

Social networks

Agent-based modelling, Konstanz, 2024

Henri Kauhanen

11 June 2024

i Update 25 June 2024

As some of you cleverly pointed out, there is actually a direct link from *Switzerland* to *Albert Einstein* on Wikipedia, rendering my six degrees of separation exercise somewhat vacuous...

To better get the desired effect, try navigating between two pages which are less obviously related, such as *Easter Island* and *Albert Einstein*.

Also: I've fixed a typo: "each node's side" → "each node's size".

Plan

- In our models so far, agent interactions have been either
 1. Completely random
 2. Random, but within a local spatial neighbourhood
- Today, we will take a step towards generalizing our models by assuming that agents are connected through a social network
- We will require *Graphs.jl* and *GraphPlot.jl*:

```
using Pkg
Pkg.add(["Graphs", "GraphPlot"])
```

Networks

- A **network** (also known as a **graph**) consists of
 1. A set of **nodes** (a.k.a. **vertices** (singular: *vertex*)) – for us, these are the agents

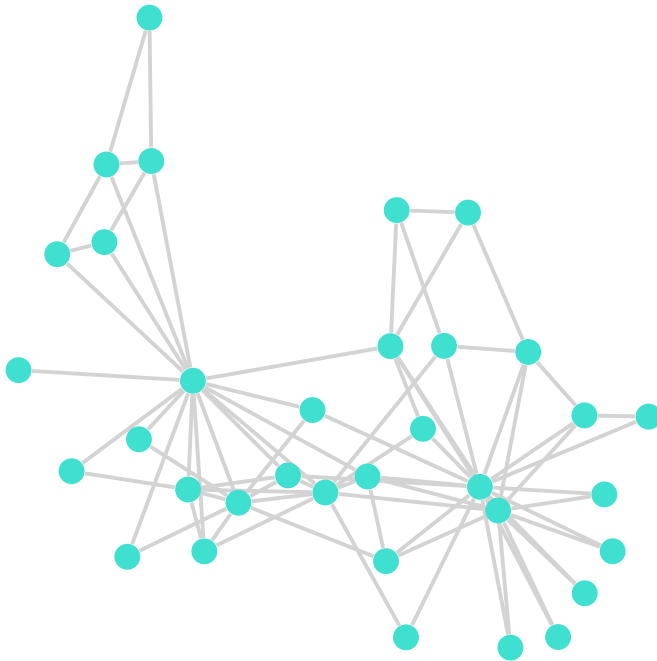
2. A set of **connections** (a.k.a. **links** or **edges**) between nodes – for us, these define the interaction pattern

- Connections can be
 1. Unidirectional (A is connected to B, but B is not connected to A)
 2. Bidirectional (A is connected to B and B is also connected to A)
 3. Weighted or not

Drawing networks

- Nodes typically drawn as points / filled circles
- A unidirectional connection is drawn as an arrow
- A bidirectional connection is drawn as a line segment
- If connections are weighted, this can be represented e.g. by line width

Example (**Zachary's karate club**)



Networks in Julia

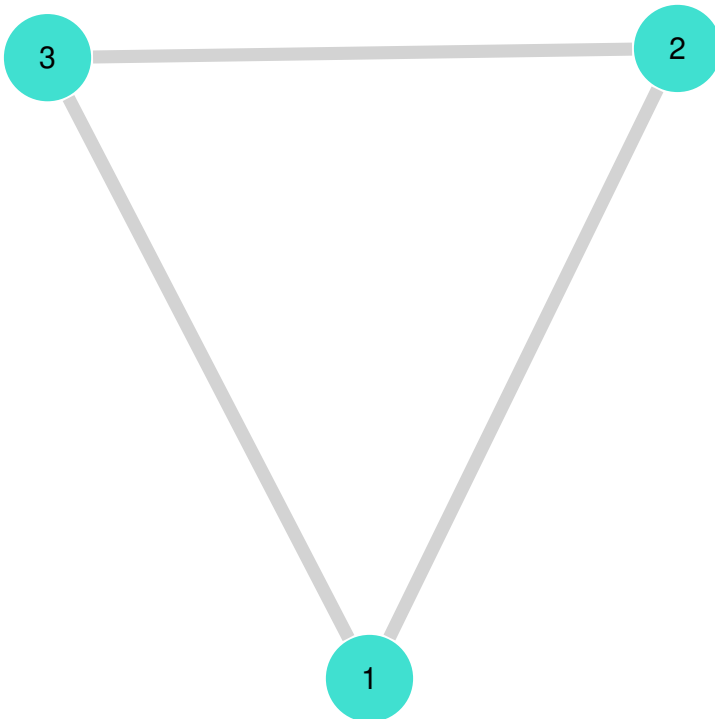
- In Julia, networks/graphs are handled by the [Graphs.jl](#) package
- Simple example: create an undirected graph of three nodes, connecting each node:

```
using Graphs, GraphPlot  
  
G = Graph(3)  
  
add_edge!(G, 1, 2)  
add_edge!(G, 1, 3)  
add_edge!(G, 2, 3)
```

Plotting networks

- Networks can be plotted with the `gplot` function from *GraphPlot.jl*:

```
gplot(G, nodelabel=1:3)
```



! Important

The `gplot` function in fact returns something known as a “composition”; this may or may not be actually drawn as a picture, depending on your working environment. If you’re in VSCode, a picture will be displayed, but if you’re using the ordinary Julia REPL, you will not see a picture. In the latter case, you need to save the composition to an image file as follows:

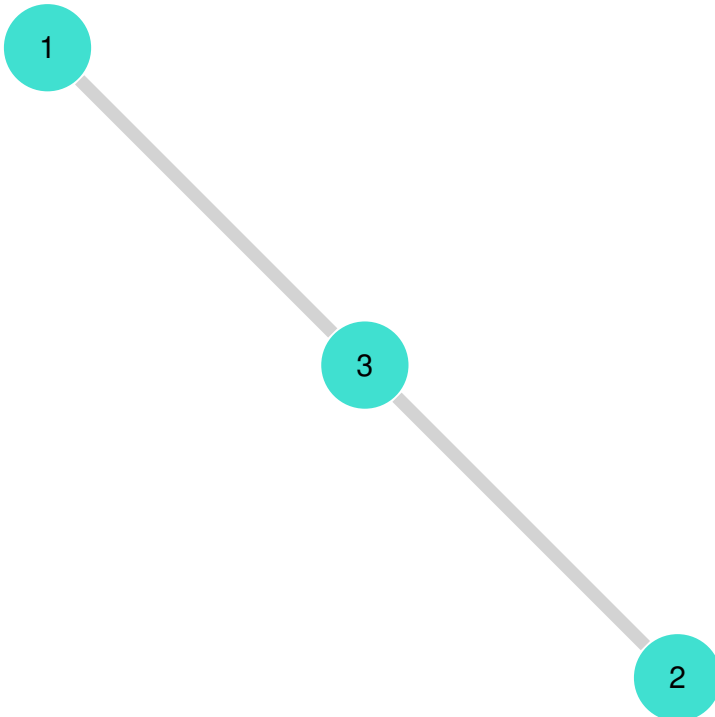
```
using Compose

pic = gplot(G, nodelabel=1:3)
draw(PNG("mypicture.png"), pic)
```

Removing connections

- `rem_edge!` can be used to remove existing connections:

```
rem_edge!(G, 1, 2)
gplot(G, nodelabel=1:3)
```



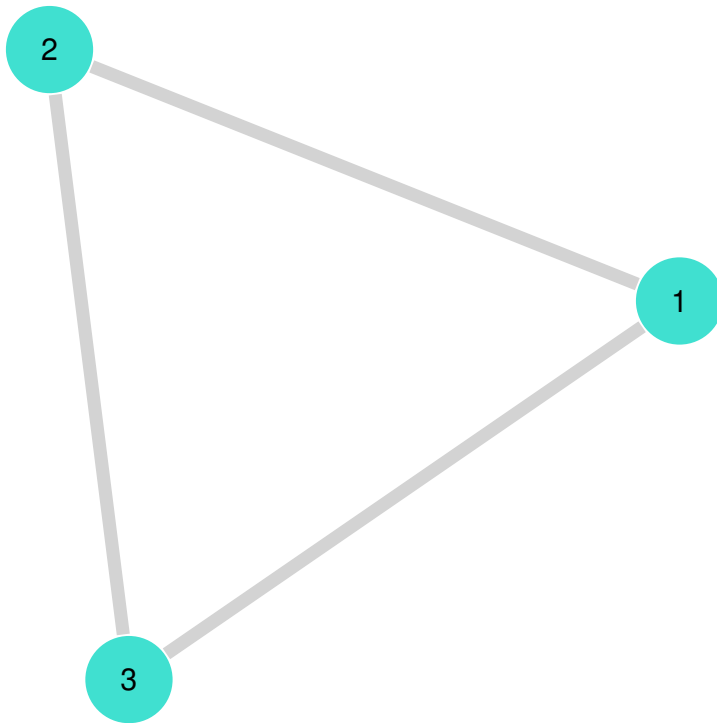
Constructing a network from an adjacency matrix

- Using lots of `add_edge!` calls quickly becomes tedious...
- Often, a better way of constructing a graph is by way of its **adjacency matrix**
- This is a matrix (two-dimensional array) of numbers such that:
 - if there is a 1 in the cell on the i th row, j th column, then nodes i and j are connected
 - if there is a 0 there, the nodes are not connected
- Example:

```
A = [0 1 1  
     1 0 1  
     1 1 0]
```

```
G2 = Graph(A)
```

```
gplot(G2, nodelabel=1:3)
```



Directed graphs

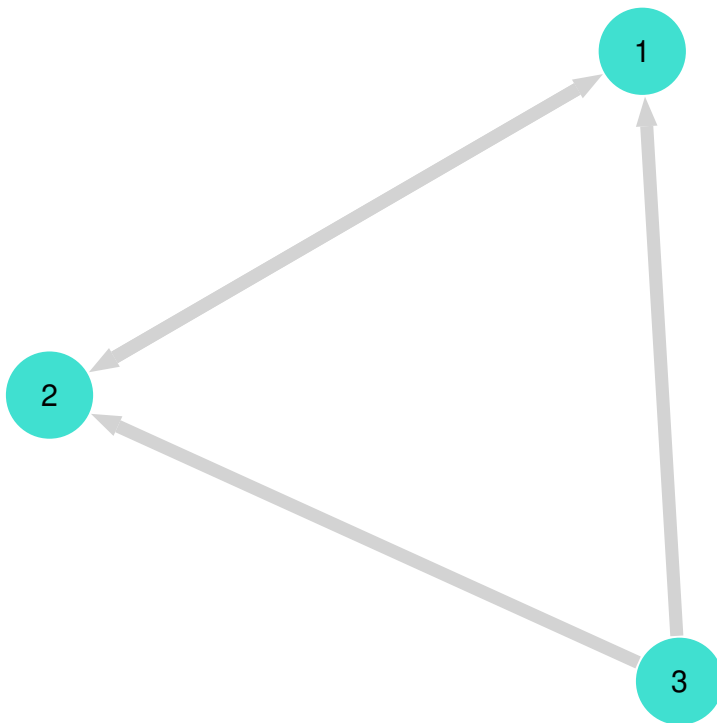
- What if we need a network in which at least some connections are unidirectional?
- We need a **directed graph**, implemented by the `DiGraph` type

```
A = [0 1 0  
     1 0 0  
     1 1 0]
```

```
G3 = DiGraph(A)
```

{3, 4} directed simple Int64 graph

```
gplot(G3, nodelabel=1:3)
```



! Important

Notice that, since `Graph` constructs an undirected graph, it expects a **symmetric** adjacency matrix as argument. If you try to pass an asymmetric adjacency matrix (such as

the one above) to `Graph`, you will get an error.

In other words, if your adjacency matrix is asymmetric, you are dealing with a directed graph, and you *must* use `DiGraph`.

Exercise

What kind of network do the following adjacency matrices represent? **Think about it first** (draw with your “mind’s eye”), then implement the code and plot the graphs.

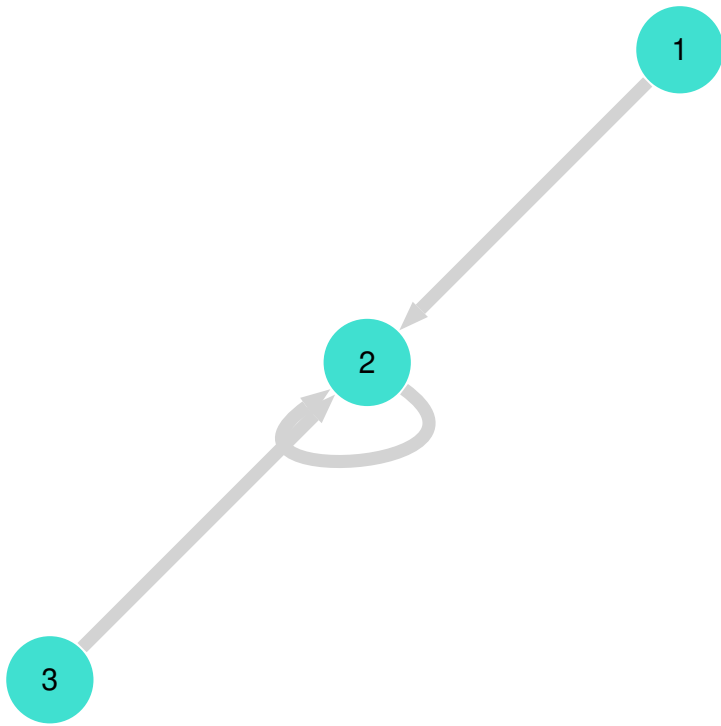
```
A = [0 1 0
      0 1 0
      0 1 0]
```

```
B = [0 0 0
      1 1 1
      0 0 0]
```

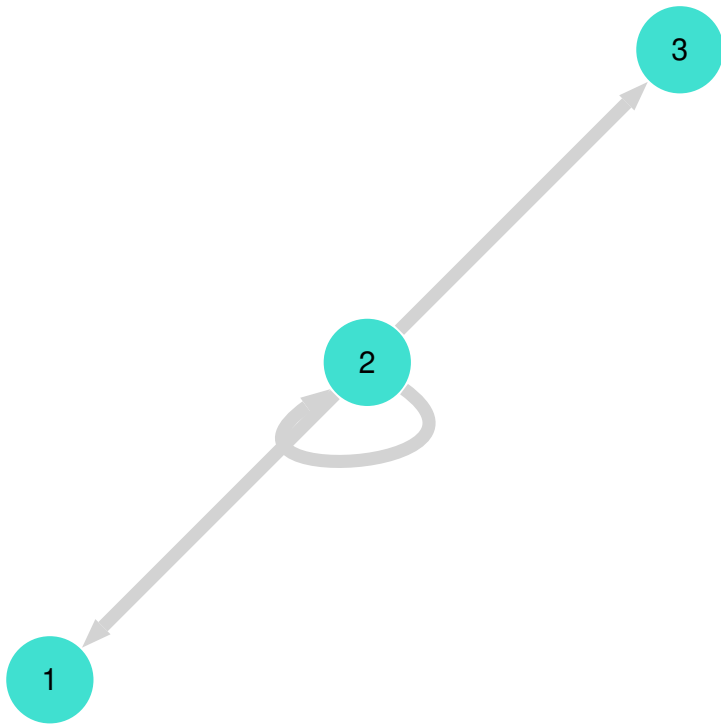
```
C = [1 0 0
      0 1 0
      0 0 1]
```

Solution

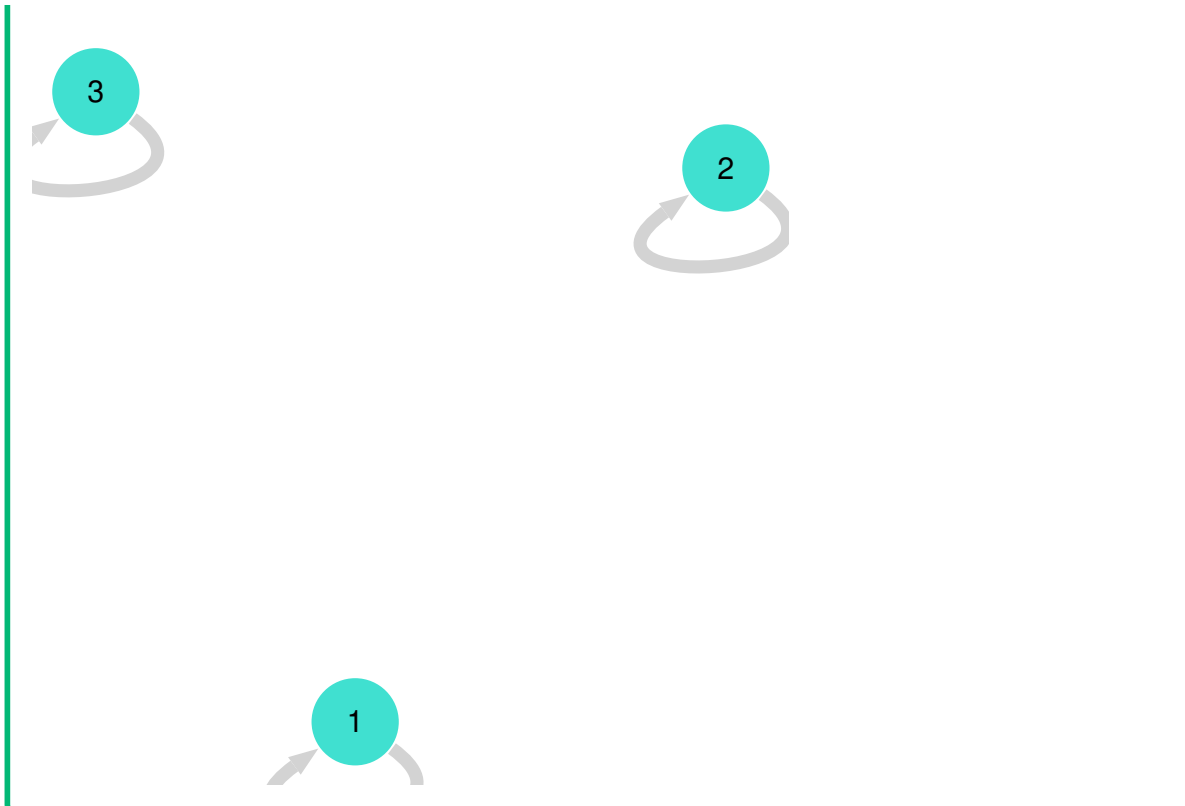
```
GA = DiGraph(A)
gplot(GA, nodelabel=1:3)
```



```
GB = DiGraph(B)  
gplot(GB, nodelabel=1:3)
```

```
GC = DiGraph(C)  
gplot(GC, nodelabel=1:3)
```



Graph generators

- For large networks, it would be tedious to construct them by hand...
- And in any case, we are rarely interested in the **precise** construction of a network
- What's more important are the **statistical characteristics** of the network
 - How many connections does a node have on average?
 - Are some nodes much more connected than others?
 - And so on.
- Large graphs with known statistical properties can be constructed using **generators**

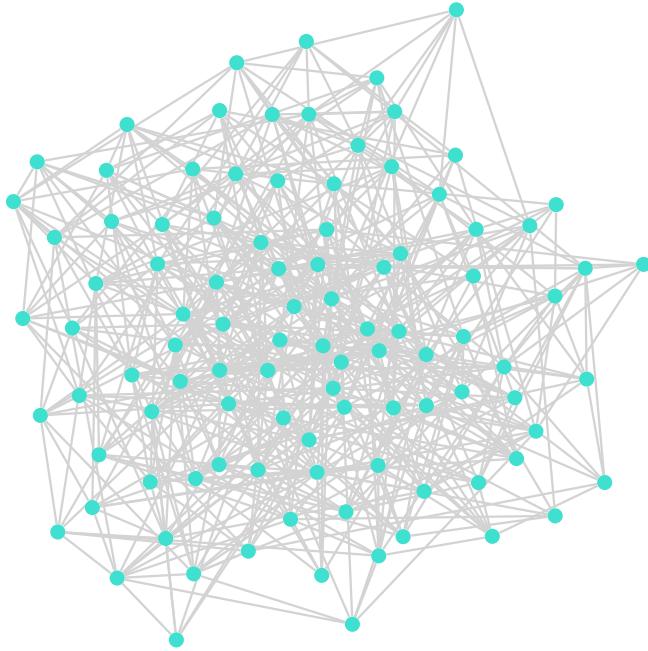
Erdős-Rényi graphs

- Take n nodes, initially unconnected. Cycle through each pair of nodes, connecting them with probability p .
 - In other words: At each node pair, you flip a biased coin that lands heads with prob. p and tails with prob. $1 - p$. If you get heads, you connect the nodes; if tails, you don't.

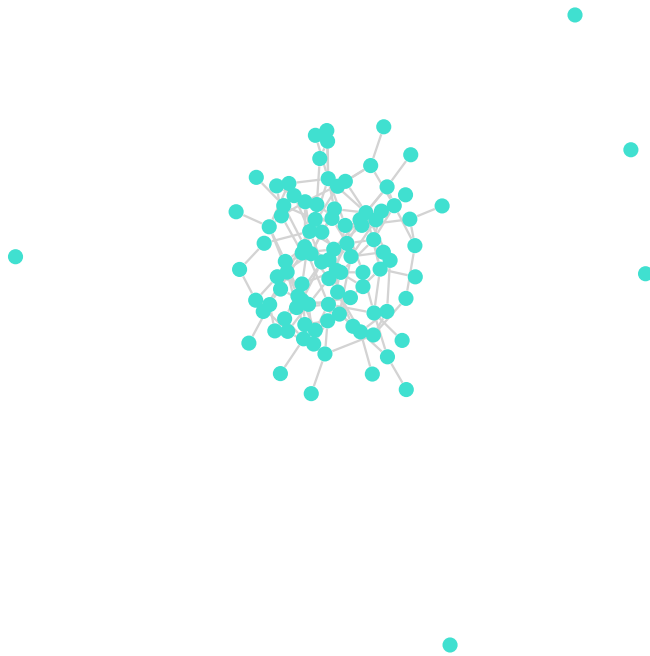
- This algorithm results in a so-called **Erdős-Rényi random graph**.
- In Graphs.jl, the `erdos_renyi` function can be used:

```
GER = erdos_renyi(100, 0.1)  
GER2 = erdos_renyi(100, 0.03)
```

```
gplot(GER)
```



```
gplot(GER2)
```



Accessing graph properties

- Number of nodes (vertices):

```
nv(GER)
```

```
100
```

```
nv(GER2)
```

```
100
```

- Number of connections (edges):

```
ne(GER)
```

```
548
```

```
ne(GER2)
```

161

- Number of connections for each node (called the node's **degree**):

```
degree(GER)
```

```
100-element Vector{Int64}:
```

11

12

9

10

18

12

9

10

10

7

11

12

12

8

7

10

14

9

6

13

13

9

9

9

12

```
degree(GER2)
```

```
100-element Vector{Int64}:
```

1

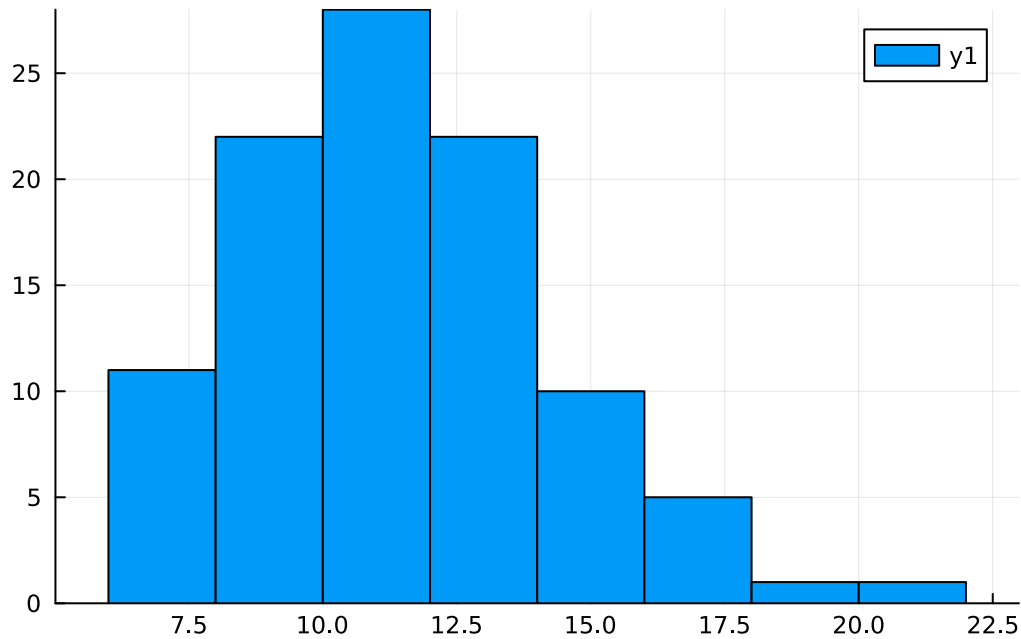
1
3
0
2
1
3
7
6
2
5
5
3

1
2
3
5
3
4
5
3
7
3
1
5

Degree distribution

- This makes it easy to plot a graph's **degree distribution**:

```
using Plots          # for the histogram() function
d = degree(GER)
histogram(d)
```



Exercise

Open Wikipedia on the page for *Switzerland*. Then, **using only links on the page**, try to navigate to the page for *Albert Einstein*.

How many links do you need to go through to reach the destination?

💡 Answer

Here's one possible path:

Switzerland

→ *German*

→ *Germany*

→ *Education in Germany*

→ *Max Planck Institute for Plasma Physics*

→ *Physics*

→ *Albert Einstein*

That's 6 links.

But it's not the shortest path. You can also go, if you're in the know:

Switzerland

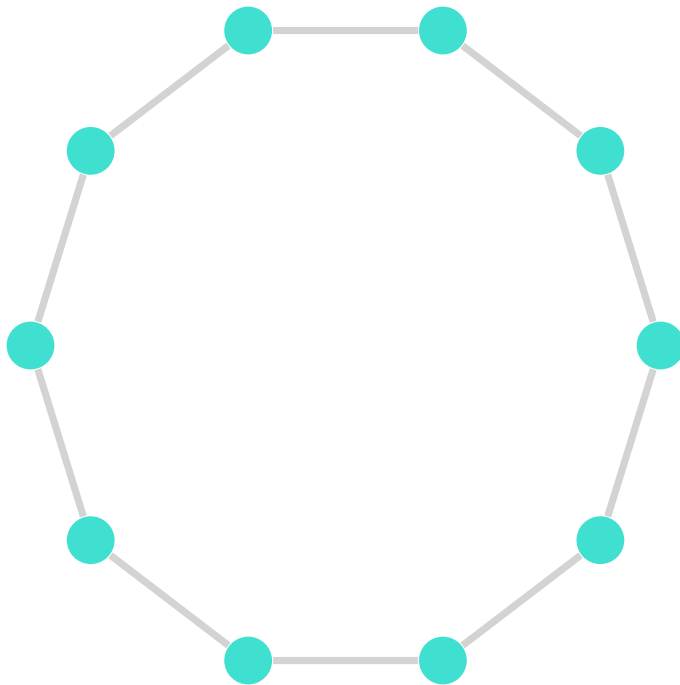
→ *University of Zurich*

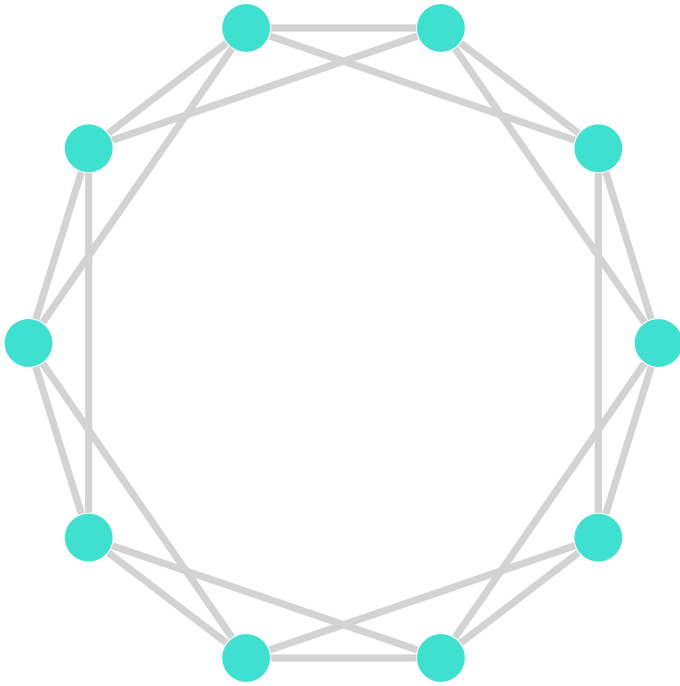
→ *Albert Einstein*

(2 links.)

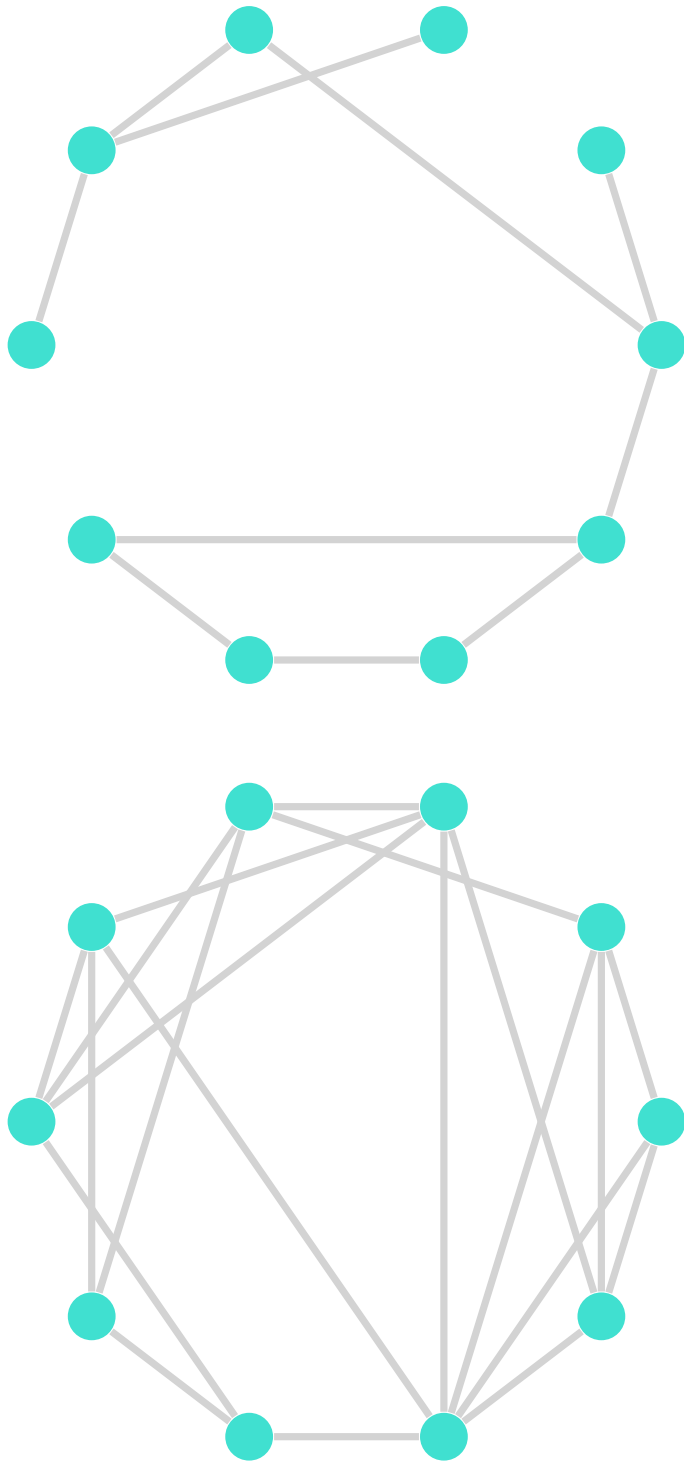
“Six degrees of separation”

- Some networks have the “small-world property”: there is a short path from every node to every other node
 - The **Watts-Strogatz model** or **small world graph** is one way of modelling this
 - To obtain such a graph, one does a random rewiring of a ring lattice
1. Start with a ring lattice (on the left, each node has $k = 2$ neighbours; on the right, each node has $k = 4$ neighbours):





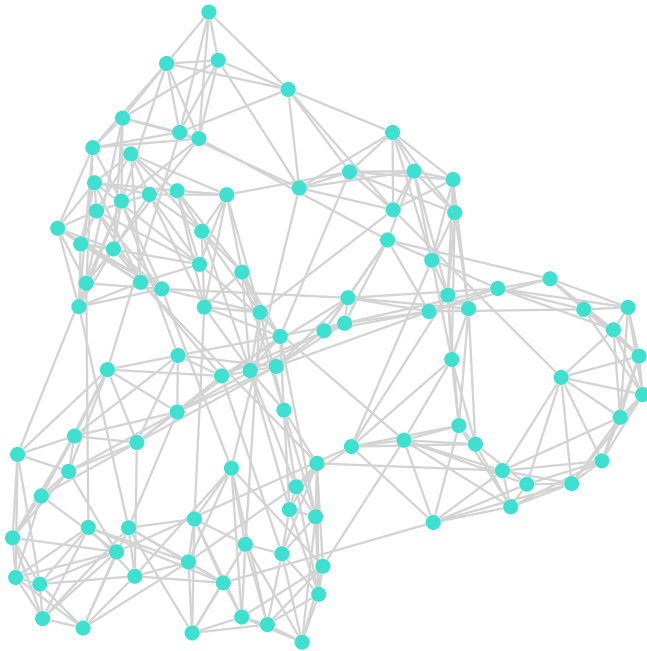
2. Randomly rewired each edge with probability β to a randomly chosen destination (here, $\beta = 0.4$):



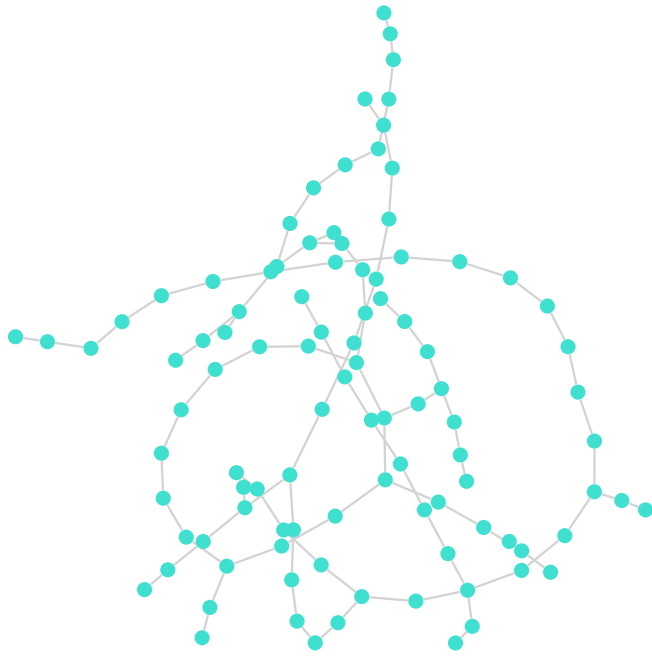
- In Graphs.jl, small-world networks can be created with `watts_strogatz(n, k, p)`
 - `n`: number of nodes
 - `k`: initial degree of every node
 - `p`: rewiring probability
- For example:

```
GWS1 = watts_strogatz(100, 8, 0.1)
GWS2 = watts_strogatz(100, 2, 0.1)
```

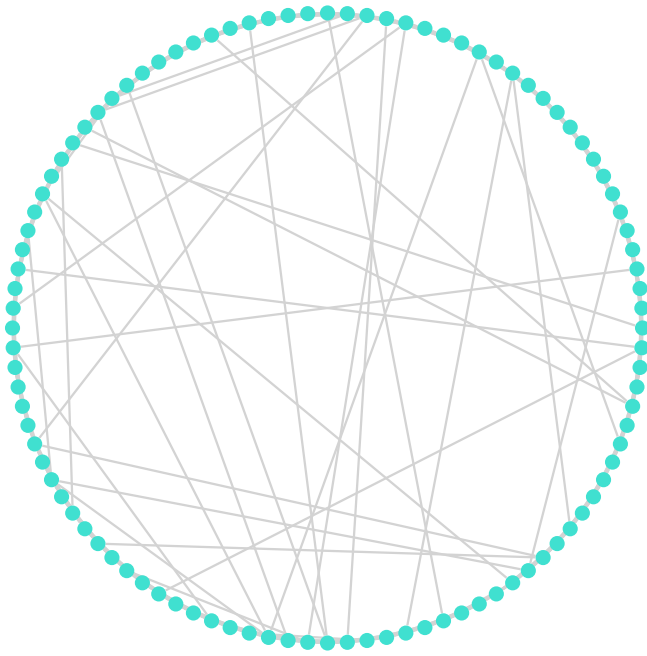
```
gplot(GWS1)
```



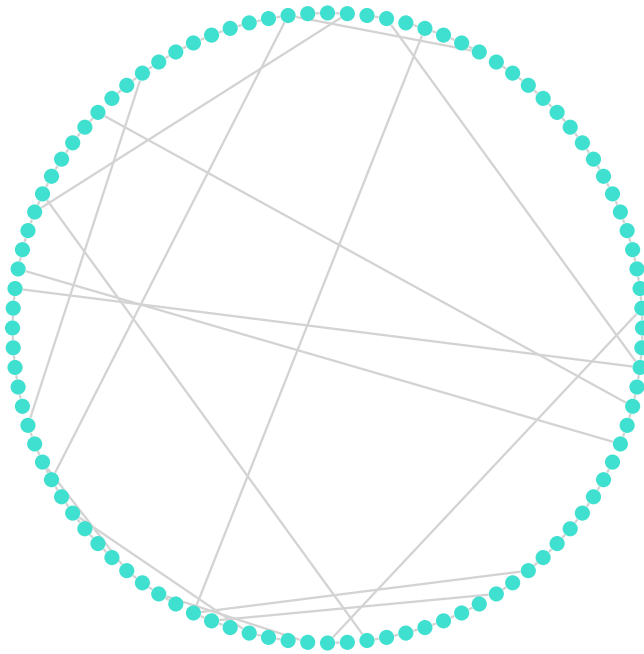
```
gplot(GWS2)
```



```
gplot(GWS1, layout=circular_layout)
```



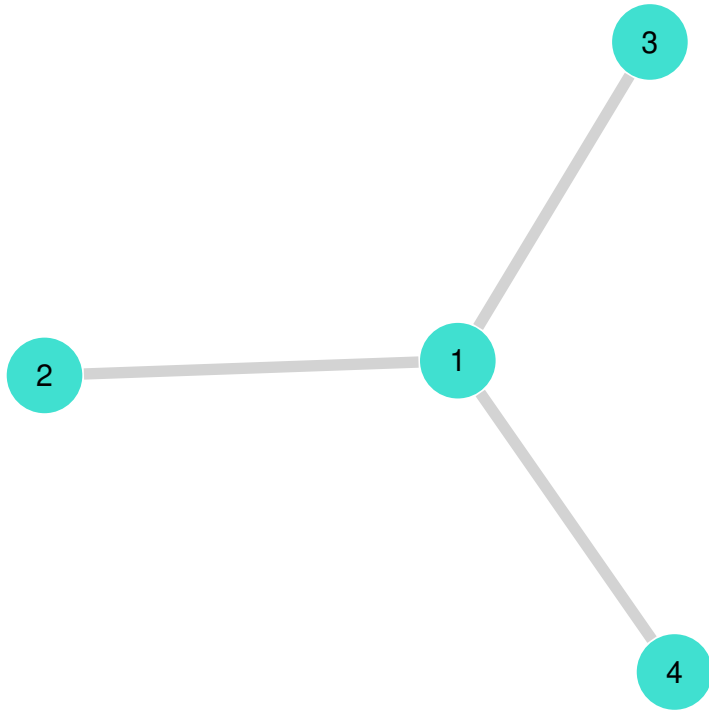
```
gplot(GWS2, layout=circular_layout)
```



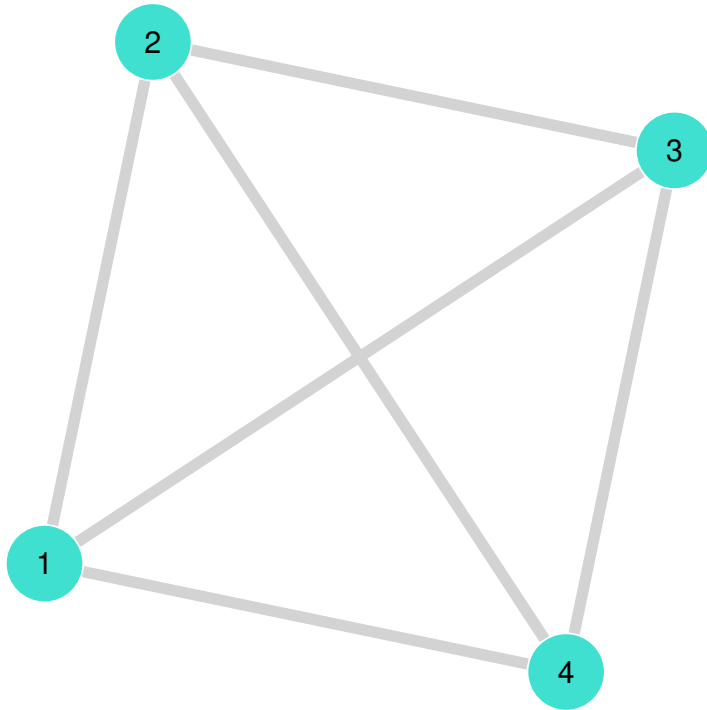
Clustering

- Short path lengths are not the only way in which small-world networks are interesting
- They also exhibit high clustering
- The **local clustering coefficient** of a node v is defined as the proportion of neighbours of v which are neighbours amongst themselves
- A highly clustered network contains **cliques**, subnetworks in which all nodes are connected to each other

Low clustering for node 1:



High clustering for node 1:



- In Graphs.jl, we can use `local_clustering_coefficient`
- E.g. to get the average local clustering coefficient:

```
using Statistics
G = erdos_renyi(100, 0.1)
mean([local_clustering_coefficient(G, v) for v in vertices(G)])
```

0.09162485991820357

```
using Statistics
G = watts_strogatz(100, 10, 0.1)
mean([local_clustering_coefficient(G, v) for v in vertices(G)])
```

0.5072842157842158

Exercise

1. Download and unzip the dolphin social network data from <https://networkrepository.com/soc-dolphins.php>

2. Import these data into Julia, construct a graph, and plot the network
3. Modify the plot so that each node's size is proportional to its degree
4. Plot a histogram of the degree distribution
5. Plot a histogram of the distribution of local clustering coefficient

You will need [MatrixMarket.jl](#) and [GraphPlot.jl documentation](#)

Going forward

- Next time, we will learn how to interface Graph.jl with Agents.jl, so that we can run ABM simulations on networks
- Homework:
 1. Read Smaldino (2023), chapter 9
 2. Complete the [homework assignment](#)

Smaldino, Paul E. 2023. *Modeling Social Behavior: Mathematical and Agent-Based Models of Social Dynamics and Cultural Evolution*. Princeton, NJ: Princeton University Press.