mode for analysis. The **researcher projection** links two researchers if they attended at least one common event; the **event projection** links two events if they share at least one attendee. `bipartite_projection()` computes both simultaneously and records the number of shared co-attendees as an edge `weight` attribute.

```
# Compute both projections
projections <- bipartite_projection(g_bip)

g_researchers <- projections$proj1   # researcher co-attendance
 ↪  network
g_events      <- projections$proj2   # event co-attendance network

# Edge weights record the number of shared events
E(g_researchers)$weight
```

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

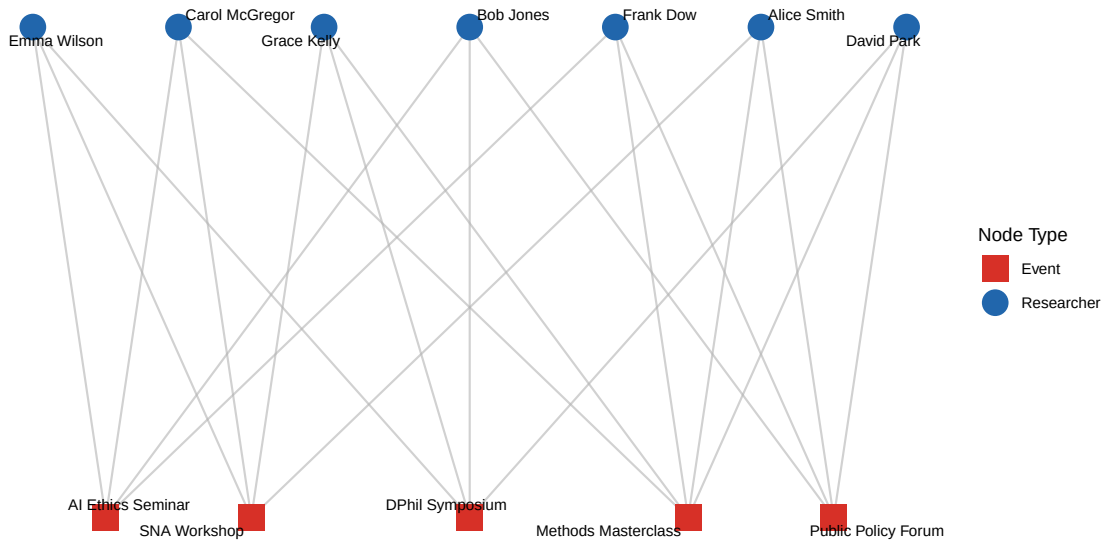
> **Information Loss in Projection**
>
> Projecting a bipartite network onto one mode inevitably discards structural information. Two researchers who co-attended five events and two who co-attended one event are both represented by a single edge — distinguished only by the weight attribute. For many analyses, working directly with the bipartite structure (or using the dual-projection approach of Everett and Borgatti (2013)) preserves more of the original relational signal.

5.5.5 Visualisation

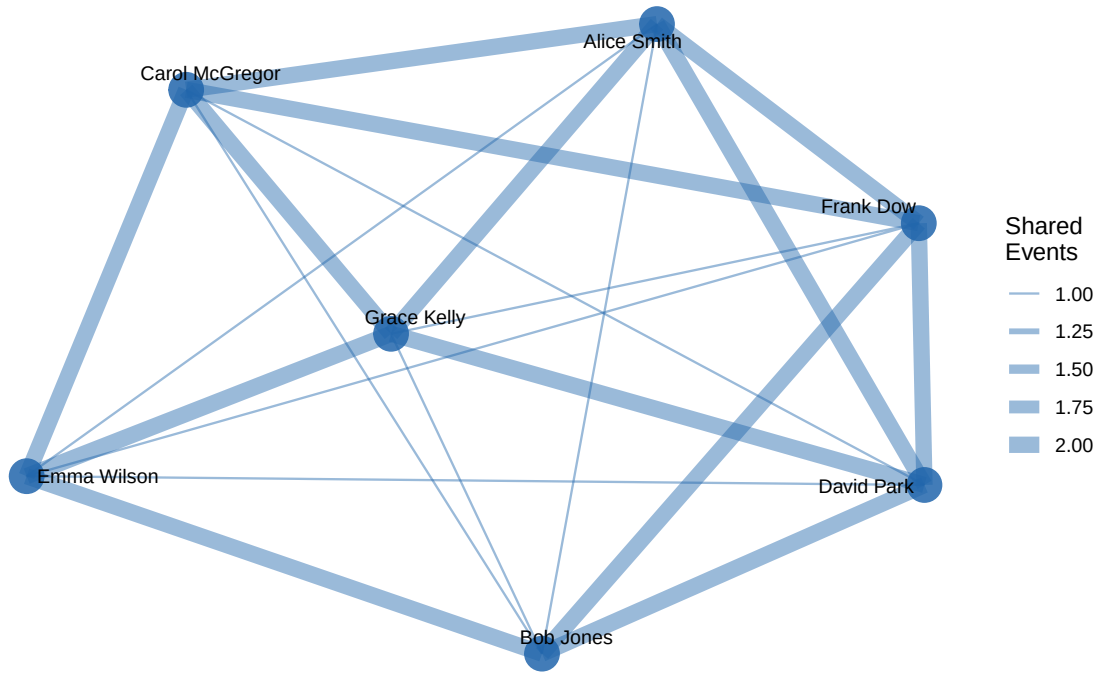
```
# Bipartite two-column layout
ggraph(g_bip, layout = "bipartite") +
  geom_edge_link(colour = "grey70", width = 0.6, alpha = 0.6) +
  geom_node_point(
    aes(
      colour = ifelse(type, "Event", "Researcher"),
      shape  = ifelse(type, "Event", "Researcher")
    ),
    size = 7
  ) +
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +
  scale_colour_manual(
    values = c("Researcher" = "#2166ac", "Event" = "#d73027"),
    name   = "Node Type"
  ) +
  scale_shape_manual(
    values = c("Researcher" = 16, "Event" = 15),
    name   = "Node Type"
  ) +
  theme_graph(base_family = "sans") +
  labs(
    title    = "Seminar Attendance Network (Bipartite)",
    subtitle = "Circles = researchers; squares = events; edges =
    ↪  attendance"
  )
```

Seminar Attendance Network (Bipartite)

Circles = researchers; squares = events; edges = attendance




```
# Researcher co-attendance projection
ggraph(g_researchers, layout = "fr") +
  geom_edge_link(aes(width = weight), colour = "#2166ac", alpha =
  ↪  0.45) +
  geom_node_point(colour = "#2166ac", size = 7, alpha = 0.85) +
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +
  scale_edge_width(range = c(0.5, 3.5), name = "Shared\nEvents") +
  theme_graph(base_family = "sans") +
  labs(
    title    = "Researcher Co-attendance Network (Projected)",
    subtitle = "Edge weight = number of events attended together"
  )
```



5.6 Part 4: Importing from External Files

5.6.1 Reading a CSV Edge List

The most common import task in applied research is loading a flat CSV file containing an edge list. The workflow is consistent regardless of whether the file is stored locally or retrieved from a URL: read it into a `tibble`, inspect and clean it, then pass it to `graph_from_data_frame()`.

For this example of reading csv files we will use a Game of Thrones network dataset. It shows data for character relationships within George R. R. Martin's A Storm of Swords, the third novel in his series A Song of Ice and Fire (also known as the HBO television adaptation Game of Thrones). This data was originally compiled by A. Beveridge and J. Shan, "Network of Thrones," Math Horizons Magazine , Vol. 23, No. 4 (2016), pp. 18-22.

The nodes csv contains 107 different characters, and the edges csv contains 353 weighted relationships between those characters, which were calculated based on how many times two characters' names appeared within 15 words of one another in the novel.

```
edge_path <- "data/got-edges.csv"

edges_raw <- read_csv(edge_path, show_col_types = FALSE)
glimpse(edges_raw)
```

```
Rows: 352
Columns: 3
$ Source <chr> "Aemon", "Aemon", "Aerys", "Aerys", "Aerys", "Aerys",
 ↪  "Alliser"~
$ Target <chr> "Grenn", "Samwell", "Jaime", "Robert", "Tyrion", "Tywin",
 ↪  "Manc~
$ Weight <dbl> 5, 31, 18, 6, 5, 8, 5, 5, 11, 23, 9, 6, 5, 43, 7, 11, 6,
 ↪  7, 8, ~
```

```
# To verify the weights are there:
```

A well-formed edge list CSV requires at minimum two columns (source and target). Additional columns are automatically treated as edge attributes by `graph_from_data_frame()`:

```
# Standardise column names and apply basic cleaning
edges_clean <- edges_raw |>
  rename(from = 1, to = 2) |>         # ensure first two cols are
   ↪  from/to
  filter(!is.na(from), !is.na(to)) |>  # remove rows with missing
   ↪  endpoints
  mutate(across(c(from, to), as.character)) |>  # enforce character
   ↪  IDs
  distinct(from, to, .keep_all = TRUE)           # remove duplicate
   ↪  edges

# Build the igraph object
g_csv <- graph_from_data_frame(edges_clean, directed = FALSE)

# You could also add the edge weights manually:
# E(g_csv)$weight <- edges_raw$Weight

# Convert to tbl_graph for tidyverse workflows
```

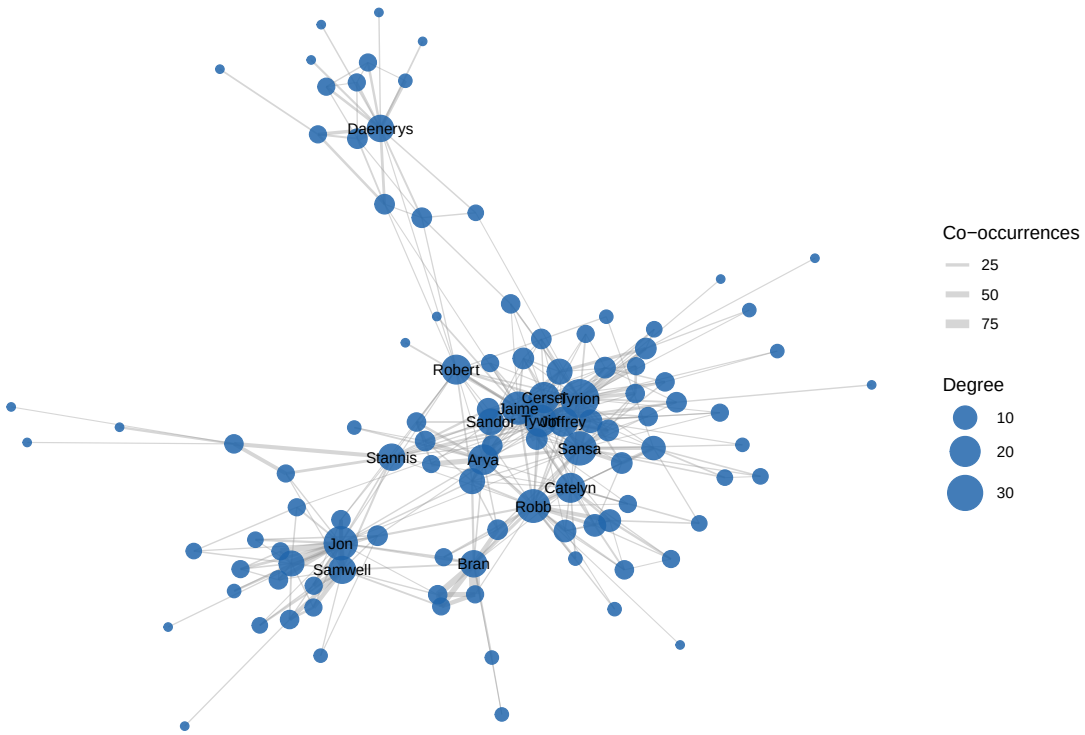
```
tg_csv <- as_tbl_graph(g_csv)

g_csv
```

```
IGRAPH 9db4bf5 UN-- 107 352 --
+ attr: name (v/c), Weight (e/n)
+ edges from 9db4bf5 (vertex names):
 [1] Aemon  --Grenn     Aemon  --Samwell   Aerys  --Jaime     Aerys
  ↪  --Robert
 [5] Aerys  --Tyrion    Aerys  --Tywin     Alliser--Mance     Amory
  ↪  --Oberyn
 [9] Arya   --Anguy     Arya   --Beric     Arya   --Bran      Arya
  ↪  --Brynden
[13] Arya   --Cersei    Arya   --Gendry    Arya   --Gregor    Arya
 ↪  --Jaime
[17] Arya   --Joffrey   Arya   --Jon       Arya   --Rickon    Arya
 ↪  --Robert
[21] Arya   --Roose     Arya   --Sandor    Arya   --Thoros    Arya
 ↪  --Tyrion
[25] Balon  --Loras     Belwas --Barristan Belwas --Illyrio   Beric
 ↪  --Anguy
[29] Beric  --Gendry    Beric  --Thoros    Bran   --Hodor     Bran
 ↪  --Jojen
+ ... omitted several edges
```

5.6.2 Visualisation

```
# Re-attach the weight from the original data (Weight column
 ↪  preserved as edge attr)
E(g_csv)$weight <- edges_raw$Weight

tg_csv <- as_tbl_graph(g_csv) |>
  activate(nodes) |>
  mutate(degree = igraph::degree(g_csv))

ggraph(tg_csv, layout = "fr") +
  geom_edge_link(aes(width = weight), colour = "grey60", alpha =
   ↪  0.4) +
  geom_node_point(aes(size = degree), colour = "#2166ac", alpha =
   ↪  0.85) +
  geom_node_text(
```

```
    aes(label = ifelse(degree > quantile(degree, 0.85), name, "")),
    repel = F, size = 3
  ) +
  scale_edge_width(range = c(0.3, 3), name = "Co-occurrences") +
  scale_size_continuous(range = c(2, 10), name = "Degree") +
  theme_graph(base_family = "sans") +
  labs(
    title    = "Game of Thrones Character Network",
    subtitle = "Edge width = co-occurrence weight; node size =
     ↪  degree; labels show top 15% by degree"
  )
```



5.6.3 Reading a GraphML File

GraphML is an XML-based format that stores both graph topology and arbitrary node and edge attributes in a single self-describing file — making it the standard exchange format for tools such as Gephi, Cytoscape, and NetworkX. Unlike a CSV edge list, a

GraphML file requires no separate attribute-joining step: all metadata travels with the network.

`igraph` reads GraphML directly via `read_graph()` with `format = "graphml"`.

```
g_marvel <- read_graph("data/marvel-network.graphml", format =
 ↪  "graphml")

g_marvel
```

```
IGRAPH 06d5c26 U-W- 327 9891 --
+ attr: label (v/c), id (v/c), Edge Label (e/c), weight (e/n), id (e/c)
+ edges from 06d5c26:
 [1] 1-- 2 1-- 3 1-- 4 1-- 5 1-- 6 1-- 7 1-- 8 1-- 9 1--10 1--11 1--12
  ↪  1--13
[13] 1--14 1--15 1--16 1--17 1--18 1--19 1--20 1--21 1--22 1--23 1--24
 ↪  1--25
[25] 1--26 1--27 1--28 1--29 1--30 1--31 1--32 1--33 1--34 1--35 1--36
 ↪  1--37
[37] 1--38 1--39 1--40 1--41 1--42 1--43 1--44 1--45 1--46 1--47 1--48
 ↪  1--49
[49] 1--50 1--51 1--52 1--53 1--54 1--55 1--56 1--57 1--58 1--59 1--60
 ↪  1--61
[61] 1--62 1--63 1--64 1--65 1--66 1--67 1--68 1--69 1--70 1--71 1--72
 ↪  1--73
[73] 1--74 1--75 1--76 1--77 1--78 1--79 1--80 1--81 1--82 1--83 1--84
 ↪  1--85
[85] 1--86 1--87 1--88 1--89 1--90 1--91 1--92 1--93 1--94 1--95 1--96
 ↪  1--97
+ ... omitted several edges
```

```
# Inspect what vertex and edge attributes the file contains
vertex_attr_names(g_marvel)
```

```
[1] "label" "id"
```

```
edge_attr_names(g_marvel)
```

```
[1] "Edge Label" "weight"     "id"
```

```
# Preview node attributes as a tibble
igraph::as_data_frame(g_marvel, what = "vertices") |> head(10)
```

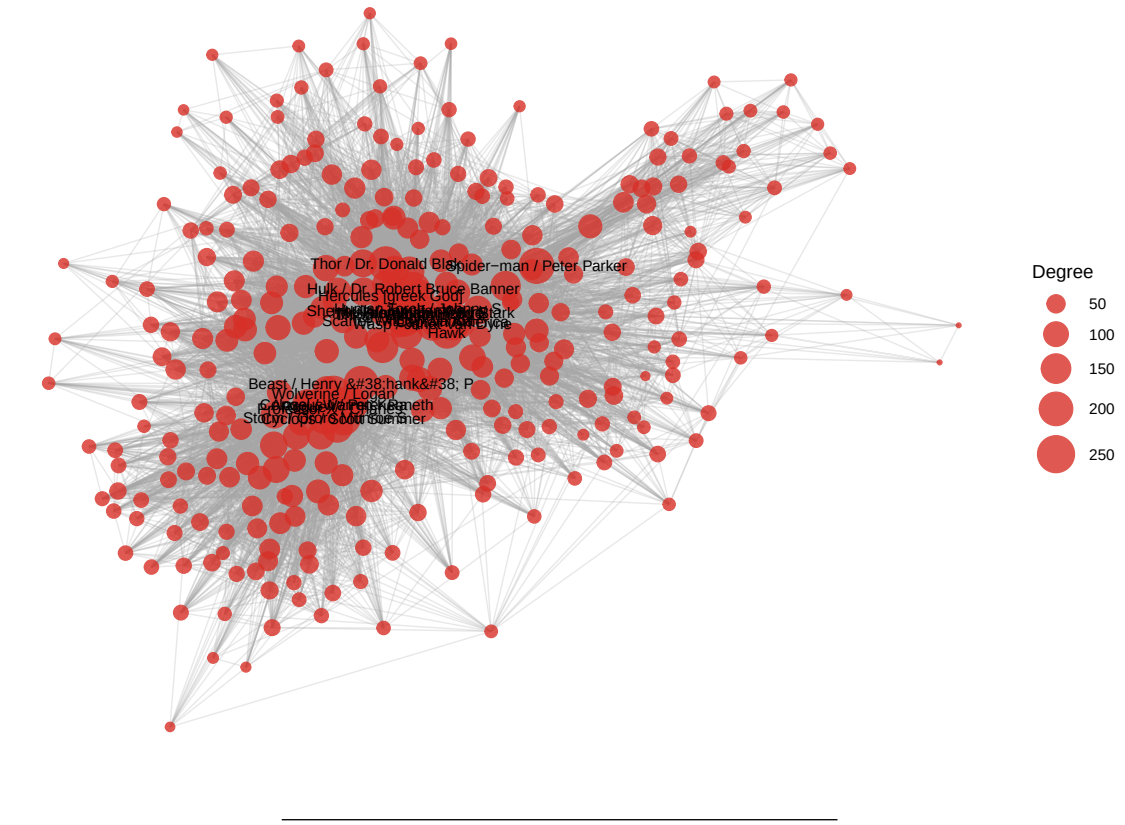
```
   label                      id
1         Black Panther / T'chal
2               Loki [asgardian]
3              Mantis / ? Brandt
4           Iceman / Robert Bobby
5         Marvel Girl / Jean Grey
6          Cyclops / Scott Summer
7             Klaw / Ulysses Klaw
8          Human Torch / Johnny S
9            Richards, Franklin B
10              Wolverine / Logan
```

```
# Convert to tbl_graph - all attributes are carried over
 ↪  automatically
tg_marvel <- as_tbl_graph(g_marvel)
tg_marvel
```

```
# A tbl_graph: 327 nodes and 9891 edges
#
# An undirected simple graph with 1 component
#
# A tibble: 327 x 2
  label id
  <chr> <chr>
1 ""    Black Panther / T'chal
2 ""    Loki [asgardian]
3 ""    Mantis / ? Brandt
4 ""    Iceman / Robert Bobby
5 ""    Marvel Girl / Jean Grey
6 ""    Cyclops / Scott Summer
# i 321 more rows
#
# A tibble: 9,891 x 5
   from    to `Edge Label` weight id
  <int> <int> <chr>         <dbl> <chr>
1     1     2 ""               10 105996
2     1     3 ""               23 105997
3     1     4 ""               12 105998
# i 9,888 more rows
```

5.6.4 Visualisation

```
tg_marvel <- tg_marvel |>
  activate(nodes) |>
  mutate(degree = igraph::degree(g_marvel))

ggraph(tg_marvel, layout = "fr") +
  geom_edge_link(colour = "grey65", alpha = 0.25, width = 0.4) +
  geom_node_point(aes(size = degree), colour = "#d73027", alpha =
  ↪  0.8) +
  geom_node_text(
    aes(label = ifelse(degree > quantile(degree, 0.93), id, "")),
    repel = F, size = 3
  ) +
  scale_size_continuous(range = c(1, 10), name = "Degree") +
  theme_graph(base_family = "sans") +
  labs(
    title    = "Marvel Character Co-appearance Network",
    subtitle = "Node size = degree centrality; labels show top 7%
    ↪  most connected characters"
  )
```



5.7 Part 5: Data Validation

Before proceeding to analysis, it is good practice to validate the integrity of a newly imported network. Three pathologies are worth checking systematically: **isolates** (disconnected nodes), **self-loops** (edges from a node to itself), and **fragmented components** (multiple disconnected subgraphs). Each may reflect a genuine substantive feature of the network or a data import error — and the distinction matters for interpretation.

5.7.1 Checking for Isolates

Isolates are nodes with degree zero: actors present in the node list but with no recorded ties. They may indicate data entry errors (e.g., a respondent appearing in a node list but absent from the edge list), or they may be substantively meaningful (e.g., survey respondents who reported no social contacts).

```
# Using the Karate Club as our working example
isolate_idx <- which(igraph::degree(g_karate) == 0)

if (length(isolate_idx) == 0) {
  cat("No isolates detected.\n")
} else {
  cat("Isolates found at indices:", isolate_idx, "\n")
}
```

```
No isolates detected.
```

5.7.2 Checking for Self-Loops

Self-loops — edges from a node to itself — are rarely substantively meaningful in social networks and often indicate data import errors, such as duplicate identifiers that collapse into self-referential ties. `igraph`'s `which_loop()` returns a logical vector over all edges.

```
n_loops <- sum(which_loop(g_karate))

if (n_loops == 0) {
  cat("No self-loops detected.\n")
} else {
  cat(n_loops, "self-loop(s) found. Removing with simplify()...\n")
  g_karate_clean <- simplify(g_karate, remove.multiple = FALSE,
 ↪  remove.loops = TRUE)
}
```

```
No self-loops detected.
```

5.7.3 Checking Connectedness

A **connected** graph has a single component: every node can reach every other node through some path. Disconnected graphs have multiple components, which has direct implications for analysis — distances between nodes in different components are undefined, and centrality measures computed on the full graph may be misleading.

```
is_connected(g_karate)
```

```
[1] TRUE
```

```
comps <- igraph::components(g_karate)
cat("Number of components:", comps$no, "\n")
```

```
Number of components: 1
```

```
cat("Component sizes     :", comps$csize, "\n")
```

```
Component sizes     : 34
```

5.7.4 A Reusable Validation Function

Wrapping these checks into a single function makes validation a consistent step in any import pipeline.

```
validate_graph <- function(g, name = "graph") {
  cat("=== Network Validation:", name, "===\n\n")

  cat(sprintf("  %-18s %d\n", "Vertices:",   vcount(g)))
  cat(sprintf("  %-18s %d\n", "Edges:",      ecount(g)))
  cat(sprintf("  %-18s %s\n", "Directed:",   is_directed(g)))
  cat(sprintf("  %-18s %s\n", "Weighted:",   is_weighted(g)))
  cat(sprintf("  %-18s %s\n", "Bipartite:",  is_bipartite(g)))
  cat("\n")

  n_isolates <- sum(igraph::degree(g) == 0)
  n_loops    <- sum(which_loop(g))
  n_comps    <- igraph::components(g)$no
  connected  <- is_connected(g)

  cat(sprintf("  %-18s %d%s\n", "Isolates:",
              n_isolates, ifelse(n_isolates > 0, "  [CHECK]", "
↪  [OK]")))
  cat(sprintf("  %-18s %d%s\n", "Self-loops:",
              n_loops,    ifelse(n_loops > 0,    "  [CHECK]", "
↪  [OK]")))
  cat(sprintf("  %-18s %s%s\n", "Connected:",
              connected,  ifelse(!connected,
                                 paste0("  [", n_comps, "
                                  ↪  components]"),
                                 "  [OK]")))
  cat(sprintf("  %-18s %.4f\n", "Density:", edge_density(g)))
```

```
  cat("\n")

  invisible(g)
}
```

```
validate_graph(g_flo,        name = "Florentine Families
 ↪  (Marriage)")
```

```
=== Network Validation: Florentine Families (Marriage) ===

  Vertices:          16
  Edges:             20
  Directed:          FALSE
  Weighted:          FALSE
  Bipartite:         FALSE

  Isolates:          1  [CHECK]
  Self-loops:        0  [OK]
  Connected:         FALSE  [2 components]
  Density:           0.1667
```

```
validate_graph(g_karate,     name = "Zachary's Karate Club")
```

```
=== Network Validation: Zachary's Karate Club ===

  Vertices:          34
  Edges:             78
  Directed:          FALSE
  Weighted:          FALSE
  Bipartite:         FALSE

  Isolates:          0  [OK]
  Self-loops:        0  [OK]
  Connected:         TRUE  [OK]
  Density:           0.1390
```

```
validate_graph(g_bip,        name = "Seminar Attendance
 ↪  (Bipartite)")
```

```
=== Network Validation: Seminar Attendance (Bipartite) ===
```

```
Vertices:           12
Edges:              21
Directed:           FALSE
Weighted:           FALSE
Bipartite:          TRUE

Isolates:           0  [OK]
Self-loops:         0  [OK]
Connected:          TRUE  [OK]
Density:            0.3182
```

5.8 Summary

This chapter has covered the four principal import pathways for network data in R:

Source	Raw Format	igraph Constructor
`ergm` / `network` package	`network` class object	`as.matrix()` → `graph_from_adjacency_matrix()`
Flat file / database	Edge list `data.frame`	`graph_from_data_frame()`
Affiliation records	Incidence matrix	`graph_from_incidence_matrix()`
CSV file or URL	Edge list CSV	`read_csv()` → `graph_from_data_frame()`

Across all four pathways, the workflow follows a consistent logic: (1) load the raw data into an appropriate R object; (2) convert to `igraph` (or other graph object) using the relevant constructor; (3) attach node and edge attributes via `left_join()` within a `tbl_graph`; and (4) validate the resulting network before analysis.

The `tbl_graph` representation from `tidygraph` provides a tidyverse-compatible wrapper around `igraph` objects, enabling `dplyr` pipelines for attribute manipulation and `ggraph` for publication-quality visualisation. Together, these tools constitute a coherent and extensible analytical stack for social network research — one that connects cleanly to both the `statnet` ecosystem (via `intergraph`) and to the broader tidyverse.

6 Analysing Networks in R

From Networks to Insights: centrality, clustering, community detection, and network-level measures

Now that you can build basic networks, it's time to analyse them. This chapter introduces the fundamental descriptive tools: centrality measures that identify important nodes, clustering coefficients and transitivity that capture local structure, community detection algorithms that partition networks, and global indices like density. We'll also touch on random graph generation as a baseline for comparison.

6.1 Setup and Essentials

6.1.1 Loading packages

```
library(igraph)
library(network)
library(tidygraph)
library(ggraph)
library(tidyverse)
library(sna)
library(patchwork)
```

6.2 Importing and Creating Networks

6.2.1 Built-in network data

```
data(flo)               ## Medici family network
flo
```

```
             Acciaiuoli Albizzi Barbadori Bischeri Castellani Ginori
 ↪ Guadagni
Acciaiuoli            0       0         0        0          0      0
 ↪ 0
Albizzi               0       0         0        0          0      1
 ↪ 1
Barbadori             0       0         0        0          1      0
 ↪ 0
Bischeri              0       0         0        0          0      0
 ↪ 1
Castellani            0       0         1        0          0      0
 ↪ 0
Ginori                0       1         0        0          0      0
 ↪ 0
Guadagni              0       1         0        1          0      0
 ↪ 0
Lamberteschi          0       0         0        0          0      0
 ↪ 1
Medici                1       1         1        0          0      0
 ↪ 0
Pazzi                 0       0         0        0          0      0
 ↪ 0
Peruzzi               0       0         0        1          1      0
 ↪ 0
Pucci                 0       0         0        0          0      0
 ↪ 0
Ridolfi               0       0         0        0          0      0
 ↪ 0
Salviati              0       0         0        0          0      0
 ↪ 0
Strozzi               0       0         0        1          1      0
 ↪ 0
Tornabuoni            0       0         0        0          0      0
 ↪ 1
             Lamberteschi Medici Pazzi Peruzzi Pucci Ridolfi Salviati
 ↪ Strozzi
Acciaiuoli              0      1     0       0     0       0        0
 ↪ 0
Albizzi                 0      1     0       0     0       0        0
 ↪ 0
Barbadori               0      1     0       0     0       0        0
 ↪ 0
Bischeri                0      0     0       1     0       0        0
 ↪ 1
```

```
Castellani              0       0       0       1       0       0       0
 ↪  1
Ginori                  0       0       0       0       0       0       0
 ↪  0
Guadagni                1       0       0       0       0       0       0
 ↪  0
Lamberteschi            0       0       0       0       0       0       0
 ↪  0
Medici                  0       0       0       0       0       1       1
 ↪  0
Pazzi                   0       0       0       0       0       0       1
 ↪  0
Peruzzi                 0       0       0       0       0       0       0
 ↪  1
Pucci                   0       0       0       0       0       0       0
 ↪  0
Ridolfi                 0       1       0       0       0       0       0
 ↪  1
Salviati                0       1       1       0       0       0       0
 ↪  0
Strozzi                 0       0       0       1       0       1       0
 ↪  0
Tornabuoni              0       1       0       0       0       1       0
 ↪  0
            Tornabuoni
Acciaiuoli           0
Albizzi              0
Barbadori            0
Bischeri             0
Castellani           0
Ginori               0
Guadagni             1
Lamberteschi         0
Medici               1
Pazzi                0
Peruzzi              0
Pucci                0
Ridolfi              1
Salviati             0
Strozzi              0
Tornabuoni           0
```

6.3 Creating network objects with igraph

The `igraph` package is now the standard for network analysis in R. It's faster, more flexible, and better maintained than older packages.

```
# From adjacency matrix
g_flo <- graph_from_adjacency_matrix(flo, mode = "undirected",
 ↪  weighted = NULL)

# Quick inspection
g_flo
```

```
IGRAPH 89aef22 UN-- 16 20 --
+ attr: name (v/c)
+ edges from 89aef22 (vertex names):
 [1] Acciaiuoli--Medici       Albizzi   --Ginori       Albizzi
 ↪  --Guadagni
 [4] Albizzi   --Medici       Barbadori --Castellani   Barbadori
 ↪  --Medici
 [7] Bischeri  --Guadagni     Bischeri  --Peruzzi      Bischeri
 ↪  --Strozzi
[10] Castellani--Peruzzi      Castellani--Strozzi      Guadagni
 ↪  --Lamberteschi
[13] Guadagni  --Tornabuoni   Medici    --Ridolfi      Medici
 ↪  --Salviati
[16] Medici    --Tornabuoni   Pazzi     --Salviati     Peruzzi
 ↪  --Strozzi
[19] Ridolfi   --Strozzi      Ridolfi   --Tornabuoni
```

```
# Create from edge list
edges <- tibble(
  from = c(1, 1, 2, 2, 3),
  to   = c(2, 3, 3, 4, 4)
)
edges
```

```
# A tibble: 5 x 2
   from    to
  <dbl> <dbl>
1     1     2
2     1     3
3     2     3
4     2     4
```

```
5     3     4
```

```
g <- graph_from_data_frame(edges, directed = FALSE)
g
```

```
IGRAPH c1e45da UN-- 4 5 --
+ attr: name (v/c)
+ edges from c1e45da (vertex names):
[1] 1--2 1--3 2--3 2--4 3--4
```

6.4 Inspecting igraph objects

Once we have an igraph object (i.e. a network) we can inspect it.

```
# Printing the igraph object name returns descriptive information
 ↪  about the network
class(g_flo)
```

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

```
g_flo
```

```
IGRAPH 89aef22 UN-- 16 20 --
+ attr: name (v/c)
+ edges from 89aef22 (vertex names):
 [1] Acciaiuoli--Medici       Albizzi   --Ginori       Albizzi
  ↪  --Guadagni
 [4] Albizzi   --Medici       Barbadori --Castellani   Barbadori
  ↪  --Medici
 [7] Bischeri  --Guadagni     Bischeri  --Peruzzi      Bischeri
  ↪  --Strozzi
[10] Castellani--Peruzzi      Castellani--Strozzi      Guadagni
 ↪  --Lamberteschi
[13] Guadagni  --Tornabuoni   Medici    --Ridolfi      Medici
 ↪  --Salviati
[16] Medici    --Tornabuoni   Pazzi     --Salviati     Peruzzi
 ↪  --Strozzi
[19] Ridolfi   --Strozzi      Ridolfi   --Tornabuoni
```

To examine vertex attributes:

```
vertex_attr(g_flo)
```

```
$name
 [1] "Acciaiuoli"   "Albizzi"      "Barbadori"    "Bischeri"
  ↪  "Castellani"
 [6] "Ginori"       "Guadagni"     "Lamberteschi" "Medici"       "Pazzi"
[11] "Peruzzi"      "Pucci"        "Ridolfi"      "Salviati"
 ↪  "Strozzi"
[16] "Tornabuoni"
```

```
igraph::degree(g_flo)
```

```
  Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
   ↪  Ginori
           1            3            2            3            3
            ↪  1
    Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
     ↪  Pucci
           4            1            6            1            3
            ↪  0
     Ridolfi     Salviati      Strozzi   Tornabuoni
           3            2            4            3
```

We can see there are `$name` and `$degree` attributes. Access them using igraph syntax: `V(igraph_name)$attribute_name`.

```
# Name attribute
 V(g_flo)$name
```

```
 [1] "Acciaiuoli"   "Albizzi"      "Barbadori"    "Bischeri"
  ↪  "Castellani"
 [6] "Ginori"       "Guadagni"     "Lamberteschi" "Medici"       "Pazzi"
[11] "Peruzzi"      "Pucci"        "Ridolfi"      "Salviati"
 ↪  "Strozzi"
[16] "Tornabuoni"
```

```
# Degree attribute

V(g_flo)$degree <-  igraph::degree(g_flo)

V(g_flo)$degree
```

```
 [1] 1 3 2 3 3 1 4 1 6 1 3 0 3 2 4 3
```

```
g_flo
```

```
IGRAPH 89aef22 UN-- 16 20 --
+ attr: name (v/c), degree (v/n)
+ edges from 89aef22 (vertex names):
 [1] Acciaiuoli--Medici       Albizzi   --Ginori       Albizzi
  ↪  --Guadagni
 [4] Albizzi   --Medici       Barbadori --Castellani   Barbadori
  ↪  --Medici
 [7] Bischeri  --Guadagni     Bischeri  --Peruzzi      Bischeri
  ↪  --Strozzi
[10] Castellani--Peruzzi      Castellani--Strozzi      Guadagni
 ↪  --Lamberteschi
[13] Guadagni  --Tornabuoni   Medici    --Ridolfi      Medici
 ↪  --Salviati
[16] Medici    --Tornabuoni   Pazzi     --Salviati     Peruzzi
 ↪  --Strozzi
[19] Ridolfi   --Strozzi      Ridolfi   --Tornabuoni
```

We can add new attributes to vertices in the same way:

```
V(g_flo)
```

```
+ 16/16 vertices, named, from 89aef22:
 [1] Acciaiuoli   Albizzi      Barbadori    Bischeri     Castellani
 [6] Ginori       Guadagni     Lamberteschi Medici       Pazzi
[11] Peruzzi      Pucci        Ridolfi      Salviati     Strozzi
[16] Tornabuoni
```

```
V(g_flo)$fav_pizza <- rep(c("Margherita", "Pepperoni",
 ↪  "Vegetarian", "Hawaiian", "Quattro Formaggi"), times = 4)[1:16]

g_flo
```

```
IGRAPH 89aef22 UN-- 16 20 --
+ attr: name (v/c), degree (v/n), fav_pizza (v/c)
+ edges from 89aef22 (vertex names):
 [1] Acciaiuoli--Medici       Albizzi   --Ginori       Albizzi
  ↪  --Guadagni
 [4] Albizzi   --Medici       Barbadori --Castellani   Barbadori
  ↪  --Medici
```

```
 [7] Bischeri  --Guadagni     Bischeri  --Peruzzi      Bischeri
  ↪  --Strozzi
[10] Castellani--Peruzzi      Castellani--Strozzi      Guadagni
 ↪  --Lamberteschi
[13] Guadagni  --Tornabuoni   Medici    --Ridolfi      Medici
 ↪  --Salviati
[16] Medici    --Tornabuoni   Pazzi     --Salviati     Peruzzi
 ↪  --Strozzi
[19] Ridolfi   --Strozzi      Ridolfi   --Tornabuoni
```

```
E(g_flo)
```

```
+ 20/20 edges from 89aef22 (vertex names):
 [1] Acciaiuoli--Medici       Albizzi   --Ginori       Albizzi
  ↪  --Guadagni
 [4] Albizzi   --Medici       Barbadori --Castellani   Barbadori
  ↪  --Medici
 [7] Bischeri  --Guadagni     Bischeri  --Peruzzi      Bischeri
  ↪  --Strozzi
[10] Castellani--Peruzzi      Castellani--Strozzi      Guadagni
 ↪  --Lamberteschi
[13] Guadagni  --Tornabuoni   Medici    --Ridolfi      Medici
 ↪  --Salviati
[16] Medici    --Tornabuoni   Pazzi     --Salviati     Peruzzi
 ↪  --Strozzi
[19] Ridolfi   --Strozzi      Ridolfi   --Tornabuoni
```

Edge attributes work identically. Access them with `E(igraph_name)$attribute_name`:

```
# Add a new edge attribute
E(g_flo)$trust_level <- rep(c("high", "low", "medium", "high",
 ↪  "low"), times = 4)[1:20]

# View existing edge attributes
edge_attr(g_flo)
```

```
$trust_level
 [1] "high"   "low"    "medium" "high"   "low"    "high"   "low"
  ↪  "medium"
 [9] "high"   "low"    "high"   "low"    "medium" "high"   "low"
  ↪  "high"
[17] "low"    "medium" "high"   "low"
```

```
# Access a specific edge attribute
E(g_flo)$trust_level
```

```
 [1] "high"   "low"    "medium" "high"   "low"    "high"   "low"
  ↪  "medium"
 [9] "high"   "low"    "high"   "low"    "medium" "high"   "low"
  ↪  "high"
[17] "low"    "medium" "high"   "low"
```

You can also query attributes conditionally. For example, to find vertices with degree greater than 2

```
# 1. Create the attribute
V(g_flo)$degree <- igraph::degree(g_flo)

# 2. Now you can filter using the attribute
V(g_flo)[degree > 2]
```

```
+ 9/16 vertices, named, from 89aef22:
[1] Albizzi    Bischeri   Castellani Guadagni   Medici     Peruzzi
 ↪  Ridolfi
[8] Strozzi    Tornabuoni
```

6.5 Network Description and Visualization

6.5.1 Basic network properties

```
vcount(g_flo)                  # number of vertices
```

```
[1] 16
```

```
ecount(g_flo)                  # number of edges
```

```
[1] 20
```

```
is_directed(g_flo)
```

```
[1] FALSE
```

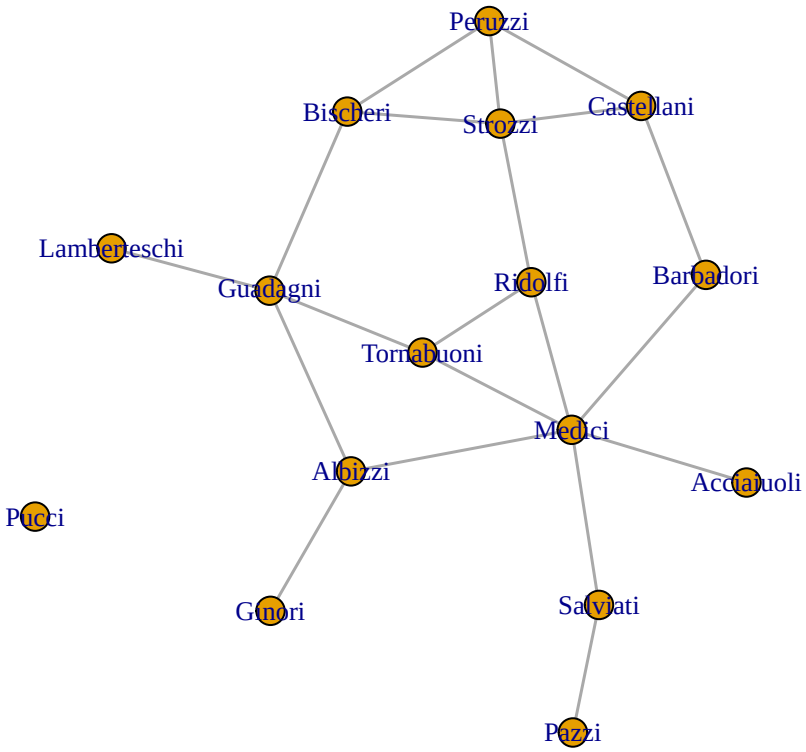
```
is_weighted(g_flo)
```

```
[1] FALSE
```

6.6 Static visualisation

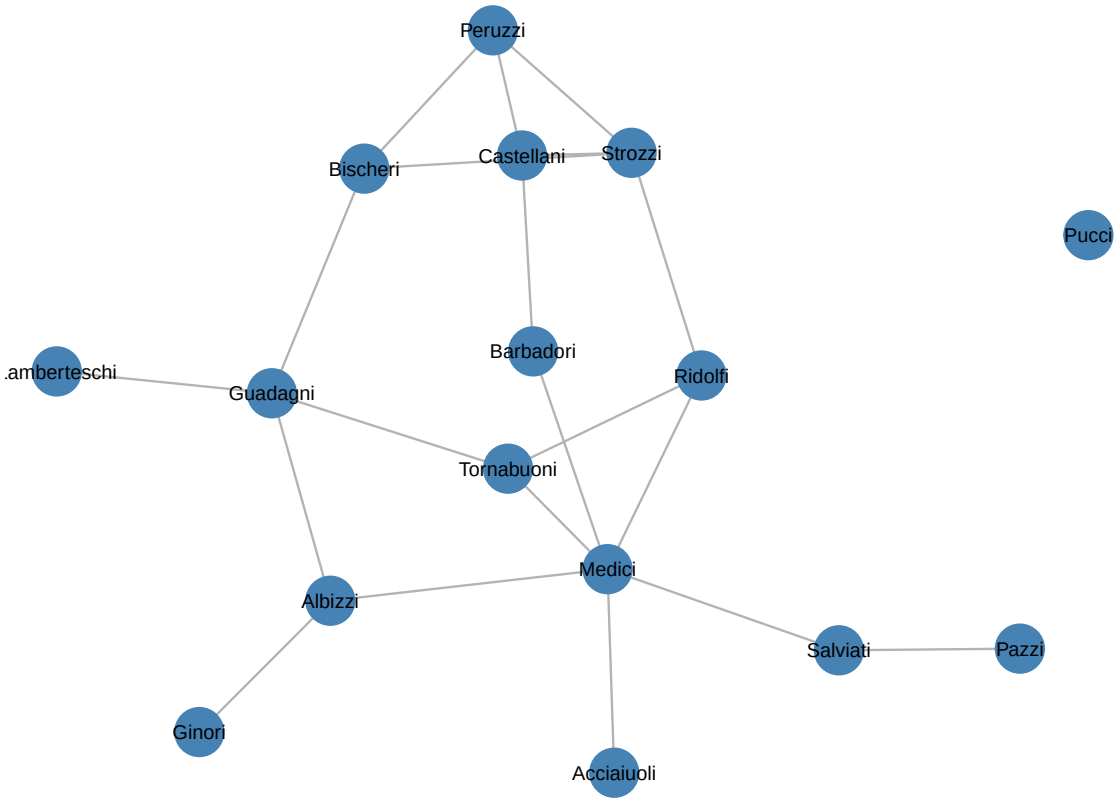
6.6.1 Base igraph plotting

```
plot(g_flo,
     vertex.size = 8,
     vertex.label.cex = 0.8,
     edge.width = 1.5)
```



6.6.2 ggraph for publication-quality plots

```
ggraph(g_flo, layout = 'fr') +
  geom_edge_link(edge_width = 0.5, edge_colour = 'grey70') +
  geom_node_point(size = 10, colour = 'steelblue') +
  geom_node_text(aes(label = name), repel = F, size = 3) +
  theme_graph(base_family = "sans")
```



6.7 Working with Edges and Vertices

6.7.1 Adding and removing edges

```
g <- make_empty_graph(n = 5)

# Add edges by indexing
g <- g + edge(c(1, 2, 2, 3))
g
```

```
IGRAPH f643126 D--- 5 2 --
+ edges from f643126:
[1] 1->2 2->3
```

6.7.2 Vertex and edge attributes

```
# Add vertex attributes
V(g_flo)$degree <- igraph::degree(g_flo)
V(g_flo)
```

```
+ 16/16 vertices, named, from 89aef22:
 [1] Acciaiuoli   Albizzi      Barbadori    Bischeri     Castellani
 [6] Ginori       Guadagni     Lamberteschi Medici       Pazzi
[11] Peruzzi      Pucci        Ridolfi      Salviati     Strozzi
[16] Tornabuoni
```

```
# Add edge attributes (on weighted example)
g_weighted <- g_flo
E(g_weighted)$weight <- runif(ecount(g_flo))
```

6.8 Centrality Measures

6.8.1 Degree centrality

Indicators of centrality assign numbers or rankings to nodes within a graph corresponding to their network position. Degree centrality is defined as the number of links incident upon a node (i.e., the number of ties that a node has).

```
deg <- igraph::degree(g_flo)
deg
```

```
Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
 ↪  Ginori
         1            3            2            3            3
          ↪  1
  Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
   ↪  Pucci
         4            1            6            1            3
          ↪  0
   Ridolfi     Salviati      Strozzi   Tornabuoni
         3            2            4            3
```

6.8.2 Betweenness centrality

Betweenness centrality is a measure based on shortest paths in a graph. It measures how frequently a node appears on the shortest path between other nodes in the graph.

```
bet <- igraph::betweenness(g_flo)
bet
```

```
Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
 ↪  Ginori
  0.000000    19.333333     8.500000     9.500000     5.000000
   ↪  0.000000
  Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
   ↪  Pucci
 23.166667     0.000000    47.500000     0.000000     2.000000
  ↪  0.000000
   Ridolfi     Salviati      Strozzi   Tornabuoni
 10.333333    13.000000     9.333333     8.333333
```

6.8.3 Closeness centrality

Closeness centrality of a node is the average length of the shortest path between the node and all other nodes in the graph. Thus the more central a node is, the closer it is to all other nodes.

```
clo <- igraph::closeness(g_flo)
clo
```

```
  Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
   ↪  Ginori
0.02631579   0.03448276   0.03125000   0.02857143   0.02777778
   ↪  0.02380952
    Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
     ↪  Pucci
0.03333333   0.02325581   0.04000000   0.02040816   0.02631579
   ↪  NaN
     Ridolfi     Salviati      Strozzi   Tornabuoni
0.03571429   0.02777778   0.03125000   0.03448276
```

6.8.4 Eigenvector centrality

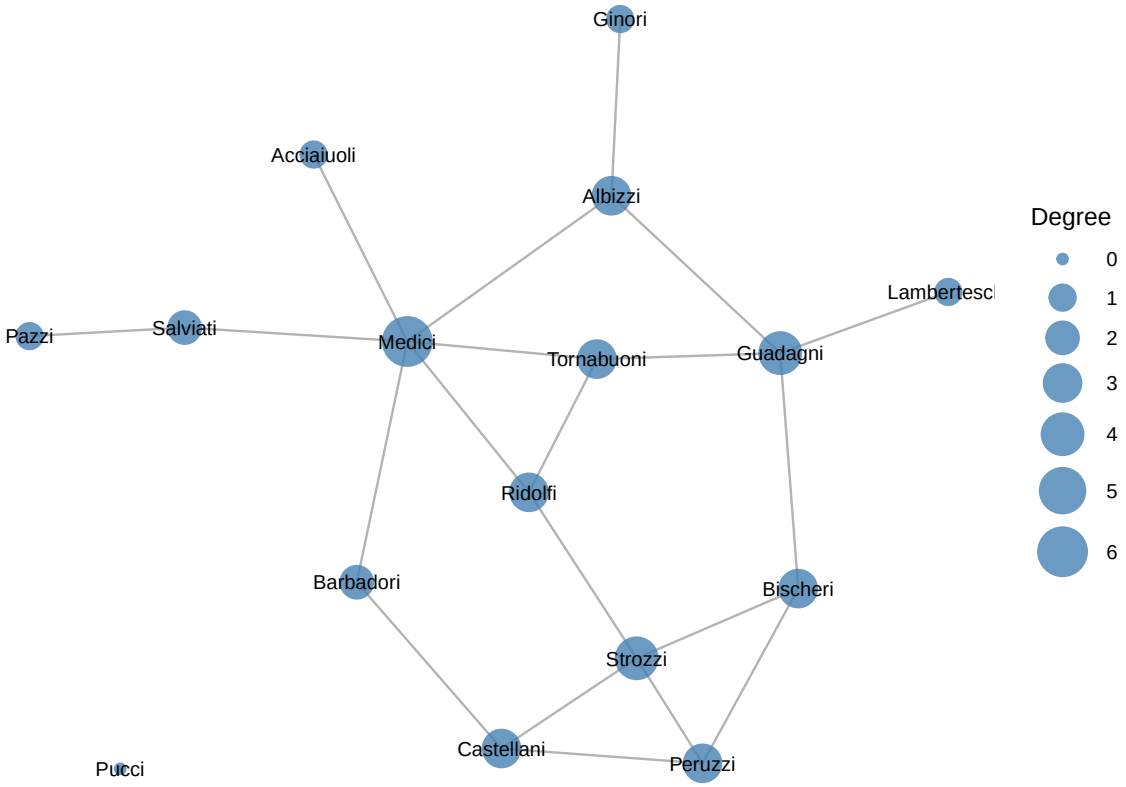
Eigenvector centrality is a measure of the influence of a node in a network. It assigns relative scores to all nodes in the network based on the concept that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes.

```
eig <- igraph::eigen_centrality(g_flo)$vector
eig
```

```
  Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
   ↪  Ginori
3.071155e-01 5.669336e-01 4.919853e-01 6.572037e-01 6.019551e-01
   ↪  1.741141e-01
    Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
     ↪  Pucci
6.718805e-01 2.063449e-01 1.000000e+00 1.041427e-01 6.407743e-01
   ↪  8.652292e-18
     Ridolfi     Salviati      Strozzi   Tornabuoni
7.937398e-01 3.390994e-01 8.272688e-01 7.572302e-01
```

6.8.5 Visualising centrality

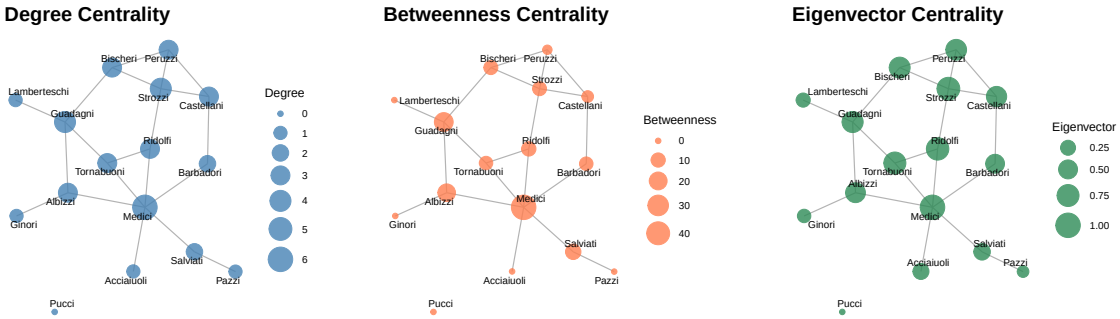
```
ggraph(g_flo, layout = 'fr') +
  geom_edge_link(edge_width = 0.5, edge_colour = 'grey70') +
  geom_node_point(aes(size = deg), colour = 'steelblue', alpha =
  ↪  0.8) +
  geom_node_text(aes(label = name), repel = F, size = 3) +
  scale_size(range = c(2, 10)) +
  theme_graph(base_family = "sans") +
  labs(size = "Degree")
```



6.8.6 Visualising centrality

```
# Calculate centrality measures
V(g_flo)$degree <- igraph::degree(g_flo)
V(g_flo)$betweenness <- igraph::betweenness(g_flo)
V(g_flo)$eigenvector <- igraph::eigen_centrality(g_flo)$vector
# Create a common layout for consistency across plots
```

```
layout_fr <- layout_with_fr(g_flo)

p1 <- ggraph(g_flo, layout = layout_fr) +
  geom_edge_link(edge_width = 0.5, edge_colour = 'grey70') +
  geom_node_point(aes(size = degree), colour = 'steelblue', alpha =
   ↪  0.8) +
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +
  scale_size(range = c(2, 10)) +
  theme_graph(base_family = "sans") +
  labs(title = "Degree Centrality", size = "Degree")

p2 <- ggraph(g_flo, layout = layout_fr) +
  geom_edge_link(edge_width = 0.5, edge_colour = 'grey70') +
  geom_node_point(aes(size = betweenness), colour = 'coral', alpha
   ↪  = 0.8) +
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +
  scale_size(range = c(2, 10)) +
  theme_graph(base_family = "sans") +
  labs(title = "Betweenness Centrality", size = "Betweenness")

p3 <- ggraph(g_flo, layout = layout_fr) +
  geom_edge_link(edge_width = 0.5, edge_colour = 'grey70') +
  geom_node_point(aes(size = eigenvector), colour = 'seagreen',
   ↪  alpha = 0.8) +
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +
  scale_size(range = c(2, 10)) +
  theme_graph(base_family = "sans") +
  labs(title = "Eigenvector Centrality", size = "Eigenvector")

(p1 | p2 | p3)
```



6.9 Network-Level Indices

6.9.1 Density and clustering

Edge density measures what proportion of all possible connections actually exist in the network. A value of 1 means the network is complete (everyone connected to everyone); 0 means no edges at all. For the Florentine families, this tells us how tightly knit the merchant elite were.

Transitivity (the global clustering coefficient) captures the tendency for triangles to form: if A is connected to B and B to C, how likely is it that A connects directly to C? Values range from 0 to 1, where higher values indicate more clustering. This is often interpreted as evidence of in-group cohesion or information flow redundancy in the network.

```
edge_density(g_flo)
```

```
[1] 0.1666667
```

```
transitivity(g_flo)              # global clustering coefficient
```

```
[1] 0.1914894
```

In this example, an edge density of 0.167 means only about one in six possible connections exist—the merchant elite were far from fully interconnected. In social networks, resource constraints (time, trust, capital) limit how many relationships any actor can maintain.

The transitivity of 0.191 is notably similar to the edge density, which suggests the network lacks strong clustering. If the families formed tight cliques—say, through repeated intermarriage within regional or trade blocs—we'd expect transitivity to substantially exceed density.

6.9.2 Reciprocity

```
# Create a directed example
g_dir <- graph_from_data_frame(
  tibble(from = c(1, 2, 2, 3), to = c(2, 1, 3, 2)),
  directed = TRUE
)

reciprocity(g_dir)
```

```
[1] 1
```

6.9.3 Components and connectivity

```
igraph::components(g_flo)
```

```
$membership
  Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
   ↪  Ginori
           1            1            1            1            1
            ↪  1
    Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
     ↪  Pucci
           1            1            1            1            1
            ↪  2
     Ridolfi     Salviati      Strozzi   Tornabuoni
           1            1            1            1

$csize
[1] 15  1

$no
[1] 2
```

```
is_connected(g_flo)
```

```
[1] FALSE
```

A **component** is a maximal subset of vertices where every node can reach every other node through some path. `components()` returns the number of components and their sizes. If `is_connected()` returns TRUE, the network has a single component; otherwise, it's fragmented into disconnected subgroups. For the Florentine families, this reveals whether the entire merchant elite formed one integrated social system or split into separate factions with no bridges between them.

```
## Geodesic distances

distances(g_flo)[1:3, 1:3]
```

```
           Acciaiuoli Albizzi Barbadori
Acciaiuoli          0       2         2
```

```
Albizzi             2       0         2
Barbadori           2       2         0
```

```
diameter(g_flo)
```

```
[1] 5
```

Geodesic distance is the shortest path length between two nodes. The distance matrix shows pairwise distances; the **diameter** is the longest shortest path in the network—the maximum "steps" needed to reach one node from any other. A small diameter indicates efficient information or resource flow across the network; a large diameter suggests bottlenecks or structural holes. For these families, diameter tells us whether influence could spread quickly through intermarriage chains or whether some families were isolated from broader networks.

6.10 Subgroups and Cohesion

6.10.1 k-cores

```
kc <- coreness(g_flo)
kc
```

```
Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
 ↪  Ginori
         1            2            2            2            2
          ↪  1
  Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
   ↪  Pucci
         2            1            2            1            2
          ↪  0
   Ridolfi     Salviati      Strozzi   Tornabuoni
         2            1            2            2
```

A **k-core** is a maximal subgraph where every node has degree at least k within that subgraph. `coreness()` assigns each vertex its core number—the highest k for which it belongs to a k-core. Core numbers reveal the "nested" structure of a network: high-coreness nodes form a densely connected inner circle, while low-coreness nodes sit on the periphery. For the Florentine families, this identifies the most influential merchant families (high coreness) who maintained tight reciprocal ties, versus those with fewer connections.

6.10.2 Cliques

```
cliques(g_flo, min = 3)
```

```
[[1]]
+ 3/16 vertices, named, from 89aef22:
[1] Medici     Ridolfi    Tornabuoni

[[2]]
+ 3/16 vertices, named, from 89aef22:
[1] Castellani Peruzzi    Strozzi

[[3]]
+ 3/16 vertices, named, from 89aef22:
[1] Bischeri Peruzzi  Strozzi
```

A **clique** is a subset of vertices where every pair is connected; i.e. a complete subgraph. `cliques(g_flo, min = 3)` finds all groups of three or more families where all members are mutually connected. Cliques represent the tightest cohesive structures in the network and often indicate shared interests or formal alliances. However, cliques can overlap substantially, so they capture local cohesion rather than global community structure.

6.10.3 Communities

```
comm <- cluster_louvain(g_flo) # modularity-based communities
membership(comm)
```

```
  Acciaiuoli      Albizzi    Barbadori     Bischeri   Castellani
   ↪  Ginori
           1            2            1            3            3
            ↪  2
    Guadagni Lamberteschi       Medici        Pazzi      Peruzzi
     ↪  Pucci
           2            2            1            4            3
            ↪  5
     Ridolfi     Salviati      Strozzi   Tornabuoni
           1            4            3            1
```

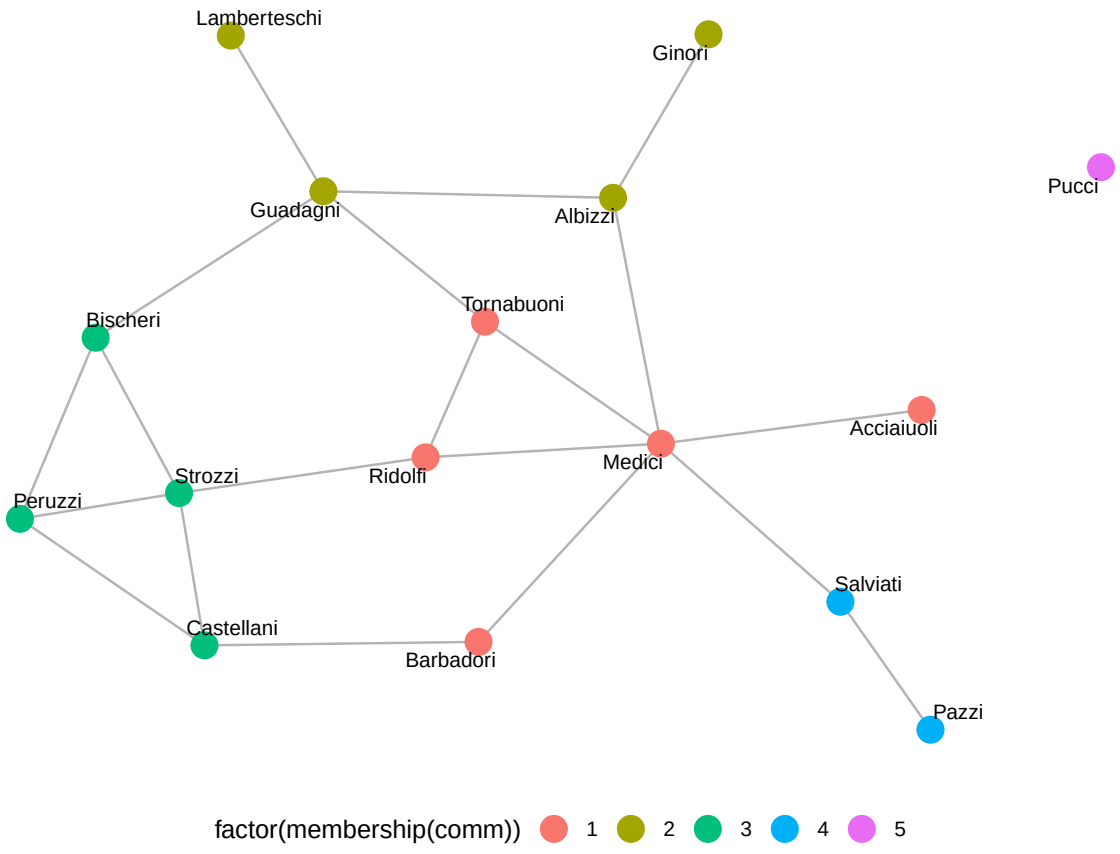
```
modularity(comm)
```

```
[1] 0.3975
```

```
?cluster_louvain()
```

Communities are groups of vertices that are more densely connected internally than to the rest of the network. The Louvain algorithm optimises **modularity**—a measure of how well the partition maximises within-group edges relative to between-group edges. `membership(comm)` assigns each vertex to a community; `modularity(comm)` returns a score from 0 to 1, where higher values indicate stronger community structure. Unlike cliques, communities partition the entire network into non-overlapping groups, offering a global view of the network's factional organisation.

```
ggraph(g_flo, layout = 'fr') +
  geom_edge_link(edge_width = 0.5, edge_colour = 'grey70') +
  geom_node_point(aes(colour = factor(membership(comm))), size = 5)
  ↪ +
  geom_node_text(aes(label = name), repel = TRUE, size = 3) +
  theme_graph(base_family = "sans") +
  theme(legend.position = 'bottom')
```



6.11 Community detection

Community detection (or cluster analysis) seeks to find groups of nodes that are internally densely connected and externally sparsely connected—that is, communities or clusters. The distinction between a network with clear cluster structure and one without becomes apparent visually. Below are two examples.

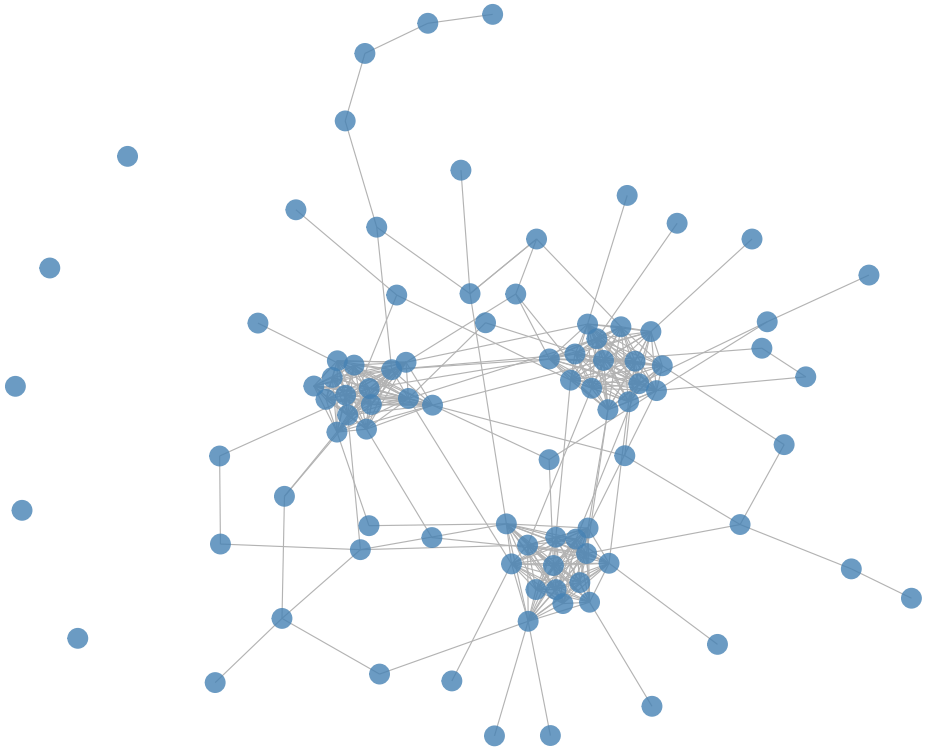
A network with visible and intuitive cluster structure:

```
set.seed(42)

# Create three separate cliques
clique1 <- graph_from_edgelist(
  expand.grid(1:15, 1:15)[expand.grid(1:15, 1:15)[, 1] <
                            expand.grid(1:15, 1:15)[, 2], ] |>
                             ↪  as.matrix()
)

clique2 <- graph_from_edgelist(
  expand.grid(16:30, 16:30)[expand.grid(16:30, 16:30)[, 1] <
                              expand.grid(16:30, 16:30)[, 2], ] |>
                               ↪  as.matrix()
)

clique3 <- graph_from_edgelist(
  expand.grid(31:45, 31:45)[expand.grid(31:45, 31:45)[, 1] <
                              expand.grid(31:45, 31:45)[, 2], ] |>
                               ↪  as.matrix()
)

# Combine cliques with a few sparse between-cluster edges
g_clustered <- clique1 + clique2 + clique3
g_clustered
```

```
IGRAPH d06bed0 D--- 90 315 --
+ edges from d06bed0:
 [1]  1-> 2  1-> 3  2-> 3  1-> 4  2-> 4  3-> 4  1-> 5  2-> 5  3-> 5  4->
  ↪  5
[11]  1-> 6  2-> 6  3-> 6  4-> 6  5-> 6  1-> 7  2-> 7  3-> 7  4-> 7  5->
 ↪  7
[21]  6-> 7  1-> 8  2-> 8  3-> 8  4-> 8  5-> 8  6-> 8  7-> 8  1-> 9  2->
 ↪  9
[31]  3-> 9  4-> 9  5-> 9  6-> 9  7-> 9  8-> 9  1->10  2->10  3->10
 ↪  4->10
```

```
[41]  5->10  6->10  7->10  8->10  9->10  1->11  2->11  3->11  4->11
 ↪  5->11
[51]  6->11  7->11  8->11  9->11 10->11  1->12  2->12  3->12  4->12
 ↪  5->12
[61]  6->12  7->12  8->12  9->12 10->12 11->12  1->13  2->13  3->13
 ↪  4->13
[71]  5->13  6->13  7->13  8->13  9->13 10->13 11->13 12->13  1->14
 ↪  2->14
[81]  3->14  4->14  5->14  6->14  7->14  8->14  9->14 10->14 11->14
 ↪  12->14
+ ... omitted several edges
```

```
set.seed(12)
n_edges_add <- ceiling(ecount(g_clustered) * 0.3)
random_edges <- sample(1:vcount(g_clustered), 2 * n_edges_add,
 ↪  replace = TRUE)
g_clustered <- add_edges(g_clustered, random_edges)

ggraph(g_clustered, layout = 'fr') +
  geom_edge_link(edge_width = 0.2, edge_colour = 'grey70') +
  geom_node_point(colour = 'steelblue', size = 3, alpha = 0.8) +
  theme_graph(base_family = "sans")
```



```
g_clustered
```

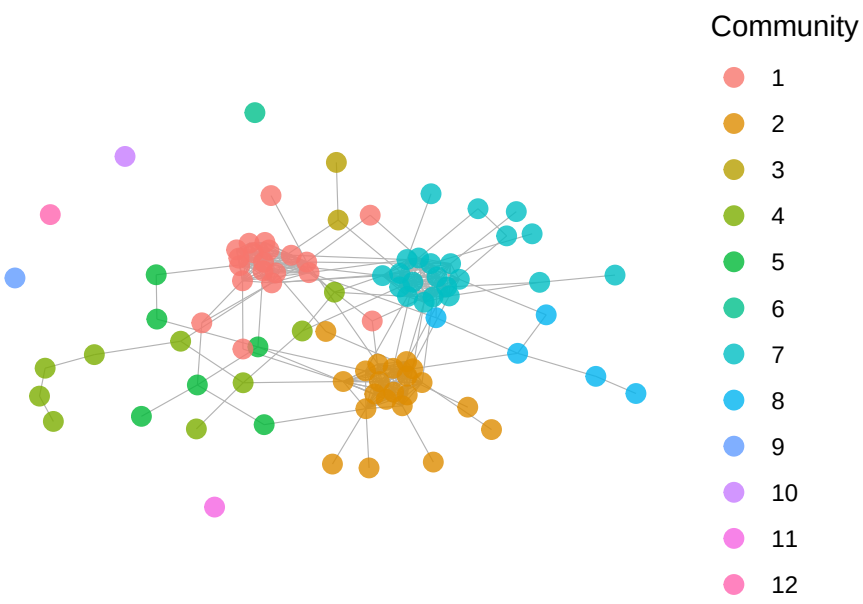
```
IGRAPH c283546 D--- 90 410 --
+ edges from c283546:
 [1]  1-> 2  1-> 3  2-> 3  1-> 4  2-> 4  3-> 4  1-> 5  2-> 5  3-> 5  4->
  ↪  5
[11]  1-> 6  2-> 6  3-> 6  4-> 6  5-> 6  1-> 7  2-> 7  3-> 7  4-> 7  5->
 ↪  7
[21]  6-> 7  1-> 8  2-> 8  3-> 8  4-> 8  5-> 8  6-> 8  7-> 8  1-> 9  2->
 ↪  9
[31]  3-> 9  4-> 9  5-> 9  6-> 9  7-> 9  8-> 9  1->10  2->10  3->10
 ↪  4->10
[41]  5->10  6->10  7->10  8->10  9->10  1->11  2->11  3->11  4->11
 ↪  5->11
[51]  6->11  7->11  8->11  9->11 10->11  1->12  2->12  3->12  4->12
 ↪  5->12
[61]  6->12  7->12  8->12  9->12 10->12 11->12  1->13  2->13  3->13
 ↪  4->13
[71]  5->13  6->13  7->13  8->13  9->13 10->13 11->13 12->13  1->14
 ↪  2->14
[81]  3->14  4->14  5->14  6->14  7->14  8->14  9->14 10->14 11->14
 ↪  12->14
```

```
+ ... omitted several edges
```

```
# Detect communities
g_clustered <- as.undirected(g_clustered)
comm_clustered <- cluster_louvain(g_clustered)
V(g_clustered)$community <- igraph::membership(comm_clustered)

g_clustered
```

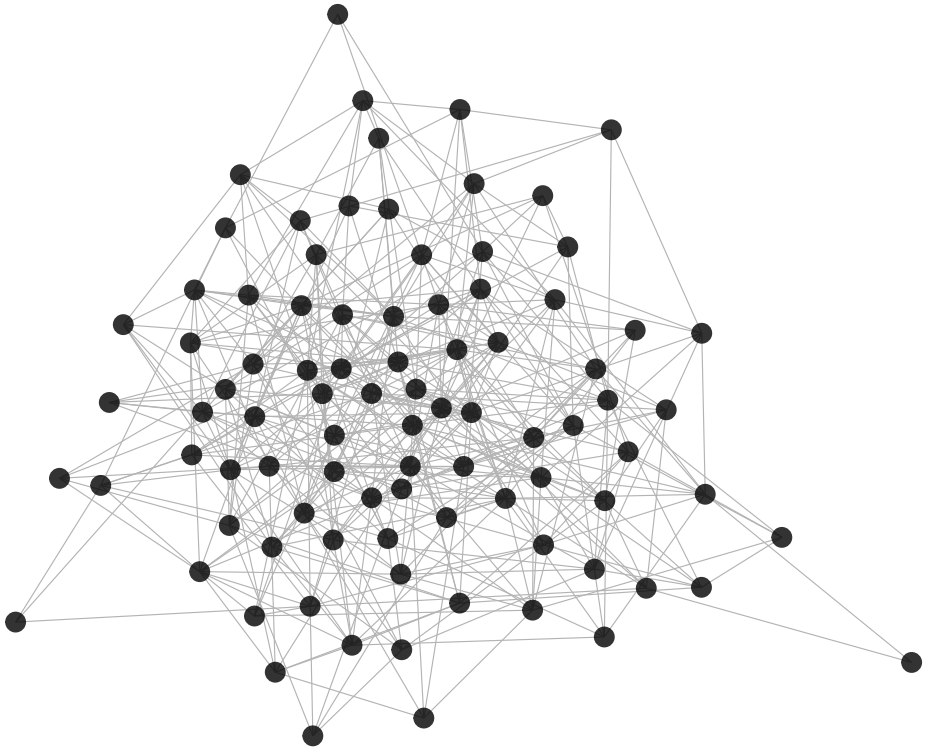
```
IGRAPH 679903f U--- 90 393 --
+ attr: community (v/n)
+ edges from 679903f:
 [1]  1-- 2  1-- 3  2-- 3  1-- 4  2-- 4  3-- 4  1-- 5  2-- 5  3-- 5  4--
  ↪  5
[11]  1-- 6  2-- 6  3-- 6  4-- 6  5-- 6  1-- 7  2-- 7  3-- 7  4-- 7  5--
 ↪  7
[21]  6-- 7  1-- 8  2-- 8  3-- 8  4-- 8  5-- 8  6-- 8  7-- 8  1-- 9  2--
 ↪  9
[31]  3-- 9  4-- 9  5-- 9  6-- 9  7-- 9  8-- 9  1--10  2--10  3--10
 ↪  4--10
[41]  5--10  6--10  7--10  8--10  9--10  1--11  2--11  3--11  4--11
 ↪  5--11
[51]  6--11  7--11  8--11  9--11 10--11  1--12  2--12  3--12  4--12
 ↪  5--12
[61]  6--12  7--12  8--12  9--12 10--12 11--12  1--13  2--13  3--13
 ↪  4--13
[71]  5--13  6--13  7--13  8--13  9--13 10--13 11--13 12--13  1--14
 ↪  2--14
+ ... omitted several edges
```

```
ggraph(g_clustered, layout = 'fr') +
  geom_edge_link(edge_width = 0.2, edge_colour = 'grey70') +
  geom_node_point(aes(colour = factor(community)), size = 3, alpha
   ↪  = 0.8) +
  theme_graph(base_family = "sans") +
  labs(title = "", colour = "Community")
```



A network with weak or absent cluster structure:

```
set.seed(42)

# Create a random Erdős-Rényi network
g_random <- sample_gnp(90, 0.1)

ggraph(g_random, layout = 'fr') +
  geom_edge_link(edge_width = 0.2, edge_colour = 'grey70') +
  geom_node_point(colour = 'black', size = 3, alpha = 0.8) +
  theme_graph(base_family = "sans")
```



The clustering algorithms implemented in `igraph` are:

```
 [1] "cluster_edge_betweenness"  "cluster_fast_greedy"
 [3] "cluster_fluid_communities" "cluster_infomap"
 [5] "cluster_label_prop"        "cluster_leading_eigen"
 [7] "cluster_leiden"            "cluster_louvain"
 [9] "cluster_optimal"           "cluster_spinglass"
[11] "cluster_walktrap"
```

Most of these optimise **modularity**—the fraction of edges within groups minus the expected fraction under random assignment. These range from greedy optimisation (fast but approximate) to spectral methods and information-theoretic approaches, each with different computational costs and sensitivity to network structure. `cluster_louvain()` offers a good balance of speed and quality for most applications, though `cluster_leiden()` addresses known limitations with smaller clusters.

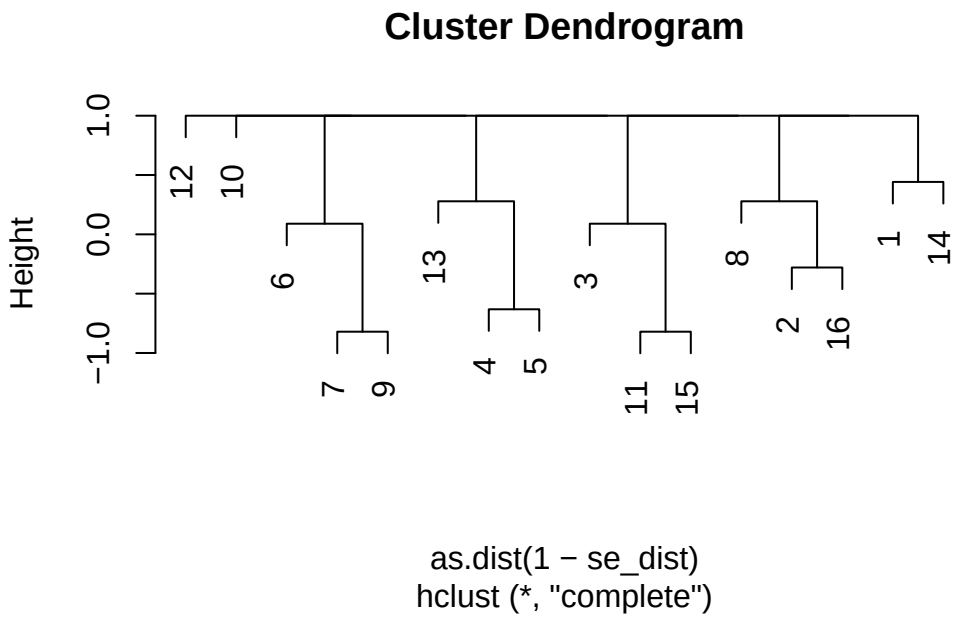
6.12 Structural Equivalence

Two people are structurally equivalent if they have the exact same connections to the same people, even if they don't know each other directly. Put simply, it is trying to

identify "who plays the same role" in a network.

We use `method = "invlogweight"`, which is a Inverse Log-Weight to weight connections. It suggests that sharing a connection with a "popular" person (a hub) is less telling than sharing a connection with a "lonely" person. If you and I both know the same person who has 100 friends, it might be a coincidence; if we both know the same person who only has one friend, we are likely very similar in the social structure.

```
##  Correlation of profiles
se_dist <- similarity(g_flo, method = "invlogweight")

##  Hierarchical clustering
hc <- hclust(as.dist(1 - se_dist))
plot(hc)
```



Nodes that are joined together at the very bottom of the tree are the most "structurally equivalent." As you move up the tree, the groups become broader and less similar.

```
# Create Index-Name data frame
flo_families <- data.frame(
  Index = 1:vcount(g_flo),
  Name = V(g_flo)$name
)

# View the table
```

```
print(flo_families)
```

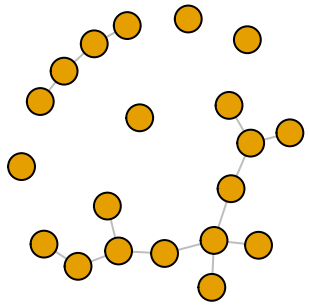
```
   Index         Name
1      1   Acciaiuoli
2      2      Albizzi
3      3    Barbadori
4      4     Bischeri
5      5   Castellani
6      6       Ginori
7      7     Guadagni
8      8 Lamberteschi
9      9       Medici
10    10        Pazzi
11    11      Peruzzi
12    12        Pucci
13    13      Ridolfi
14    14     Salviati
15    15      Strozzi
16    16   Tornabuoni
```

6.13 Random Graph Models

6.13.1 Erdős–Rényi

Each possible edge is included independently with probability p. Produces a network with roughly uniform randomness and low clustering.

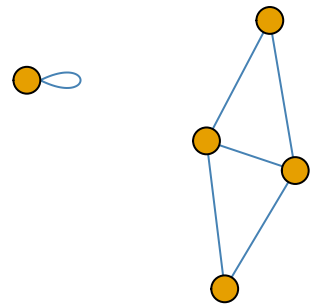
```
g_er <- sample_gnp(n = 20, p = 0.1)

plot(g_er, main = "Erdős-Rényi (Gnp)", vertex.size = 20,
 ↪  vertex.label = NA, edge.color = "gray")
```



6.13.2 Configuration model (fixed degree sequence)

Generates a random graph that preserves a specified degree sequence. Useful to test network properties while controlling for node degrees.

```
g_conf <- igraph::sample_degseq(c(3,3,2,2,2))
plot(g_conf, main = "Configuration Model", vertex.size = 20,
 ↪  vertex.label = NA, edge.color = "steelblue")
```

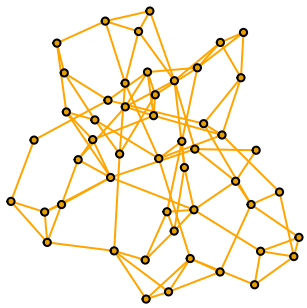


6.13.3 Small-world (Watts–Strogatz)

Starts with a regular lattice and randomly rewires edges with probability p. Produces networks with high clustering and short path lengths.

```
g_ws <- sample_smallworld(dim = 1, size = 50, nei = 2, p = 0.1)
```

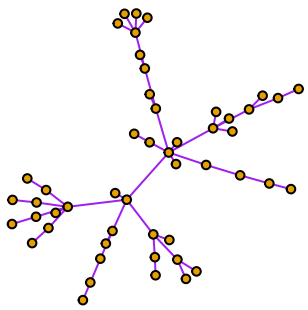
```
plot(g_ws, main = "Small-World (Watts-Strogatz)", vertex.size = 5,
 ↪  vertex.label = NA, edge.color = "orange")
```



6.13.4 Scale-free (preferential attachment)

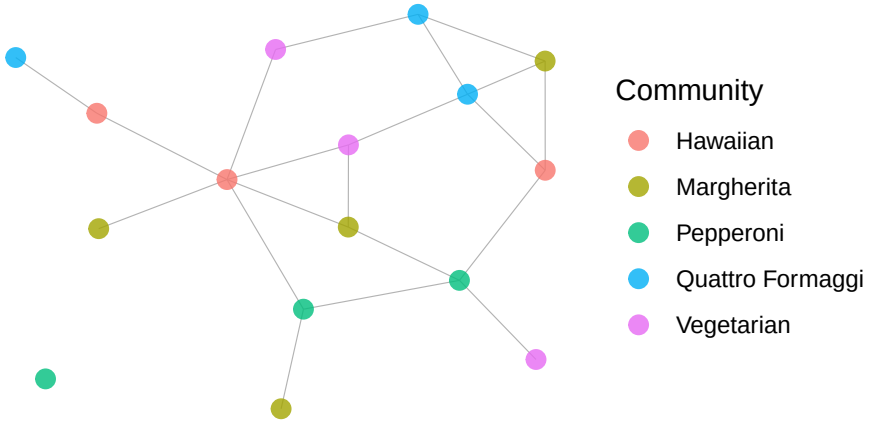
Nodes are added one by one, preferentially attaching to high-degree nodes. Produces networks with hubs and a power-law degree distribution.

```
g_sf <- sample_pa(n = 50, power = 1, directed = FALSE)
plot(g_sf, main = "Scale-Free (Preferential Attachment)",
 ↪  vertex.size = 6, vertex.label = NA, edge.color = "purple")
```




```
ggraph(g_flo, layout = 'fr') +
  geom_edge_link(edge_width = 0.2, edge_colour = 'grey70') +
  geom_node_point(aes(colour = factor(fav_pizza)), size = 3, alpha
   ↪  = 0.8) +
```

```
  theme_graph(base_family = "sans") +
  labs(title = "", colour = "Community")
```




```
# Assortativity test for favourite pizza
assortativity_nominal(g_flo, types =
 ↪  as.numeric(factor(V(g_flo)$fav_pizza)))
```

```
[1] -0.07424961
```

7 Network Visualisation

Principles and practice with igraph and ggraph in R

7.1 1. Introduction

Network visualisation is deceptively simple to produce and genuinely difficult to do well. A network plot is a two-dimensional projection of a high-dimensional relational structure — no single drawing captures everything. Every choice you make (layout algorithm, node size, edge width, colour scheme) emphasises some features of the network while suppressing others. The same network can look sparse or dense, hierarchical or flat, central or diffuse, depending entirely on how it is drawn.

This matters because audiences read meaning directly from visual structure. If your layout algorithm happens to cluster two unrelated nodes near each other, viewers will infer a relationship. If you size nodes by degree but the degree distribution is highly skewed, a handful of hubs will dominate the image and everything else will be invisible. These are not aesthetic concerns — they are epistemic ones.

The practical implication: treat network visualisation as an argumentative act, not a neutral rendering. Choose parameters deliberately, justify your choices, and show multiple views when a single one would mislead.

This tutorial covers the mechanics of network visualisation in R using `igraph` (base-R style plots) and `ggraph` (grammar-of-graphics style). For a more comprehensive tutorial, see resources in the reference list.

7.2 2. Colour in Network Plots

Colour is the most powerful and most abused visual channel in network plots. Used well, it immediately communicates group structure, centrality gradients, or tie strength. Used carelessly, it produces rainbow soup that confuses rather than informs.

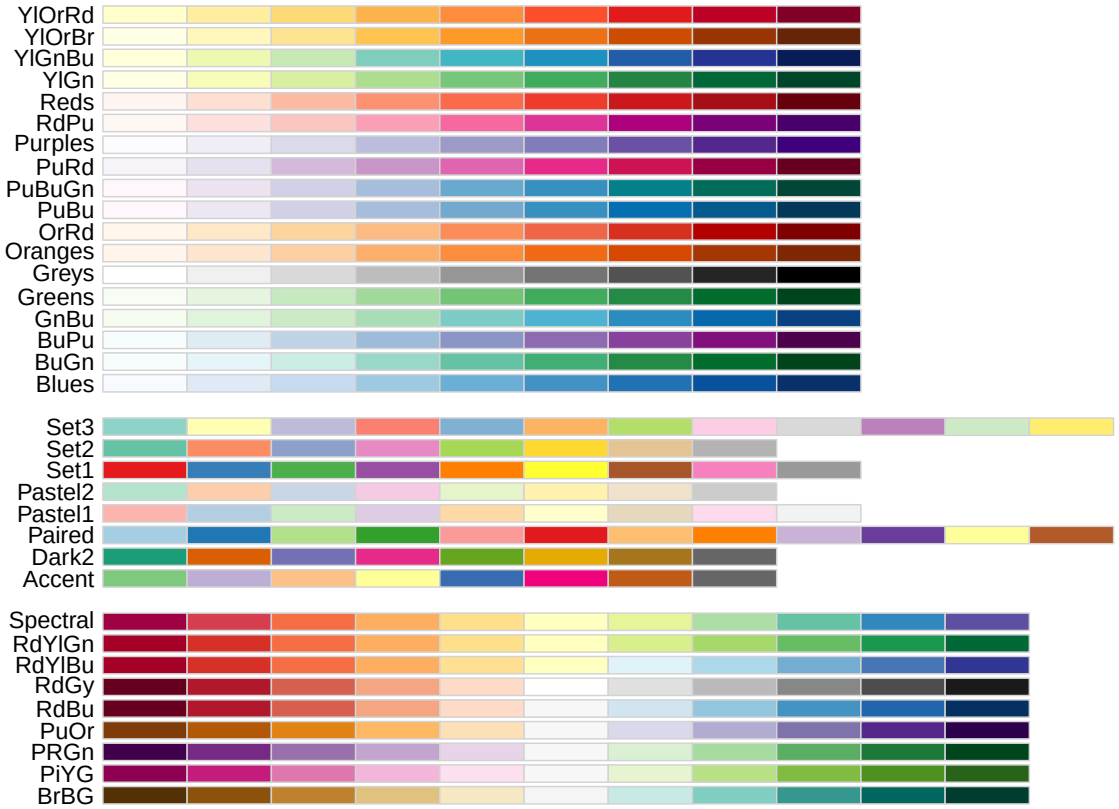
7.2.1 Colour as a data channel

Match your colour scale type to your data type:

Data type	Scale type	R palette
Categorical (communities, groups)	Qualitative	`brewer.pal(n, "Set1")`, `brewer.pal(n, "Dark2")`
Ordered / continuous (degree, weight)	Sequential	`brewer.pal(9, "Blues")`, `viridis::viridis(n)`
Diverging (positive/negative)	Diverging	`brewer.pal(11, "RdBu")`

7.2.2 Available palettes

```
display.brewer.all()
```



For accessibility, prefer palettes that remain distinguishable under colour-blindness. The `"Dark2"` qualitative palette and the viridis sequential palettes (`"viridis"`, `"magma"`, `"plasma"`) are designed to be perceptually uniform and colour-blind safe.

7.2.3 Creating colour vectors

The core workflow: generate a palette vector, then index into it using a node or edge attribute.

```
# Qualitative: 4-community network
pal_qual <- brewer.pal(4, "Dark2")
# e.g. membership vector 1:4 indexes directly into pal_qual

# Sequential: map continuous values to a colour ramp
pal_seq <- colorRampPalette(brewer.pal(9, "Blues"))(100)
# usage: pal_seq[round(rescaled_value * 99) + 1]

# Quick helper: map a numeric vector to a colour palette
val_to_col <- function(x, palette = "YlOrRd", n = 100) {
```

```
  ramp <- colorRampPalette(brewer.pal(9, palette))(n)
  ramp[cut(x, breaks = n, labels = FALSE, include.lowest = TRUE)]
}

# Example: colour 20 nodes by a random score
set.seed(1)
scores <- runif(20)
node_cols <- val_to_col(scores, "YlOrRd")
```

Rule of thumb: limit categorical colours to 8 distinct hues. Beyond that, human perception degrades rapidly and the plot becomes uninterpretable.

7.3 3. Data Format, Size, and Preparation

7.3.1 Loading the data

We use two datasets throughout this tutorial:

1. **Game of Thrones** — a weighted co-occurrence network of characters from the books (107 nodes, 351 edges).
2. **Florentine families** — the classic Padgett marriage network of 16 Renaissance Florentine families.

```
#   Game of Thrones
edges_got <- read.csv("data/got-edges.csv")   # Source, Target,
 ↪  Weight
nodes_got <- read.csv("data/got-nodes.csv")   # Id, Label

g_got <- graph_from_data_frame(
  d        = edges_got,
  directed = FALSE,
  vertices = nodes_got
)

#   Florentine marriage network
data(flo)   # adjacency matrix, from the network package
g_flo <- graph_from_adjacency_matrix(flo, mode = "undirected", diag
 ↪  = FALSE)
```

7.3.2 Inspecting the graph object

```
cat("=== Game of Thrones ===\n")
```

```
=== Game of Thrones ===
```

```
cat("Nodes:", vcount(g_got), " | Edges:", ecount(g_got), "\n")
```

```
Nodes: 107  | Edges: 352
```

```
cat("Node attributes:", paste(vertex_attr_names(g_got), collapse =
 ↪  ", "), "\n")
```

```
Node attributes: name, Label
```

```
cat("Edge attributes:", paste(edge_attr_names(g_got), collapse = ",
 ↪  "), "\n\n")
```

```
Edge attributes: Weight
```

```
cat("=== Florentine Families ===\n")
```

```
=== Florentine Families ===
```

```
cat("Nodes:", vcount(g_flo), " | Edges:", ecount(g_flo), "\n")
```

```
Nodes: 16  | Edges: 20
```

```
cat("Node names:", paste(V(g_flo)$name, collapse = ", "), "\n")
```

```
Node names: Acciaiuoli, Albizzi, Barbadori, Bischeri, Castellani,
 ↪  Ginori, Guadagni, Lamberteschi, Medici, Pazzi, Peruzzi, Pucci,
 ↪  Ridolfi, Salviati, Strozzi, Tornabuoni
```

7.3.3 Adding computed attributes

Attributes added to vertex or edge sequences are stored on the graph object and can be used directly in `plot()` calls. Derive key structural properties upfront:

```
# Node-level measures
V(g_got)$degree      <- degree(g_got)
V(g_got)$betweenness <- betweenness(g_got, normalized = TRUE)
V(g_got)$community   <- membership(cluster_louvain(g_got,
 ↪  resolution = 1))

V(g_flo)$degree      <- degree(g_flo)
V(g_flo)$community   <- membership(cluster_walktrap(g_flo))

# Edge-level: normalise weight to [0, 1] for visual scaling
E(g_got)$weight_norm <- E(g_got)$Weight / max(E(g_got)$Weight)
```

7.3.4 Working with large networks: subsetting

The full GoT network with 107 nodes is readable but crowded. For pedagogical clarity we will often use the **main connected component** filtered to characters with at least 10 co-occurrence connections:

```
# Retain only the largest connected component
comp     <- components(g_got)
g_main   <- induced_subgraph(g_got, which(comp$membership ==
 ↪  which.max(comp$csize)))

# Further filter to higher-degree nodes for clarity
g_sub    <- induced_subgraph(g_main, V(g_main)[degree(g_main) >=
 ↪  10])

cat("Subset: ", vcount(g_sub), "nodes,", ecount(g_sub), "edges\n")
```

```
Subset:  20 nodes, 87 edges
```

> **When to filter?** Subsetting changes the network — betweenness and degree will shift. Always clarify whether you are plotting a subgraph or the full network, and recompute centrality measures after filtering.

7.4 4. Plotting Networks with `igraph`

`igraph`'s `plot()` function provides direct control over every visual element through a family of named parameters. The mental model is simple: vertex and edge parameters

accept either a scalar (applied uniformly) or a vector of the same length as the number of vertices/edges (applied per element).

7.4.0.1 The parameter taxonomy

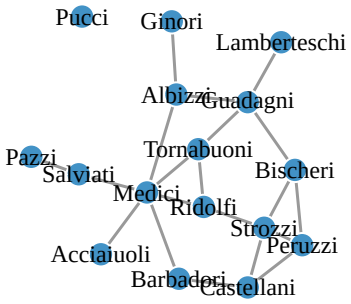
Scope	Parameter	Controls
Vertex	`vertex.color`	fill colour
Vertex	`vertex.frame.color`	border colour
Vertex	`vertex.size`	radius (default 15)
Vertex	`vertex.shape`	`"circle"`, `"square"`, `"csquare"`, `"rectangle"`, `"none"`
Vertex	`vertex.label`	label text (`NA` suppresses)
Vertex	`vertex.label.color`	label colour
Vertex	`vertex.label.cex`	label size multiplier
Vertex	`vertex.label.dist`	offset label from node centre
Edge	`edge.color`	edge colour
Edge	`edge.width`	line width
Edge	`edge.lty`	line type (1 = solid, 2 = dashed …)
Edge	`edge.curved`	curvature 0–1 (or negative)
Edge	`edge.arrow.size`	arrowhead size (directed graphs)
Edge	`edge.label`	label text
Graph	`layout`	layout matrix or function
Graph	`main`	plot title
Graph	`margin`	plot margins (default `c(0,0,0,0)`)

7.4.0.2 A baseline plot

```
set.seed(42)
plot(
  g_flo,
  main          = "Florentine Marriage Network",
  vertex.color  = "#4292c6",
  vertex.size   = 18,
  vertex.frame.color = "white",
  vertex.label.color = "black",
```

```
  vertex.label.cex   = 0.75,
  edge.color    = "grey60",
  edge.width    = 1.5
)
```

Florentine Marriage Network

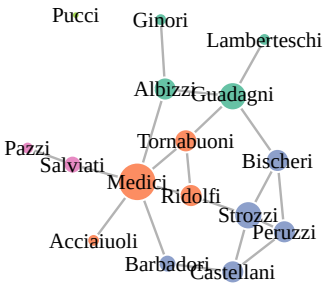


7.4.0.3 Mapping attributes to parameters

The power of the vector-valued parameters: map a node attribute directly to size or colour.

```
# Size by degree, colour by community
n_comm <- max(V(g_flo)$community)
comm_pal <- brewer.pal(max(n_comm, 3), "Set2")[1:n_comm]

set.seed(42)
plot(
  g_flo,
  main               = "Size  degree, colour = community",
  vertex.color       = comm_pal[V(g_flo)$community],
  vertex.size        = 6 + V(g_flo)$degree * 4,
  vertex.frame.color = "white",
  vertex.label.color = "black",
  vertex.label.cex   = 0.65,
  edge.color         = "grey70",
  edge.width         = 1.2
)
```

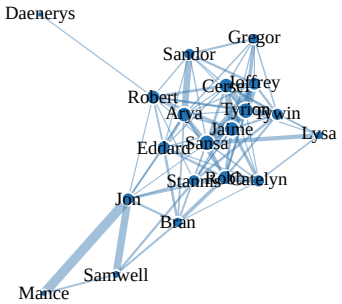
Size ... degree, colour = community



7.4.0.4 Encoding edge weight

```
# Edge width and opacity scaled by normalised weight
E(g_sub)$weight_norm <- E(g_sub)$Weight / max(E(g_sub)$Weight)

set.seed(7)
plot(
  g_sub,
  main               = "GoT subset - edge width  co-occurrence
   ↪  weight",
  vertex.color       = "#2171b5",
  vertex.size        = 4 + degree(g_sub) * 0.6,
  vertex.frame.color = "white",
  vertex.label.color = "black",
  vertex.label.cex   = 0.55,
  edge.width         = E(g_sub)$weight_norm * 5 + 0.3,
  edge.color         = adjustcolor("steelblue", alpha.f = 0.5)
)
```

GoT subset − edge width ... co−occurrence weight



7.5 5. Network Layouts

A layout is a function that assigns `(x, y)` coordinates to every node. The choice of layout is one of the highest-leverage decisions in network visualisation: it determines which nodes appear close together, which edges seem to cross, and whether the network looks like a hairball or a comprehensible structure.

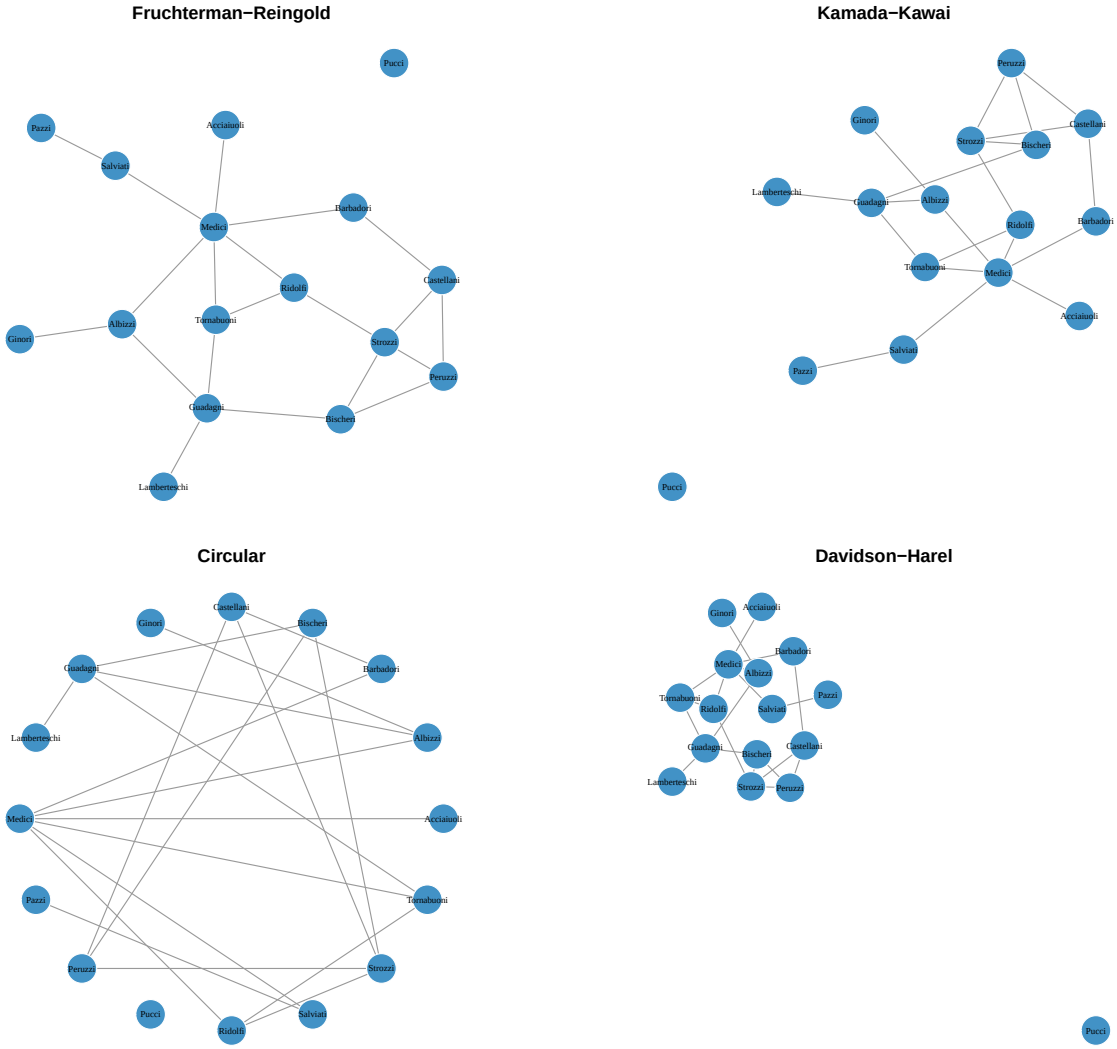
7.5.1 Layout Fundamentals

7.5.1.1 Key layout functions

Function	Algorithm	Best for
`layout_with_fr`	Fruchterman–Reingold	General purpose; nodes repel, edges pull
`layout_with_kk`	Kamada–Kawai	Smaller networks; minimises edge-length variance
`layout_in_circle`	Circular	Comparing degree sequence; showing cycles
`layout_as_star`	Star	Hub-and-spoke networks
`layout_as_tree`	Reingold–Tilford	Hierarchical / tree-like graphs
`layout_with_sugiyama`	Sugiyama	DAGs and directed hierarchies
`layout_with_dh`	Davidson–Harel	Often cleaner than FR for sparse graphs
`layout_with_gem`	GEM force-directed	Similar to FR, different energy function
`layout_randomly`	Uniform random	Baseline / stress-test of data
`layout_nicely`	Auto-selects	Good default for unknown graphs

7.5.1.2 Side-by-side layout comparison

```
layouts <- list(
  "Fruchterman-Reingold" = layout_with_fr(g_flo),
  "Kamada-Kawai"         = layout_with_kk(g_flo),
  "Circular"             = layout_in_circle(g_flo),
  "Davidson-Harel"       = layout_with_dh(g_flo)
)

par(mfrow = c(2, 2), mar = c(1, 1, 2, 1))
for (nm in names(layouts)) {
  plot(
    g_flo,
    layout             = layouts[[nm]],
    main               = nm,
    vertex.color       = "#4292c6",
    vertex.size        = 14,
    vertex.frame.color = "white",
    vertex.label.cex   = 0.65,
    vertex.label.color = "black",
    edge.color         = "grey60"
  )
}
```




```
par(mfrow = c(1, 1), mar = c(5.1, 4.1, 4.1, 2.1))  # restore
 ↪  defaults
```

Which layout should I use?

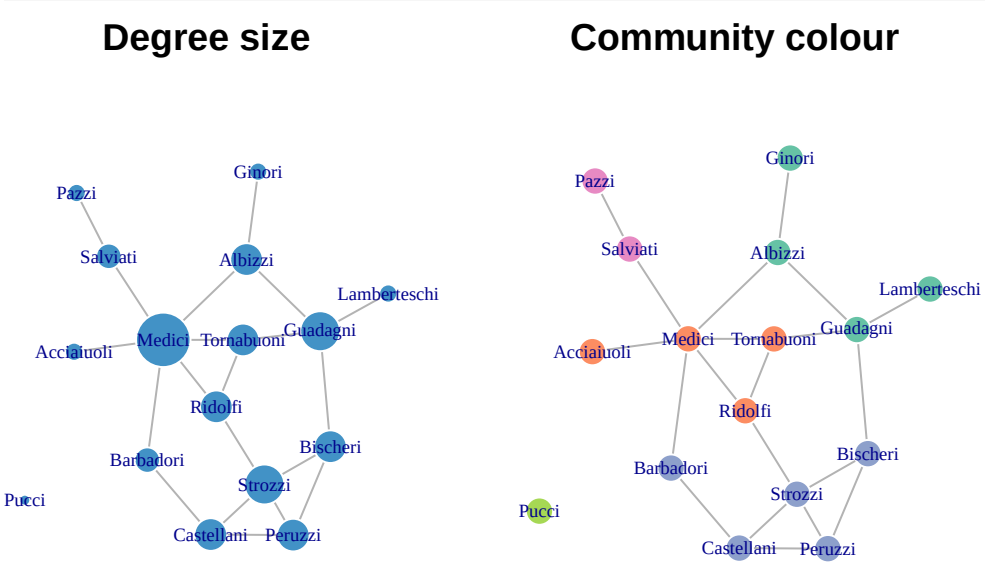
- **Force-directed** (FR, KK) work well for most social networks. FR is fast and handles larger graphs; KK tends to produce more even edge lengths for smaller graphs.
- **Circular** layouts are ideal for showing that a network has a cycle backbone, or for comparing degree across all nodes simultaneously.
- **Tree/hierarchical** layouts (Reingold–Tilford, Sugiyama) are only meaningful when the network actually has a hierarchical structure; applying them to non-hierarchical networks produces misleading images.

- Always use `set.seed()` before stochastic layouts (FR, GEM) to ensure reproducibility.

7.5.1.3 Fixing and reusing a layout

Store the layout as a matrix so that multiple plots of the same graph share the same node positions. This is essential when you want to compare different visual encodings of the same network.

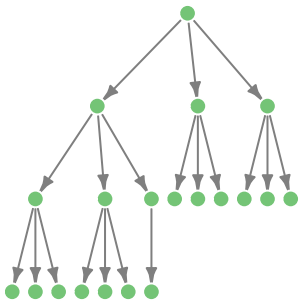
```
set.seed(2024)
flo_layout <- layout_with_fr(g_flo)   # nx2 matrix of (x,y)
 ↪  coordinates

# Now both plots use identical positions - differences are due to
 ↪  encoding only
par(mfrow = c(1, 2), mar = c(1, 1, 2, 1))

plot(g_flo, layout = flo_layout, main = "Degree size",
     vertex.size  = 6 + V(g_flo)$degree * 4,
     vertex.color = "#4292c6", vertex.frame.color = "white",
     vertex.label.cex = 0.6, edge.color = "grey70")

plot(g_flo, layout = flo_layout, main = "Community colour",
     vertex.color = comm_pal[V(g_flo)$community],
     vertex.size  = 14, vertex.frame.color = "white",
     vertex.label.cex = 0.6, edge.color = "grey70")
```



```
par(mfrow = c(1, 1), mar = c(5.1, 4.1, 4.1, 2.1))
```

7.5.1.4 Tree layout for directed / hierarchical graphs

```
# Create a small directed tree for demonstration
g_tree <- make_tree(n = 20, children = 3, mode = "out")

plot(
  g_tree,
  layout             = layout_as_tree(g_tree),
  main               = "Tree layout (make_tree, 3 children)",
  vertex.size        = 12,
  vertex.color       = "#74c476",
  vertex.frame.color = "white",
  vertex.label       = NA,
  edge.arrow.size    = 0.4,
  edge.color         = "grey50"
)
```



7.5.2 Layout Algorithm Comparison

A layout algorithm takes a graph and assigns each node an `(x, y)` position on a 2-D canvas. The *same* network can look radically different depending on which algorithm you choose, and choosing the right one matters because it affects which structures are visually salient. This section uses a synthetic **small-world network** as a shared test case, runs six algorithms side-by-side, and then dives into how the key parameters of Fruchterman–Reingold change the resulting picture.

7.5.2.1 Generating a test network

We use the **Watts-Strogatz** model (`sample_smallworld`): a classic benchmark that produces networks with high clustering and short average path lengths — properties found in many real-world social, biological, and technological networks.

```
set.seed(42)

# 200 nodes on a 1-D ring; each connected to its 3 nearest
 ↪  neighbours on each side;
# each edge independently rewired with probability 0.03
g_sw <- sample_smallworld(dim = 1, size = 200, nei = 3, p = 0.03)

cat("Nodes:                    ", vcount(g_sw), "\n")
```

```
Nodes:                     200
```

```
cat("Edges:                    ", ecount(g_sw), "\n")
```

```
Edges:                     600
```

```
cat("Average path length:    ", round(mean_distance(g_sw), 3),
 ↪  "\n")
```

```
Average path length:     5.297
```

```
cat("Clustering coefficient: ", round(transitivity(g_sw, type =
 ↪  "average"), 3), "\n")
```

```
Clustering coefficient:  0.518
```

> Short average path length + high clustering = hallmarks of a small-world network.

7.5.2.2 Shared visual style

To make a fair comparison we fix every visual parameter and only vary the `layout` argument.

```
sw_node_col  <- "#4682B4"   # steel blue
sw_edge_col  <- "#AAAAAA"   # light grey
sw_node_size <- 5
sw_edge_wid  <- 0.6
```

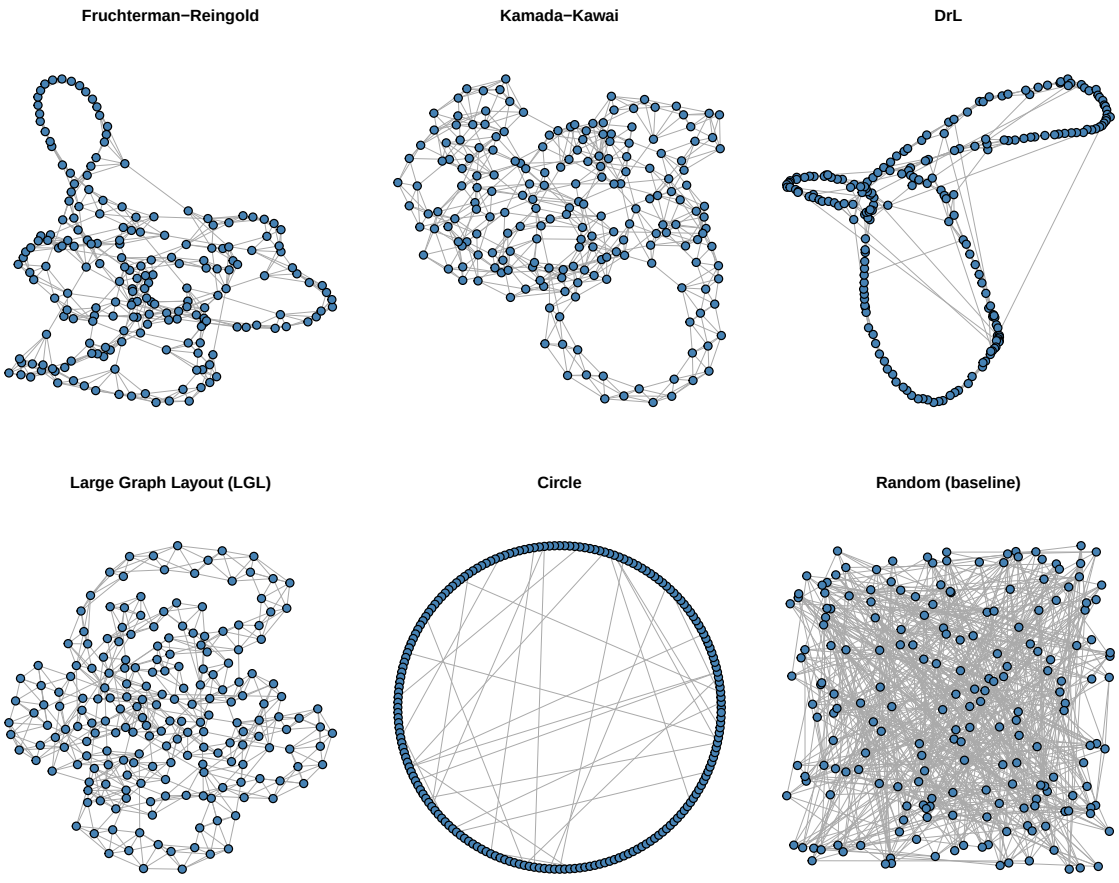
7.5.2.3 Six layouts side-by-side

All layouts are computed once before plotting to keep run-times predictable.

```
layout_fr     <- layout_with_fr(g_sw)     # Fruchterman-Reingold
 ↪  (force-directed)
layout_kk     <- layout_with_kk(g_sw)     # Kamada-Kawai
 ↪  (spring-electrical)
layout_drl    <- layout_with_drl(g_sw)    # DrL - designed for
 ↪  large graphs
layout_lgl    <- layout_with_lgl(g_sw)    # Large Graph Layout
layout_circle <- layout_in_circle(g_sw)   # Circular - purely
 ↪  deterministic
layout_random <- layout_randomly(g_sw)    # Random - no
 ↪  optimisation (baseline)
```

```
par(mfrow = c(2, 3), mar = c(1, 1, 2.5, 1))

plot(g_sw, layout = layout_fr,
     main = "Fruchterman-Reingold",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = layout_kk,
     main = "Kamada-Kawai",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = layout_drl,
     main = "DrL",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)
```

```
plot(g_sw, layout = layout_lgl,
     main = "Large Graph Layout (LGL)",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = layout_circle,
     main = "Circle",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = layout_random,
     main = "Random (baseline)",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)
```



```
par(mfrow = c(1, 1), mar = c(5, 4, 4, 2) + 0.1)
```

7.5.2.3.1 Algorithm cheat-sheet

Layout algorithms are not neutral — each one privileges certain structural properties. Before choosing a layout:

- **Consider graph size.** FR and KK work well up to a few thousand nodes; DrL and LGL are designed for much larger graphs.
- **Consider what you want to show.** Geodesic distance → KK. Clusters → FR with enough iterations. Hierarchy → LGL. Pure enumeration → Circle.
- **Tune parameters** when the default layout looks too crowded or too spread out. Start by adjusting `niter`, then `start.temp`.

Algorithm	Core idea	Best used when…
Fruchterman-Reingold	Nodes repel; edges act as springs	General purpose; medium-sized graphs
Kamada-Kawai	Minimises difference between graph-theoretic and geometric distance	Smaller graphs where geodesic distance should drive positions
DrL	Multilevel force-directed; groups nodes before placing them	Large graphs (thousands of nodes)
LGL	Builds outward from a root using minimum spanning tree	Large, sparse graphs with a clear hub
Circle	Equally spaced nodes on a ring	Showing all nodes at equal visual weight
Random	Positions drawn uniformly at random	Baseline only — never use in final visualisations

7.5.2.4 Fruchterman–Reingold parameter tuning

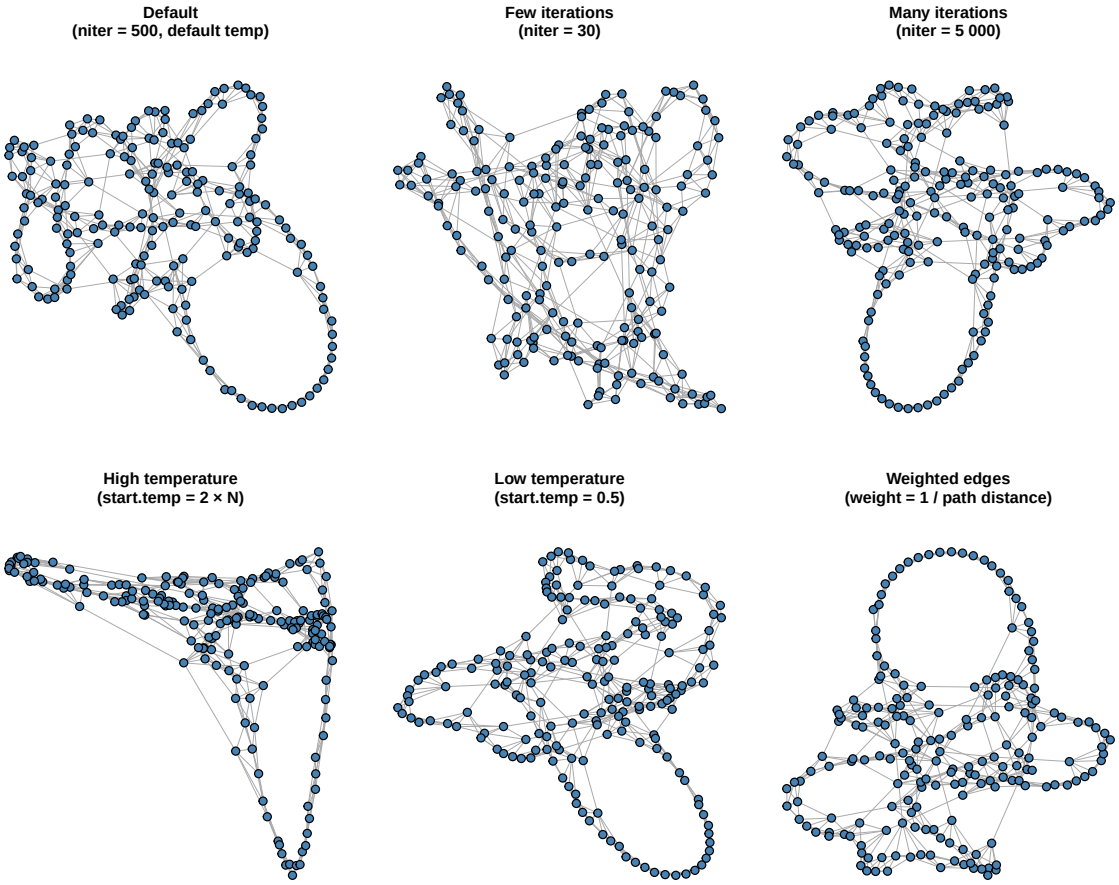
FR is one of the most widely used force-directed algorithms. Two parameters control its behaviour:

Parameter	Default	Effect
`niter`	500	Iterations the algorithm runs. More = more time to settle.
`start.temp`	`sqrt(node count)`	Starting "temperature" — the maximum distance a node can move per iteration. High temp → broad global exploration; low temp → local fine-tuning only.

```
# 1. Default - reference point
fr_default <- layout_with_fr(g_sw)

# 2. Very few iterations - layout has not had time to converge
fr_few_iter <- layout_with_fr(g_sw, niter = 30)

# 3. Many iterations - fully converged
fr_many_iter <- layout_with_fr(g_sw, niter = 5000)

# 4. High starting temperature - large displacements; global
 ↪  structure prioritised
fr_high_temp <- layout_with_fr(g_sw, start.temp = vcount(g_sw) * 2)

# 5. Low starting temperature - nodes barely move from random start
fr_low_temp <- layout_with_fr(g_sw, start.temp = 0.5)

# 6. Edge weights from graph-theoretic path distances:
#    weight = 1/distance pulls geodesically close neighbours
 ↪  tighter visually
path_dist    <- distances(g_sw)
edge_list    <- as_edgelist(g_sw, names = FALSE)
edge_weights <- 1 / path_dist[edge_list]
fr_weighted  <- layout_with_fr(g_sw, weights = edge_weights)
```

```
par(mfrow = c(2, 3), mar = c(1, 1, 3, 1))

plot(g_sw, layout = fr_default,
     main = "Default\n(niter = 500, default temp)",
```

```
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = fr_few_iter,
     main = "Few iterations\n(niter = 30)",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = fr_many_iter,
     main = "Many iterations\n(niter = 5 000)",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = fr_high_temp,
     main = "High temperature\n(start.temp = 2 × N)",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = fr_low_temp,
     main = "Low temperature\n(start.temp = 0.5)",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)

plot(g_sw, layout = fr_weighted,
     main = "Weighted edges\n(weight = 1 / path distance)",
     vertex.size = sw_node_size, vertex.color = sw_node_col,
     vertex.label = NA, edge.color = sw_edge_col, edge.width =
      ↪  sw_edge_wid)
```




```
par(mfrow = c(1, 1), mar = c(5, 4, 4, 2) + 0.1)
```

What to notice:

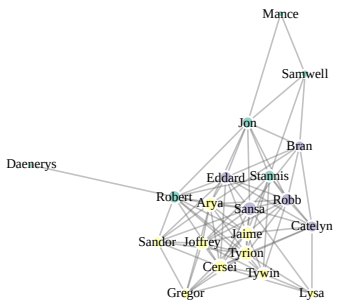
- **Few iterations (niter = 30):** Clusters are roughly visible but poorly resolved — the algorithm ran out of steps before nodes reached stable positions.
- **Many iterations (niter = 5 000):** More refined, but for a network this size the visual gain over the default is modest; FR converges well before 5 000 steps.
- **High temperature:** Nodes spread aggressively across the canvas — global cluster separation is exaggerated.
- **Low temperature:** Resembles a random layout because nodes never moved far from their initial positions.
- **Weighted edges:** Nodes that are close in graph-theoretic terms are pulled tightly together, making dense local clusters compact and their separation from the rest more obvious.

7.6 6. Highlighting Aspects of the Network

Static node-link diagrams work best when they draw attention to a specific structural feature. The most common targets: **community structure**, **centrality gradients**, and **cohesive subgroups**. This section shows how to highlight each using both igraph and ggraph.

7.6.0.1 Community detection and colouring

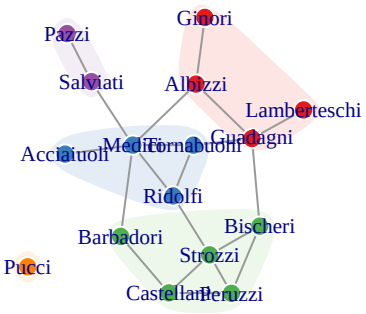
```
set.seed(42)
comm_got <- cluster_louvain(g_sub, resolution = 1)

n_comm_got <- length(unique(membership(comm_got)))
pal_got    <- brewer.pal(min(n_comm_got, 11),
 ↪  "Set3")[seq_len(n_comm_got)]

plot(
  g_sub,
  layout             = layout_with_fr(g_sub),
  main               = "GoT - Louvain communities",
  vertex.color       = pal_got[membership(comm_got)],
  vertex.size        = 4 + degree(g_sub) * 0.5,
  vertex.frame.color = "white",
  vertex.label.cex   = 0.45,
  vertex.label.color = "black",
  edge.color         = adjustcolor("grey40", 0.4),
  edge.width         = 0.8
)
```



7.6.0.2 Using `mark.groups` to draw hulls

The `mark.groups` parameter draws a convex hull around each community, making group structure immediately visible even in dense networks.

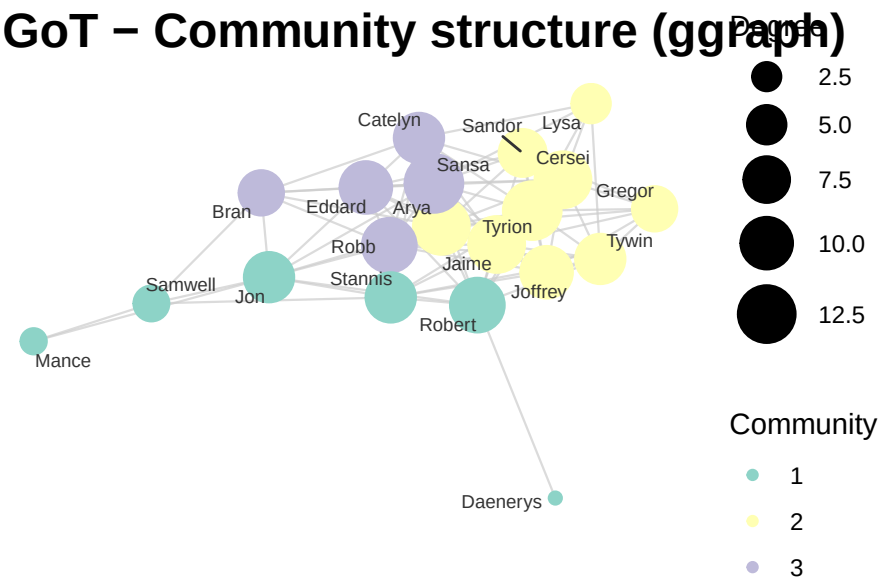
```
set.seed(42)
comm_flo  <- cluster_walktrap(g_flo)
hull_cols <- adjustcolor(brewer.pal(max(membership(comm_flo), 3),
 ↪  "Pastel1"), alpha.f = 0.35)

plot(
  g_flo,
  layout             = flo_layout,
  main               = "Florentine families - community hulls",
  mark.groups        = communities(comm_flo),
  mark.col           = hull_cols[seq_along(communities(comm_flo))],
  mark.border        = NA,
  vertex.color       = brewer.pal(max(membership(comm_flo), 3),
   ↪  "Set1")[membership(comm_flo)],
  vertex.size        = 14,
  vertex.frame.color = "white",
  vertex.label.cex   = 0.65,
  edge.color         = "grey60"
)
```



7.6.0.3 ggraph: grammar-of-graphics approach

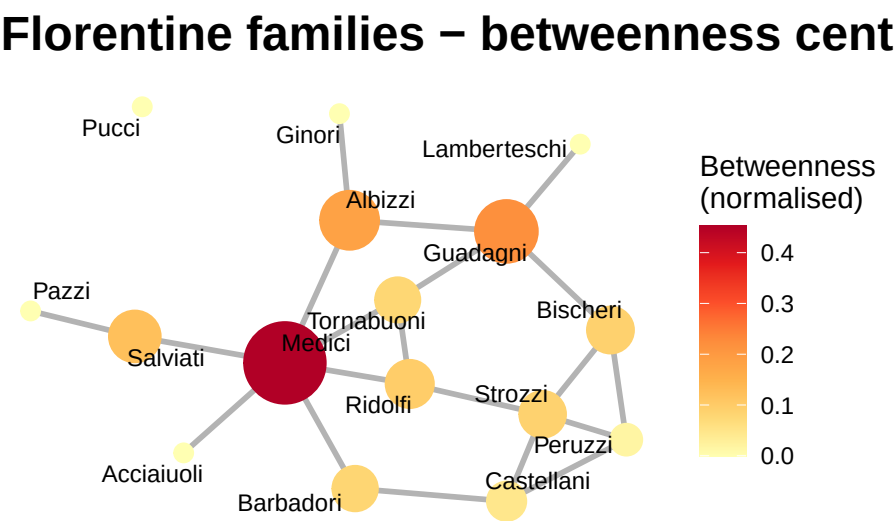
`ggraph` wraps `igraph` objects in ggplot2 syntax, enabling layered aesthetics, faceting, and full theme control. Convert any igraph object with `as_tbl_graph()`.

```
tbl_got <- as_tbl_graph(g_sub) |>
  activate(nodes) |>
  mutate(
    community  = as.factor(membership(cluster_louvain(g_sub,
     ↪  resolution = 1))),
    degree_val = degree(g_sub)
  )

set.seed(42)
ggraph(tbl_got, layout = "fr") +
  geom_edge_link(colour = "grey80", width = 0.4, alpha = 0.7) +
  geom_node_point(aes(colour = community, size = degree_val),
   ↪  show.legend = TRUE) +
  geom_node_text(aes(label = name), size = 2.5, repel = TRUE,
                 max.overlaps = 20, colour = "grey20") +
  scale_colour_brewer(palette = "Set3", name = "Community") +
  scale_size_continuous(range = c(2, 10), name = "Degree") +
  labs(title = "GoT - Community structure (ggraph)") +
  theme_graph(base_family = "sans") +
  theme(legend.position = "right")
```



7.6.0.4 Sizing nodes by centrality

```
tbl_flo <- as_tbl_graph(g_flo) |>
  activate(nodes) |>
  mutate(
    btw  = betweenness(g_flo, normalized = TRUE),
    deg  = degree(g_flo)
  )

set.seed(42)
ggraph(tbl_flo, layout = "fr") +
  geom_edge_link(colour = "grey70", width = 1) +
  geom_node_point(aes(size = btw, colour = btw)) +
  geom_node_text(aes(label = name), size = 3, repel = TRUE) +
  scale_colour_distiller(palette = "YlOrRd", direction = 1,
                         name = "Betweenness\n(normalised)") +
  scale_size_continuous(range = c(3, 14), guide = "none") +
  labs(title = "Florentine families - betweenness centrality") +
  theme_graph(base_family = "sans")
```



The Medici's structural dominance — the result of strategic marriage ties rather than raw wealth — is immediately visible as their node towers over the rest of the network.

7.7 7. Highlighting Specific Nodes or Edges

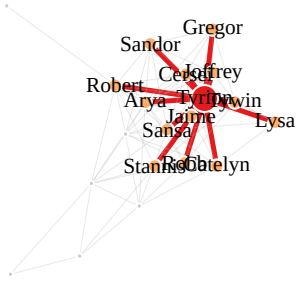
Sometimes the analysis question is about a particular actor (ego network), a specific pathway (shortest path), or a structural role (articulation points). These can all be highlighted by assigning a contrasting colour or size to the target vertices/edges.

7.7.0.1 Highlighting a named node (ego)

```
# Identify "Tyrion" and his immediate neighbours
target    <- "Tyrion"
ego_v     <- neighbors(g_sub, target)
all_ego_v <- c(V(g_sub)[target], ego_v)

# Build colour and size vectors: highlight target and neighbours
v_col  <- rep("grey80", vcount(g_sub))
v_col[V(g_sub)[target]] <- "#e41a1c"          # target = red
v_col[ego_v]            <- "#fdae6b"          # neighbours = orange

v_size <- rep(4, vcount(g_sub))
v_size[V(g_sub)[target]] <- 20
v_size[ego_v]             <- 10

# Edge highlight: dim all edges not touching the ego
e_touching <- incident(g_sub, V(g_sub)[target])
e_col <- rep(adjustcolor("grey70", 0.3), ecount(g_sub))
e_col[e_touching] <- "#e41a1c"
e_wid <- rep(0.4, ecount(g_sub))
e_wid[e_touching] <- 2.5

set.seed(7)
plot(
  g_sub,
  main              = paste0("Ego network: ", target),
  vertex.color      = v_col,
  vertex.size       = v_size,
  vertex.frame.color = "white",
  vertex.label      = ifelse(V(g_sub)$name %in% c(target,
   ↪  V(g_sub)[ego_v]$name),
                             V(g_sub)$name, NA),
  vertex.label.cex   = 0.7,
  vertex.label.color = "black",
  edge.color        = e_col,
```

```
  edge.width          = e_wid
)
```

Ego network: Tyrion

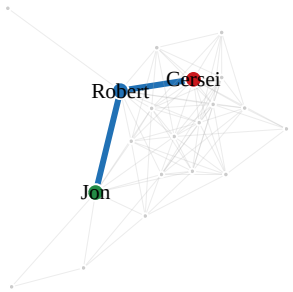


7.7.0.2 Highlighting a shortest path

```
from_node <- "Jon"
to_node   <- "Cersei"

sp <- shortest_paths(g_sub, from = from_node, to = to_node, output
 ↪  = "both")
sp_vids <- sp$vpath[[1]]
sp_eids <- sp$epath[[1]]

# Colour vectors
v_col2 <- rep("grey80", vcount(g_sub))
v_col2[sp_vids] <- "#2171b5"
v_col2[V(g_sub)[from_node]] <- "#238b45"
v_col2[V(g_sub)[to_node]]   <- "#cb181d"

e_col2 <- rep(adjustcolor("grey70", 0.25), ecount(g_sub))
e_col2[sp_eids] <- "#2171b5"

e_wid2 <- rep(0.3, ecount(g_sub))
e_wid2[sp_eids] <- 3

set.seed(7)
plot(
  g_sub,
```

```
  main               = paste0("Shortest path: ", from_node, " → ",
   ↪  to_node),
  vertex.color       = v_col2,
  vertex.size        = ifelse(V(g_sub)$name %in% sp_vids$name, 12,
   ↪  4),
  vertex.frame.color = "white",
  vertex.label       = ifelse(V(g_sub)$name %in% sp_vids$name,
   ↪  V(g_sub)$name, NA),
  vertex.label.cex   = 0.7,
  vertex.label.color = "black",
  edge.color         = e_col2,
  edge.width         = e_wid2
)
```

Shortest path: Jon –> Cersei



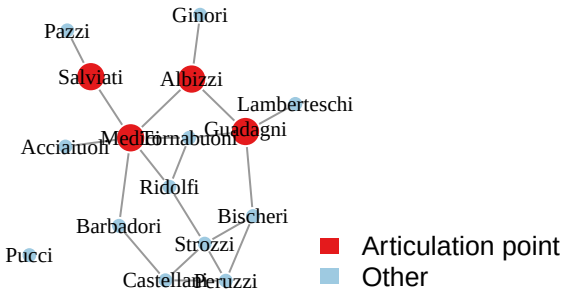
7.7.0.3 Highlighting articulation points (bridges)

Articulation points are nodes whose removal would disconnect the network — structurally critical even if their degree is modest.

```
art_pts <- articulation_points(g_flo)

v_col3 <- rep("#9ecae1", vcount(g_flo))
v_col3[art_pts] <- "#e41a1c"

v_size3 <- rep(12, vcount(g_flo))
v_size3[art_pts] <- 22

plot(
  g_flo,
```

```
  layout             = flo_layout,
  main               = "Articulation points (red) - Florentine
   ↪  families",
  vertex.color       = v_col3,
  vertex.size        = v_size3,
  vertex.frame.color = "white",
  vertex.label.cex   = 0.65,
  vertex.label.color = "black",
  edge.color         = "grey60"
)
legend("bottomright",
       legend = c("Articulation point", "Other"),
       fill   = c("#e41a1c", "#9ecae1"),
       border = NA, bty = "n", cex = 0.8)
```

Articulation points (red) − Florentine families



7.8 8. Plotting Bipartite (Two-Mode) Networks

A **bipartite** (two-mode) network contains two disjoint sets of nodes — actors and events, authors and papers, species and habitats — with edges running only *between* sets, never within. Visualising them requires communicating the two node types clearly.

7.8.0.1 Constructing a bipartite graph

```
# Define node sets
researchers <- c(
  "Alice Smith", "Bob Jones",   "Carol McGregor",
  "David Park",  "Emma Wilson", "Frank Dow",  "Grace Kelly"
)

events <- c(
  "SNA Workshop", "AI Ethics Seminar", "Public Policy Forum",
  "DPhil Symposium", "Methods Masterclass"
)

# Incidence matrix: rows = researchers, columns = events
# Cell (i, j) = 1 if researcher i attended event j
attendance_mat <- matrix(
  c(1, 0, 1, 0, 1,
    0, 1, 1, 1, 0,
    1, 1, 0, 0, 1,
    0, 0, 1, 1, 1,
    1, 1, 0, 1, 0,
    0, 1, 1, 0, 1,
    1, 0, 0, 1, 1),
  nrow     = 7,
  byrow    = TRUE,
  dimnames = list(researchers, events)
)

g_bip <- graph_from_incidence_matrix(attendance_mat, directed =
 ↪  FALSE)

# Confirm bipartite structure
is_bipartite(g_bip)
```

```
[1] TRUE
```

```
# Inspect node types
tibble(
  name = V(g_bip)$name,
  type = ifelse(V(g_bip)$type, "Event", "Researcher")
)
```

```
# A tibble: 12 x 2
```

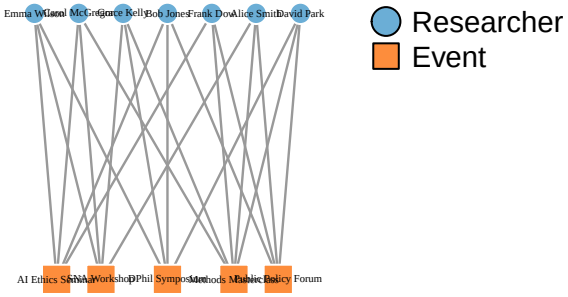
```
   name                type
   <chr>               <chr>
 1 Alice Smith         Researcher
 2 Bob Jones           Researcher
 3 Carol McGregor      Researcher
 4 David Park          Researcher
 5 Emma Wilson         Researcher
 6 Frank Dow           Researcher
 7 Grace Kelly         Researcher
 8 SNA Workshop        Event
 9 AI Ethics Seminar   Event
10 Public Policy Forum Event
11 DPhil Symposium     Event
12 Methods Masterclass Event
```

7.8.0.2 Plotting with shape and colour by type

```
# Visual encoding: shape distinguishes node type, colour reinforces
 ↪  it
# type == FALSE → Researcher; type == TRUE → Event
V(g_bip)$shape <- ifelse(V(g_bip)$type, "square", "circle")
V(g_bip)$color <- ifelse(V(g_bip)$type, "#fd8d3c", "#6baed6")
V(g_bip)$size  <- ifelse(V(g_bip)$type, 22, 16)

plot(
  g_bip,
  layout             = layout_as_bipartite(g_bip),
  main               = "Researcher-event attendance network",
  vertex.shape       = V(g_bip)$shape,
  vertex.color       = V(g_bip)$color,
  vertex.size        = V(g_bip)$size,
  vertex.frame.color = "white",
  vertex.label.cex   = 0.3,
  vertex.label.color = "black",
  edge.color         = "grey60",
  edge.width         = 1.2
)

legend("topright",
       legend = c("Researcher", "Event"),
       pch    = c(21, 22),
       pt.bg  = c("#6baed6", "#fd8d3c"),
```

```
        pt.cex = 2, bty = "n", cex = 0.9)
```

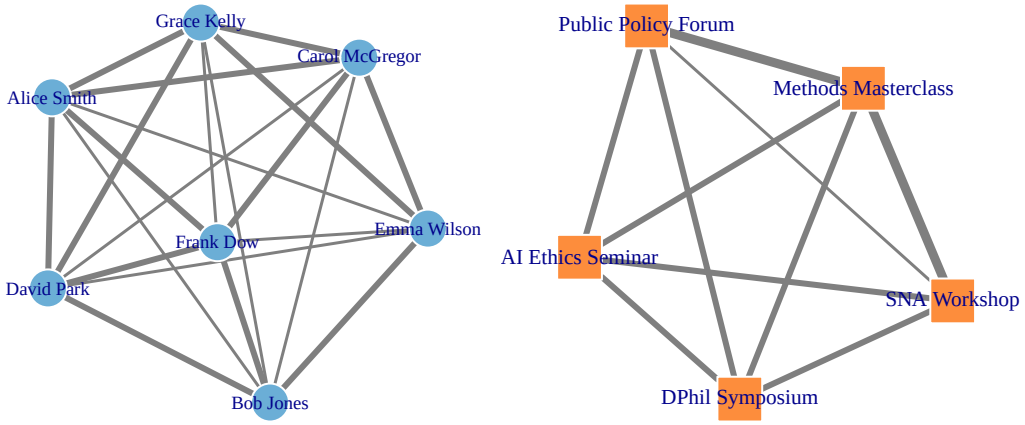
Researcher–event attendance network



7.8.0.3 One-mode projections

A bipartite network can be projected onto either node set: connect two researchers if they attended at least one common event, or connect two events if they share at least one attendee.

```
proj <- bipartite_projection(g_bip)

par(mfrow = c(1, 2), mar = c(1, 1, 2.5, 1))

plot(proj$proj1,
     main             = "Researcher co-attendance network",
     vertex.color     = "#6baed6",
     vertex.size      = 20,
     vertex.frame.color = "white",
     vertex.label.cex = 0.6,
     edge.color       = "grey50",
     edge.width       = E(proj$proj1)$weight * 1.5)

plot(proj$proj2,
     main             = "Event co-attendance network",
     vertex.color     = "#fd8d3c",
     vertex.size      = 24,
     vertex.frame.color = "white",
     vertex.label.cex = 0.65,
     edge.color       = "grey50",
     edge.width       = E(proj$proj2)$weight * 1.5)
```



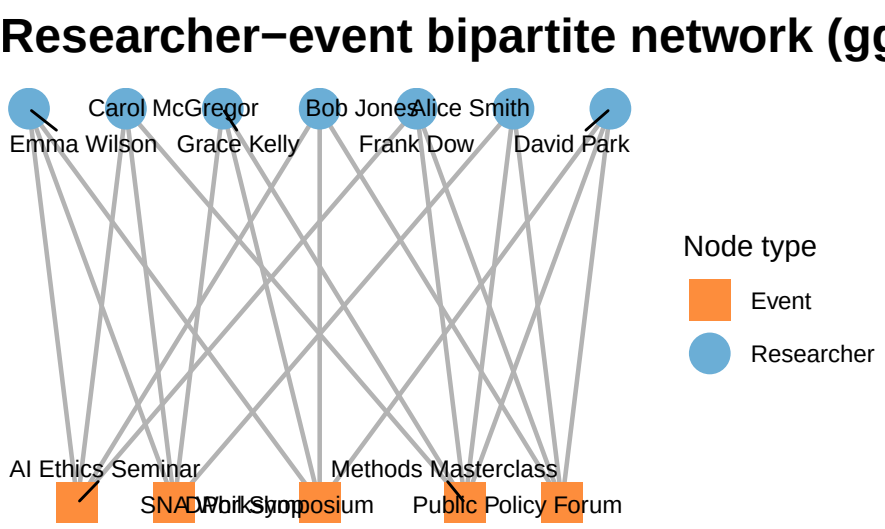

```
par(mfrow = c(1, 1), mar = c(5.1, 4.1, 4.1, 2.1))
```

Edge weight in the projection counts shared neighbours in the original bipartite network — here, the number of events two researchers both attended, or the number of researchers two events share.

7.8.0.4 ggraph bipartite layout

```
tbl_bip <- as_tbl_graph(g_bip) |>
  activate(nodes) |>
  mutate(
    node_type = ifelse(type, "Event", "Researcher")
  )

ggraph(tbl_bip, layout = "bipartite") +
  geom_edge_link(colour = "grey70", width = 0.8) +
  geom_node_point(aes(colour = node_type, shape = node_type), size
   ↪  = 7) +
  geom_node_text(aes(label = name), size = 3, repel = TRUE) +
  scale_colour_manual(values = c("Researcher" = "#6baed6", "Event"
   ↪  = "#fd8d3c"),
                      name = "Node type") +
  scale_shape_manual(values = c("Researcher" = 16, "Event" = 15),
                     name = "Node type") +
  labs(title = "Researcher-event bipartite network (ggraph)") +
```

```
    theme_graph(base_family = "sans")
```



7.9 9. Summary

Network visualisation is part craft, part argument. The technical choices covered in this tutorial — layout, colour, size, shape, filtering — are also interpretive choices. None of them is neutral. Every plot you produce is a claim about which aspects of the network deserve attention.

A few principles to carry forward:

- **Fix your layout** before comparing encodings. Identical node positions are the only fair basis for comparison.
- **Match colour scale to data type.** Qualitative palettes for categories; sequential or diverging for continuous measures.
- **Filter deliberately.** Subgraphs change all centrality measures — always recompute and always state what was removed.
- **Show multiple views.** No single plot captures a network fully. Pair a global overview with a targeted highlight.
- **Label your choices.** Titles, legends, and captions are not decoration — they are part of the argument.

7.9.1 Quick-reference: key functions

Data

Task	Function
Load from edge list	`graph_from_data_frame()`
Load from adjacency matrix	`graph_from_adjacency_matrix()`
Load from incidence matrix	`graph_from_incidence_matrix()`
Extract largest component	`induced_subgraph()` + `components()`
Filter edges by type	`subgraph.edges()`

Layout

Algorithm	Function	Best for
Fruchterman–Reingold	`layout_with_fr()`	General purpose
Kamada–Kawai	`layout_with_kk()`	Small graphs, geodesic fidelity
DrL	`layout_with_drl()`	Large graphs (1 000+ nodes)
Large Graph Layout	`layout_with_lgl()`	Large, sparse, hub-structured
Sugiyama (layered)	`layout_with_sugiyama()`	Hierarchical / multiplex panels
Circular	`layout_in_circle()`	Equal visual weight; cycle structure

Visual encoding

Task	Function / parameter
Qualitative colour palette	`brewer.pal(n, "Dark2")`
Sequential colour ramp	`colorRampPalette(brewer.pal(9, "Blues"))(n)`
Map continuous values to colour	`val_to_col()` (custom helper in §2)
Community detection	`cluster_louvain()`, `cluster_walktrap()`
Community hulls	`mark.groups` in `plot.igraph()`
Highlight specific nodes/edges	Build colour/size vectors indexed by `V(g)` / `E(g)`

Special network types

Type	Key functions
Bipartite (two-mode)	`graph_from_incidence_matrix()`, `layout_as_bipartite()`, `bipartite_projection()`
Multiplex (multi-layer)	`subgraph.edges()` + shared layout matrix
Grammar-of-graphics	`as_tbl_graph()` → `ggraph()` + `geom_edge_link()` + `geom_node_point()`

7.9.2 Further reading

- Ognyanova, K. (2025). *Network visualization with R.* kateto.net/network-visualization
- Luke, D. A. (2015). *A User's Guide to Network Analysis in R.* Springer.

8 Statistical Models for Networks

Inferential approaches to network structure

Descriptive measures tell us what networks look like; statistical models let us test hypotheses about *why* they look that way. This chapter introduces inferential frameworks for networks, focusing on **Exponential Random Graph Models (ERGMs)** — the workhorse of statistical network analysis in the social sciences.

8.1 Why Standard Regression Fails for Networks

Before diving into ERGMs, it is worth understanding why we cannot simply regress a tie indicator (edge = 1, no edge = 0) on node or dyad attributes using OLS or logistic regression.

The core problem is **non-independence**. In standard regression, we assume that each observation is independent of every other. In a network, that assumption is violated almost by definition:

- Whether Alice is friends with Bob likely depends on whether Alice is friends with Carol, and whether Carol is friends with Bob (transitivity / triadic closure).
- A node with many friends is more likely to acquire *even more* friends (degree-based effects / preferential attachment).
- Ties between pairs of nodes are not drawn independently — the whole point of network analysis is that structure is relational and recursive.

If you ignore this and run a naive logistic regression on dyad-level tie indicators, your standard errors are likely wrong, and you may overstate statistical significance. The network is not a bag of independent coin flips — it is a *system*.

Exponential Random Graph Models (ERGMs, also called p* models) address this directly. An ERGM defines a probability distribution over the entire space of possible networks of a given size. It asks: given some set of network statistics (edge count,

triangles, degree distribution, nodal covariates), what is the probability of observing *this particular network*?

Formally, an ERGM specifies:

$$P(Y = y) = \frac{\exp\left(\theta^\top \mathbf{g}(y)\right)}{\kappa(\theta)}$$

where:

- Y is the random network (each Y_{ij} is a potential tie),
- y is the observed network,
- $\mathbf{g}(y)$ is a vector of network statistics (e.g., number of edges, triangles, attribute-based counts),
- θ is the corresponding vector of parameters to be estimated,
- $\kappa(\theta)$ is a normalising constant summing over *all possible networks* of the same size — which is what makes these models computationally demanding (and is why estimation uses MCMC).

Positive θ for a statistic means that networks with more of that feature are more likely than chance; negative means fewer. This gives us an intuitive, hypothesis-driven framework for asking: do triangles appear more often than we would expect at random? Do wealthy nodes preferentially connect to one another?

8.2 Setup

```
# Load only the statnet components needed for ERGMs
# (avoids dependency on tergm and other optional suite packages)
library(network)   # network objects
library(ergm)      # ERGM estimation and GoF
library(sna)       # network descriptives (gden, etc.)
library(ggplot2)   # for any custom plotting
```

8.3 The Florentine Families Dataset

We will work with the **Florentine Families** dataset — a classic in social network analysis, first described by Padgett & Ansell (1993). It captures **marriage ties** among 16 prominent Florentine families in 15th-century Renaissance Italy. The Medici family's rise to political dominance is a central story in the dataset: their strategic position in the marriage network gave them outsized influence.

The dataset is bundled with `ergm` (part of `statnet`) as `flomarriage`.

```
# Load the dataset
# Load the florentine families data
data(florentine)
data(flomarriage)

# Inspect the network object
flomarriage
```

```
Network attributes:
 vertices = 16
 directed = FALSE
 hyper = FALSE
 loops = FALSE
 multiple = FALSE
 bipartite = FALSE
 total edges= 20
   missing edges= 0
   non-missing edges= 20

Vertex attribute names:
   priorates totalties vertex.names wealth

No edge attributes
```

```
# Node-level attributes
flomarriage %v% "wealth"     # Annual income in thousands of lira
 ↪  (1427)
```

```
[1]  10  36  55  44  20  32   8  42 103  48  49   3  27  10 146  48
```

```
flomarriage %v% "priorates"  # Number of seats on city council
 ↪  1282-1344
```

```
 [1] 53 65  0 12 22  0 21  0 53  0 42  0 38 35 74  0
```

```
flomarriage %v% "vertex.names"
```

```
 [1] "Acciaiuoli"   "Albizzi"      "Barbadori"    "Bischeri"
 ↪  "Castellani"
 [6] "Ginori"       "Guadagni"     "Lamberteschi" "Medici"       "Pazzi"
[11] "Peruzzi"      "Pucci"        "Ridolfi"      "Salviati"
 ↪  "Strozzi"
[16] "Tornabuoni"
```

```
# Basic network descriptives
network.size(flomarriage)        # Number of nodes
```

```
[1] 16
```

```
network.edgecount(flomarriage)   # Number of edges
```

```
[1] 20
```

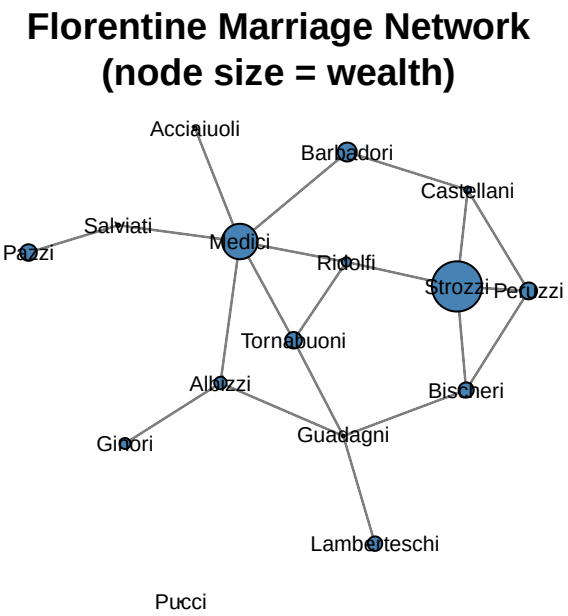
```
gden(flomarriage)                # Density
```

```
[1] 0.1666667
```

The network has 16 nodes (families) and 20 marriage ties. With a density of roughly 0.17, ties are sparse — typical of elite social networks where strategic marriages were carefully chosen, not randomly distributed.

```
# Visualise the network with wealth encoded in node size
par(mar = c(1, 1, 2, 1))

plot(
  flomarriage,
  vertex.cex    = flomarriage %v% "wealth" / 30,
  vertex.col    = "steelblue",
  edge.col      = "grey50",
  label         = flomarriage %v% "vertex.names",
  label.cex     = 0.7,
  label.pos     = 5,
  main          = "Florentine Marriage Network\n(node size =
   ↪  wealth)"
```

```
)
```



The Medici sit at the structural heart of this network. Despite not being the wealthiest family, they bridged otherwise disconnected clusters — a textbook example of brokerage and betweenness centrality translating into political power.

8.4 Model 1: The Bernoulli (Null) Model

We start with the simplest possible ERGM: a **Bernoulli graph** or *Erdős–Rényi* model. The only term is `edges`, which acts like an intercept — it captures the baseline probability of a tie forming, and nothing else.

```
model_null <- ergm(flomarriage ~ edges)
summary(model_null)
```

```
Call:
ergm(formula = flomarriage ~ edges)

Maximum Likelihood Results:

      Estimate Std. Error MCMC % z value Pr(>|z|)
```

```
edges  -1.6094     0.2449      0  -6.571   <1e-04 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

     Null Deviance: 166.4  on 120  degrees of freedom
 Residual Deviance: 108.1  on 119  degrees of freedom

AIC: 110.1  BIC: 112.9  (Smaller is better. MC Std. Err. = 0)
```

The coefficient for `edges` is negative, which makes sense: most possible ties are *absent* (the network is sparse). This log-odds can be converted to a probability:

```
# Tie probability under null model
plogis(coef(model_null)["edges"])
```

```
    edges
0.1666667
```

This matches the network density exactly — as it should. The null model is just a fancy way of saying "each tie forms independently with probability equal to the overall density." It is our baseline: every more complex model will be judged against it.

8.5 Model 2: Structural Effects — Transitivity and GWESP

Real social networks are not Bernoulli. One of the most robust findings in network science is **triadic closure**: if A is tied to B, and B is tied to C, then A and C are more likely to be tied too. Friends of friends become friends. In the Florentine context: if the Medici married into the Tornabuoni *and* the Tornabuoni are linked to the Strozzi, the Medici-Strozzi tie becomes more likely.

The naive way to capture this is the `triangle` term — counting the number of closed triangles in the network. However, `triangle` is notorious for causing **model degeneracy**: the ERGM collapses onto a near-complete or near-empty graph, producing nonsensical estimates. This is a known pathology of the raw triangle term.

The solution is the **Geometrically Weighted Edgewise Shared Partners** (GWESP) statistic, introduced by Hunter & Handcock (2006). GWESP counts shared partners but applies a geometric decay: the *first* shared partner between a dyad contributes fully, the *second* contributes less, and so on. This prevents the model from being dominated by a few highly-triangulated nodes and greatly improves stability.

```r
model_structural <- ergm(
  flomarriage ~ edges + triangle + gwesp(0.5, fixed = TRUE),
  control = control.ergm(seed = 42)
)
summary(model_structural)
```

```
Call:
ergm(formula = flomarriage ~ edges + triangle + gwesp(0.5, fixed =
 ↪  TRUE),
    control = control.ergm(seed = 42))

Monte Carlo Maximum Likelihood Results:

                Estimate Std. Error MCMC % z value Pr(>|z|)
edges            -1.7666     0.3811      0  -4.635   <1e-04 ***
triangle         -2.3475     4.8488      0  -0.484    0.628
gwesp.fixed.0.5   1.0522     1.8857      0   0.558    0.577
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

     Null Deviance: 166.4  on 120  degrees of freedom
 Residual Deviance: 107.6  on 117  degrees of freedom

AIC: 113.6  BIC: 122  (Smaller is better. MC Std. Err. = 0.02477)
```

Note on `triangle`: Including `triangle` alongside `gwesp` is illustrative here — in practice you would typically use one or the other (preferring `gwesp`). If you encounter degeneracy warnings, drop `triangle` and rely solely on `gwesp`.

```r
# Fit GWESP-only structural model (more stable)
model_gwesp <- ergm(
  flomarriage ~ edges + gwesp(0.5, fixed = TRUE),
  control = control.ergm(seed = 42)
)
summary(model_gwesp)
```

```
Call:
ergm(formula = flomarriage ~ edges + gwesp(0.5, fixed = TRUE),
    control = control.ergm(seed = 42))

Monte Carlo Maximum Likelihood Results:

                Estimate Std. Error MCMC % z value Pr(>|z|)
```

```
edges           -1.66281    0.38595      0  -4.308   <1e-04 ***
gwesp.fixed.0.5  0.06435    0.27643      0   0.233    0.816
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

     Null Deviance: 166.4  on 120  degrees of freedom
 Residual Deviance: 108.0  on 118  degrees of freedom

AIC: 112  BIC: 117.6  (Smaller is better. MC Std. Err. = 0.009081)
```

A positive and significant `gwesp` coefficient would tell us that triadic closure is a real force in this network — dyads that share at least one partner are more likely to be tied. However, the results are not statistical significant, so we may need to dig deeper.

8.6 Model 3: Homophily — Does Wealth Drive Tie Formation?

Network structure is shaped not only by internal dynamics (transitivity, degree) but also by **node attributes**. A key question for the Florentine data: did wealthier families preferentially form marriage ties?

We test two hypotheses:

1. **Wealth activity effect** (`nodecov`): Do wealthier families have *more ties in general*? A positive `nodecov` coefficient means that nodes with higher wealth values are more likely to be involved in any given dyad being a tie.
2. **Wealth homophily** (`absdiff`): Do families of *similar wealth* preferentially marry one another? The `absdiff` term computes the absolute difference in wealth between each pair of nodes; a *negative* coefficient would indicate homophily (smaller differences → more ties).

```
model_wealth <- ergm(
  flomarriage ~ edges + gwesp(0.5, fixed = TRUE) +
 ↪  nodecov("wealth") + absdiff("wealth"),
  control = control.ergm(seed = 42)
)
summary(model_wealth)
```

```
Call:
ergm(formula = flomarriage ~ edges + gwesp(0.5, fixed = TRUE) +
```

```
    nodecov("wealth") + absdiff("wealth"), control = control.ergm(seed =
    ↪  42))

Monte Carlo Maximum Likelihood Results:

                 Estimate Std. Error MCMC % z value Pr(>|z|)
edges           -2.502124   0.583978      0  -4.285   <1e-04 ***
gwesp.fixed.0.5 -0.024287   0.274399      0  -0.089    0.929
nodecov.wealth   0.004649   0.006870      0   0.677    0.499
absdiff.wealth   0.011303   0.008852      0   1.277    0.202
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

     Null Deviance: 166.4  on 120  degrees of freedom
 Residual Deviance: 101.5  on 116  degrees of freedom

AIC: 109.5  BIC: 120.7  (Smaller is better. MC Std. Err. = 0.005503)
```

Interpreting the results:

- A positive `nodecov("wealth")` suggests wealthier families are more active in forming ties overall.
- A negative `absdiff("wealth")` would indicate wealth homophily — families with similar wealth are drawn together.
- We evaluate statistical significance through the z-scores (estimate / std. error) and p-values in the summary output.

8.7 Model Comparison: AIC and BIC

With multiple models in hand, we need a principled basis for comparison. **AIC** (Akaike Information Criterion) and **BIC** (Bayesian Information Criterion) penalise model complexity to guard against overfitting: lower is better.

```r
# Extract AIC and BIC for each model
model_list <- list(
  Null       = model_null,
  GWESP      = model_gwesp,
  Wealth     = model_wealth
)
```

```r
comparison_table <- data.frame(
  Model  = names(model_list),
  AIC    = sapply(model_list, AIC),
  BIC    = sapply(model_list, BIC),
  LogLik = sapply(model_list, logLik)
)

comparison_table
```

```
        Model      AIC      BIC    LogLik
Null     Null 110.1347 112.9222 -54.06735
GWESP   GWESP 112.0071 117.5821 -54.00357
Wealth Wealth 109.5343 120.6843 -50.76716
```

```r
# Tidy comparison with knitr (renders nicely in HTML)
knitr::kable(
  comparison_table,
  digits  = 2,
  caption = "Model comparison: Florentine marriage network ERGMs"
)
```

Table 8.1: Model comparison: Florentine marriage network ERGMs

	Model	AIC	BIC	LogLik
Null	Null	110.13	112.92	-54.07
GWESP	GWESP	112.01	117.58	-54.00
Wealth	Wealth	109.53	120.68	-50.77

A model with substantially lower AIC/BIC than the null model would suggest that the added structural or covariate terms meaningfully improve fit beyond what chance predicts.

In our example, the wealth covariates provide the only substantive improvement in fit (log-likelihood gain of ~3.3), and AIC marginally favours including them; but the effect is small. Moreover, GWESP does essentially nothing here. Triadic closure may simply not be a strong structural force in this small, sparse network, or, the likely reason here, the network is too small to estimate it reliably. BIC's preference for the Null is largely a function of the tiny sample (n = 16, 20 edges) — BIC is very conservative in small networks. The honest conclusion: with 16 nodes, you do not have the statistical power to decisively distinguish these models on AIC/BIC alone.

8.8 MCMC Diagnostics

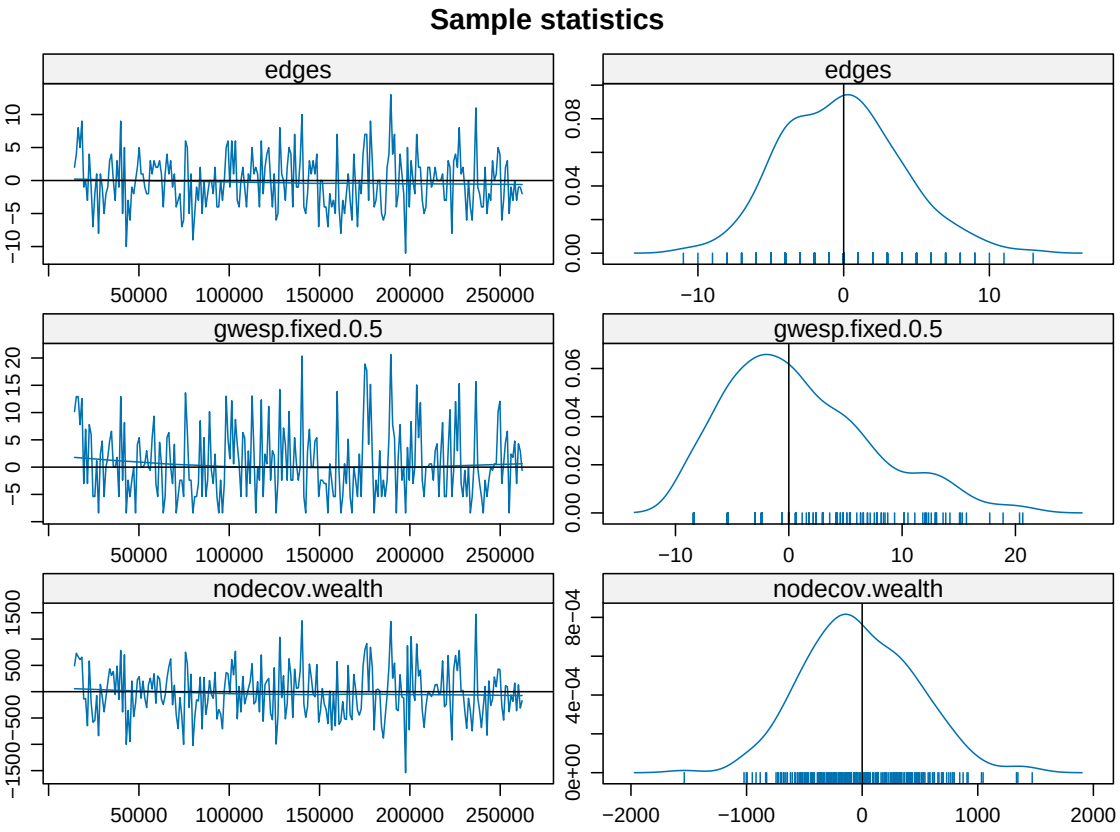
ERGMs are estimated using **Markov Chain Monte Carlo (MCMC)**. The MCMC sampler proposes changes to a simulated network and accepts or rejects them to approximate the normalising constant $\kappa(\theta)$ that cannot be computed analytically.

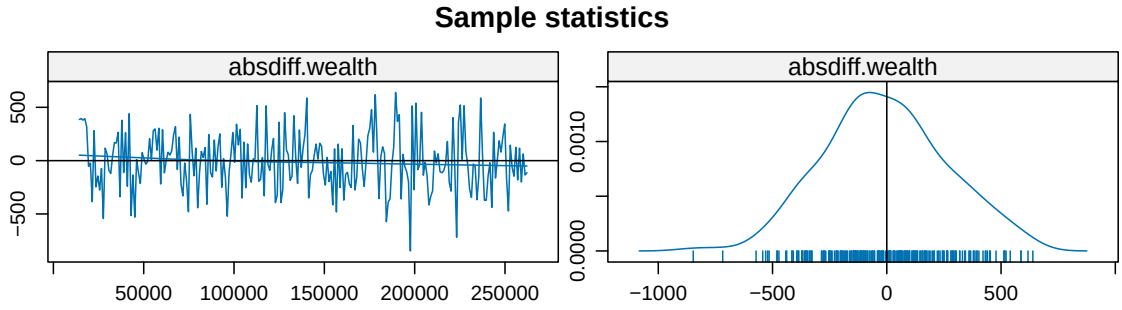
Before trusting your estimates, you *must* check that the MCMC chains have converged. A non-converged chain means the parameter estimates are unreliable, regardless of how significant the z-scores look.

`mcmc.diagnostics()` produces trace plots and autocorrelation diagnostics for each parameter. What you want to see:

- **Trace plots** that look like "fuzzy caterpillars" — bouncing randomly around a stable mean with no obvious trends or drift.
- **Low autocorrelation** — the chain is mixing well and not getting stuck.
- **Sample statistics close to observed** — the mean of the simulated statistics should match the observed statistics.

```
mcmc.diagnostics(model_wealth)
```






```
Sample statistics summary:

Iterations = 14336:262144
Thinning interval = 1024
Number of chains = 1
Sample size per chain = 243

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

                     Mean      SD Naive SE Time-series SE
edges             -0.1646   4.072   0.2612         0.2936
gwesp.fixed.0.5    0.9282   6.405   0.4109         0.4586
nodecov.wealth   -22.3704 479.054  30.7313        30.7313
absdiff.wealth    -8.5761 268.982  17.2552        17.2552

2. Quantiles for each variable:

                     2.5%    25%        50%     75%  97.5%
edges              -7.000   -3.0  0.000e+00   2.000   8.00
gwesp.fixed.0.5    -8.393   -3.0 -1.963e-13   4.787  15.16
nodecov.wealth   -916.200 -358.5 -6.700e+01 300.500 905.30
absdiff.wealth   -513.300 -188.0 -2.100e+01 164.000 515.95


Are sample statistics significantly different from observed?
                edges gwesp.fixed.0.5 nodecov.wealth absdiff.wealth
diff.      -0.1646091      0.92820257    -22.3703704     -8.5761317
test stat. -0.5606455      2.02400991     -0.7279333     -0.4970176
P-val.      0.5750392      0.04296913      0.4666544      0.6191766
                 (Omni)
diff.                NA
test stat. 2.495732e+01
P-val.     9.881306e-05
```

```
Sample statistics cross-correlations:
                   edges gwesp.fixed.0.5 nodecov.wealth absdiff.wealth
edges           1.0000000       0.8023854      0.8929658      0.7947443
gwesp.fixed.0.5 0.8023854       1.0000000      0.7939204      0.7070275
nodecov.wealth  0.8929658       0.7939204      1.0000000      0.9061106
absdiff.wealth  0.7947443       0.7070275      0.9061106      1.0000000

Sample statistics auto-correlation:
Chain 1
                edges gwesp.fixed.0.5 nodecov.wealth absdiff.wealth
Lag 0     1.000000000     1.000000000     1.00000000     1.00000000
Lag 1024  0.114214072     0.107340438     0.03285185    -0.02057090
Lag 2048  0.073221327     0.006196320     0.08855593     0.09934353
Lag 3072 -0.008798426     0.038623733    -0.01453905     0.02248722
Lag 4096  0.024251648     0.007719675    -0.02313542    -0.01861284
Lag 5120 -0.077534116    -0.048523005    -0.08001356    -0.11783132

Sample statistics burn-in diagnostic (Geweke):
Chain 1

Fraction in 1st window = 0.1
Fraction in 2nd window = 0.5

          edges gwesp.fixed.0.5  nodecov.wealth  absdiff.wealth
      0.7856964       1.2235716       0.6996823       0.5879705

Individual P-values (lower = worse):
          edges gwesp.fixed.0.5  nodecov.wealth  absdiff.wealth
      0.4320454       0.2211139       0.4841258       0.5565521
Joint P-value (lower = worse):  0.7542335

Note: MCMC diagnostics shown here are from the last round of
  simulation, prior to computation of final parameter estimates.
  Because the final estimates are refinements of those used for this
  simulation run, these diagnostics may understate model performance.
  To directly assess the performance of the final model on in-model
  statistics, please use the GOF command: gof(ergmFitObject,
  GOF=~model).
```

If you see systematic drift, poor mixing, or large discrepancies between simulated and observed statistics, you may need to adjust MCMC settings (longer chains, thinner, more burnin) via `control.ergm()`.

8.9 Goodness of Fit

Even if MCMC has converged, we need to check whether the model actually *fits* the observed network. **Goodness of Fit (GoF)** works as follows:

1. Simulate a large number of networks from the fitted ERGM.
2. Compute key network statistics for each simulated network.
3. Compare those distributions to the corresponding statistics of the *observed* network.

If the observed statistics fall within the bulk of the simulated distribution, the model fits. If they fall in the tails, the model is missing something important.

The standard GoF diagnostic in `ergm` checks four statistics:

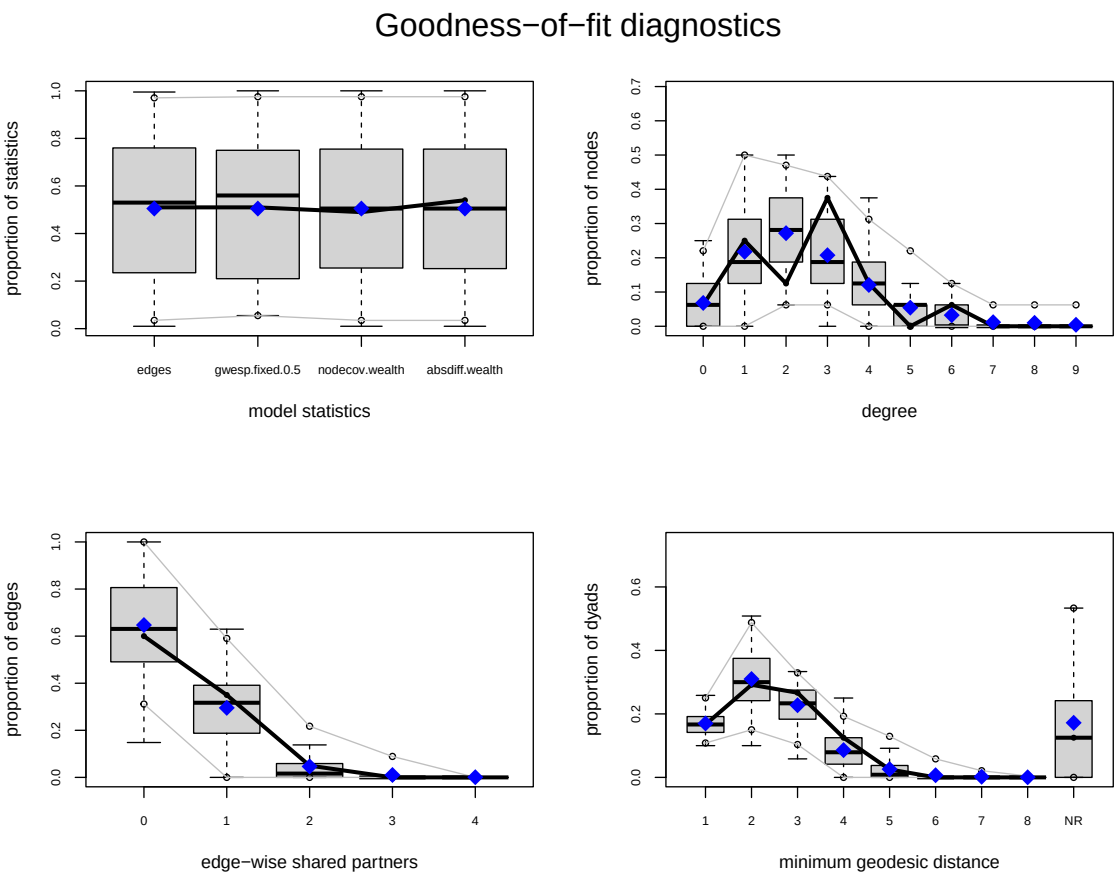
- **Degree distribution**: does the model reproduce the distribution of node degrees?
- **Edgewise shared partners (ESP)**: does the model reproduce local clustering structure?
- **Geodesic distance distribution**: does the model reproduce path length structure?
- **Model statistics**: do simulated networks reproduce the terms in the model?

```
gof_wealth <- gof(model_wealth)
gof_wealth
```

```
Goodness-of-fit for degree

         obs min mean max MC p-value
degree0    1   0 1.09   4       1.00
degree1    4   0 3.48  11       0.88
degree2    2   1 4.35   8       0.32
degree3    6   0 3.32   7       0.24
degree4    2   0 1.93   6       1.00
degree5    0   0 0.87   6       0.88
degree6    1   0 0.52   6       0.84
degree7    0   0 0.19   2       1.00
degree8    0   0 0.16   1       1.00
degree9    0   0 0.07   1       1.00
degree10   0   0 0.01   1       1.00
degree11   0   0 0.01   1       1.00

Goodness-of-fit for edgewise shared partner

     obs min  mean max MC p-value
```

```
esp0  12   4 12.71  24       0.98
esp1   7   0  6.32  19       0.98
esp2   1   0  1.09  11       1.00
esp3   0   0  0.25   3       1.00
esp4   0   0  0.02   1       1.00

Goodness-of-fit for minimum geodesic distance

    obs min  mean max MC p-value
1    20  12 20.39  33       1.00
2    35  12 37.18  61       0.96
3    32   5 27.28  40       0.60
4    15   0 10.31  30       0.52
5     3   0  3.10  17       0.82
6     0   0  0.82  13       1.00
7     0   0  0.24  10       1.00
8     0   0  0.04   2       1.00
Inf  15   0 20.64  89       1.00

Goodness-of-fit for model statistics

                         obs  min        mean        max MC p-value
edges              20.000000   12   20.390000   33.00000       1.00
gwesp.fixed.0.5     8.393469    0    8.258138   30.87645       1.00
nodecov.wealth   2168.000000 1208 2201.530000 3404.00000       0.98
absdiff.wealth   1146.000000  637 1162.990000 1990.00000       0.94
```

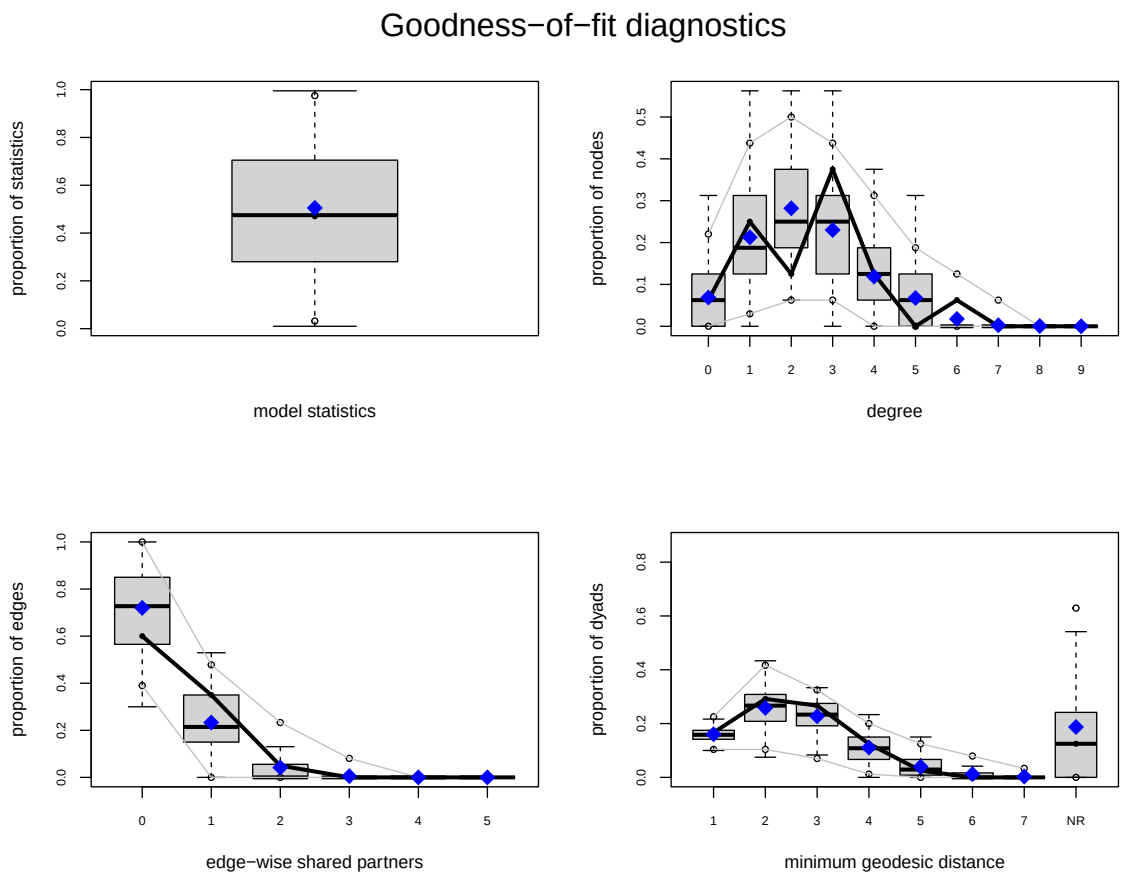
```
par(mfrow = c(2, 2))
plot(gof_wealth)
```



In each panel, the **black line** is the observed statistic; the **boxplots** summarise the distribution across simulated networks; the **grey band** is the 95% confidence envelope. Observed values that fall comfortably within the envelope indicate a well-fitting model.

Compare GoF for the null model:

```
gof_null <- gof(model_null)
par(mfrow = c(2, 2))
plot(gof_null)
```



The null model typically fails on degree distribution and ESP — it cannot reproduce the clustering we see in the observed network. The richer models should show a marked improvement.

8.10 Interpreting ERGM Coefficients

ERGM parameters are on the **log-odds** scale for individual tie changes, *holding the rest of the network fixed*. This "change statistic" interpretation is subtle but important:

- The `edges` coefficient is the log-odds of a tie forming between two nodes when no other network statistics would change — i.e., the baseline propensity.
- The `gwesp` coefficient is the *additional* log-odds of a tie forming for each additional (geometrically weighted) shared partner the dyad has.
- The `nodecov("wealth")` coefficient is the additional log-odds contribution per unit increase in the *sum* of the two nodes' wealth values.

- The `absdiff("wealth")` coefficient is the additional log-odds contribution per unit increase in the *absolute difference* in wealth.

```
# Convert log-odds to odds ratios for easier interpretation
exp(coef(model_wealth))
```

```
      edges gwesp.fixed.0.5  nodecov.wealth  absdiff.wealth
 0.08191087      0.97600552      1.00465963      1.01136696
```

An odds ratio > 1 increases the probability of a tie; < 1 decreases it. For example, if `exp(coef(model_wealth)["gwesp.fixed.0.5"])` is 2.5, then each additional (weighted) shared partner multiplies the odds of a tie by 2.5.

8.11 Summary and Key Takeaways

ERGMs provide a principled, statistically rigorous framework for asking whether network structure arises from:

- **Random chance** (the null/Bernoulli model),
- **Endogenous structural forces** like transitivity and degree-based effects (GWESP),
- **Exogenous nodal attributes** like wealth (nodecov, absdiff, nodematch), or
- **Some combination** of all of the above.

For the Florentine marriage network, the evidence points to a network shaped by both structural closure and wealth-based sorting — elite families in Renaissance Florence did not marry randomly, and the resulting network topology reflected deliberate social strategy.

The workflow introduced here — **Specification → Estimation → MCMC Diagnostics → Goodness of Fit** — is the standard pipeline for any applied ERGM analysis. Never skip diagnostics: a model that has not converged or that fails GoF checks should not be interpreted, regardless of how tidy the summary table looks.

8.12 Further Reading

- Lusher, Dean, Johan Koskinen, and Garry Robins. (2013). Exponential Random Graph Models for Social Networks : Theories, Methods, and Applications. Cambridge: Cambridge University Press.
- Rawlings, Craig M., Jeffrey A. Smith, James W. Moody, and Daniel McFarland. (2023) Network Analysis : Integrating Social Network Theory, Method, and Application with R. Cambridge: Cambridge University Press. https://inarwhal.github.io/NetworkAnalysisR-book/.
- Padgett, J. F., & Ansell, C. K. (1993). Robust action and the rise of the Medici. *American Journal of Sociology*, 98(6), 1259–1319.
- The `statnet` project: statnet.org — documentation, tutorials, and workshops.

9 Design Your Network Project

Synthesising theory and code to scope a research question

This session moves from learning to doing. You have encountered the core theory — how networks are defined, what structural properties they have, and how to measure them — and you have practised building and analysing networks in R. Now it is time to use those tools in service of a research question that you find genuinely interesting.

Working in small groups, your task is to **design a network-based research project**. You do not need to execute the full analysis today. What you do need to produce is a clear, defensible project design: a research question, a network definition, a data strategy, and a plan for analysis.

9.1 Step 1: Find a Network Idea

Every network project starts with a claim, however informal, that *relationships matter* for some outcome or phenomenon you care about. The relationships might be between people, organisations, countries, texts, genes, places — the node type matters less than the theoretical justification for why connections between them are consequential.

Start by asking: **in what domain do you think structure shapes outcomes?**

Some starting points to spark ideas

People and organisations

- Co-authorship among researchers in a field — does collaboration structure predict citation impact?
- Board interlocks between corporations — do shared directors transmit strategic practices?
- Friendship networks in a school — how does social position relate to academic performance or wellbeing?
- Communication networks within an organisation — where are the bottlenecks

and brokers?

Politics and policy

- Co-sponsorship of legislation in parliament — which MPs occupy central bridging roles?
- Alliance networks among countries — how do structural positions shape foreign policy choices?
- Policy networks connecting government agencies, NGOs, and think tanks — who controls the flow of ideas?

Culture and knowledge

- Citation networks in academic literature — which papers bridge otherwise disconnected subfields?
- Semantic networks derived from text — how do concepts cluster and evolve over time?
- Collaboration networks in creative industries (film, music) — does structural position predict career longevity?

Infrastructure and digital systems

- Hyperlink networks between websites — how does the web's structure shape information access?
- Transport networks — which nodes are critical for system resilience?
- Twitter/Mastodon follower networks — how do information cascades spread?

Biological and environmental

- Species interaction networks (food webs, pollination) — how does structural position affect extinction risk?
- Trade networks for specific commodities — which countries are structurally indispensable?

Once you have a general domain, narrow your focus to a specific, bounded population of actors and a specific type of relationship. A network is not an entire social world — it is a theoretically motivated slice of it.

9.2 Step 2: Define Your Network

Before you can analyse a network, you must define it precisely. Vague network definitions produce uninterpretable results. Work through each decision below and record your answers.

9.2.1 2.1 Nodes

Who or what are the nodes in your network?

Be specific. "People" is not enough. "MPs who served in the House of Commons between 2015 and 2024" is a network boundary. A clear boundary tells you who is *in* the network and, crucially, who is *excluded* — and that exclusion must be theoretically justified.

Decision	Your answer
What entity type are the nodes?	
What is the population?	
What time period does the network cover?	
How many nodes (approximate)?	

9.2.2 2.2 Edges

What is the relationship that connects nodes?

An edge is not just a vague association — it must be a *specific, observable interaction or relationship*. Consider:

- Is the relationship **directed** (A sends to B, but B does not send to A) or **undirected** (the relationship is symmetric)?
- Is it **weighted** (some ties are stronger than others) or **binary** (present or absent)?
- Is it **static** (measured at one point in time) or **dynamic** (evolving over time)?

Decision	Your answer
What is the relationship?	
Directed or undirected?	
Weighted or binary?	
If weighted, what does weight represent?	
Static or dynamic?	

9.2.3 2.3 Network Type

Based on your decisions above, classify your network. More than one may apply.

Type	Description	Does this apply?
Unipartite (one-mode)	All nodes are of the same type	
Bipartite (two-mode)	Nodes belong to two distinct sets (e.g., people and events)	
Multiplex	Multiple types of edges connect the same nodes	
Temporal / longitudinal	Network structure changes across time points	
Signed	Edges can be positive or negative (e.g., alliance vs. conflict)	

9.3 Step 3: Formulate a Research Question

A good network research question has three components: a **structural concept** (what network property you will measure), an **outcome** (what you expect that property to explain or predict), and a **theoretical mechanism** (why you expect the relationship to hold).

> **i** The anatomy of a network research question
>
> *"Does [structural property X] among [nodes] predict [outcome Y], via the mechanism of [theoretical logic Z]?"*
> **Example:** "Do researchers who occupy bridging positions (high betweenness centrality) across disciplinary clusters produce more cited work, because their brokerage role grants access to non-redundant information from multiple fields?"
>
> - **Structural property:** betweenness centrality / bridging position
> - **Outcome:** citation impact
> - **Mechanism:** Burt's structural holes argument — brokers gain informational advantages

Avoid questions that are purely descriptive ("what does the network look like?") unless description is genuinely needed before theory can be applied. The strongest questions connect network structure to something that matters beyond the network itself.

Draft your research question here and test it against the three-part criteria:

Component	Your answer
What structural property will you measure?	
What outcome or phenomenon does it help explain?	
What is the theoretical mechanism linking structure to outcome?	
Full research question (in one or two sentences):	

9.4 Step 4: Data Strategy

How will you get your data? There are three broad routes.

9.4.1 4.1 Use an Existing Dataset

Many high-quality network datasets are publicly available. Some useful repositories and sources:

- **ICON: Index of Complex Networks** — large index of networks across many domains
- **Stanford SNAP** — social, communication, and web graphs
- **KONECT** — network collection with standardised formats
- **UCINet datasets** — classic social network datasets

R packages with built-in network data

Several R packages ship with well-documented network datasets useful for learning and pilot analysis:

- `igraphdata` — karate club, yeast protein interactions, US airports, Facebook ego-networks
- `ergm` / `network` — Florentine families, Sampson's monastery, Padgett's factions
- `networkdata` — 979 network datasets covering a broad range of domains

9.4.2 4.2 Construct Data from an API or Digital Trace

Many platforms expose relational data via APIs:

- **OpenAlex / Semantic Scholar** — co-authorship and citation networks
- **GitHub REST API** — collaboration networks among developers
- **Hansard / parliamentary APIs** — voting and co-sponsorship records
- **Wikipedia API** — hyperlink structure between articles

9.4.3 4.3 Collect Primary Data

Surveys, interviews, or direct observation can generate network data where none exists. Classic instruments include:

- **Name generators** ("List up to five people you discuss important matters with")
- **Roster methods** (rate your relationship with each person on a fixed list)
- **Event participation logs** (who attended which meetings, co-authored which documents)

For a workshop project, primary data collection is usually impractical — but it is worth knowing what the gold standard looks like.

Record your data strategy:

Question	Your answer
Where will your data come from?	
In what format will it arrive (edge list, matrix, API JSON…)?	
What are the main limitations or biases of this source?	
What variables beyond the edges will you need (node attributes, etc.)?	

9.5 Step 5: Plan Your Analysis

Map your research question to specific network measures and analytical steps.

9.5.1 5.1 Descriptive Analysis (always a useful starting point)

- What is the size and density of your network?
- Is it connected? How many components?
- What does the degree distribution look like?
- Are there visible communities or clusters?

9.5.2 5.2 Node-Level Analysis

Which centrality measures are theoretically appropriate for your question?

Measure	When to use
Degree centrality	When raw connectivity (number of ties) is theoretically meaningful
Betweenness centrality	When brokerage and control over information flow matter
Closeness centrality	When speed of access to all other nodes matters
Eigenvector / PageRank	When being connected to well-connected others matters
Constraint / Brokerage	When structural holes and bridging positions are your focus

9.5.3 5.3 Global Network Properties

Property	When to use
Clustering coefficient	When local cohesion or transitivity is theoretically relevant
Average path length	When efficiency of information diffusion matters
Assortativity	When homophily or heterophily in tie formation is the focus
Community detection	When identifying subgroups or factions is the goal

9.5.4 5.4 Inferential Analysis (optional, but adds rigour)

If you have covered Chapter 7, consider whether a statistical model is appropriate:

- **ERGM** — model the probability of tie formation as a function of structural and nodal covariates
- **Permutation tests** — test whether an observed pattern (e.g., modularity, correlation) exceeds chance
- **QAP regression** — regression where both predictor and outcome are network matrices

Sketch your analytical plan:

Analysis step	Method	Addresses which part of your research question?
1.		
2.		
3.		
4. (optional)		

9.6 Step 6: Group Presentation

Each group will present their project design in **five minutes**, covering:

1. **Research question** — one or two sentences, clearly stated
2. **Network definition** — nodes, edges, directionality, any weights
3. **Data source** — where the data comes from and key limitations
4. **Analytical plan** — which measures or models you will use and why
5. **Expected finding** — what result would confirm or challenge your theoretical argument?

You are not expected to have results. You are expected to have a coherent, theoretically grounded design that could plausibly be executed.

> **!** What makes a strong project design?
>
> - The research question connects **network structure** to an **outcome that matters**
> - The network boundary (who is in, who is out) is **explicitly defined and justified**
> - The choice of analytical methods follows from the **theoretical mechanism**, not convenience
> - Limitations of the data are **acknowledged**, not ignored

Feedback

How to Cite

You can cite this tutorial book as:

Jarnach, Clemens. 2026. Social Network Analysis Workshop: A Practical Introduction from Theory to Code. Oxford. https://github.com/clemensjarnach/social_network_analysis_workshop

References

Avrachenkov, Konstantin, and Maximilien Dreveton. 2022. *Statistical Analysis of Networks*. NowOpen. Now Publishers.

Borgatti, Stephen P., Martin G. Everett, and Jeffrey C. Johnson. 2013. *Analyzing Social Networks*. Los Angeles: SAGE.

Borgatti, Stephen P., Martin G. Everett, Jeffrey C. Johnson, and Filip Agneessens. 2022. *Analyzing Social Networks Using R*. Los Angeles London New Delhi Singapore Washington DC Melbourne: SAGE.

Everett, M. G., and S. P. Borgatti. 2013. "The Dual-Projection Approach for Two-Mode Networks." *Social Networks* 35 (2): 204–10.

Khokhar, Devangana. 2015. *Gephi Cookbook: Over 90 Hands-on Recipes to Master the Art of Network Analysis and Visualization with Gephi*. 1st ed. Erscheinungsort nicht ermittelbar: Packt Publishing.

Luke, Douglas. 2015. *A User's Guide to Network Analysis in R*. Use R! Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-23883-8.

Lusher, Dean, Johan Koskinen, and Garry Robins. 2013. *Exponential Random Graph Models for Social Networks : Theories, Methods, and Applications*. Structural Analysis in the Social Sciences ; 35. Cambridge: Cambridge University Press.

McLevey, John, John Scott, and Peter J. Carrington. 2024. *The Sage Handbook of Social Network Analysis*. Second edition. London ; Sage.

Moreno, J. L. 1934. *Who Shall Survive?: A New Approach to the Problem of Human Interrelations.* Washington: Nervous and Mental Disease Publishing Co. https://doi.org/10.1037/10648-000.

———. 1951. *Sociometry, Experimental Method and the Science of Society; an Approach to a New Political Orientation.* Beacon, N.Y: Beacon House.

Rawlings, Craig M., Jeffrey A. Smith, James W. Moody, and Daniel McFarland. 2023. *Network Analysis : Integrating Social Network Theory, Method, and Application with R*. Structural Analysis in the Social Sciences. Cambridge: Cambridge University Press.

Wasserman, Stanley, and Katherine Faust. 1997. *Social Network Analysis : Methods and Applications*. Reprint with corrections. Structural Analysis in the Social Sciences ; 8. New York: Cambridge University Press.