Network analysis with R and igraph

Ottawa Graph meetup, June 2018

Katherine Ognyanova, www.kateto.net

Contents

1 Networks in igraph **3**
- 1.1 *Create networks* . . . 3
- 1.2 *Edge, vertex, and network attributes* . . . 6
- 1.3 *Specific graphs and graph models* . . . 9

2 Reading network data **14**
- 2.1 *Reading edgelist data (Dataset 1)* . . . 14
- 2.2 *Reading matrix data (Dataset 2)* . . . 15
- 2.3 *Creating igraph objects* . . . 15

3 Plotting network data **19**
- 3.1 *Plotting parameters* . . . 19
- 3.2 *Network layouts* . . . 23
- 3.3 *Interactive plotting with tkplot* . . . 30

4 Network and node descriptives **31**
- 4.1 *Density* . . . 31
- 4.2 *Reciprocity* . . . 31
- 4.3 *Transitivity* . . . 31
- 4.4 *Diameter* . . . 32
- 4.5 *Node degrees* . . . 33
- 4.6 *Degree distribution* . . . 34
- 4.7 *Centrality & centralization* . . . 35

5 Exploring network data **35**
- 5.1 *Hubs and authorities* . . . 35
- 5.2 *Distances and paths* . . . 36
- 5.3 *Subgroups and communities* . . . 40
- 5.4 *Assortativity and homophily* . . . 44



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

This workshop provides a brief introduction to the igraph R package (maintained by Gabor Csardi and Tamas Nepusz). The tutorial covers basics of network analysis and visualization with R and `igraph`. The igraph library provides versatile options for descriptive network analysis and visualization in R, Python, and C/C++. This workshop will focus on the R implementation. You will need an R installation, and RStudio. You should also install the latest version of `igraph` for R:

```
install.packages("igraph")
```

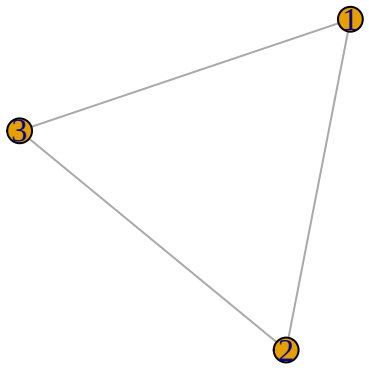
1 Networks in igraph

```
library(igraph) # Load the igraph package
```

1.1 *Create networks*

The code below generates an undirected graph with three edges. The numbers are interpreted as vertex IDs, so the edges are 1–>2, 2–>3, 3–>1.

```
g1 <- graph( edges=c(1,2, 2,3, 3, 1), n=3, directed=F )
plot(g1) # A simple plot of the network - we'll talk more about plots later
```



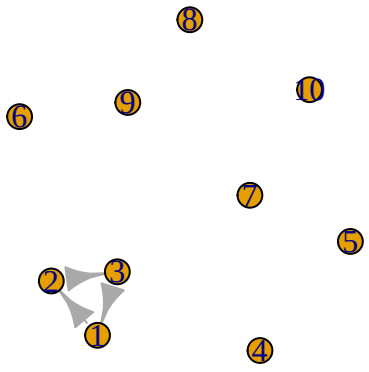

```
class(g1)
```

```
## [1] "igraph"
```

```
g1
```

```
## IGRAPH c9d0e79 U--- 3 3 --
## + edges from c9d0e79:
## [1] 1--2 2--3 1--3
```

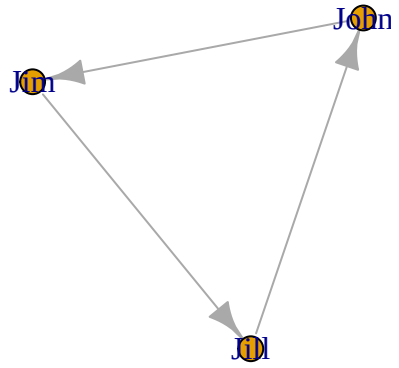
```
# Now with 10 vertices, and directed by default:
g2 <- graph( edges=c(1,2, 2,3, 3, 1), n=10 )
plot(g2)
```



```
g2
```

```
## IGRAPH c9de329 D--- 10 3 --
## + edges from c9de329:
## [1] 1->2 2->3 3->1
```

```
g3 <- graph( c("John", "Jim", "Jim", "Jill", "Jill", "John")) # named vertices
# When the edge list has vertex names, the number of nodes is not needed
plot(g3)
```

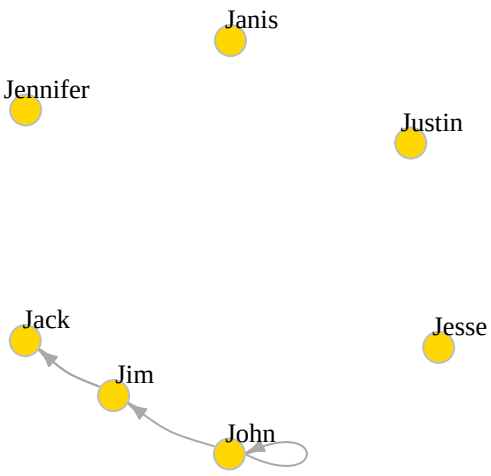



```
g3
```

```
## IGRAPH c9e17ef DN-- 3 3 --
## + attr: name (v/c)
## + edges from c9e17ef (vertex names):
## [1] John->Jim  Jim ->Jill Jill->John
```

```
g4 <- graph( c("John", "Jim", "Jim", "Jack", "Jim", "Jack", "John", "John"),
             isolates=c("Jesse", "Janis", "Jennifer", "Justin") )
# In named graphs we can specify isolates by providing a list of their names.

plot(g4, edge.arrow.size=.5, vertex.color="gold", vertex.size=15,
     vertex.frame.color="gray", vertex.label.color="black",
     vertex.label.cex=0.8, vertex.label.dist=2, edge.curved=0.2)
```

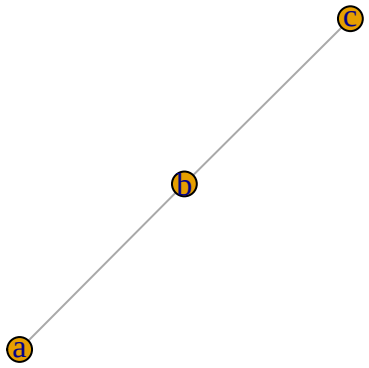


As in the example above, in most R functions, you can use named colors, hex, or rgb values. For example: "blue" "dark red" "#557799" or rgb(.25, .5, .3)) You can see the built-in colors name with `colors()`:

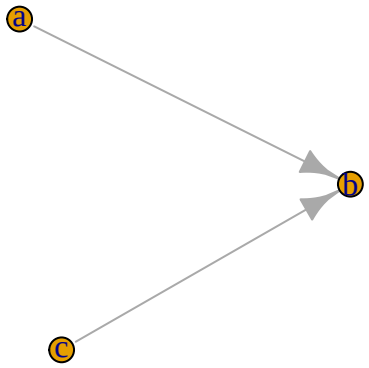
```
colors()
```

Small graphs can also be generated with a description of this kind: `-` for undirected tie, `+-` or `-+` for directed ties pointing left & right, `++` for a symmetric tie, and ":" for sets of vertices.

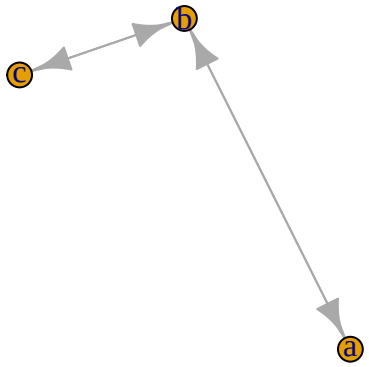
```
plot(graph_from_literal(a---b, b---c)) # the number of dashes doesn't matter
```



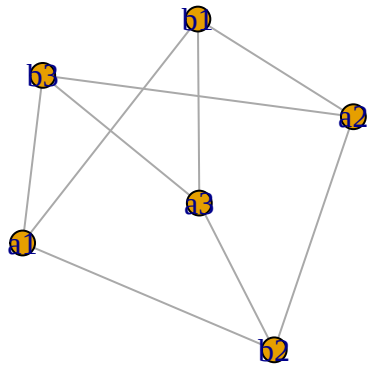

```
plot(graph_from_literal(a--+b, b+--c))
```



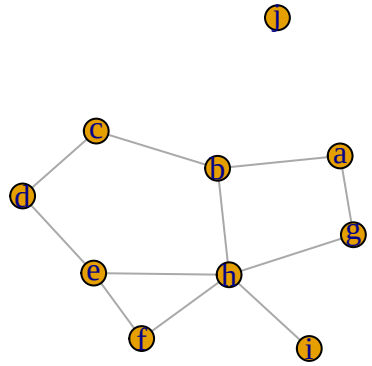

```
plot(graph_from_literal(a+-+b, b+-+c))
```



```
plot(graph_from_literal(a1:a2:a3---b1:b2:b3))
```




```
gl <- graph_from_literal(a-b-c-d-e-f, a-g-h-b, h-e:f:i, j)
plot(gl)
```



1.2 *Edge, vertex, and network attributes*

Access vertices and edges:

```
E(g4) # The edges of the object
```

```
## + 4/4 edges from c9e6267 (vertex names):
## [1] John->Jim  Jim ->Jack Jim ->Jack John->John
```

```
V(g4) # The vertices of the object
```

```
## + 7/7 vertices, named, from c9e6267:
## [1] John     Jim      Jack     Jesse    Janis    Jennifer Justin
```

You can also examine the network matrix directly:

```
g4[]
```

```
## 7 x 7 sparse Matrix of class "dgCMatrix"
##          John Jim Jack Jesse Janis Jennifer Justin
## John        1   1    .     .     .        .      .
## Jim         .   .    2     .     .        .      .
```

```
## Jack        .   .    .     .     .        .      .
## Jesse       .   .    .     .     .        .      .
## Janis       .   .    .     .     .        .      .
## Jennifer    .   .    .     .     .        .      .
## Justin      .   .    .     .     .        .      .
```

```
g4[1,]
```

```
##     John      Jim     Jack    Jesse    Janis Jennifer   Justin
##        1        1        0        0        0        0        0
```

Add attributes to the network, vertices, or edges:

```
V(g4)$name # automatically generated when we created the network.
```

```
## [1] "John"     "Jim"      "Jack"     "Jesse"    "Janis"    "Jennifer"
## [7] "Justin"
```

```
V(g4)$gender <- c("male", "male", "male", "male", "female", "female", "male")
E(g4)$type <- "email" # Edge attribute, assign "email" to all edges
E(g4)$weight <- 10    # Edge weight, setting all existing edges to 10
```

Examine attributes:

```
edge_attr(g4)
```

```
## $type
## [1] "email" "email" "email" "email"
##
## $weight
## [1] 10 10 10 10
```

```
vertex_attr(g4)
```

```
## $name
## [1] "John"     "Jim"      "Jack"     "Jesse"    "Janis"    "Jennifer"
## [7] "Justin"
##
## $gender
## [1] "male"   "male"   "male"   "male"   "female" "female" "male"
```

```
graph_attr(g4)
```

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

Another way to set attributes (you can similarly use `set_edge_attr()`, `set_vertex_attr()`, etc.):

```
g4 <- set_graph_attr(g4, "name", "Email Network")
g4 <- set_graph_attr(g4, "something", "A thing")

graph_attr_names(g4)
```

```
## [1] "name"      "something"
```

```
graph_attr(g4, "name")
```

```
## [1] "Email Network"
```

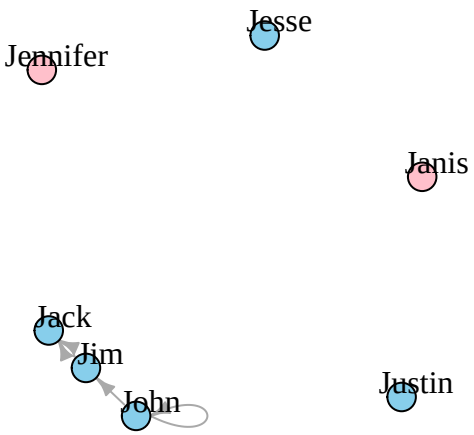
```
graph_attr(g4)
```

```
## $name
## [1] "Email Network"
##
## $something
## [1] "A thing"
```

```
g4 <- delete_graph_attr(g4, "something")
graph_attr(g4)
```

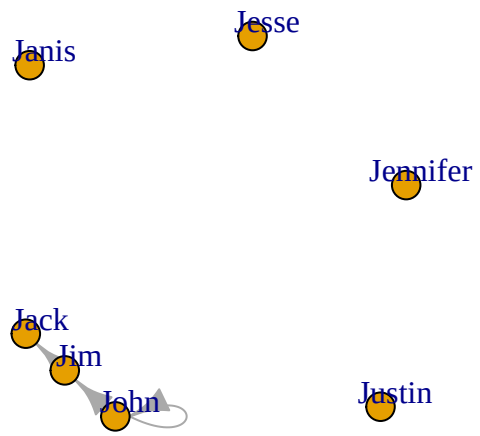
```
## $name
## [1] "Email Network"
```

```
plot(g4, edge.arrow.size=.5, vertex.label.color="black", vertex.label.dist=1.5,
     vertex.color=c( "pink", "skyblue")[1+(V(g4)$gender=="male")] )
```



The graph `g4` has two edges going from Jim to Jack, and a loop from John to himself. We can simplify our graph to remove loops & multiple edges between the same nodes. Use `edge.attr.comb` to indicate how edge attributes are to be combined - possible options include `sum`, `mean`, `prod` (product), `min`, `max`, `first/last` (selects the first/last edge's attribute). Option "ignore" says the attribute should be disregarded and dropped.

```
g4s <- simplify( g4, remove.multiple = T, remove.loops = F,
                 edge.attr.comb=c(weight="sum", type="ignore") )
plot(g4s, vertex.label.dist=1.5)
```




```
g4s
```

```
## IGRAPH ca7358c DNW- 7 3 -- Email Network
## + attr: name (g/c), name (v/c), gender (v/c), weight (e/n)
## + edges from ca7358c (vertex names):
## [1] John->John John->Jim  Jim ->Jack
```

The description of an igraph object starts with up to 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)

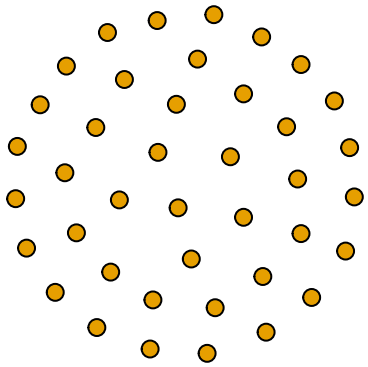
The two numbers that follow (7 5) refer to the number of nodes and edges in the graph. The description also lists node & edge attributes, for example:

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

1.3 *Specific graphs and graph models*

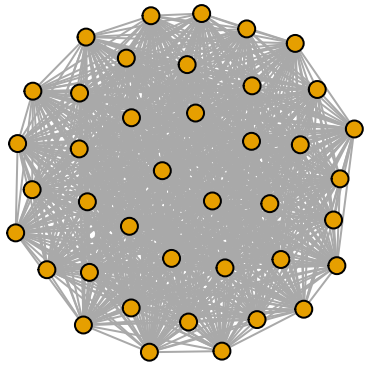
Empty graph

```
eg <- make_empty_graph(40)
plot(eg, vertex.size=10, vertex.label=NA)
```



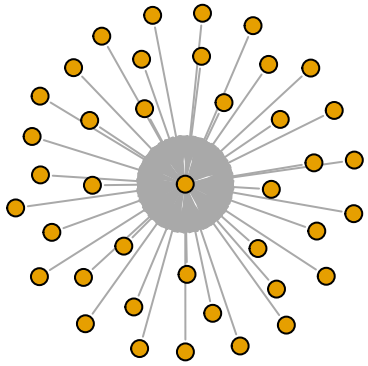
Full graph

```
fg <- make_full_graph(40)
plot(fg, vertex.size=10, vertex.label=NA)
```



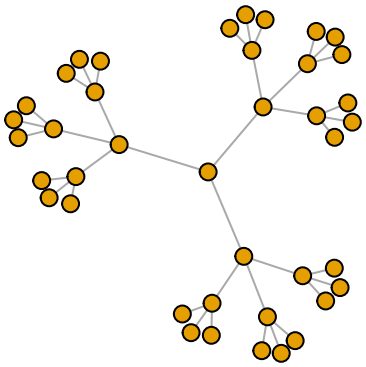
Simple star graph

```
st <- make_star(40)
plot(st, vertex.size=10, vertex.label=NA)
```



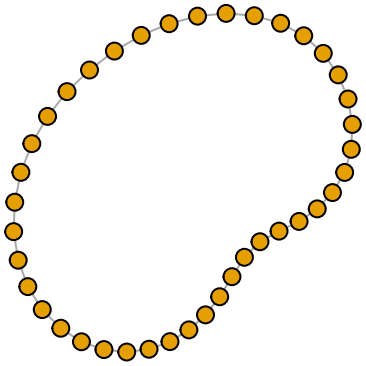
Tree graph

```
tr <- make_tree(40, children = 3, mode = "undirected")
plot(tr, vertex.size=10, vertex.label=NA)
```



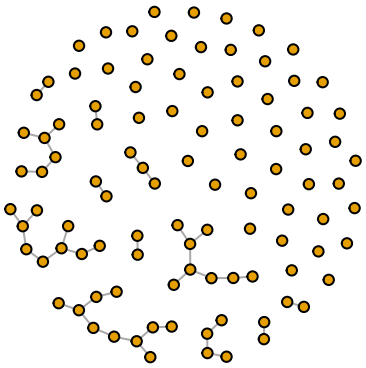
Ring graph

```
rn <- make_ring(40)
plot(rn, vertex.size=10, vertex.label=NA)
```



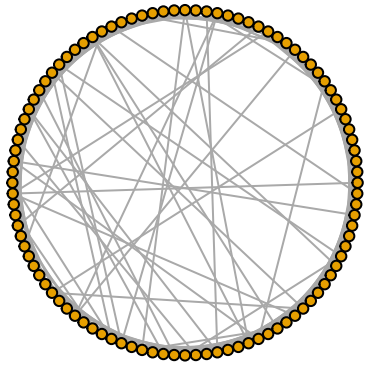
Erdos-Renyi random graph model
('n' is number of nodes, 'm' is the number of edges).

```
er <- sample_gnm(n=100, m=40)
plot(er, vertex.size=6, vertex.label=NA)
```



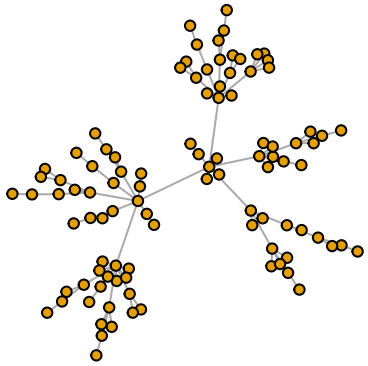
Watts-Strogatz small-world model
Creates a lattice (with `dim` dimensions and `size` nodes across dimension) and rewires edges randomly with probability `p`. The neighborhood in which edges are connected is `nei`. You can allow `loops` and `multiple` edges.

```
sw <- sample_smallworld(dim=2, size=10, nei=1, p=0.1)
plot(sw, vertex.size=6, vertex.label=NA, layout=layout_in_circle)
```



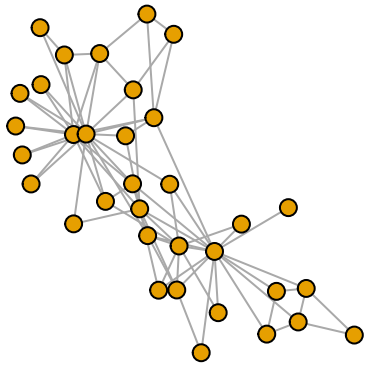
Barabasi-Albert preferential attachment model for scale-free graphs
(`n` is number of nodes, `power` is the power of attachment (`1` is linear); `m` is the number of edges added on each time step)

```
ba <-  sample_pa(n=100, power=1, m=1,  directed=F)
plot(ba, vertex.size=6, vertex.label=NA)
```



`igraph` can also give you some notable historical graphs. For instance:

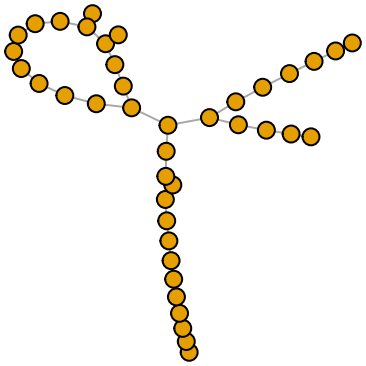
```
zach <- graph("Zachary") # the Zachary carate club
plot(zach, vertex.size=10, vertex.label=NA)
```



Rewiring a graph

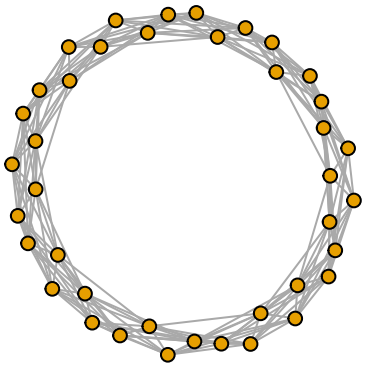
`each_edge()` is a rewiring method that changes the edge endpoints uniformly randomly with a probability `prob`.

```
rn.rewired <- rewire(rn, each_edge(prob=0.1))
plot(rn.rewired, vertex.size=10, vertex.label=NA)
```



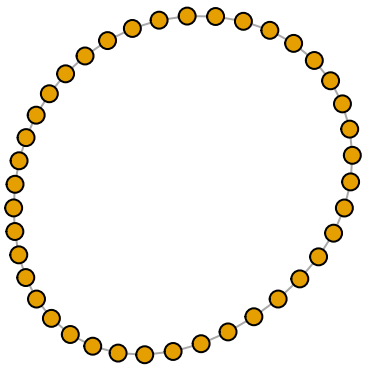
Rewire to connect vertices to other vertices at a certain distance.

```
rn.neigh = connect.neighborhood(rn, 5)
plot(rn.neigh, vertex.size=8, vertex.label=NA)
```

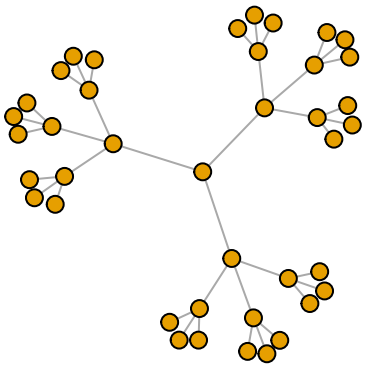


Combine graphs (disjoint union, assuming separate vertex sets): `%du%`

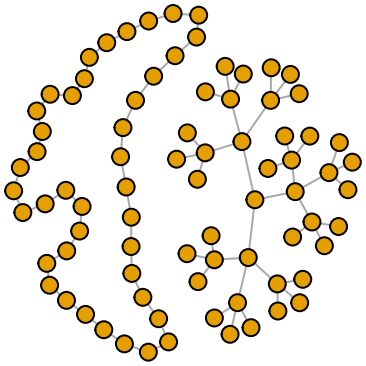
```
plot(rn, vertex.size=10, vertex.label=NA)
```



```
plot(tr, vertex.size=10, vertex.label=NA)
```



```
plot(rn %du% tr, vertex.size=10, vertex.label=NA)
```



---

2 Reading network data

In the following sections of the 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 analyses and visualizations we will generate apply to medium and large-scale networks.

2.1 *Reading edgelist data (Dataset 1)*

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". We don't use `simplify()` here so as not to collapse different link types.

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

2.2 *Reading matrix data (Dataset 2)*

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.

```
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)
```

2.3 *Creating igraph objects*

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.

Dataset 1

```
library(igraph)

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

```
## [1] "igraph"
```

```
net
```

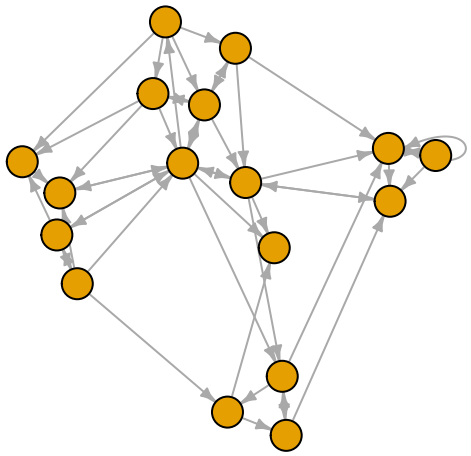
```
## IGRAPH cac4bcb DNW- 17 49 --
## + attr: name (v/c), media (v/c), media.type (v/n), type.label
## | (v/c), audience.size (v/n), type (e/c), weight (e/n)
## + edges from cac4bcb (vertex names):
##  [1] s01->s02 s01->s03 s01->s04 s01->s15 s02->s01 s02->s03 s02->s09
##  [8] s02->s10 s03->s01 s03->s04 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
```

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"
```

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

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



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").

If you need them, you can extract an edge list or a matrix from igraph networks.

```
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")
```

Dataset 2

As we have seen above, this time the edges of the 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)
```

```
##    id   media media.type media.name audience.size
## 1 s01     NYT          1  Newspaper            20
## 2 s02    WaPo          1  Newspaper            25
## 3 s03     WSJ          1  Newspaper            30
## 4 s04    USAT          1  Newspaper            32
## 5 s05 LATimes          1  Newspaper            20
## 6 s06     CNN          2         TV            56
```

```
head(links2)
```

```
##     U01 U02 U03 U04 U05 U06 U07 U08 U09 U10 U11 U12 U13 U14 U15 U16 U17
## s01   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## s02   0   0   0   1   1   0   0   0   0   0   0   0   0   0   0   0   0
## s03   0   0   0   0   0   1   1   1   1   0   0   0   0   0   0   0   0
## s04   0   0   0   0   0   0   0   0   1   1   1   0   0   0   0   0   0
## s05   0   0   0   0   0   0   0   0   0   0   1   1   1   0   0   0   0
## s06   0   0   0   0   0   0   0   0   0   0   0   0   1   1   0   0   1
##     U18 U19 U20
## s01   0   0   0
## s02   0   0   1
## s03   0   0   0
## s04   0   0   0
## s05   0   0   0
## s06   0   0   0
```

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

```
##
```

```
## FALSE  TRUE
##    10    20
```

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

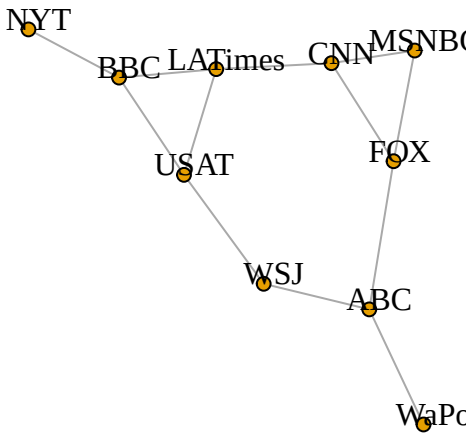
We can also easily generate bipartite projections for the two-mode network: (co-memberships are easy to calculate by multiplying the network matrix by its transposed matrix, or using igraph's `bipartite.projection()` function).

```
net2.bp <- bipartite.projection(net2)
```

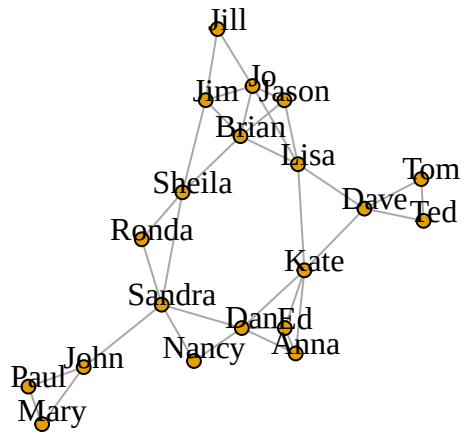
We can calculate the projections manually as well:

```
   as_incidence_matrix(net2)  %*% t(as_incidence_matrix(net2))
 t(as_incidence_matrix(net2)) %*%   as_incidence_matrix(net2)
```

```
plot(net2.bp$proj1, vertex.label.color="black", vertex.label.dist=1,
     vertex.size=7, vertex.label=nodes2$media[!is.na(nodes2$media.type)])
```




```
plot(net2.bp$proj2, vertex.label.color="black", vertex.label.dist=1,
     vertex.size=7, vertex.label=nodes2$media[ is.na(nodes2$media.type)])
```



3 Plotting network data

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.

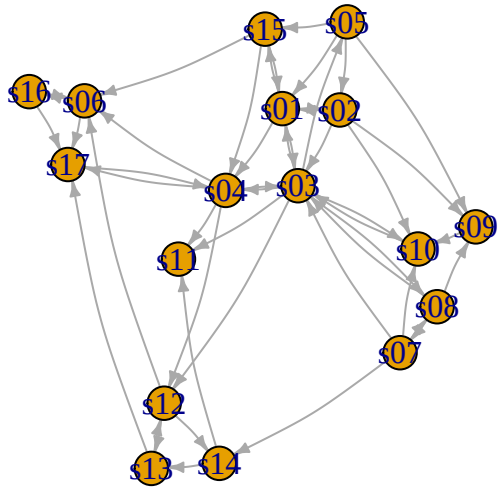
The igraph plotting parameters include (among others):

3.1 *Plotting parameters*

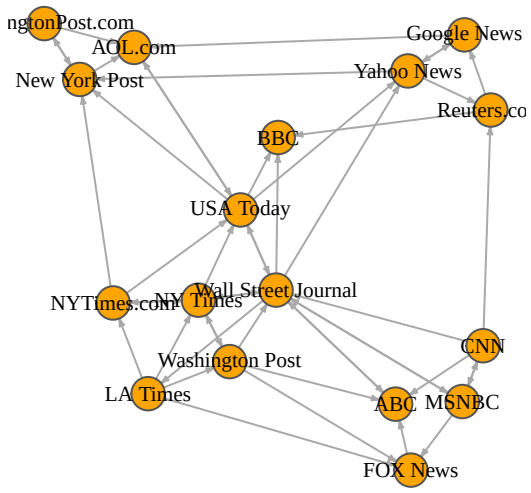
NODES	
vertex.color	Node color
vertex.frame.color	Node border color
vertex.shape	One of "none", "circle", "square", "csquare", "rectangle" "crectangle", "vrectangle", "pie", "raster", or "sphere"
vertex.size	Size of the node (default is 15)
vertex.size2	The second size of the node (e.g. for a rectangle)
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 gray, and the node color to orange.
# Replace the vertex label with the node names stored in "media"
plot(net, edge.arrow.size=.2, edge.curved=0,
     vertex.color="orange", vertex.frame.color="#555555",
     vertex.label=V(net)$media, vertex.label.color="black",
     vertex.label.cex=.7)
```



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 audience size (larger audience -> 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]

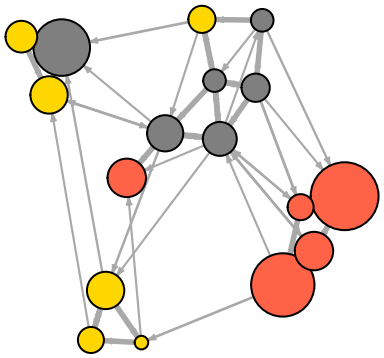
# Set node size based on audience size:
V(net)$size <- V(net)$audience.size*0.7

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

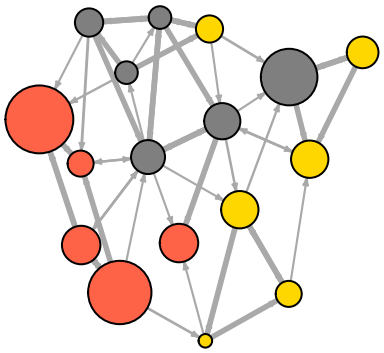
# Set edge width based on weight:
E(net)$width <- E(net)$weight/6

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

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

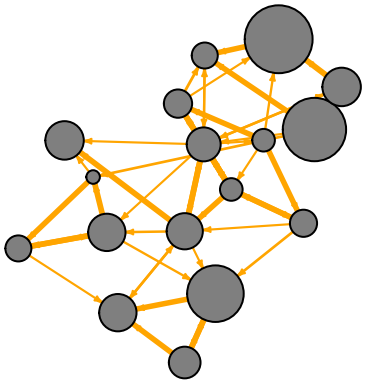


```
# We can even set the network layout:
graph_attr(net, "layout") <- layout_with_lgl
plot(net)
```



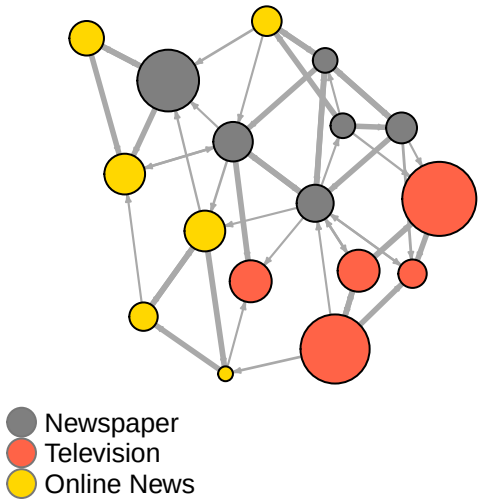
We can also override the attributes explicitly in the plot:

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



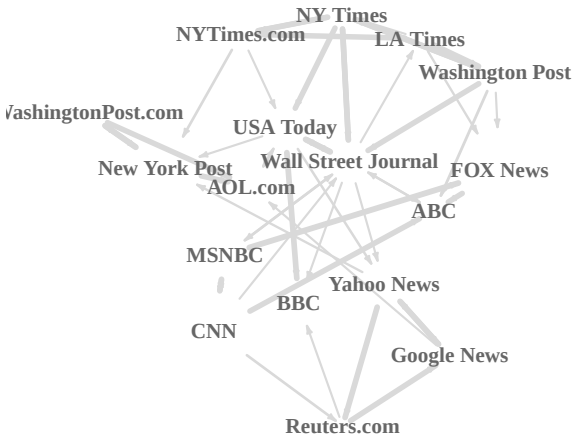
It helps to add a legend explaining the meaning of the colors we used:

```
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:

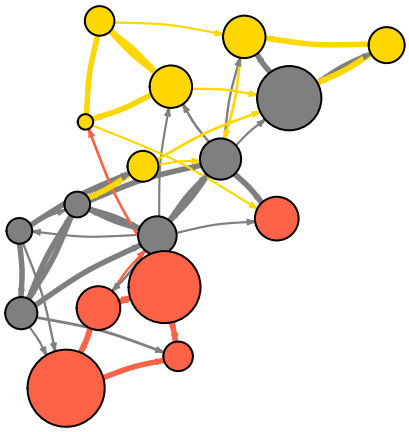
```
plot(net, vertex.shape="none", vertex.label=V(net)$media,
     vertex.label.font=2, vertex.label.color="gray40",
     vertex.label.cex=.7, edge.color="gray85")
```



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.

```
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)
```

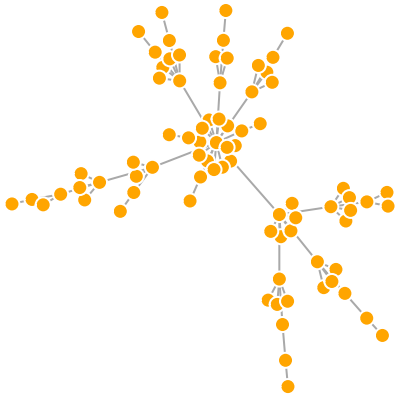


3.2 *Network layouts*

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

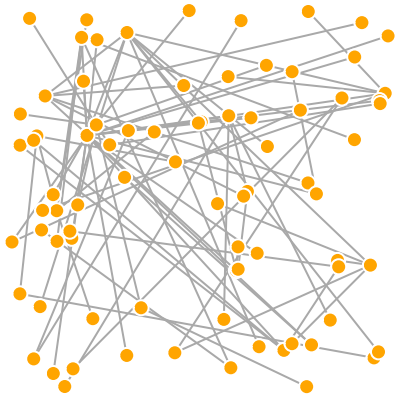
For the purposes of exploring layouts, we will generate a slightly larger 80-node graph. We use the `sample_pa()` function which generates a simple graph starting from one node and adding more nodes and links based on a preset level of preferential attachment (Barabasi-Albert model).

```
net.bg <- sample_pa(80)
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)
```



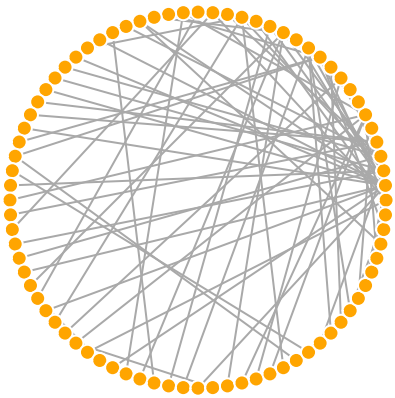
You can set the layout in the plot function:

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



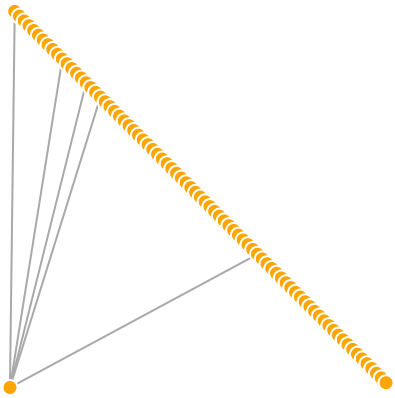
Or you can calculate the vertex coordinates in advance:

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



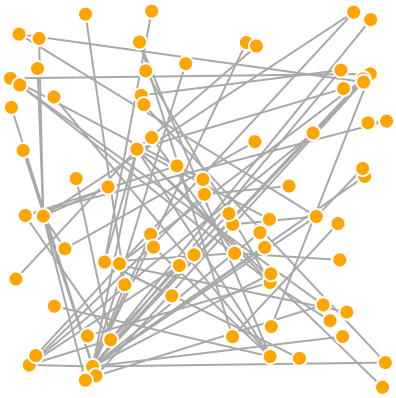
`l` is simply a matrix of x, y coordinates (N x 2) for the N nodes in the graph. You can easily generate your own:

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

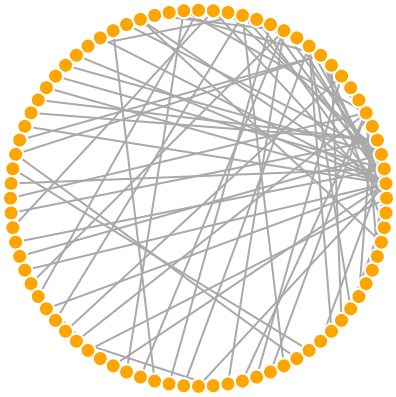


This layout is just an example and not very helpful - thankfully igraph has a number of built-in layouts, including:

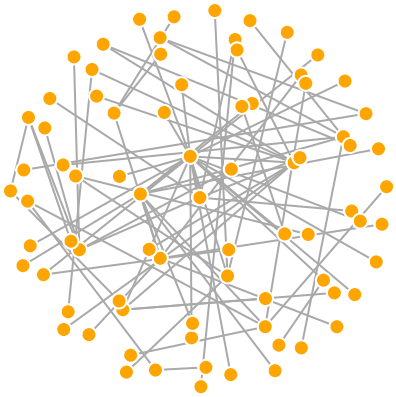
```
# Randomly placed vertices
l <- layout_randomly(net.bg)
plot(net.bg, layout=l)
```



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



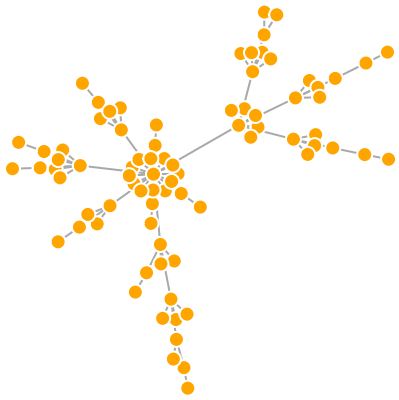
```
# 3D sphere layout
l <- layout_on_sphere(net.bg)
plot(net.bg, layout=l)
```



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

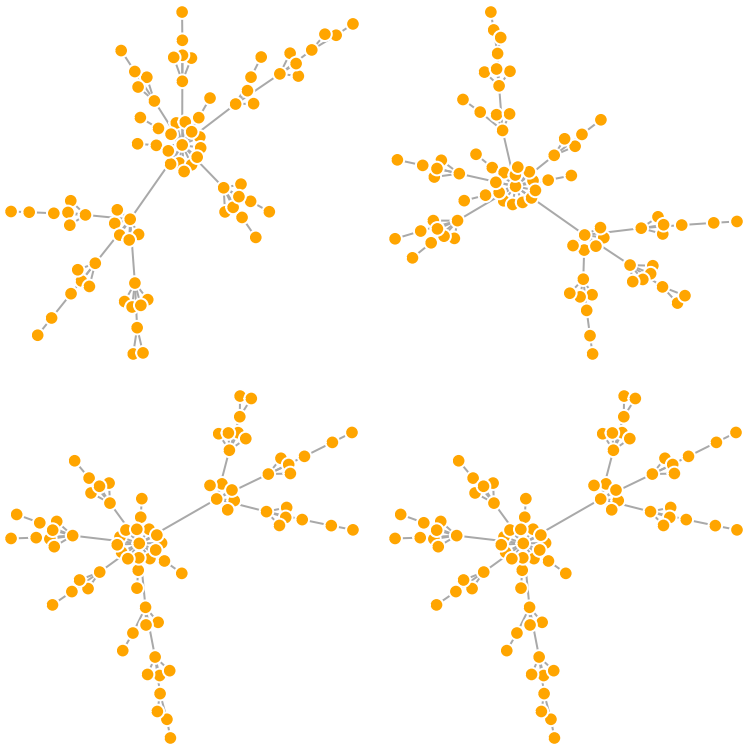
Force-directed layouts try to get a nice-looking graph where edges are similar in length and cross each other as little as possible. They simulate the graph as a physical system. Nodes are electrically charged particles that repulse each other when they get too close. The edges act as springs that attract 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. You can set the "weight" parameter which increases the attraction forces among nodes connected by heavier edges.

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



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.

```
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)
```



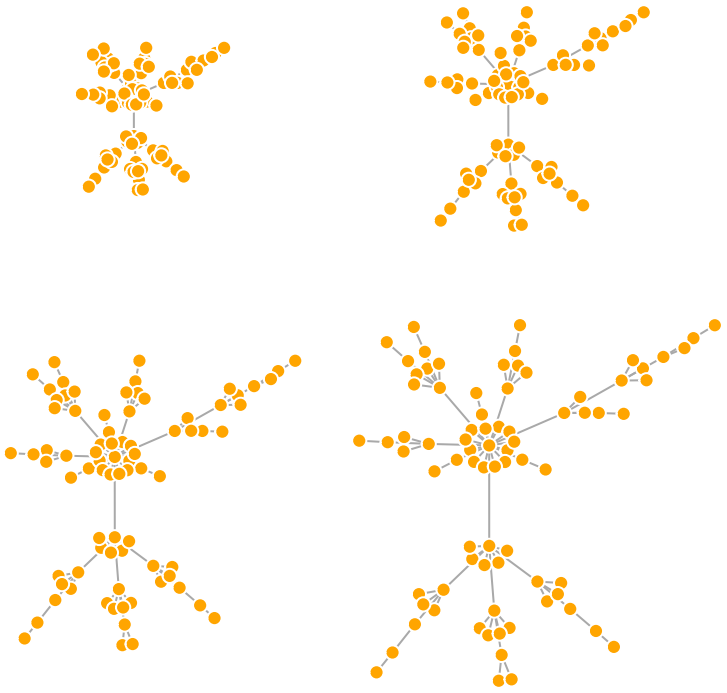
```
dev.off()
```

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.

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

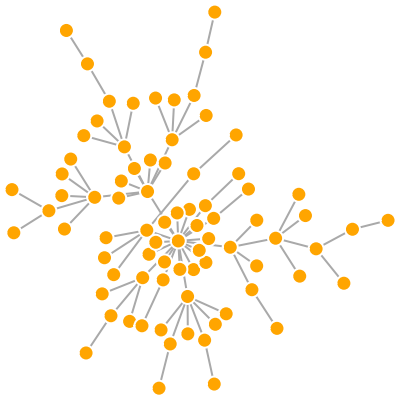
par(mfrow=c(2,2), mar=c(0,0,0,0))
plot(net.bg, rescale=F, layout=l*0.4)
plot(net.bg, rescale=F, layout=l*0.6)
plot(net.bg, rescale=F, layout=l*0.8)
plot(net.bg, rescale=F, layout=l*1.0)
```



```
dev.off()
```

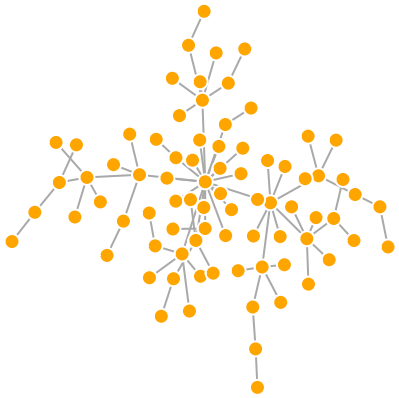
Another popular force-directed algorithm that produces nice results for connected graphs is Kamada Kawai. Like Fruchterman Reingold, it attempts to minimize the energy in a spring system.

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



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.

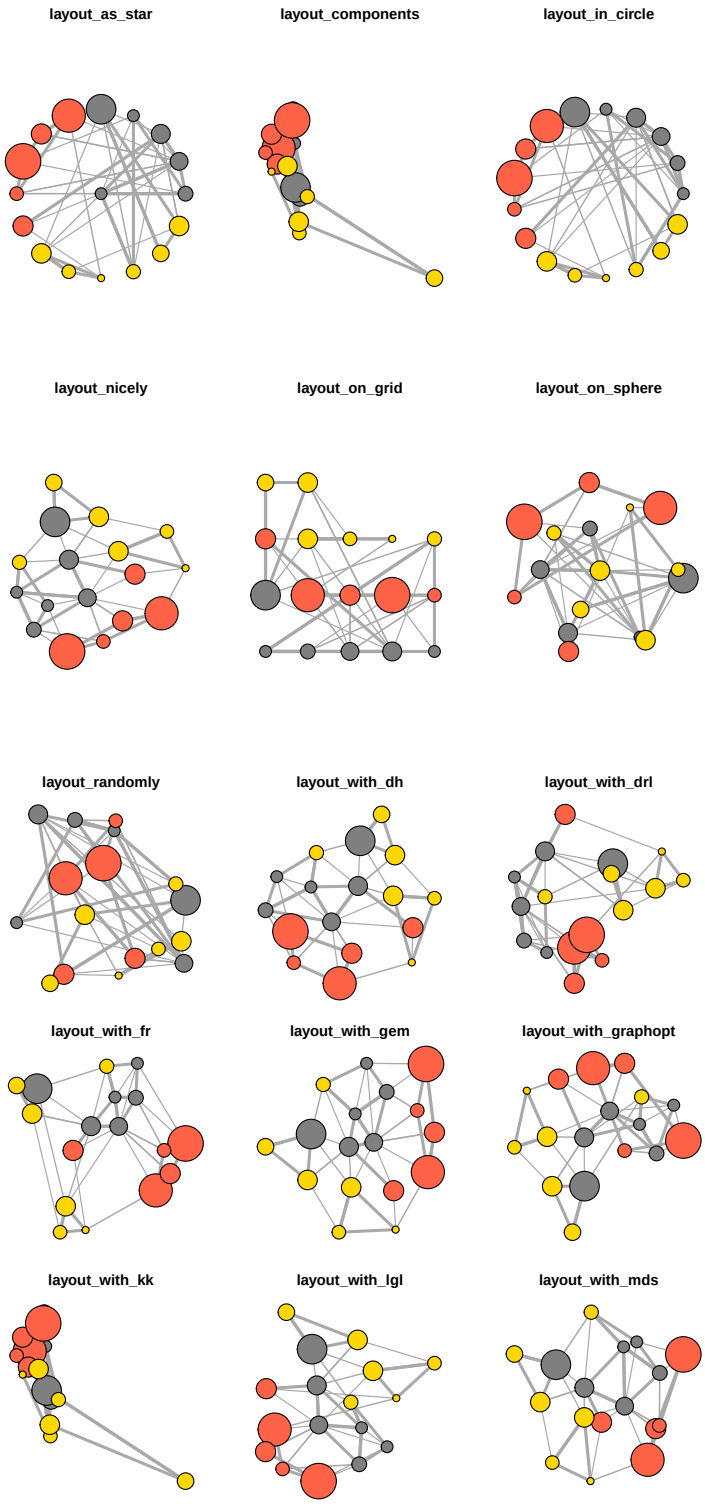
```
plot(net.bg, layout=layout_with_lgl)
```



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) }
```

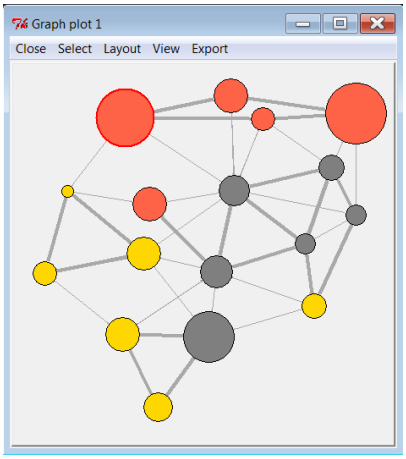


3.3 *Interactive plotting with tkplot*

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
tk_close(tkid, window.close = T)
plot(net, layout=l)
```



---

4 Network and node descriptives

4.1 *Density*

The proportion of present edges from all possible edges in the network.

```
edge_density(net, loops=F)
ecount(net)/(vcount(net)*(vcount(net)-1)) #for a directed network
```

4.2 *Reciprocity*

The proportion of reciprocated ties (for a directed network).

```
reciprocity(net)
dyad_census(net) # Mutual, asymmetric, and null node pairs
2*dyad_census(net)$mut/ecount(net) # Calculating reciprocity
```

4.3 *Transitivity*

- global - ratio of triangles (direction disregarded) to connected triples.
- local - ratio of triangles to connected triples each vertex is part of.

```
transitivity(net, type="global")  # net is treated as an undirected network
transitivity(as.undirected(net, mode="collapse")) # same as above
```

```
transitivity(net, type="local")
triad_census(net) # for directed networks
```

Triad types (per Davis & Leinhardt):

- 003 A, B, C, empty triad.
- 012 A->B, C
- 102 A<->B, C
- 021D A<-B->C
- 021U A->B<-C
- 021C A->B->C
- 111D A<->B<-C
- 111U A<->B->C
- 030T A->B<-C, A->C
- 030C A<-B<-C, A->C.
- 201 A<->B<->C.
- 120D A<-B->C, A<->C.
- 120U A->B<-C, A<->C.
- 120C A->B->C, A<->C.
- 210 A->B<->C, A<->C.
- 300 A<->B<->C, A<->C, completely connected.

4.4 *Diameter*

A network diameter is the longest geodesic distance (length of the shortest path between two nodes) in the network. In `igraph`, `diameter()` returns the distance, while `get_diameter()` returns the nodes along the first found path of that distance.

Note that edge weights are used by default, unless set to NA.

```
diameter(net, directed=F, weights=NA)
```

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

```
diameter(net, directed=F)
```

```
## [1] 28
```

```
diam <- get_diameter(net, directed=T)
diam
```

```
## + 7/17 vertices, named, from cacbf6e:
## [1] s12 s06 s17 s04 s03 s08 s07
```

Note that `get_diameter()` returns a vertex sequence. Note though that when asked to behaved as a vector, a vertex sequence will produce the numeric indexes of the nodes in it. The same applies for edge sequences.

```
class(diam)
```

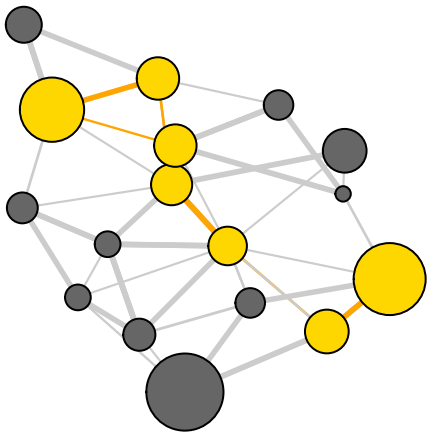
```
## [1] "igraph.vs"
```

```
as.vector(diam)
```

```
## [1] 12  6 17  4  3  8  7
```

Color nodes along the diameter:

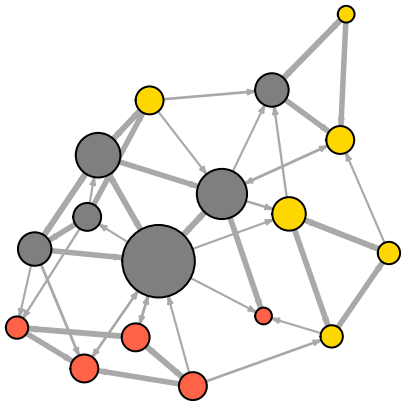
```
vcol <- rep("gray40", vcount(net))
vcol[diam] <- "gold"
ecol <- rep("gray80", ecount(net))
ecol[E(net, path=diam)] <- "orange"
# E(net, path=diam) finds edges along a path, here 'diam'
plot(net, vertex.color=vcol, edge.color=ecol, edge.arrow.mode=0)
```



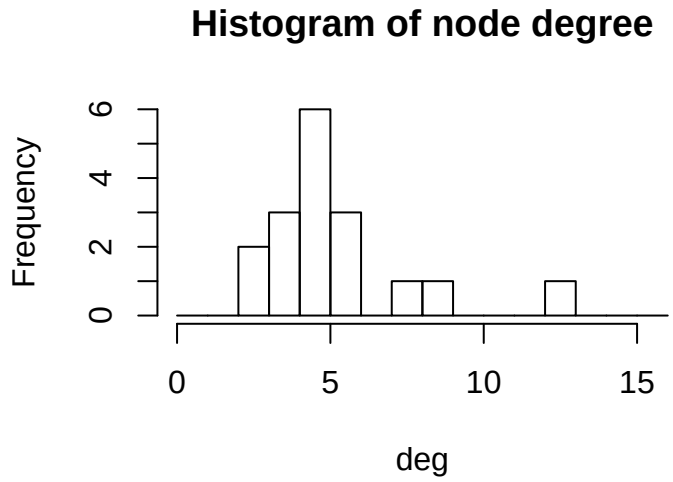
4.5 *Node degrees*

The function `degree()` has a mode of *`in`* for in-degree, *`out`* for out-degree, and *`all`* or *`total`* for total degree.

```
deg <- degree(net, mode="all")
plot(net, vertex.size=deg*3)
```

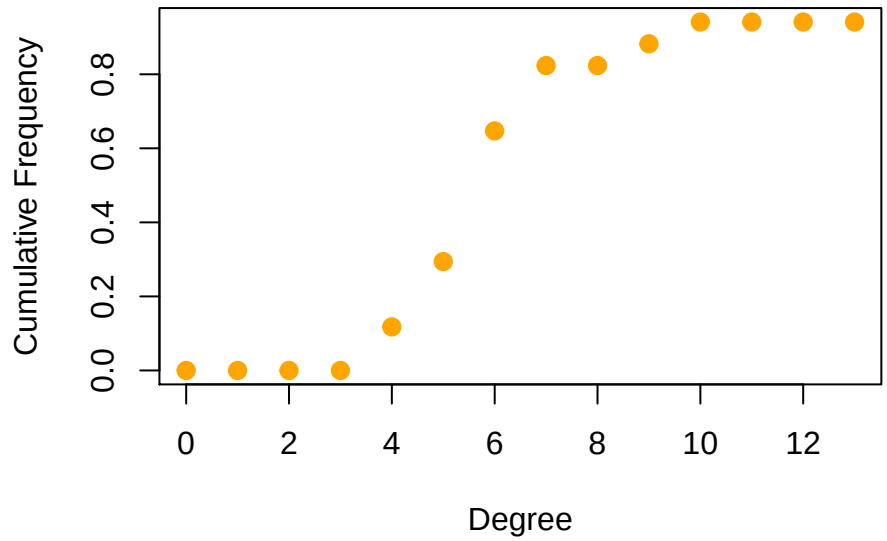


```r
hist(deg, breaks=1:vcount(net)-1, main="Histogram of node degree")
```



4.6 *Degree distribution*

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



4.7 *Centrality & centralization*

Centrality functions (vertex level) and centralization functions (graph level). The centralization functions return `res` - vertex centrality, `centralization`, and `theoretical_max` - maximum centralization score for a graph of that size. The centrality function can run on a subset of nodes (set with the `vids` parameter). This is helpful for large graphs where calculating all centralities may be a resource-intensive and time-consuming task.

Degree (number of ties)

```
degree(net, mode="in")
centr_degree(net, mode="in", normalized=T)
```

Closeness (centrality based on distance to others in the graph)
Inverse of the node's average geodesic distance to others in the network.

```
closeness(net, mode="all", weights=NA)
centr_clo(net, mode="all", normalized=T)
```

Eigenvector (centrality proportional to the sum of connection centralities)
Values of the first eigenvector of the graph matrix.

```
eigen_centrality(net, directed=T, weights=NA)
centr_eigen(net, directed=T, normalized=T)
```

Betweenness (centrality based on a broker position connecting others)
Number of geodesics that pass through the node or the edge.

```
betweenness(net, directed=T, weights=NA)
edge_betweenness(net, directed=T, weights=NA)
centr_betw(net, directed=T, normalized=T)
```

---

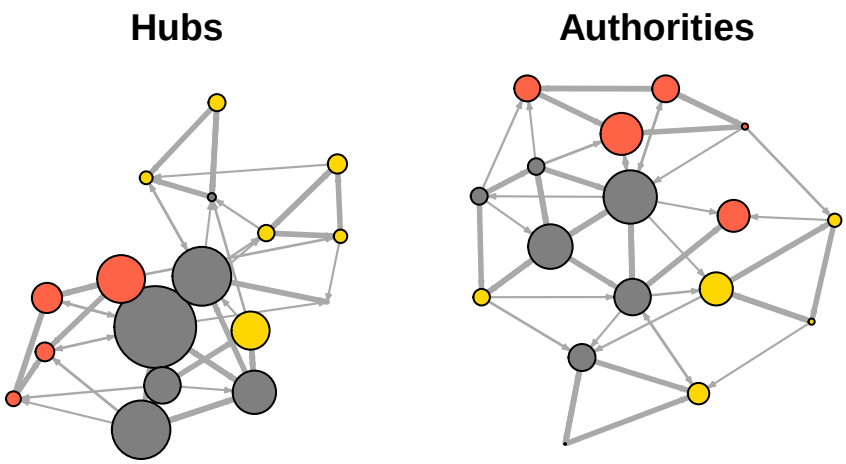
5 Exploring network data

5.1 *Hubs and authorities*

The hubs and authorities algorithm developed by Jon Kleinberg was initially used to examine web pages. Hubs were expected to contain catalogs with a large number of outgoing links; while authorities would get many incoming links from hubs, presumably because of their high-quality relevant information.

```
hs <- hub_score(net, weights=NA)$vector
as <- authority_score(net, weights=NA)$vector

par(mfrow=c(1,2))
 plot(net, vertex.size=hs*50, main="Hubs")
 plot(net, vertex.size=as*30, main="Authorities")
```




```
dev.off()
```

5.2 *Distances and paths*

Average path length: the mean of the shortest distance between each pair of nodes in the network (in both directions for directed graphs).

```
mean_distance(net, directed=F)
```

```
## [1] 2.058824
```

```
mean_distance(net, directed=T)
```

```
## [1] 2.742188
```

We can also find the length of all shortest paths in the graph:

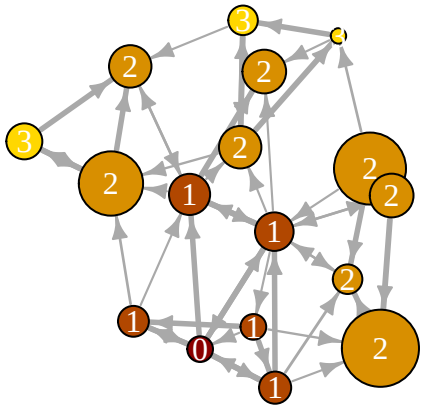
```
distances(net) # with edge weights
distances(net, weights=NA) # ignore weights
```

We can extract the distances to a node or set of nodes we are interested in. Here we will get the distance of every media from the New York Times.

```
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")
```

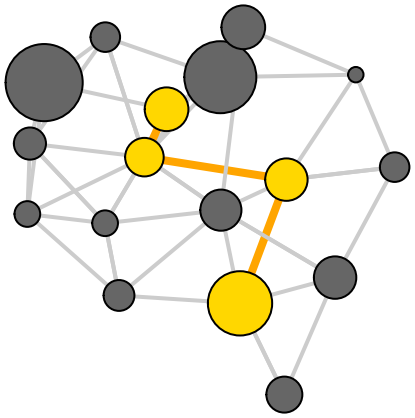


We can also find the shortest path between specific nodes. Say here between MSNBC and the New York Post:

```
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"

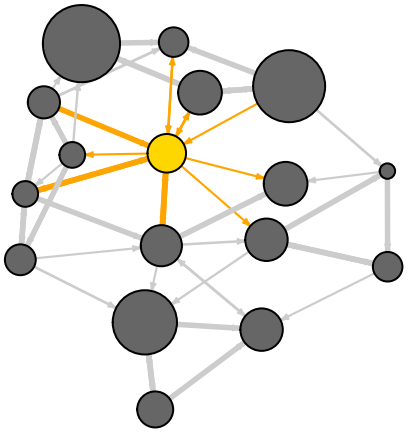
plot(net, vertex.color=vcol, edge.color=ecol,
     edge.width=ew, edge.arrow.mode=0)
```



Identify 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")

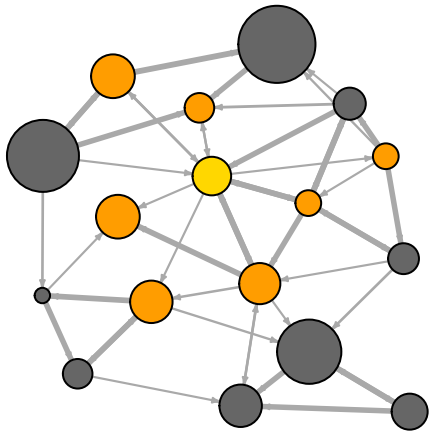
# Set colors to plot the selected edges.
ecol <- rep("gray80", ecount(net))
ecol[inc.edges] <- "orange"
vcol <- rep("grey40", vcount(net))
vcol[V(net)$media=="Wall Street Journal"] <- "gold"
plot(net, vertex.color=vcol, edge.color=ecol)
```



We can also easily identify 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)
```



Special operators for the indexing of edge sequences: %–%, %->%, %<-%
* E(network)[X %–% Y] selects edges between vertex sets X and Y, ignoring direction
* E(network)[X %->% Y] selects edges from vertex sets X to vertex set Y
* E(network)[X %->% Y] selects edges from vertex sets Y to vertex set X

For example, select edges from newspapers to online sources:

```
E(net)[ V(net)[type.label=="Newspaper"] %->% V(net)[type.label=="Online"] ]
```

```
## + 7/48 edges from cacbf6e (vertex names):
## [1] s01->s15 s03->s12 s04->s12 s04->s17 s05->s15 s06->s16 s06->s17
```

Co-citation (for a couple of nodes, how many shared nominations they have):

```
cocitation(net)
```

```
##     s01 s02 s03 s04 s05 s06 s07 s08 s09 s10 s11 s12 s13 s14 s15 s16 s17
## s01   0   1   1   2   1   1   0   1   2   2   1   1   0   0   1   0   0
## s02   1   0   1   1   0   0   0   0   1   0   0   0   0   0   2   0   0
## s03   1   1   0   1   0   1   1   1   2   2   1   1   0   1   1   0   1
## s04   2   1   1   0   1   1   0   1   0   1   1   1   0   0   1   0   0
## s05   1   0   0   1   0   0   0   1   0   1   1   1   0   0   0   0   0
## s06   1   0   1   1   0   0   0   0   0   0   1   1   1   1   0   0   2
## s07   0   0   1   0   0   0   0   0   1   0   0   0   0   0   0   0   0
## s08   1   0   1   1   1   0   0   0   0   2   1   1   0   1   0   0   0
## s09   2   1   2   0   0   0   1   0   0   1   0   0   0   0   1   0   0
## s10   2   0   2   1   1   0   0   2   1   0   1   1   0   1   0   0   0
## s11   1   0   1   1   1   1   0   1   0   1   0   2   1   0   0   0   1
## s12   1   0   1   1   1   1   0   1   0   1   2   0   0   0   0   0   2
## s13   0   0   0   0   0   1   0   0   0   0   1   0   0   1   0   0   0
## s14   0   0   1   0   0   1   0   1   0   1   0   0   1   0   0   0   0
## s15   1   2   1   1   0   0   0   0   1   0   0   0   0   0   0   0   0
## s16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1
## s17   0   0   1   0   0   2   0   0   0   0   1   2   0   0   0   1   0
```

5.3 *Subgroups and communities*

Before we start, we will make our network undirected. There are several ways to do that conversion:

- We can create an undirected link between any pair of connected nodes (*mode="collapse"*)
- Create undirected link for each directed one in the network, potentially ending up with a multiplex graph (*mode="each"*)
- Create undirected link for each symmetric link in the graph (*mode="mutual"*).

In cases when we may have ties *A -> B* and *B -> A* ties collapsed into a single undirected link, we need to specify what to do with their edge attributes using the parameter 'edge.attr.comb' as we did earlier with `simplify()`. Here we have said that the 'weight' of the links should be summed, and all other edge attributes ignored and dropped.

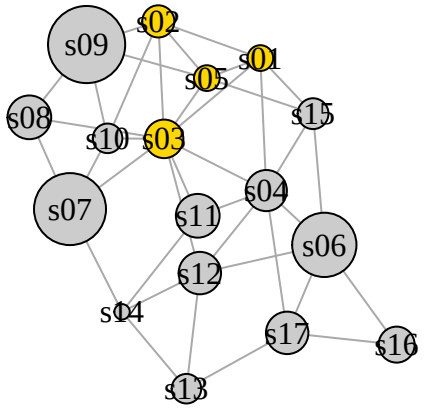
```
net.sym <- as.undirected(net, mode= "collapse",
                         edge.attr.comb=list(weight="sum", "ignore"))
```

5.3.1 *Cliques*

Find cliques (complete subgraphs of an undirected graph)

```
cliques(net.sym) # list of cliques
sapply(cliques(net.sym), length) # clique sizes
largest_cliques(net.sym) # cliques with max number of nodes
```

```
vcol <- rep("grey80", vcount(net.sym))
vcol[unlist(largest_cliques(net.sym))] <- "gold"
plot(as.undirected(net.sym), vertex.label=V(net.sym)$name, vertex.color=vcol)
```



5.3.2 *Community detection*

A number of algorithms aim to detect groups that consist of densely connected nodes with fewer connections across groups.

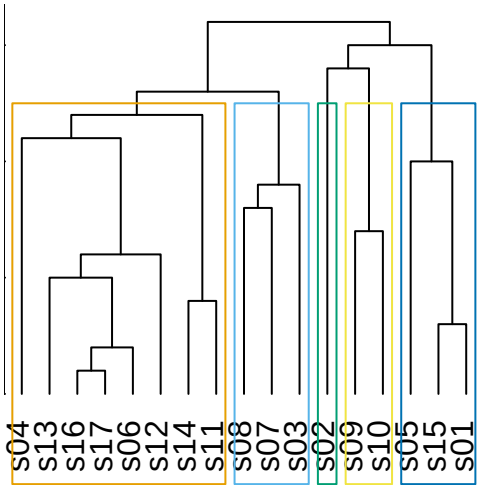
Community detection based on edge betweenness (Newman-Girvan)
High-betweenness edges are removed sequentially (recalculating at each step) and the best partitioning of the network is selected.

```
ceb <- cluster_edge_betweenness(net)
```

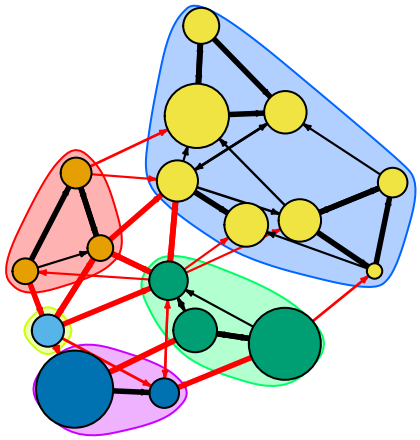
```
## Warning in cluster_edge_betweenness(net): At community.c:460 :Membership
## vector will be selected based on the lowest modularity score.
```

```
## Warning in cluster_edge_betweenness(net): At community.c:467 :Modularity
## calculation with weighted edge betweenness community detection might not
## make sense -- modularity treats edge weights as similarities while edge
## betwenness treats them as distances
```

```
dendPlot(ceb, mode="hclust")
```




```
plot(ceb, net)
```



Let's examine the community detection igraph object:

```
class(ceb)
```

```
## [1] "communities"
```

```
length(ceb)      # number of communities
```

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

```
membership(ceb) # community membership for each node
```

```
## s01 s02 s03 s04 s05 s06 s07 s08 s09 s10 s11 s12 s13 s14 s15 s16 s17
##   1   2   3   4   1   4   3   3   5   5   4   4   4   4   1   4   4
```

```
modularity(ceb) # how modular the graph partitioning is
```

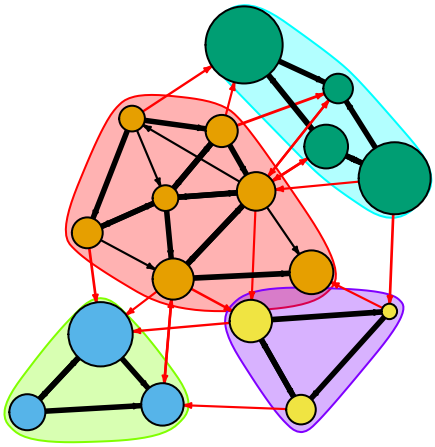
```
## [1] 0.292476
```

```
crossing(ceb, net)    # boolean vector: TRUE for edges across communities
```

High modularity for a partitioning reflects dense connections within communities and sparse connections across communities.

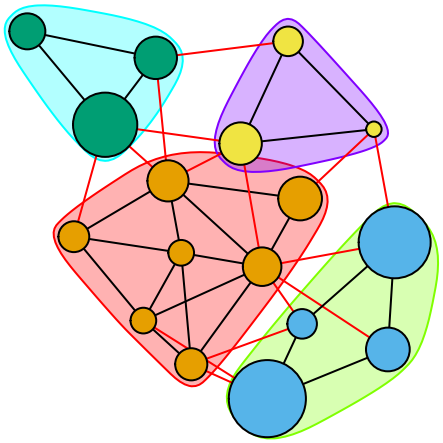
Community detection based on based on propagating labels
Assigns node labels, randomizes, than replaces each vertex's label with the label that appears most frequently among neighbors. Those steps are repeated until each vertex has the most common label of its neighbors.

```
clp <- cluster_label_prop(net)
plot(clp, net)
```



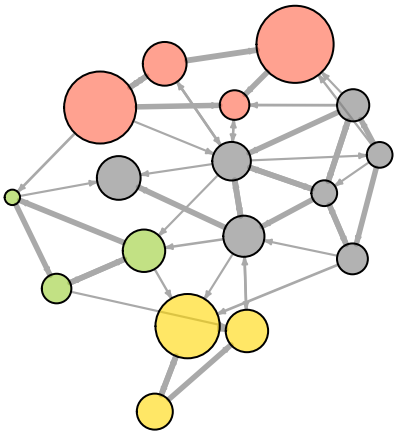
Community detection based on greedy optimization of modularity

```
cfg <- cluster_fast_greedy(as.undirected(net))
plot(cfg, as.undirected(net))
```



We can also plot the communities without relying on their built-in plot:

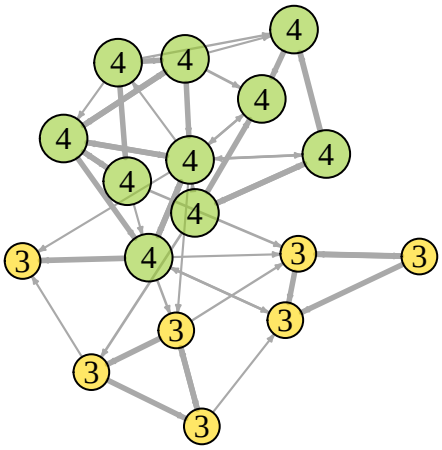
```
V(net)$community <- cfg$membership
colrs <- adjustcolor( c("gray50", "tomato", "gold", "yellowgreen"), alpha=.6)
plot(net, vertex.color=colrs[V(net)$community])
```



5.3.3 *K-core decomposition*

The k-core is the maximal subgraph in which every node has degree of at least k. This also means that the (k+1)-core will be a subgraph of the k-core.
The result here gives the coreness of each vertex in the network. A node has coreness D if it belongs to a D-core but not to (D+1)-core.

```
kc <- coreness(net, mode="all")
plot(net, vertex.size=kc*6, vertex.label=kc, vertex.color=colrs[kc])
```



5.4 *Assortativity and homophily*

Homophily: the tendency of nodes to connect to others who are similar on some variable.

- assortativity_nominal() is for categorical variables (labels)
- assortativity() is for ordinal and above variables
- assortativity_degree() checks assortativity in node degrees

```
assortativity_nominal(net, V(net)$media.type, directed=F)
```

```
## [1] 0.1715568
```

```
# Matching of attributes across connected nodes more than expected by chance

assortativity(net, V(net)$audience.size, directed=F)
```

```
## [1] -0.1102857
```

```
# Correlation of attributes across connected nodes

assortativity_degree(net, directed=F)
```

```
## [1] -0.009551146
```

```
# As above, with the focal attribute being the node degree D-1
```
