SICSS: Social Network Analysis

Dr. Oliver Posegga

7/31/2019

Contents

R Basics

Analyzing networks with igraph requires a basic understanding of the fundamental data structures, operations, and patterns commonly used in R. In the following, we are going to recapitulate the basics.

Data types and structures

R offers a variety of data structures that can be used store and manipulate data of different types. Vectors, lists, factors, matrices, and data frames are the most common structures.

Data types

The six basic data types offered by R are: character, numeric (real or decimal), integer, logical, and complex.

```r
c <- "Hello world!"   # Character
n <- 5.5              # Numeric
i <- 42               # Integer
l <- TRUE             # Logical
x <- 1+4i             # Complex
```

The following functions can be used to inspect and determine the properties of R objects:

```r
class(c)      # Returns the class of the object (high-level)
```

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

```r
typeof(c)     # Returns the data type of the object (low level)
```

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

```r
length(c)     # Returns the lenght of the object
```

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

```r
attributes(c) # Returns meta data attached to the object
```

```
## NULL
```

Objects can be cast from one data type to another using the functions: `as.character`, `as.numeric`, `as.integer`, etc.

```r
typeof(i)             # The integer stored above is actually treated as a double
```

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

```r
i <- as.integer(i)    # Let's fix that
typeof(i)
```

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

Vectors

Vectors are one-dimensional collections of elements that are often of one of the basic data types mentioned above. Elements of an atomic vector have to be of a single data type. If values of different data types are provided, R automatically adjusts the data type.

```r
v1 <- c("Hello", "world", "!")
v2 <- c(1.1, 2.2, 3.3, 4.4, 5.5)
v3 <- c(1, 2, 3, 4, 5)
v4 <- c(TRUE, FALSE)
```

Some R functions can be used to generate vectors that contain specific sequences of values:

```r
v5 <- 1:5             # Sequence from 1 to 5
v5 <- seq(1, 10, 2)   # Sequence of numbers from 1 to 10 in steps of 2
v5 <- rep(1, 10)      # Sequence of 1, repeated 10 times
v5
```

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

Let's inspect some of the properties of the last example:

```r
class(v5)   # The values belong to the numeric class
```

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

```r
typeof(v5)  # More specifically, the rep function returned doubles
```

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

```r
length(v5)  # The vecor contains 10 elements
```

```
## [1] 10
```

Individual elements of vectors can be accessed using an index:

```r
v <- c("one", "two", "three", "four", "five")
v[1]              # Gets the frist element
```

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

```r
v[1:3]            # Gets elements on position 1 to three
```

```
## [1] "one"   "two"   "three"
```

```r
v[-3]             # Returns all elements except the third
```

```
## [1] "one"  "two"  "four" "five"
```

Using an index and a logical comparison, elements can also be selected based on logical conditions:

```r
v == "three"      # Returns a logical vector that is only true for element 3
```

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

```r
v[v == "three"]   # Masks the vector with the logical vector and extracts element 3
```

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

Adding elements:

```r
v <- c(v, "six")  # Creates a new vector from v and the element "six"
v
```

```
## [1] "one"   "two"   "three" "four"  "five"  "six"
```

Factors

Factors are very similar to vectors and can be used to store categorical data. The function `factor()` identifies all distinct values in a vector and assigns each unique value to a factor level. In the example below, the vector v has five elements, but only three distinct values, which become the three factor levels.

```r
v <- c("one", "one", "two", "two", "three")    # Vector
v
```

```
## [1] "one"   "one"   "two"   "two"   "three"
```

```r
f <- factor(c(v))                          # Factor
f
```

```
## [1] one   one   two   two   three
## Levels: one three two
```

Using the factor, we can now work with the categorial data it represents:

```r
levels(f)               # Access the factor levels
```

```
## [1] "one"   "three" "two"
```

```r
as.numeric(f)           # Numeric representation of the factor
```

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

```r
as.numeric(v)           # Won't work with the original vector!
```

```
## Warning: NAs introduced by coercion
```

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

Matrices

Matrices are extensions of vectors and, technically, not a separate data type. They are vectors with dimensions and can be created from vectors by assigning them dimensions. There are several other methods of creating matrices, e.g. by using the `matrix()` function.

```r
v <- 1:16         # Creates a one-dimensional vector 1..16
v
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

```r
dim(v) <- c(4,4)  # Sets the dimensions of the vector to 4 rows and 4 columns
v
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

We can create the same matrix using the `matrix()` function:

```r
m <- matrix(data=1:16, nrow=4, ncol=4)   # Same as above
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

We can inspect the dimensions of a matrix using the `dim()` function:

```r
dim(m)
```

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

Elements of a matrix can be selected using a two-dimensional index:

```r
m[1,2]      # Select cell in row 1, column 2
```

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

```r
m[1,]       # Select row 1
```

```
## [1]  1  5  9 13
```

```r
m[,2]       # Select column 2
```

```
## [1] 5 6 7 8
```

```r
m[-4,]      # Submatrix without row 4
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
```

```r
m[1:3, 1:3] # Submatrix from row 1 to 3 and column 1 to 3
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
```

```r
m[m>6]      # Select only elements greater than 6
```

```
##  [1]  7  8  9 10 11 12 13 14 15 16
```

Some important matrix operations:

```r
m * m     # Element-wise multiplication
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   25   81  169
## [2,]    4   36  100  196
## [3,]    9   49  121  225
## [4,]   16   64  144  256
```

```r
m %*% m   # Matrix multiplication
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   90  202  314  426
## [2,]  100  228  356  484
## [3,]  110  254  398  542
## [4,]  120  280  440  600
```

```r
t(m)      # Transpose matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

Lists

Lists are containers that can contain elements of different data types. In contrast to atomic vectors, which can only contain elements of a single data type, lists are therefore referred to as generic vectors.

```r
v1 <- c(1, 2, 3)                      # Numeric vector
v2 <- c("a", "b", "c")                # Character vector
l <- list(1, 2.0, TRUE, "foo", v1, v2)  # List
l
```

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

```
## [1] 2
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] "foo"
##
## [[5]]
## [1] 1 2 3
##
## [[6]]
## [1] "a" "b" "c"
```

List elements can be accessed by their position in the list. When using the single bracket notation as before, the list will return a list of all elements at the selected position. A double bracket notation allows the extraction of the elements themselves.

```
l[5]    # Returns a list containing the element at position 5 (v1)
```

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

```
l[[5]]  # Returns the element at position 5 (v1)
```

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

Another interesting property of lists is that elements can be named.

```
ln <- list(foo=v1)    # Creates a list with v1 as a named element "foo"
ln$bar <- v2          # Adds v2 as named element "bar"
ln
```

```
## $foo
## [1] 1 2 3
##
## $bar
## [1] "a" "b" "c"
```

```
ln["foo"]             # Returns a list containing v1
```

```
## $foo
## [1] 1 2 3
```

```
ln[["foo"]]           # Returns v1
```

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

```
ln$foo                # Returns v1
```

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

Data Frames

Data frames are one of the most important and most frequently used data structures in R. Think of them as as tables with rows and columns that can be named and store multiple data types. In essence, they are lists of vectors (or factors) of the same length, each vector representing a separate column.

```r
df <- data.frame(
  id = 1:5,
  name = c("Andreas", "Martin", "Oliver", "Robert", "Dan"),
  semester = c(12, 6, 6, 7, 2),
  graduated = c(FALSE, TRUE, TRUE, TRUE, FALSE)
  )
df
```

```
##   id    name semester graduated
## 1  1 Andreas       12     FALSE
## 2  2  Martin        6      TRUE
## 3  3  Oliver        6      TRUE
## 4  4  Robert        7      TRUE
## 5  5     Dan        2     FALSE
```

Similar to lists, named dimensions of data frames can be accessed by their name. Otherwise, elements of data frames can be selected in the same way we already discussed for matrices.

```r
df$name       # Returns the name column
```

```
## [1] Andreas Martin  Oliver  Robert  Dan
## Levels: Andreas Dan Martin Oliver Robert
```

```r
df[["name"]]  # Returns the name column
```

```
## [1] Andreas Martin  Oliver  Robert  Dan
## Levels: Andreas Dan Martin Oliver Robert
```

```r
df["name"]    # Returns a data frame containing the node column
```

```
##      name
## 1 Andreas
## 2  Martin
## 3  Oliver
## 4  Robert
## 5     Dan
```

```r
df[1,]        # Returns a data frame containing the first row
```

```
##   id    name semester graduated
## 1  1 Andreas       12     FALSE
```

```r
df[1:3,]      # Returns a data frame containing rows 1 to 3
```

```
##   id    name semester graduated
## 1  1 Andreas       12     FALSE
## 2  2  Martin        6      TRUE
## 3  3  Oliver        6      TRUE
```

```r
df[df$graduated == TRUE,]  # Returns a data frame with rows where graudated is TRUE
```

```
##   id   name semester graduated
## 2  2 Martin        6      TRUE
## 3  3 Oliver        6      TRUE
## 4  4 Robert        7      TRUE
```

A couple of helpful libraries facilitate working with data frames. Based on the `tidyverse` libraries, which include `dplyr`, we can select, filter, group, and aggregate contents of data frames using chained arguments. The following is just a glimpse into the features of dplyr:

```r
df %>% select(name, graduated)    # Selects the columns name and graduated
```

```
##      name graduated
## 1 Andreas     FALSE
## 2  Martin      TRUE
## 3  Oliver      TRUE
## 4  Robert      TRUE
## 5     Dan     FALSE
```

```r
df %>% filter(graduated == TRUE)  # Filters the rows based on the graduated column
```

```
##   id   name semester graduated
## 1  2 Martin        6      TRUE
## 2  3 Oliver        6      TRUE
## 3  4 Robert        7      TRUE
```

```r
# Both operations chained
df %>%
  select(name, graduated) %>%
  filter(graduated == TRUE)
```

```
##     name graduated
## 1 Martin      TRUE
## 2 Oliver      TRUE
## 3 Robert      TRUE
```

```r
# Groups by graduated and computes the mean per group
df %>%
  group_by(graduated) %>%
  summarize(mean_semester = mean(semester))
```

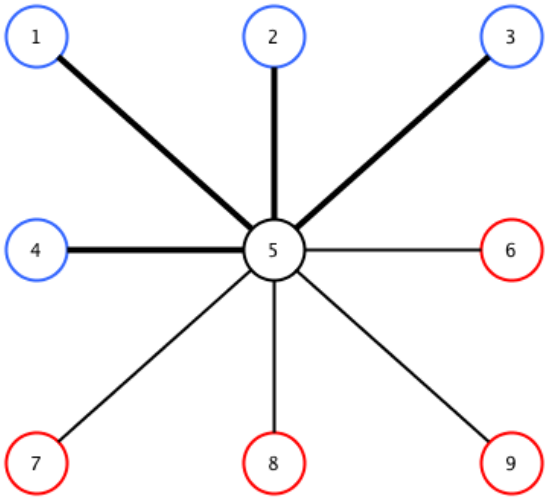
```
## # A tibble: 2 x 2
##   graduated mean_semester
##   <lgl>              <dbl>
## 1 FALSE               7
## 2 TRUE                6.33
```

Introduction to igraph

When it comes to analyzing relational data and networks in R, there are a couple of valid libraries that offer comprehensive imeplentations of the most commonly used methods and measures used in social network analysis (e.g. `statnet` or `tidygraph`), and some of those offer solutions vor very specific problems (e.g. `rsiena`). Here, we focus on analyzing networks using `igraph`, which provides a wide range of efficient data structures and algorithms. One of the reasons why igraph is a good starting point for network analysis in R lies within its availability in several other languages, most notably Python.

Example

For the next few steps, we will try to recreate the following network, which comprises nine nodes. The nodes are coloured according to a node attribute. The edges connecting them have a weight; one half of the edges is stronger than the other half. The edges are undirected and all nodes are connected to node five, but not to each other.



Setup

Beofre we start, if we haven't already done so, we need to install the igraph package using the command `install.packages("igraph")`. Once it's installed, we can simply load the library. It's also a good idea to clean the objects in our memory to get rid of the objects created earlier.

Creating networks

First, we need to create a network object. There are multiple ways to create networks using igraph, e.g. by fully specifying the number of nodes and a list of all edges, by only providing a list of edges, or a matrix. We can also generate networks from a variety of models implented in igraph (e.g. small-world networks, scale-free networks, and random networks).

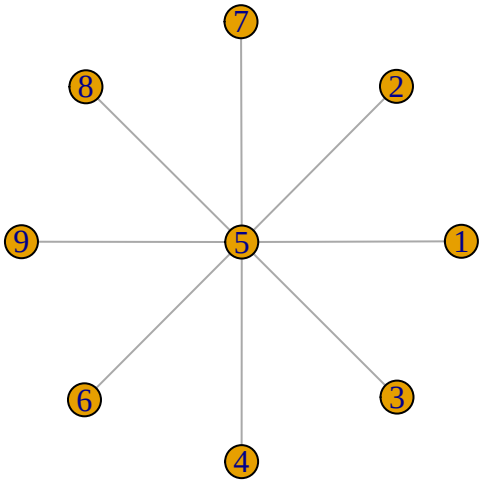
```
# Let's start by adding all edges to an undirected graph
e <- c(1,5, 2,5, 3,5, 4,5, 6,5, 7,5, 8,5, 9,5)
g <- graph(edges=e, n=9, directed=FALSE)
g
```

```
## IGRAPH 75dbc77 U--- 9 8 --
## + edges from 75dbc77:
## [1] 1--5 2--5 3--5 4--5 5--6 5--7 5--8 5--9
```

Note that the first line of the generated output gives you some information about the graph object we just created. The `U` flag in the first line tells us that this network is undirected. For a directed network, the flag would be set to `D`. There are thre flags that are currently not set: `N` for named networks, where nodes have a name; `W` for weighted networks, and `B` for two-mode networks. The next two numbers in the first line refer to the number of nodes and edges, followed by the name of the network, which is currently not set. If our network had more attributes, we would find those in the following lines.

Let's have a look at simple plot of the network:

```
plot(g)
```



Looking at the output, we can see that the network now contains all the edges and nodes we need. However, let's take a look at another handy way to create networks by using literals, a shorthand notation:

```
g <- graph_from_literal(1-5, 2-5, 3-5, 4-5, 6-5, 7-5, 8-5, 9-5)
g
```

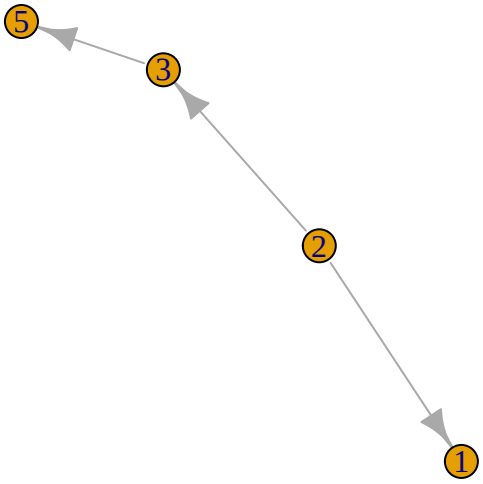
```
## IGRAPH b0b0242 UN-- 9 8 --
## + attr: name (v/c)
## + edges from b0b0242 (vertex names):
## [1] 1--5 5--2 5--3 5--4 5--6 5--7 5--8 5--9
```

This time, our nodes got autmatically named. Their name, in this case the numbers, are stored in the `name` name attribute. Note that the number of dashes used to specify the edges doesn't matter. We can also add a `+` on one side of an edge to specify an edge direction:

```
g2 <- graph_from_literal(1+-2, 2-+3, 3-+5)
g2
```

```
## IGRAPH c653c15 DN-- 4 3 --
## + attr: name (v/c)
## + edges from c653c15 (vertex names):
## [1] 2->1 2->3 3->5
```

```
plot(g2)
```



Sometimes we want to remove nodes and edges from the network. We can delete an edge by removing it from the edge sequence displayed in the last line of the output above. The following example removes the second edge `2->3`:

```
delete_edges(g2, c(2))
```

```
## IGRAPH 48a15f6 DN-- 4 2 --
## + attr: name (v/c)
## + edges from 48a15f6 (vertex names):
## [1] 2->1 3->5
```

Removing vertices works in a similar way. We can remove vertices by their index or name attribute. If we remove them, all adjacent edges are removed, too:

```
delete_vertices(g2, 3)    # Removes vertex at index 3 (node 5)
```

```
## IGRAPH fe55db6 DN-- 3 1 --
## + attr: name (v/c)
## + edge from fe55db6 (vertex names):
## [1] 2->1
```

```
delete_vertices(g2, "5")  # Removes vertex with name "5"
```

```
## IGRAPH db67036 DN-- 3 2 --
## + attr: name (v/c)
## + edges from db67036 (vertex names):
## [1] 2->1 2->3
```

We can also add nodes and edges to the network. The following example adds the node `4` to `g2` and connects it to `3` and `5`. Note that we use the same handy `%>%` notation we used for chaining data frame operations earlier:

```
g2 %>%
  add_vertices(1, name= "4") %>%  # Adds 1 vertex with the name attribute "4"
  add_edges(c("4","3", "4", "5")) # Adds an edge from 4 to 3 and 4 to 5
```

```
## IGRAPH 35d2d63 DN-- 5 5 --
## + attr: name (v/c)
## + edges from 35d2d63 (vertex names):
## [1] 2->1 2->3 3->5 4->3 4->5
```

Network, node, and edge attributes

To recreate our example, we need to add node and edge attributes. Before we can do this, we need to learn how to access the network, nodes, and edges:

```
g     # Returns the network
```

```
## IGRAPH b0b0242 UN-- 9 8 --
## + attr: name (v/c)
## + edges from b0b0242 (vertex names):
## [1] 1--5 5--2 5--3 5--4 5--6 5--7 5--8 5--9
```

```
E(g)  # Returns the edges of the network
```

```
## + 8/8 edges from b0b0242 (vertex names):
## [1] 1--5 5--2 5--3 5--4 5--6 5--7 5--8 5--9
```

```
V(g)  # Returns the vertices of the network
```

```
## + 9/9 vertices, named, from b0b0242:
## [1] 1 5 2 3 4 6 7 8 9
```

We can get a named list of all network, node, and edge attributes as follows:

```
graph_attr(g)
```

```
## named list()
```

```
edge_attr(g)
```

```
## named list()
```

```
vertex_attr(g)
```

```
## $name
## [1] "1" "5" "2" "3" "4" "6" "7" "8" "9"
```

We can see that we have no graph and edge attributes in our example. Our nodes already have a `name` attribute, which we can access as follows:

```
V(g)$name
```

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

```
get.vertex.attribute(g, "name")
```

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

Let's name our network by assigning it a graph attribute `name`:

```
g <- set_graph_attr(g, "name", "Example network")
g$name
```

```
## [1] "Example network"
```

Now let's set some node attributes. Our example networks has three groups of nodes. Nodes 1-4, 5, and 6-9 each belong to one group:

```
g <- g %>%
  set_vertex_attr("group", c("1", "2", "3", "4"), "A") %>%
  set_vertex_attr("group", c("5"), "B") %>%
  set_vertex_attr("group", c("6", "7", "8", "9"), "C")
vertex_attr(g)
```

```
## $name
## [1] "1" "5" "2" "3" "4" "6" "7" "8" "9"
##
## $group
## [1] "A" "B" "A" "A" "A" "C" "C" "C" "C"
```

The last step is to add edge attributes. The edges from 1, 2, 3, and 4 to node 5 are strong ties, while the other four edges are weak.

```
E(g)  # The first four edges are the one's we want to assign a higher weight
```

```
## + 8/8 edges from b0b0242 (vertex names):
## [1] 1--5 5--2 5--3 5--4 5--6 5--7 5--8 5--9
```

```
g <- g %>%
  set_edge_attr("weight", c(1:4), 2) %>%
  set_edge_attr("weight", c(5:8), 1)
edge_attr(g)
```

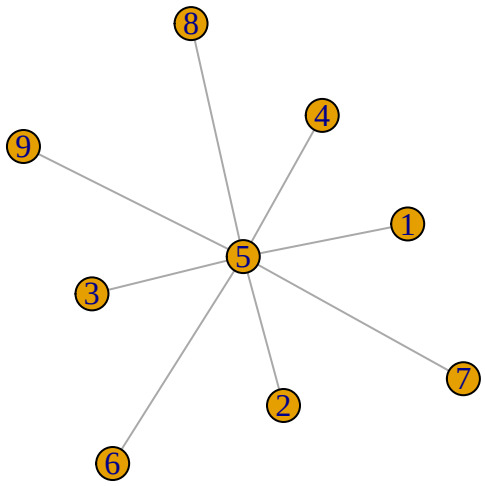
```
## $weight
## [1] 2 2 2 2 1 1 1 1
```

Looks like we're done! Our network contains all the data we need. The summary now shows the network attribute (`g/c`), the node attributes (`v/c`), and the edge attributes (`e/n`).

```
g
```

```
## IGRAPH b0b0242 UNW- 9 8 -- Example network
## + attr: name (g/c), name (v/c), group (v/c), weight (e/n)
## + edges from b0b0242 (vertex names):
## [1] 1--5 5--2 5--3 5--4 5--6 5--7 5--8 5--9
```

```
plot(g)
```



Our plot hasn't changed. We'll work on that in the next chapter. Before we do that, let's save the network.

Write and read network data

Writing networks in igraph is done with the simple command `write_graph()`; igraph supportes the most common formats to store network data, including edgelists, pajek, and graphml. Since we want to store our node and edge attributes with the network, we choose the `graphml` format.

```
write_graph(g, "./graphs/star-undirected.gml", "graphml")
```

Now let's read the graph and check if everything worked:

```
read_graph("./graphs/star-undirected.gml", "graphml")
```

```
## IGRAPH 3b284d2 UNW- 9 8 -- Example network
## + attr: name (g/c), name (v/c), group (v/c), id (v/c), weight
## | (e/n)
## + edges from 3b284d2 (vertex names):
## [1] 1--5 5--2 5--3 5--4 5--6 5--7 5--8 5--9
```

Looks good! Let's move on to fixing the plot.

Plotting networks

When plotting networks, we have a couple of options to consider for the display of nodes, edges, and the network as a whole. We've already used the `plot()` function without additional parameters. In the following, we will continue using it to specify additional display attributes. You can find a comperehensive overview of the various attributes in igraph's documentation by calling `?igraph.plotting`.

The most commonly modified node properties are color, size, and shape.

Modifying the display properties of nodes

In our star network, for example, we would like to colour the nodes blue, black, and red, according to their attribute "group." R provides a set of colour palettes, which allow us to refer to certain basic colours by name. You can print a list of them using the command `color()`. Here, we're simply using the colours `blue`, `black`, and `red`.

```
V(g)$name   # Our nodes
```

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

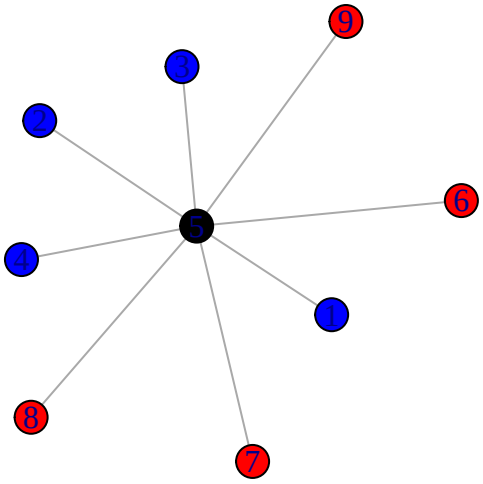
```
V(g)$group  # The group attribute we want to map to colors
```

```
## [1] "A" "B" "A" "A" "A" "C" "C" "C" "C"
```

```
clr <- c("blue", "black", "red")  # A vector containing the colours we want to use

# We're using a little trick here. We'll use the colour vector as a lookup table
# Remember how factors work? If we create a factor for our group variable,
# it will identify the three distinct groups of our example, which we can use
# to access the index of the vector.
V(g)$color <- clr[factor(V(g)$group)]
V(g)$color
```

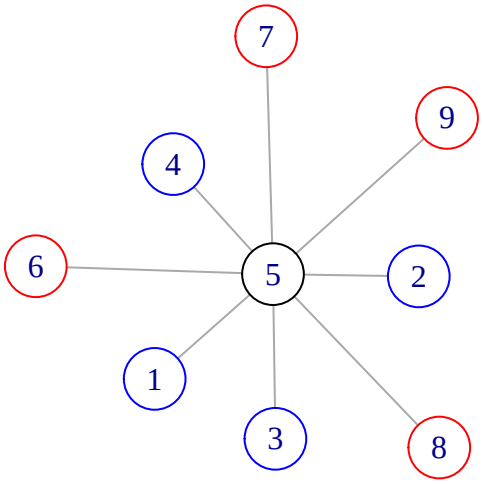
```
## [1] "blue"  "black" "blue"  "blue"  "blue"  "red"   "red"   "red"   "red"
```

```
plot(g)
```



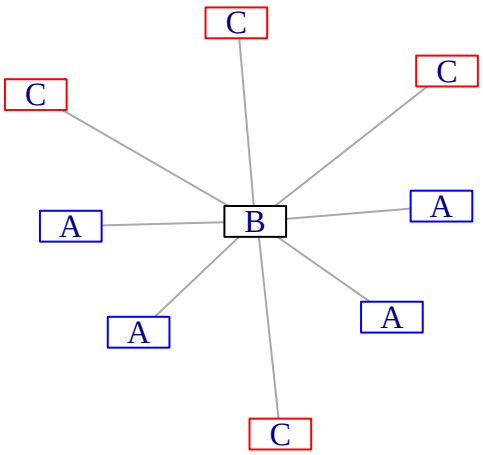
As you can see, igraph used the color attribute to colour the nodes. Looking at our example, we see that we actually wanted to color the borders, not the nodes themselves. We can simply fix that by modifying a different visual property of the network, `frame.color`. We'll also increase the size of the nodes:

```
V(g)$frame.color <- V(g)$color
V(g)$color <- "white"
V(g)$size <- 30
plot(g)
```



Adjusting other node attributes, including the labels, works similarly. Here's an example of chained attribute modifications that change the shape and labels of our nodes:

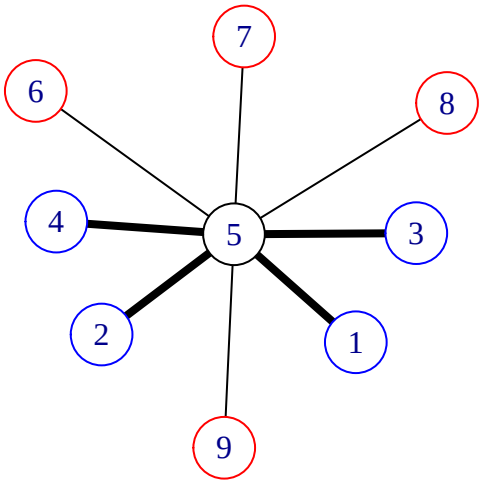
```
plot(
  g %>%
    set_vertex_attr("shape", V(g), "rectangle") %>%
    set_vertex_attr("label", V(g), V(g)$group)
)
```



Modifying the display properties of edges

Moving on to edge attributes, we still have to visualize the `weight` attribute of our example. This requires us to modify the edge property `width`. We'll also rescale our weights, since they are either 1 or 2, depending on their strength. Last but not least, we change their color to black to match our example:

```
E(g)$width <- E(g)$weight^2   # We simply assign the squared weight
E(g)$color <- "black"
plot(g)
```



We're pretty close to our example. The layout is still off. We'll fix that in the next step.

Modifying the display properties of the network

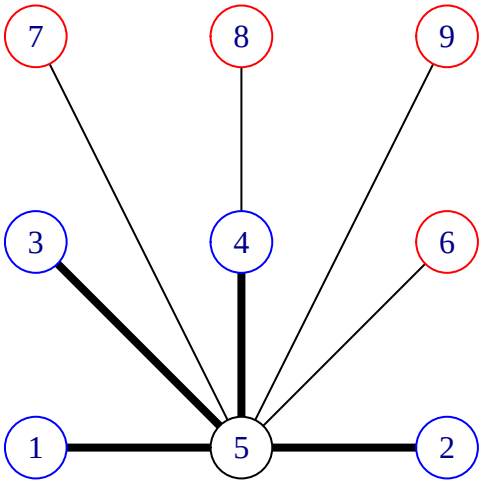
Here, we will briefly discuss the layout options provided by igraph. There are many options to choose from, including force-directed and geometrical layouts. We will fix our example first and then explore some of the options using a randomly generated network.

The layout of the network is determined by the `layout` parameter of the `plot` function. Layouts can be generated with igraphs `layout_*` functions or created manually. A layout is just a 2x3 matrix, which assigns each of the nodes in the graph a (x,y) coordinate. Let's take a look at the `layout_on_grid()` function, which should provide a layout that is simlar to our example network:

```
layout_on_grid(g) # Generates the layout matrix
```

```
##       [,1] [,2]
##  [1,]    0    0
##  [2,]    1    0
##  [3,]    2    0
##  [4,]    0    1
##  [5,]    1    1
##  [6,]    2    1
##  [7,]    0    2
##  [8,]    1    2
##  [9,]    2    2
```

```
plot(g, layout=layout_on_grid(g)) # Plots the network
```



We're moving in the right direction, but our nodes are not correctly positioned. Looking at the layout matrix, we can see that this makes sense. The layout function assigned the wrong position to our nodes. Let's simply make our own layout:

```
V(g) # Prints the node sequence that we need to assign positions at the right index
```

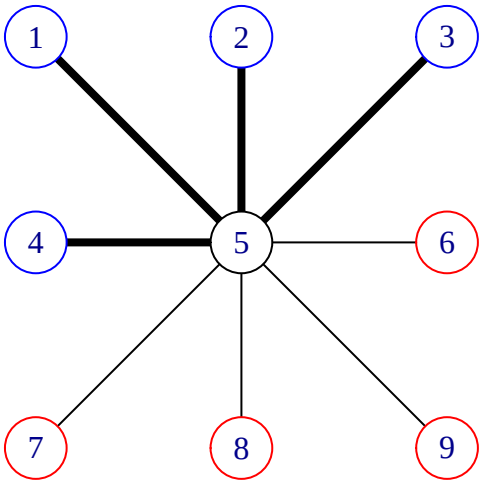
```
## + 9/9 vertices, named, from b0b0242:
## [1] 1 5 2 3 4 6 7 8 9
```

```
# We specify x and y coordinates for each node separately.
# Example: Node 5 is at position two in our node set.
#   To position it in the center, we set the second coordinate
#   to x=1 and y=1. Similarly, we place the last node in the
#   sequence, node 9, at position x=2 and y=0.
# Note: igraph coordinates start in the lower left corner
#   with x=0 and y=0.
x <- c(0, 1, 1, 2, 0, 2, 0, 1, 2)
y <- c(2, 1, 2, 2, 1, 1, 0, 0, 0)
l <- cbind(x, y) # Combines both vectors to a matrix
l
```

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

```
##  [5,] 0 1
##  [6,] 2 1
##  [7,] 0 0
##  [8,] 1 0
##  [9,] 2 0
```

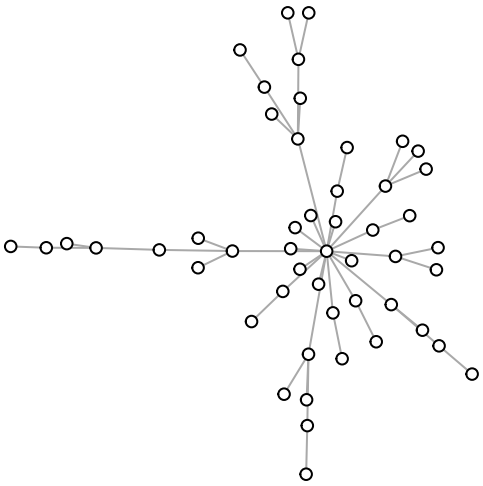
```
plot(g, layout=l)
```



Mission accomplished. We successfully recreated the example introduced earlier!

This example is a little to simple to explore more complex layout options. We simply generate a new network, e.g. based on the famous preferential attachment model:

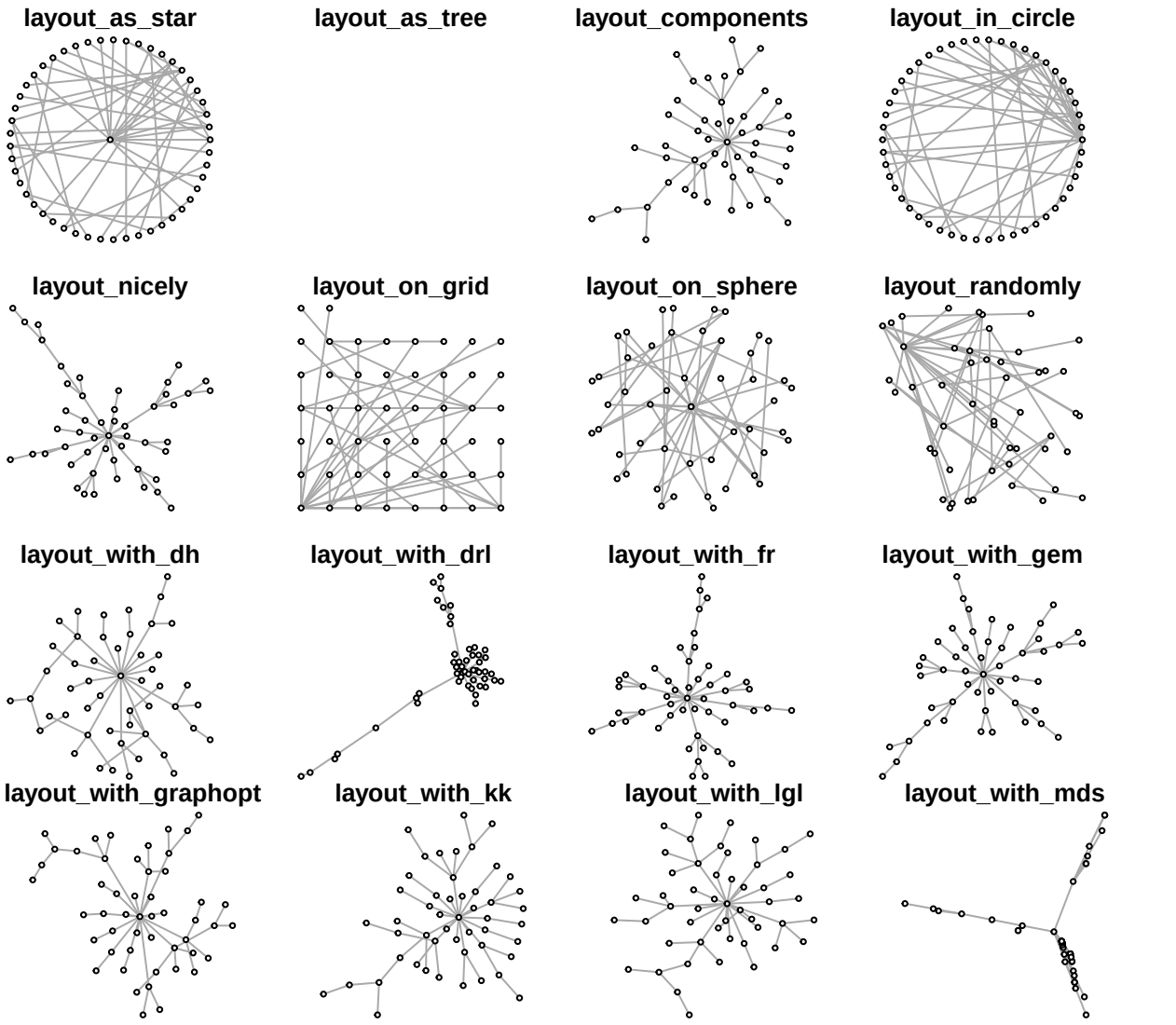
```
pa <- sample_pa(50)
V(pa)$label <- ""
V(pa)$color <- "white"
V(pa)$size <- 5
E(pa)$arrow.mode <- 0
plot(pa)
```



Here's an overview of the layouts available in igraph:

```
# Get all layout functions from the igraph package
# and remove the layout for bipartite and layered graphs
ls <- grep("^layout_", ls("package:igraph"), value=TRUE)[-1]
```

```
ls <- ls[!grepl("bipartite|sugiyama", ls)]

# Sets the parameters for our grid
par(mfrow=c(3,4), mar=c(1,1,1,1))

# Loops over all layouts and plots our graph
# with each layout
for (l_name in ls) {
  l <- do.call(l_name, list(pa))
  plot(pa, layout=l, main=l_name)
}
```



Interactive network visualization

Plotting with igraph is limited to static plots. There are a couple of interesting solutions to produce interactive plots in R. Here, we will look at a simple example using the library `visNetwork`, which allows for a simple manupulation of the most commonly adjusted display properties similar to igraph. Like many of the interactive visualization libraries, visNetwork is based on a javascript library (i.e. `vis.js`).

To plot our example network g, we have to convert it from an igraph object to something that can be interpreted by visNetwork. To plot a network, visNetwork needs a list of nodes (at least with some kind of node id) or a data frame, and an edge list (or data frame) that contains at least a `from` and a `to` field per edge. Gladly, the function `as_data_frame()` returns exactly that from an igraph object:

```
# We'll use a new color palette for this one
clr <- c("cornflowerblue", "darkgray", "coral")

# Create a node data frame from the igraph object.
# We keep only the name and the group field
n <- as_data_frame(g, what="vertices") %>%
  select(name, group)

# Set the node id, as required by visNetwork and set some visual properties
n$id <- n$name
n$label <- n$name
n$color.background <- clr[factor(n$group)]
n$color.border <- "black"
n$color.highlight.background <- "white"
n
```

```
##   name group id label color.background color.border
## 1    1     A  1     1   cornflowerblue        black
## 5    5     B  5     5         darkgray        black
## 2    2     A  2     2   cornflowerblue        black
## 3    3     A  3     3   cornflowerblue        black
## 4    4     A  4     4   cornflowerblue        black
## 6    6     C  6     6            coral        black
## 7    7     C  7     7            coral        black
## 8    8     C  8     8            coral        black
## 9    9     C  9     9            coral        black
##   color.highlight.background
## 1                      white
## 5                      white
## 2                      white
## 3                      white
## 4                      white
## 6                      white
## 7                      white
## 8                      white
## 9                      white
```

```
# Create an edge data frame from the igraph object.
# We keep only the from, to, and weight field
e <- as_data_frame(g, what="edges") %>%
  select(from, to, weight)
e$width <- e$weight^2
e
```

```
##   from to weight width
## 1    1  5      2     4
## 2    5  2      2     4
## 3    5  3      2     4
## 4    5  4      2     4
## 5    5  6      1     1
## 6    5  7      1     1
```

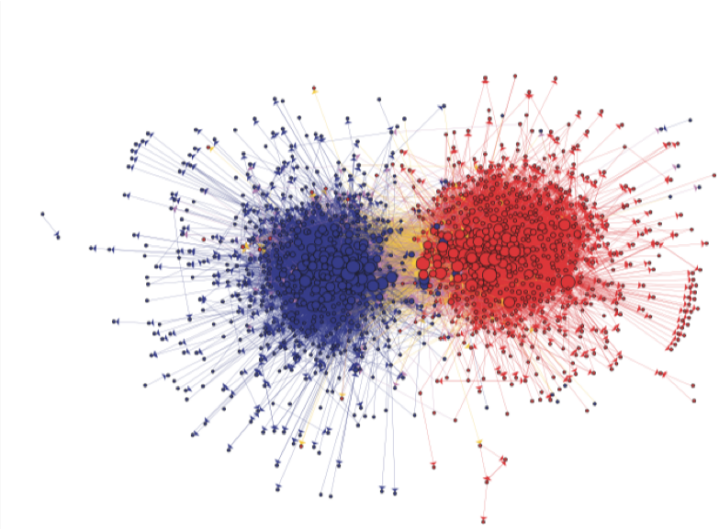
```
## 7    5  8      1     1
## 8    5  9      1     1
```

We can now use the two data frames to plot the network with visNetwork:

```
library("visNetwork")
visNetwork(nodes=n, edges=e)
```

Node and network level measures

Before we start computing node and network level measures, it's time to switch to a real dataset. We use the publicly available dataset on the political blogosphere during the 2004 US election.



Source: Adamic, L. A., & Glance, N. (2005, August). The political blogosphere and the 2004 US election: divided they blog. *In Proceedings of the 3rd international workshop on Link discovery* (pp. 36-43). ACM.

```
g <- read_graph("./graphs/polblogs.gml", format="gml")
g
```

```
## IGRAPH 1b09984 D--- 1490 19025 --
## + attr: id (v/n), label (v/c), value (v/c), source (v/c), id
## | (e/n), value (e/n)
## + edges from 1b09984:
##  [1]  267->1394  267-> 483  267->1051  904->1479  904-> 919  904->1045
##  [7]  904->1330  904->1108  904-> 995  904-> 963  904->1000  904->1192
## [13]  904->1067  904->1270  904->1437  904->1037  903-> 855  903->1008
## [19]  982->1429  982-> 952  982-> 892  982->1479  982-> 956  982->1306
## [25]  982-> 810  982->1437  982->1112  982->1051  982-> 826  982->1478
## [31]  982-> 979  982->1255 1167-> 963 1167-> 802 1167-> 842 1167->1408
## [37] 1167-> 862 1167->1437  708-> 374  708-> 508  708->  99  708-> 483
## + ... omitted several edges
```

Here a little background on the dataset:

- The network comprises political blogs and their link structure
- It covers 1490 blogs and 19025 links between them
- Adamic & Glance compiled the dataset during the US Presidential Elections in 2004
- Nodes have two attributes: *source* and *value*

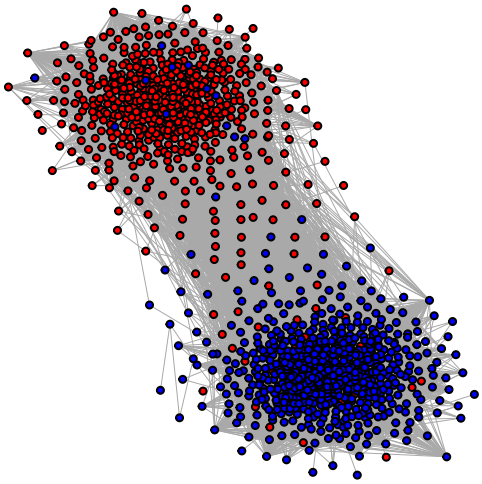
- Source refers to the blog directory the blog has been found on
- Value refers to the political leaning of a blog (0 for left/liberal ~ blue; 1 for right/conservative ~ red)
- The weight of an edge refers to the amount of times a blog has been cited
- This edge attribute is incomplete in the public dataset and can be omitted

Before we take a closer look at the network, let's remove loops and focus on the main component of the network, which implies removing all isolated nodes and focusing just on the largest group of connected nodes:

```
# Reduce the network to the largest component
cWeak <- clusters(g, mode="weak")    # Identify all components
maxID <- which.max(sizes(cWeak))     # Get the ID of the largest component
g <- induced.subgraph(g, V(g)[cWeak$membership==maxID]) # Extract the subgraph

# Now let's remove loops, which can be done with simplify
g <- simplify(g, remove.loops = TRUE)
```

Let's plot the result:

```
# The DRL algorithm works well with large networks
l <- layout_with_drl(
  as.undirected(g),
  options=list(simmer.attraction=0)
)

plot(
  as.undirected(g),
  vertex.color = c("red", "blue")[factor(V(g)$value)],
  vertex.frame.color = "black",
  vertex.size = 3,
  vertex.label = "",
  edge.width = 0.05,
  layout = l
)
```



Distribution of node attributes

First, let's have a closer look at the node and edge attributes:

```
vcount(g)                # Gets the number of nodes
```

```
## [1] 1222
```

```
vertex_attr_names((g))  # Node attributes
```

```
## [1] "id"     "label"  "value"  "source"
```

```
ecount(g)               # Gets the number of edges
```

```
## [1] 19021
```

```
edge_attr_names(g)      # Edge attribute names
```

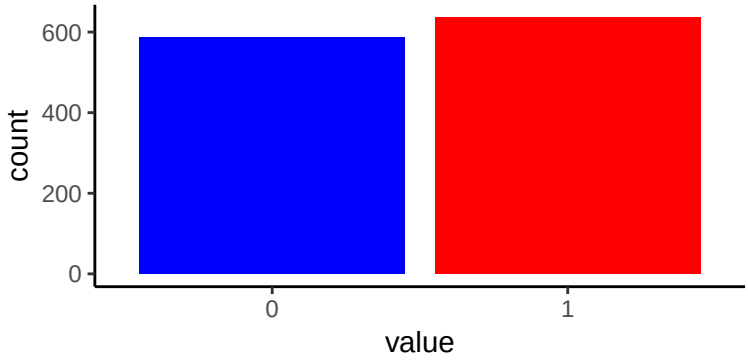
```
## character(0)
```

We already know that nodes have two interesting properties: `value` and `source`. The former referring to the political leaning of the blog and the latter to the page it has been found on. Let's explore both attributes:

```
n <- as_data_frame(g, what="vertices") # Get the node data frame
table(n$value) # Simple frequency table
```

```
##
##   0   1
## 586 636
```

```
# Get a new color palette to match the example
clr <- c("blue", "red")

# We use ggplot to produce a simple bar plot
ggplot(n, aes(x=value, fill=value)) +
  geom_bar() +
  scale_fill_manual(values=clr) +
  theme_classic() +
  theme(legend.position="none")
```



We can also inspect the sources and count how many blogs they contributed to the dataset:

```
n %>%
  group_by(source) %>%
  count() %>%
  arrange(desc(n))
```

```
## # A tibble: 47 x 2
## # Groups:   source [47]
##    source                      n
##    <chr>                   <int>
##  1 Blogarama                 464
##  2 BlogCatalog               132
```

```
##  3 eTalkingHead               124
##  4 LabeledManually            120
##  5 CampaignLine                67
##  6 Blogarama,BlogCatalog       48
##  7 LeftyDirectory              44
##  8 Blogarama,eTalkingHead      36
##  9 Blogarama,LeftyDirectory    36
## 10 BlogPulse                   31
## # ... with 37 more rows
```

Node degrees

The *degree* of a node is the number of links that involve that node, which is the cardinality of the node's neighbourhood. Thus node is degree in a network g, denoted by $d_i(g)$, is

$$d_i(g) = \#\{j : \ g_{ji} = 1\} = \#N_i(g)$$

In case of a directed network, the previous calculation is the node's *in-degree*:

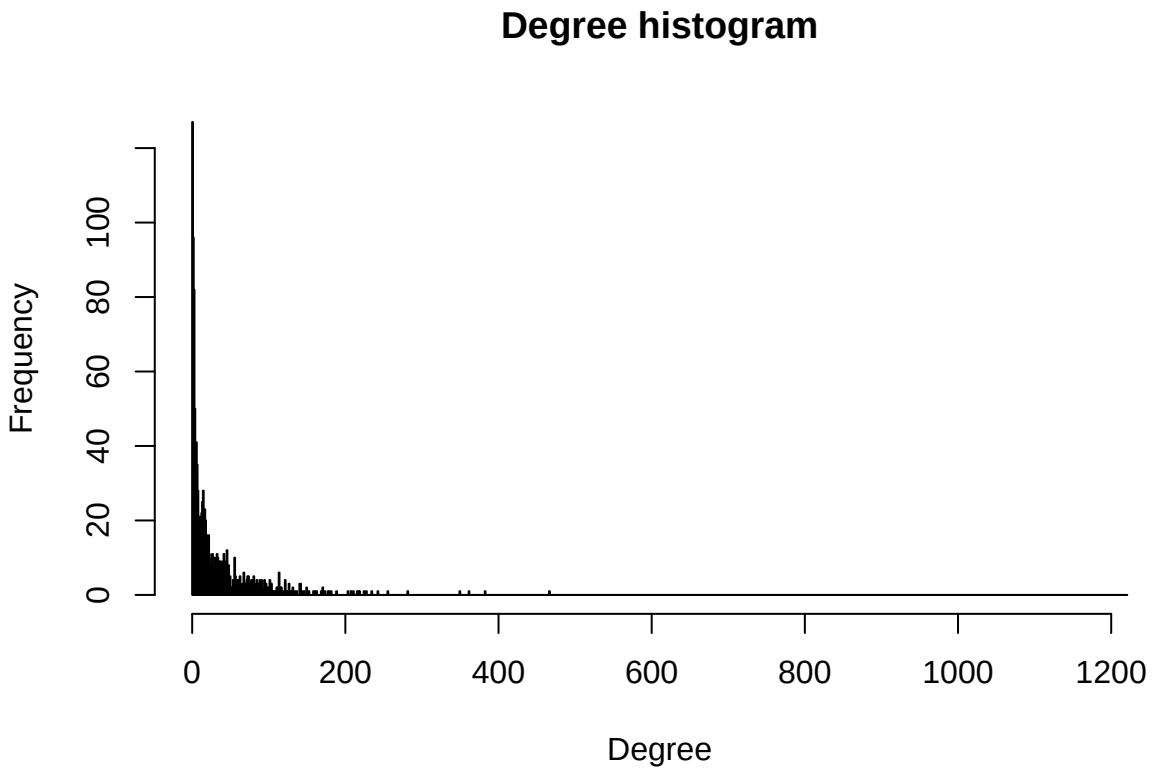
$$\#\{j : \ g_{ji} = 1\}$$

The *out-degree* of a node i is denoted by

$$\#\{j : \ g_{ij} = 1\}$$

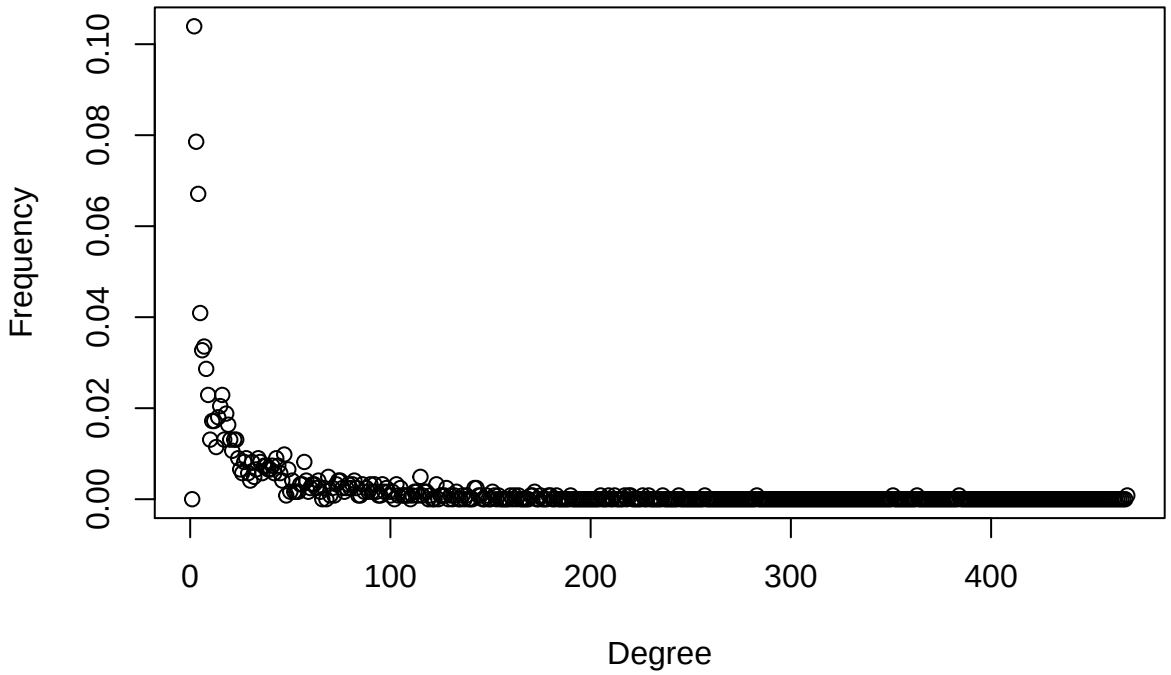
In igraph, the degree can be computed using the `degree()` function. The parameter `mode` can be used to specify the type of degree that we want to compute (i.e. `all`,`in` or `out`):

```
V(g)$degree <- degree(g, mode="all")
hist(V(g)$degree, breaks=1:vcount(g)-1, main="Degree histogram", xlab = "Degree")
```



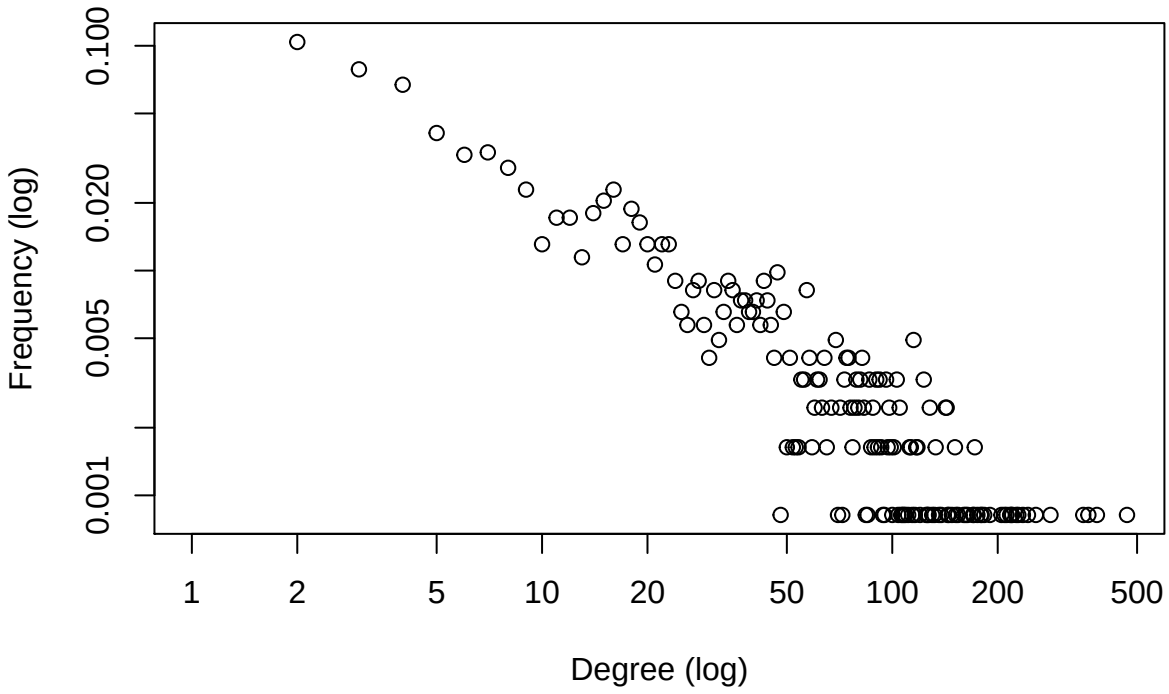
Instead of looking at the histogram, we can also plot the degree distribution:

```
# We compute and plot the cumulative degree distribution
d.dist <- degree_distribution(g, cumulative=FALSE, mode="all")
plot(d.dist, ylab="Frequency", xlab="Degree")
```




```
# Sometimes it's interesting to investigate the log-log plot.
# For example when fitting power laws.
plot(d.dist, log="xy", ylab="Frequency (log)", xlab="Degree (log)")
```

```
## Warning in xy.coords(x, y, xlabel, ylabel, log): 309 y values <= 0 omitted
## from logarithmic plot
```



Centralities

Measures of centrality can be categorized into four main groups depending on the types of statistics on which they are based:

- Degree: how connected a node is.
- Neighbours' characteristics (e.g., Eigenvector): how important, central, or influential a node's neighbours are.
- Closeness: how easily a node can reach other nodes.
- Betweenness: how important a node is in terms of connecting other nodes.

Centralities describe the position of a node, while *centralizations* refer to the distribution of specific centralities in the network.

Degree centrality

The *degree centrality* of a node i is

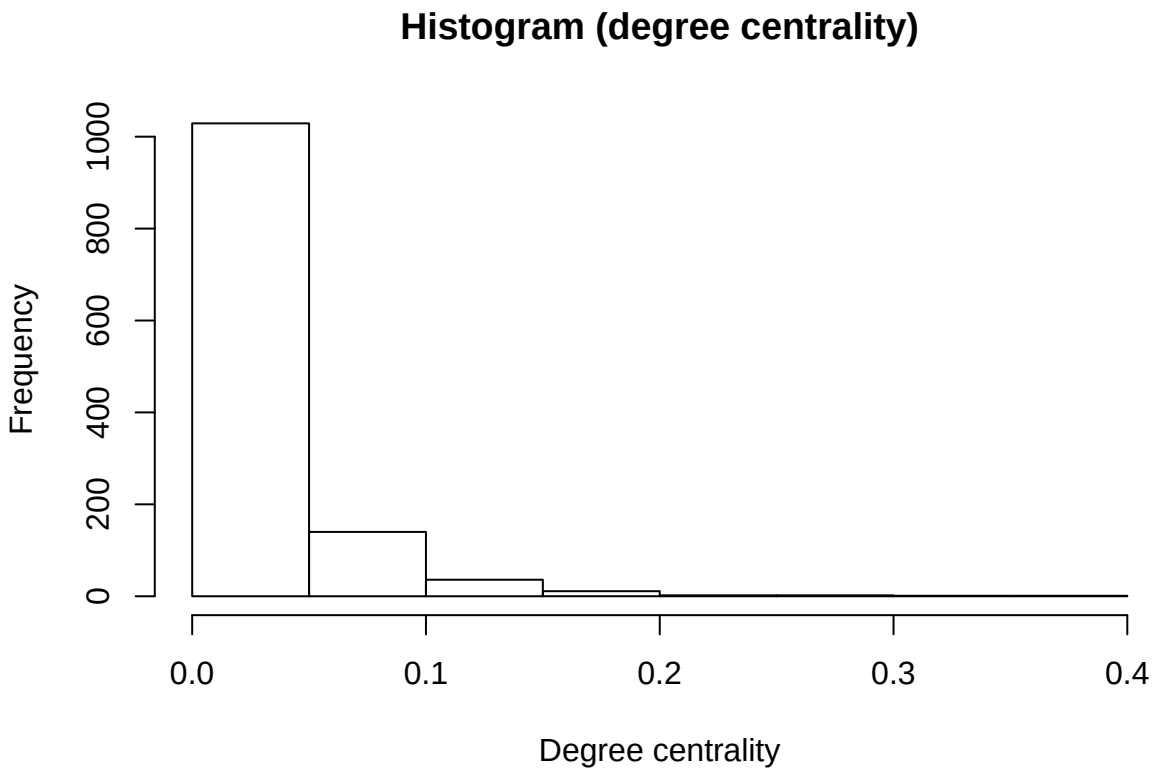
$$Ce_i^D(g) = d_i(g)$$

and its *normalized degree centrality* is

$$Ce_i^{D_n}(g) = \frac{d_i(g)}{n-1}$$

We already calculated the degree above. To get the degree centrality, we normalize the degree output:

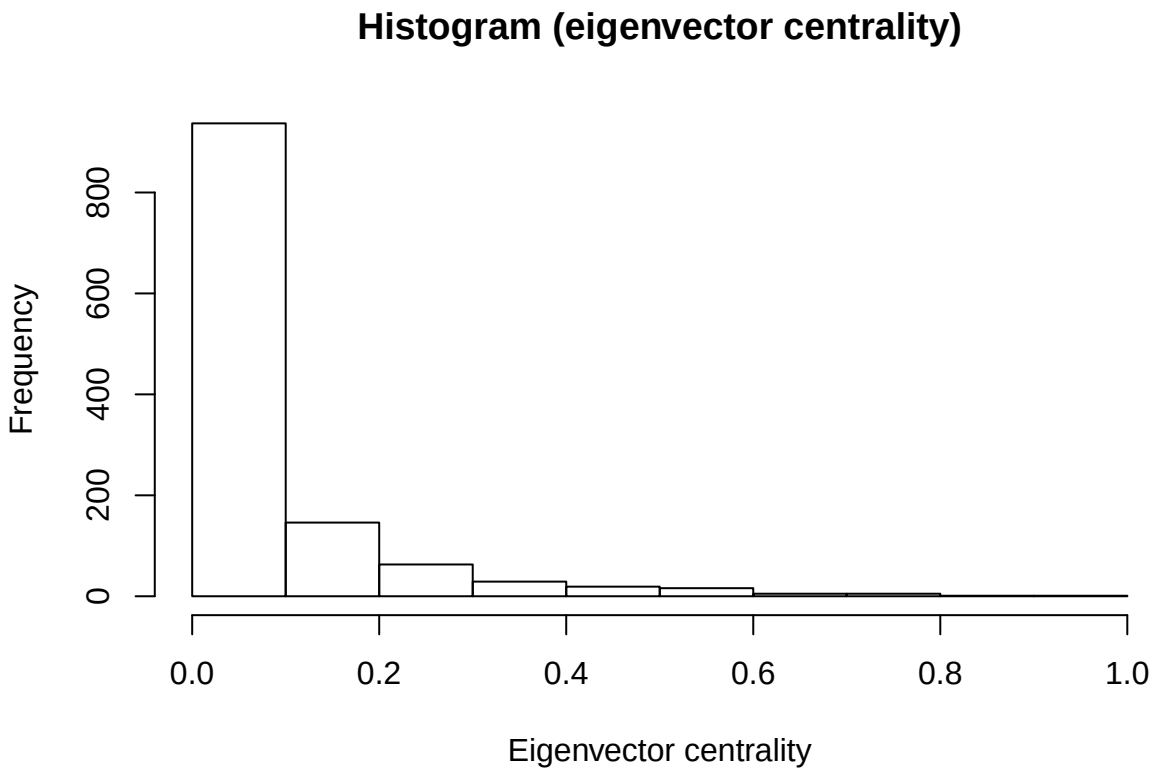
```
V(g)$centr.deg <- degree(g, mode="all", normalize=TRUE)
hist(
  V(g)$centr.deg,
  main="Histogram (degree centrality)",
  xlab="Degree centrality"
)
```



Eigenvector centrality

The *eigenvector centrality* of a node is a slightly more complex measure than the degree centrality. Instead of just counting a node's neighbours, this measure weights a node's adjacent ties based on the centrality of it's neighbours.

```
eig <- eigen_centrality(g, directed=FALSE)
eig <- unlist(eig$vector, use.names=FALSE)
V(g)$centr.eig <- eig
hist(
  V(g)$centr.eig,
  main="Histogram (eigenvector centrality)",
  xlab="Eigenvector centrality"
)
```



Closeness centrality

This class of measures tracks how close a given node is to any other node. One closeness-based measure is just the inverse of the average distance between i and any other node j:

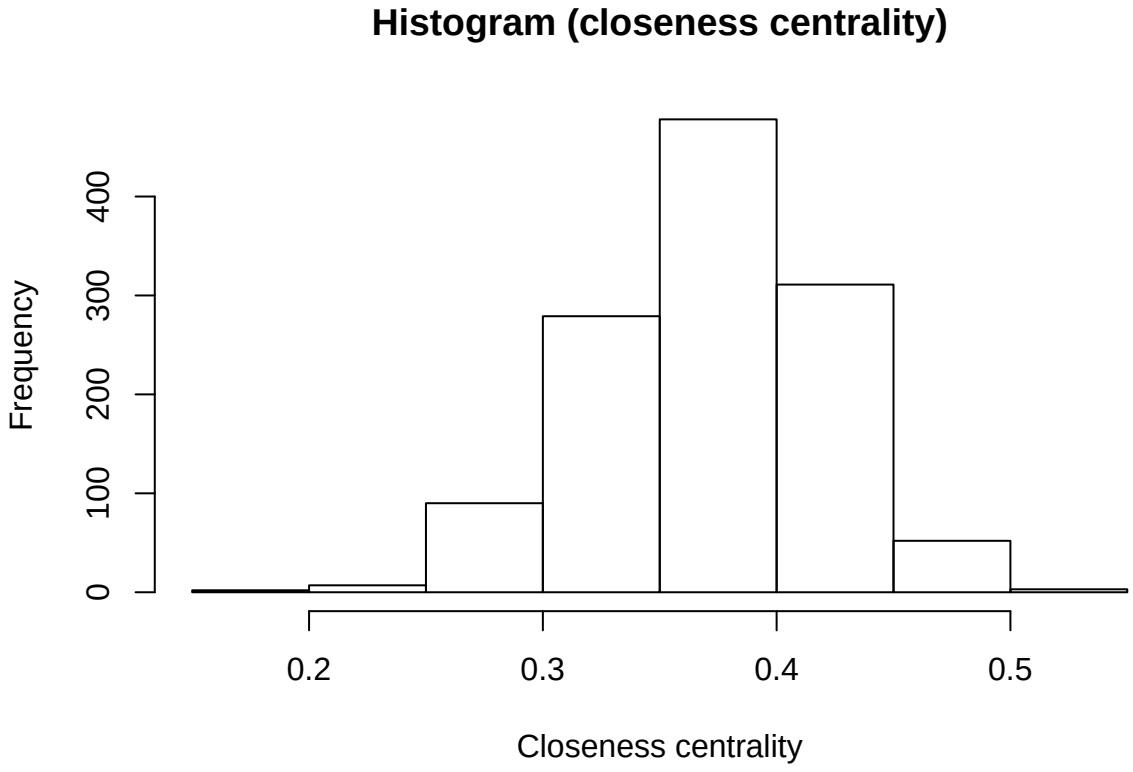
$$Ce_i^C(g) = \frac{1}{\sum_{j \neq i} l(i,j)}$$

and

$$Ce_i^{C_n}(g) = \frac{n-1}{\sum_{j \neq i} l(i,j)}$$

where $l(i,j)$ is the number of links in the shortest path between i and j.

```
V(g)$centr.clo <- closeness(g, mode="all", weights=NA, normalized = TRUE)
hist(
  V(g)$centr.clo,
  main="Histogram (closeness centrality)",
  xlab="Closeness centrality"
)
```



Betweenness centrality

A measure of centrality that is based on how well situated a node is in terms of the paths that it lies on was first proposed by Freeman.

Let $P_i(k_j)$ denote the number of geodesics (shortest paths) between k and j that i lies on, and let $P(k_j)$ be the total number of geodesics between k and j. We can estimate how important i is in terms of connecting k and j by looking at the ratio

$$\frac{P_i(k_j)}{P(k_j)}$$

If this ratio is close to 1, then i lies on most of the shortest paths connecting k to j, while if it is close to 0, then i is less critical to k and j.

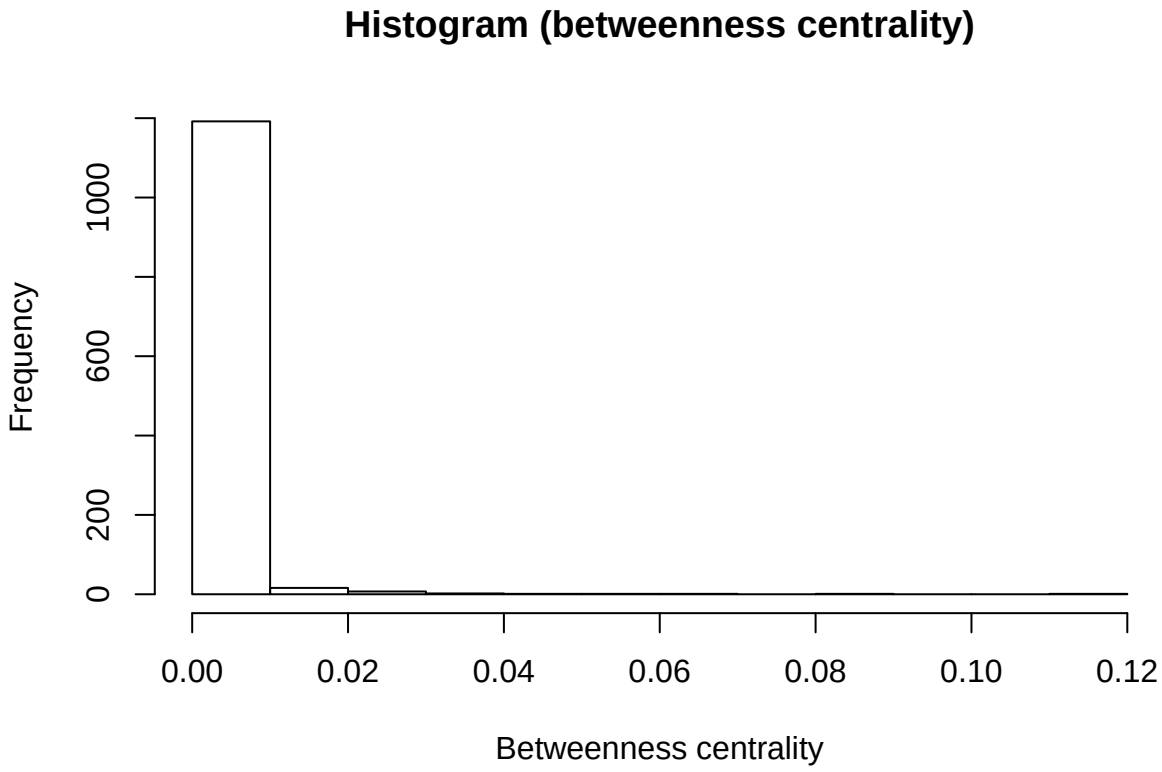
Averaging across all pairs of nodes, the *betweenness centrality* of a node i is

$$Ce_i^B(g) = \sum_{k \neq j; i \notin \{k,j\}} \frac{P_i(k_j)}{P(k_j)}$$

and its *normalized betweenness centrality* is

$$Ce_i^{B_n}(g) = \sum_{k \neq j; i \notin \{k,j\}} \frac{\frac{P_i(k_j)}{P(k_j)}}{\frac{(n-1)(n-2)}{2}}$$

```
V(g)$centr.bet <- betweenness(g, directed=FALSE, weights=NA, normalized = TRUE)
hist(
  V(g)$centr.bet,
  main="Histogram (betweenness centrality)",
  xlab="Betweenness centrality"
)
```



Centralizations

Centralization measures the extent to which a network consisted of a highly central actor surrounded by peripheral actors. Centralization is the sum of the difference in centrality of the most central actor to all others, normalized by the maximum possible over all connected graphs:

$$\frac{\sum[c_* - c_i]}{max \sum[c_* - c_i]},$$

where c_i is the centrality of node i and c_* is the centrality of the most central node.

Note that igraph's centralization functions return a lot of values, including the `theoretical_max` used to compute the centralization. Here, we are just interested in the `centralization` value.

```
g$centr.deg <- centr_degree(g, mode="all", normalized=TRUE)$centralization
g$centr.eig <- centr_eigen(g, directed=FALSE, normalized=TRUE)$centralization
g$centr.clo <- centr_clo(g, mode="all", normalized=TRUE)$centralization
g$centr.bet <- centr_betw(g, directed=FALSE, normalized=TRUE)$centralization
graph_attr(g)
```

```
## $centr.deg
## [1] 0.1786347
##
## $centr.eig
## [1] 0.9227435
##
## $centr.clo
## [1] 0.2943235
##
## $centr.bet
## [1] 0.1093113
```

Reciprocity

Reciprocity is a dyiadic measure, which refers to the fraction of reciprocal ties in a directed network. It can be computed using the `reciprocity` function. The `dyad_census` function computes the number of mutual ties and the number of asymmetric ties.

```
g$reciprocity <- reciprocity(g)
dc <- dyad_census(g)
g$mut <- dc$mut
g$asym <- dc$asym

g$reciprocity
```

```
## [1] 0.242574
```

```
g$mut
```

```
## [1] 2307
```

```
g$asym
```

```
## [1] 14407
```

Transitivity

Moving away from the dyiadic level of analysis, *transitivity* is a measure that refers to triadic relationships in the network. It has a *local* and a *global* definition. On the global level, it is defined as the fraction of all closed triangles and the number of all triangles in the network. On the local level, it refers the fraction of all closed triangles an individual node is part of and the number of total triangles the node is involved in. Among the various types available for this measure, there is also a `localaverage` version, which computes the average version of the local transitivy. Both measures are frequently used to measure the degree of clustering present in the network.

Note that while there's an `undirected` type, the `global` type always treats the network as undirected.

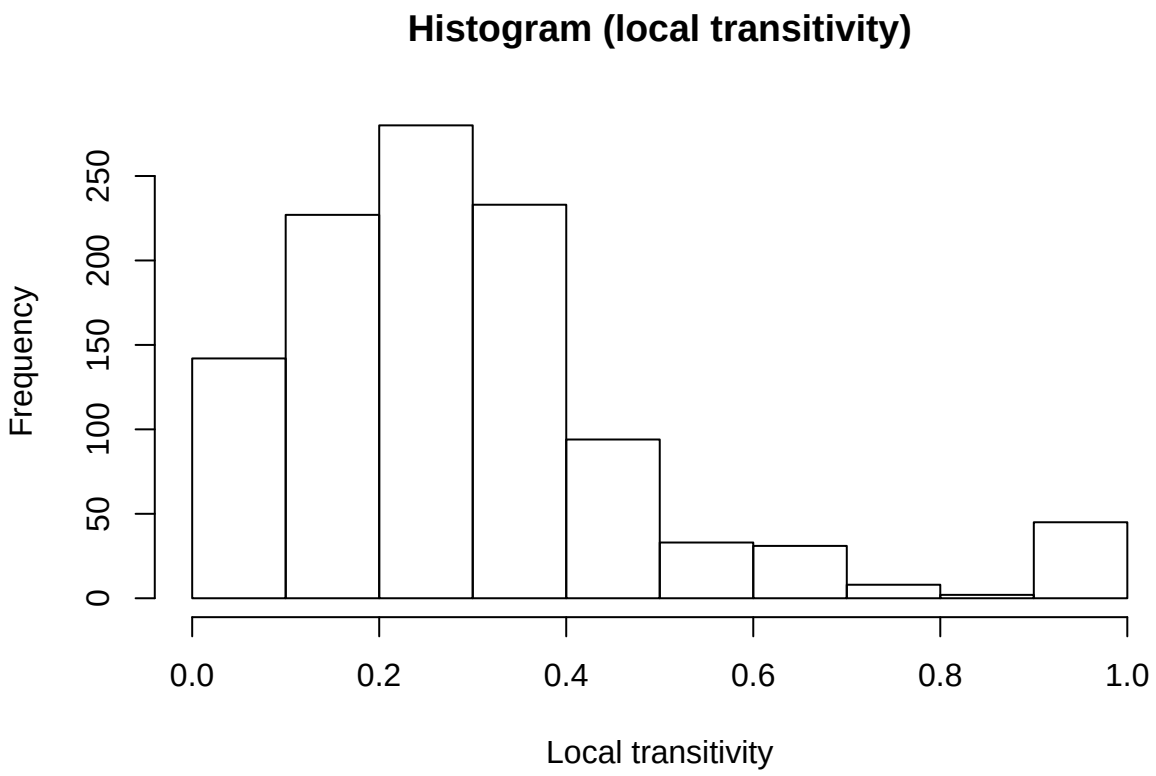
```
g$trans_global <- transitivity(g, type="global")
g$trans_localavg <- transitivity(g, type="localaverage")
V(g)$trans <- transitivity(g, type="local")

g$trans_global
```

```
## [1] 0.2259585
```

```
g$trans_localavg
```

```
## [1] 0.3600287
```

```
hist(V(g)$trans, main="Histogram (local transitivity)", xlab="Local transitivity")
```



There are other ways to investigate triadic structures in a network. The function `triad_census`, for example, counts a variety of possible triadic constellations in a network. Those cannot only tell us something about the degree of clustering in a network but also about other interesting properties, like hierarchical tendencies.

Density

The *density* of a network keeps track of the relative fraction of links that are present. Given an undirected graph without loops g (i.e. a simple graph) with n nodes and e edges, the maximum number of edges equals

$$\frac{n \times (n-1)}{2}$$

Consequently, the density is

$$\frac{2 \times e}{n \times (n-1)}$$

where e is the number of edges in a graph g.

Accordingly, given a directed graph without loops g with n nodes and e edges, the maximum number of edges equals

$$n \times (n-1)$$

and the density is

$$\frac{e}{n \times (n-1)}$$

```
g$density_directed <- edge_density(g)
g$density_undirected <- edge_density(as.undirected(g, mode="collapse"))

g$density_directed
```

```
## [1] 0.01274813
```

```
g$density_undirected
```

```
## [1] 0.02240389
```

Paths and distances

The lenght of paths, especially the shortest paths, between nodes, tells us something about the connectedness of the network. Using a couple of igraph functions, we can quickly get an overview of the distances between the nodes in our network.

Using the function `distance table`, we can get the number of nodes connected by various shortes path lentgths. Note that for the undirected version, we get nodes which are only weakly connected to the network, which the function reports as *unconnected*.

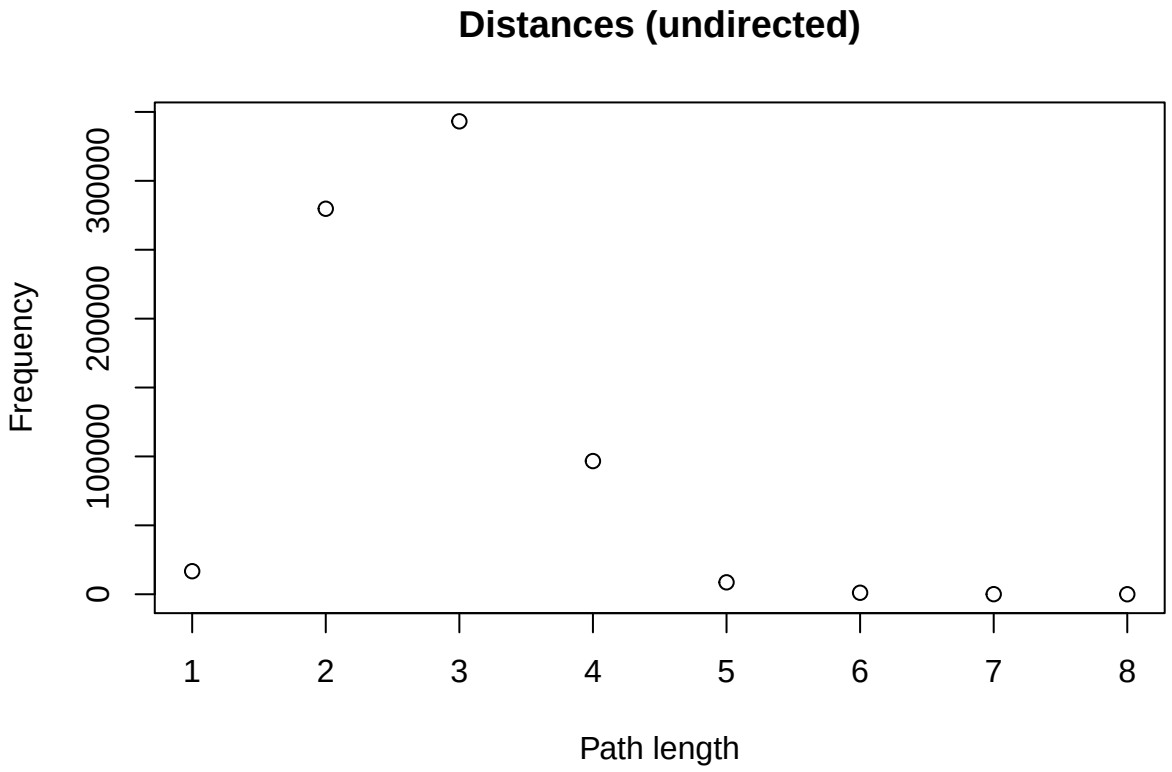
```
distance_table(g, directed=FALSE)
```

```
## $res
## [1]  16714 279748 343167  96629   8639   1079     54      1
##
## $unconnected
## [1] 0
```

```
distance_table(g, directed=TRUE)
```

```
## $res
## [1]  19021 193830 348198 275702 107394  25602  10092   1371     37
##
## $unconnected
## [1] 510815
```

```
plot(
  distance_table(g, directed=FALSE)$res,
  main="Distances (undirected)",
  xlab="Path length",
  ylab="Frequency"
)
```



We can also compute the mean distance directy:

```
g$mean_dist_d <- mean_distance(g, directed=TRUE)
g$mean_dist_u <- mean_distance(g, directed=FALSE)
g$mean_dist_d
```

```
## [1] 3.390186
```

```
g$mean_dist_u
```

```
## [1] 2.73753
```

If we are interested in the shortest path between two nodes or all shortest paths:

```
# The shortest path frome node 1 to 5
shortest_paths(g, 1, 5)$vpath
```

```
## [[1]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1  48 358   5
```

```
# All shortest paths from 1 to 5
all_shortest_paths(g, from=1, to=5, mode="all")$res
```

```
## [[1]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1  48 358   5
##
## [[2]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1  48 358   5
##
## [[3]]
```

```
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 127 358   5
##
## [[4]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 192 358   5
##
## [[5]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 203 358   5
##
## [[6]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 203 358   5
##
## [[7]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 204 358   5
##
## [[8]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 340 358   5
##
## [[9]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 376 358   5
##
## [[10]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 389 358   5
##
## [[11]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 389 358   5
##
## [[12]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 497 358   5
##
## [[13]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 498 358   5
##
## [[14]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 500 358   5
##
## [[15]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 500 358   5
##
## [[16]]
## + 4/1222 vertices, from 9f7fd0a:
## [1]   1 517 358   5
```

An interesting path is the *diameter*, i.e. the longest shortest path between two nodes in the network.

```
g$diameter_u <- diameter(g, directed=FALSE)
g$diameter_d <- diameter(g, directed=TRUE)
g$diameter_u
```

```
## [1] 8
```

```
g$diameter_d
```

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

Similarly, the *girth* of a network is the length of the smallest circle. Note that the `girth` function in igraph automatically searches for undirected circles, even in a directed network.

```
g$girth <- girth(g, circle = FALSE)$girth
g$girth
```

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

```
# We can also inspect the circle itself
girth(g, circle = TRUE)$circle
```

```
## + 3/1222 vertices, from 9f7fd0a:
## [1]  2  1 48
```

Homophily

Another interesting network property is the tendency of nodes to be connected to others who are similar with regards to a specific variable. This tendency is called *homophily*. Homophily is one of the fundamental mechanisms that have been found to influence the structure of networks. It is also referred to as *assortativity* and can be computed in igraph using the `assortativity` function. There's also the `assortativity_degree` function, which simply uses the degrees as variable.

```
g$assortativity <- assortativity(g, V(g)$value, directed=FALSE)
g$assortativity_deg <- assortativity_degree(g, directed=FALSE)
g$assortativity
```

```
## [1] 0.8228798
```

```
g$assortativity_deg
```

```
## [1] -0.1961797
```

Exploring the results

Now that we have computed some of the most commonly used metrics, which we have stored in the igraph object, we can start exploring the results.

First, let's have a look at the network object:

```
g
```

```
## IGRAPH 9f7fd0a D--- 1222 19021 --
## + attr: centr.deg (g/n), centr.eig (g/n), centr.clo (g/n),
## | centr.bet (g/n), reciprocity (g/n), mut (g/n), asym (g/n),
## | trans_global (g/n), trans_localavg (g/n), density_directed
## | (g/n), density_undirected (g/n), mean_dist_d (g/n), mean_dist_u
```

```
## | (g/n), diameter_u (g/n), diameter_d (g/n), girth (g/n),
## | assortativity (g/n), assortativity_deg (g/n), id (v/n), label
## | (v/c), value (v/c), source (v/c), degree (v/n), centr.deg (v/n),
## | centr.eig (v/n), centr.clo (v/n), centr.bet (v/n), trans (v/n)
## + edges from 9f7fd0a:
##  [1] 1-> 21 1-> 48 1-> 72 1->127 1->253 1->286 1->340 1->376 1->450 1->497
## + ... omitted several edges
```

Network level measures:

```
as.matrix(graph_attr(g))
```

```
##                     [,1]
## centr.deg          0.1786347
## centr.eig          0.9227435
## centr.clo          0.2943235
## centr.bet          0.1093113
## reciprocity        0.242574
## mut                2307
## asym               14407
## trans_global       0.2259585
## trans_localavg     0.3600287
## density_directed   0.01274813
## density_undirected 0.02240389
## mean_dist_d        3.390186
## mean_dist_u        2.73753
## diameter_u         8
## diameter_d         9
## girth              3
## assortativity      0.8228798
## assortativity_deg  -0.1961797
```

If we want to dive deeper into the results, we'll most likely need the node attributes in a more convenient format. Let's put them into a data frame:

```
data <- as_data_frame(g, what="vertices")
head(data)
```

```
##   id                      label value                    source degree
## 1  1       100monkeystyping.com     0                 Blogarama     27
## 2  2 12thharmonic.com/wordpress     0               BlogCatalog     48
## 3  5      750volts.blogspot.com     0                 Blogarama      4
## 4  6      95theses.blogspot.com     0                 Blogarama      1
## 5  7  abbadabbaduo.blogspot.com     0 Blogarama,LeftyDirectory      1
## 6  8 aboutpolitics.blogspot.com     0   Blogarama,eTalkingHead     38
##      centr.deg   centr.eig centr.clo    centr.bet     trans
## 1 0.0221130221 0.098512431 0.4032365 3.297375e-04 0.3732194
## 2 0.0393120393 0.182550536 0.3937440 2.394211e-03 0.3377660
## 3 0.0032760033 0.005843143 0.3485584 5.242911e-05 0.1666667
## 4 0.0008190008 0.001353241 0.3028274 0.000000e+00       NaN
## 5 0.0008190008 0.005725731 0.3088012 0.000000e+00       NaN
## 6 0.0311220311 0.130519949 0.3703367 4.719844e-04 0.1849218
```

From here on, we are free to work with the results. We could, for example, compare the two groups:

```
data %>%
  group_by(value) %>%
```

```
  summarize(
    degree.mean=mean(degree),
    degree.median=median(degree),
    centr.bet.mean=mean(centr.bet),
    centr.bet.median=median(centr.bet),
    centr.clo.mean=mean(centr.clo),
    centr.clo.median=mean(centr.clo),
    centr.eig.mean=mean(centr.eig),
    centr.eig.median=mean(centr.eig),
  )
```

```
## # A tibble: 2 x 9
##   value degree.mean degree.median centr.bet.mean centr.bet.median
##   <chr>       <dbl>         <dbl>          <dbl>            <dbl>
## 1 0            31.5            11        0.00141        0.0000809
## 2 1            30.8            16        0.00144        0.000106
## # ... with 4 more variables: centr.clo.mean <dbl>, centr.clo.median <dbl>,
## #   centr.eig.mean <dbl>, centr.eig.median <dbl>
```

Another common operation is to extract the most central nodes:

```
n = 10

# Get the top n nodes for all groups by betwenness
data %>%
  top_n(n, wt = centr.bet) %>%
  arrange(desc(centr.bet))
```

```
##      id                     label value
## 1   855          blogsforbush.com     1
## 2   155              dailykos.com     0
## 3   963          drudgereport.com     1
## 4  1051           instapundit.com     1
## 5   641     talkingpointsmemo.com     0
## 6  1437         truthlaidbear.com     1
## 7    55       atrios.blogspot.com     0
## 8   729     washingtonmonthly.com     0
## 9   979          etalkinghead.com     1
## 10  454 newleftblogs.blogspot.com     0
##                                        source degree  centr.deg
## 1                       BlogPulse,CampaignLine    467 0.38247338
## 2  LeftyDirectory,LabeledManually,CampaignLine    383 0.31367731
## 3                               LabeledManually    243 0.19901720
## 4                                     BlogPulse    362 0.29647830
## 5                    BlogPulse,LeftyDirectory    282 0.23095823
## 6               LeftyDirectory,LabeledManually    204 0.16707617
## 7        BlogPulse,LeftyDirectory,CampaignLine    350 0.28665029
## 8                    BlogPulse,LeftyDirectory    256 0.20966421
## 9                                     Blogarama     75 0.06142506
## 10                              LabeledManually    179 0.14660115
##     centr.eig centr.clo  centr.bet      trans
## 1  0.26826092 0.4784483 0.11064608 0.02058615
## 2  0.82598888 0.5186916 0.08712083 0.07261493
## 3  0.18397229 0.4851013 0.06483173 0.03812536
## 4  0.54330880 0.5193535 0.05307687 0.07780720
```

```
## 5  0.65204349 0.5030902 0.04520570 0.10797304
## 6  0.24283517 0.4810875 0.03987666 0.07799672
## 7  1.00000000 0.4983673 0.03706263 0.08759722
## 8  0.75241567 0.4901646 0.02871522 0.11712623
## 9  0.06329852 0.4296270 0.02770359 0.04036036
## 10 0.51953202 0.4465984 0.02768066 0.13501977
```

```
# Get the topn nodes by group
data %>%
  group_by(value) %>%
  top_n(n, wt=centr.bet)
```

```
## # A tibble: 20 x 10
## # Groups:   value [2]
##       id label value source degree centr.deg centr.eig centr.clo centr.bet
##    <dbl> <chr> <chr> <chr>   <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
##  1    55 atri~ 0     BlogP~    350    0.287     1          0.498    0.0371
##  2   119 buzz~ 0     BlogP~    112    0.0917    0.230      0.468    0.0134
##  3   155 dail~ 0     Lefty~    383    0.314     0.826      0.519    0.0871
##  4   170 demo~ 0     Label~     95    0.0778    0.166      0.446    0.0157
##  5   323 juan~ 0     BlogP~    174    0.143     0.461      0.455    0.0138
##  6   387 madk~ 0     Lefty~    170    0.139     0.586      0.473    0.0147
##  7   454 newl~ 0     Label~    179    0.147     0.520      0.447    0.0277
##  8   641 talk~ 0     BlogP~    282    0.231     0.652      0.503    0.0452
##  9   729 wash~ 0     BlogP~    256    0.210     0.752      0.490    0.0287
## 10   741 wonk~ 0     Label~    129    0.106     0.307      0.465    0.0181
## 11   855 blog~ 1     BlogP~    467    0.382     0.268      0.478    0.111
## 12   963 drud~ 1     Label~    243    0.199     0.184      0.485    0.0648
## 13   979 etal~ 1     Bloga~     75    0.0614    0.0633     0.430    0.0277
## 14  1051 inst~ 1     BlogP~    362    0.296     0.543      0.519    0.0531
## 15  1101 lash~ 1     Bloga~    218    0.179     0.245      0.474    0.0186
## 16  1112 litt~ 1     Label~    208    0.170     0.245      0.495    0.0231
## 17  1153 mich~ 1     BlogP~    228    0.187     0.269      0.484    0.0265
## 18  1245 powe~ 1     BlogP~    235    0.192     0.286      0.475    0.0223
## 19  1437 trut~ 1     Lefty~    204    0.167     0.243      0.481    0.0399
## 20  1479 wizb~ 1     BlogP~    219    0.179     0.253      0.468    0.0208
## # ... with 1 more variable: trans <dbl>
```

Finally, we should save our results:

```
write_graph(g, "./graphs/polblogs-analyzed.gml", format="gml")
write.csv(data, "./data/data.csv")
```