Network visualization with R

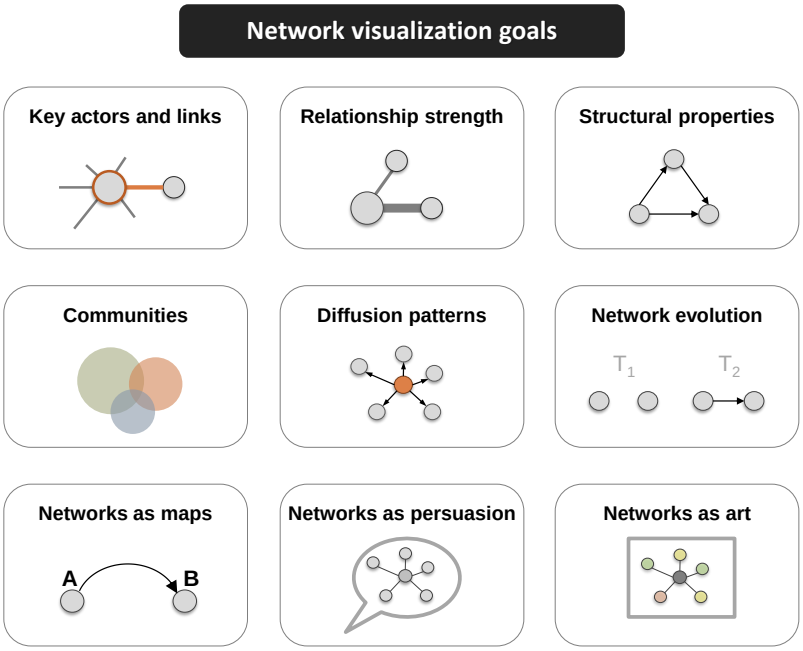
Sunbelt 2019 Workshop, Montreal, Canada

Katherine Ognyanova, *Rutgers University*
Web: *www.kateto.net*, Twitter: *ognyanova*

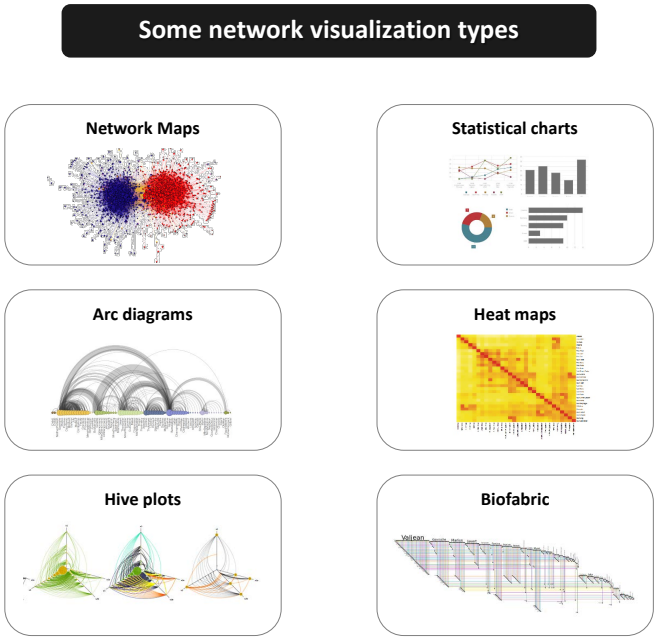
Contents

1 Introduction: network visualization

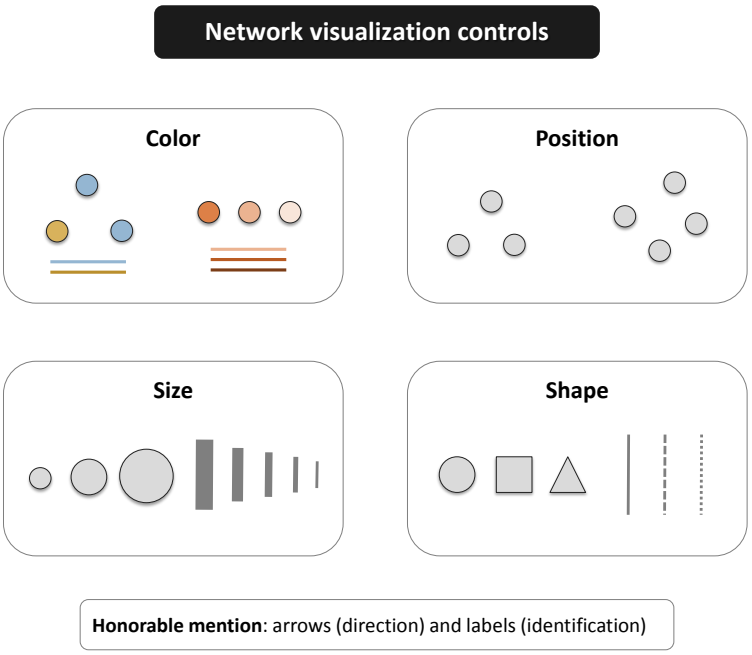
The main concern in designing a network visualization is the purpose it has to serve. What are the structural properties that we want to highlight? What are the key concerns we want to address?



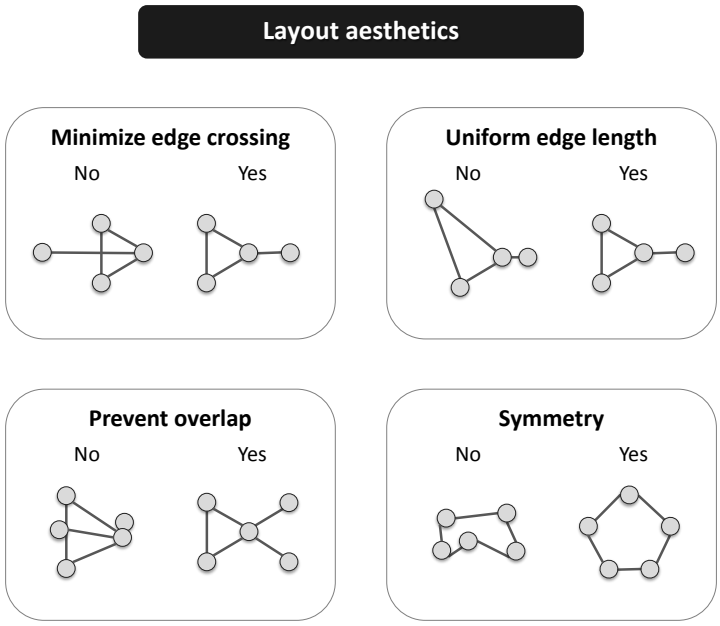
Network maps are far from the only visualization available for graphs - other network representation formats, and even simple charts of key characteristics, may be more appropriate in some cases.

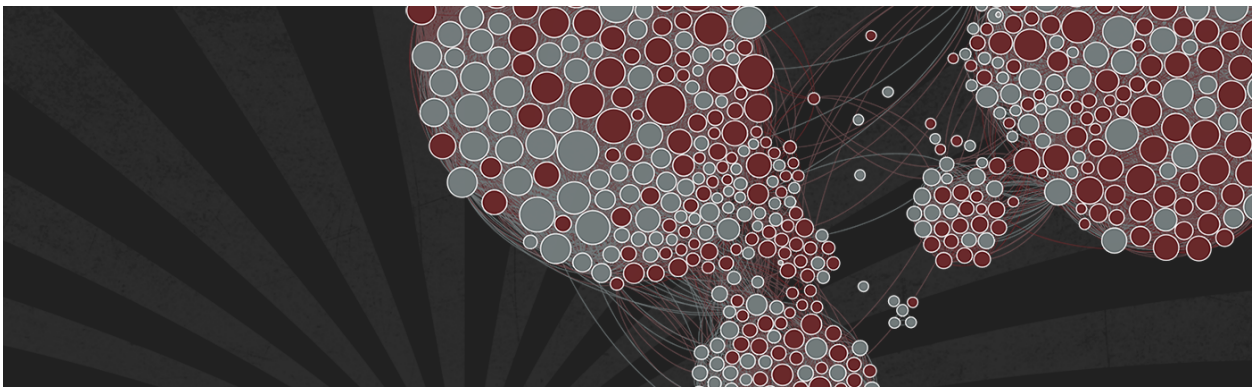


In network maps, as in other visualization formats, we have several key elements that control the outcome. The major ones are color, size, shape, and position.



Modern graph layouts are optimized for speed and aesthetics. In particular, they seek to minimize overlaps and edge crossing, and ensure similar edge length across the graph.





Note: You can download all workshop materials here, or visit kateto.net/sunbelt2019.

This tutorial uses several key packages that you will need to install in order to follow along. Other packages will be mentioned along the way, but those are not critical and can be skipped.

The main packages we are going to use are igraph (maintained by Gabor Csardi and Tamas Nepusz), sna & network (maintained by Carter Butts and the Statnet team), ggraph(maintained by Thomas Lin Pederson), visNetwork (maintained by Benoit Thieurmel), threejs (maintained by Bryan W. Lewis), NetworkD3 (maintained by Christopher Gandrud), and ndtv (maintained by Skye Bender-deMoll).

```
install.packages("igraph")
install.packages("network")
install.packages("sna")
install.packages("ggraph")
install.packages("visNetwork")
install.packages("threejs")
install.packages("networkD3")
install.packages("ndtv")
```

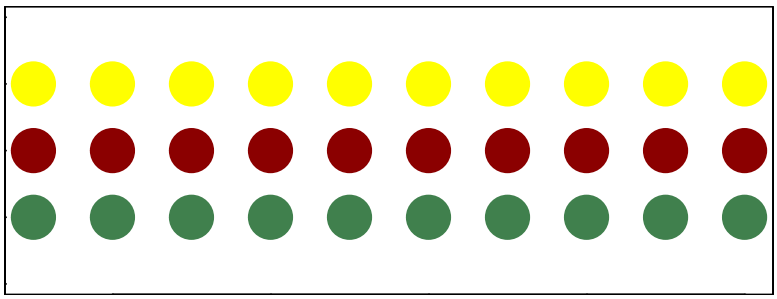
---

2 Colors in R plots

Colors are pretty, but more importantly, they help people differentiate between types of objects or levels of an attribute. In most R functions, you can use *named colors*, *hex*, or *RGB* values.

In the simple base R plot chart below, `x` and `y` are the point coordinates, `pch` is the point symbol shape, `cex` is the point size, and `col` is the color. To see the parameters for plotting in base R, check out `?par`.

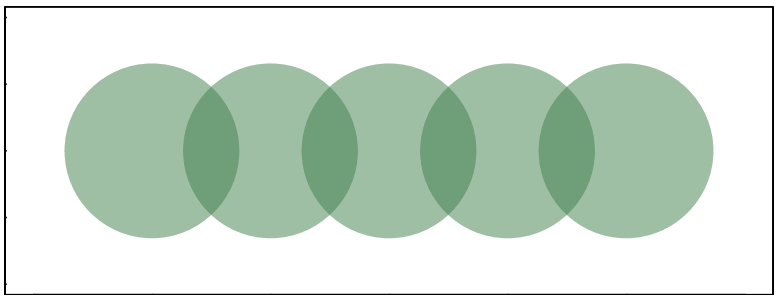
```
plot(x=1:10, y=rep(5,10), pch=19, cex=3, col="dark red")
points(x=1:10, y=rep(6, 10), pch=19, cex=3, col="557799")
points(x=1:10, y=rep(4, 10), pch=19, cex=3, col=rgb(.25, .5, .3))
```



You may notice that RGB here ranges from 0 to 1. While this is the R default, you can also set it to the 0-255 range using something like `rgb(10, 100, 100, maxColorValue=255)`.

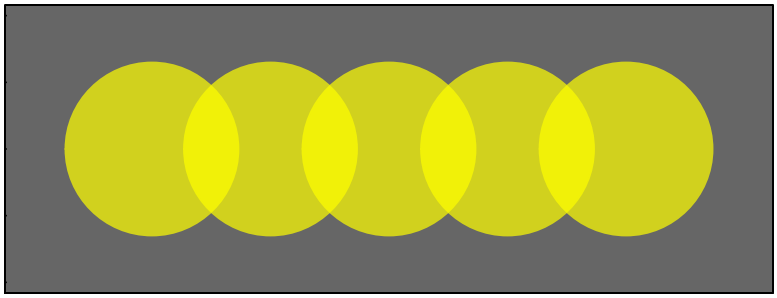
We can set the opacity/transparency of an element using the parameter `alpha` (range 0-1):

```
plot(x=1:5, y=rep(5,5), pch=19, cex=12, col=rgb(.25, .5, .3, alpha=.5), xlim=c(0,6))
```



If we have a hex color representation, we can set the transparency alpha using `adjustcolor` from package `grDevices`. For fun, let's also set the plot background to gray using the `par()` function for graphical parameters. We won't do that below, but we could set the margins of the plot with `par(mar=c(bottom, left, top, right))`, or tell R not to clear the previous plot before adding a new one with `par(new=TRUE)`.

```
par(bg="gray40")
col.tr <- grDevices::adjustcolor("557799", alpha=0.7)
plot(x=1:5, y=rep(5,5), pch=19, cex=12, col=col.tr, xlim=c(0,6))
```

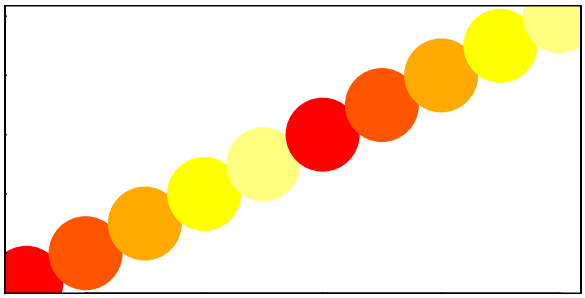


If you plan on using the built-in color names, here's how to list all of them:

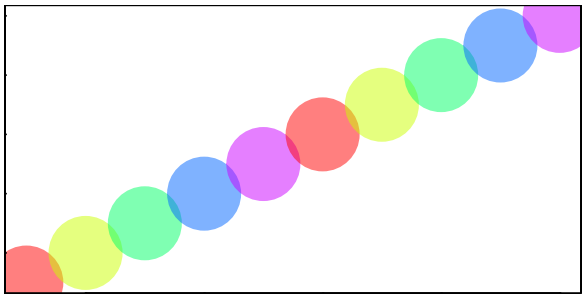
```
colors()                            # List all named colors
grep("blue", colors(), value=T)     # Colors that have "blue" in the name
```

In many cases, we need a number of contrasting colors, or multiple shades of a color. R comes with some predefined palette function that can generate those for us. For example:

```
pal1 <- heat.colors(5, alpha=1)    #  5 colors from the heat palette, opaque
pal2 <- rainbow(5, alpha=.5)       #  5 colors from the heat palette, transparent
plot(x=1:10, y=1:10, pch=19, cex=5, col=pal1)
```

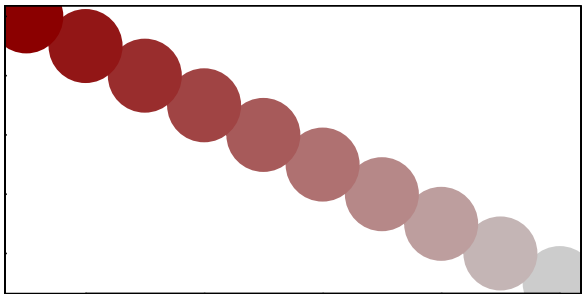


```
plot(x=1:10, y=1:10, pch=19, cex=5, col=pal2)
```



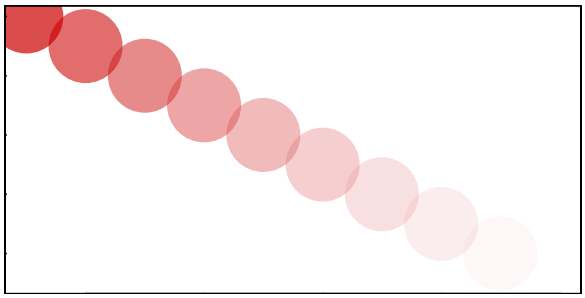
We can also generate our own gradients using `colorRampPalette`. Note that `colorRampPalette` returns a *function* that we can use to generate as many colors from that palette as we need.

```
palf <- colorRampPalette(c("gray80", "dark red"))
plot(x=10:1, y=1:10, pch=19, cex=5, col=palf(10))
```



To add transparency to colorRampPalette, you need to use a parameter `alpha=TRUE`:

```
palf <- colorRampPalette(c(rgb(1,1,1, .2),rgb(.8,0,0, .7)), alpha=TRUE)
plot(x=10:1, y=1:10, pch=19, cex=5, col=palf(10))
```



Finding good color combinations is a tough task - and the built-in R palettes are rather limited. Thankfully there are other available packages for this:

```
# If you don't have R ColorBrewer already, you will need to install it:
install.packages('RColorBrewer')
library('RColorBrewer')
display.brewer.all()
```

This package has one main function, called `brewer.pal`. To use it, you just need to select the desired palette and a number of colors. Let's take a look at some of the `RColorBrewer` palettes:

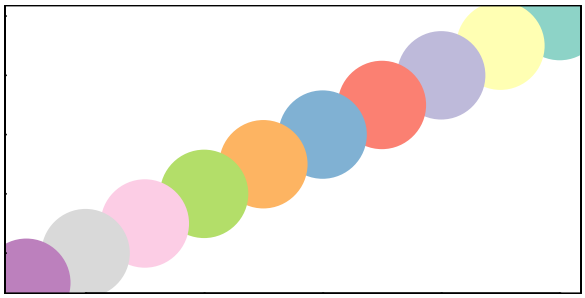
```
display.brewer.pal(8, "Set3")
```

```
display.brewer.pal(8, "Spectral")
```

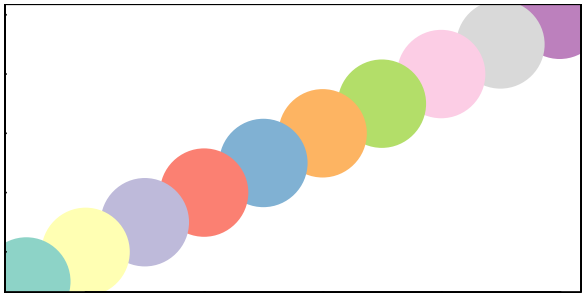
```
display.brewer.pal(8, "Blues")
```

Using `RColorBrewer` palettes in plots:

```
pal3 <- brewer.pal(10, "Set3")
plot(x=10:1, y=10:1, pch=19, cex=6, col=pal3)
```



```
plot(x=10:1, y=10:1, pch=19, cex=6, col=rev(pal3)) # backwards
```



3 Data format, size, and preparation

In this tutorial, we will work primarily with two small example data sets. Both contain data about media organizations. One involves a network of hyperlinks and mentions among news sources. The second is a network of links between media venues and consumers.

While the example data used here is small, many of the ideas behind the visualizations we will generate apply to medium and large-scale networks. This is also the reason why we will rarely use certain visual properties such as the shape of the node symbols: those are impossible to distinguish in larger graph maps. In fact, when drawing very big networks we may even want to hide the network edges, and focus on identifying and visualizing communities of nodes.

At this point, the size of the networks you can visualize in R is limited mainly by the RAM of your machine. One thing to emphasize though is that in many cases, visualizing larger networks as giant hairballs is less helpful than providing charts that show key characteristics of the graph.

3.1 *DATASET 1: edgelist*

The first data set we are going to work with consists of two files, *"Dataset1-Media-Example-NODES.csv"* and *"Dataset1-Media-Example-EDGES.csv"* (download here).

```
nodes <- read.csv("Dataset1-Media-Example-NODES.csv", header=T, as.is=T)
links <- read.csv("Dataset1-Media-Example-EDGES.csv", header=T, as.is=T)
```

Examine the data:

```
head(nodes)
head(links)
```

3.2 *Creating an igraph object*

Next we will convert the raw data to an igraph network object. To do that, we will use the `graph_from_data_frame()` function, which takes two data frames: `d` and `vertices`.

- **d** describes the edges of the network. Its first two columns are the IDs of the source and the target node for each edge. The following columns are edge attributes (weight, type, label, or anything else).
- **vertices** starts with a column of node IDs. Any following columns are interpreted as node attributes.

```
library('igraph')
net <- graph_from_data_frame(d=links, vertices=nodes, directed=T)
net
```

```
## IGRAPH 3cbdde0 DNW- 17 49 --
## + attr: name (v/c), media (v/c), media.type (v/n), type.label
## | (v/c), audience.size (v/n), type (e/c), weight (e/n)
## + edges from 3cbdde0 (vertex names):
```

```
##  [1] s01->s02 s01->s03 s01->s04 s01->s15 s02->s01 s02->s03 s02->s09
##  [8] s02->s10 s03->s01 s03->s04 s03->s05 s03->s08 s03->s10 s03->s11
## [15] s03->s12 s04->s03 s04->s06 s04->s11 s04->s12 s04->s17 s05->s01
## [22] s05->s02 s05->s09 s05->s15 s06->s06 s06->s16 s06->s17 s07->s03
## [29] s07->s08 s07->s10 s07->s14 s08->s03 s08->s07 s08->s09 s09->s10
## [36] s10->s03 s12->s06 s12->s13 s12->s14 s13->s12 s13->s17 s14->s11
## [43] s14->s13 s15->s01 s15->s04 s15->s06 s16->s06 s16->s17 s17->s04
```

The description of an igraph object starts with four letters:

1. D or U, for a directed or undirected graph
2. N for a named graph (where nodes have a `name` attribute)
3. W for a weighted graph (where edges have a `weight` attribute)
4. B for a bipartite (two-mode) graph (where nodes have a `type` attribute)

The two numbers that follow (17 49) refer to the number of nodes and edges in the graph. The description also lists node & edge attributes, for example:

- `(g/c)` - graph-level character attribute
- `(v/c)` - vertex-level character attribute
- `(e/n)` - edge-level numeric attribute

We also have easy access to nodes, edges, and their attributes with:

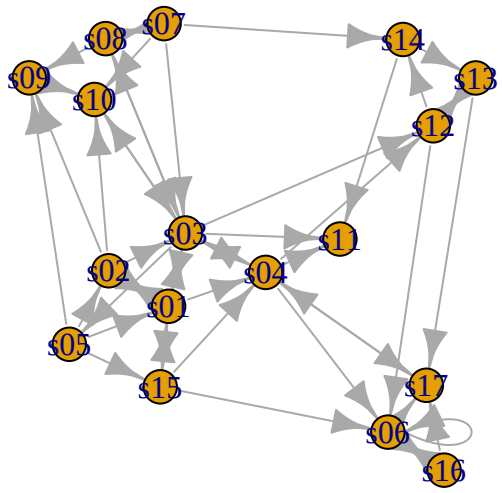
```
E(net)       # The edges of the "net" object
V(net)       # The vertices of the "net" object
E(net)$type  # Edge attribute "type"
V(net)$media # Vertex attribute "media"

# Find nodes and edges by attribute:
# (that returns oblects of type vertex sequence/edge sequence)
V(net)[media=="BBC"]
E(net)[type=="mention"]

# You can also examine the network matrix directly:
net[1,]
net[5,7]
```

It is also easy to extract an edge list or matrix back from the `igraph` network:

```
# Get an edge list or a matrix:
as_edgelist(net, names=T)
as_adjacency_matrix(net, attr="weight")

# Or data frames describing nodes and edges:
as_data_frame(net, what="edges")
as_data_frame(net, what="vertices")
```

Now that we have our igraph network object, let's make a first attempt to plot it.

```
plot(net) # not a pretty picture!
```



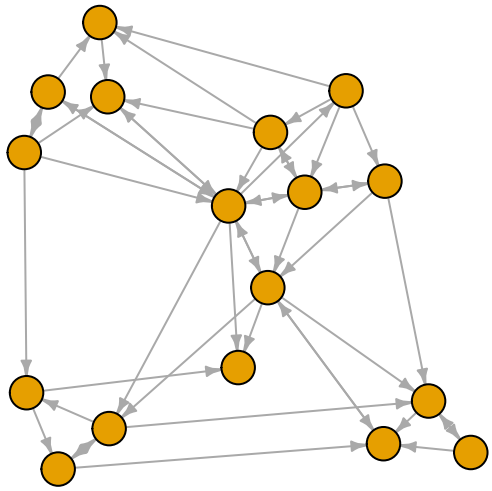
That doesn't look very good. Let's start fixing things by removing the loops in the graph.

```
net <- simplify(net, remove.multiple = F, remove.loops = T)
```

We could also use `simplify` to combine multiple edges by summing their weights with a command like `simplify(net, edge.attr.comb=list(Weight="sum","ignore"))`. Note, however, that this would also combine multiple edge types (in our data: "hyperlinks" and "mentions").

Let's and reduce the arrow size and remove the labels (we do that by setting them to `NA`):

```
plot(net, edge.arrow.size=.4,vertex.label=NA)
```



3.3 *DATASET 2: matrix*

Our second dataset is a network of links between news outlets and consumers. It includes two files, *"Dataset2-Media-Example-NODES.csv"* and *"Dataset2-Media-Example-EDGES.csv"* (download here).

```
nodes2 <- read.csv("Dataset2-Media-User-Example-NODES.csv", header=T, as.is=T)
links2 <- read.csv("Dataset2-Media-User-Example-EDGES.csv", header=T, row.names=1)
```

Examine the data:

```
head(nodes2)
head(links2)
```

3.4 *Two-mode (bipartite) networks in igraph*

We can see that `links2` is an adjacency matrix for a two-mode network. Two-mode or bipartite graphs have two different types of actors and links that go across, but not within each type. Our second media example is a network of that kind, examining links between news sources and their consumers.

```
links2 <- as.matrix(links2)
dim(links2)
dim(nodes2)
```

Next we will convert our second network into an igraph object.

As we have seen above, the edges of our second network are in a matrix format. We can read those into a graph object using `graph_from_incidence_matrix()`. In igraph, bipartite networks have a node attribute called `type` that is `FALSE` (or 0) for vertices in one mode and `TRUE` (or 1) for those in the other mode.

```
head(nodes2)
head(links2)

net2 <- graph_from_incidence_matrix(links2)
table(V(net2)$type)
```

To transform a one-mode network matrix into an `igraph` object, use `graph_from_adjacency_matrix()`.

---

4 Plotting networks with *igraph*

4.1 *Plotting parameters*

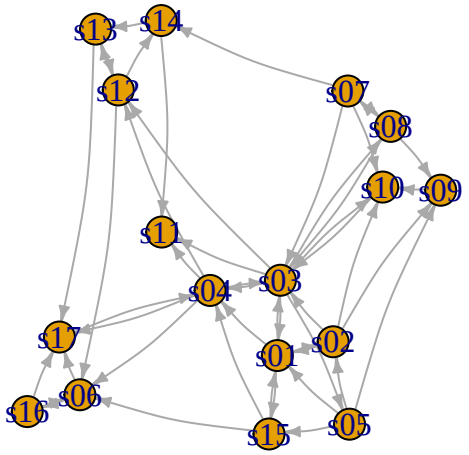
Plotting with `igraph`: the network plots have a wide set of parameters you can set. Those include node options (starting with `vertex.`) and edge options (starting with `edge.`). A list of selected options is included below, but you can also check out `?igraph.plotting` for more information.

The igraph plotting parameters include (among others):

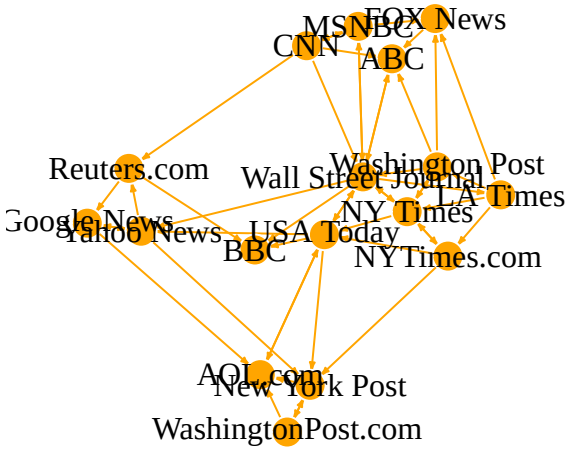
NODES	
vertex.color	Node color
vertex.frame.color	Node border color
vertex.shape	One of “none”, “circle”, “square”, “csquare”, “rectangle” “crectangle”, “vrectangle”, “pie”, “raster”, or “sphere”
vertex.size	Size of the node (default is 15)
vertex.size2	The second size of the node (e.g. for a rectangle)
vertex.label	Character vector used to label the nodes
vertex.label.family	Font family of the label (e.g.“Times”, “Helvetica”)
vertex.label.font	Font: 1 plain, 2 bold, 3, italic, 4 bold italic, 5 symbol
vertex.label.cex	Font size (multiplication factor, device-dependent)
vertex.label.dist	Distance between the label and the vertex
vertex.label.degree	The position of the label in relation to the vertex, where 0 is right, “pi” is left, “pi/2” is below, and “-pi/2” is above
EDGES	
edge.color	Edge color
edge.width	Edge width, defaults to 1
edge.arrow.size	Arrow size, defaults to 1
edge.arrow.width	Arrow width, defaults to 1
edge.lty	Line type, could be 0 or “blank”, 1 or “solid”, 2 or “dashed”, 3 or “dotted”, 4 or “dotdash”, 5 or “longdash”, 6 or “twodash”
edge.label	Character vector used to label edges
edge.label.family	Font family of the label (e.g.“Times”, “Helvetica”)
edge.label.font	Font: 1 plain, 2 bold, 3, italic, 4 bold italic, 5 symbol
edge.label.cex	Font size for edge labels
edge.curved	Edge curvature, range 0-1 (FALSE sets it to 0, TRUE to 0.5)
arrow.mode	Vector specifying whether edges should have arrows, possible values: 0 no arrow, 1 back, 2 forward, 3 both
OTHER	
margin	Empty space margins around the plot, vector with length 4
frame	if TRUE, the plot will be framed
main	If set, adds a title to the plot
sub	If set, adds a subtitle to the plot
asp	Numeric, the aspect ratio of a plot (y/x).
palette	A color palette to use for vertex color
rescale	Whether to rescale coordinates to [-1,1]. Default is TRUE.

We can set the node & edge options in two ways - the first one is to specify them in the `plot()` function, as we are doing below.

```
# Plot with curved edges (edge.curved=.1) and reduce arrow size:
# Note that using curved edges will allow you to see multiple links
# between two nodes (e.g. links going in either direction, or multiplex links)
plot(net, edge.arrow.size=.4, edge.curved=.1)
```

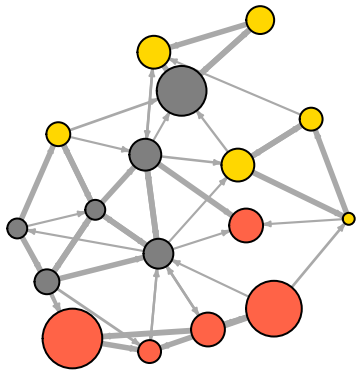



```
# Set edge color to light gray, the node & border color to orange
# Replace the vertex label with the node names stored in "media"
plot(net, edge.arrow.size=.2, edge.color="orange",
     vertex.color="orange", vertex.frame.color="#ffffff",
     vertex.label=V(net)$media, vertex.label.color="black")
```



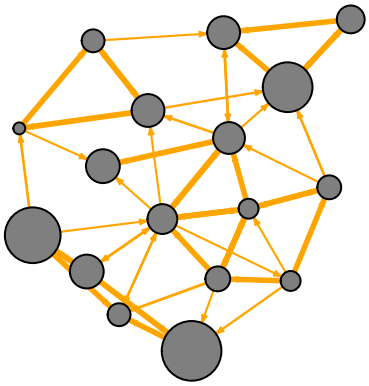
The second way to set attributes is to add them to the `igraph` object. Let's say we want to color our network nodes based on type of media, and size them based on degree centrality (more links -> larger node) We will also change the width of the edges based on their weight.

```
# Generate colors based on media type:
colrs <- c("gray50", "tomato", "gold")
V(net)$color <- colrs[V(net)$media.type]

# Compute node degrees (#links) and use that to set node size:
deg <- degree(net, mode="all")
V(net)$size <- deg*3
# We could also use the audience size value:
V(net)$size <- V(net)$audience.size*0.6

# The labels are currently node IDs.
# Setting them to NA will render no labels:
V(net)$label <- NA

# Set edge width based on weight:
E(net)$width <- E(net)$weight/6

#change arrow size and edge color:
E(net)$arrow.size <- .2
E(net)$edge.color <- "gray80"

# We can even set the network layout:
graph_attr(net, "layout") <- layout_with_lgl
plot(net)
```



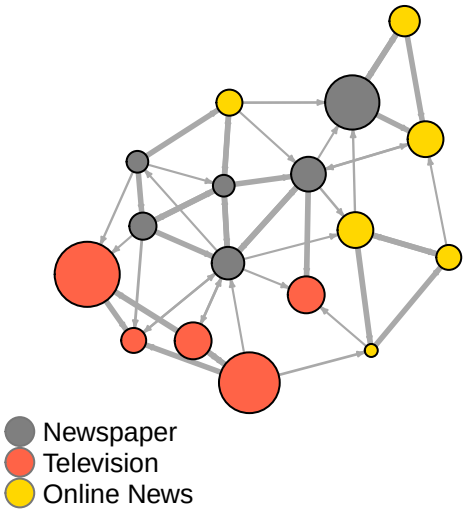
We can also override the attributes explicitly in the plot:

```
plot(net, edge.color="orange", vertex.color="gray50")
```



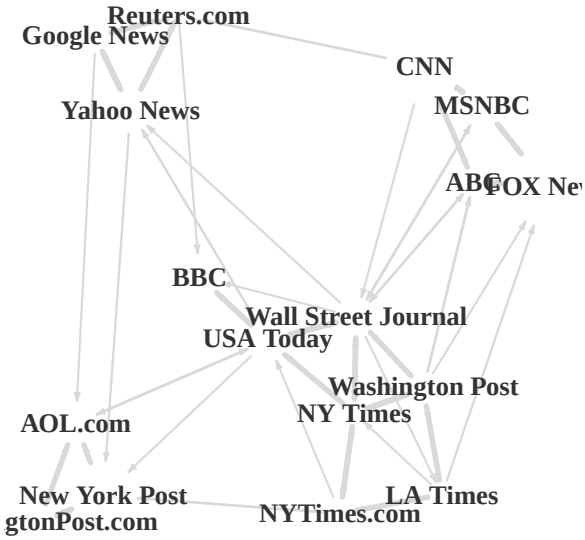
It helps to add a legend explaining the meaning of the colors we used:

```
plot(net)
legend(x=-1.5, y=-1.1, c("Newspaper","Television", "Online News"), pch=21,
       col="#777777", pt.bg=colrs, pt.cex=2, cex=.8, bty="n", ncol=1)
```



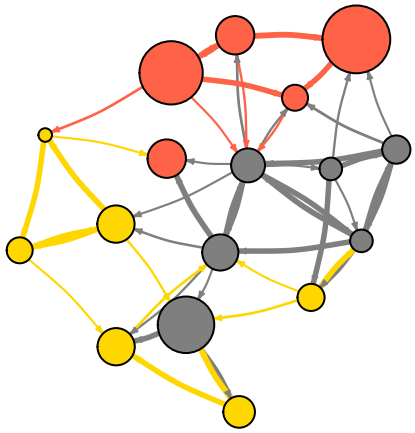
Sometimes, especially with semantic networks, we may be interested in plotting only the labels of the nodes:

```
plot(net, vertex.shape="none", vertex.label=V(net)$media,
     vertex.label.font=2, vertex.label.color="gray40",
     vertex.label.cex=.7, edge.color="gray85")
```



Let's color the edges of the graph based on their source node color. We can get the starting node for each edge with the `ends()` igraph function. It returns the start and end vertex for edges listed in the `es` parameter. The `names` parameter control whether the function returns edge names or IDs.

```
edge.start <- ends(net, es=E(net), names=F)[,1]
edge.col <- V(net)$color[edge.start]

plot(net, edge.color=edge.col, edge.curved=.1)
```

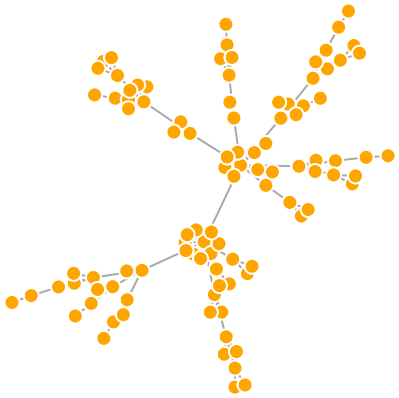


4.2 *Network layouts*

Network layouts are simply algorithms that return coordinates for each node in a network.

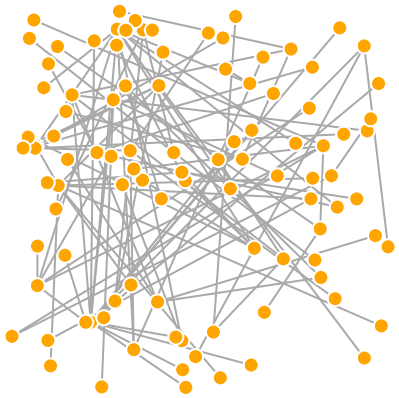
For the purposes of exploring layouts, we will generate a slightly larger 100-node graph. We use the `sample_pa()` function which generates a simple graph starting from one node and adding more nodes and links based on a preset level of preferential attachment (Barabasi-Albert model).

```
net.bg <- sample_pa(100)
V(net.bg)$size <- 8
V(net.bg)$frame.color <- "white"
V(net.bg)$color <- "orange"
V(net.bg)$label <- ""
E(net.bg)$arrow.mode <- 0
plot(net.bg)
```



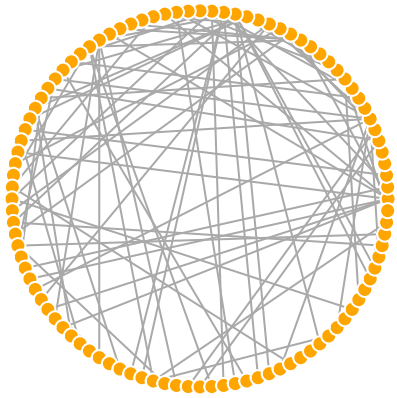
You can set the layout in the plot function:

```
plot(net.bg, layout=layout_randomly)
```



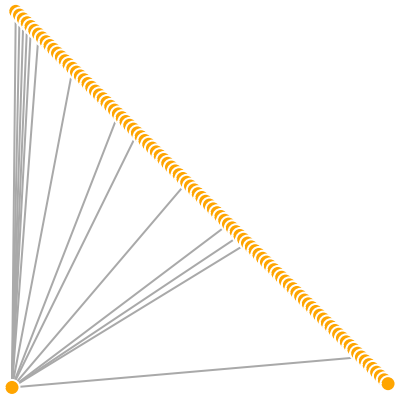
Or you can calculate the vertex coordinates in advance:

```
l <- layout_in_circle(net.bg)
plot(net.bg, layout=l)
```



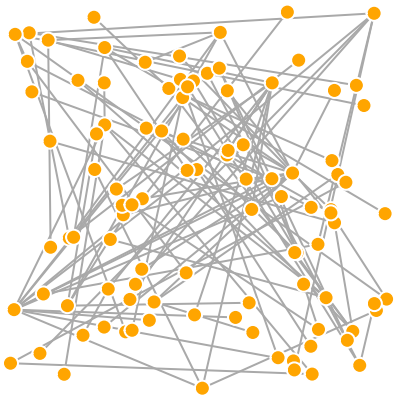
`l` is simply a matrix of x, y coordinates (N x 2) for the N nodes in the graph. For 3D layouts, it has x, y, and z coordinates (N x 3). You can easily generate your own:

```
l <- cbind(1:vcount(net.bg), c(1, vcount(net.bg):2))
plot(net.bg, layout=l)
```

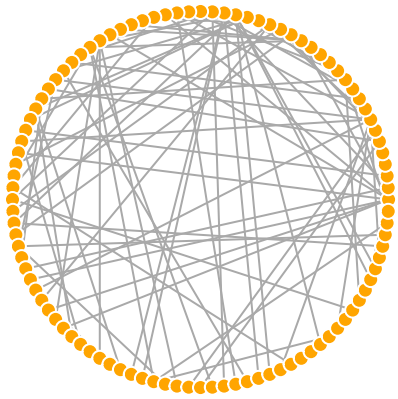


This layout is just an example and not very helpful - thankfully igraph has a number of built-in layouts, including:

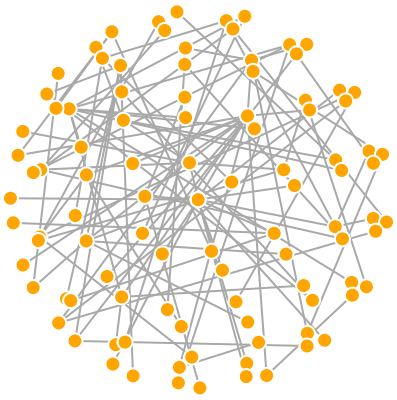
```
# Randomly placed vertices
l <- layout_randomly(net.bg)
plot(net.bg, layout=l)
```



```
# Circle layout
l <- layout_in_circle(net.bg)
plot(net.bg, layout=l)
```



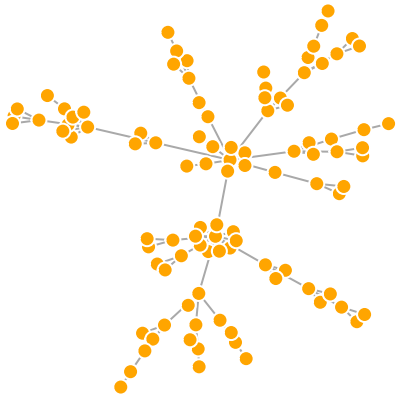
```
# 3D sphere layout
l <- layout_on_sphere(net.bg)
plot(net.bg, layout=l)
```



Fruchterman-Reingold is one of the most used force-directed layout algorithms out there.

Force-directed layouts try to get a nice-looking graph where edges are similar in length and cross each other as little as possible. They simulate the graph as a physical system. Nodes are electrically charged particles that repulse each other when they get too close. The edges act as springs that attract connected nodes closer together. As a result, nodes are evenly distributed through the chart area, and the layout is intuitive in that nodes which share more connections are closer to each other. The disadvantage of these algorithms is that they are rather slow and therefore less often used in graphs larger than ~1000 vertices.

```
l <- layout_with_fr(net.bg)
plot(net.bg, layout=l)
```

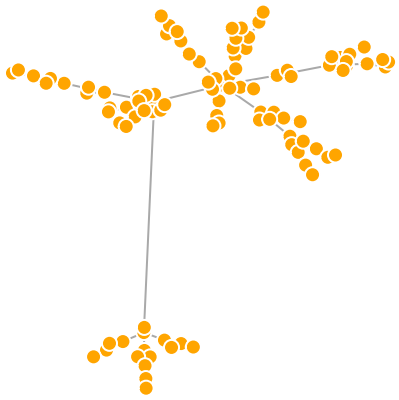


With force-directed layouts, you can use the `niter` parameter to control the number of iterations to perform. The default is set at 500 iterations. You can lower that number for large graphs to get results faster and check if they look reasonable.

```
l <- layout_with_fr(net.bg, niter=50)
plot(net.bg, layout=l)
```

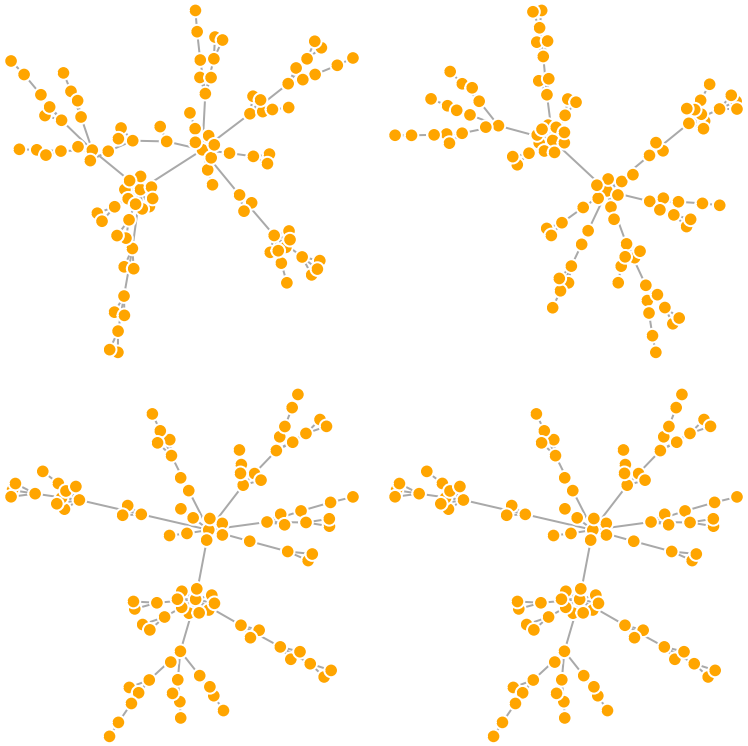
The layout can also interpret edge weights. You can set the "weights" parameter which increases the attraction forces among nodes connected by heavier edges.

```
ws  <-  c(1, rep(100, ecount(net.bg)-1))
lw <- layout_with_fr(net.bg, weights=ws)
plot(net.bg, layout=lw)
```



You will also notice that the Fruchterman-Reingold layout is not deterministic - different runs will result in slightly different configurations. Saving the layout in `l` allows us to get the exact same result multiple times, which can be helpful if you want to plot the time evolution of a graph, or different relationships – and want nodes to stay in the same place in multiple plots.

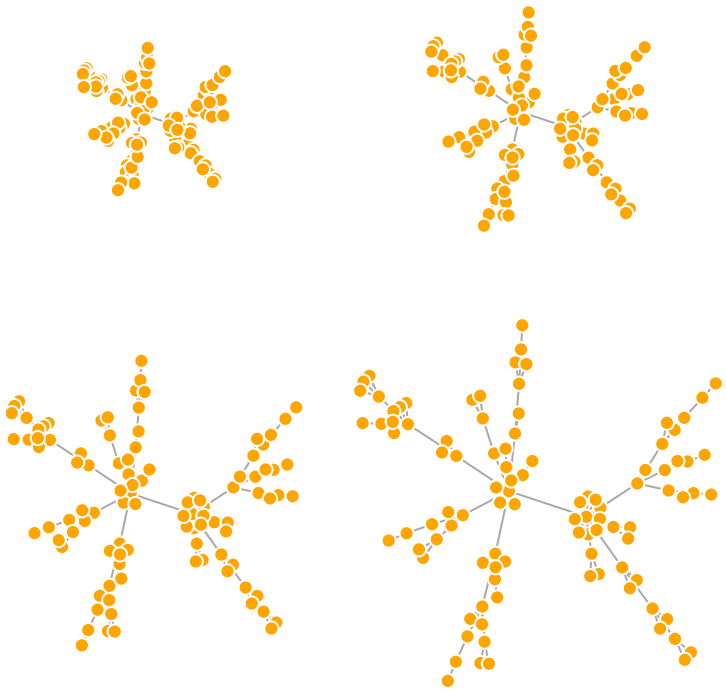
```
par(mfrow=c(2,2), mar=c(0,0,0,0))    # plot four figures - 2 rows, 2 columns
plot(net.bg, layout=layout_with_fr)
plot(net.bg, layout=layout_with_fr)
plot(net.bg, layout=l)
plot(net.bg, layout=l)
```



```
dev.off()
```

By default, the coordinates of the plots are rescaled to the [-1,1] interval for both x and y. You can change that with the parameter `rescale=FALSE` and rescale your plot manually by multiplying the coordinates by a scalar. You can use `norm_coords` to normalize the plot with the boundaries you want. This way you can create more compact or spread out layout versions.

```
l <- layout_with_fr(net.bg)
l <- norm_coords(l, ymin=-1, ymax=1, xmin=-1, xmax=1)

par(mfrow=c(2,2), mar=c(0,0,0,0))
plot(net.bg, rescale=F, layout=l*0.4)
plot(net.bg, rescale=F, layout=l*0.6)
plot(net.bg, rescale=F, layout=l*0.8)
plot(net.bg, rescale=F, layout=l*1.0)
```



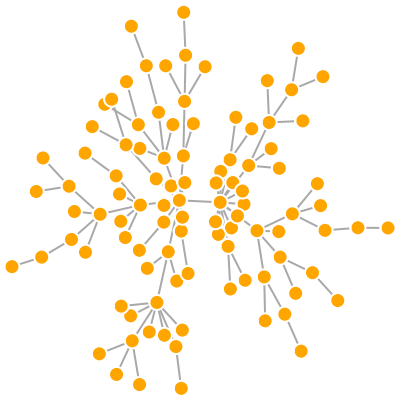
```
dev.off()
```

Some layouts have 3D versions that you can use with parameter `dim=3`. As you might expect, a 3D layout returns a matrix with 3 columns containing the X, Y, and Z coordinates of each node.

```
l <- layout_with_fr(net.bg, dim=3)
plot(net.bg, layout=l)
```

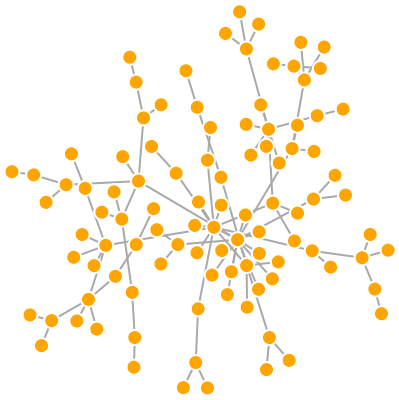
Another popular force-directed algorithm that produces nice results for connected graphs is Kamada Kawai. Like Fruchterman Reingold, it attempts to minimize the energy in a spring system.

```
l <- layout_with_kk(net.bg)
plot(net.bg, layout=l)
```



Graphopt is a nice force-directed layout implemented in `igraph` that uses layering to help with visualizations of large networks.

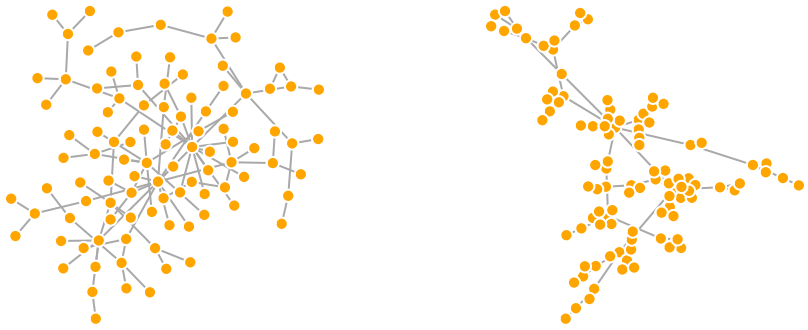
```
l <- layout_with_graphopt(net.bg)
plot(net.bg, layout=l)
```



The available `graphopt` parameters can be used to change the mass and electric charge of nodes, as well as the optimal spring length and the spring constant for edges. The parameter names are `charge` (defaults to 0.001), `mass` (defaults to 30), `spring.length` (defaults to 0), and `spring.constant` (defaults to 1). Tweaking those can lead to considerably different graph layouts.

```
l1 <- layout_with_graphopt(net.bg, charge=0.02)
l2 <- layout_with_graphopt(net.bg, charge=0.00000001)

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

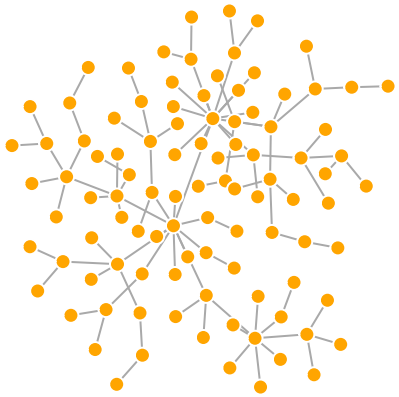
```
plot(net.bg, layout=l1)
plot(net.bg, layout=l2)
```



```
dev.off()
```

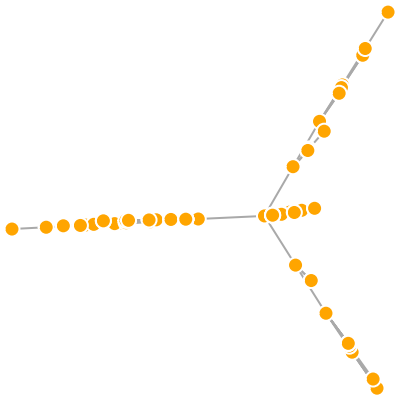
The LGL algorithm is meant for large, connected graphs. Here you can also specify a root: a node that will be placed in the middle of the layout.

```
plot(net.bg, layout=layout_with_lgl)
```



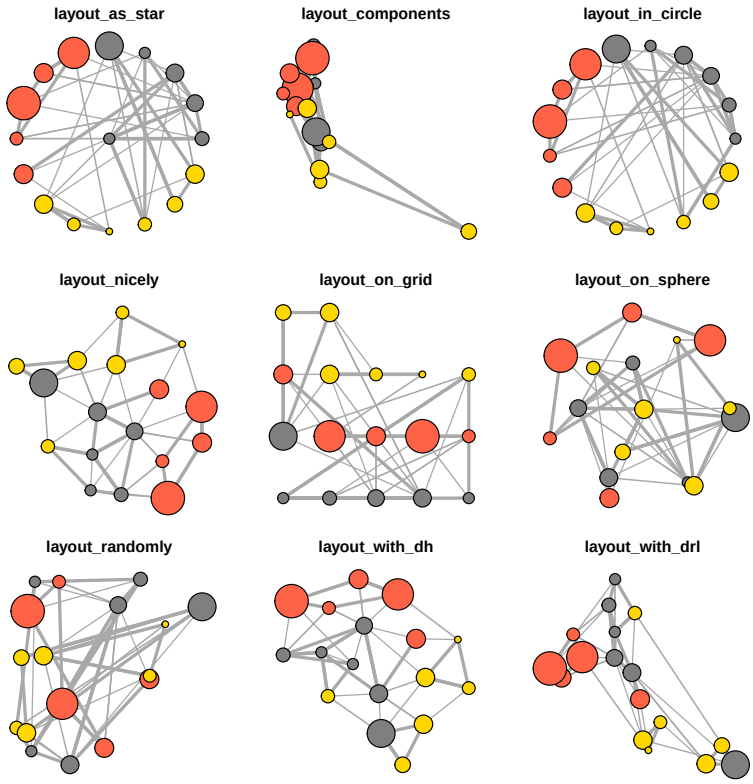
The MDS (multidimensional scaling) algorithm tries to place nodes based on some measure of similarity or distance between them. More similar nodes are plotted closer to each other. By default, the measure used is based on the shortest paths between nodes in the network. We can change that by using our own distance matrix (however defined) with the parameter `dist`. MDS layouts are nice because positions and distances have a clear interpretation. The problem with them is visual clarity: nodes often overlap, or are placed on top of each other.

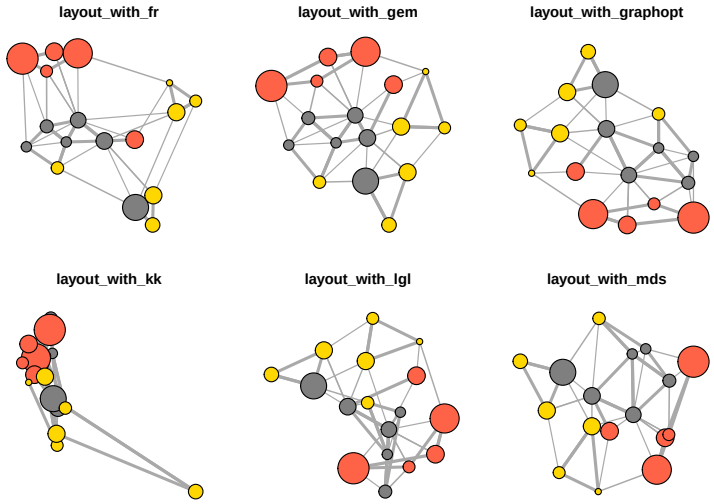
```
plot(net.bg, layout=layout_with_mds)
```



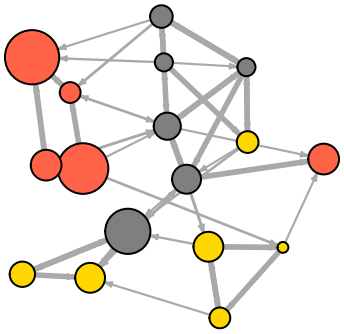
Let's take a look at all available layouts in igraph:

```
layouts <- grep("^layout_", ls("package:igraph"), value=TRUE)[-1]
# Remove layouts that do not apply to our graph.
layouts <- layouts[!grepl("bipartite|merge|norm|sugiyama|tree", layouts)]

par(mfrow=c(3,3), mar=c(1,1,1,1))
for (layout in layouts) {
  print(layout)
  l <- do.call(layout, list(net))
  plot(net, edge.arrow.mode=0, layout=l, main=layout) }
```





4.3 *Highlighting aspects of the network*

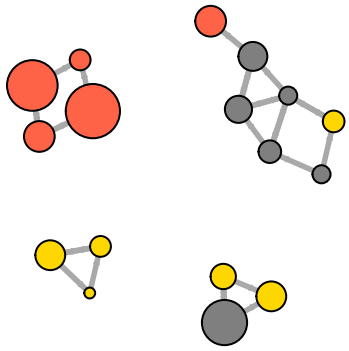


Notice that our network plot is still not too helpful. We can identify the type and size of nodes, but cannot see much about the structure since the links we're examining are so dense. One way to approach this is to see if we can sparsify the network, keeping only the most important ties and discarding the rest.

```
hist(links$weight)
mean(links$weight)
sd(links$weight)
```

There are more sophisticated ways to extract the key edges, but for the purposes of this exercise we'll only keep ones that have weight higher than the mean for the network. In igraph, we can delete edges using `delete_edges(net, edges)`:

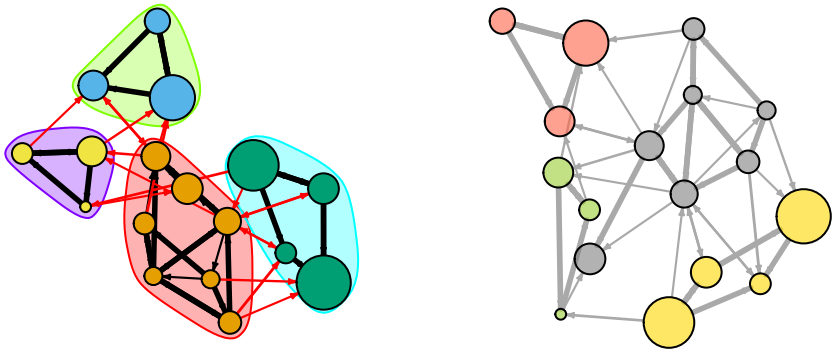
```
cut.off <- mean(links$weight)
net.sp <- delete_edges(net, E(net)[weight<cut.off])
plot(net.sp, layout=layout_with_kk)
```



Another way to think about this is to plot the two tie types (hyperlink & mention) separately. We will do that in section 5 of this tutorial: Plotting multiplex networks.

We can also try to make the network map more useful by showing the communities within it:

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

# Community detection (by optimizing modularity over partitions):
clp <- cluster_optimal(net)
class(clp)

# Community detection returns an object of class "communities"
# which igraph knows how to plot:
plot(clp, net)

# We can also plot the communities without relying on their built-in plot:
V(net)$community <- clp$membership
colrs <- adjustcolor( c("gray50", "tomato", "gold", "yellowgreen"), alpha=.6)
plot(net, vertex.color=colrs[V(net)$community])
```



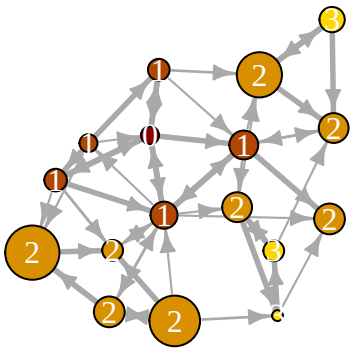
```
dev.off()
```

4.4 *Highlighting specific nodes or links*

Sometimes we want to focus the visualization on a particular node or a group of nodes. In our example media network, we can examine the spread of information from focal actors. For instance, let's represent distance from the NYT.

The `distances` function returns a matrix of shortest paths from nodes listed in the `v` parameter to ones included in the `to` parameter.

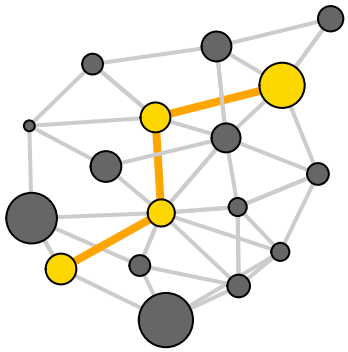
```
dist.from.NYT <- distances(net, v=V(net)[media=="NY Times"],
                           to=V(net), weights=NA)

# Set colors to plot the distances:
oranges <- colorRampPalette(c("dark red", "gold"))
col <- oranges(max(dist.from.NYT)+1)
col <- col[dist.from.NYT+1]

plot(net, vertex.color=col, vertex.label=dist.from.NYT, edge.arrow.size=.6,
     vertex.label.color="white")
```



We can also highlight a path in the network:

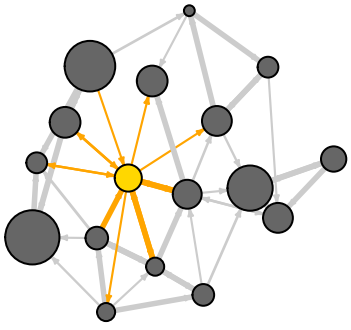
```
news.path <- shortest_paths(net,
                            from = V(net)[media=="MSNBC"],
                             to  = V(net)[media=="New York Post"],
                             output = "both") # both path nodes and edges

# Generate edge color variable to plot the path:
ecol <- rep("gray80", ecount(net))
ecol[unlist(news.path$epath)] <- "orange"
# Generate edge width variable to plot the path:
ew <- rep(2, ecount(net))
```

```
ew[unlist(news.path$epath)] <- 4
# Generate node color variable to plot the path:
vcol <- rep("gray40", vcount(net))
vcol[unlist(news.path$vpath)] <- "gold"

plot(net, vertex.color=vcol, edge.color=ecol,
     edge.width=ew, edge.arrow.mode=0)
```



We can highlight the edges going into or out of a vertex, for instance the WSJ. For a single node, use `incident()`, for multiple nodes use `incident_edges()`

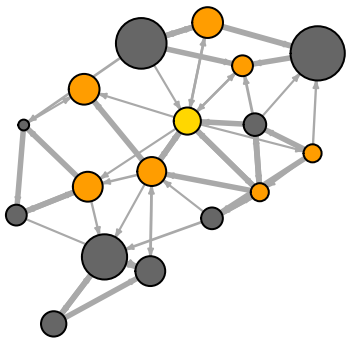
```
inc.edges <- incident(net,  V(net)[media=="Wall Street Journal"], mode="all")

# Set colors to plot the selected edges.
ecol <- rep("gray80", ecount(net))
ecol[inc.edges] <- "orange"
vcol <- rep("grey40", vcount(net))
vcol[V(net)$media=="Wall Street Journal"] <- "gold"
plot(net, vertex.color=vcol, edge.color=ecol)
```



We can also point to the immediate neighbors of a vertex, say WSJ. The `neighbors` function finds all nodes one step out from the focal actor.To find the neighbors for multiple nodes, use `adjacent_vertices()` instead of `neighbors()`. To find node neighborhoods going more than one

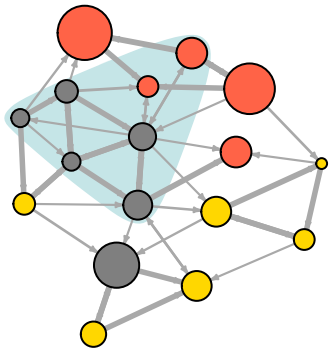
step out, use function `ego()` with parameter *`order`* set to the number of steps out to go from the focal node(s).

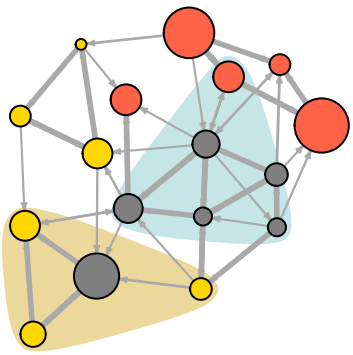
```
neigh.nodes <- neighbors(net, V(net)[media=="Wall Street Journal"], mode="out")

# Set colors to plot the neighbors:
vcol[neigh.nodes] <- "#ff9d00"
plot(net, vertex.color=vcol)
```



A way to draw attention to a group of nodes (we saw this before with communities) is to "mark" them:

```
par(mfrow=c(1,2))
plot(net, mark.groups=c(1,4,5,8), mark.col="#C5E5E7", mark.border=NA)

# Mark multiple groups:
plot(net, mark.groups=list(c(1,4,5,8), c(15:17)),
          mark.col=c("#C5E5E7","#ECD89A"), mark.border=NA)
```



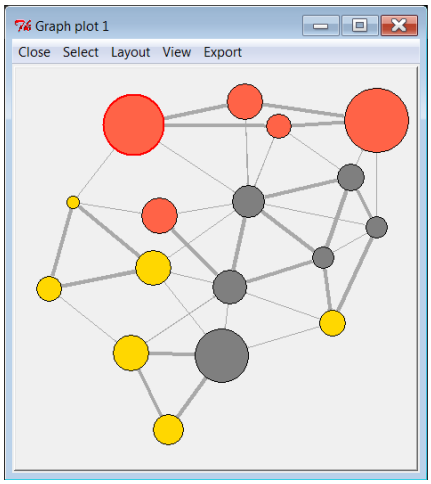


```
dev.off()
```

4.5 *Interactive plotting with tkplot*

R and igraph allow for interactive plotting of networks. This might be a useful option for you if you want to tweak slightly the layout of a small graph. After adjusting the layout manually, you can get the coordinates of the nodes and use them for other plots.

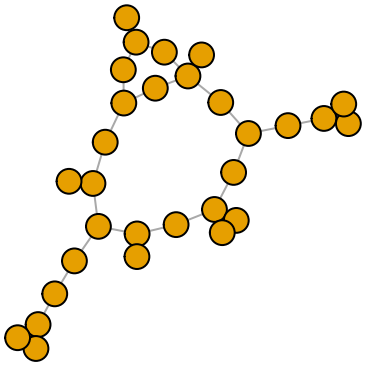
```
tkid <- tkplot(net) #tkid is the id of the tkplot that will open
l <- tkplot.getcoords(tkid) # grab the coordinates from tkplot
plot(net, layout=l)
```



4.6 *Plotting two-mode networks*

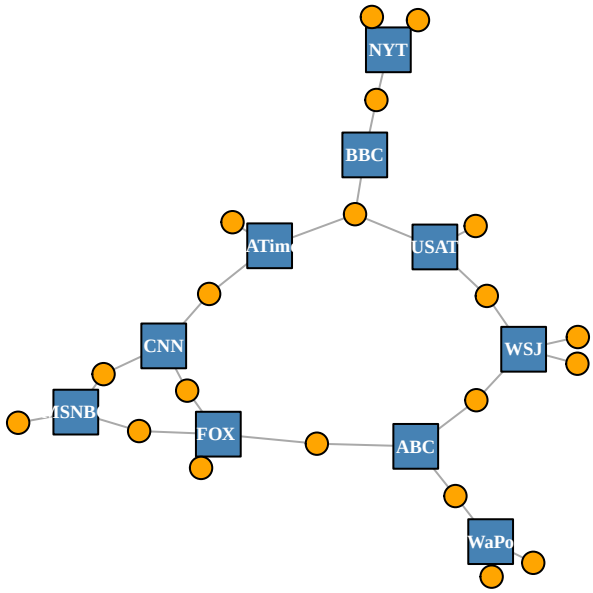
As you might remember, our second media example is a two-mode network examining links between news sources and their consumers.

```
head(nodes2)
head(links2)
plot(net2, vertex.label=NA)
```



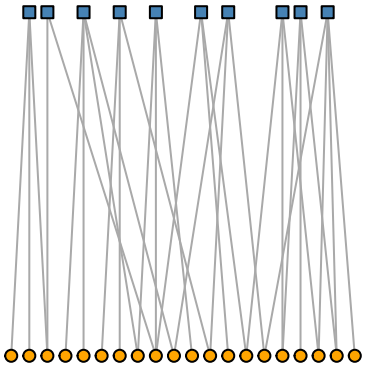
As with one-mode networks, we can modify the network object to include the visual properties that will be used by default when plotting the network. Notice that this time we will also change the shape of the nodes - media outlets will be squares, and their users will be circles.

```
# Media outlets are blue squares, audience nodes are orange circles:
V(net2)$color <- c("steel blue", "orange")[V(net2)$type+1]
V(net2)$shape <- c("square", "circle")[V(net2)$type+1]

# Media outlets will have name labels, audience members will not:
V(net2)$label <- ""
V(net2)$label[V(net2)$type==F] <- nodes2$media[V(net2)$type==F]
V(net2)$label.cex=.6
V(net2)$label.font=2

plot(net2, vertex.label.color="white", vertex.size=(2-V(net2)$type)*8)
```



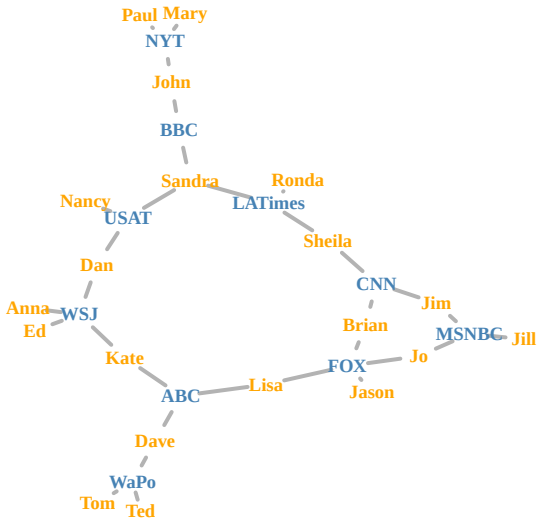
In `igraph`, there is also a special layout for bipartite networks (though it doesn't always work great, and you might be better off generating your own two-mode layout).

```
plot(net2, vertex.label=NA, vertex.size=7, layout=layout_as_bipartite)
```



Using text as nodes may be helpful at times:

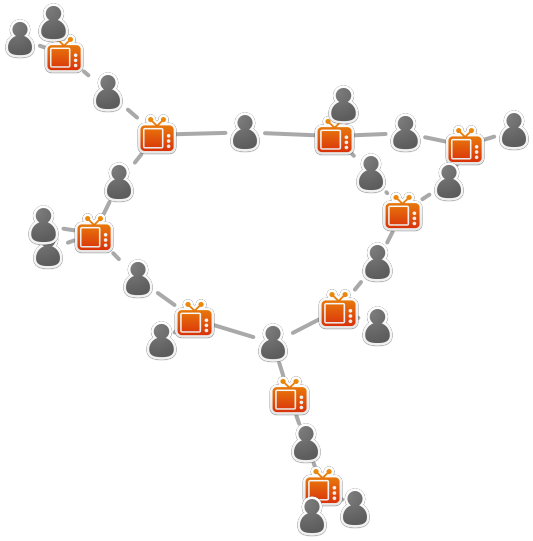
```
plot(net2, vertex.shape="none", vertex.label=nodes2$media,
     vertex.label.color=V(net2)$color, vertex.label.font=2,
     vertex.label.cex=.6, edge.color="gray70",  edge.width=2)
```



In this example, we will also experiment with the use of images as nodes. In order to do this, you will need the `png` package (if missing, install with `install.packages('png')`

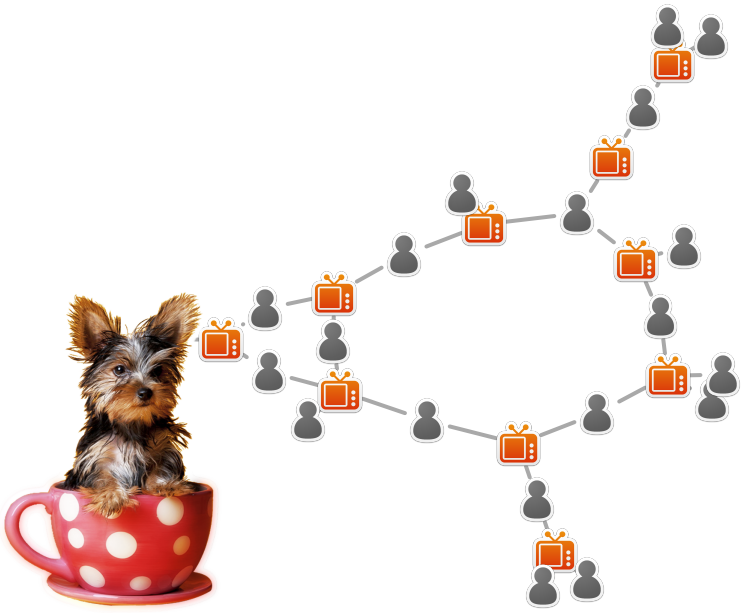
```
# install.packages('png')
library('png')

img.1 <- readPNG("./images/news.png")
img.2 <- readPNG("./images/user.png")

V(net2)$raster <- list(img.1, img.2)[V(net2)$type+1]
```

```
plot(net2, vertex.shape="raster", vertex.label=NA,
     vertex.size=16, vertex.size2=16, edge.width=2)
```



By the way, we can also add any image we want to a plot. For example, many network graphs can be largely improved by a photo of a puppy in a teacup.

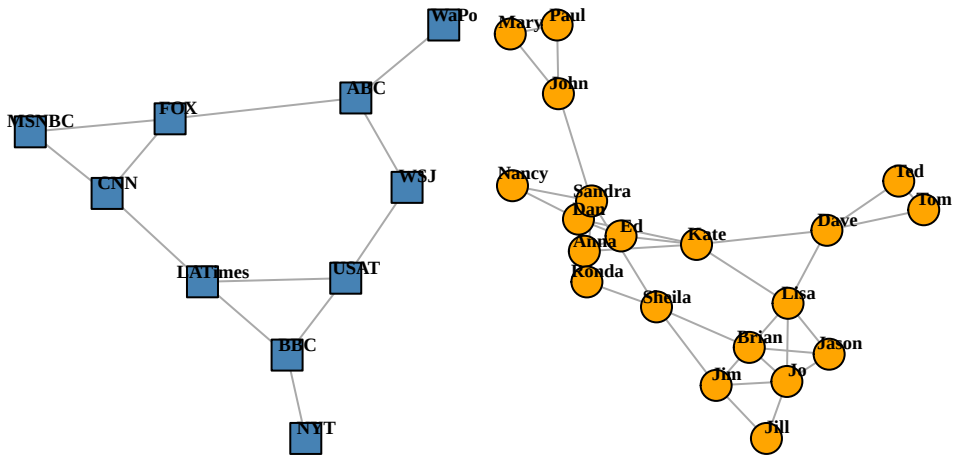
```
plot(net2, vertex.shape="raster", vertex.label=NA,
     vertex.size=16, vertex.size2=16, edge.width=2)

img.3 <- readPNG("./images/puppy.png")
rasterImage(img.3,  xleft=-1.6, xright=-0.6, ybottom=-1.1, ytop=0.1)
```



```
# The numbers after the image are its coordinates
# The limits of your plotting area are given in par()$usr
```

We can also generate and plot bipartite projections for the two-mode network: co-memberships are easy to calculate by multiplying the network matrix by its transposed matrix, or using igraph's `bipartite.projection()` function.

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

net2.bp <- bipartite.projection(net2)

plot(net2.bp$proj1, vertex.label.color="black", vertex.label.dist=1,
     vertex.label=nodes2$media[!is.na(nodes2$media.type)])

plot(net2.bp$proj2, vertex.label.color="black", vertex.label.dist=1,
     vertex.label=nodes2$media[ is.na(nodes2$media.type)])
```




```
dev.off()
```

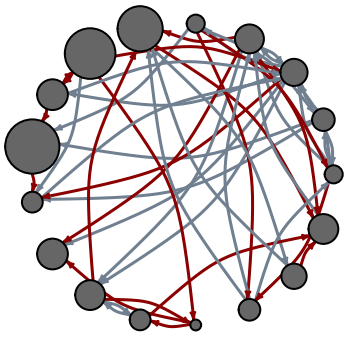
4.7 *Plotting multiplex networks*

In some cases, the networks we want to plot are *multigraphs*: they can have multiple edges connecting the same two nodes. A related concept, *multiplex* networks, contain multiple types of ties. For instance, we can represent friendship, romantic, and work relationships between individuals in a

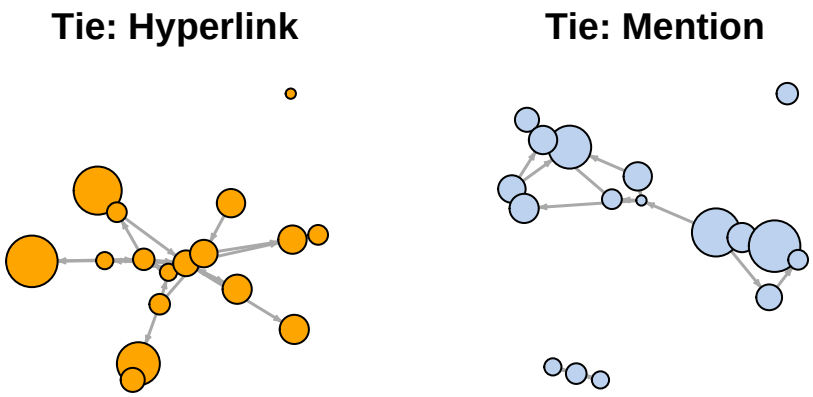
single multiplex network.

In our example network, we also have two tie types: hyperlinks and mentions. One thing we can do with them is plot each type of tie separately:

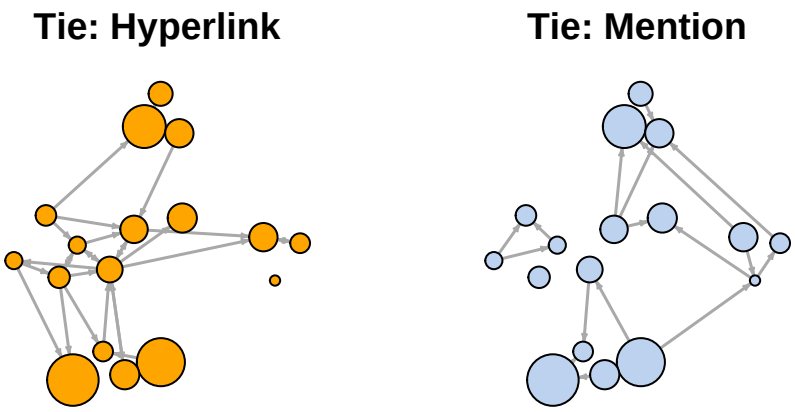
```
E(net)$width <- 1.5
plot(net, edge.color=c("dark red", "slategrey")[(E(net)$type=="hyperlink")+1],
      vertex.color="gray40", layout=layout_in_circle, edge.curved=.3)
```



```
net.m <- net - E(net)[E(net)$type=="hyperlink"] # another way to delete edges:
net.h <- net - E(net)[E(net)$type=="mention"]   # using the minus operator

# Plot the two links separately:
par(mfrow=c(1,2))
plot(net.h, vertex.color="orange", layout=layout_with_fr, main="Tie: Hyperlink")
plot(net.m, vertex.color="lightsteelblue2", layout=layout_with_fr, main="Tie: Mention")
```




```
# Make sure the nodes stay in place in both plots:
l <- layout_with_fr(net)
plot(net.h, vertex.color="orange", layout=l, main="Tie: Hyperlink")
plot(net.m, vertex.color="lightsteelblue2", layout=l, main="Tie: Mention")
```

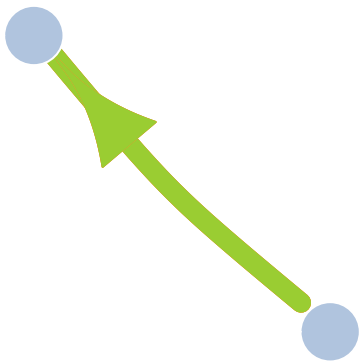



```
dev.off()
```

In our example network, it so happens that we do not have node dyads connected by multiple types of connections. That is to say, we never have both a 'hyperlink' and a 'mention' tie between the same two news outlets. However, this could easily happen in a multiplex network.

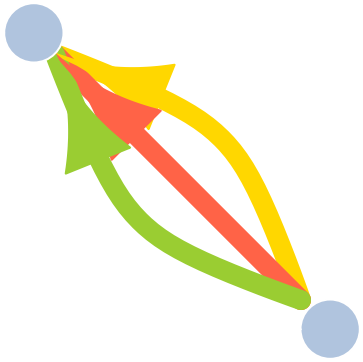
One challenge in visualizing multigraphs is that multiple edges between the same two nodes may get plotted on top of each other in a way that makes impossible to see them clearly. For example, let us generate a very simple multiplex network with two nodes and three ties between them:

```
multigtr <- graph( edges=c(1,2, 1,2, 1,2), n=2 )
l <- layout_with_kk(multigtr)

# Let's just plot the graph:
plot(multigtr, vertex.color="lightsteelblue", vertex.frame.color="white",
     vertex.size=40, vertex.shape="circle", vertex.label=NA,
     edge.color=c("gold", "tomato", "yellowgreen"), edge.width=10,
     edge.arrow.size=3, edge.curved=0.1, layout=l)
```



Because all edges in the graph have the same curvature, they are drawn over each other so that we only see one of them. What we can do is assign each edge a different curvature. One useful function in `igraph` called `curve_multiple` can help us here. For a graph `G`, `curve.multiple(G)` will generate a curvature for each edge that maximizes visibility.

```
plot(multigtr, vertex.color="lightsteelblue", vertex.frame.color="white",
     vertex.size=40, vertex.shape="circle", vertex.label=NA,
     edge.color=c("gold", "tomato", "yellowgreen"), edge.width=10,
     edge.arrow.size=3, edge.curved=curve_multiple(multigtr), layout=l)
```



It is a good practice to detach packages when we stop needing them. Try to remember that especially with `igraph` and the `statnet` family packages, as bad things tend to happen if you have them loaded together.

```
detach('package:igraph')
```

---

5 Beyond *igraph*: *Statnet*, *ggraph*, and simple charts

The `igraph` package is only one of many available network visualization options in `R`. This section provides a few quick examples illustrating other available approaches to static network visualization.

5.1 *A network package example (for Statnet users)*

Plotting with the `network` package is very similar to that with `igraph` - although the notation is slightly different (a whole new set of parameter names!). This package also uses less default controls obtained by modifying the network object, and more explicit parameters in the plotting function.

Here is a quick example using the (by now familiar) media network. We will begin by converting the data into the `network` format used by the Statnet family of packages (including `network`, `sna`, `ergm`, `stergm`, and others).

As in `igraph`, we can generate a 'network' object from an edge list, an adjacency matrix, or an incidence matrix. You can get the specifics with `?edgeset.constructors`. Here we will use the edge list and the node attribute data frames to create the network object. One specific thing to pay attention to here is the `ignore.eval` parameter. It is set to `TRUE` by default, and that setting causes the network object to disregard edge weights.

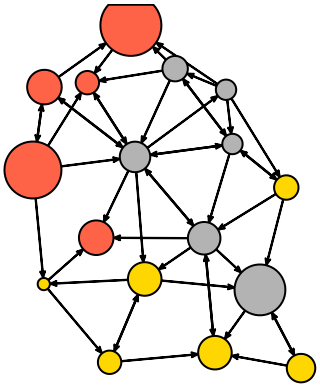
```
library('network')

net3 <- network(links,  vertex.attr=nodes, matrix.type="edgelist",
                loops=F, multiple=F, ignore.eval = F)
```

Here again we can easily access the edges, vertices, and the network matrix:

```
net3[,]
net3 %n% "net.name" <- "Media Network" #  network attribute
net3 %v% "media"    # Node attribute
net3 %e% "type"     # Node attribute
```

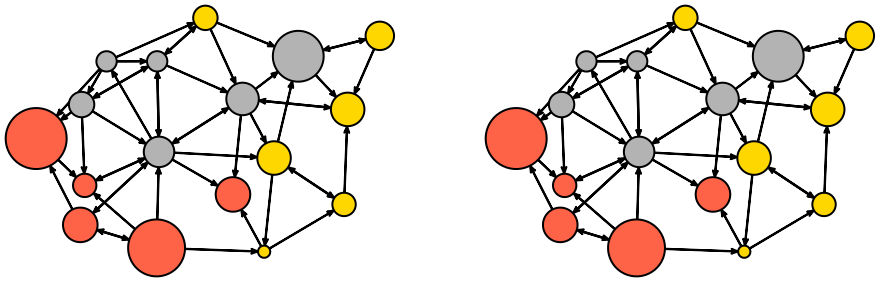
Let's plot our media network once again:

```
net3 %v% "col" <- c("gray70", "tomato", "gold")[net3 %v% "media.type"]
plot(net3, vertex.cex=(net3 %v% "audience.size")/7, vertex.col="col")
```



Note that - as in igraph - the plot returns the node position coordinates. You can use them in other plots using the `coord` parameter.

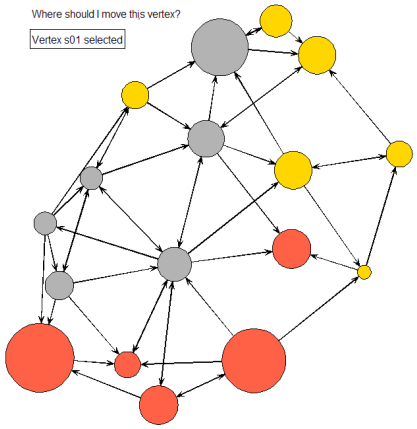
```
l <- plot(net3, vertex.cex=(net3 %v% "audience.size")/7, vertex.col="col")
plot(net3, vertex.cex=(net3 %v% "audience.size")/7, vertex.col="col", coord=l)
```



```
detach('package:network')
```

The network package also offers the option to edit a plot interactively, by setting the parameter `interactive=T`:

```
plot(net3, vertex.cex=(net3 %v% "audience.size")/7, vertex.col="col", interactive=T)
```



For a full list of parameters that you can use in the `network` package, check out `?plot.network`.

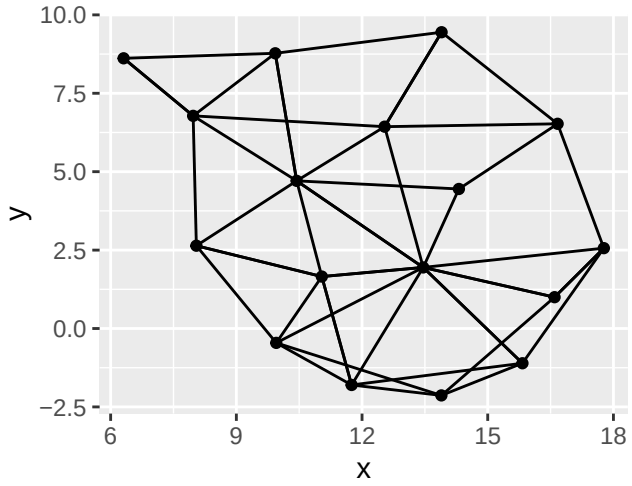
5.2 *A ggraph package example (for ggplot2 users)*

The `ggplot2` package and its extensions are known for offering the most meaningfully structured and advanced way to visualize data in R. In `ggplot2`, you can select from a variety of visual building blocks and add them to your graphics one by one, a layer at a time.

The `ggraph` package takes this principle and extends it to network data. In this section, we'll only cover the basics without providing a detailed overview of the grammar of graphics approach. For a deeper look, it would be best to get familiar with `ggplot2` first, then learn the specifics of `ggraph`.

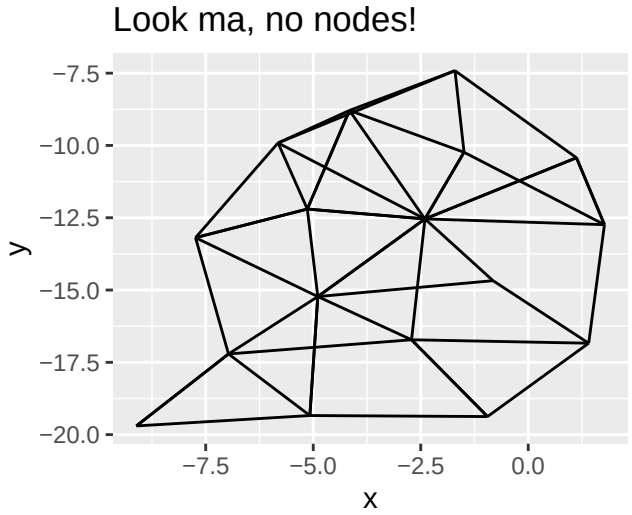
One good news is that we can use our `igraph` objects directly with the `ggraph` package. The following code gets the data and adds separate layers for nodes and links.

```
library(ggraph)
library(igraph)

ggraph(net) +
  geom_edge_link() +   # add edges to the plot
  geom_node_point()    # add nodes to the plot
```



You will also recognize here some network layouts familiar from `igraph` plotting: 'star', 'circle', 'grid', 'sphere', 'kk', 'fr', 'mds', 'lgl', etc.

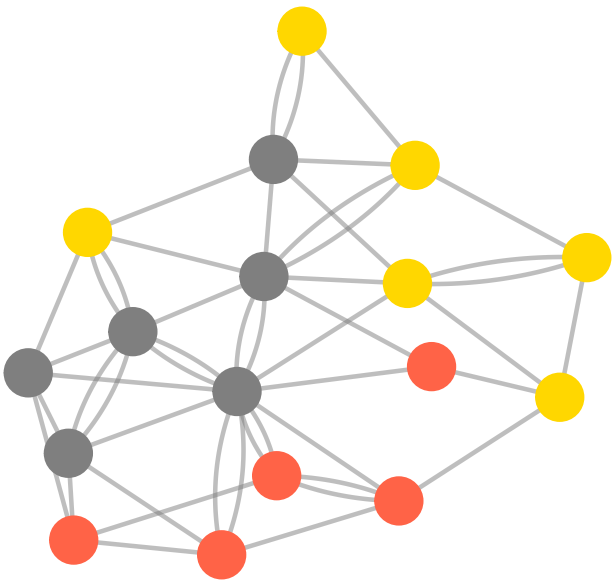
```
ggraph(net, layout="lgl") +
  geom_edge_link() +
  ggtitle("Look ma, no nodes!")  # add title to the plot
```



Here we can use `geom_edge_link()` for straight edges, `geom_edge_arc()` for curved ones, and `geom_edge_fan()` when we want to make sure any overlapping multiplex edges will be fanned out.

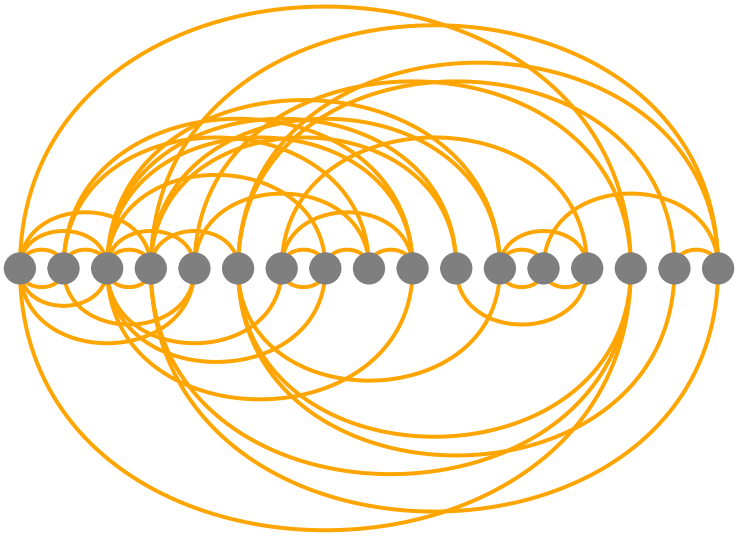
As in other packages, we can set visual properties for the network plot by using key function parameters. For instance, nodes have `color`, `fill`, `shape`, `size`, and `stroke`. Edges have `color`, `width`, and `linetype`. Here too the `alpha` parameter controls transparency.

```
ggraph(net, layout="lgl") +
  geom_edge_fan(color="gray50", width=0.8, alpha=0.5) +
  geom_node_point(color=V(net)$color, size=8) +
  theme_void()
```



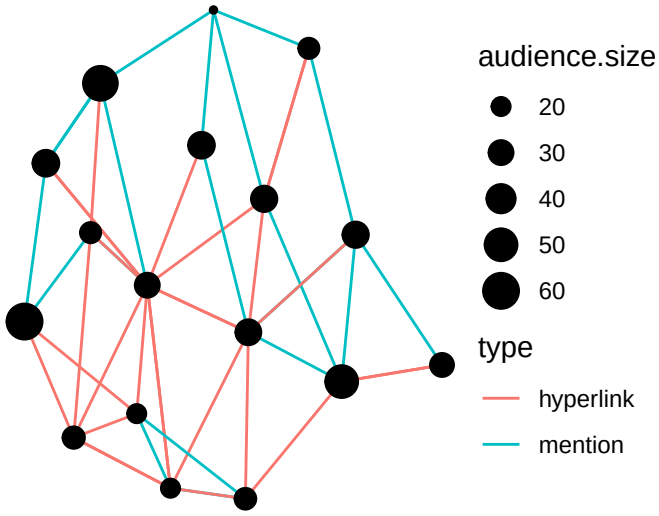
As in ggplot2, we can add different themes to the plot. For a cleaner look, you can use a minimal or empty theme with `theme_minimal()` or `theme_void()`.

```
ggraph(net, layout = 'linear') +
    geom_edge_arc(color = "orange", width=0.7) +
    geom_node_point(size=5, color="gray50") +
    theme_void()
```



The ggraph package also uses the traditional ggplot2 way of mapping aesthetics: that is to say, of specifying which elements of the data should correspond to different visual properties of the graphic. This is done using the `aes()` function that matches visual parameters with attribute names from the data. In the code below, the edge attribute `type` and node attribute `audience.size` are taken from our data as they are included in the `igraph` object.

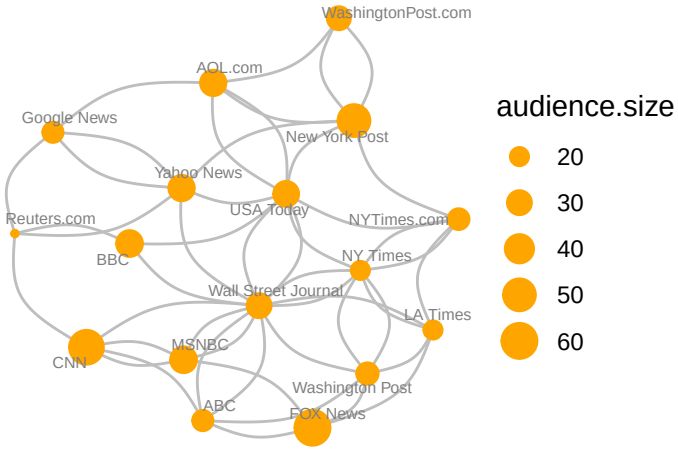
```
ggraph(net, layout="lgl") +
  geom_edge_link(aes(color = type)) +           # colors by edge type
  geom_node_point(aes(size = audience.size)) +  # size by audience size
  theme_void()
```



One great thing about `ggplot2` and `ggraph` you can see above is that they automatically generate a legend which makes plots easier to interpret.

We can add a layer with node labels using `geom_node_text()` or `geom_node_label()` which correspond to similar functions in `ggplot2`.

```
ggraph(net,  layout = 'lgl') +
  geom_edge_arc(color="gray", curvature=0.3) +
  geom_node_point(color="orange", aes(size = audience.size)) +
  geom_node_text(aes(label = media), size=2, color="gray50", repel=T) +
  theme_void()
```



```
detach("package:ggraph")
```

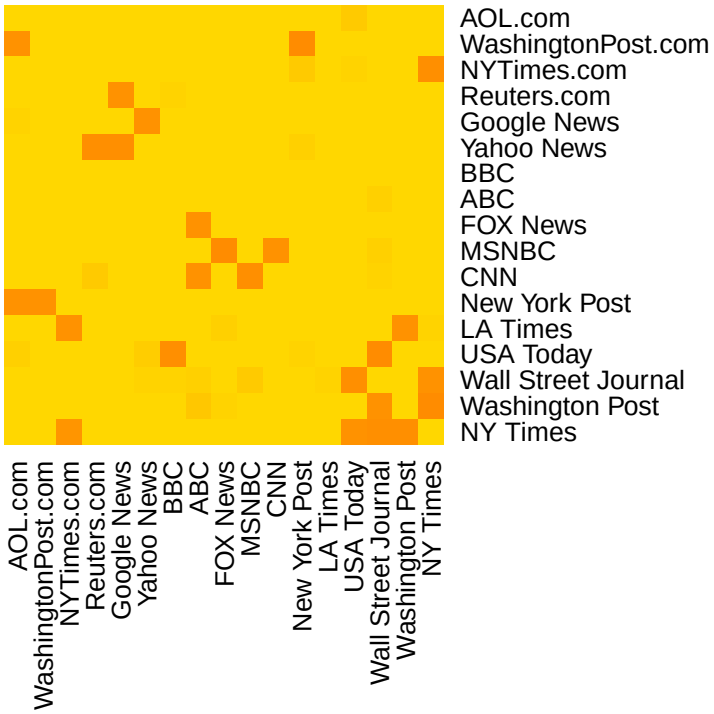
While those are not discussed here, note that `ggraph` offers a number of other interesting ways to represent networks, including dendrograms, treemaps, hive plots, and circle plots.

5.3 *Other ways to represent a network*

At this point it might be useful to provide a quick reminder that there are many ways to represent a network not limited to a hairball plot.

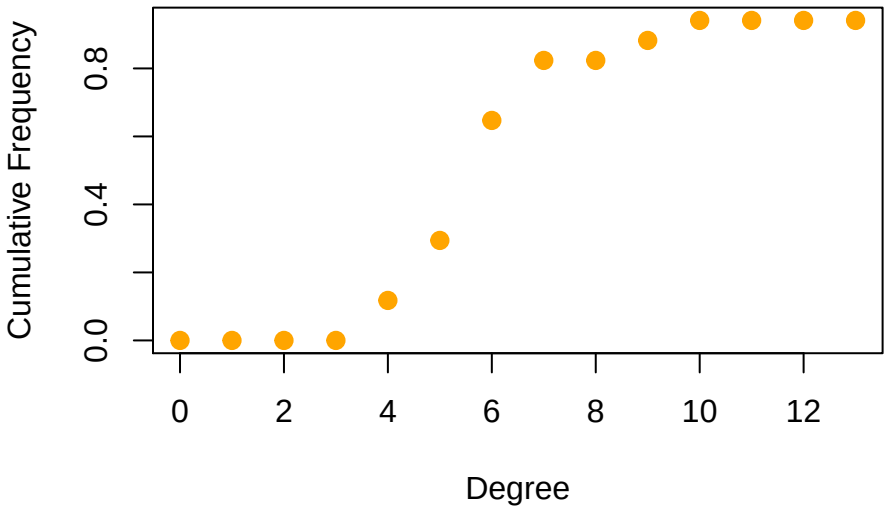
For example, here is a quick heatmap of the network matrix:

```
netm <- get.adjacency(net, attr="weight", sparse=F)
colnames(netm) <- V(net)$media
rownames(netm) <- V(net)$media

palf <- colorRampPalette(c("gold", "dark orange"))
heatmap(netm[,17:1], Rowv = NA, Colv = NA, col = palf(100),
        scale="none", margins=c(10,10) )
```



Depending on what properties of the network or its nodes and edges are most important to you, simple graphs can often be more informative than network maps.

```r
# Plot the egree distribution for our network:
deg.dist <- degree_distribution(net, cumulative=T, mode="all")
plot( x=0:max(degree(net)), y=1-deg.dist, pch=19, cex=1.2, col="orange",
      xlab="Degree", ylab="Cumulative Frequency")
```



---

6 Interactive network visualizations

6.1 *Simple plot animations in R*

If you have already installed "ndtv", you should also have a package used by it called "animation". If not, now is a good time to install it with `install.packages('animation')`. Note that this package provides a simple technique to create various (not necessarily network-related) animations in R. It works by generating multiple plots and combining them in an animated GIF.

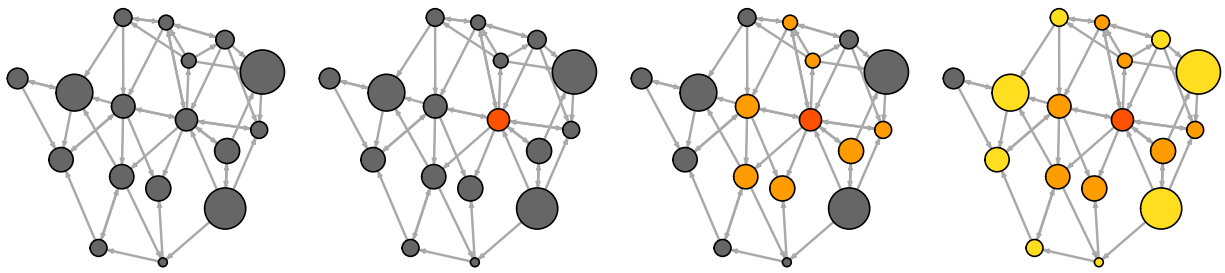
The catch here is that in order for this to work, you need not only the R package, but also an additional software called ImageMagick (http://imagemagick.org). You probably don't want to install that during the workshop, but you can try it at home.

The good news is that once you figure this out, you can turn any series of R plots (network or not!) into an animated GIF.

```r
library('animation')
library('igraph')

ani.options("convert") # Check that the package knows where to find ImageMagick
# If it doesn't know where to find it, give it the correct path for your system.
ani.options(convert="C:/Program Files/ImageMagick-6.8.8-Q16/convert.exe")
```

We will now generate 4 network plots (the same way we did before), only this time we'll do it within the `saveGIF` command. The animation interval is set with `interval`, and the `movie.name` parameter controls name of the gif.

```
l <- layout_with_lgl(net)

saveGIF( {  col <- rep("grey40", vcount(net))
            plot(net, vertex.color=col, layout=l)

            step.1 <- V(net)[media=="Wall Street Journal"]
            col[step.1] <- "#ff5100"
            plot(net, vertex.color=col, layout=l)

            step.2 <- unlist(neighborhood(net, 1, step.1, mode="out"))
            col[setdiff(step.2, step.1)] <- "#ff9d00"
            plot(net, vertex.color=col, layout=l)

            step.3 <- unlist(neighborhood(net, 2, step.1, mode="out"))
            col[setdiff(step.3, step.2)] <- "#FFDD1F"
            plot(net, vertex.color=col, layout=l)  },
          interval = .8, movie.name="network_animation.gif" )

 detach('package:igraph')
 detach('package:animation')
```



6.2 *Interactive JS visualization with visNetwork*

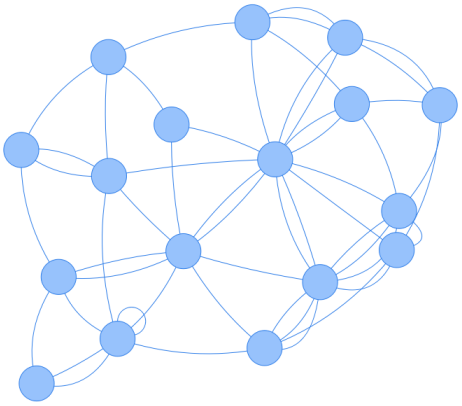
These days it is fairly easy to export R plots to HTML/JavaScript output. There are a number of packages like `rcharts` and `htmlwidgets` that can help you create interactive web charts right from R. One thing to keep in mind though is that the network visualizations created that way are most helpful as a starting point for further work. If you know a little bit of javascript, you can use them

as a first step and tweak the results to get closer to what you want.

Here we will take a quick look at `visNetwork` which generates interactive network visualizations using the `vis.js` javascript library. You can install the package with `install.packages('visNetwork')`.

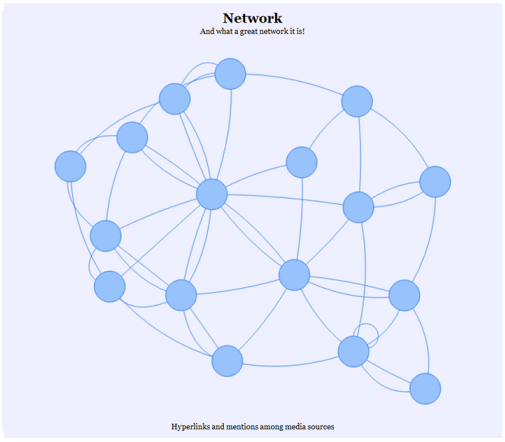
We can visualize our media network right away: `visNetwork()` will accept our node and link data frames. As usual, the node data frame needs to have an `id` column, and the link data needs to have `from` and `to` columns denoting the start and end of each tie.

```
library('visNetwork')
visNetwork(nodes, links)
```



If we want to set specific height and width for the interactive plot, we can do that with the `height` and `width` parameters. As is often the case in `R`, the title of the plot is set with the `main` parameter. The subtitle and footer can be set with `submain` and `footer` respectively.

```
visNetwork(nodes, links, height="600px", width="100%", background="#eeefff",
           main="Network", submain="And what a great network it is!",
           footer= "Hyperlinks and mentions among media sources")
```



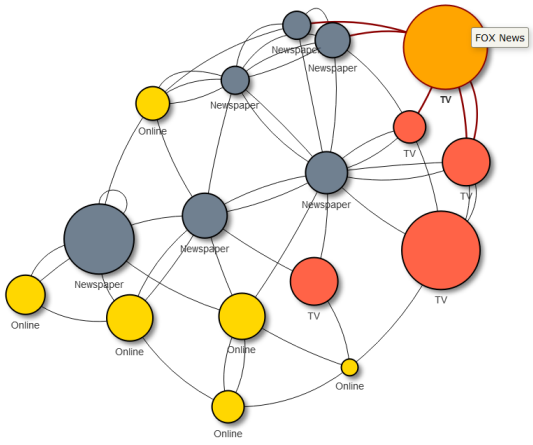
Like the `igraph` package, `visNetwork` allows us to set graphic properties as node or edge attributes. We can simply add them as columns in our data before we call the `visNetwork()` function. Check

out the available options with:

```
?visNodes
?visEdges
```

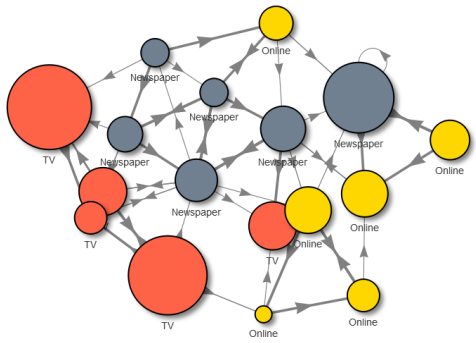
In the following code, we are changing some of the visual parameters for nodes. We start with the node shape (the available options for it include `ellipse`, `circle`, `database`, `box`, `text`, `image`, `circularImage`, `diamond`, `dot`, `star`, `triangle`, `triangleDown`, `square`, and `icon`). We are also going to change the color of several node elements. In this package, `background` controls the node color, `border` changes the frame color; `highlight` sets the color on mouse click, and `hover` sets the color on mouseover.

```
# We'll start by adding new node and edge attributes to our dataframes.
vis.nodes <- nodes
vis.links <- links

vis.nodes$shape  <- "dot"
vis.nodes$shadow <- TRUE # Nodes will drop shadow
vis.nodes$title  <- vis.nodes$media # Text on click
vis.nodes$label  <- vis.nodes$type.label # Node label
vis.nodes$size   <- vis.nodes$audience.size # Node size
vis.nodes$borderWidth <- 2 # Node border width

vis.nodes$color.background <- c("slategrey", "tomato", "gold")[nodes$media.type]
vis.nodes$color.border <- "black"
vis.nodes$color.highlight.background <- "orange"
vis.nodes$color.highlight.border <- "darkred"

visNetwork(vis.nodes, vis.links)
```



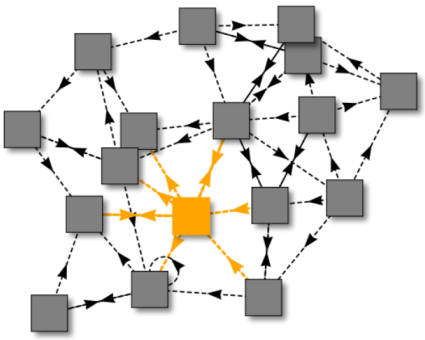
Next we will change some of the visual properties of the edges.

```
vis.links$width <- 1+links$weight/8 # line width
vis.links$color <- "gray"    # line color
vis.links$arrows <- "middle" # arrows: 'from', 'to', or 'middle'
vis.links$smooth <- FALSE    # should the edges be curved?
vis.links$shadow <- FALSE    # edge shadow

visnet <- visNetwork(vis.nodes, vis.links)
visnet
```



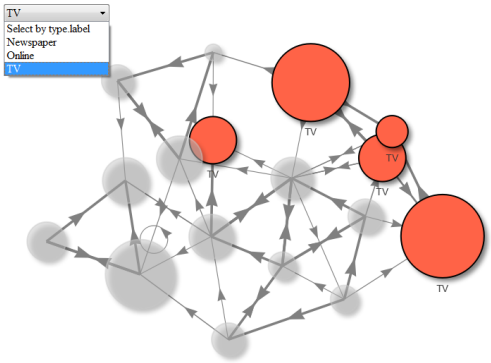
We can also set the visualization options directly with `visNodes()` and `visEdges()`.

```
visnet2 <- visNetwork(nodes, links)
visnet2 <- visNodes(visnet2, shape = "square", shadow = TRUE,
            color=list(background="gray", highlight="orange", border="black"))
visnet2 <- visEdges(visnet2, color=list(color="black", highlight = "orange"),
                    smooth = FALSE, width=2, dashes= TRUE, arrows = 'middle' )
visnet2
```



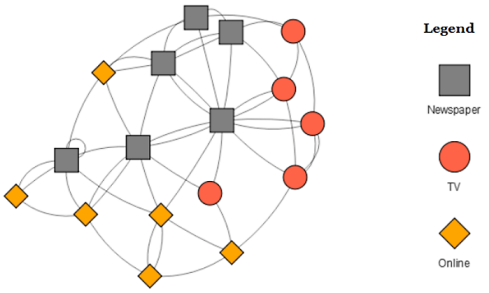
`visNetwork` offers a number of other options in the `visOptions()` function. For instance, we can highlight all neighbors of the selected node (`highlightNearest`), or add a drop-down menu to select subset of nodes (`selectedBy`). The subsets are based on a column from our data - here we use the type label.

```
visOptions(visnet, highlightNearest = TRUE, selectedBy = "type.label")
```



visNetwork can also work with predefined groups of nodes. The visual characteristics for nodes belonging in each group can be set with visGroups(). We can add an automatically generated group legend with visLegend().

```
nodes$group <- nodes$type.label
visnet3 <- visNetwork(nodes, links)
visnet3 <- visGroups(visnet3, groupname = "Newspaper", shape = "square",
                     color = list(background = "gray", border="black"))
visnet3 <- visGroups(visnet3, groupname = "TV", shape = "dot",
                     color = list(background = "tomato", border="black"))
visnet3 <- visGroups(visnet3, groupname = "Online", shape = "diamond",
                     color = list(background = "orange", border="black"))
visLegend(visnet3, main="Legend", position="right", ncol=1)
```



For more information, you can also check out:

```
?visOptions # available options
?visLayout  # available layouts
?visGroups  # using node groups
?visLegend  # adding a legend

# Detach the package since we're done with it.
```

```
detach('package:visNetwork')
```

6.3 *Interactive JS visualization with threejs*

Another good package exporting networks from R to javascript is `threejs`, which generates interactive network visualizations using the `three.js` javascript library and the `htmlwidgets` R package. One nice thing about `threejs` is that it can directly read `igraph` objects.

You can install the package with `install.packages('threejs')`. If you get errors or warnings using this library with the latest version of R, try also installing the development version of the `htmlwidgets` package which may have bug fixes that will help:

```
devtools::install_github('ramnathv/htmlwidgets')
```

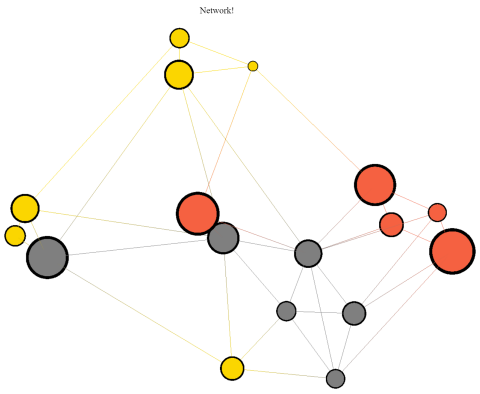
The main network plotting function here,`graphjs`, will take an `igraph` object. We could use our initial `net` object with a slight modification: we will delete its graph layout and let `threejs` generate one on its own. We cheated a bit earlier by assigning a function to the layout attribute in the `igraph` object rather than giving it a table of node coordinates. This is fine by `igraph`, but `threejs` will not let us do it.

```
library(threejs)
library(htmlwidgets)
library(igraph)

net.js <- net
graph_attr(net.js, "layout") <- NULL
```

Note that `RStudio` for Windows may not render the `threejs` graphics properly. We will save the output in an HTML file and open it in a browser. Some of the parameters that we can add include **main** for the plot title; **curvature** for the edge curvature; **bg** for background color; **showLabels** to set labels to visible (`TRUE`) or not (`FALSE`); **attraction** and **repulsion** to set how much nodes attract and repulse each other in the layout; **opacity** for node transparency (range 0 to 1); **stroke** to indicate whether nodes should be framed in a black circle (`TRUE`) or not (`FALSE`), etc.

For the full list of parameters, check out `?graphjs`.

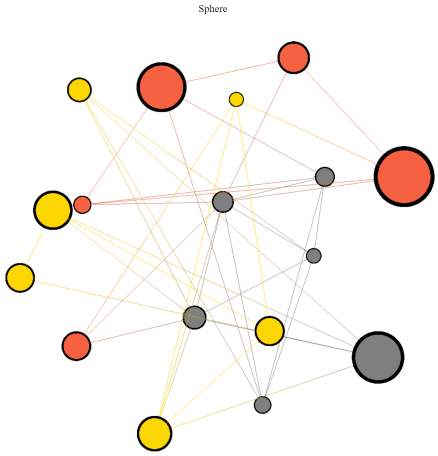
```
gjs <- graphjs(net.js, main="Network!", bg="gray10", showLabels=F, stroke=F,
               curvature=0.1, attraction=0.9, repulsion=0.8, opacity=0.9)
print(gjs)
saveWidget(gjs, file="Media-Network-gjs.html")
browseURL("Media-Network-gjs.html")
```



Once we open the resulting visualization in a browser, we can use the mouse *scrollwheel* to zoom in and out, the *left mouse button* to rotate the network, and the *right mouse button* to pan.

We can also create simple animations with `threejs` by using lists of layouts, vertex colors, and edge colors that will switch at each step.

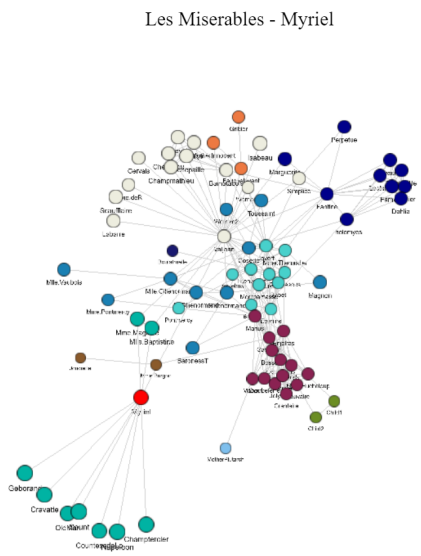
```
gjs.an <- graphjs(net.js, bg="gray10", showLabels=F, stroke=F,
                  layout=list(layout_randomly(net.js, dim=3),
                              layout_with_fr(net.js,  dim=3),
                              layout_with_drl(net.js, dim=3),
                              layout_on_sphere(net.js)),
                  vertex.color=list(V(net.js)$color, "gray", "orange",
                                    V(net.js)$color),
                  main=list("Random Layout", "Fruchterman-Reingold",
                            "DrL layout", "Sphere" ) )
print(gjs.an)
saveWidget(gjs.an, file="Media-Network-gjs-an.html")
browseURL("Media-Network-gjs-an.html")
```



As an additional example, we can take a look at the *Les Miserables* network included with the package:

```
data(LeMis)
lemis.net <- graphjs(LeMis, main="Les Miserables", showLabels=T)
print(lemis.net)
```

```
saveWidget(lemis.net, file="LeMis-Network-gjs.html")
browseURL("LeMis-Network-gjs.html")
```



6.4 *Interactive JS visualization with networkD3*

We will also take a quick look at networkD3 which - as its name suggests - generates interactive network visualizations using the D3 javascript library. If you d not have the `networkD3` library, install it with `install.packages("networkD3")`.

The data that this library needs from is is in the standard edge list form, with a few little twists. In order for things to work, the node IDs have to be numeric, and they also have to start from 0. An easy was to get there is to transform our character IDs to a factor variable, transform that to numeric, and make sure it starts from zero by subtracting 1.

```
library(networkD3)

links.d3 <- data.frame(from=as.numeric(factor(links$from))-1,
                       to=as.numeric(factor(links$to))-1 )
```

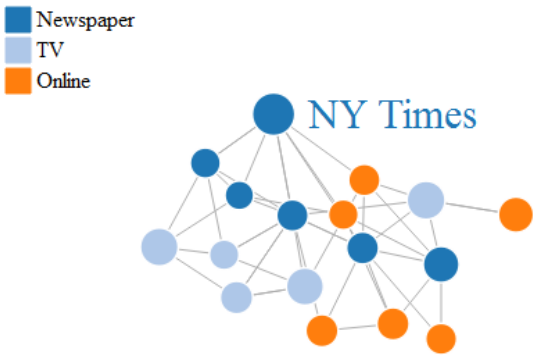
The nodes need to be in the same order as the "source" column in links:

```
nodes.d3 <- cbind(idn=factor(nodes$media, levels=nodes$media), nodes)
```

Now we can generate the interactive chart. The `Group` parameter in it is used to color the nodes. `Nodesize` is not (as one might think) the size of the node, but the number of the column in the node

data that should be used for sizing. The `charge` parameter controls node repulsion (if negative) or attraction (if positive).

```
forceNetwork(Links = links.d3, Nodes = nodes.d3, Source="from", Target="to",
             NodeID = "idn", Group = "type.label",linkWidth = 1,
             linkColour = "#afafaf", fontSize=12, zoom=T, legend=T,
             Nodesize=6, opacity = 1, charge=-600,
             width = 600, height = 600)
```



---

7 Dynamic network visualizations with *ndtv-d3*

7.1 *Interactive plots of static networks in ndtv*

Here we will create D3 visualizations using the `ndtv` package. You should not need additional software to produce web animations with `ndtv`. If you want to save the animations as video files (see `?saveVideo`), you have to install a video converter called FFmpeg (http://ffmpg.org). To find out how to get the right installation for your OS, check out `?install.ffmpeg`. To use all available layouts, you would also need to have Java installed on your machine.

```
install.packages('ndtv', dependencies=T)
```

As `ndtv` is part of the Statnet family, it will accept objects from the `network` package such as the one we created earlier (`net3`).

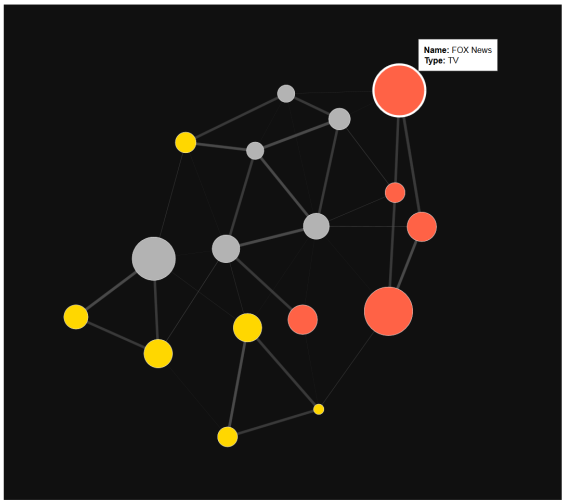
```
library('ndtv')
net3
```

Most of the parameters below are self-explanatory at this point (`bg` is the background color of the plot). Two new parameters we haven't used before are `vertex.tooltip` and `edge.tooltip`. Those contain the information that we can see when moving the mouse cursor over network elements. Note that the tooltip parameters accepts html tags – for example we will use the line break tag `<br>`.

The parameter `launchBrowser` instructs R to open the resulting visualization file (`filename`) in the browser.

```
render.d3movie(net3, usearrows = F, displaylabels = F, bg="#111111",
       vertex.border="#ffffff", vertex.col =  net3 %v% "col",
       vertex.cex = (net3 %v% "audience.size")/8,
       edge.lwd = (net3 %e% "weight")/3, edge.col = '#55555599',
       vertex.tooltip = paste("<b>Name:</b>", (net3 %v% 'media') , "<br>",
                              "<b>Type:</b>", (net3 %v% 'type.label')),
       edge.tooltip = paste("<b>Edge type:</b>", (net3 %e% 'type'), "<br>",
                            "<b>Edge weight:</b>", (net3 %e% "weight" ) ),
       launchBrowser=F, filename="Media-Network.html" )
```

If you are going to embed the plot in a markdown document, use `output.mode='inline'` above.



7.2 *Network evolution animations in ndtv*

Animations are a good way to show the evolution of small to medium size networks over time. At present, `ndtv` is the best R package for that – especially since it now has D3 capabilities and allows easy export for the Web.

In order to work with the network animations in `ndtv`, we need to understand `Statnet`'s dynamic network format, implemented in the `networkDynamic` package. The format can be used to represent longitudinal structures, both discrete (if you have multiple snapshots of your network at different time points) and continuous (if you have timestamps indicating when edges and/or nodes appear and disappear from the network). The examples below will only scratch the surface of temporal networks in Statnet - for a deeper dive, check out Skye Bender-deMoll's Temporal network tools tutorial and the networkDynamic package vignette.

Let's look at one example dataset included in the package, containing simulation data based on a network of business connections among Renaissance Florentine families:

```
data(short.stergm.sim)
short.stergm.sim
head(as.data.frame(short.stergm.sim))
```

```
##   onset terminus tail head onset.censored
## 1     0        1    3    5          FALSE
## 2    10       20    3    5          FALSE
## 3     0       25    3    6          FALSE
## 4     0        1    3    9          FALSE
## 5     2       25    3    9          FALSE
## 6     0        4    3   11          FALSE
```

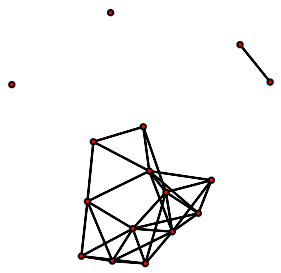
```
##   terminus.censored duration edge.id
## 1             FALSE        1       1
## 2             FALSE       10       1
## 3             FALSE       25       2
## 4             FALSE        1       3
## 5             FALSE       23       3
## 6             FALSE        4       4
```

What we see here is a temporal edge list. An edge goes from a node with ID in the `tail` column to a node with ID in the `head` column. Edges exist from time point `onset` to time point `terminus`. As you can see in our example, there may be multiple periods (*activity spells*) where an edge is present. Each of those periods is recorded on a separate row in the data frame above.

The idea of onset and terminus *censoring* refers to start and end points enforced by the beginning and end of network observation rather than by actual tie formation/dissolution.

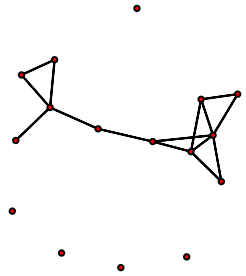
We can simply plot the network disregarding its time component (combining all nodes and edges that were ever present):

```
plot(short.stergm.sim)
```



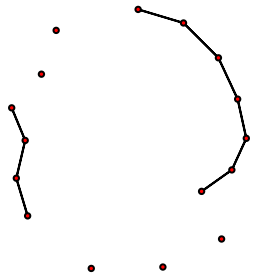
We can also use `network.extract()` to get a network that only contains elements active at a given point, or during a given time interval. For instance, we can plot the network at time 1 (`at=1`):

```
plot( network.extract(short.stergm.sim, at=1) )
```



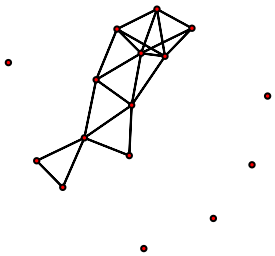
Plot nodes and edges that were active for the entire period (`rule=all`) from time 1 to time 5:

```
plot( network.extract(short.stergm.sim, onset=1, terminus=5, rule="all") )
```



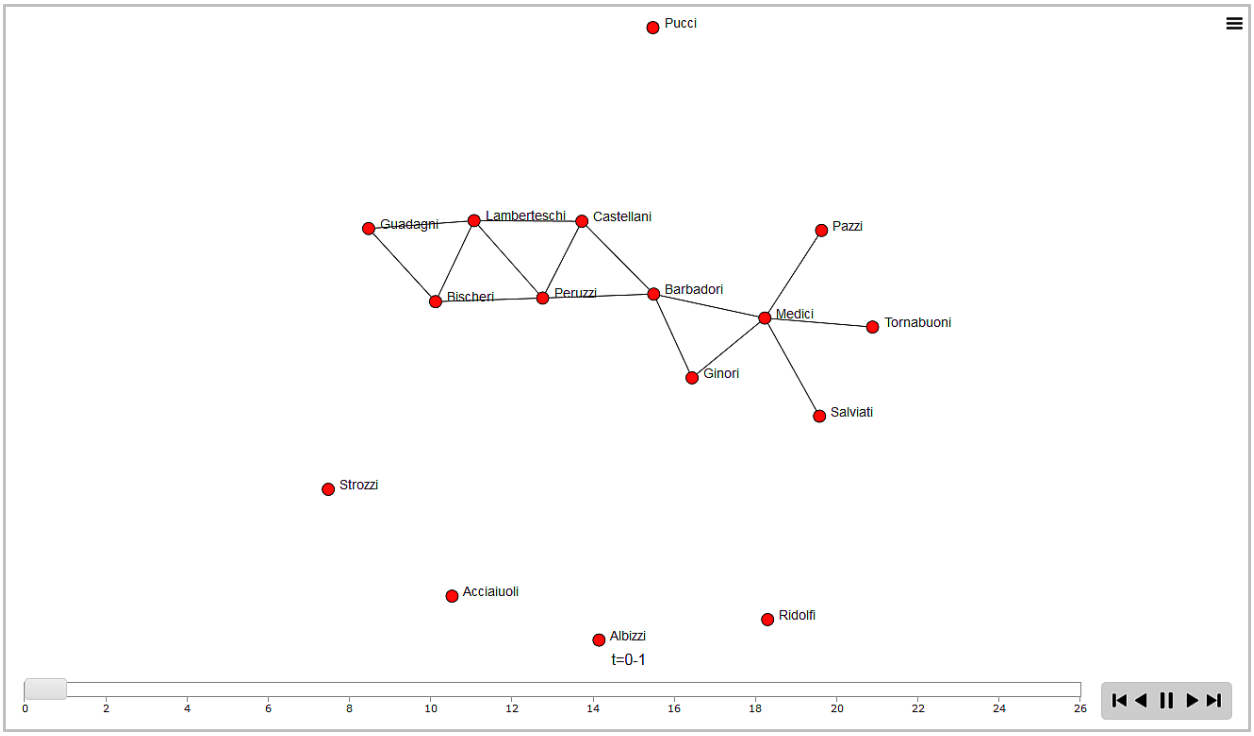
Plot nodes and edges that were active at any point (`rule=any`) between time 1 and time 10:

```
plot( network.extract(short.stergm.sim, onset=1, terminus=10, rule="any") )
```



Let's make a quick d3 animation from the example network:

```
render.d3movie(short.stergm.sim,displaylabels=TRUE)
```



Next, we will create and animate our own dynamic network. Dynamic network objects can be generated in a number of ways: from a set of networks/matrices representing different time points; from data frames/matrices with node lists and edge lists indicating when each is active, or when they switch state. You can check out `?networkDynamic` for more information.

We are going to add a time component to our media network example. The code below takes a 0-to-50 time interval and sets the nodes in the network as active throughout (time 0 to 50). The edges of the network appear one by one, and each one is active from their first activation until time point 50. We generate this longitudinal network using `networkDynamic` with our node times as `node.spells` and edge times as `edge.spells`.

```
vs <- data.frame(onset=0, terminus=50, vertex.id=1:17)
es <- data.frame(onset=1:49, terminus=50,
                 head=as.matrix(net3, matrix.type="edgelist")[,1],
                 tail=as.matrix(net3, matrix.type="edgelist")[,2])

net3.dyn <- networkDynamic(base.net=net3, edge.spells=es, vertex.spells=vs)
```

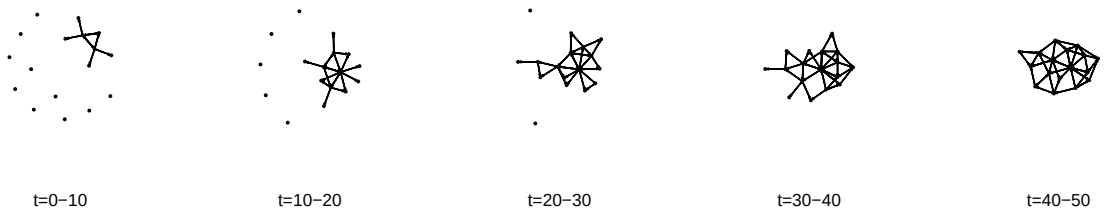
If we try to just plot the `networkDynamic` network, what we get is a combined network for the entire time period under observation – or as it happens, our original media example.

```
plot(net3.dyn, vertex.cex=(net3 %v% "audience.size")/7, vertex.col="col")
```

One way to show the network evolution is through static images from different time points. While

we can generate those one by one as we did above, `ndtv` offers an easier way. The command to do that is `filmstrip()`. As in the `par()` function controlling base R plot parameters, here `mfrow` sets the number of rows and columns in the multi-plot grid.

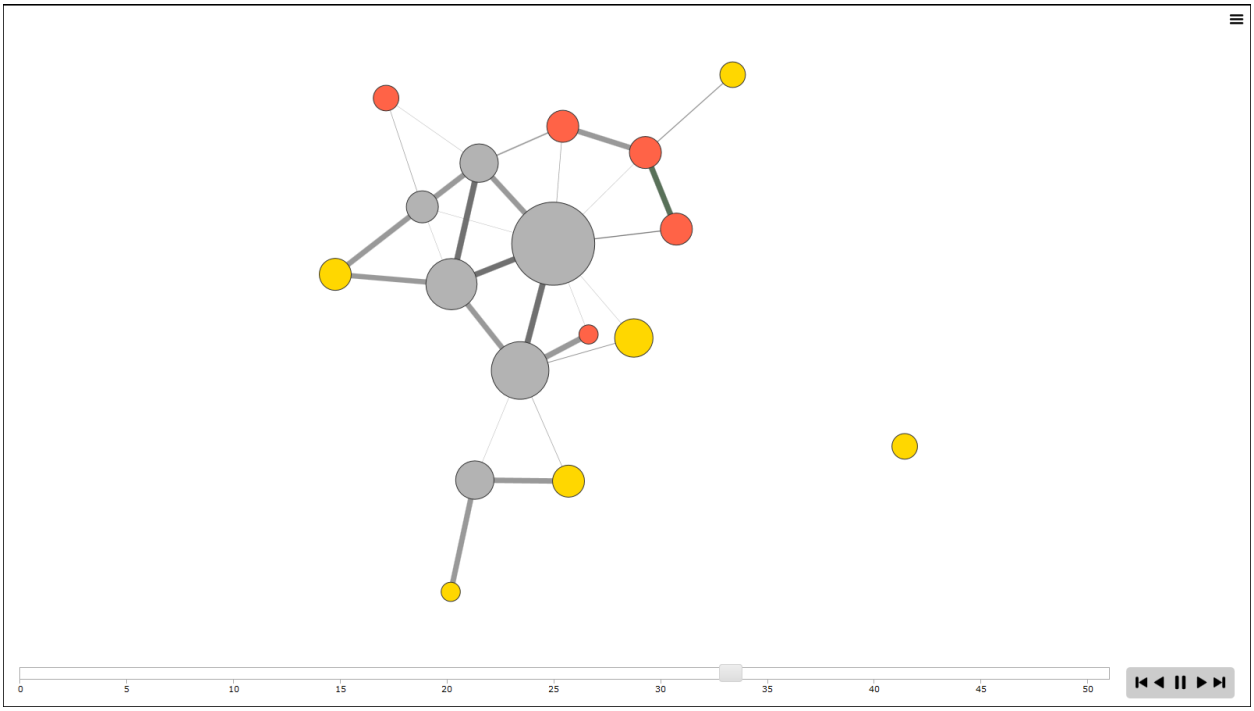
```
filmstrip(net3.dyn, displaylabels=F, mfrow=c(1, 5),
          slice.par=list(start=0, end=49, interval=10,
                         aggregate.dur=10, rule='any'))
```



Next, let's generate a network animation. We can pre-compute the coordinates for it (otherwise they get calculated when we generate the animation). Here `animation.mode` is the layout algorithm - one of "kamadakawai", "MDSJ", "Graphviz" and "useAttribute" (user-generated coordinates).

In `filmstrip()` above and in the animation computation below, `slice.par` is a list of parameters controlling how the network visualization moves through time. The parameter `interval` is the time step between layouts, `aggregate.dur` is the period shown in each layout, `rule` is the rule for displaying elements (e.g. `any`: active at any point during that period, `all`: active during the entire period, etc).

```
compute.animation(net3.dyn, animation.mode = "kamadakawai",
                  slice.par=list(start=0, end=50, interval=1,
                         aggregate.dur=1, rule='any'))

render.d3movie(net3.dyn, usearrows = F,
       displaylabels = F, label=net3 %v% "media",
       bg="#ffffff", vertex.border="#333333",
       vertex.cex = degree(net3)/2,
       vertex.col = net3.dyn %v% "col",
       edge.lwd = (net3.dyn %e% "weight")/3,
       edge.col = '#55555599',
       vertex.tooltip = paste("<b>Name:</b>", (net3.dyn %v% "media") , "<br>",
                              "<b>Type:</b>", (net3.dyn %v% "type.label")),
       edge.tooltip = paste("<b>Edge type:</b>", (net3.dyn %e% "type"), "<br>",
                            "<b>Edge weight:</b>", (net3.dyn %e% "weight" ) ),
       launchBrowser=T, filename="Media-Network-Dynamic.html",
       render.par=list(tween.frames = 30, show.time = F),
       plot.par=list(mar=c(0,0,0,0)), output.mode='inline' )
```

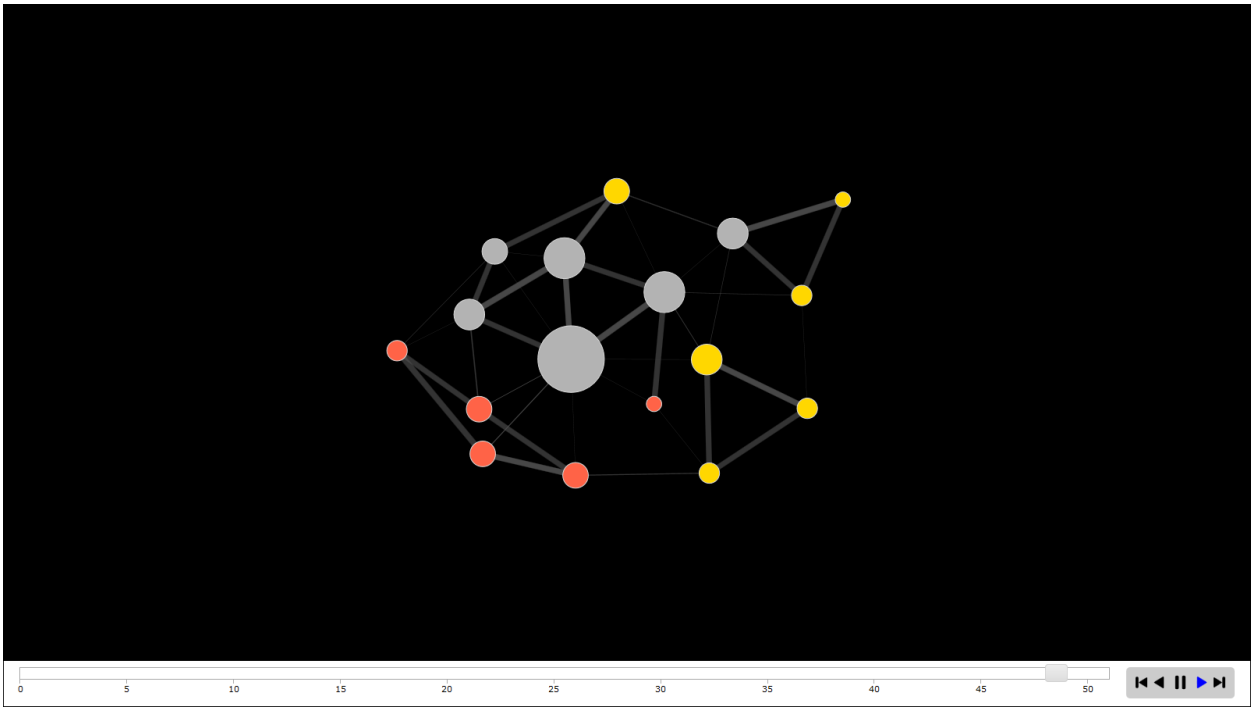


To embed this animation, we add the parameter `output.mode='inline'`.

In addition to dynamic nodes and edges, `ndtv` also takes dynamic attributes. We could have added those to the `es` and `vs` data frames above. However, the plotting function can also evaluate special parameters and generate dynamic arguments on the fly. For example, `function(slice) {` do some calculations with slice `}` will perform operations on the current time slice of the network, allowing us to change parameters dynamically.

See the node size below:

```
render.d3movie(net3.dyn, usearrows = F,
       displaylabels = F, label=net3 %v% "media",
       bg="#000000", vertex.border="#dddddd",
       vertex.cex = function(slice){ degree(slice)/2.5 },
       vertex.col = net3.dyn %v% "col",
       edge.lwd = (net3.dyn %e% "weight")/3,
       edge.col = '#55555599',
       vertex.tooltip = paste("<b>Name:</b>", (net3.dyn %v% "media") , "<br>",
                              "<b>Type:</b>", (net3.dyn %v% "type.label")),
       edge.tooltip = paste("<b>Edge type:</b>", (net3.dyn %e% "type"), "<br>",
                            "<b>Edge weight:</b>", (net3.dyn %e% "weight" ) ),
       launchBrowser=T, filename="Media-Network-even-more-Dynamic.html",
       render.par=list(tween.frames = 15, show.time = F), output.mode='inline',
       slice.par=list(start=0, end=50, interval=4, aggregate.dur=4, rule='any'))
```



---

8 Overlaying networks on geographic maps

The example presented in this section uses only base R and mapping packages. If you have experience with `ggplot2`, that package does provide a more versatile way of approaching this task. The code using `ggplot()` would be similar to what you will see below, but you would use 'borders()' to plot the map and 'geom_path()' for the edges.

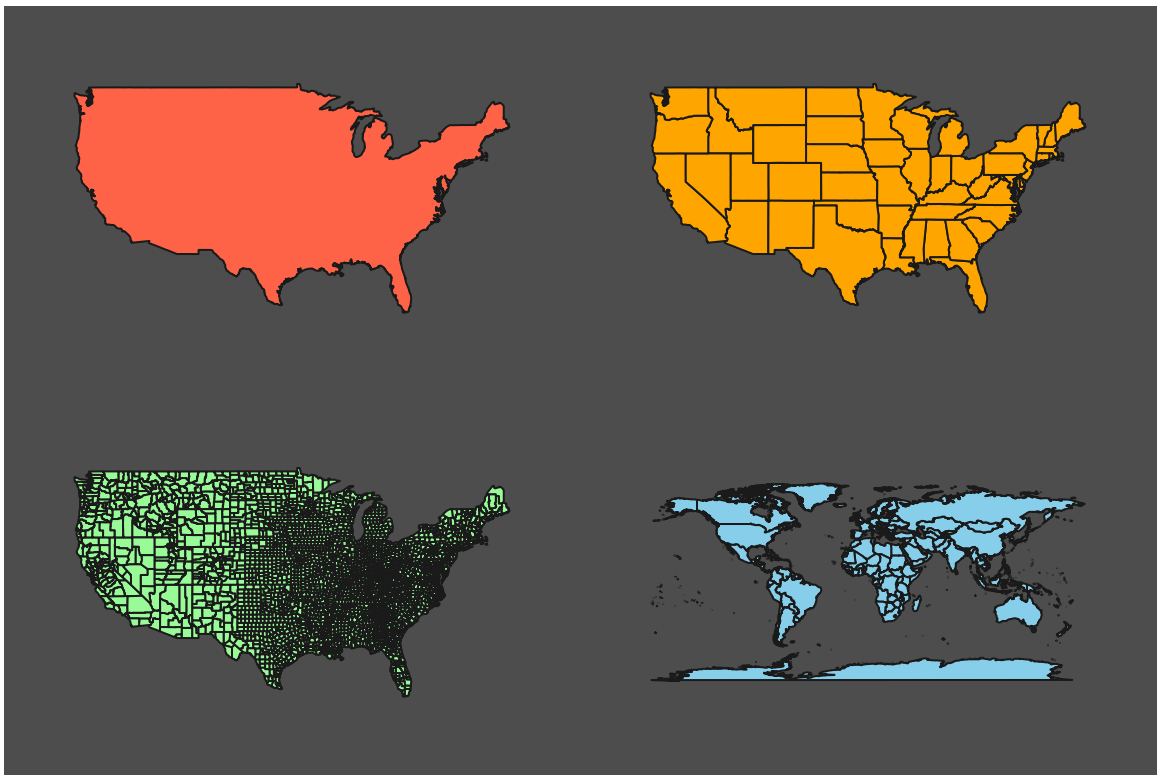
In order to plot on a map, we will need a few more packages. As you will see below, `maps` will let us generate a geographic map to use as background, and `geosphere` will help us generate arcs representing our network edges. If you do not already have them, install the two packages, then load them.

```
install.packages('maps')
install.packages('geosphere')
```

```
library('maps')
library('geosphere')
```

Let us plot some example maps with the `maps` library. The parameters of `maps()` include `col` for the map fill, `border` for the border color, and `bg` for the background color.

```
par(mfrow = c(2,2), mar=c(0,0,0,0))

map("usa", col="tomato",  border="gray10", fill=TRUE, bg="gray30")
map("state", col="orange",  border="gray10", fill=TRUE, bg="gray30")
map("county", col="palegreen",  border="gray10", fill=TRUE, bg="gray30")
map("world", col="skyblue",  border="gray10", fill=TRUE, bg="gray30")
```



```
dev.off()
```

The data we will use here contains US airports and flights among them. The airport file includes geographic coordinates - latitude and longitude. If you do not have those in your data, you can the `geocode()` function from package `ggmap` to grab the latitude and longitude for an address.

```
airports <- read.csv("Dataset3-Airlines-NODES.csv", header=TRUE)
flights <- read.csv("Dataset3-Airlines-EDGES.csv", header=TRUE, as.is=TRUE)

head(flights)
```

```
##   Source Target Freq
## 1      0    109   10
## 2      1     36   10
## 3      1     61   10
## 4      2    152   10
## 5      3    104   10
## 6      4    132   10
```

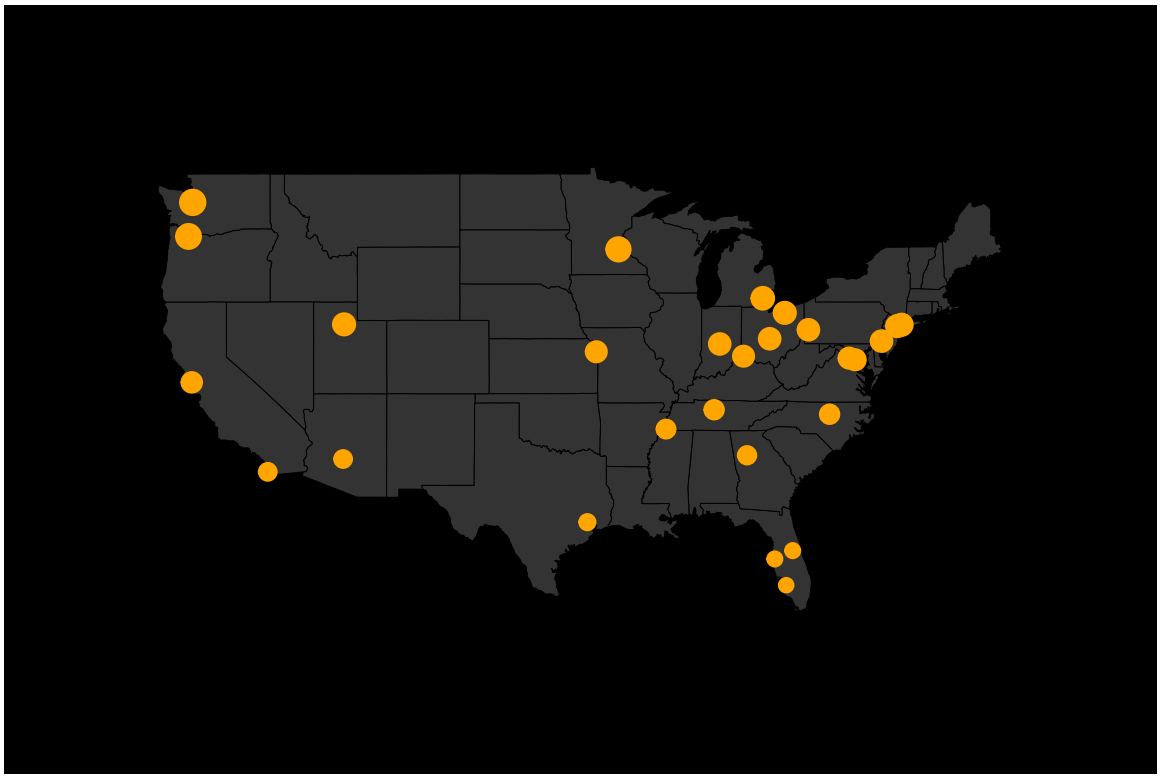
```
head(airports)
```

```
##   ID                     Label Code             City latitude  longitude
## 1  0       Adams Field Airport  LIT  Little Rock, AR 34.72944  -92.22444
## 2  1     Akron/canton Regional  CAK Akron/Canton, OH 40.91611  -81.44222
## 3  2      Albany International  ALB           Albany 42.73333  -73.80000
## 4  3                 Albemarle  CHO  Charlottesville 38.13333  -78.45000
## 5  4 Albuquerque International  ABQ      Albuquerque 35.04028 -106.60917
## 6  5  Alexandria International  AEX   Alexandria, LA 31.32750  -92.54861
##   ToFly Visits
## 1     0    105
## 2     0    123
## 3     0    129
## 4     1    114
## 5     0    105
## 6     0     93
```

```
# Select only large airports: ones with more than 10 connections in the data.
tab <- table(flights$Source)
big.id <- names(tab)[tab>10]
airports <- airports[airports$ID %in% big.id,]
flights  <- flights[flights$Source %in% big.id &
                    flights$Target %in% big.id, ]
```

In order to generate our plot, we will first add a map of the United states. Then we will add a point on the map for each airport:

```
# Plot a map of the united states:
map("state", col="grey20", fill=TRUE, bg="black", lwd=0.1)

# Add a point on the map for each airport:
points(x=airports$longitude, y=airports$latitude, pch=19,
       cex=airports$Visits/80, col="orange")
```



Next we will generate a color gradient to use for the edges in the network. Heavier edges will be lighter in color.

```
col.1 <- adjustcolor("orange red", alpha=0.4)
col.2 <- adjustcolor("orange", alpha=0.4)
edge.pal <- colorRampPalette(c(col.1, col.2), alpha = TRUE)
edge.col <- edge.pal(100)
```

For each flight in our data, we will use `gcIntermediate()` to generate the coordinates of the shortest arc that connects its start and end point (think distance on the surface of a sphere). After that, we will plot each arc over the map using `lines()`.

```
for(i in 1:nrow(flights))  {
    node1 <- airports[airports$ID == flights[i,]$Source,]
    node2 <- airports[airports$ID == flights[i,]$Target,]


    arc <- gcIntermediate( c(node1[1,]$longitude, node1[1,]$latitude),
                           c(node2[1,]$longitude, node2[1,]$latitude),
                           n=1000, addStartEnd=TRUE )
    edge.ind <- round(100*flights[i,]$Freq / max(flights$Freq))

    lines(arc, col=edge.col[edge.ind], lwd=edge.ind/30)
}
```

Note that if you are plotting the network on a full world map, there might be cases when the shortest arc goes "behind" the map – e.g. exits it on the left side and enters back on the right (since the left-most and right-most points on the map are actually next to each other). In order to avoid that, we can use `greatCircle()` to generate the full great circle (circle going through those two points and around the globe, with a center at the center of the earth). Then we can extract from it the arc connecting our start and end points which does not cross "behind" the map, regardless of whether it is the shorter or the longer of the two.

This is the end of our tutorial. If you have comments, questions, or want to report typos, please e-mail netvis@ognyanova.net. Check for updated versions of the tutorial at kateto.net.