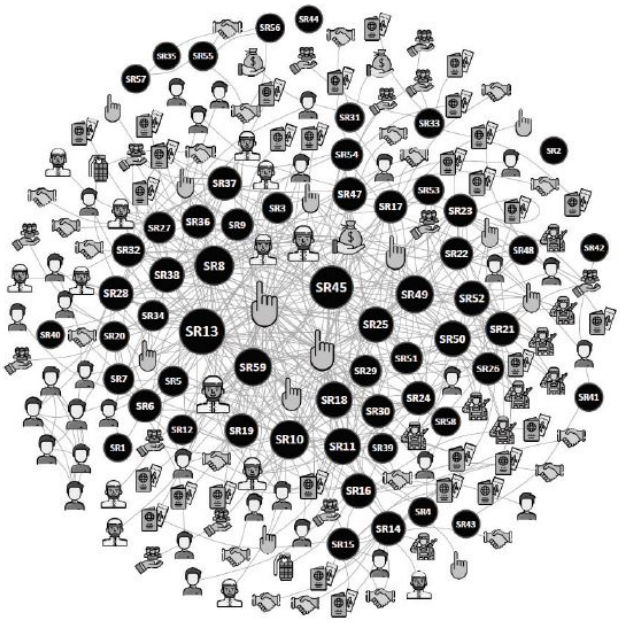
Advanced Network Visualization with R and Gephi

Annika Hamachers
German Police University, Münster
annika.hamachers@dhpol.de



Abstract

`R` in general, and the R package `igraph` in particular are powerful tools for network *analysis*. However, producing beautiful network *visualizations* from R that are suitable to actually be published in academic papers or presentations can be quite a hassle. This tutorial demonstrates how to create visually appealing network graphs by focusing on solutions for two common problems: 1) incorporating compelling and easily generated network layouts from `Gephi` and 2) adding custom icons to be able to distinguish between a large variety of attributes even in black and white publications.

Contents

1 Set-up

This paper is neither an introduction to network visualization with R nor to Gephi. It rather focuses on the two specific problems mentioned in the abstract and assumes that the readers already have some experience with both tools. If however, you are completely new to this topic, I strongly recommend you read this awesome tutorial by Katherine Ognyanova and check out her blog www.kateto.net - this will definitely get *you* set up.

1.1 Preparing the data

Disclaimer: The data we toy around with for the purpose of this tutorial stem from a project on online radicalization – particularly on the social ties so-called foreign fighters had with members of the Salafist milieu in Germany. If you are curious about the results, follow this link to request the full-text of the final publication.

Luckily, preparing our data for what we are up to, is quite easy because R and Gephi can work with (almost) equally presented material: In both cases we need to prepare two spreadsheets – one containing information on all the nodes of the network and one containing the edge information.

The node list (see Figure 1) may hold as many variables describing the nodes as we like (e.g. their gender, their age etc.). However, two columns are mandatory: a numeric identifier variable called 'Id' and the 'Label' (aka the name) of the node we want later to be displayed in the graphs.

	A	B	C	D	E
1	Id	Label	Geschlecht	Rolle	Rolle_aggreg
2	6	SR6	weiblich	Rückkehrer	SR
3	7	SR7	männlich	Rückkehrer	SR
4	8	SR8	männlich	Rückkehrer	SR
5	20	SR20	weiblich	Rückkehrer	SR
6	21	SR21	männlich	Rückkehrer	SR
7	22	SR22	männlich	Rückkehrer	SR
8	23	SR23	männlich	Rückkehrer	SR
9	24	SR24	männlich	Rückkehrer	SR
10	25	SR25	männlich	Rückkehrer	SR
11	26	SR26	männlich	Rückkehrer	SR
12	40	SR40	weiblich	Rückkehrer	SR

Figure 1: The node list

The edge list (see Figure 2) works quite similar. We can add as many variables describing the relationship between two nodes as we like, as long as we provide two columns that identify which nodes the edges are supposed to connect. For R this means telling the system 'from' which node 'to' which other node a line is to be drawn (even if it is an undirected graph). In Gephi however, the associated columns must be called 'Source' and 'Target'. What is important in both cases is, that this will not work providing the node labels, these two columns have to contain the right node 'Id's matching the ones in the node list.

	A	B	C	D
1	from	to	Weight	Kommunikat
2	1	7	1	offline
3	3	5	1	offline
4	3	8	1	offline
5	3	135	1	offline
6	3	145	1	offline
7	3	122	1	unklar
8	4	61	1	offline
9	4	64	1	offline
10	5	3	1	offline

(a) Declaration for R

	A	B	C	D
1	Source	Target	Weight	Kommunikat
2	1	7	1	offline
3	3	5	1	offline
4	3	8	1	offline
5	3	135	1	offline
6	3	145	1	offline
7	3	122	1	unklar
8	4	61	1	offline
9	4	64	1	offline
10	5	3	1	offline

(b) Declaration for Gephi

Figure 2: The edge list

We can save these spreadsheets either in .xlsx or .csv format, however, for convenience, I'd recommend saving the Gephi input as .xlsx and the R input as .csv – in my case both times with UTF-8 encoding since I use data that contain non-ASCII characters.

1.2 Getting the R session ready

Now, that the raw data are ready, we also need to set up R itself. The first thing we would want to do therefore is to load all necessary packages: We rely on `readr` for the data import, `igraph` for the basic network visualization, `extrafont` for better readability, `xml2` for the Gephi import, and `png` for incorporating our custom icons: If those packages are not yet in your R library, please run the `install.packages()` command first for each package.

```
#install.packages('readr')
#install.packages('igraph')
#install.packages('extrafont')
#install.packages('xml2')
#install.packages('png')
library(readr) # package to read in the CSV
library(igraph) #R's most powerful network tool
library(extrafont) #access beautiful fonts
library(xml2) #XML importer, necessary for parsing Gephi output
library(png) #needed to transform pngs into rasters for the net
```

The `extrafont` package we use right away, so we can access all the fonts that are installed on our device. Choosing a custom font can enhance the readability/quality of network graphs tremendously!

The very first time, we want to grant R access to our font library, we'll have to run `font_import()`. This registers these permanently, so you will never have to run this again and can comment it out as I have done.

With `fonts()` you can check which fonts are available to you.

LaTeX users like myself, however should include one additional step because they might find it difficult to embed fonts other than from the 'Computer Modern' family. So, just run `font_install('fontcm')` to also include those.

```
#font_import()
# install.packages('fontcm')
font_install('fontcm') # import 'CM' font families for LaTeX
fonts() # See what font families are available to you now.

##   [1] "Agency FB"
##   [2] "Algerian"
##   [3] "Arial Black"
##   [4] "Arial"
##   [5] "Arial Narrow"
##   [6] "Arial Rounded MT Bold"
...
```

Then load the fonts matching your output device (e.g. 'win' on a Windows machine):

```
loadfonts(device = "win") # use device = "pdf" for pdf plot output.
```

2 Creating Basic (aka ugly) Networks in R

Building network objects with plain igraph is easy – though hardly ever results in something pretty right away.

2.1 Creating a Network Object

Creating the basic network from the prepared data is pretty straight forward:

First, we read in the nodes and the edges separately with `read.csv()` (remembering to keep the attribute names with `header=TRUE` and choosing the right separator, here `;`, and encoding, in our case `UTF-8-BOM`):

```
links <- read.csv("RData/Edges_neu.csv", header=TRUE, sep =";",
                  as.is=T, fileEncoding="UTF-8-BOM")
nodes <- read.csv("RData/Nodes_neu.csv", header=TRUE, sep =";",
                  as.is=T, fileEncoding="UTF-8-BOM")
```

Then, we pass the edge dataframe to a command called `graph_from_data_frame()`. We tell this function that the associated nodes (here called 'vertices') are to be found in the nodes dataframe and also provide the information that our edges are to be treated as undirected (`directed=FALSE`).

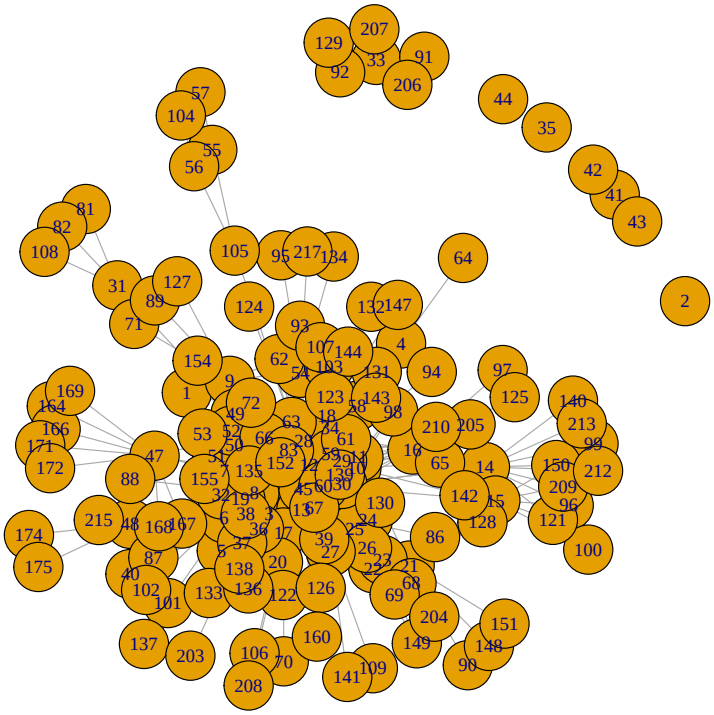
Calling the created network object, we simply named 'net', shows us that everything worked out fine:

```
net <- graph_from_data_frame(d=links, vertices=nodes, directed=FALSE)
net

## IGRAPH 022699e UN-- 152 478 --
## + attr: name (v/c), Label (v/c), Geschlecht (v/c),
## | Rolle_aggregiert (v/c), Kern (v/c), Weight (e/n),
## | Kommunikation (e/c), Unterst (e/c)
## + edges from 022699e (vertex names):
##  [1] 9 --49  13--126 13--135 13--160 14--99  14--121 15--86
##  [8] 15--121 17--83  19--51  21--204 22--204 23--204 50--135
## [15] 54--134 55--104 56--104 57--104 1 --7   3 --5   3 --8
## [22] 3 --45  3 --122 3 --135 4 --18  4 --61  4 --64  5 --13
## [29] 5 --45  5 --135 5 --137 5 --6   6 --7   6 --8   6 --19
## [36] 6 --20  6 --51  7 --8   7 --20  5 --8   8 --12  8 --17
## + ... omitted several edges
```

The output tells us that we successfully created an `IGRAPH` which is undirected (UN) and has 152 nodes and 478 edges. It then lists the attributes of the network – each time indicating if it applies to the nodes (v) or the edges (e) and if it is a character (c) or numeric (n) variable. Finally the output starts to list all edges.

But how does this network look like? We access the default network graph by simply calling R's `plot()` function:

```
plot(net)
```



2.2 Modifying a Network Graph within R

This default graph, however, is not a pretty sight: The nodes display their IDs instead of comprehensive labels and cling together so much that they seem to sit on top of each other. Let's try to use R's built-in features to tidy things up a little bit!

Since we would like to make the important players in the network stand out immediately, we start by creating a variable that saves each node's degree – which is just the number of connections it has to other nodes in the network.

```
degrees <- degree(net)# save the number of edges for each node
```

From now on, we can use this variable to make the size of the nodes proportional to this value.

Next, we would like to adjust the network layout itself (aka the coordinates underlying the distribution of the nodes). In R, we specify those coordinates by adding a `layout` parameter to the `plot()` function. In igraph, there are quite a lot of nice layouts to choose from that come handy in different situations (for large networks, for small ones, for ego networks etc.):

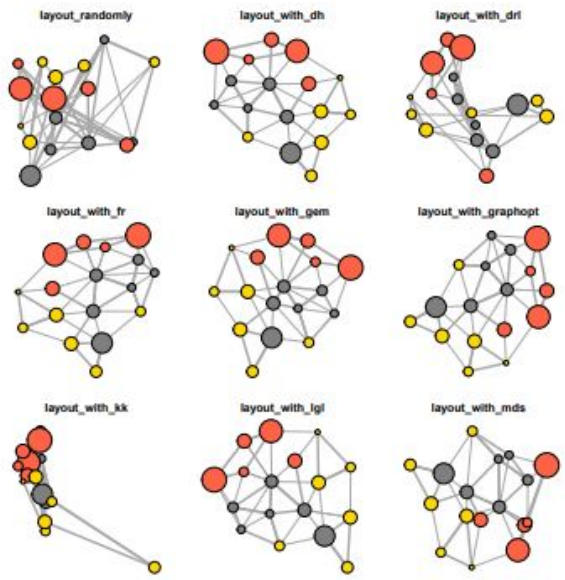


Figure 3: Examples of igraph layout options; source: https://kateto.net/

The default algorithm, R automatically uses, if no layout is specified, is `layout_with_fr`. This is igraph's implementation of the Fruchterman-Reingold algorithm. This force-directed algorithm usually is a good choice because it reduces edge crossing to a minimum, reflects inherent symmetries, and makes edge lengths uniform (for technical details please refer to the original paper (Fruchterman & Reingold 1991)). The result is usually an easy to interpret graph that displays nodes with similar connections close to each other.

However, since most layouting algorithms initialize weights at random, we'd get quite a different graph, each time we would run e.g. `plot(net, layout=layout_with_fr)`. To ensure, the coordinates stay exactly the same, we therefore first save the layout into its own variable:

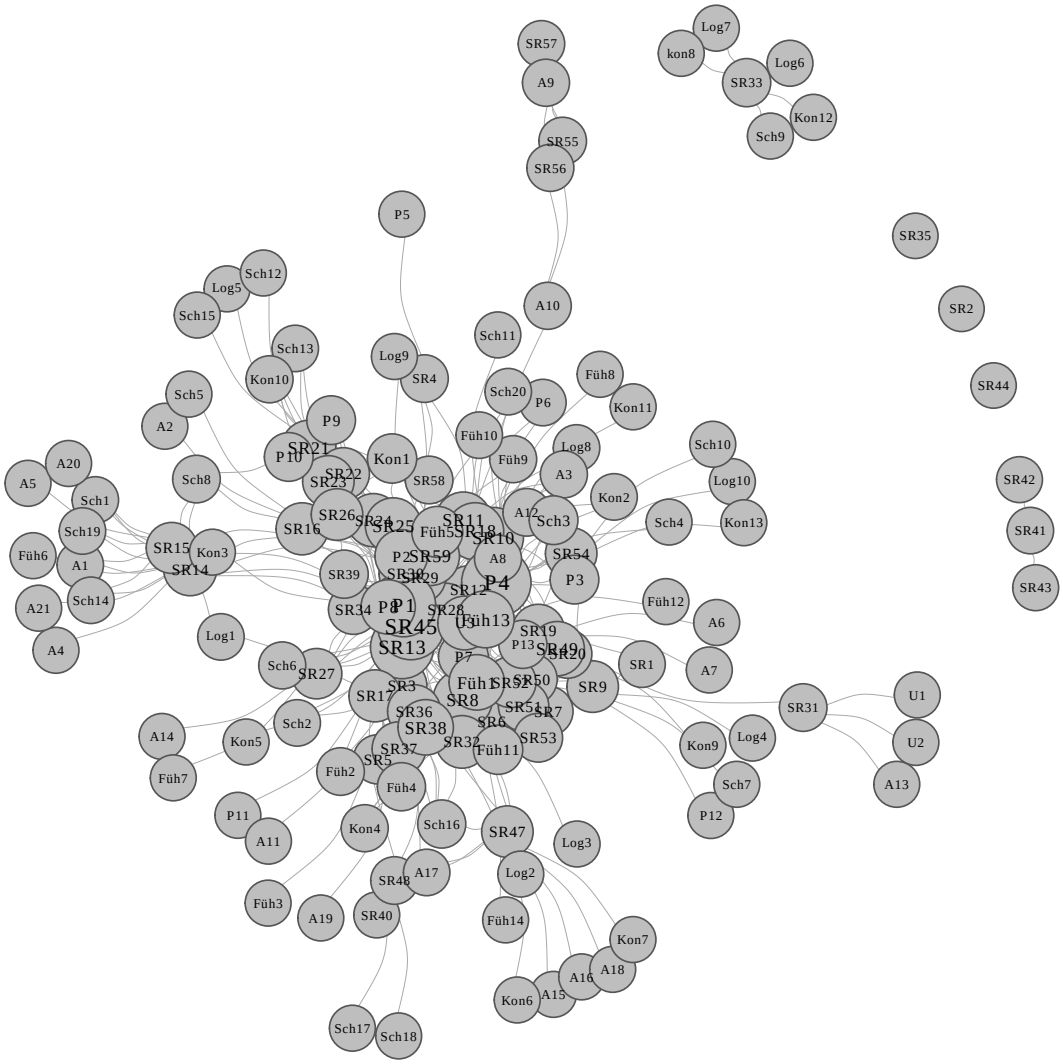
```
fr <- layout_with_fr(net)
```

Depending on the layout, you'd like to use, it can proof useful to normalize it by setting the boudaries for the x and y coordinates to -1 to 1:

```
fr_norm <- norm_coords(fr, ymin=-1, ymax=1, xmin=-1, xmax=1)
```

Making sure that the normalized graph does not rescale (`rescale=F`), we can now use this layout along with the degree variable for the `vertex.size` parameter. At the same time, we add some further styling. We tell R to slightly curve the edges (`edge.arrow.size=.2`), make them thinner(`edge.width=.5`), label the nodes according to the actual 'Label' column in the dataframe (`vertex.label=V(net)$Label`), make the label color "black", the node itself "gray", adjust the label size (`vertex.label.cex`) also proportionally to the degree, change the font (`vertex.label.family`) to something we made available with the `extrafont` package, and make this font bold (`vertex.label.font=2`):

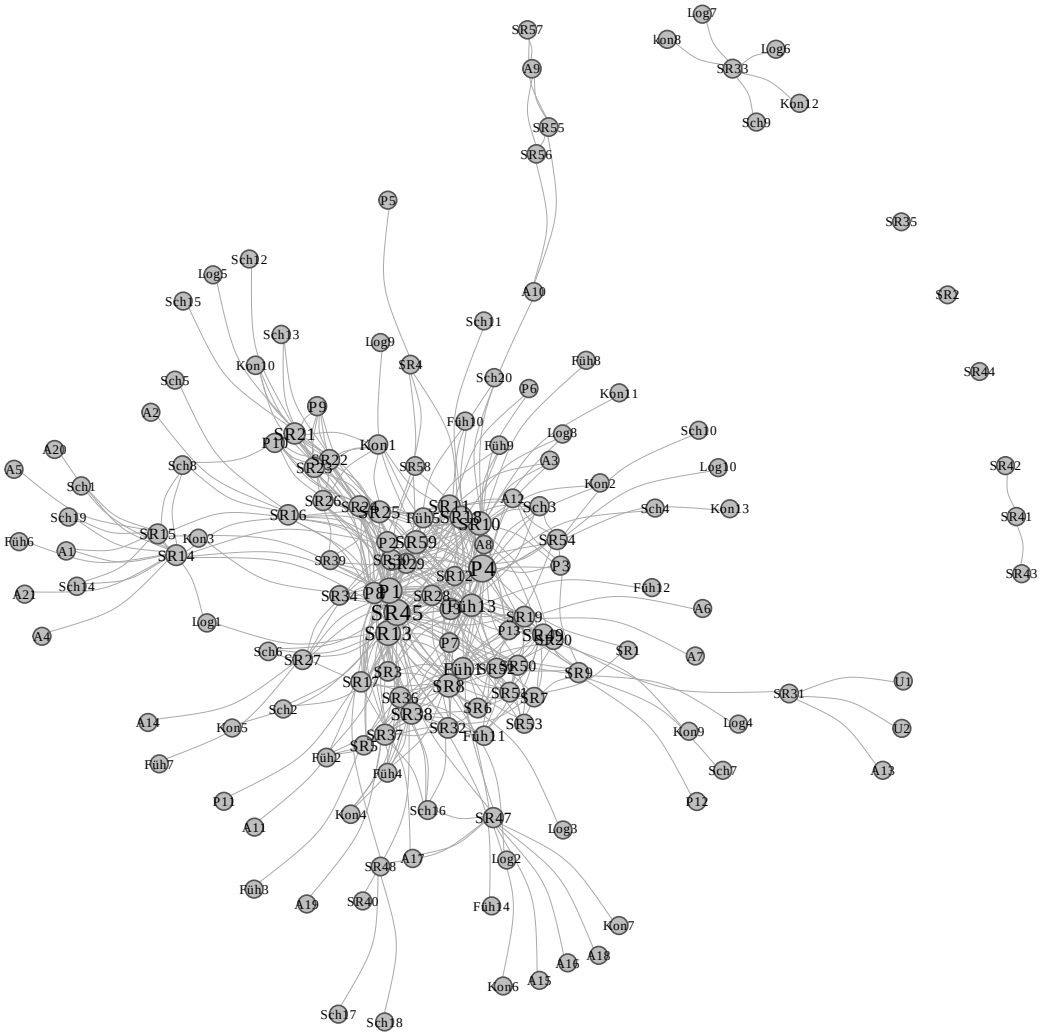
```
plot(net, layout=fr_norm, rescale=F, edge.width=.5,
     edge.curved=.3, vertex.label=V(net)$Label,
     vertex.label.color="black", vertex.color="gray",
     vertex.frame.color="#555555", vertex.size= 9+degrees/9,
     vertex.label.cex=0.5+degrees*0.008,
     vertex.label.family="CM Sans", vertex.label.font=2)
```



This already looks much more professional. Yet, as you can probably tell right away, getting the node and label sizes right takes a lot of fumbling around in trial and error mode. Moreover, we still ended up with a lot of overlapping nodes.

A common way to fix this in R, is to try to find a multiplying factor for the layout that spreads out everything as much as needed and then set the boundaries for the x and y axes to the range of these multiplied values:

```
fr_new <- fr_norm*2.6

plot(net, layout=fr_new , rescale=F, edge.width=.5,
     xlim=range(fr_new [,1]), ylim=range(fr_new [,2]),
     edge.curved=.3, vertex.label=V(net)$Label,
     vertex.label.color="black", vertex.color="gray",
     vertex.frame.color="#555555", vertex.size= 9+degrees/9,
     vertex.label.cex=0.5+degrees*0.008,
     vertex.label.family="CM Sans", vertex.label.font=2)
```



In our case, a factor of 2.6 seems just right, however, this also took some time to figure out and the resulting node labels are barely dechifferable anymore. So, this is also not a good option and we should maybe switch to a tool which is more convenient for network visualization.

3 Integrating R and Gephi

While R is extremely powerful for statistical analyses of network data, the free and open source software Gephi certainly outperforms it when it comes to drawing network graphs, especially when you are short in time and do not want to have to tweak layout parameters manually in try and error mode until the graph looks at least somewhat similar to what you want it to. Gephi features a graphical user interface that lets you intuitively create beautiful graphs within seconds with just a few clicks.

If you are completely new to Gephi and want to dive deeper into network viz with that tool, I recommend this introduction.

In this paper, I won't explore those basics but concentrate on one particular feature of Gephi that even many advanced users do not know of: You can import Gephi graphs into R for further analyses or additional custom styling.

3.1 Creating Beautiful Layouts with Gephi

What makes Gephi so handy are the build-in layouting algorithms that are programmed to suit many different situations and automatically produce easily interpretable graphs. Here you can find descriptions for the different algorithms you can choose from in Gephi's layout panel (see Figure 4). Two algorithms, I particularly like, are Gephi's interpretation of the `Fruchterman-Reingold` algorithm, we already found implemented in igraph, (though differently, as we will see), and the before mentioned `Force-Atlas2` – the 'home-brew layout of Gephi'.

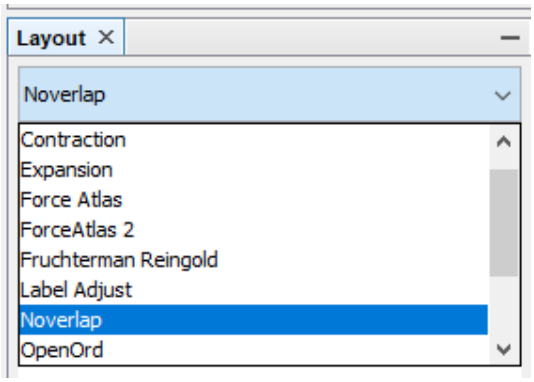


Figure 4: The layout panel in Gephi

In Figure 6 you can see Gephi's Fruchterman-Reingold version at work: Similar as in igraph, it tries to reduce edge crossing and arranges nodes according to shared connections, thereby displaying actors with similar ties close to each other. This time however, the entire network is shaped into a round sphere canceling out the different repulsive powers of the nodes. To create such a network, just select the algorithm from the drop-down menu of the layout panel in the overview window of Gephi, hit "Run" and wait until all nodes are put into place. If you, however, later want to render node sizes proportional to their degree in R (like we do), make sure adjust the sizes already in Gephi in order 'save' enough space around important nodes. Select "Node" and "Ranking" from the appearance panel above and choose values that represent appropriate degree differences for your data:

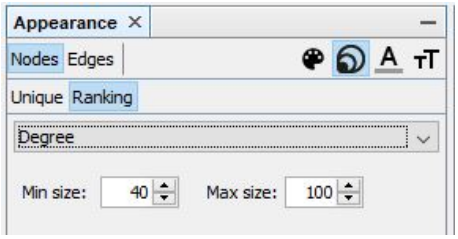


Figure 5: Applying the Fruchterman-Reingold algorithm in Gephi's

Such a layout is not only highly visually appealing but also useful to give an overview of all the actors in a network and their relative importance because it automatically places actors with high degrees in the center and lesser connected actors quite literally in the periphery.

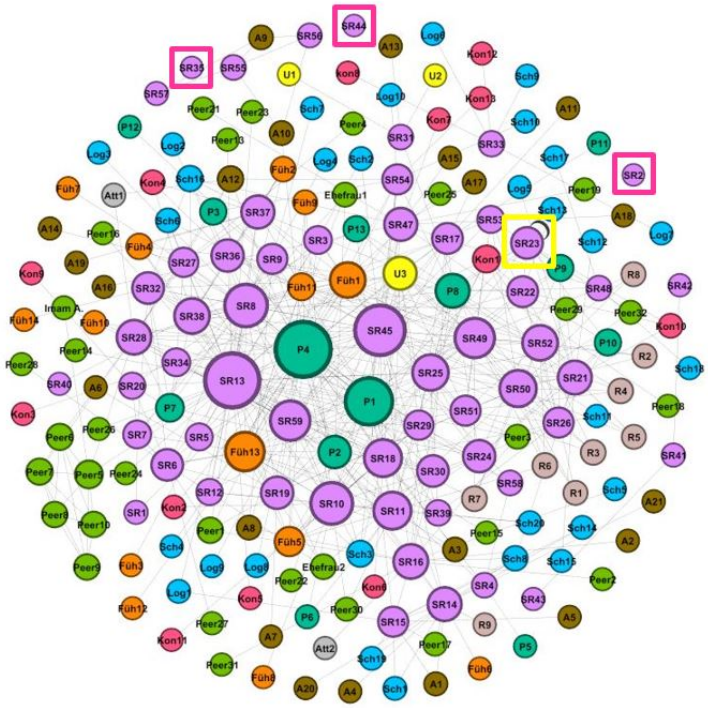


Figure 6: Applied Fruchterman-Reingold algorithm in Gephi

Yet, because the nodes lost their different repulsive powers, unconnected nodes are now placed as close to each other as connected ones which makes it quite difficult to recognize the actual relationships. Moreover, recursive relationships (marked by the yellow square) and isolates (marked in pink) become almost impossible to spot. Therefore, I like to use this visualization only to introduce a network as a whole and the different node types, I'll perform my analyses on but switch to Force-Atlas2 when I go to the nitty-gritty of the underlying structures.

Force-Atlas2 however, usually needs two or three clicks more to look good: 'Close' nodes unfortunately have the save tendency to sit on top of each other as in R. Therefore, it is usually a good workaround not to wait until the algorithm has finished running but rather hit "Stop" immediately after starting "ForceAtlas2", then change the layout to "Expansion", and hit "Run" several times to spread the nodes out as much as desired and finally run "Noverlap" to eliminate node overlapping.

This normally produces beautiful graphs that are much easier to interpret because unconeccted components stand out quite well. And even better: In Gephi you can manually adjust those nodes and drag them exactly where you want them to be:

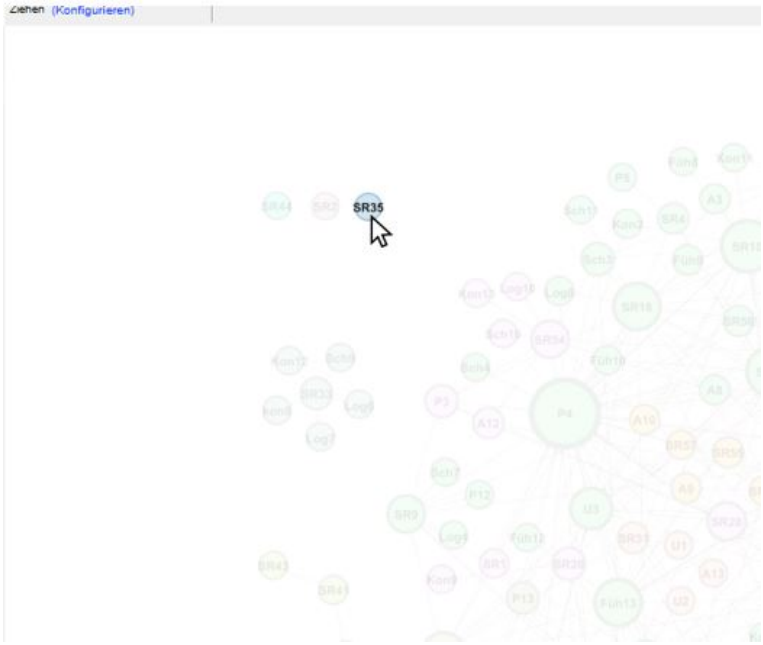


Figure 7: Repositioning nodes manually in Gephi

Personally, I like to drag all isolates and small components to the left of the main graph to make them stand out even more:

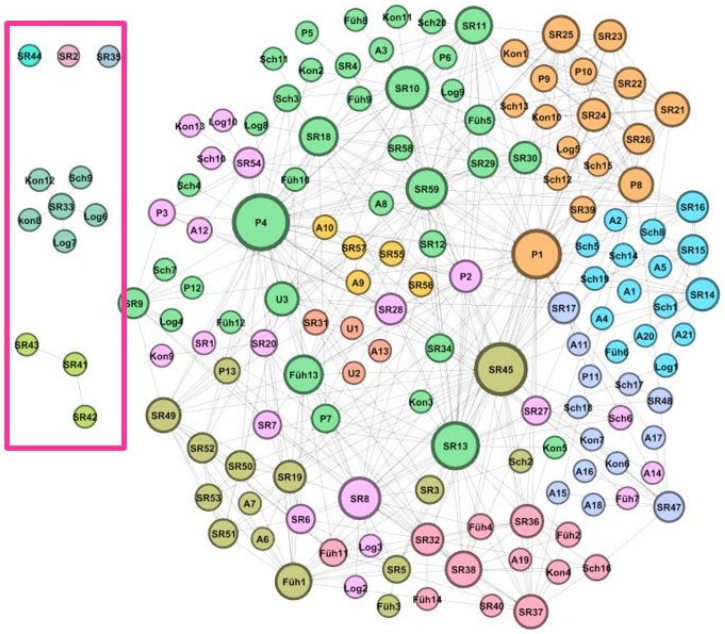


Figure 8: The final Force-Atlas2 layout

3.2 Exporting Gephi Layouts

Once you are satisfied with your layout, exporting it is easy: Just click "File > Export > Graph file"!

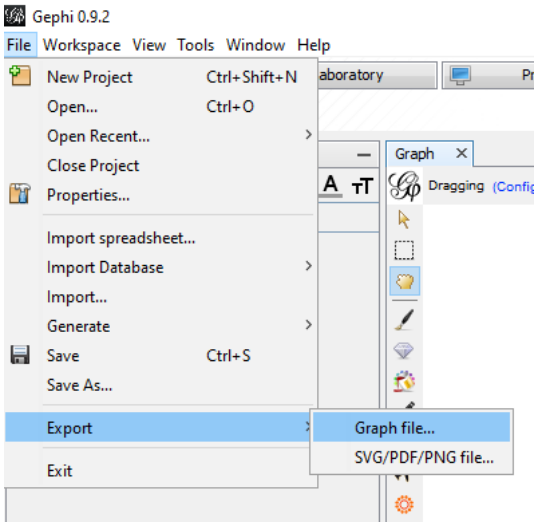


Figure 9: Exporting a graph file

In the next window, give it a name and location of your pleasing and choose ".gexf" as file type:

Figure 10: Selecting a file type

3.3 Importing Gephi Layouts into R

The file that has just been created is a special type of xml file, which looks quite intimidating at first sight. Luckily, we are basically only interested in two attributes hidden in it: the x and the y value each node's `viz:position`:

To access those values from R, we now need to write a block of code that 1) reads in xml-formatted data, 2) parses it and saves all position tags, and 3) extracts all x and y values:

```
myXml <- read_xml("RData/core.gexf")
position <- xml_find_all(myXml, "//viz:position")
x_vals <- xml_attr(position, "x")
y_vals <- xml_attr(position, "y")
```

Now, we combine both data columns into a single dataframe – keeping eight decimal places and then turn those values into proper numeric values.

Figure 11: Anatomy of the gexf file

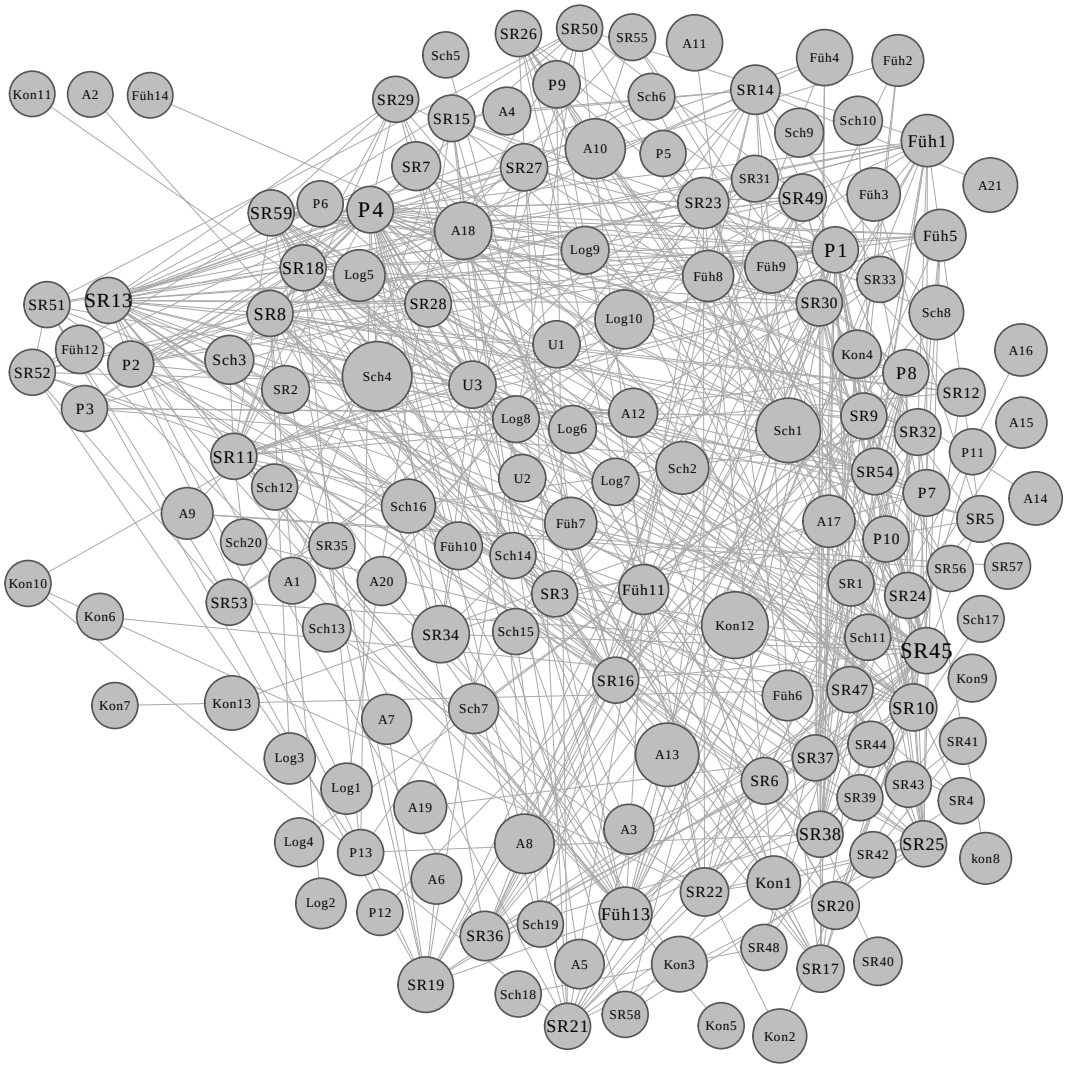
```
df <- data.frame(cbind(x_vals, y_vals))
options(digits=8)
df$x_vals <- as.numeric(df$x_vals)
df$y_vals <- as.numeric(df$y_vals)
```

As the final importing step, we transform the dataframe into a matrix, adjust the matrix dimensions according to our two columns, and normalize also this layout to make x and y each range from -1 to 1.

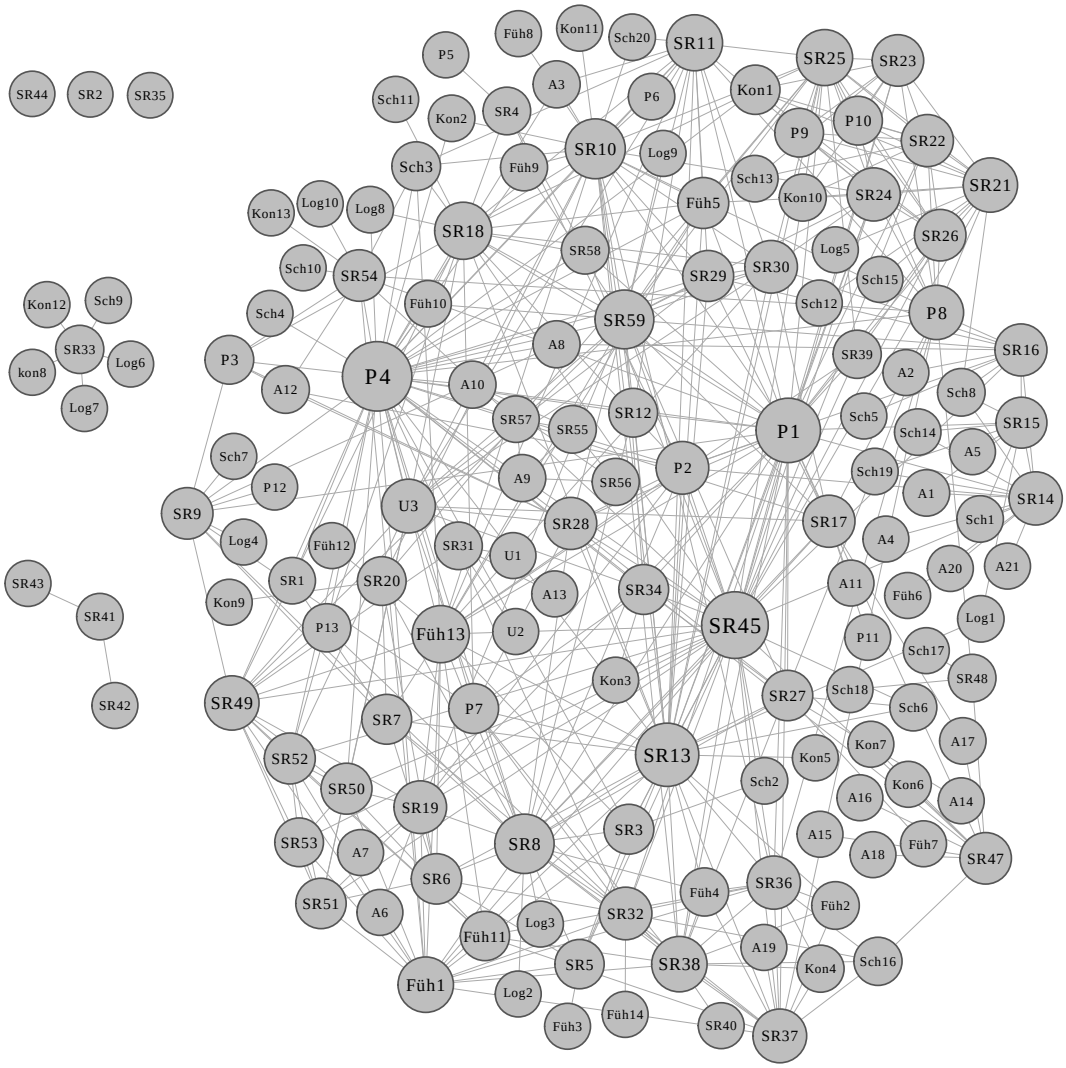
```
mat <- as.matrix(df)
mat2 <- matrix(mat, ncol = ncol(df), dimnames = NULL)
norm_mat2 <- norm_coords(mat2, xmin = -1, xmax = 1, ymin = -1,
                         ymax = 1, zmin = -1, zmax = 1)
```

Now, we can plot our original network onto the coordinates from Gephi:

```
plot(net, layout=norm_mat2, rescale=F, edge.width=.5,
     vertex.label=V(net)$Label,
     vertex.label.color="black", vertex.color="gray",
     vertex.frame.color="#555555", vertex.size= 9+degrees/9,
     vertex.label.cex=0.5+degrees*0.008,
     vertex.label.family="CM Sans", vertex.label.font=2)
```



But wait! Doesn't this plot look somehow odd? Yes, indeed: The layout is right but the nodes themselves are all in the wrong places. This is because our node list (Figure 1) was not sorted, Gephi however does sort the nodes and now R is assigning the coordinates in the wrong order. As soon as we sort the nodes in our original csv file by Id and read in the data, again, everything is fine:



4 Differentiating Nodes

Now we have quite a decent layout to work with. Yet, the graph still does not tell us much.

Looking at the levels of the variable "Rolle_aggregiert" (German for "aggregated role"), we find that our data feature actors with seven different roles within the network:

```
levels(as.factor(V(net)$Rolle_aggregiert))
```

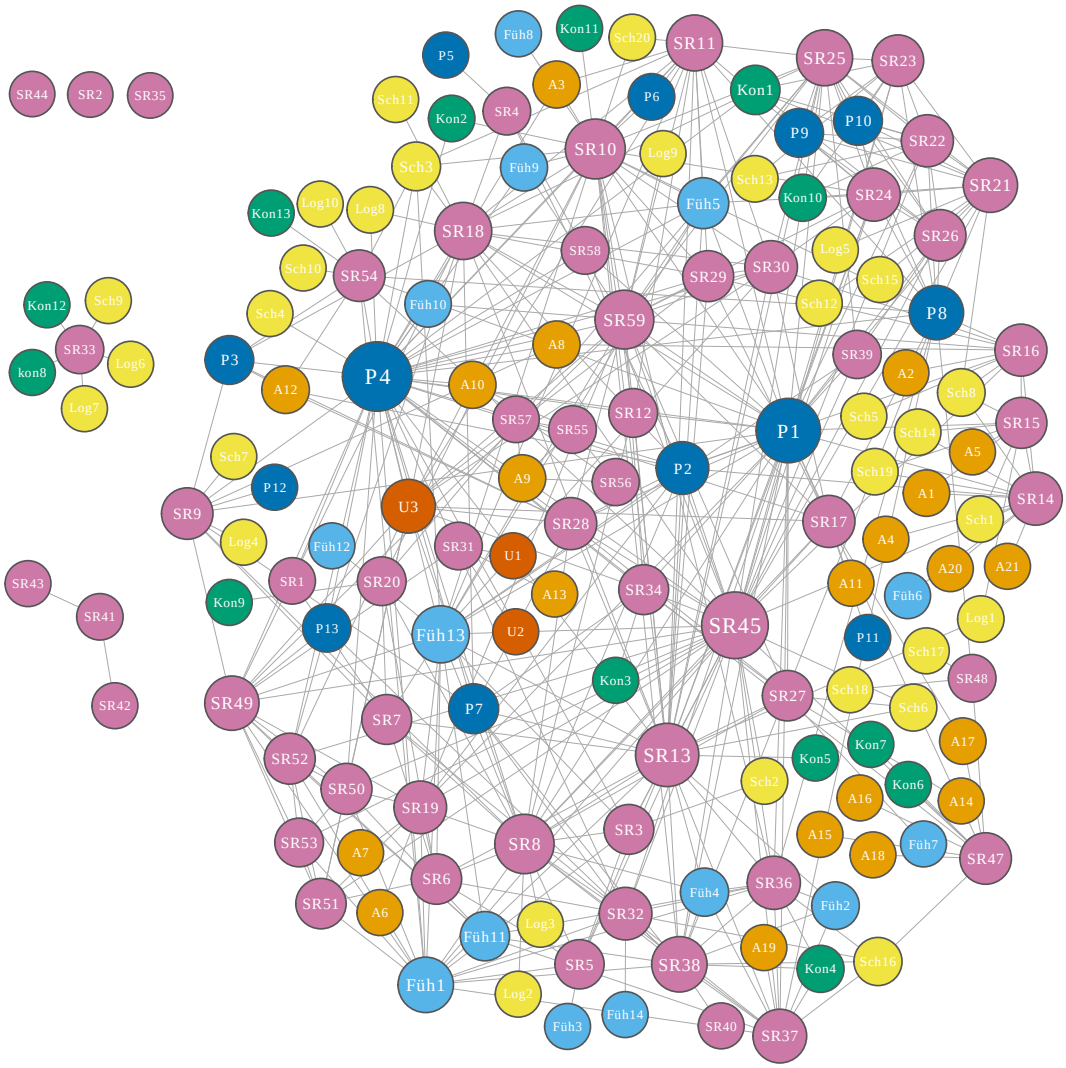
```
## [1] "Anwerber"        "Führungsperson"  "Kontaktmann"
## [4] "LogSchleu"       "Prediger"        "sonstigeUnterst"
## [7] "SR"
```

Currently, it is relatively hard to tell those roles apart immediately just by their label (the "P"s are for instance all Salafist preachers, "SR" are Syria returnees etc.).

It would be much easier, if we would find a sleek way to differentiate different roles visually from one another.

The easiest way to achieve this, is to color different values of a (factor = categorial) variable differently by simply setting the `vertex.color` to this factor:

```
plot(net, layout=norm_mat2, rescale=F, edge.width=.5,
     vertex.color= factor(V(net)$Rolle_aggregiert),
     vertex.label.color="white",
     vertex.frame.color="#555555", vertex.size= 9+degrees/9,
     vertex.label=V(net)$Label, vertex.label.cex=0.5+degrees*0.008,
     vertex.label.family="CM Sans", vertex.label.font=2)
```



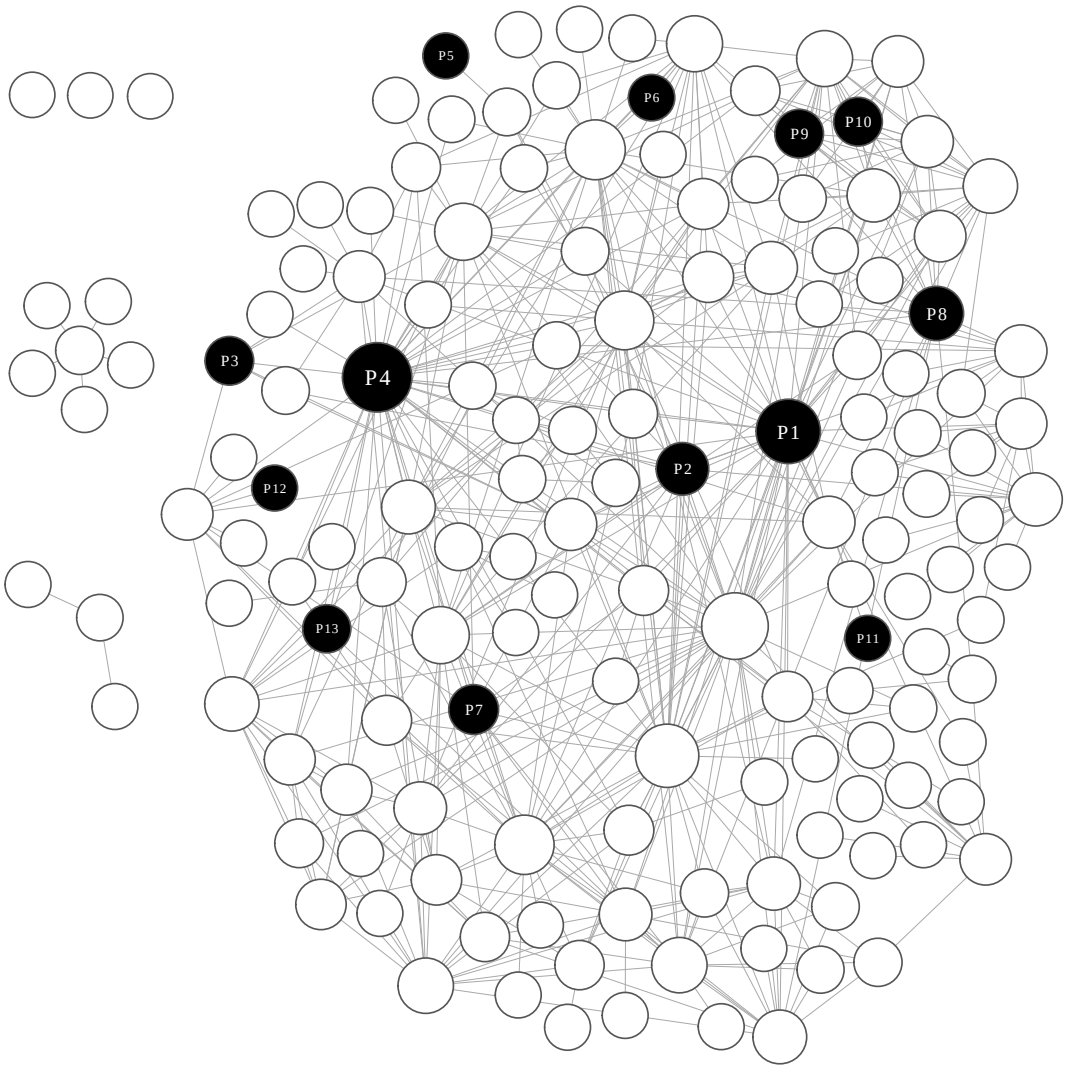
Such a network is much more interesting. However, the editors of many academic journals or books do not accept colored graphs. For those cases, we would have to find a way to convey differentiating information in black and white or grayscales.

One approach would be to only ever highlight one particular role at a time. We could for instance choose to only make the preachers stand out. Therefore, we start by creating a new, blank grouping variable "role_2" and assign all empty content to it. We then override those empty cells with the role "Prediger" but only for rows, were the original role was also "Prediger". Likewise, we create a new labeling variable "label_2" that only copies the original label for rows where the role is 'Prediger' and leaves all other cells blank.

```
V(net)$role_2 <- ""
V(net)$role_2[V(net)$Rolle_aggregiert=="Prediger"] <-
  nodes$Rolle_aggregiert[V(net)$Rolle_aggregiert=="Prediger"]

V(net)$label_2 <- ""
V(net)$label_2[V(net)$Rolle_aggregiert=="Prediger"] <-
  nodes$Label[V(net)$Rolle_aggregiert=="Prediger"]
```

We can then hover over the two different values of those new varibles, assigning them the levels "black" and "white" for the `vertex.color` and `vertex.label.color` (make sure to reverse the order here).

```
plot(net, layout=norm_mat2, rescale=F, edge.width=.5,
     vertex.color= as.character(factor(
       V(net)$role_2,labels = c("white", "black"))),
     vertex.frame.color="#555555", vertex.size= 9+degrees/9,
     vertex.label=V(net)$label_2, vertex.label.color=
       as.character(factor(V(net)$role_2,labels =
                           c("black", "white"))),
     vertex.label.cex=0.5+degrees*0.008,
     vertex.label.family="CM Sans", vertex.label.font=2)
```



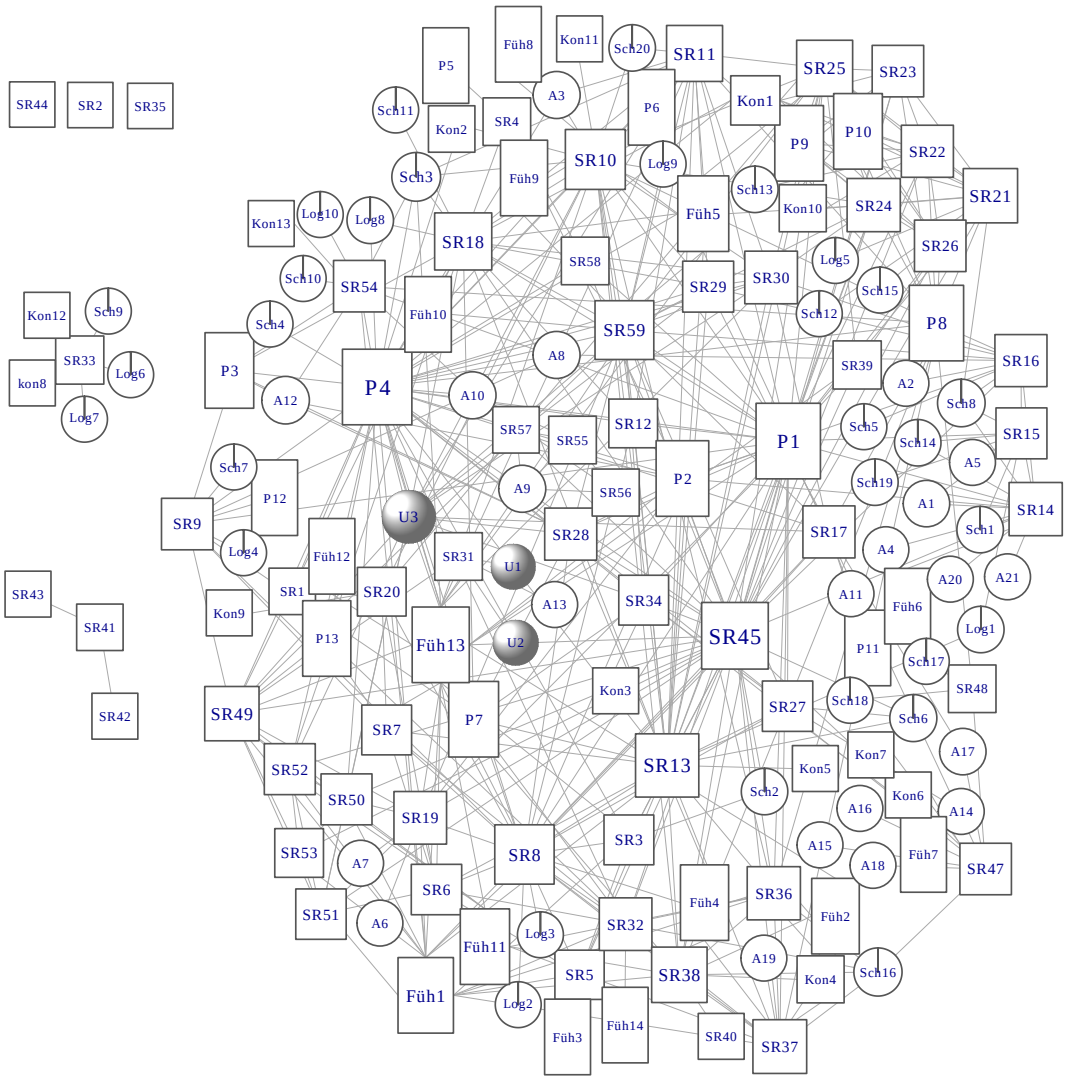
This would look certainly look good in a publication. However, it is still frustating to loose all the information on the other nodes. An idea to solve this problem would be to differentiate nodes not by color but by node shape. And indeed, igraph has a built-in function `shapes()` that yields ten different shape strings:

```
shapes()

##  [1] "circle"     "crectangle" "csquare"    "none"
##  [5] "pie"        "raster"     "rectangle"  "sphere"
##  [9] "square"     "vrectangle"
```

Of those ten, however, only eight are useful straight away ("none" would mean not to draw a shape at all, and raster is reserved for a raster image that has to be defined). Luckily, in our case we would only need to display seven different roles, so we should be good to go by saving seven favorite strings (dropping "none" at the fourth Position, "raster" at seventh position, and maybe also "vrectangle" at the tenth position - just because it sounds similar in shape to the ordinary. "rectangle").

We can then create different shapes from the numerically transformed factor levels of this variable:

```
best_shapes <- shapes()[c(1,2,3,5,7,8,9)]

plot(net, layout=norm_mat2, rescale=F, edge.width=.5,
     vertex.shape = best_shapes[as.numeric(
       factor(V(net)$Rolle_aggregiert))],
     vertex.color= "white", vertex.label=V(net)$Label,
     vertex.frame.color="#555555", vertex.size= 9+degrees/9,
     vertex.label.cex=0.5+degrees*0.008, vertex.label.family=
       "CM Sans", vertex.label.font=2)
```



Yikes!!!

5 Adding PNGs as Shapes

Nice try... but nah. Those built in shapes do not seem to get us very far either.

But what if we could use hand-picked icons instead of those shapes? Well, actually, we can. The are several different ways of integrating images into network graphs.

You could for instance install the Gephi plugin Image Preview, then link each node to an image path and select the option 'Render nodes as images' in the output panel (see figure 12). This is however not very flexible and quite memory consuming and might cause Gephi to crash for large networks.

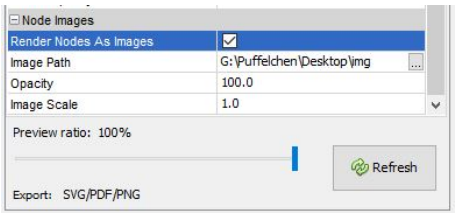


Figure 12: Linking to images directly in Gephi

Another alternative would be to use the R package `qgraph` (as demonstrated in this blog post by Katherine Ognyanova) which has a built-in `image` property. Yet, since qgraph and igraph objects are different object classes in R and I do not like to switch back and forth between them, I personally prefer the following approach, with which we can directly add to the shape property in igraph:

5.1 Selecting Icons

First we have to store suitable PNG files in a folder on our computer. Suitable means that the PNGs should 1) have a transparent background (so that there are no areas unintentionally overshadowing parts of the network's edges), 2) be filled (otherwise the transparent background will cause edges to 'poke' through the icons), and 3) have the right dimensions (usually four, one for each rgb channel - red, green, blue, and alpha).

My personal go-to approach to assemble images that meet those criteria is to browse icons on flaticon.com with the filter 'lineal color' selected (see figure 13).

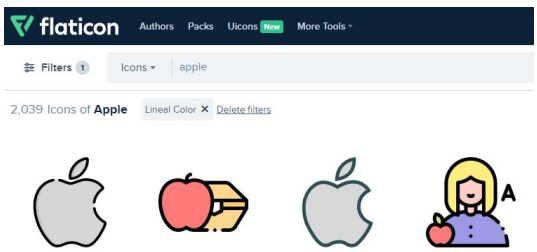


Figure 13: Selecting images

This way, I get filled images with sharp edges that I can download all in the same size (default is 512x512 px, which is usually ideal).

For black and white publications, I then use the command line tool imagemagick to batch convert all images in a folder to grayscale. Therefore, I simply navigate to the folder, where my images are at and type `mogrify -colorspace Gray *` into my cmd (not the R console!). This leaves you with nice, equally sized grayscale PNGs with the right color chanels:

Figure 14: Transformed PNGs

5.2 Adding to `shapes()`

In this Stackoverflow post I came across a neat way to transform those pics into igraph shapes. For each icon this process consists of three steps:

First, read the associated PNG file and save it into a variable:

```
img.1 <- readPNG("img_g/Anwerber.png")
```

Second, create a function for each icon that creates a rasterImage from this particular PNG that maps it to the size and coordinates of a given node v:

```
anw <- function(coords, v=NULL, params) {
  vertex.size <- 1/200 * params("vertex", "size")
  if (length(vertex.size) != 1 && !is.null(v)) {
    vertex.size <- vertex.size[v]
  }
  rasterImage(img.1,
              coords[,1]-vertex.size, coords[,2]-vertex.size,
              coords[,1]+vertex.size, coords[,2]+vertex.size)
}
```

And third, add the plotting function to the `shapes()` vector:

```
add_shape("anw",  plot=anw)
```

Now we have to repeat these steps for all images aka shapes we want to add. After this, in our case the `shapes()` function yields 16 strings, the ten original ones plus the six we added ("anw" for "Anwerber"; "fue" for "Führer"; "kon" for "Kontaktperson"; "pre" for "Prediger", "schleu" for "Schleuser"; "unt" for "Unterstützer"):

```
shapes()

##  [1] "anw"        "circle"     "crectangle" "csquare"
##  [5] "fue"        "kon"        "none"       "pie"
##  [9] "pre"        "raster"     "rectangle"  "schleu"
## [13] "sphere"     "square"     "unt"        "vrectangle"
```

From those, we now want to select the right ones and in the right order, so that the indices of the vector of custom shapes match the indices of the associated levels of our target variable.

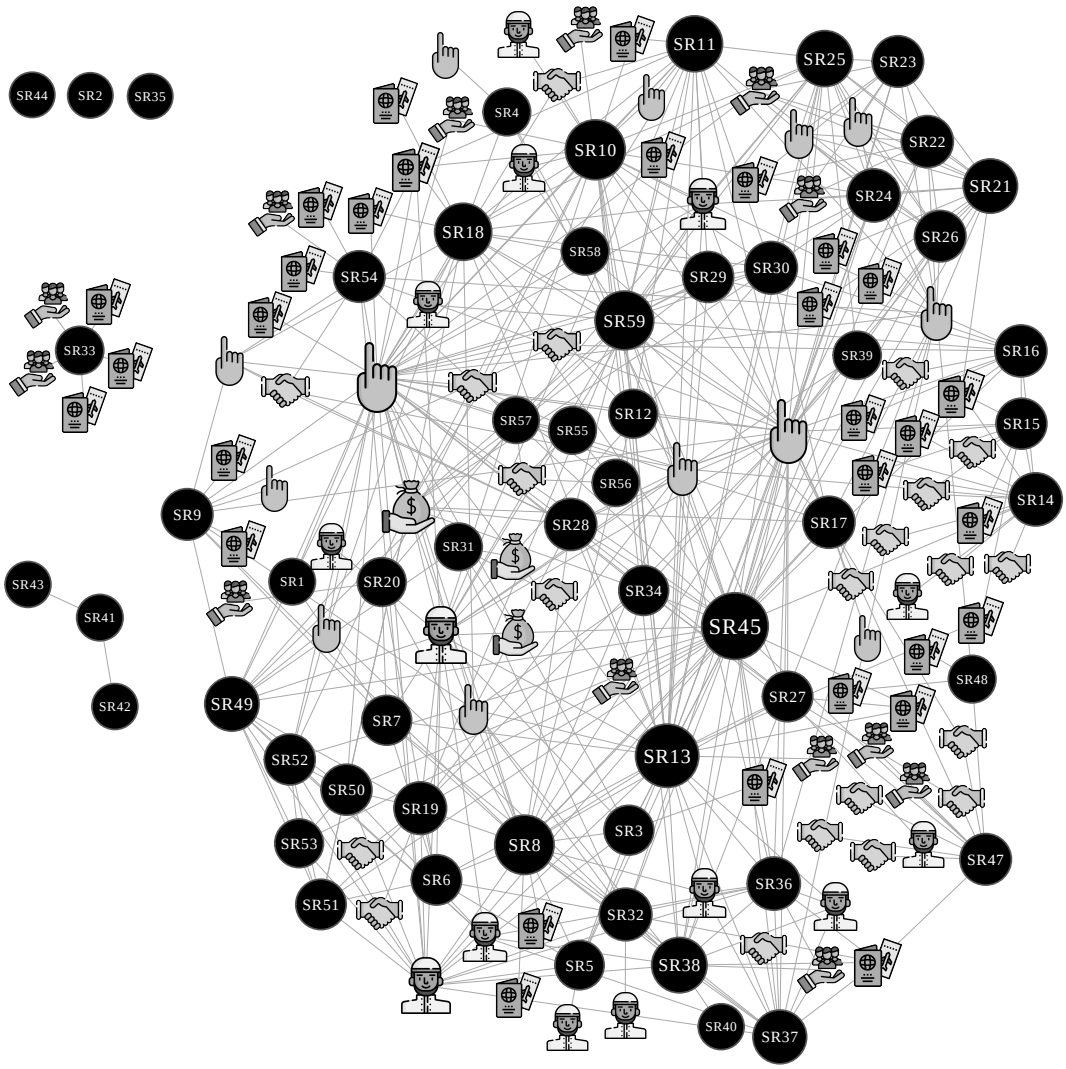
```
levels(as.factor(V(net)$Rolle_aggregiert))

## [1] "Anwerber"        "Führungsperson"  "Kontaktmann"
## [4] "LogSchleu"       "Prediger"        "sonstigeUnterst"
## [7] "SR"

custom_shapes <- shapes()[c(1,5,6,12,9,15,2)]

custom_shapes

## [1] "anw"    "fue"    "kon"    "schleu" "pre"    "unt"    "circle"
```

Since we'd like the returnees to be displayed as regular nodes with labels, we keep the "circle" shape (index=2) and, just as before, have to create a new labeling varible that only contains the values for the returnees:

```
V(net)$label3 <- ""
V(net)$label3[V(net)$Rolle_aggregiert=="SR"] <-
  nodes$Label[V(net)$Rolle_aggregiert=="SR"]
```

Equipped with all this, we can finally plot the icon network that does not only render roles as PNGs but also quite magically adjusts them according to the degree of a node:

```
plot(net, layout=norm_mat2, rescale=F, edge.width=.5,
     vertex.shape = custom_shapes[as.numeric(factor(
       V(net)$Rolle_aggregiert))],
     vertex.label=V(net)$label3, vertex.color= "black",
     vertex.label.color="white", vertex.frame.color="#555555",
     vertex.size= 9+degrees/9, vertex.label.cex=0.5+degrees*0.008,
     vertex.label.family="CM Sans", vertex.label.font=2)
```



6 Saving Network Graphics

Saving high-quality graphics from R and R Studio that are suitable for actual publications (e.g. in a scientific paper or a book) can be quite troublesome because the preview is not always very trustworthy and different devices also tend to behave quite differently. Whereas different values for the 'edge.width' seem for example to have hardly any effect for the output in the plot panel in R Studio (see figure 15, they in fact may make a great difference in the file that is actually saved.

So, though I usually prefer all my graphs to be scalable vector graphics because they just give me the most flexibility for further editing and can be nicely embedded into multiple different formats without producing ugly artefacts (e.g. into LaTeX or word docs for printing or into websites), I hardly ever add the built in `svg()` funtion to my R scripts.

Instead, I find it easiest to export the graphs into PDF format first (and turn them into SVGs or PNGs if needed with e.g. Inkscape). This way, I can a) use

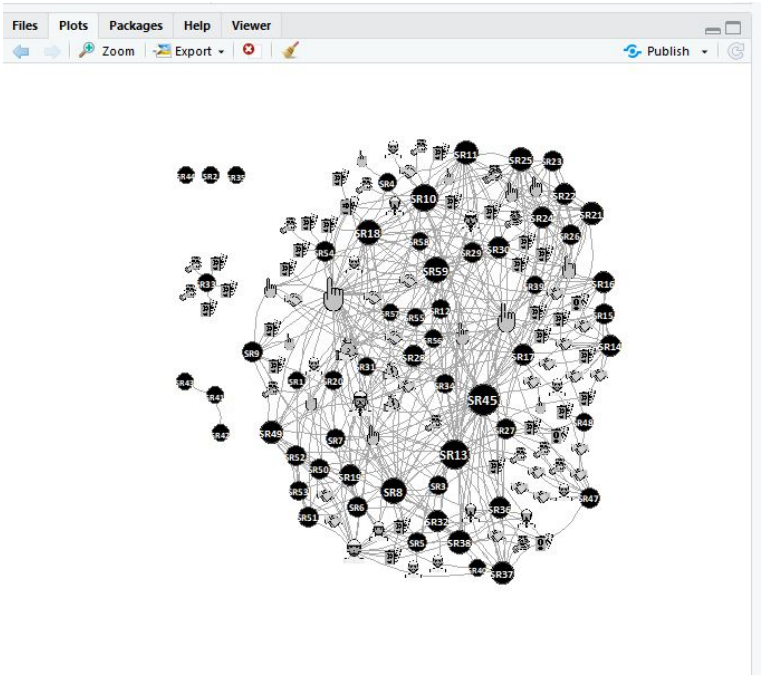


Figure 15: The R Studio plot panel

the built-in preview function and b) activate Cairo scripts which usually make a good job outputting graphics that look as expected. I therefore click on 'Export' right above the plot preview and chose PDF. In the window that pops up, I have to make sure that "Use cairo_pdf device" is enabled and that I give the plot the name I want it to have.

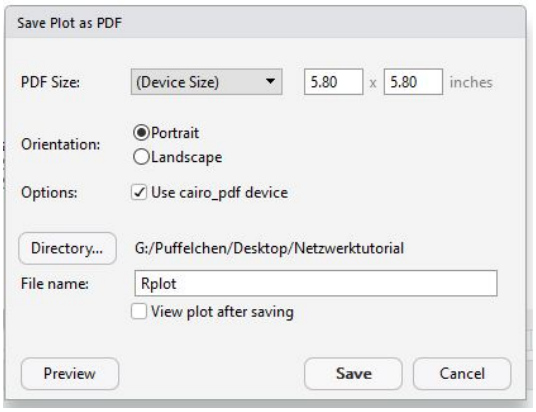
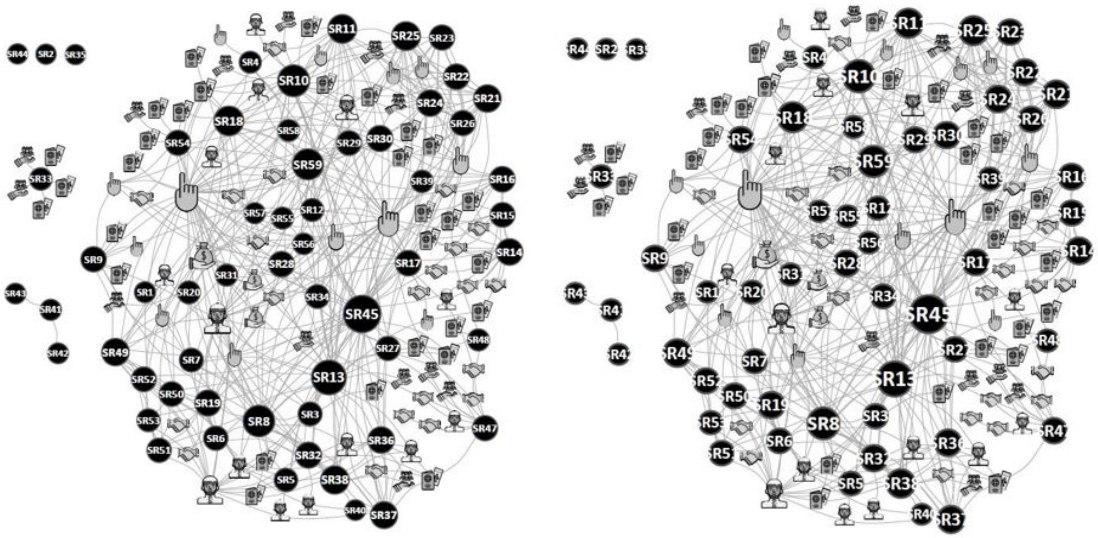


Figure 16: PDF export dialogue

And now comes the still fumbly part: The device size I choose also affects the sizing of the node labels in the output (see figure 16). Therefore, I have to alternate between this window and the preview until I find a device size that preserves the original ratio as much as possible. For most of my plots however, I find 5.8 x 5.8 inches to be the perfect size.



(a) $5.8 * 5.8$ inches

(b) $5.4 * 5.4$ inches

Figure 17: PDF output with different device size settings