Network visualization with R

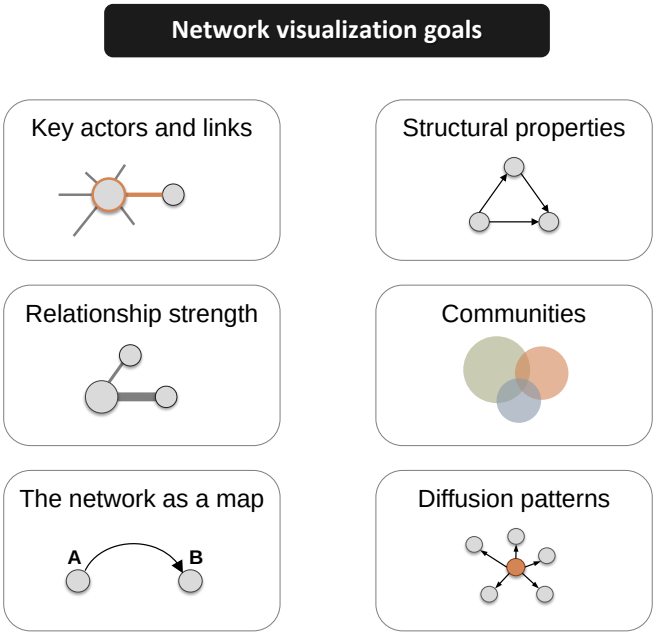
Katherine Ognyanova, www.kateto.net

POLNET 2015 Workshop, Portland OR

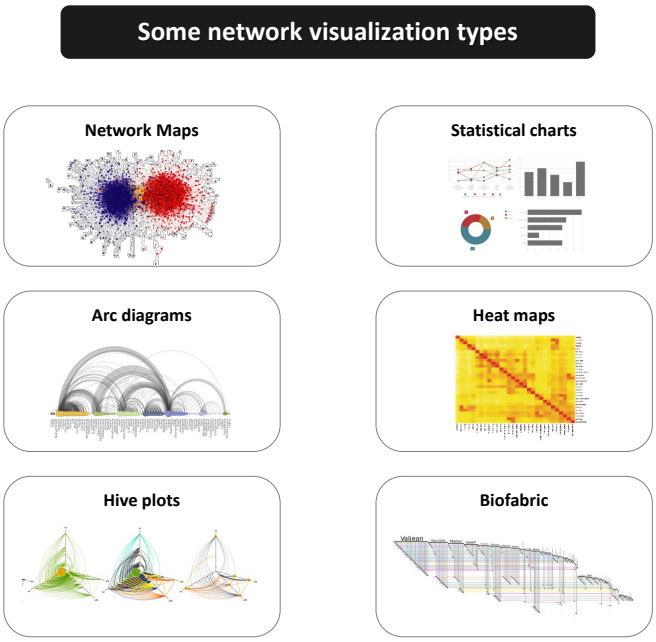
Contents

Introduction: Network Visualization

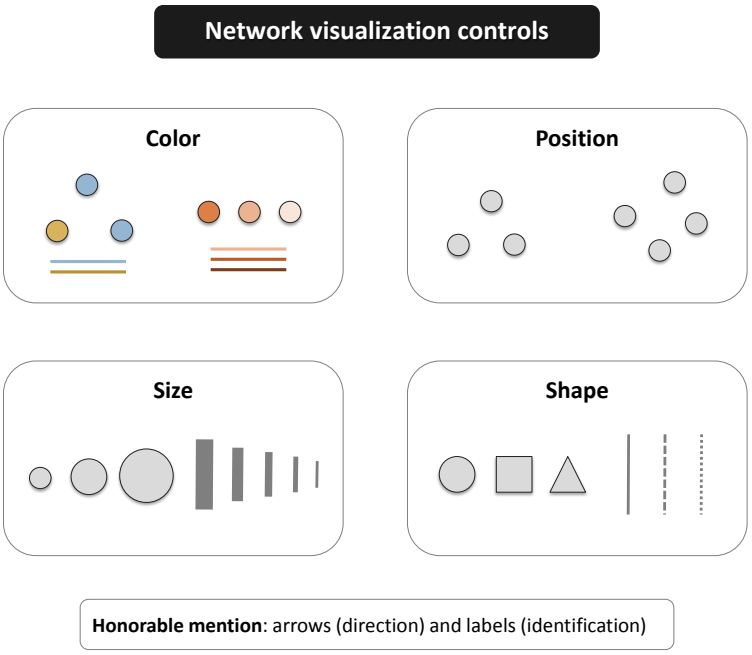
The main concern in designing a network visualization is the purpose it has to serve. What are the structural properties that we want to highlight?



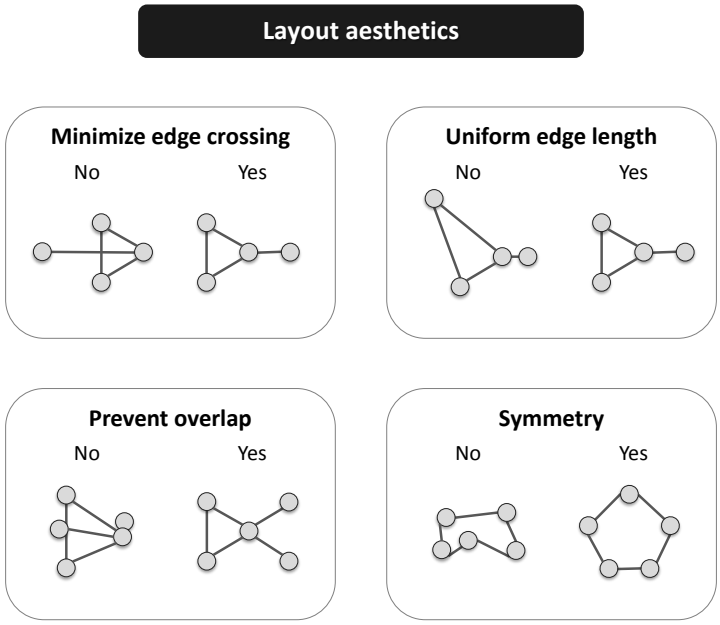
Network maps are far from the only visualization available for graphs - other network representation formats, and even simple charts of key characteristics, may be more appropriate in some cases.

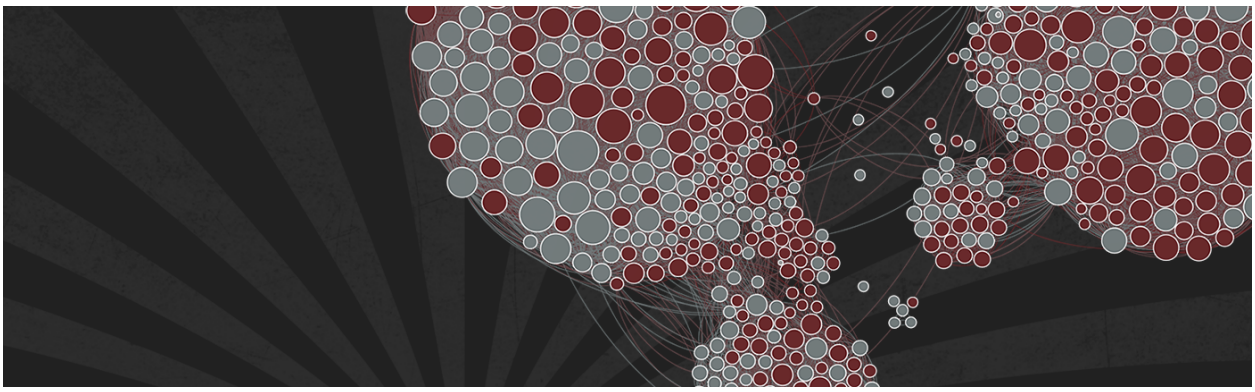


In network maps, as in other visualization formats, we have several key elements that control the outcome. The major ones are color, size, shape, and position.



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/polnet2015.

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.

This tutorial uses several key packages that you will need to install in order to follow along. Several other libraries will be mentioned along the way, but those are not critical and can be skipped. The main libraries we are going to use are igraph (maintained by Gabor Csardi and Tamas Nepusz), sna & network (maintained by Carter Butts and the Statnet team), and ndtv (maintained by Skye Bender-deMoll).

```
install.packages("igraph")
install.packages("network")
install.packages("sna")
install.packages("ndtv")
```

DATASET 1: edgelist

The first data set we are going to work with consists of two files, "Media-Example-NODES.csv" and "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)
nrow(nodes); length(unique(nodes$id))
nrow(links); nrow(unique(links[,c("from", "to")]))
```

Notice that there are more links than unique from-to combinations. That means we have cases in the data where there are multiple links between the same two nodes. We will collapse all links of the same type between the same two nodes by summing their weights, using `aggregate()` by "from", "to", & "type":

```
links <- aggregate(links[,3], links[,-3], sum)
links <- links[order(links$from, links$to),]
colnames(links)[4] <- "weight"
rownames(links) <- NULL
```

DATASET 2: matrix

```
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)
```

We can see that links2 is an adjacency matrix for a two-mode network:

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

Network visualization: first steps with *igraph*

We start by converting the raw data to an igraph network object. Here we use igraph's `graph.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.data.frame(links, nodes, directed=T)
net
```

```
## IGRAPH 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)
```

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:

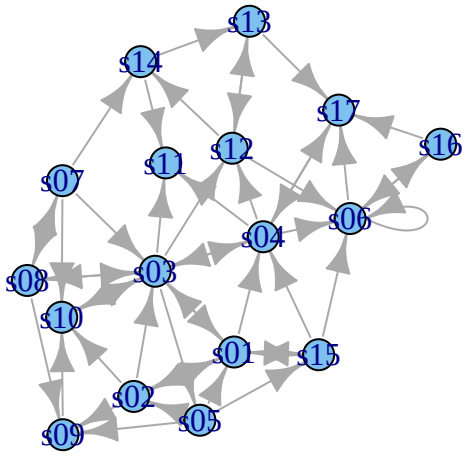
- `(g/c)` - graph-level character attribute
- `(v/c)` - vertex-level character attribute
- `(e/n)` - edge-level numeric attribute

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"

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

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

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



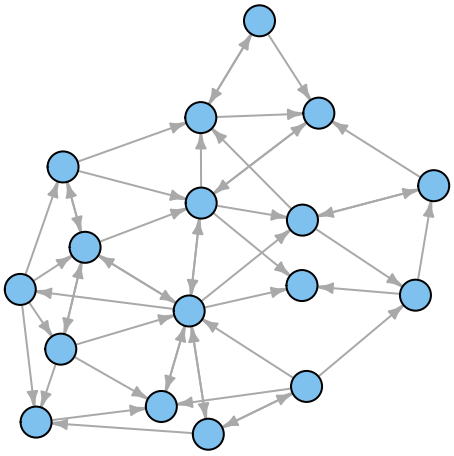
That doesn't look very good. Let's start fixing things by removing the loops in the graph.

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

You might notice that we could have used `simplify` to combine multiple edges by summing their weights with a command like `simplify(net, edge.attr.comb = list( Weight = "sum", "ignore" ))`. The problem is 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`):

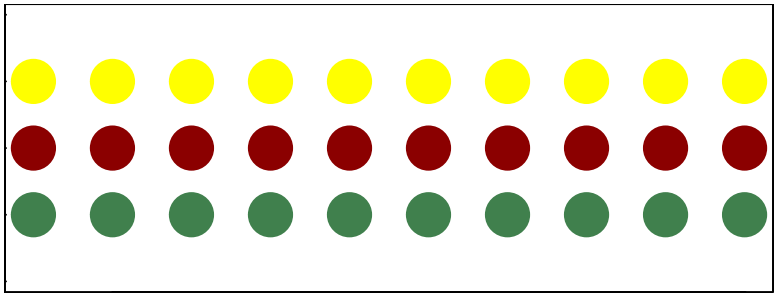
```
plot(net, edge.arrow.size=.4,vertex.label=NA)
```



A brief detour I: 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`

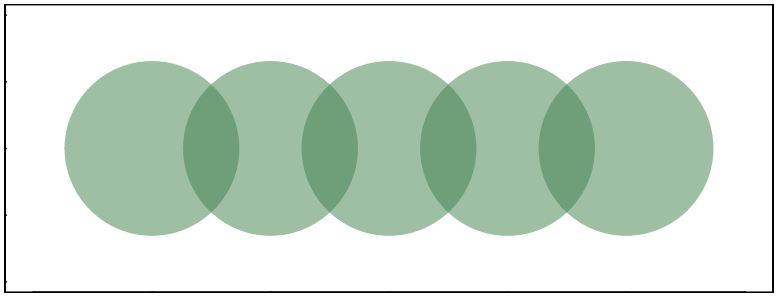
```
plot(x=1:10, y=rep(5,10), pch=19, cex=3, col="dark red")
points(x=1:10, y=rep(6, 10), pch=19, cex=3, col="557799")
points(x=1:10, y=rep(4, 10), pch=19, cex=3, col=rgb(.25, .5, .3))
```



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

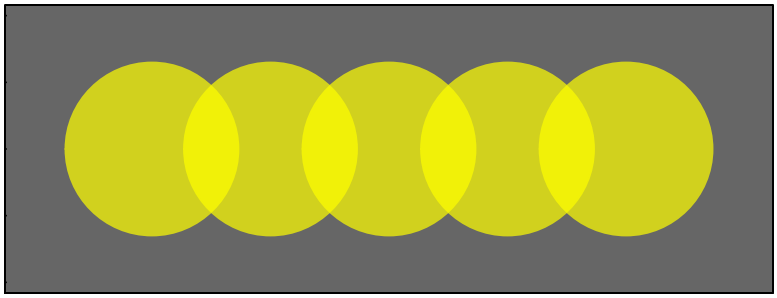
We can set the opacity/transparency of an element using the parameter `alpha` (range 0-1):

```
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.

```
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))
```

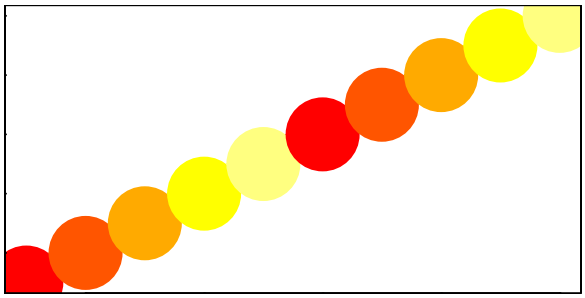


If you plan on using the built-in color names, here's how to list all of them:

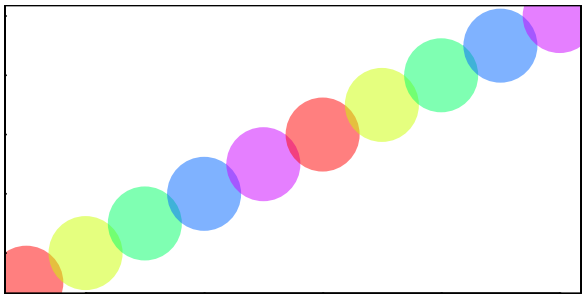
```
colors()                            # List all named colors
grep("blue", colors(), value=T)     # Colors that have "blue" in the name
```

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)
```

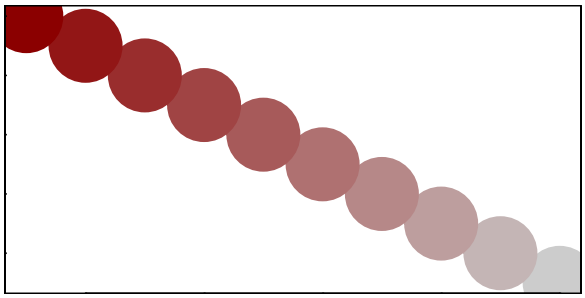


```
plot(x=1:10, y=1:10, pch=19, cex=5, col=pal2)
```



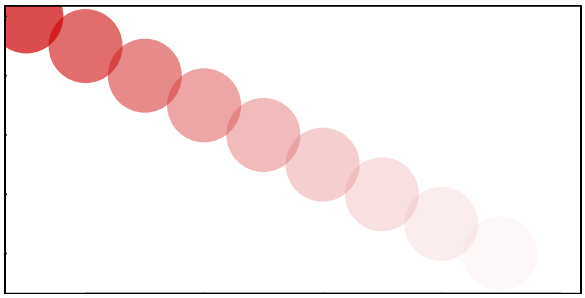
We can also generate our own gradients using `colorRampPalette`. Note that `colorRampPalette` returns a *function* that we can use to generate as many colors from that palette as we need.

```
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:

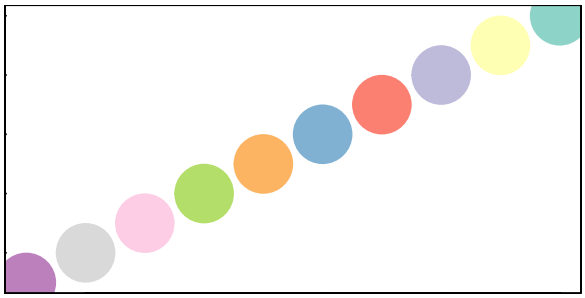
```
display.brewer.pal(8, "Set3")
```

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

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

Using `RColorBrewer` palettes in plots:

```r
pal3 <- brewer.pal(10, "Set3")
plot(x=10:1, y=10:1, pch=19, cex=4, col=pal3)
```



A brief detour II: Fonts in R plots

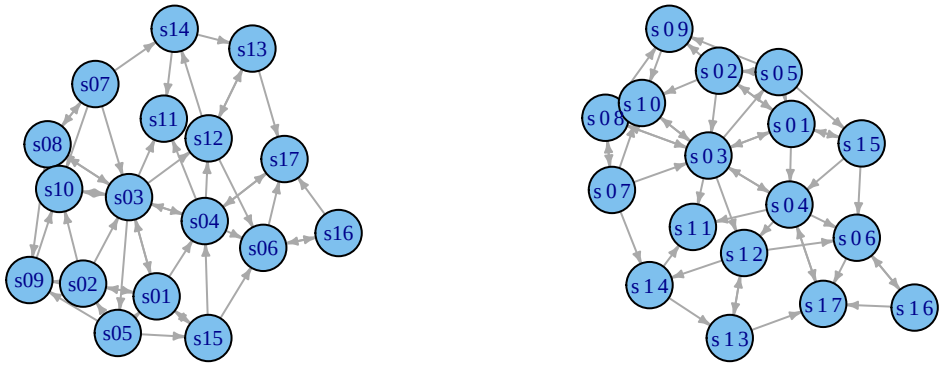
Using different fonts for R plots may take a little bit of work. This is especially true if you are using Windows - Mac & Linux users can most likely skip all of this.

In order to import fonts from the OS into R, we can use the 'extrafont' package:

```r
install.packages("extrafont")
library(extrafont)

# Import system fonts - may take a while, so DO NOT run this during the workshop.
font_import()
fonts() # See what font families are available to you now.
loadfonts(device = "win") # use device = "pdf" for pdf plot output.
```

Now that your fonts are available, you should be able to do something like this:

```
library(extrafont)
plot(net, vertex.size=30)
plot(net, vertex.size=30, vertex.label.family="Arial Black" )
```



When you save plots as PDF files, you can also embed the fonts:

```
# First you may have to let R know where to find ghostscript on your machine:
Sys.setenv(R_GSCMD = "C:/Program Files/gs/gs9.10/bin/gswin64c.exe")

# pdf() will send all the plots we output before dev.off() to a pdf file:
pdf(file="ArialBlack.pdf")
plot(net, vertex.size=30, vertex.label.family="Arial Black" )
dev.off()

embed_fonts("ArialBlack.pdf", outfile="ArialBlack_embed.pdf")
```

Back to our main plot line: plotting networks

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.

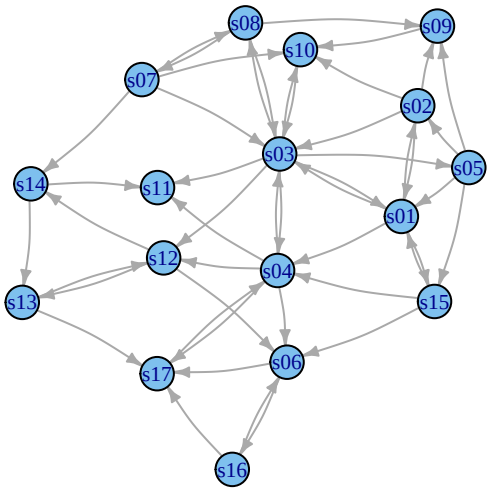
The igraph plotting parameters include (among others):

Plotting parameters

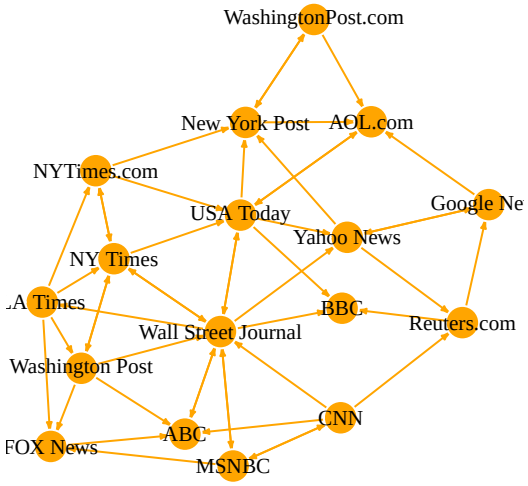
NODES	
vertex.color	Node color
vertex.frame.color	Node border color
vertex.shape	One of “none”, “circle”, “square”, “csquare”, “rectangle” “crectangle”, “vrectangle”, “pie”, “raster”, or “sphere”
vertex.size	Size of the node (default is 15)
vertex.size2	The second size of the node (e.g. for a rectangle)
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 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

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:
plot(net, edge.arrow.size=.4, edge.curved=.1)
```



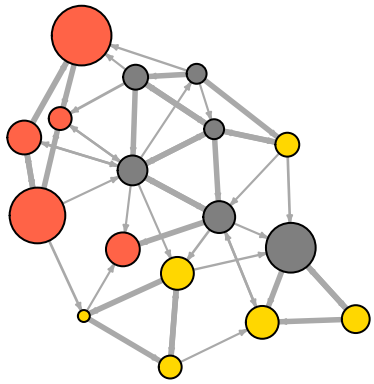

```
# Set edge color to light gray, the node & border color to orange
# Replace the vertex label with the node names stored in "media"
plot(net, edge.arrow.size=.2, edge.color="orange",
     vertex.color="orange", vertex.frame.color="#ffffff",
     vertex.label=V(net)$media, vertex.label.color="black")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



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.

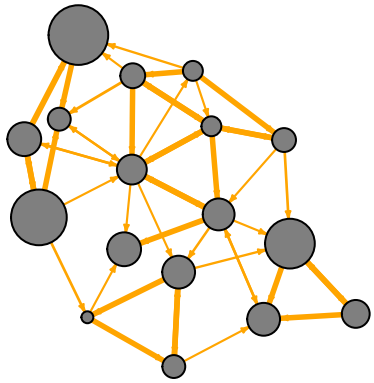
```
# Generate colors base on media type:
colrs <- c("gray50", "tomato", "gold")
V(net)$color <- colrs[V(net)$media.type]

# 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"
E(net)$width <- 1+E(net)$weight/12
plot(net)
```



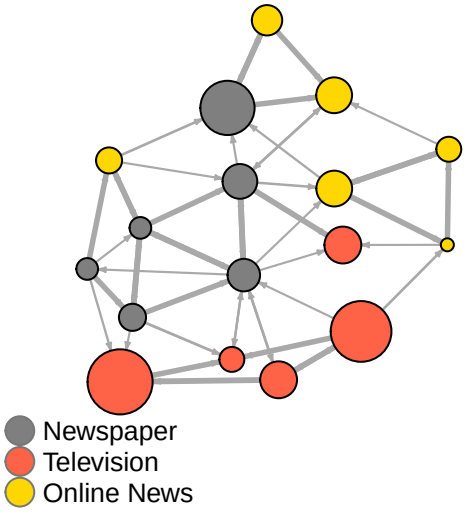
We can also override the attributes explicitly in the plot:

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



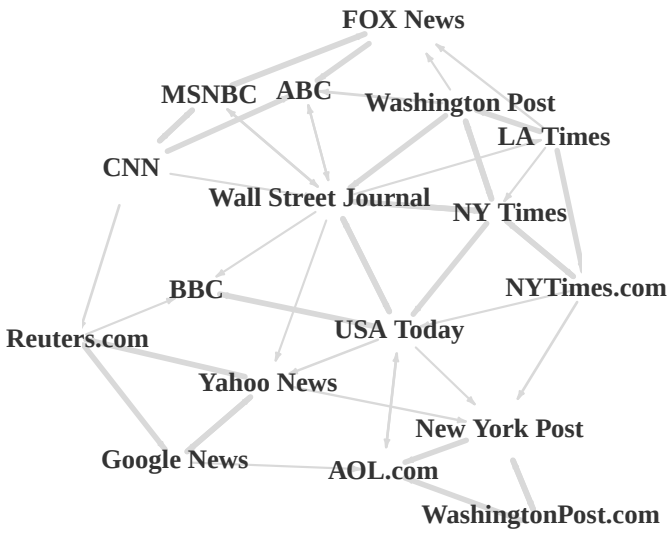
It helps to add a legend explaining the meaning of the colors we used:

```
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)
```



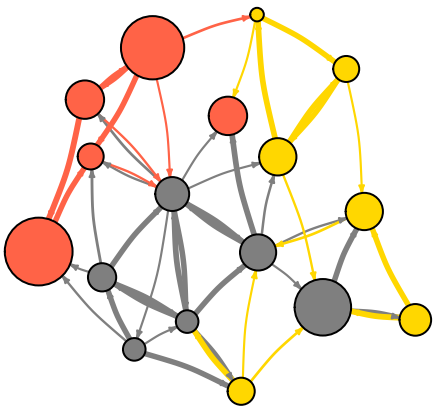
Sometimes, especially with semantic networks, we may be interested in plotting only the labels of the nodes:

```
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 `get.edges` igraph function.

```
edge.start <- get.edges(net, 1:ecount(net))[,1]
edge.col <- V(net)$color[edge.start]

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

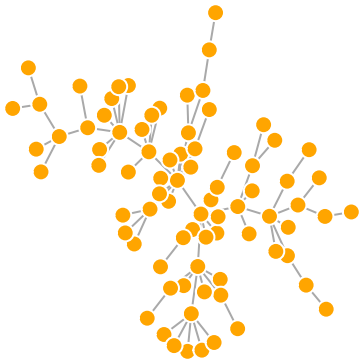


Network Layouts

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

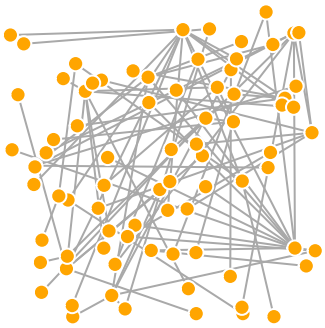
For the purposes of exploring layouts, we will generate a slightly larger 80-node graph. We use the barabasi.game function which generates a simple graph starting from one node and adding more nodes and links based on a preset level of preferential attachment (how much new actors would prefer to form links to the more popular nodes in the network).

```
net.bg <- barabasi.game(80)
V(net.bg)$frame.color <- "white"
V(net.bg)$color <- "orange"
V(net.bg)$label <- ""
V(net.bg)$size <- 10
E(net.bg)$arrow.mode <- 0
plot(net.bg)
```



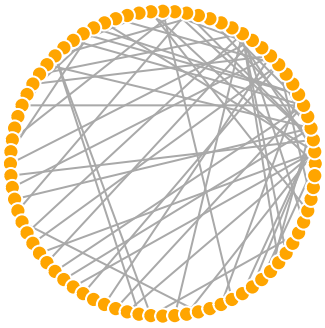
You can set the layout in the plot function:

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



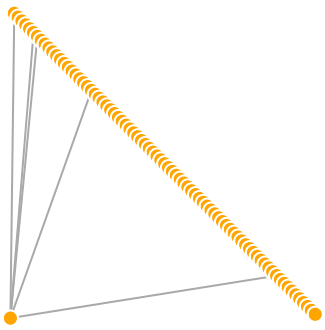
Or you can calculate the vertex coordinates in advance:

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



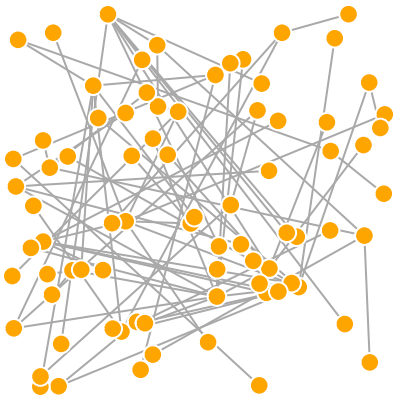
`l` is simply a matrix of x, y coordinates (N x 2) for the N nodes in the graph. You can easily generate your own:

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

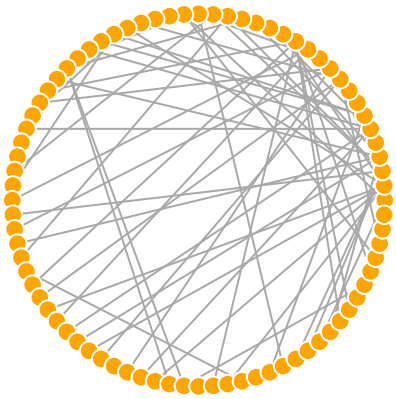


This layout is just an example and not very helpful - thankfully igraph has a number of built-in layouts, including:

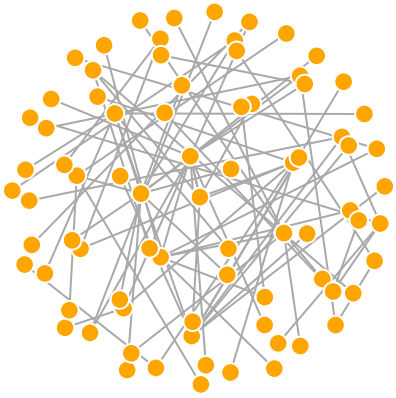
```
# Randomly placed vertices
l <- layout.random(net.bg)
plot(net.bg, layout=l)
```



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



```
# 3D sphere layout
l <- layout.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.

Some parameters you can set for this layout include `area` (the default is the square of # nodes) and `repulserad` (cancellation radius for the repulsion - the area multiplied by # nodes). Both parameters affect the spacing of the plot - play with them until you like the results.

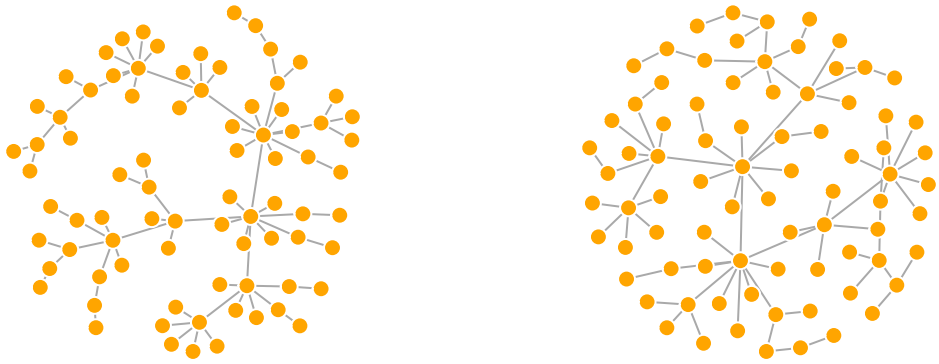
You can also set the "weight" parameter which increases the attraction forces among nodes connected by heavier edges.

You will notice that the 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.

The layout `fruchterman.reingold.grid` is similar to `fruchterman.reingold`, but faster.

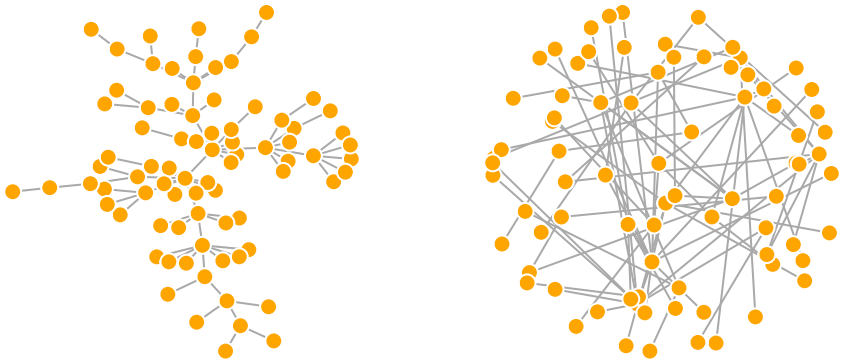
```
l <- layout.fruchterman.reingold(net.bg, repulserad=vcount(net.bg)^3,
                                 area=vcount(net.bg)^2.4)
par(mfrow=c(1,2),  mar=c(0,0,0,0)) # plot two figures - 1 row, 2 columns
plot(net.bg, layout=layout.fruchterman.reingold)
plot(net.bg, layout=l)
```



```
dev.off() # shut off the  graphic device to clear the two-figure configuration.
```

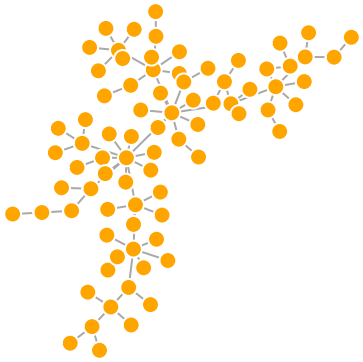
Another popular force-directed algorithm that produces nice results for connected graphs is Kamada Kawai. Like Fruchterman Reingold, it attempts to minimize the energy in a spring system. Igraph also has a spring-embedded layout called `layout.spring()`.

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

l <- layout.spring(net.bg, mass=.5)
plot(net.bg, layout=l)
```



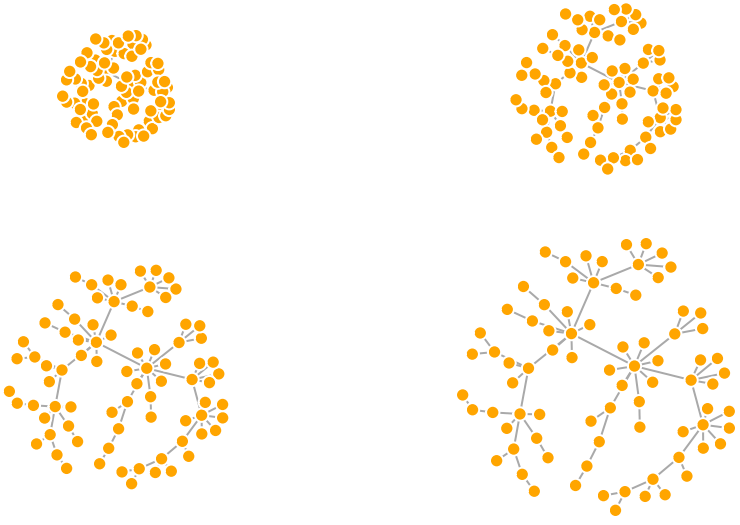
The LGL algorithm is meant for large, connected graphs. Here you can also specify a root: a node that will be placed in the middle of the layout.

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



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 layout.norm to normalize the plot with the boundaries you want.

```
l <- layout.fruchterman.reingold(net.bg)
l <- layout.norm(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)
```

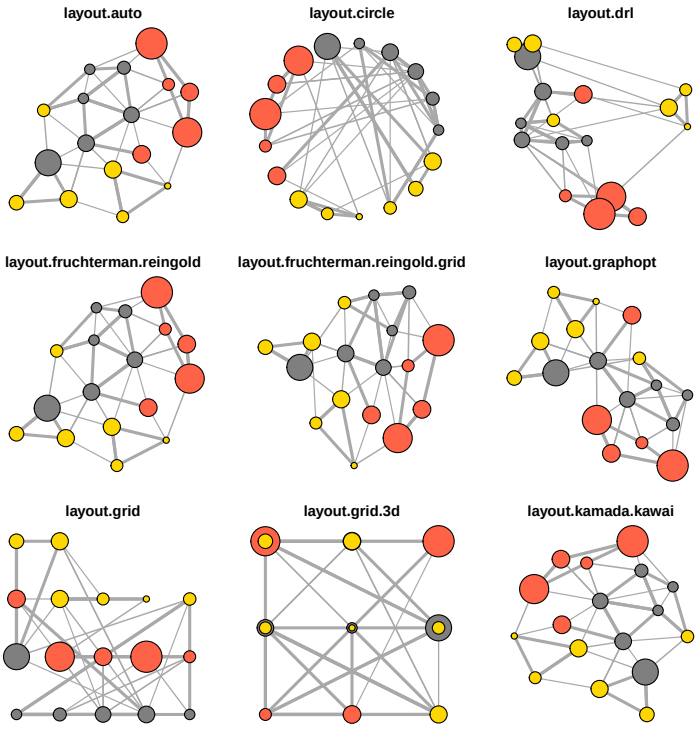


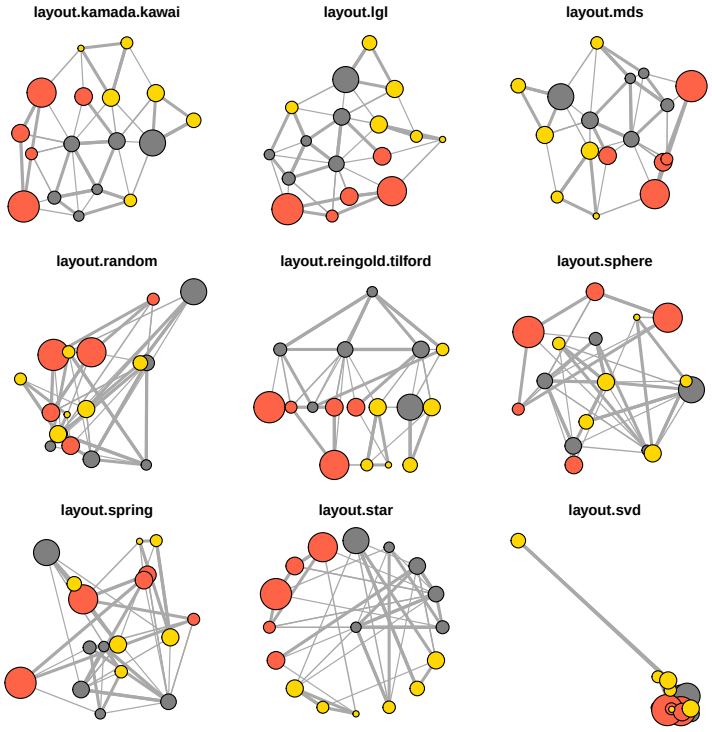
```
dev.off()
```

The layout igraph uses by default is called `layout.auto`. It automatically selects an appropriate layout algorithm based on the properties (size and connectedness) of the graph.

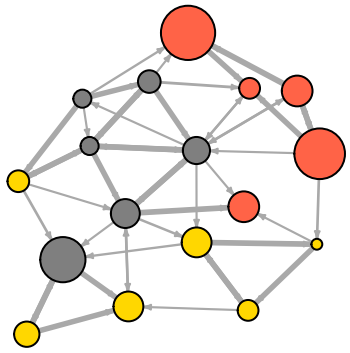
Let's take a look at all available layouts in igraph:

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

par(mfrow=c(3,3))

for (layout in layouts) {
  print(layout)
  l <- do.call(layout, list(net))
  plot(net, edge.arrow.mode=0, layout=l, main=layout) }

dev.off()
```





Highlighting aspects of the network

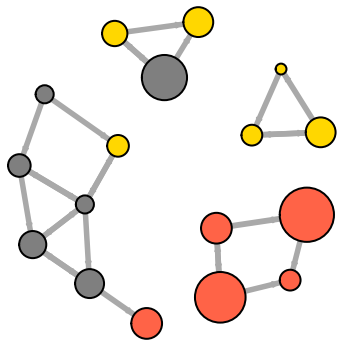


Notice that our network plot is still not too helpful. We can identify the type and size of nodes, but cannot see much about the structure since the links we're examining are so dense. One way to approach this is to see if we can sparsify the network, keeping only the most important ties and discarding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

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

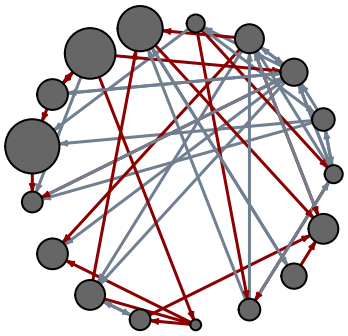
There are more sophisticated ways to extract the key edges, but for the purposes of this exercise we'll only keep ones that have weight higher than the mean for the network. In igraph, we can delete edges using `delete.edges(net, edges)`:

```
cut.off <- mean(links$weight)
net.sp <- delete.edges(net, E(net)[weight<cut.off])
l <- layout.fruchterman.reingold(net.sp, repulserad=vcount(net)^2.1)
plot(net.sp, layout=l)
```

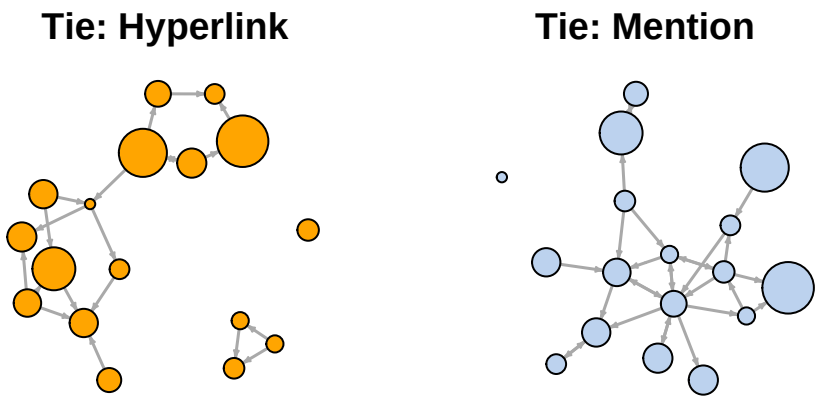


Another way to think about this is to plot the two tie types (hyperlink & mention) separately.

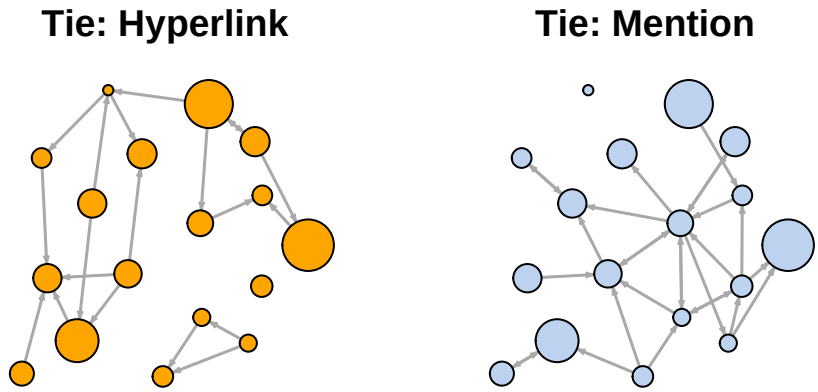
```
E(net)$width <- 1.5
plot(net, edge.color=c("dark red", "slategrey")[(E(net)$type=="hyperlink")+1],
      vertex.color="gray40", layout=layout.circle)
```



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

par(mfrow=c(1,2))
plot(net.h, vertex.color="orange", main="Tie: Hyperlink")
plot(net.m, vertex.color="lightsteelblue2", main="Tie: Mention")
```



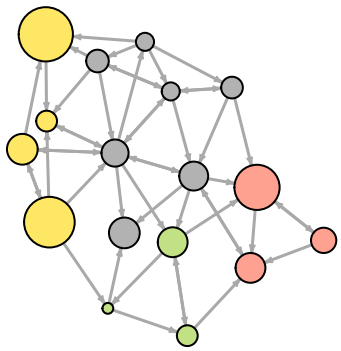

```
l <- layout.fruchterman.reingold(net)
plot(net.h, vertex.color="orange", layout=l, main="Tie: Hyperlink")
plot(net.m, vertex.color="lightsteelblue2", layout=l, main="Tie: Mention")
```




```
dev.off()
```

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

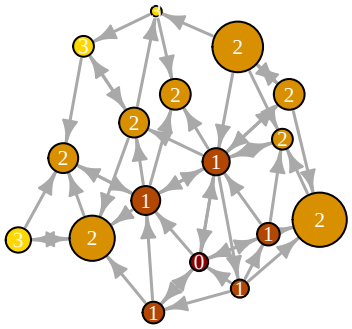
```
V(net)$community <- optimal.community(net)$membership
colrs <- adjustcolor( c("gray50", "tomato", "gold", "yellowgreen"), alpha=.6)
plot(net, vertex.color=colrs[V(net)$community])
```



Highlighting specific nodes or links

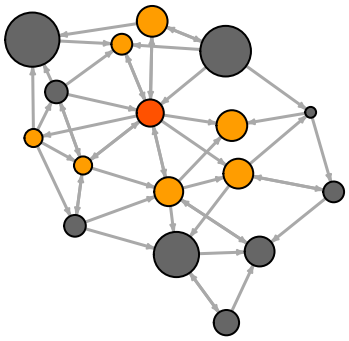
Sometimes we want to focus the visualization on a particular node or a group of nodes. In our example media network, we can examine the spread of information from focal actors. For instance, let's represent distance from the NYT. The `shortest.paths` function (as its name indicates) returns a matrix of shortest paths between nodes in the network.

```
dist.from.NYT <- shortest.paths(net, algorithm="unweighted")[1,]
oranges <- colorRampPalette(c("dark red", "gold"))
col <- oranges(max(dist.from.NYT)+1)[dist.from.NYT+1]

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



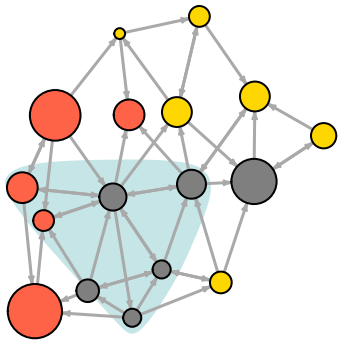
Or we can show all the immediate neighbors of the WSJ. Note that the `neighbors` function finds all nodes one step out from the focal actor. The corresponding function that finds all edges for a node is `incident`.

```
col <- rep("grey40", vcount(net))
col[V(net)$media=="Wall Street Journal"] <- "#ff5100"

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

col[neigh.nodes] <- "#ff9d00"
plot(net, vertex.color=col)
```

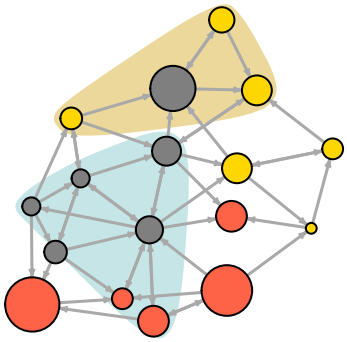


Another way to draw attention to a group of nodes is to "mark" them:

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



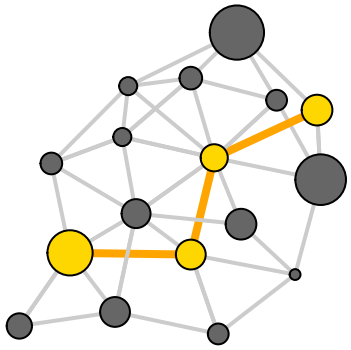
```
# Mark multiple groups:
plot(net, mark.groups=list(c(1,4,5,8), c(15:17)),
          mark.col=c("#C5E5E7","#ECD89A"), mark.border=NA)
```



We can also highlight a path in the network:

```
news.path <- get.shortest.paths(net, V(net)[media=="MSNBC"],
                                V(net)[media=="New York Post"],
                                mode="all", output="both")


# Generate edge color variable:
ecol <- rep("gray80", ecount(net))
ecol[unlist(news.path$epath)] <- "orange"

# Generate edge width variable:
ew <- rep(2, ecount(net))
ew[unlist(news.path$epath)] <- 4
```

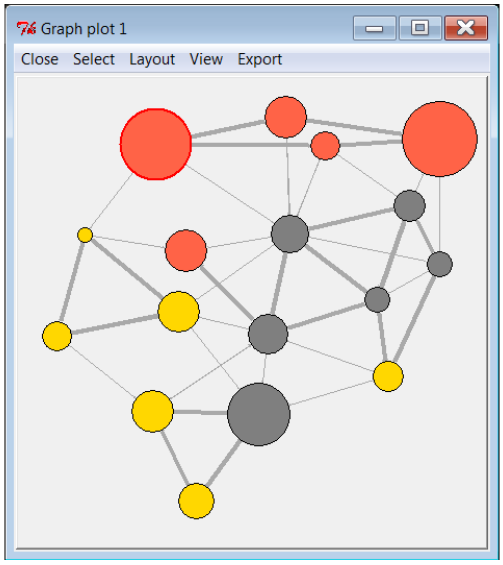
```
# Generate node color variable:
vcol <- rep("gray40", vcount(net))
vcol[unlist(news.path$vpath)] <- "gold"

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



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)
```

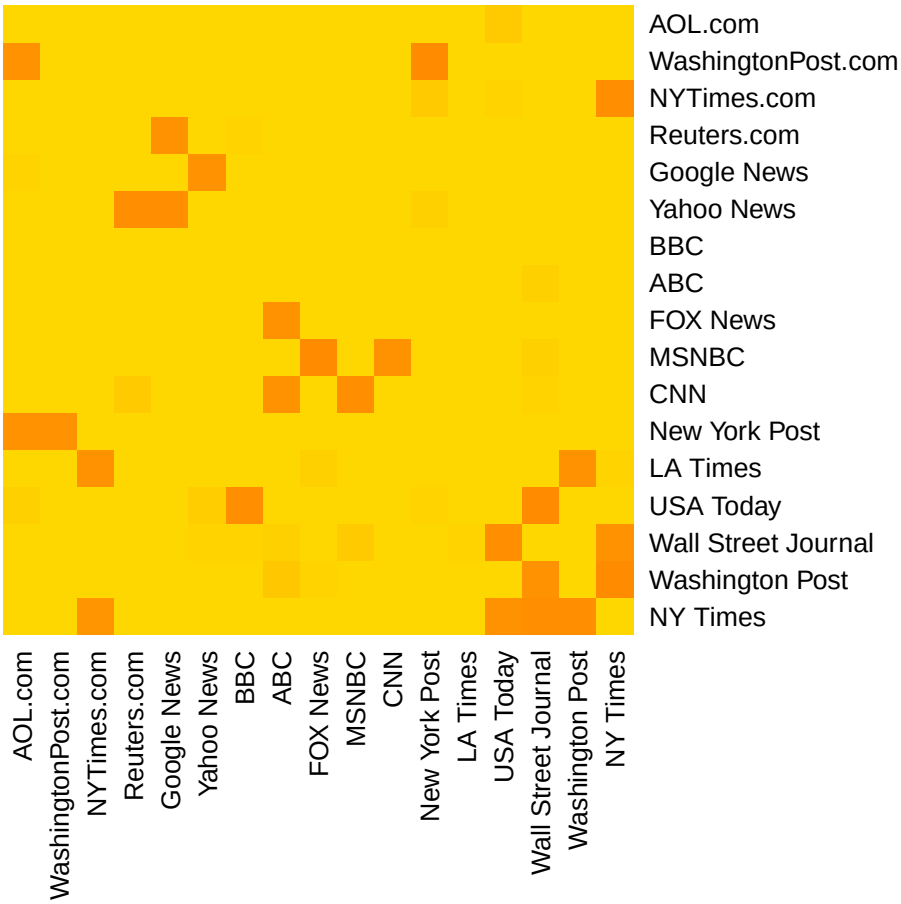


Other ways to represent a network

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.

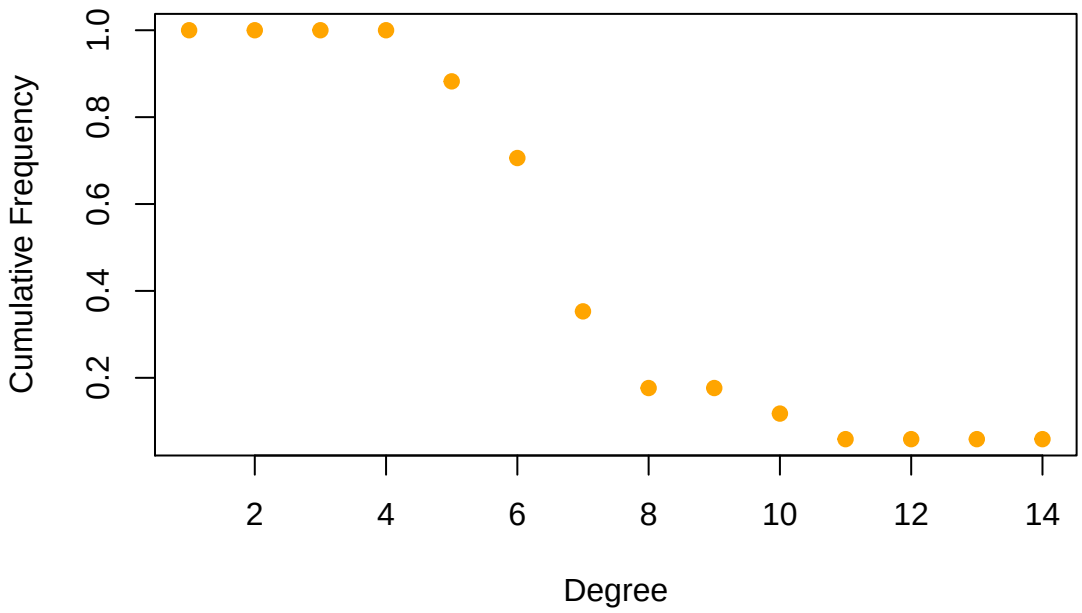
For example, here is a quick heatmap of the network matrix:

```
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.

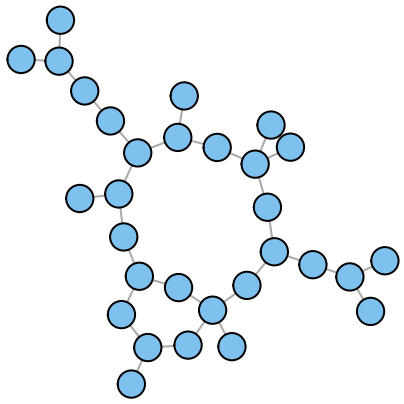
```
dd <- degree.distribution(net, cumulative=T, mode="all")
plot(dd, pch=19, cex=1, col="orange", xlab="Degree", ylab="Cumulative Frequency")
```



Plotting two-mode networks with igraph

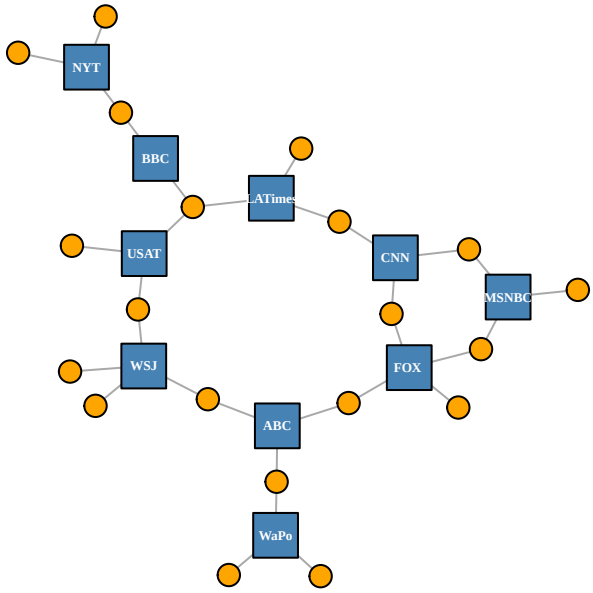
Two-mode or bipartite graphs have two different types of actors and links that go across, but not within each type. Our second media example is a network of that kind, examining links between news sources and their consumers. As you will see below, this time the edges of the network are in a matrix format. We can read those into a graph object using `graph.incidence`. In igraph, bipartite networks have an edge attribute called `type` that is 0 for one group of nodes and 1 for the other.

```
head(nodes2)
head(links2)

net2 <- graph.incidence(links2)
table(E(net2)$type)

plot(net2, vertex.label=NA)
```



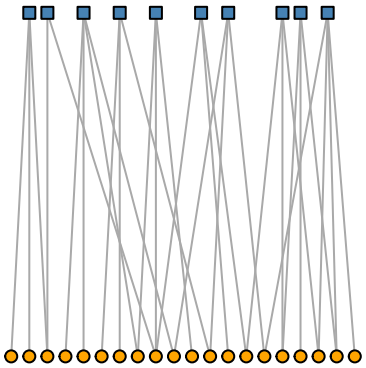
As with one-mode networks, we can modify the network object to include the visual properties that will be used by default when plotting the network. Notice that this time we will also change the shape of the nodes - media outlets will be squares, and their users will be circles.

```
V(net2)$color <- c("steel blue", "orange")[V(net2)$type+1]
V(net2)$shape <- c("square", "circle")[V(net2)$type+1]
V(net2)$label <- ""
V(net2)$label[V(net2)$type==F] <- nodes2$media[V(net2)$type==F]
V(net2)$label.cex=.4
V(net2)$label.font=2

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



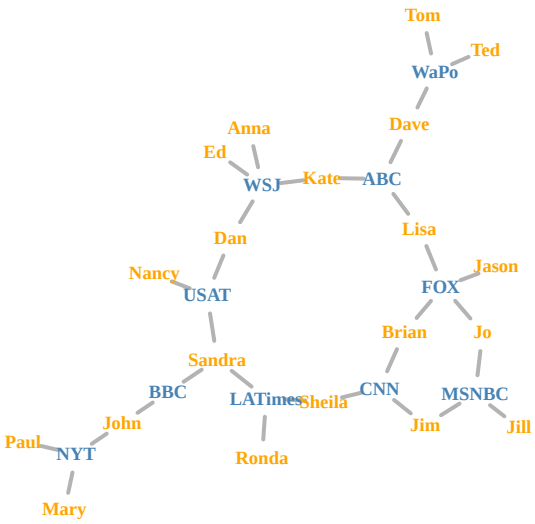
Igraph also has a special layout for bipartite networks (though it doesn't always work great, and you might be better off generating your own two-mode layout).

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



Using text as nodes may be helpful at times:

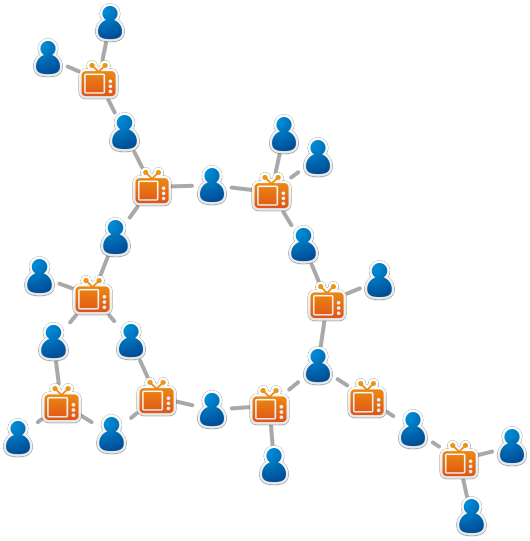
```
plot(net2, vertex.shape="none", vertex.label=nodes2$media,
     vertex.label.color=V(net2)$color, vertex.label.font=2,
     vertex.label.cex=.6, edge.color="gray70",  edge.width=2)
```



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

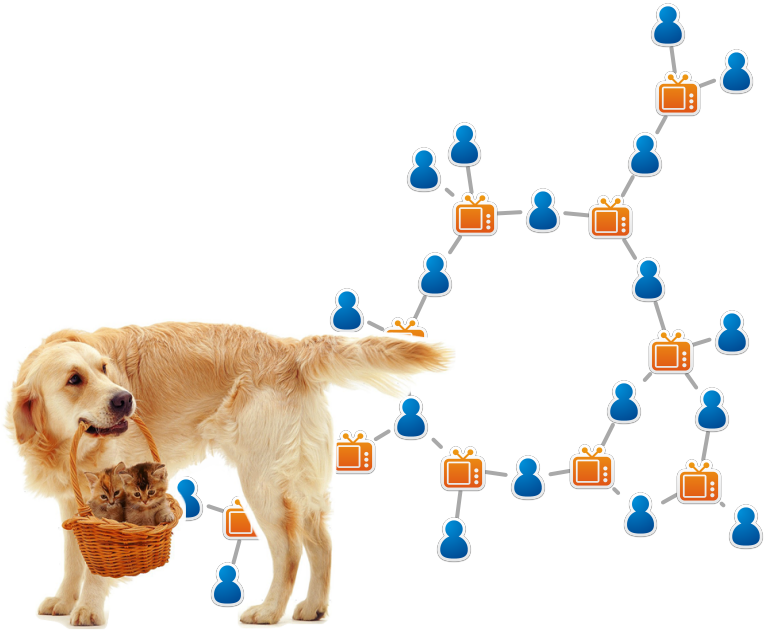
```
# install.packages("png")
library(png)

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 carrying a basket full of kittens.

```
l <- layout.auto(net2, ymin=-1.5, ymax=1.5, xmin=-1.5, xmax=1.5)

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

img.3 <- readPNG("./images/puppy.png")
rasterImage(img.3,  xleft=-1.7, xright=0, ybottom=-1.2, ytop=0)
```



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

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:png)
detach(package:igraph)
```

Quick example using the *network* package

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 as in igraph above, 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.

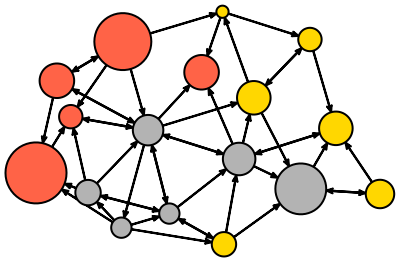
```
library(network)

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
```

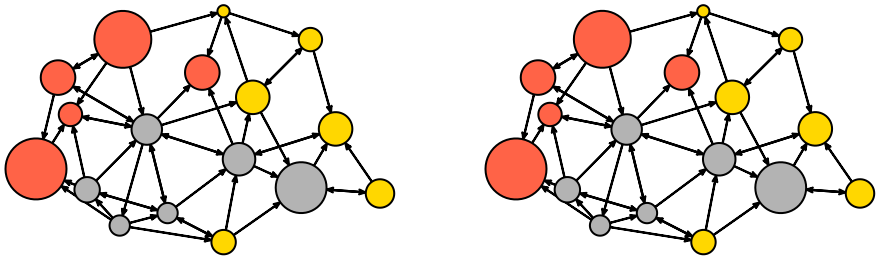
Let's plot our media network once again:

```
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)
```

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

Interactive and animated network visualizations

Interactive D3 Networks in R

These days it is increasingly easier to export R plots to html/javascript output. There are a number of libraries like `rcharts` and `htmlwidgets` that can help you create interactive web charts right from R. We'll take a quick look at networkD3 which - as its name suggests - generates interactive network visualizations using the D3 javascript library.

One thing to keep in mind is that the visualizations generated by `networkD3` are most useful 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.

If you don't have the networkD3 library, install it now:

```
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.

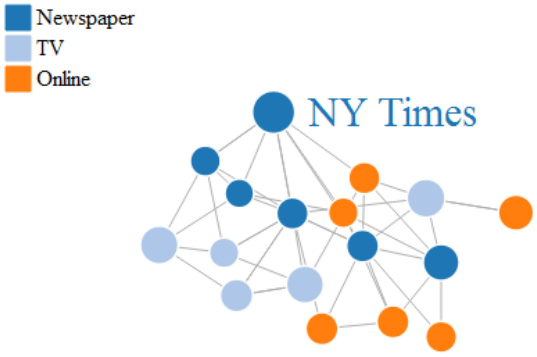
```
library(networkD3)

el <- 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:

```
nl <- 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 = el, Nodes = nl, Source="from", Target="to",
               NodeID = "idn", Group = "type.label",linkWidth = 1,
               linkColour = "#afafaf", fontSize=12, zoom=T, legend=T,
               Nodesize=6, opacity = 0.8, charge=-300,
               width = 600, height = 600)
```



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.

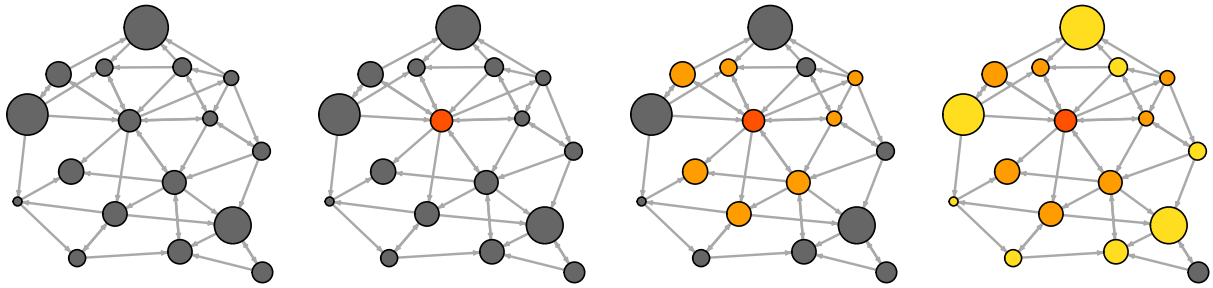
The catch here is that in order for this to work, you need not only the R package, but also an additional software called ImageMagick (imagemagick.org). You probably don't want to install that during the workshop, but you can try it at home.

```
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.fruchterman.reingold(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
```



Interactive networks with ndtv-d3

Interactive plots of static networks

Here we will create somewhat more sophisticated D3 visualizations using the `ndtv` package. You should not need additional software to produce web animations with D3. If you want to save the animations as video files (check out `?saveVideo`), you would 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)
```

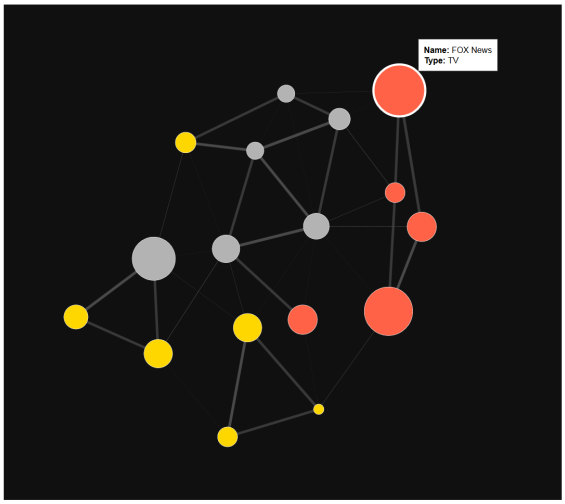
As this package is part of the Statnet family, it will accept objects from the `network` package such as the one we created earlier.

```
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 image in a markdown document, use `output.mode='inline'` above.



Network evolution animations

Animated visualizations are a good way to show the evolution of a small or medium size network over time. At present, `ndtv` is the best R package for that – especially since recently it added D3 visualizations to its capabilities.

In order to work with the network animations in ndtv, we need to understand Statnet's dynamic network format, implemented in the networkDynamic package. Let's look at one of the example data sets included in the package:

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

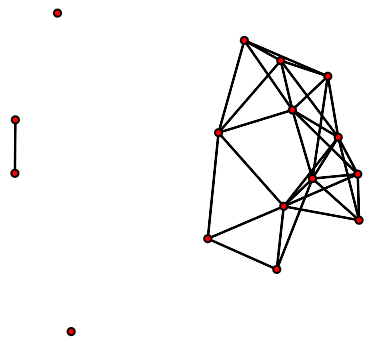
```
##   onset terminus tail head onset.censored
## 1     0        1    3    5          FALSE
## 2    10       20    3    5          FALSE
## 3     0       25    3    6          FALSE
```

```
## 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 an edge list. Each edge goes from the node with ID in the `tail` column to node with ID in the `head` column. The edge exists from time point `onset` to time point `terminus`. 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.

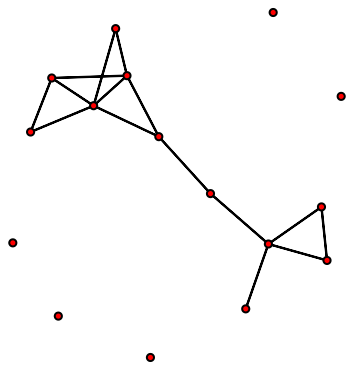
We can simply plot the network disregarding its time component (combining all nodes & edges that were ever present):

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



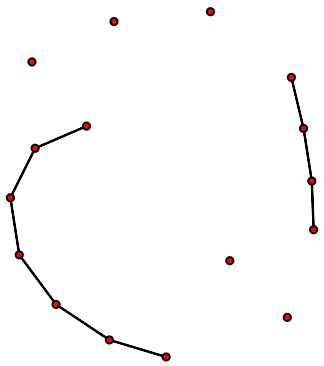
Plot the network at time 1:

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



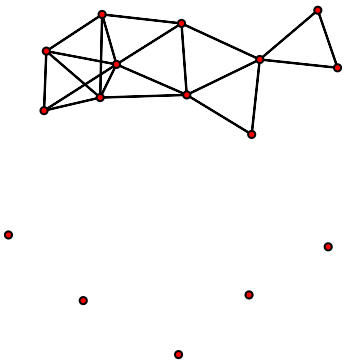
Plot nodes & vertices that were active from time 1 to time 5:

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



Plot all nodes and vertices that were active between time 1 & 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)
```

We are now ready to create and animate our own dynamic network. Dynamic network object 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`.

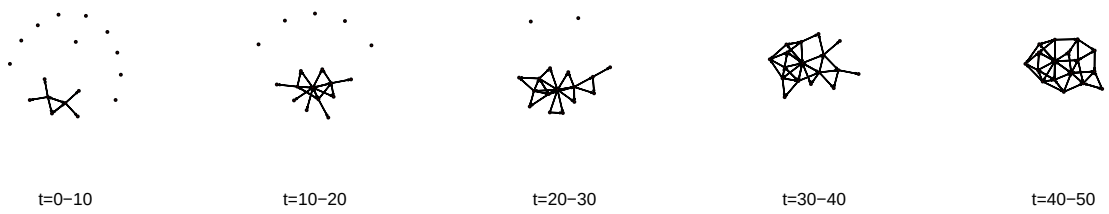
```
vs <- data.frame(onset=0, terminus=50, vertex.id=1:17)
es <- data.frame(onset=1:49, terminus=50,
                 head=as.matrix(net3, matrix.type="edgelist")[,1],
                 tail=as.matrix(net3, matrix.type="edgelist")[,2])

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.

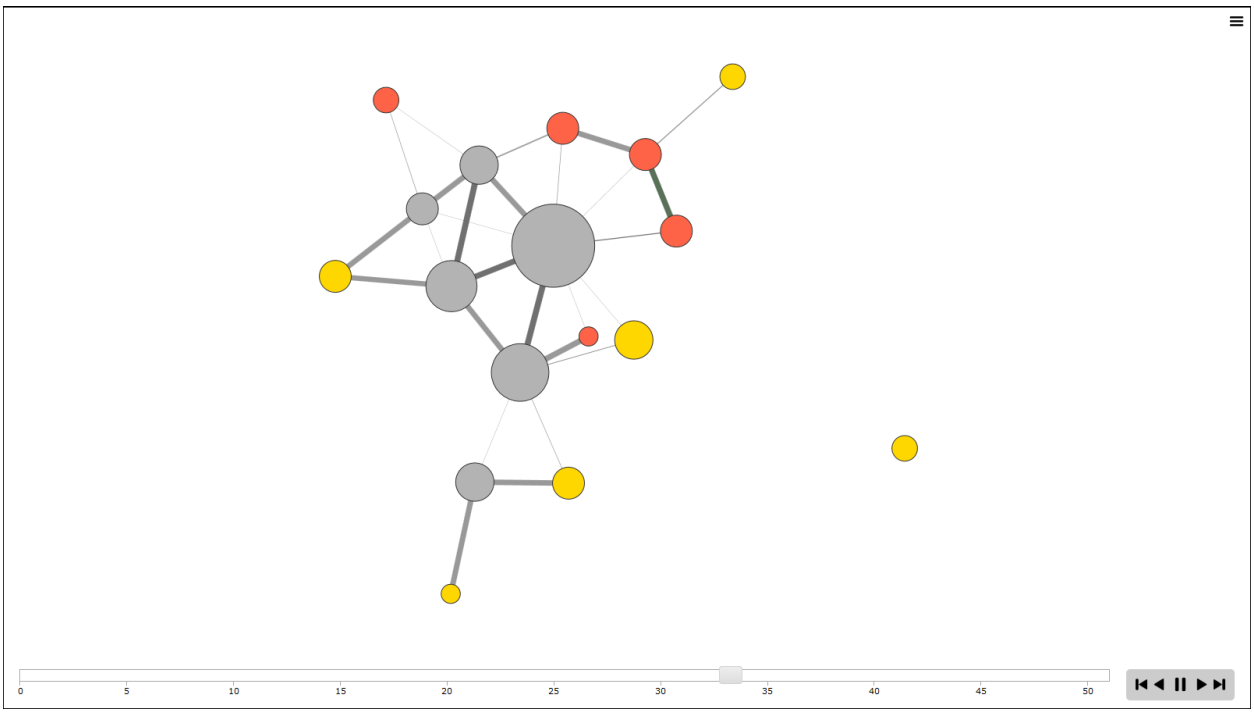
```
filmstrip(net3.dyn, displaylabels=F, mfrow=c(1, 5),
          slice.par=list(start=0, end=49, interval=10,
                         aggregate.dur=10, rule='any'))
```



We can pre-compute the animation coordinates (otherwise they get calculated when you generate the animation). Here animation.mode is the layout algorithm - one of “kamadakawai”, “MDSJ”, “Graphviz”and “useAttribute” (user-generated coordinates).

```
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, we add parameter `output.mode='inline'`.

In addition to dynamic nodes and edges, ndtv takes dynamic attributes. We could have added those to the 'es' and 'vs' data frames above. In addition, the plotting function can 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, 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'))
```