Lots of research has been done in this field, one particular popular result being the **6-Handshake-Rule**. It states that most people living on this planet are connected by a chain of six handshakes or less. In the general setting of *graphs,* in which *edges connect nodes*, this is often referred to as the so-called *small-world-effect*. That is to say, the typical number of edges needed to get from *node A* to *node B* grows logarithmically in population size (i.e., # nodes). Note that, up until now, no geographic distance has been included in our consideration, which seems inadequate as it plays a significant role in social networks.

When analyzing data from social networks such as Facebook or Instagram, three observations are especially striking:

* Individuals who are geographically farther away from each other are less likely to connect, i.e., people from the same city are more likely to connect.
* Few individuals have extremely many connections. Their number of connections follows a heavy-tailed Pareto distribution. Such individuals interact as *hubs* in the network. That could be a celebrity or just a *really* popular kid from school.
* Connected individuals tend to share a set of other individuals they are both connected to (e.g., „friend cliques“). This is called the *clustering property*.

**A model that *explains* these observations**

Clearly, due to the characteristics of social networks mentioned above, only a model that includes geographic distances of the individuals makes sense. Also, to account for the occurrence of hubs, research has shown that reasonable models attach a random weight to each node (which can be regarded as the social attractiveness of the respective individual). A model that accounts for all three properties is the following: First, randomly place nodes in space with a certain intensity , which can be done with a Poisson process. Then, with an independent uniformly distributed weight attached to each node , every two nodes get connected by an edge with a probability

where is the dimension of the model (here: as we’ll simulate the model on the plane), model parameter controls the impact of the weights, model parameter 0" title="Rendered by QuickLaTeX.com" height="17" width="44" style="vertical-align: -4px;"/> squishes the overall input to the profile function , which is a monotonously decreasing, normalized function that returns a value between and .

That is, of course, what we want because its output shall be a probability. Take a moment to go through the effects of different and on . A higher yields a smaller input value for and thereby a higher connection probability. Similarly, a high entails a lower (as ) and thus a higher connection probability. So much about the theory. Now that we have a model, we can use this to generate synthetic data that should look similar to real-world data. So let’s simulate!

**Obtain data through simulation**

From here on, the simulation is pretty straight forward. Don’t worry about specific numbers at this point.

library(tidyverse)

library(fields)

library(ggraph)

library(tidygraph)

library(igraph)

library(Matrix)

# Create a vector with plane dimensions. The random nodes will be placed on the plane.

plane <- c(1000, 1000)

poisson\_para <- .5 \* 10^(-3) # Poisson intensity parameter

beta <- .5 \* 10^3

gamma <- .4

# Number of nodes is Poisson(gamma)\*AREA - distributed

n\_nodes <- rpois(1, poisson\_para \* plane[1] \* plane[2])

weights <- runif(n\_nodes) # Uniformly distributed weights

# The Poisson process locally yields node positions that are completely random.

x = plane[1] \* runif(n\_nodes)

y = plane[2] \* runif(n\_nodes)

phi <- function(z) { # Connection function

pmin(z^(-1.8), 1)

}

What we need next is some information on which nodes are connected. That means, we need to first get the connection probability by evaluating for each pair of nodes and then flipping a biased coin, accordingly. This yields a encoding, where means that the two respective nodes are connected and that they’re not. We can gather all the information for all pairs in a matrix that is commonly known as the *adjacency matrix*.

# Distance matrix needed as input

dist\_matrix <-rdist(tibble(x,y))

weight\_matrix <- outer(weights, weights, FUN="\*") # Weight matrix

con\_matrix\_prob <- phi(1/beta \* weight\_matrix^gamma\*dist\_matrix^2)# Evaluation

con\_matrix <- Matrix(rbernoulli(1,con\_matrix\_prob), sparse=TRUE) # Sampling

con\_matrix <- con\_matrix \* upper.tri(con\_matrix) # Transform to symmetric matrix

adjacency\_matrix <- con\_matrix + t(con\_matrix)

**Visualization with ggraph**

we also have lots of love for tidyverse, and **ggraph** (spoken „g-giraffe“) as an extension of ggplot2 proves to be a comfortable alternative for non-interactive graph plots, especially when you’re already familiar with the *grammar of graphics*. In combination with **tidygraph**, which lets us describe a graph as two tidy data frames (one for the nodes and one for the edges), we obtain a full-fledged tidyverse experience. Note that tidygraph is based on a graph manipulation library called *igraph* from which it inherits all functionality and „exposes it in a tidy manner“. So before we get cracking with the visualization in ggraph, let’s first tidy up our data with tidygraph!

**Make graph data tidy again!**

Let’s attach some new columns to the node dataframe which will be useful for visualization. After we created the tidygraph object, this can be done in the usual dplyr fashion after using activate(nodes)and activate(edges)for accessing the respective dataframes.

# Create Igraph object

graph <- graph\_from\_adjacency\_matrix(adjacency\_matrix, mode="undirected")

# Make a tidygraph object from it. Igraph methods can still be called on it.

tbl\_graph <- as\_tbl\_graph(graph)

hub\_id <- which.max(degree(graph))

# Add spacial positions, hub distance and degree information to the nodes.

tbl\_graph <- tbl\_graph %>%

activate(nodes) %>%

mutate(

x = x,

y = y,

hub\_dist = replace\_na(bfs\_dist(root = hub\_id), Inf),

degree = degree(graph),

friends\_of\_friends = replace\_na(local\_ave\_degree(), 0),

cluster = as.factor(group\_infomap())

)

Tidygraph supports most of igraphs methods, either directly or in the form of wrappers. This also applies to most of the functions used above. For example breadth-first search is implemented as the bfs\_\* family, wrapping igraph::bfs(), the group\_graphfamily wraps igraphs clustering functions and local\_ave\_degree() wraps igraph::knn().

**Let’s visualize!**

GGraph is essentially built around three components: Nodes, Edges and Layouts. Nodes that are connected by edges compose a graph which can be created as an igraph object. Visualizing the igraph object can be done in numerous ways: Remember that nodes usually are not endowed with any coordinates. Therefore, arranging them in space can be done pretty much arbitrarily. In fact, there’s a specific research branch called *graph drawing* that deals with finding a good layout for a graph for a given purpose.

Usually, the main criteria of a good *layout* are aesthetics (which is often interchangeable with clearness) and capturing specific graph properties. For example, a layout may force the nodes to form a circle, a star, two parallel lines, or a tree (if the graph’s data allows for it). Other times you might want to have a layout with a minimal number of intersecting edges. Fortunately, in ggraph all the [layouts from igraph](https://igraph.org/r/doc/layout_.html) can be used.

We start with a basic plot by passing the data and the layout to ggraph(), similar to what you would do with ggplot() in ggplot2. We can then add layers to the plot. Nodes can be created by using geom\_node\_point()and edges by using geom\_edge\_link(). From then on, it’s full-on ggplot2-style.

# Add coord\_fixed() for fixed axis ratio!

basic <- tbl\_graph %>%

ggraph(layout = tibble(V(.)$x, V(.)$y)) +

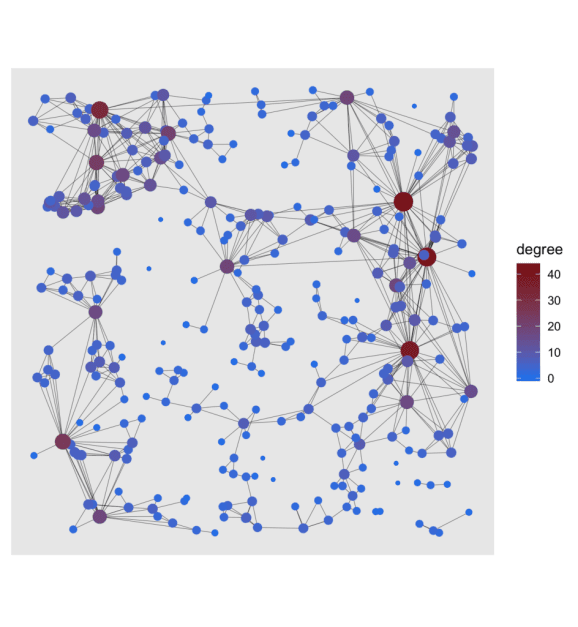
geom\_edge\_link(width = .1) +

geom\_node\_point(aes(size = degree, color = degree)) +

scale\_color\_gradient(low = "dodgerblue2", high = "firebrick4") +

coord\_fixed() +

guides(size = FALSE)



To see more clearly what nodes are essential to the network, the degree, which is the number of edges a node is connected with, was highlighted for each node. Another way of getting a good overview of the graph is to show a visual decomposition of the components. Nothing easier than that!

cluster <- tbl\_graph %>%

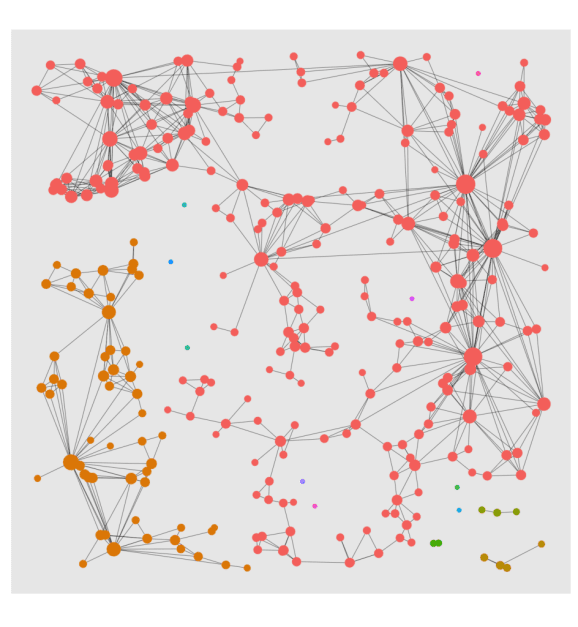
ggraph(layout = tibble(V(.)$x, V(.)$y)) +

geom\_edge\_link(width = .1) +

geom\_node\_point(aes(size = degree, color = cluster)) +

coord\_fixed() +

theme(legend.position = "none")



Wouldn’t it be interesting to visualize the reach of a hub node? Let’s do it with a facet plot:

# Copy of tbl\_graph with columns that indicate weather in n - reach of hub.

reach\_graph <- function(n) {

tbl\_graph %>%

activate(nodes) %>%

mutate(

reach = n,

reachable = ifelse(hub\_dist <= n, "reachable", "non\_reachable"),

reachable = ifelse(hub\_dist == 0, "Hub", reachable)

)

}

# Tidygraph allows to bind graphs. This means binding rows of the node and edge dataframes.

evolving\_graph <- bind\_graphs(reach\_graph(0), reach\_graph(1), reach\_graph(2), reach\_graph(3))

evol <- evolving\_graph %>%

ggraph(layout = tibble(V(.)$x, V(.)$y)) +

geom\_edge\_link(width = .1, alpha = .2) +

geom\_node\_point(aes(size = degree, color = reachable)) +

scale\_size(range = c(.5, 2)) +

scale\_color\_manual(values = c("Hub" = "firebrick4",

"non\_reachable" = "#00BFC4",

"reachable" = "#F8766D","")) +

coord\_fixed() +

facet\_nodes(~reach, ncol = 4, nrow = 1, labeller = label\_both) +

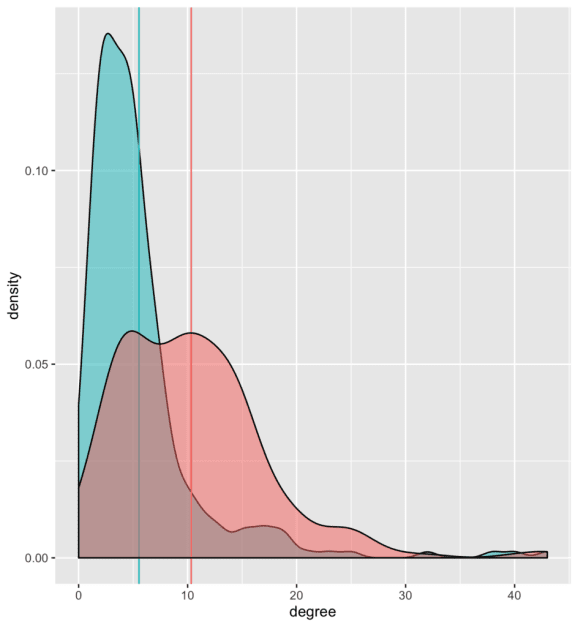
theme(legend.position = "none")



**A curious observation**

At this point, there are many graph properties (including the three above but also cluster sizes and graph distances) that are worth taking a closer look at, but this is beyond the scope of this blogpost. However, let’s look at one last thing. Somebody just recently told me about a very curious fact about social networks that seems paradoxical at first: Your average friend on Facebook (or Instagram) has way more friends than the average user of that platform.

It sounds odd, but if you think about it for a second, it is not too surprising. Sampling from the pool of *your* friends is very different from sampling from all users on the platform (entirely at random). It’s exactly those very prominent people who have a much higher probability of being among your friends. Hence, when calculating the two averages, we receive very different results.



As can be seen, the model also reflects that property: In the small excerpt of the graph that we simulate, the average node has a degree of around 5 (blue intercept). The degree of connected nodes is over 10 on average (red intercept).

**Conclusion**

In the first part, I introduced a model that describes the features of real-life data of social networks well. In the second part, we obtained artificial data from that model and used it to create an igraph object (by means of the adjacency matrix). The latter can then be transformed into a tidygraph object, allowing us to easily make manipulation on the node and edge tibble to calculate any graph statistic (e.g., the degree) we like. Further, the tidygraph object is then used for conveniently visualizing the network through Ggraph.

I hope that this post has sparked your interest in network modeling and has given you an idea of how seamlessly graph manipulation and visualization with Tidygraph and Ggraph merge into the usual tidyverse workflow. Have a wonderful day!