Lecture 5: Visualization of Network Data

Xiao Guo (Thanks for Katherine Ognyanova, www.kateto.net)

2023/3/19

5.1. Network analysis

Relationship: an irreducible property of two or more entities

- contrast to properties of entities alone ("attributes")

Focus of network analysis: The study of relational data arising from "social" entities

- Entities: people, animals, groups, locations, organizations, regions, etc.
- Relationships: communication, acquaintanceship, sexual contact, trade, migration rate, alliance/conflict, etc.

Network data: A collection of entities and a set of measured relations between them

- Entities: actors, nodes, vertices
- Relations: ties, links, edges

Relations can be

- directed or undirected
- signed or valued

Characteristics of network data

- Sparisity
- Hub
- Community
- Small-world

Network analysis

- Network modeling
- Community detection
- Link prediction

5.2. Networks in igraph

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.

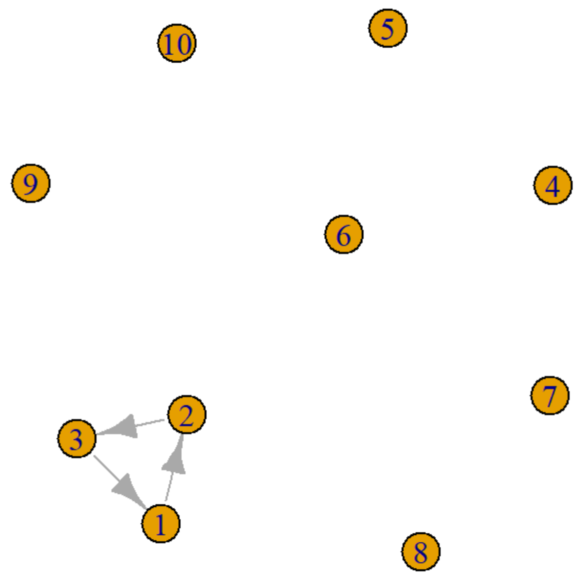
```
library(igraph)
```

```
##
## Attaching package: 'igraph'
```

```
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
```

```
## The following object is masked from 'package:base':
##
##     union
```

```
g1 <- graph( edges=c(1,2, 2,3, 3, 1), n=10, directed=T)

plot(g1) # A simple plot of the network - we'll talk more about pl
ots later
```



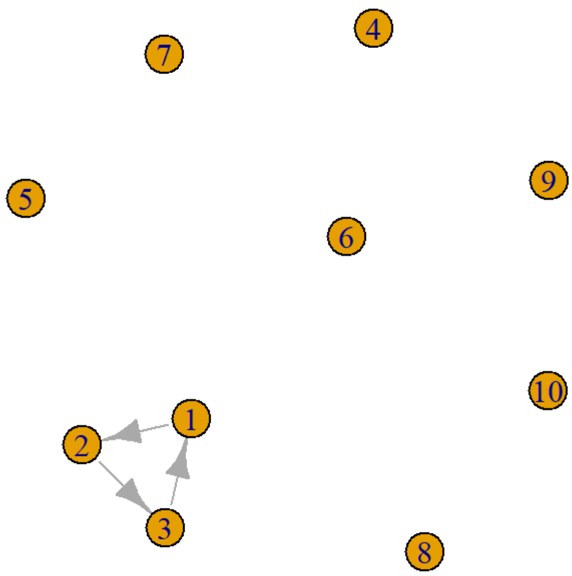

```
class(g1)
```

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

```
g1
```

```
## IGRAPH 2a4af51 D--- 10 3 --
## + edges from 2a4af51:
## [1] 1->2 2->3 3->1
```

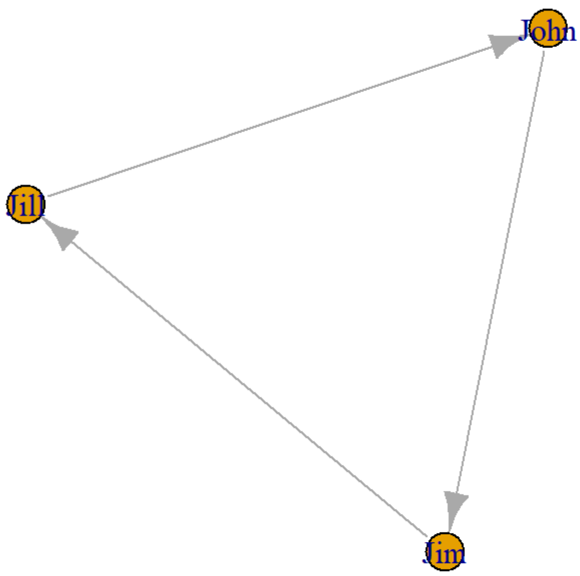
```
# Now with 10 vertices, and directed by default:

g2 <- graph( edges=c(1,2, 2,3, 3, 1), n=10 )

plot(g2)
```



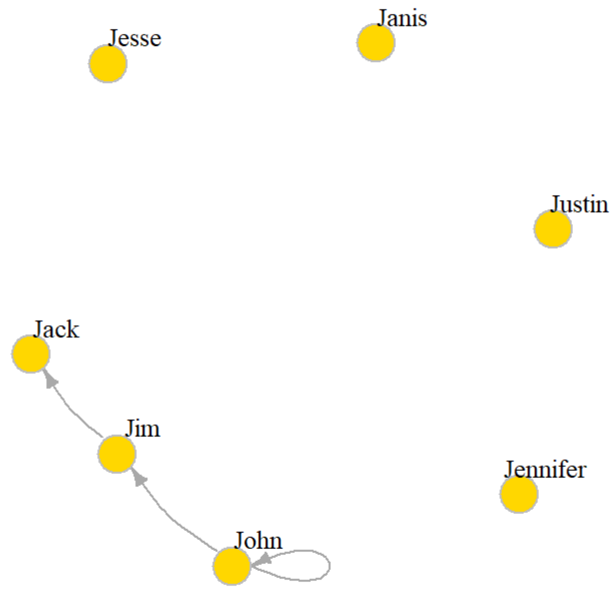

```
g2
```

```
## IGRAPH 2a9253e D--- 10 3 --
## + edges from 2a9253e:
## [1] 1->2 2->3 3->1
```

```
g3 <- graph( c("John", "Jim", "Jim", "Jill", "Jill", "John")) # na
med vertices

# When the edge list has vertex names, the number of nodes is not
 needed

plot(g3)
```

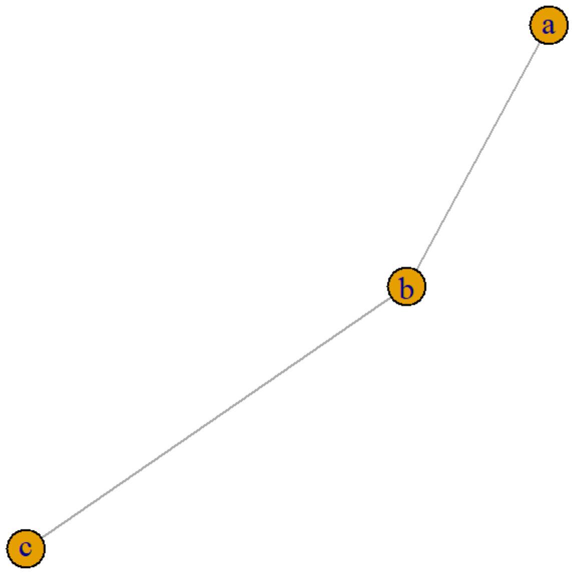



```
g4 <- graph( c("John", "Jim", "Jim", "Jack", "Jim", "Jack", "John", "John"),
             isolates=c("Jesse", "Janis", "Jennifer", "Justin") )
# In named graphs we can specify isolates by providing a list of their names.

set.seed(10)
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)
```

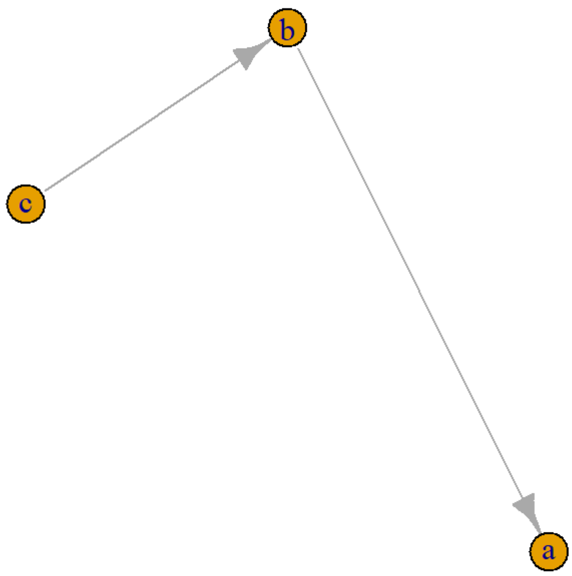


Small graphs can also be generated with a description of this kind: - for undirected tie, +- or -+ for directed ties pointing left & right, ++ for a symmetric tie, and “:” for sets of vertices.

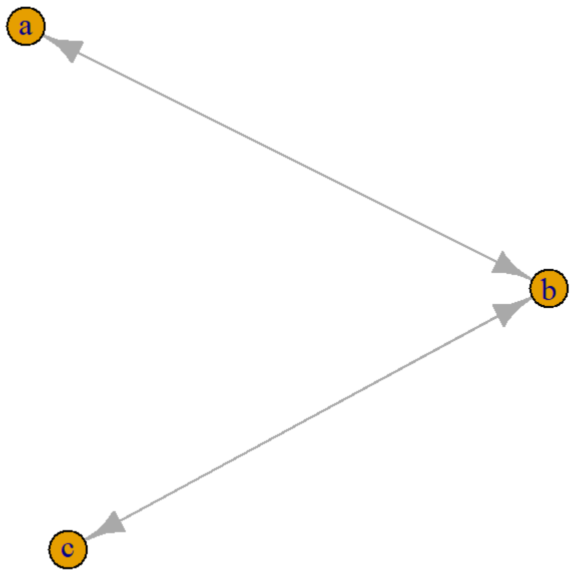
```
plot(graph_from_literal(a--b, b--c)) # the number of dashes does
n't matter
```



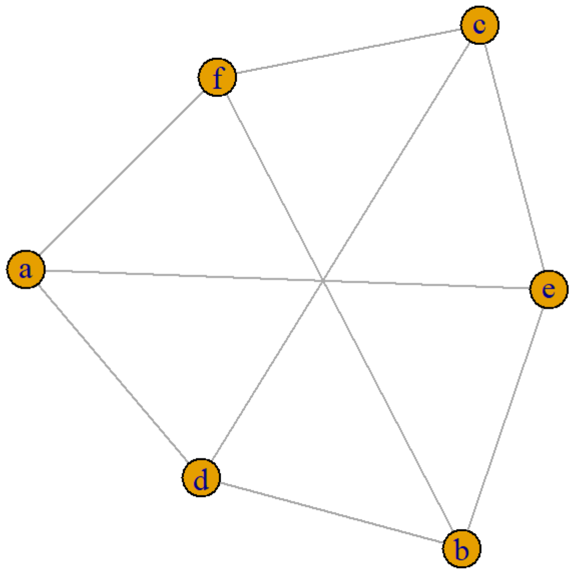

```
plot(graph_from_literal(b--+a, b+--c))
```



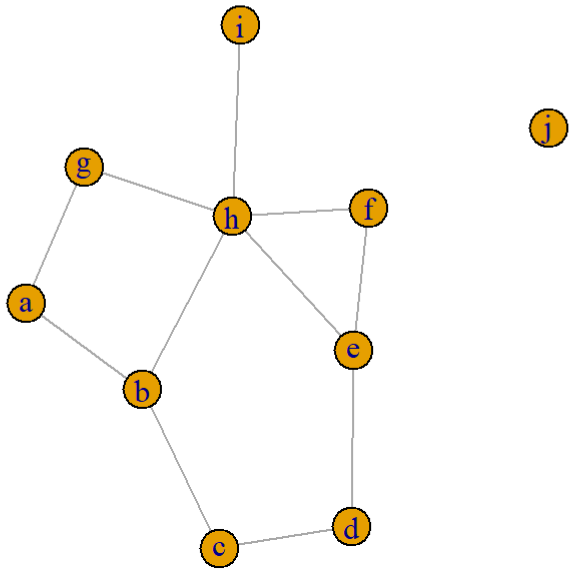
```
plot(graph_from_literal(a+-+b, b+-+c))
```




```
plot(graph_from_literal(a:b:c---d:e:f))
```




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

plot(gl)
```



Edge, vertex, and network attributes

Access vertices and edges:

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

```
## + 4/4 edges from 2ab8867 (vertex names):
## [1] John->Jim  Jim ->Jack Jim ->Jack John->John
```

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

```
## + 7/7 vertices, named, from 2ab8867:
## [1] John     Jim      Jack     Jesse    Janis    Jennifer Justi
n
```

You can also examine the network matrix directly:

```
adj <- g4[]
adjj <- as.matrix (adj)
class(adjj)
```

```
## [1] "matrix" "array"
```

```
class(adj)
```

```
## [1] "dgCMatrix"
## attr(,"package")
## [1] "Matrix"
```

```
g4[,1]
```

```
##      John       Jim      Jack     Jesse     Janis Jennifer    Justin
##         1         0         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"    "Jen
nifer" "Justin"
```

```
V(g4)$gender <- c("male", "male", "male", "male", "female", "femal
e", "male")

E(g4)$type <- "email" # Edge attribute, assign "email" to all edge
s

E(g4)$weight <- c(1,2,3,4)    # Edge weight, setting all existing
 edges to 10

E(g4)$weight
```

```
## [1] 1 2 3 4
```

```
V(g4)
```

```
## + 7/7 vertices, named, from 2ab8867:
## [1] John      Jim       Jack      Jesse     Janis     Jennifer Justi
n
```

Examine attributes:

```
edge_attr(g4)
```

```
## $type
## [1] "email" "email" "email" "email"
##
## $weight
## [1] 1 2 3 4
```

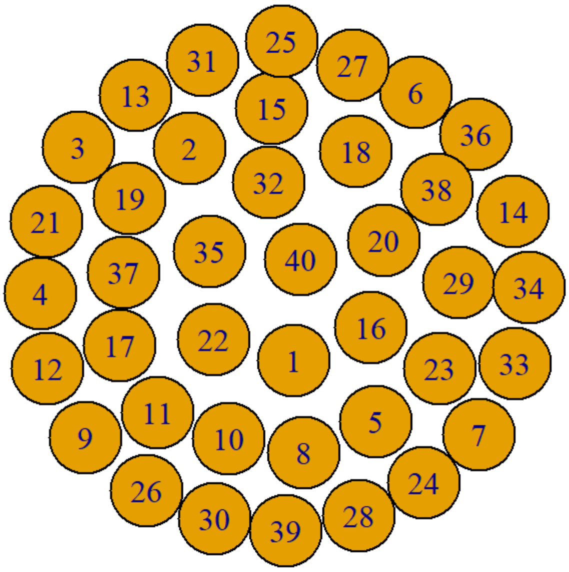
```
vertex_attr(g4)
```

```
## $name
## [1] "John"      "Jim"       "Jack"      "Jesse"     "Janis"     "Jen
nifer" "Justin"
##
## $gender
## [1] "male"    "male"    "male"    "male"    "female" "female" "mal
e"
```

Specific graphs and graph models

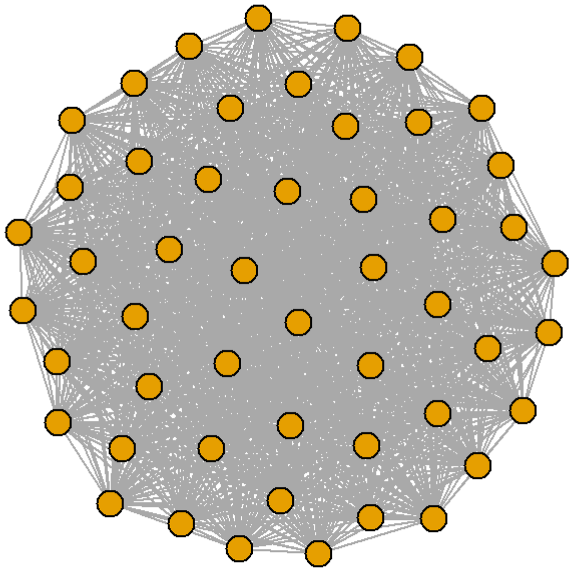
Empty graph

```
eg <- make_empty_graph(40)

plot(eg, vertex.size=30, vertex.label=1:40)
```



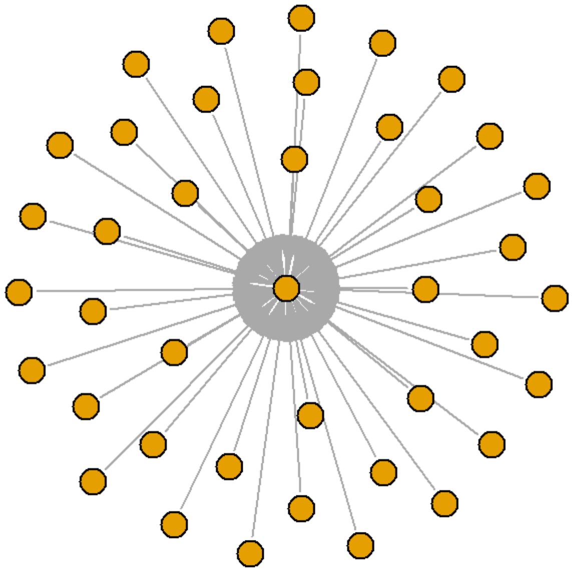
Full graph

```
fg <- make_full_graph(50)

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



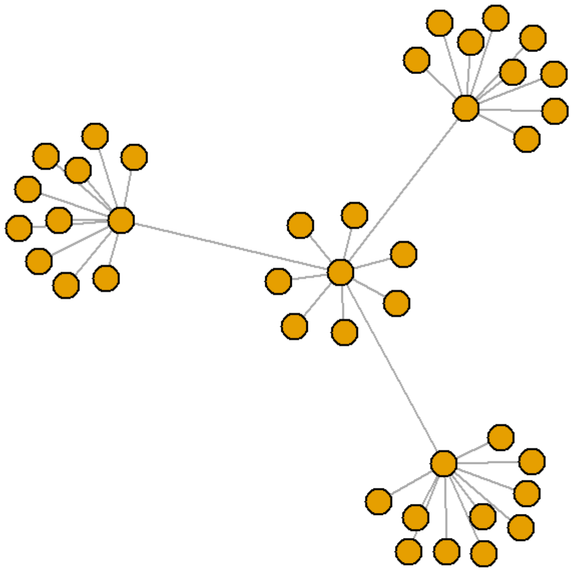
Simple star graph

```
st <- make_star(40)

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



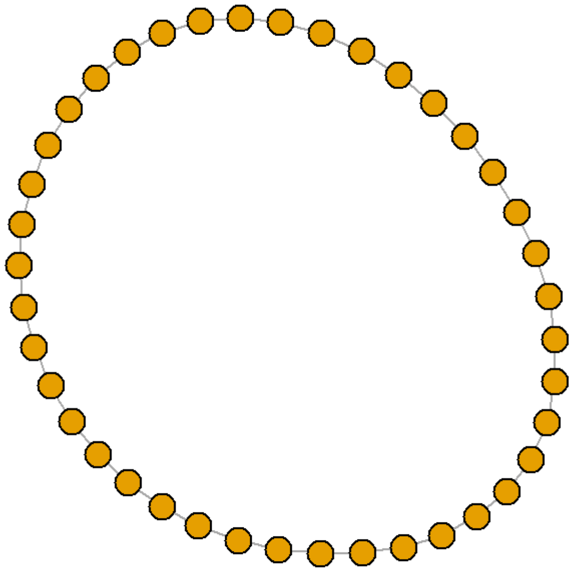
Tree graph

```
tr <- make_tree(40, children = 10, mode = "undirected")

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



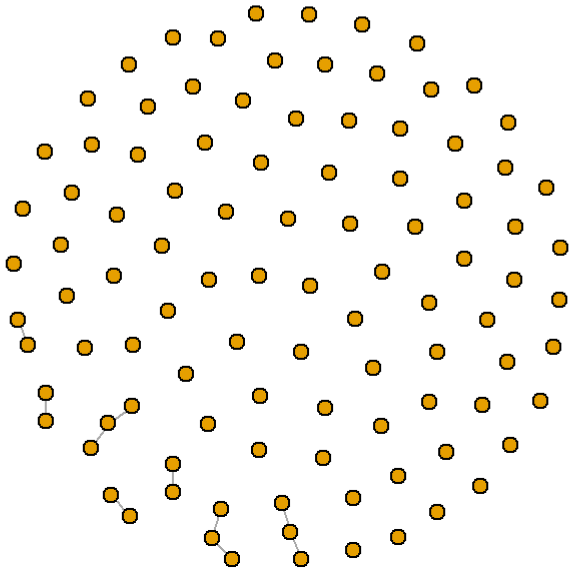
Ring graph

```
rn <- make_ring(40)

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



Erdos-Renyi random graph model

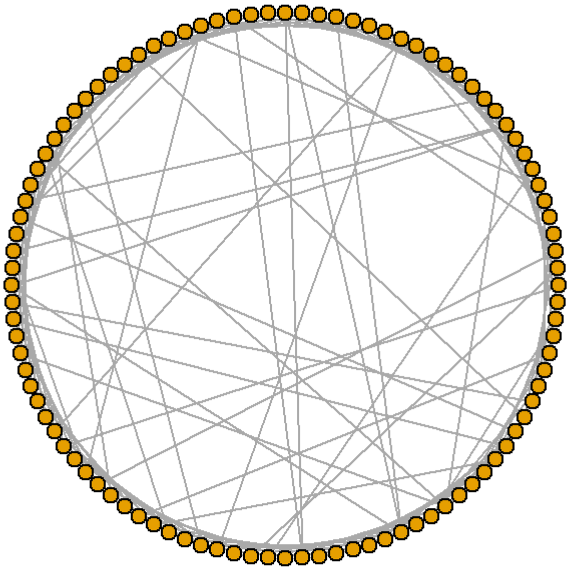
```
er <- sample_gnm(n=100, m=10)

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.

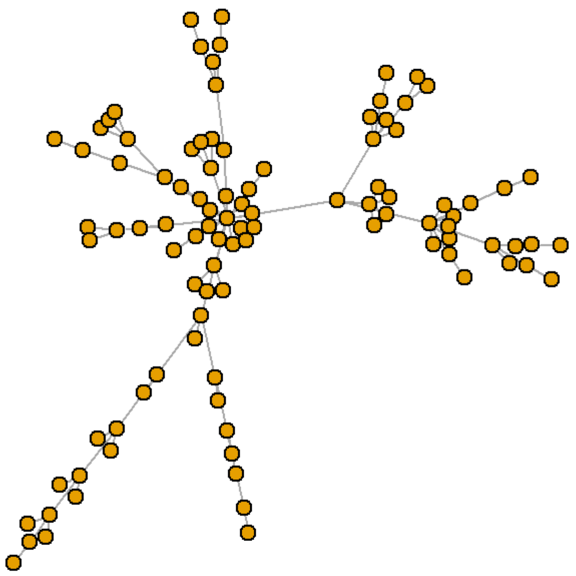
```
sw <- sample_smallworld(dim=2, size=10, nei=1, p=0.1)

plot(sw, vertex.size=6, vertex.label=NA, layout=layout_in_circle)
```



Barabasi-Albert preferential attachment model for scale-free graphs

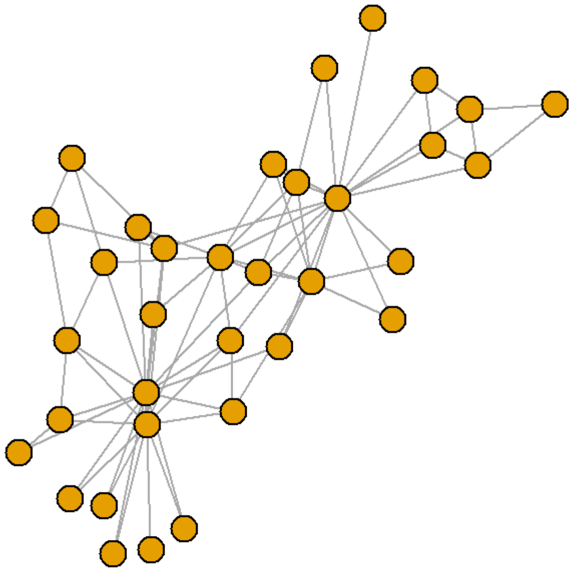
(n is number of nodes, power is the power of attachment (1 is linear); m is the number of edges added on each time step)

```
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:

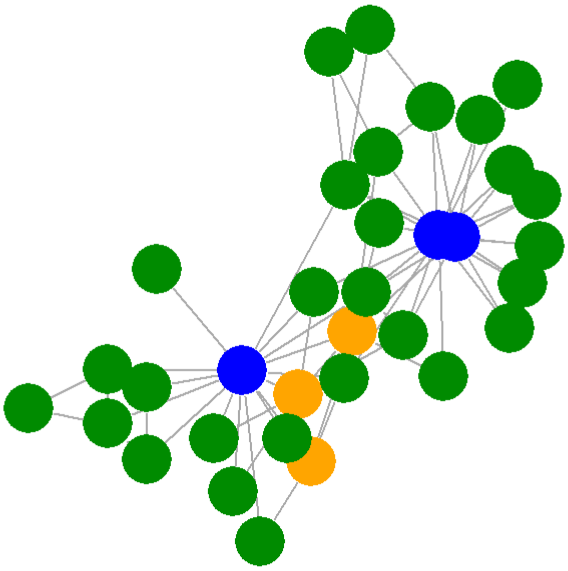
```
zach <- graph("Zachary") # the Zachary carate club

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



```
deg <- degree(zach)
deg
```

```
##  [1] 16  9 10  6  3  4  4  4  5  2  3  1  2  5  2  2  2  2  2  3  2  2  2  5  3
## [26]  3  2  4  3  4  4  6 12 17
```

```
ord <- order(deg,decreasing=T)
V(zach)[ord[1:3]]$color <- "blue"
V(zach)[ord[4:6]]$color <- "orange"
V(zach)[ord[7:34]]$color <- "green4"

plot(zach, vertex.size=20, vertex.label=NA,vertex.frame.color = NA)
```



5.3 Reading network data from files

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.

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”

```
nodes <- read.csv("Dataset1-Media-Example-NODES.csv", header=T, a
s.is=T)

links <- read.csv("Dataset1-Media-Example-EDGES.csv", header=T, a
s.is=T)
```

Examine the data:

```
head(nodes)
```

```
##    id               media media.type type.label audience.size
## 1 s01            NY Times          1  Newspaper            20
## 2 s02     Washington Post          1  Newspaper            25
## 3 s03 Wall Street Journal          1  Newspaper            30
## 4 s04           USA Today          1  Newspaper            32
## 5 s05            LA Times          1  Newspaper            20
## 6 s06       New York Post          1  Newspaper            50
```

```
head(links)
```

```
##   from  to weight      type
## 1  s01 s02     10 hyperlink
## 2  s01 s02     12 hyperlink
## 3  s01 s03     22 hyperlink
## 4  s01 s04     21 hyperlink
## 5  s04 s11     22   mention
## 6  s05 s15     21   mention
```

```
nrow(nodes); length(unique(nodes$id))
```

```
## [1] 17
```

```
## [1] 17
```

```
nrow(links); nrow(unique(links[,c("from", "to")]))
```

```
## [1] 52
```

```
## [1] 49
```

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
```

DATASET 2: matrix

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)
```

```
##    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 U18 U19
## s01   1   1   1   0   0   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   0   0
## s03   0   0   0   0   0   1   1   1   1   0   0   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   0   0
## s05   0   0   0   0   0   0   0   0   0   0   1   1   1   0   0
0   0   0   0
## s06   0   0   0   0   0   0   0   0   0   0   0   0   1   1   0
0   1   0   0
##     U20
## s01   0
## s02   1
## s03   0
## s04   0
## s05   0
## s06   0
```

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

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

dim(links2)
```

```
## [1] 10 20
```

```
dim(nodes2)
```

```
## [1] 30  5
```

5.4. Turning networks into 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 2d66513 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 2d66513 (vertex names):
##  [1] s01->s02 s01->s03 s01->s04 s01->s15 s02->s01 s02->s03 s02-
>s09 s02->s10
##  [9] s03->s01 s03->s04 s03->s05 s03->s08 s03->s10 s03->s11 s03-
>s12 s04->s03
## [17] s04->s06 s04->s11 s04->s12 s04->s17 s05->s01 s05->s02 s05-
>s09 s05->s15
## [25] s06->s06 s06->s16 s06->s17 s07->s03 s07->s08 s07->s10 s07-
>s14 s08->s03
## [33] s08->s07 s08->s09 s09->s10 s10->s03 s12->s06 s12->s13 s12-
>s14 s13->s12
## [41] s13->s17 s14->s11 s14->s13 s15->s01 s15->s04 s15->s06 s16-
>s06 s16->s17
## [49] s17->s04
```

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

```
E(net)       # The edges of the "net" object
```

```
## + 49/49 edges from 2d66513 (vertex names):
##  [1] s01->s02 s01->s03 s01->s04 s01->s15 s02->s01 s02->s03 s02->s09 s02->s10
##  [9] s03->s01 s03->s04 s03->s05 s03->s08 s03->s10 s03->s11 s03->s12 s04->s03
## [17] s04->s06 s04->s11 s04->s12 s04->s17 s05->s01 s05->s02 s05->s09 s05->s15
## [25] s06->s06 s06->s16 s06->s17 s07->s03 s07->s08 s07->s10 s07->s14 s08->s03
## [33] s08->s07 s08->s09 s09->s10 s10->s03 s12->s06 s12->s13 s12->s14 s13->s12
## [41] s13->s17 s14->s11 s14->s13 s15->s01 s15->s04 s15->s06 s16->s06 s16->s17
## [49] s17->s04
```

```
V(net)       # The vertices of the "net" object
```

```
## + 17/17 vertices, named, from 2d66513:
##  [1] s01 s02 s03 s04 s05 s06 s07 s08 s09 s10 s11 s12 s13 s14 s15 s16 s17
```

```
E(net)$type  # Edge attribute "type"
```

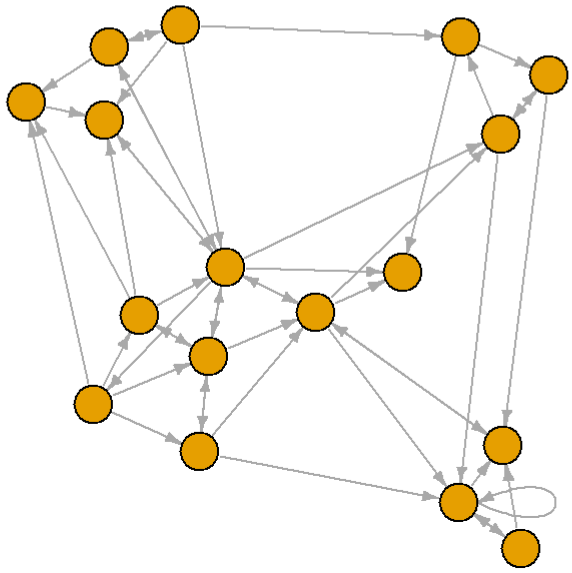
```
##  [1] "hyperlink" "hyperlink" "hyperlink" "mention"   "hyperlin
k" "hyperlink"
##  [7] "hyperlink" "hyperlink" "hyperlink" "hyperlink" "hyperlin
k" "hyperlink"
## [13] "mention"   "hyperlink" "hyperlink" "hyperlink" "mention"
"mention"
## [19] "hyperlink" "mention"   "mention"   "hyperlink" "hyperlin
k" "mention"
## [25] "hyperlink" "hyperlink" "mention"   "mention"   "mention"
"hyperlink"
## [31] "mention"   "hyperlink" "mention"   "mention"   "mention"
"hyperlink"
## [37] "mention"   "hyperlink" "mention"   "hyperlink" "mention"
"mention"
## [43] "mention"   "hyperlink" "hyperlink" "hyperlink" "hyperlin
k" "mention"
## [49] "hyperlink"
```

```
V(net)$media # Vertex attribute "media"
```

```
##  [1] "NY Times"              "Washington Post"     "Wall Street J
ournal"
##  [4] "USA Today"             "LA Times"            "New York Pos
t"
##  [7] "CNN"                   "MSNBC"               "FOX News"
## [10] "ABC"                   "BBC"                 "Yahoo News"
## [13] "Google News"           "Reuters.com"         "NYTimes.com"
## [16] "WashingtonPost.com"    "AOL.com"
```

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)
```

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 U18 U19
## s01   1   1   1   0   0   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   0   0
## s03   0   0   0   0   0   1   1   1   1   0   0   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   0   0
## s05   0   0   0   0   0   0   0   0   0   0   1   1   1   0   0
0   0   0   0
## s06   0   0   0   0   0   0   0   0   0   0   0   0   1   1   0
0   1   0   0
##     U20
## s01   0
## s02   1
## s03   0
## s04   0
## s05   0
## s06   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))
```

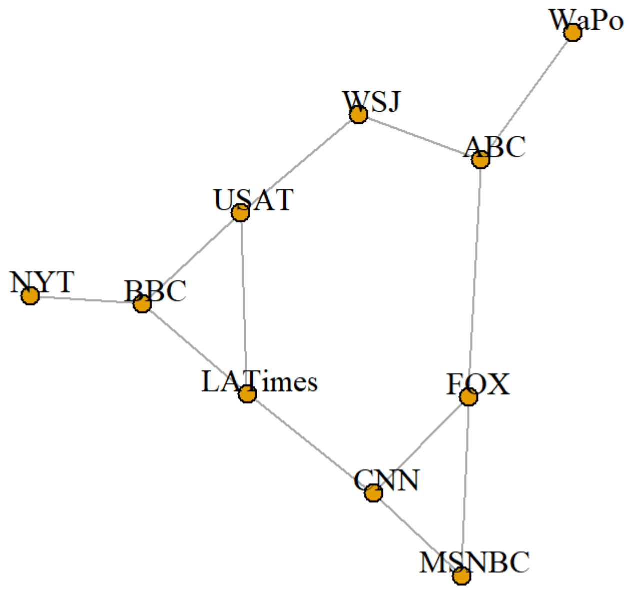
```
##     s01 s02 s03 s04 s05 s06 s07 s08 s09 s10
## s01   3   0   0   0   0   0   0   0   0   1
## s02   0   3   0   0   0   0   0   0   1   0
## s03   0   0   4   1   0   0   0   0   1   0
## s04   0   0   1   3   1   0   0   0   0   1
## s05   0   0   0   1   3   1   0   0   0   1
## s06   0   0   0   0   1   3   1   1   0   0
## s07   0   0   0   0   0   1   3   1   0   0
## s08   0   0   0   0   0   1   1   4   1   0
## s09   0   1   1   0   0   0   0   1   3   0
## s10   1   0   0   1   1   0   0   0   0   2
```

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

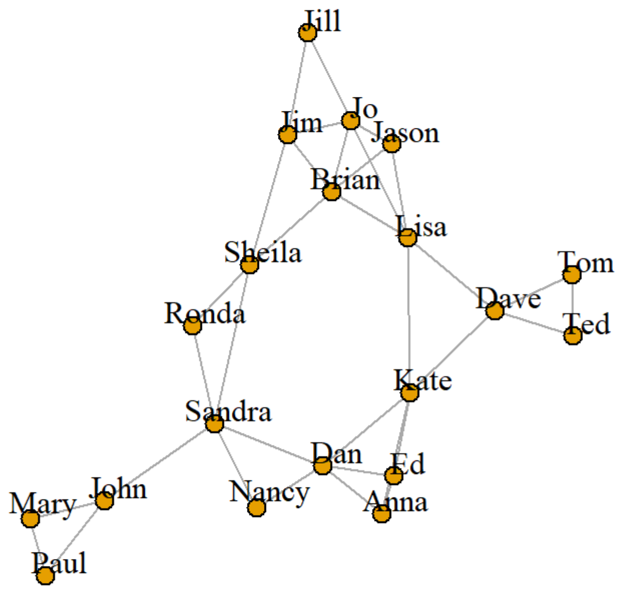
```
##     U01 U02 U03 U04 U05 U06 U07 U08 U09 U10 U11 U12 U13 U14 U15
U16 U17 U18 U19
## U01   2   1   1   0   0   0   0   0   0   0   1   0   0   0   0
0   0   0   0
## U02   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
## U03   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
## U04   0   0   0   1   1   0   0   0   0   0   0   0   0   0   0
0   0   0   0
## U05   0   0   0   1   1   0   0   0   0   0   0   0   0   0   0
0   0   0   0
## U06   0   0   0   0   0   2   1   1   1   0   0   0   0   0   0
0   0   0   1
## U07   0   0   0   0   0   1   1   1   1   0   0   0   0   0   0
0   0   0   0
## U08   0   0   0   0   0   1   1   1   1   0   0   0   0   0   0
0   0   0   0
## U09   0   0   0   0   0   1   1   1   2   1   1   0   0   0   0
0   0   0   0
## U10   0   0   0   0   0   0   0   0   1   1   1   0   0   0   0
0   0   0   0
## U11   1   0   0   0   0   0   0   0   1   1   3   1   1   0   0
0   0   0   0
## U12   0   0   0   0   0   0   0   0   0   0   1   1   1   0   0
0   0   0   0
## U13   0   0   0   0   0   0   0   0   0   0   1   1   2   1   0
0   1   0   0
## U14   0   0   0   0   0   0   0   0   0   0   0   0   1   2   1
1   1   0   0
## U15   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1
1   0   0   0
## U16   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1
2   1   1   1
## U17   0   0   0   0   0   0   0   0   0   0   0   0   1   1   0
1   2   1   1
```

```
## U18   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
1   1   1   1
## U19   0   0   0   0   0   1   0   0   0   0   0   0   0   0   0
1   1   1   2
## U20   0   0   0   1   1   1   0   0   0   0   0   0   0   0   0
0   0   0   1
##       U20
## U01   0
## U02   0
## U03   0
## U04   1
## U05   1
## U06   1
## U07   0
## U08   0
## U09   0
## U10   0
## U11   0
## U12   0
## U13   0
## U14   0
## U15   0
## U16   0
## U17   0
## U18   0
## U19   1
## U20   2
```

```
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)])
```



5.5. Plotting networks with igraph

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

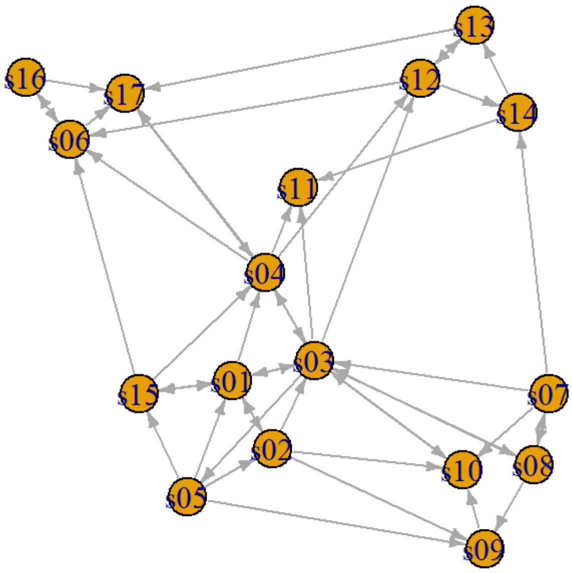
`frame` if TRUE, the plot will be framed

`main` If set, adds a title to the plot

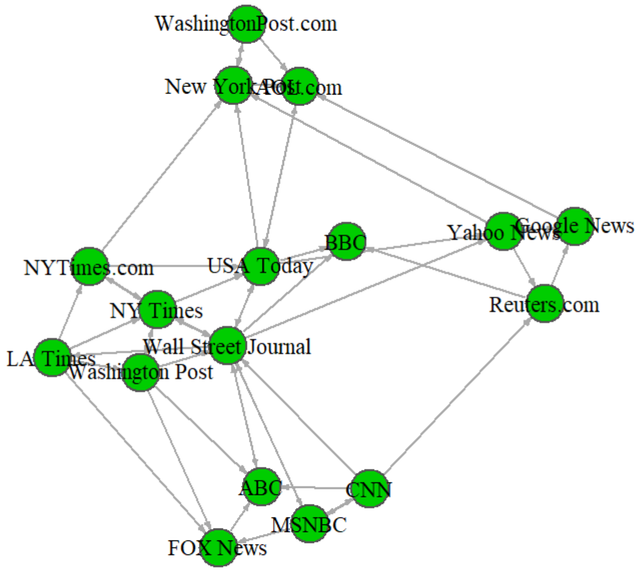
`sub` If set, adds a subtitle to the plot

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=0)
```



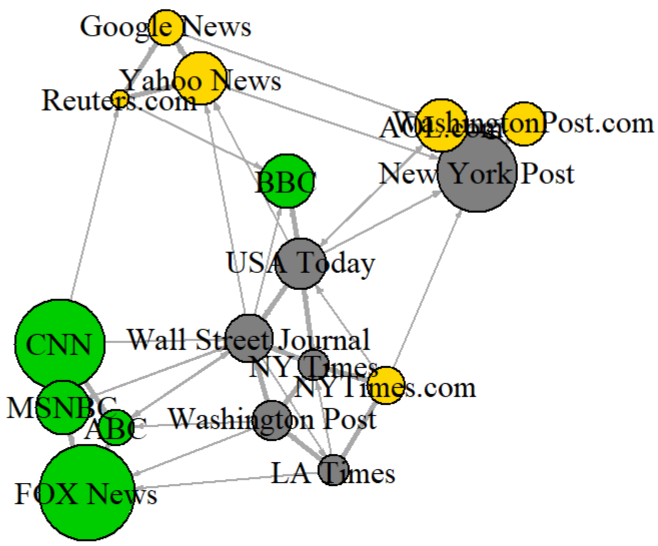

```
# Set edge color to gray, and the node color to orange.

# Replace the vertex label with the node names stored in "media"

plot(net, edge.arrow.size=.2, edge.curved=0,

     vertex.color="green3", 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.

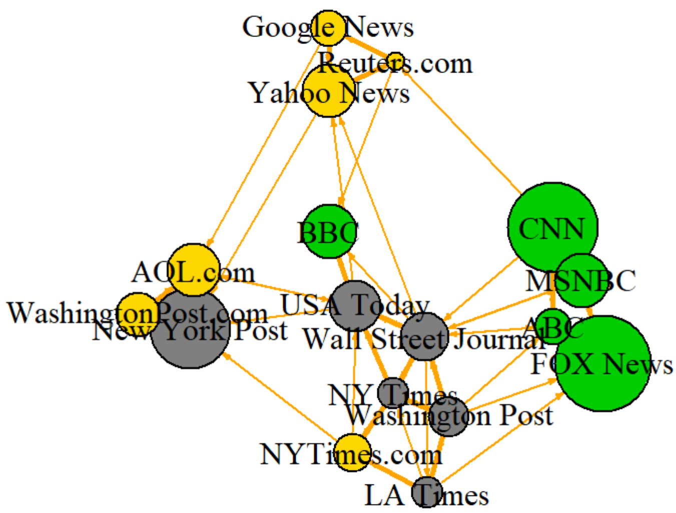
```
# Generate colors based on media type:

colrs <- c("gray50", "green3", "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

V(net)$label=V(net)$media

# 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)
```



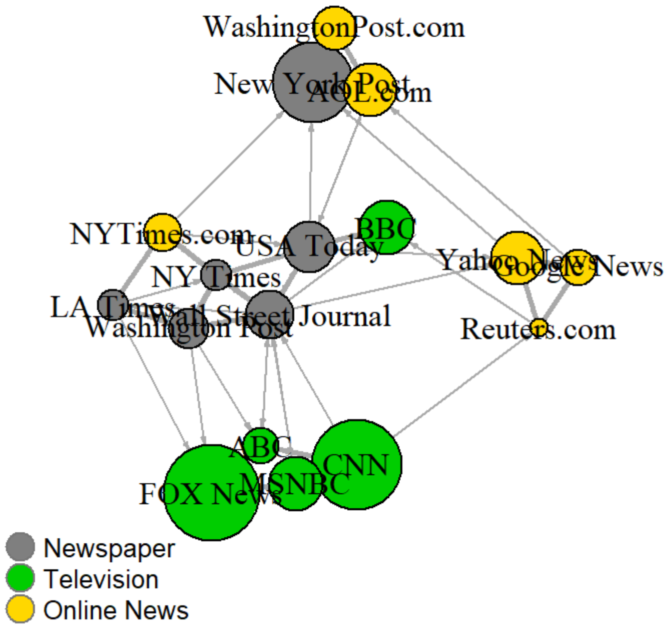
We can also override the attributes explicitly in the plot:

```
plot(net, edge.color="orange", vertex.color= colrs[V(net)$media.ty
pe])
```



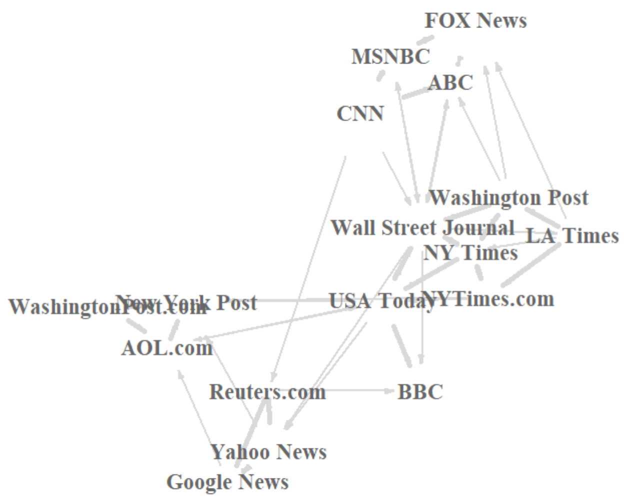
It helps to add a legend explaining the meaning of the colors we used:

```
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", nco
l=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")
```

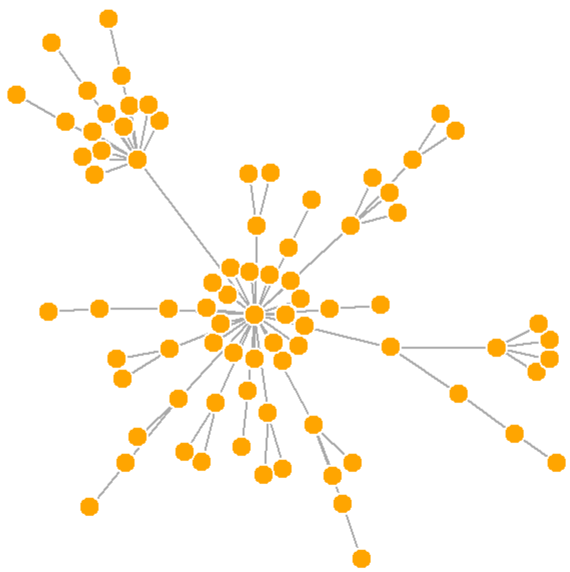


Network layouts

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

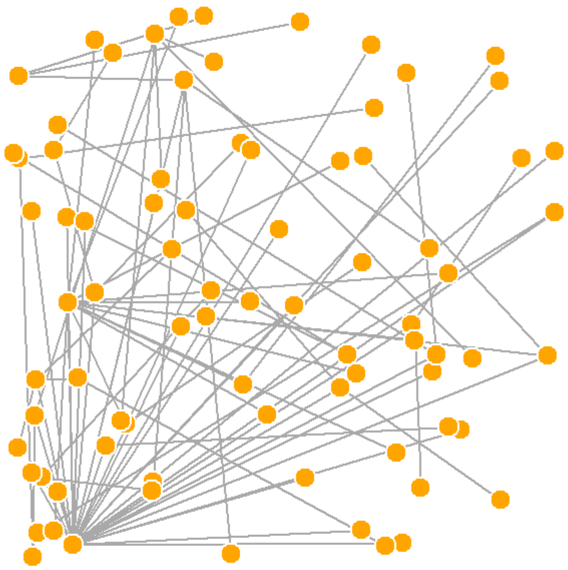
For the purposes of exploring layouts, we will generate a slightly larger 80-node graph. We use the `sample_pa()` function which generates a simple graph starting from one node and adding more nodes and links based on a preset level of preferential attachment (Barabasi-Albert model).

```
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)
```

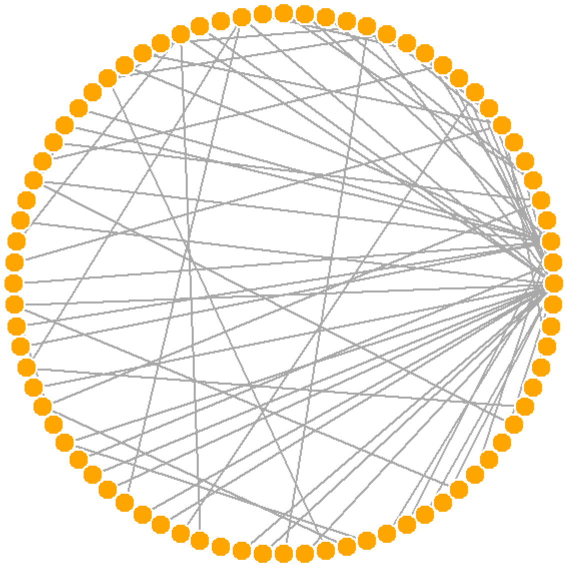


You can set the layout in the plot function:

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



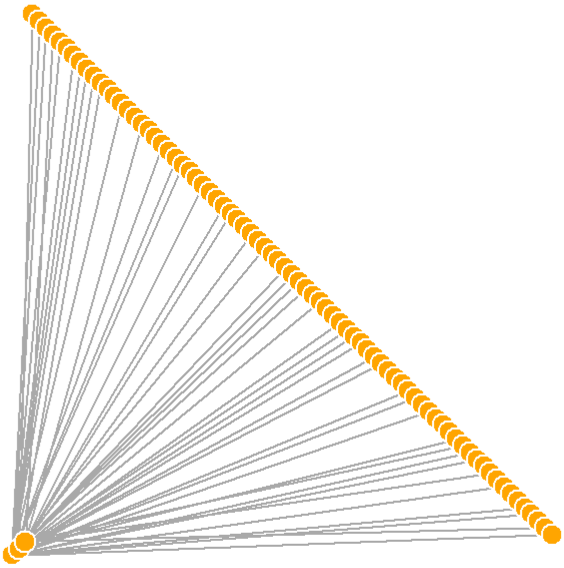
```
l <- layout_in_circle(net.bg)

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



Or you can calculate the vertex coordinates in advance:

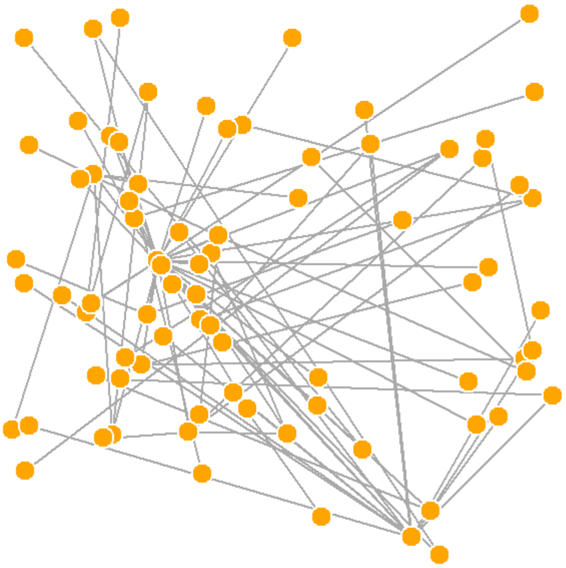
`l` is simply a matrix of x, y coordinates (N x 2) for the N nodes in the graph. You can easily generate your own:

```
l <- cbind(1:vcount(net.bg), c(1,2,3, vcount(net.bg):4))

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

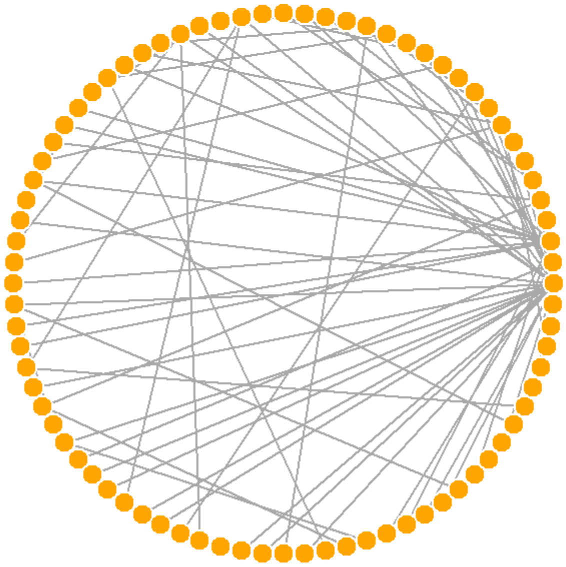


This layout is just an example and not very helpful - thankfully igraph has a number of built-in layouts, including:

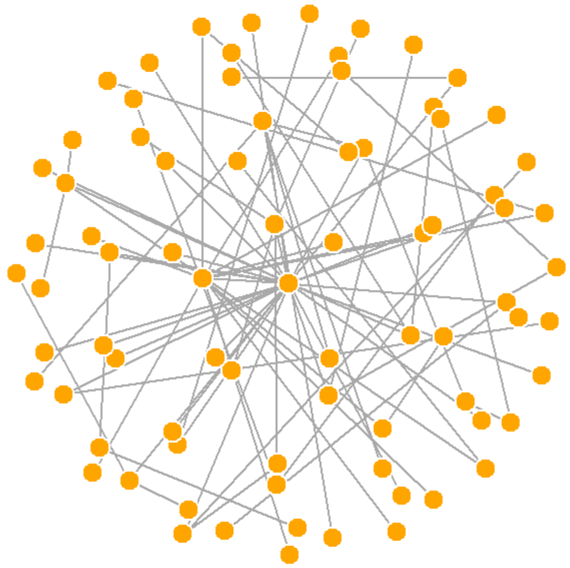
```
# Randomly placed vertices

l <- layout_randomly(net.bg)

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



```
# Circle layout

l <- layout_in_circle(net.bg)

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



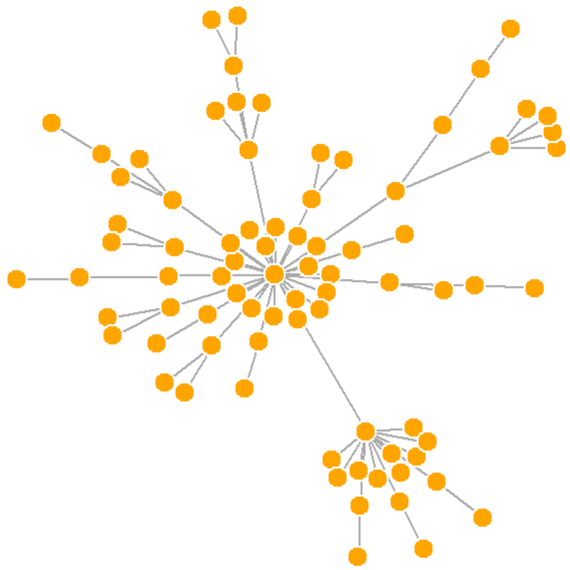
```
# 3D sphere layout

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.

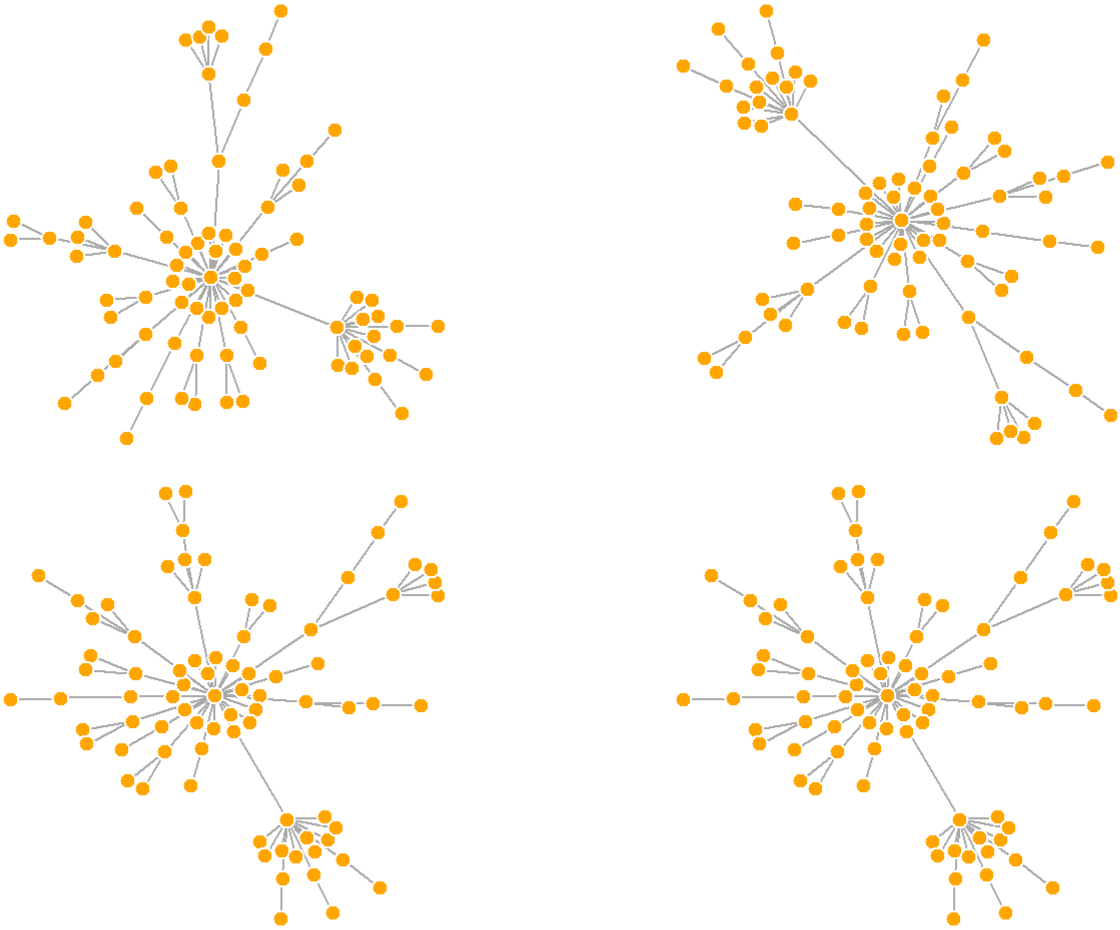
Force-directed layouts try to get a nice-looking graph where edges are similar in length and cross each other as little as possible. They simulate the graph as a physical system. Nodes are electrically charged particles that repulse each other when they get too close. The edges act as springs that attract 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()
```

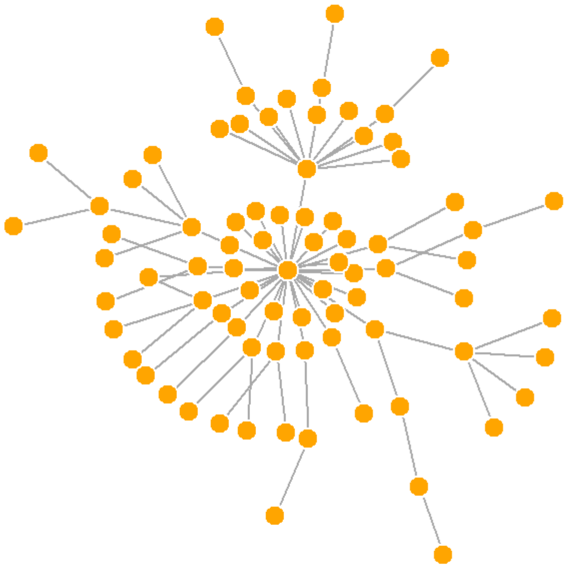
```
## null device
##           1
```

```
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*0.1)

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)
```

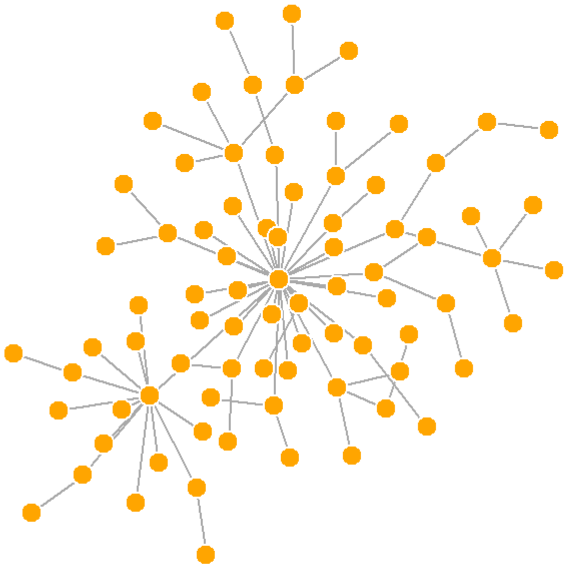
Another popular force-directed algorithm that produces nice results for connected graphs is Kamada Kawai. Like Fruchterman Reingold, it attempts to minimize the energy in a spring system.

```
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.

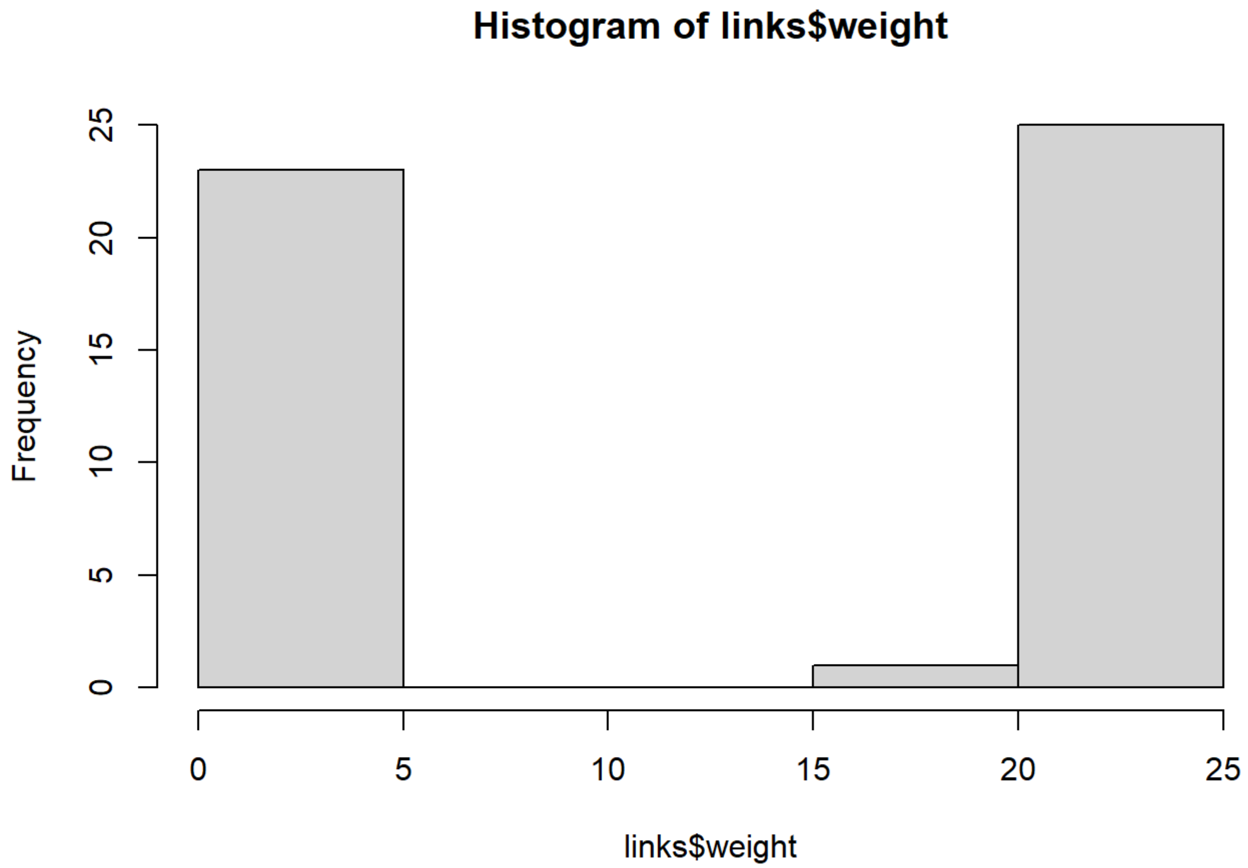
```
plot(net.bg, layout=layout_with_lgl)
```



Network layouts (detailed)

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)
```

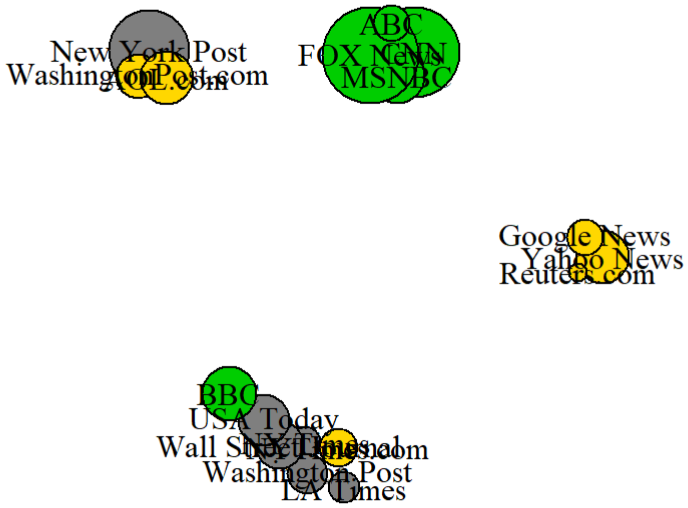
```
## [1] 12.40816
```

```
sd(links$weight)
```

```
## [1] 9.905635
```

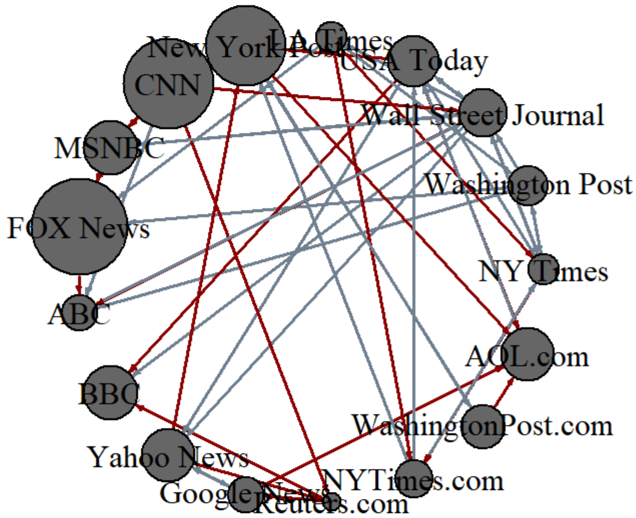
There are more sophisticated ways to extract the key edges, but for the purposes of this exercise we'll only keep ones that have weight higher than the mean for the network. In igraph, we can delete edges using `delete_edges(net, edges)`:

```
cut.off <- mean(links$weight)

net.sp <- delete_edges(net, E(net)[weight<cut.off])

plot(net.sp)
```



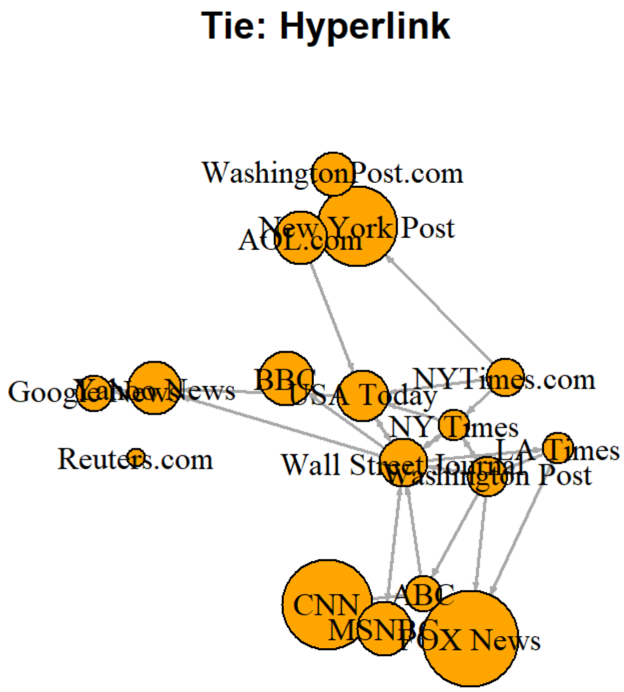
Another way to think about this is to plot the two tie types (hyperlink & mention) separately.

```
E(net)$width <- 1.5

plot(net, edge.color=c("dark red", "slategrey")[(E(net)$type=="hyperlink")+1],

      vertex.color="gray40", layout=layout.circle)
```

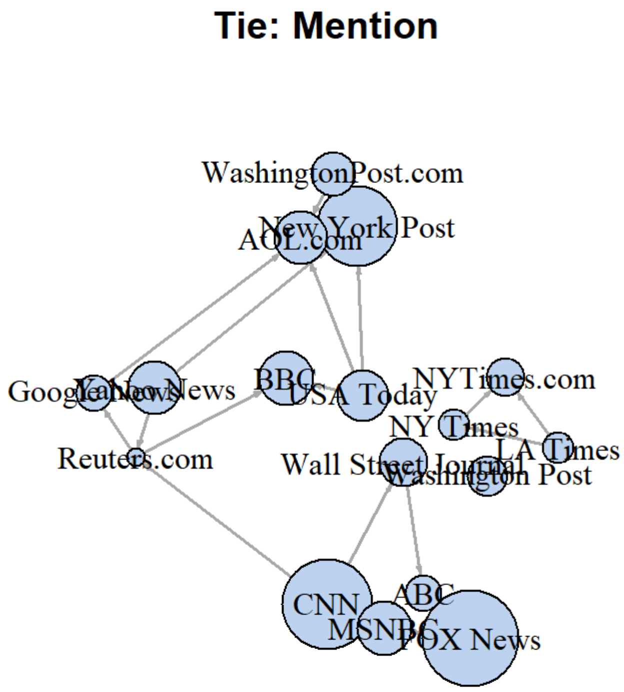



```
net.m <- net - E(net)[E(net)$type=="hyperlink"] # another way to d
elete edges

net.h <- net - E(net)[E(net)$type=="mention"]



# Plot the two links separately:

par(mfrow=c(1,2))

plot(net.h, vertex.color="orange", main="Tie: Hyperlink")

plot(net.m, vertex.color="lightsteelblue2", main="Tie: Mention")
```


```
# Make sure the nodes stay in place in both plots:

l <- layout_with_fr(net)

plot(net.h, vertex.color="orange", layout=l, main="Tie: Hyperlink"
)
```




```
plot(net.m, vertex.color="lightsteelblue2", layout=l, main="Tie: M
ention")
```

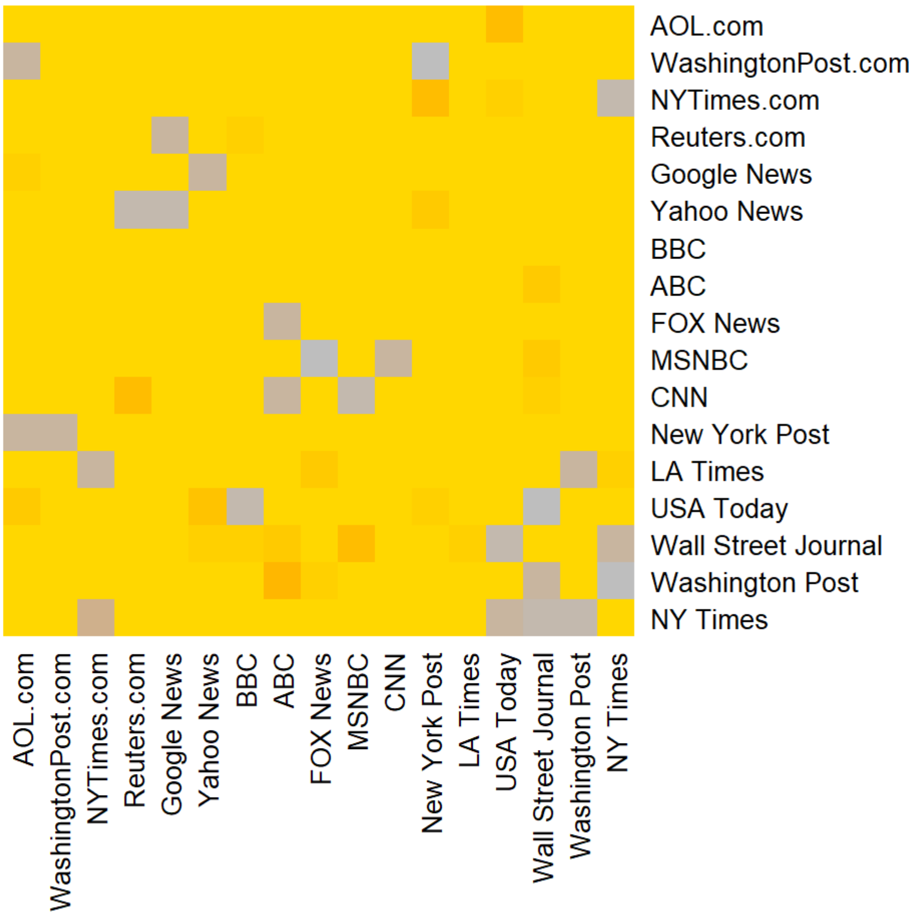


Other ways to represent a network

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:

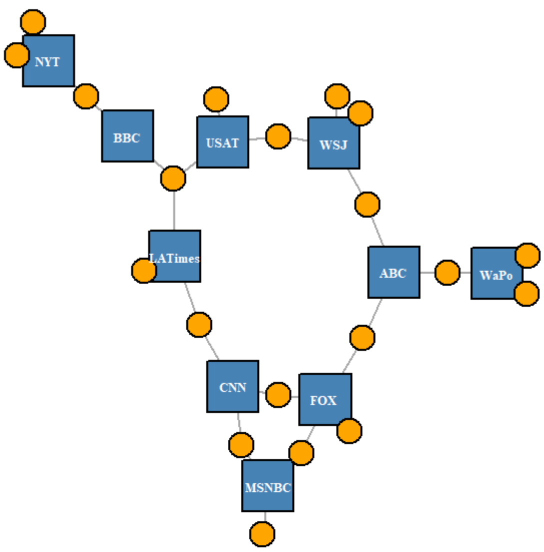
```
netm <- get.adjacency(net, attr="weight", sparse=F)

colnames(netm) <- V(net)$media

rownames(netm) <- V(net)$media



palf <- colorRampPalette(c("gold", "dark orange","gray"))

heatmap(netm[,17:1], Rowv = NA, Colv = NA, col = palf(100),

        scale="none", margins=c(10,10) )
```



Plotting two-mode networks with igraph

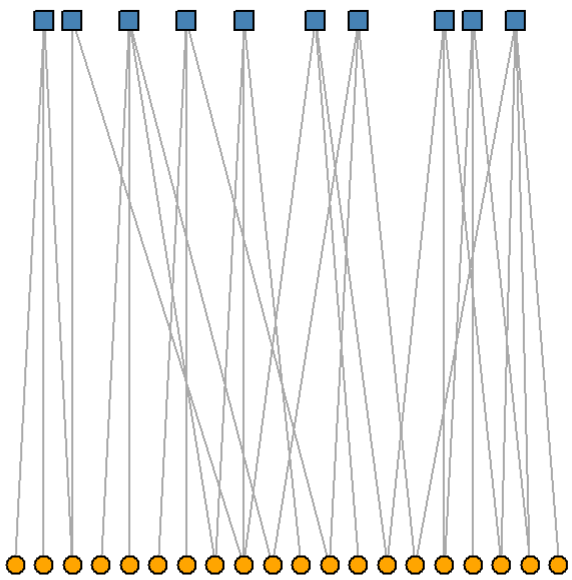
As with one-mode networks, we can modify the network object to include the visual properties that will be used by default when plotting the network. Notice that this time we will also change the shape of the nodes - media outlets will be squares, and their users will be circles.

```
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)$typ
e)*10)
```



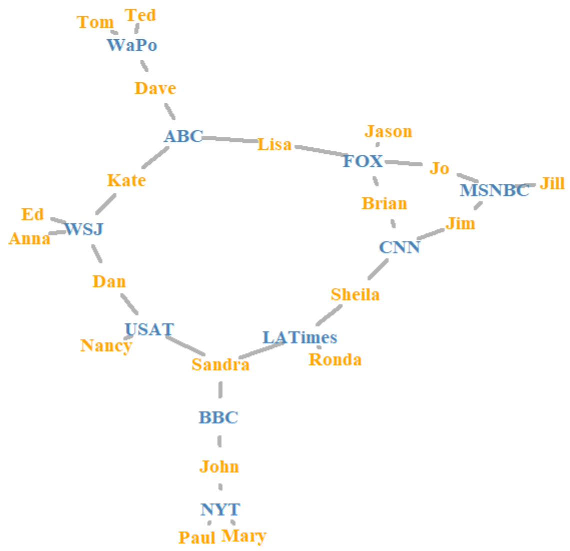
Igraph also has a special layout for bipartite networks (though it doesn't always work great, and you might be better off generating your own two-mode layout).

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



Using text as nodes may be helpful at times:

```
plot(net2, vertex.shape="none", vertex.label=nodes2$media,

     vertex.label.color=V(net2)$color, vertex.label.font=2.5,

     vertex.label.cex=.6, edge.color="gray70",  edge.width=2)
```



5.6. Network and node descriptives

Density

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

```
edge_density(net, loops=F)
```

```
## [1] 0.1764706
```

```
ecount(net)/(vcount(net)*(vcount(net)-1)) #for a directed network
```

```
## [1] 0.1764706
```

Reciprocity

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

```
reciprocity(net)
```

```
## [1] 0.4166667
```

```
dyad_census(net) # Mutual, asymmetric, and nyll node pairs
```

```
## $mut
## [1] 10
##
## $asym
## [1] 28
##
## $null
## [1] 98
```

```
2*dyad_census(net)$mut/ecount(net) # Calculating reciprocity
```

```
## [1] 0.4166667
```

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 undirecte
d network
```

```
## [1] 0.372549
```

```
transitivity(as.undirected(net, mode="collapse")) # same as above
```

```
## [1] 0.372549
```

```
transitivity(net, type="local")
```

```
##  [1] 0.2142857 0.4000000 0.1153846 0.1944444 0.5000000 0.266666
7 0.2000000
##  [8] 0.1000000 0.3333333 0.3000000 0.3333333 0.2000000 0.166666
7 0.1666667
## [15] 0.3000000 0.3333333 0.2000000
```

```
triad_census(net) # for directed networks
```

```
##  [1] 244 241  80  13  11  27  15  22   4   1   8   4   4   3
3   0
```

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.

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 2d8ea53:
## [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)
```

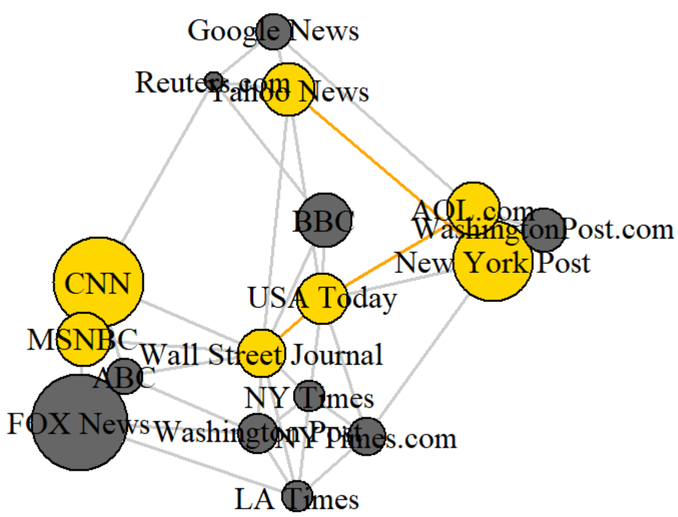
```
## [1] "igraph.vs"
```

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

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

Color nodes along the diameter:

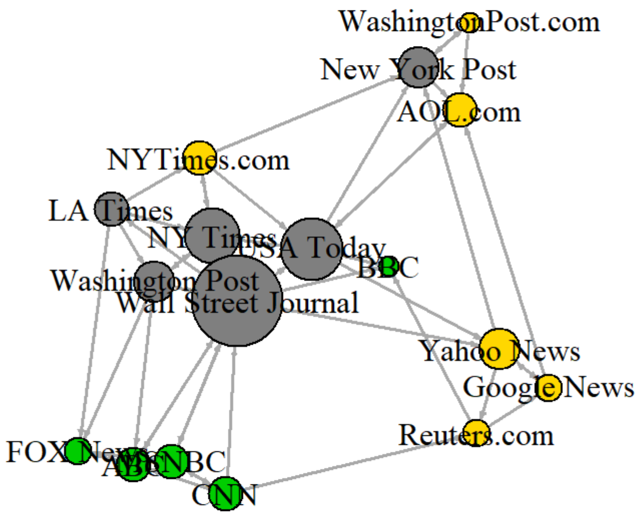
```
vcol <- rep("gray40", vcount(net))

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)
```



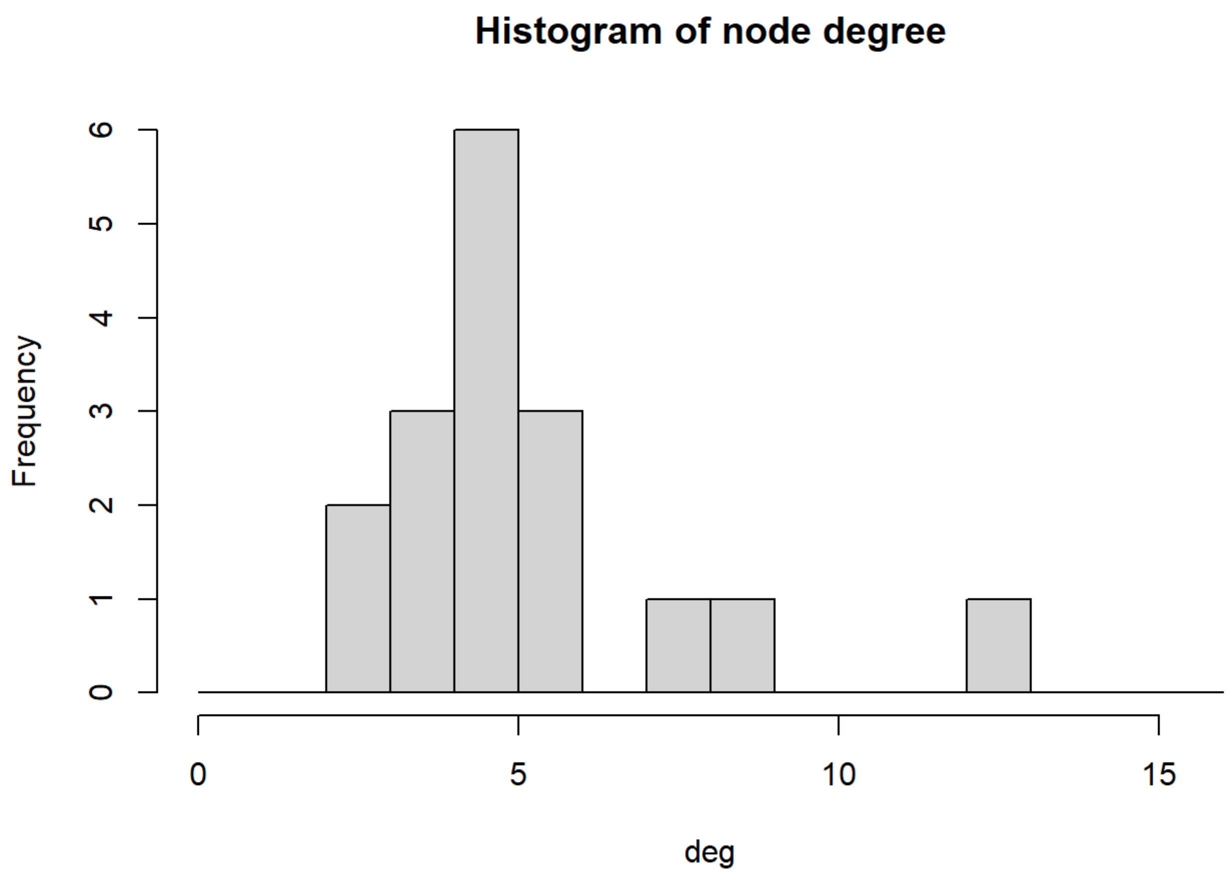
Node degrees

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)
```

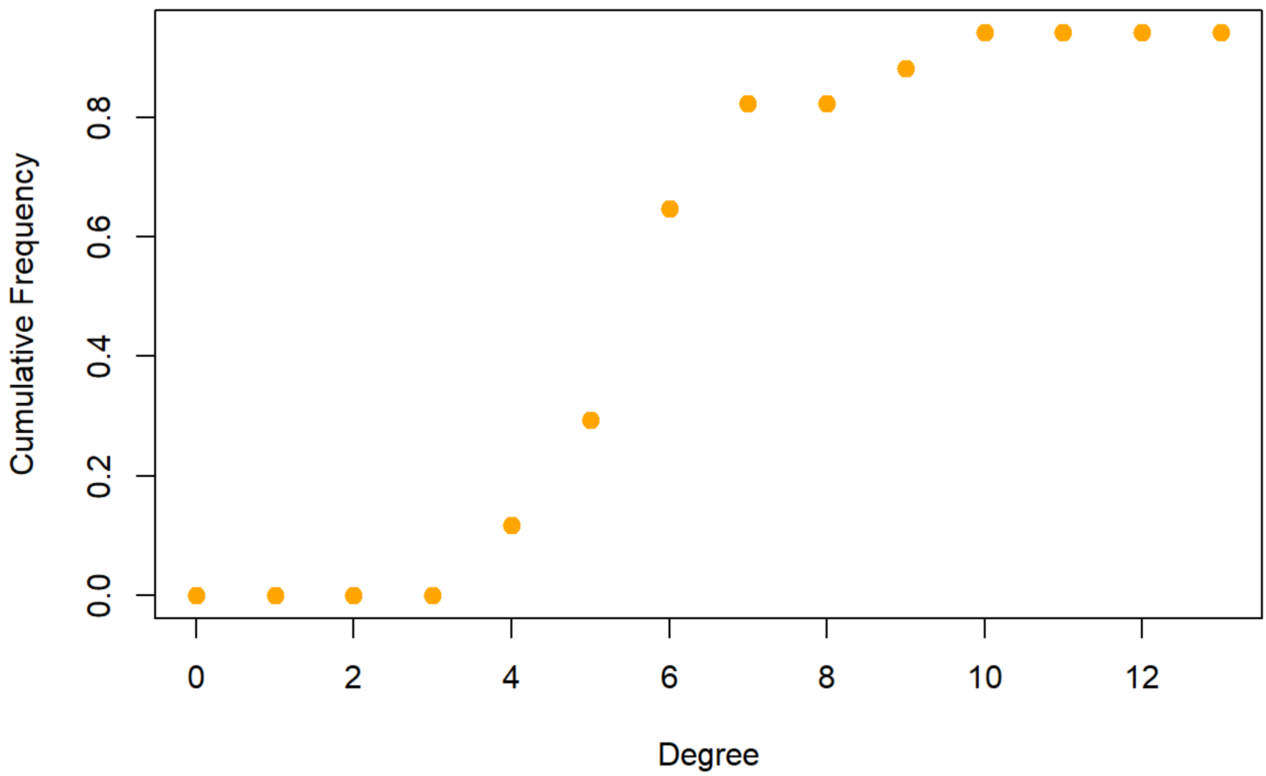



```
hist(deg, breaks=1:vcount(net)-1, main="Histogram of node degree")
```

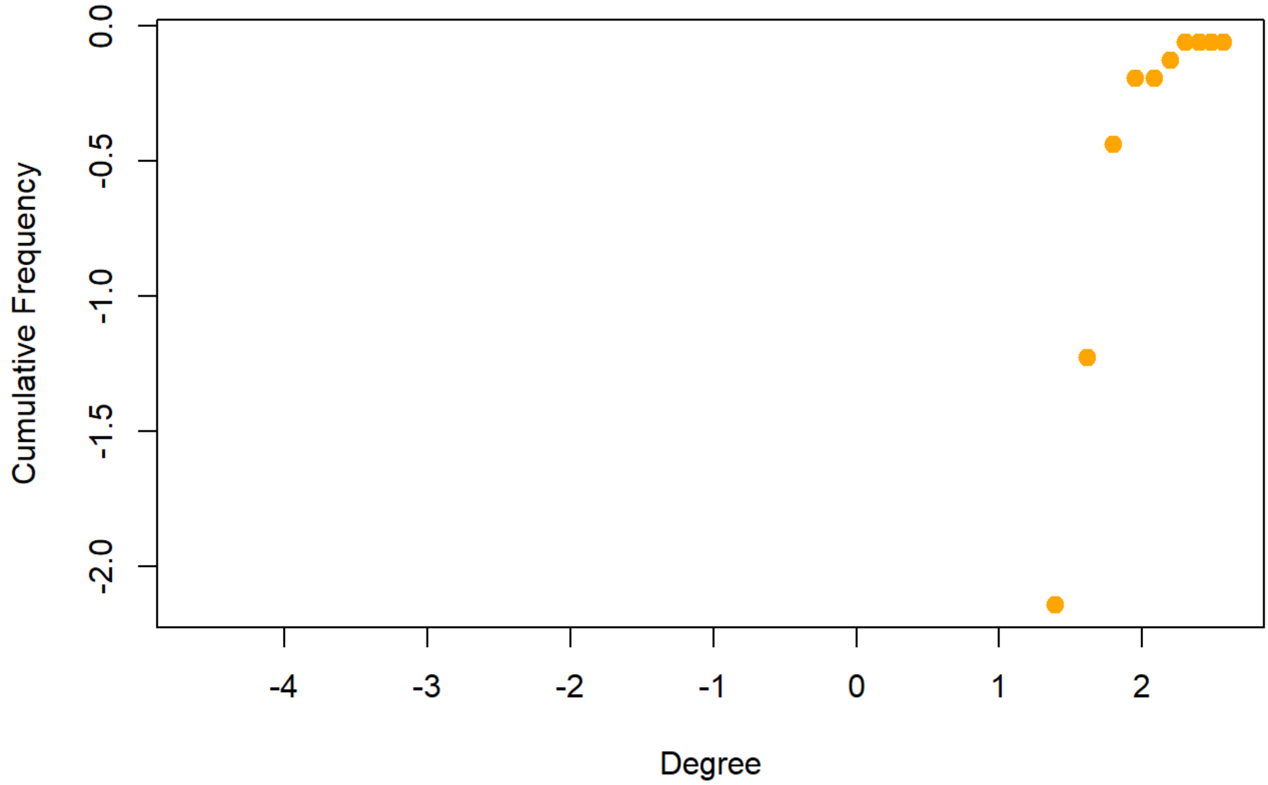


Degree distribution

```
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")
```




```
plot( x=log(0:max(deg)+0.01), y=log(1-deg.dist), pch=19, cex=1.2,
 col="orange",

      xlab="Degree", ylab="Cumulative Frequency")
```



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")
```

```
## s01 s02 s03 s04 s05 s06 s07 s08 s09 s10 s11 s12 s13 s14 s15 s16 s17
##   4   2   6   4   1   4   1   2   3   4   3   3   2   2   2   1   4
```

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

```
## $res
##  [1] 4 2 6 4 1 4 1 2 3 4 3 3 2 2 2 1 4
##
## $centralization
## [1] 0.1985294
##
## $theoretical_max
## [1] 272
```

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)
```

```
##        s01        s02        s03        s04        s05        s
06        s07
## 0.03333333 0.03030303 0.04166667 0.03846154 0.03225806 0.031250
00 0.03030303
##        s08        s09        s10        s11        s12        s
13        s14
## 0.02857143 0.02564103 0.02941176 0.03225806 0.03571429 0.027027
03 0.02941176
##        s15        s16        s17
## 0.03030303 0.02222222 0.02857143
```

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

```
## $res
##  [1] 0.5333333 0.4848485 0.6666667 0.6153846 0.5161290 0.5000000 0.4848485
##  [8] 0.4571429 0.4102564 0.4705882 0.5161290 0.5714286 0.4324324 0.4705882
## [15] 0.4848485 0.3555556 0.4571429
##
## $centralization
## [1] 0.3753596
##
## $theoretical_max
## [1] 7.741935
```

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)
```

```
## $vector
##       s01       s02       s03       s04       s05       s06 
s07       s08
## 0.6638179 0.3314674 1.0000000 0.9133129 0.3326443 0.7468249 0.1
244195 0.3740317
##       s09       s10       s11       s12       s13       s14 
s15       s16
## 0.3453324 0.5991652 0.7334202 0.7519086 0.3470857 0.2915055 0.3
314674 0.2484270
##       s17
## 0.7503292
##
## $value
## [1] 3.006215
##
## $options
## $options$bmat
## [1] "I"
##
## $options$n
## [1] 17
##
## $options$which
## [1] "LR"
##
## $options$nev
## [1] 1
##
## $options$tol
## [1] 0
##
## $options$ncv
## [1] 0
##
## $options$ldv
## [1] 0
```

```
##
## $options$ishift
## [1] 1
##
## $options$maxiter
## [1] 1000
##
## $options$nb
## [1] 1
##
## $options$mode
## [1] 1
##
## $options$start
## [1] 1
##
## $options$sigma
## [1] 0
##
## $options$sigmai
## [1] 0
##
## $options$info
## [1] 0
##
## $options$iter
## [1] 7
##
## $options$nconv
## [1] 1
##
## $options$numop
## [1] 31
##
## $options$numopb
## [1] 0
```

```
##
## $options$numreo
## [1] 18
```

```
centr_eigen(net, directed=T, normalized=T)
```

```
## $vector
##  [1] 0.6638179 0.3314674 1.0000000 0.9133129 0.3326443 0.746824
9 0.1244195
##  [8] 0.3740317 0.3453324 0.5991652 0.7334202 0.7519086 0.347085
7 0.2915055
## [15] 0.3314674 0.2484270 0.7503292
##
## $value
## [1] 3.006215
##
## $options
## $options$bmat
## [1] "I"
##
## $options$n
## [1] 17
##
## $options$which
## [1] "LR"
##
## $options$nev
## [1] 1
##
## $options$tol
## [1] 0
##
## $options$ncv
## [1] 0
##
## $options$ldv
## [1] 0
##
## $options$ishift
## [1] 1
##
## $options$maxiter
```

```
## [1] 1000
##
## $options$nb
## [1] 1
##
## $options$mode
## [1] 1
##
## $options$start
## [1] 1
##
## $options$sigma
## [1] 0
##
## $options$sigmai
## [1] 0
##
## $options$info
## [1] 0
##
## $options$iter
## [1] 7
##
## $options$nconv
## [1] 1
##
## $options$numop
## [1] 31
##
## $options$numopb
## [1] 0
##
## $options$numreo
## [1] 18
##
##
```

```
## $centralization
## [1] 0.5071775
##
## $theoretical_max
## [1] 16
```

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)
```

```
##         s01         s02         s03         s04         s05         s06
##  24.0000000   5.8333333 127.0000000  93.5000000  16.5000000  20.3333333
##         s07         s08         s09         s10         s11         s12
##   1.8333333  19.5000000   0.8333333  15.0000000   0.0000000  33.5000000
##         s13         s14         s15         s16         s17
##  20.0000000   4.0000000   5.6666667   0.0000000  58.5000000
```

```
edge_betweenness(net, directed=T, weights=NA)
```

```
##  [1] 10.833333 11.333333  8.333333  9.500000  4.000000 12.50000
0  3.000000
##  [8]  2.333333 24.000000 16.000000 31.500000 32.500000  9.50000
0  6.500000
## [15] 23.000000 65.333333 11.000000  6.500000 18.000000  8.66666
7  5.333333
## [22] 10.000000  6.000000 11.166667 15.000000 21.333333 10.00000
0  2.000000
## [29]  1.333333  4.500000 11.833333 16.833333  6.833333 16.83333
3 31.000000
## [36] 17.000000 18.000000 14.500000  7.500000 28.500000  3.00000
0 17.000000
## [43]  5.666667  9.666667  6.333333  1.000000 15.000000 74.50000
0
```

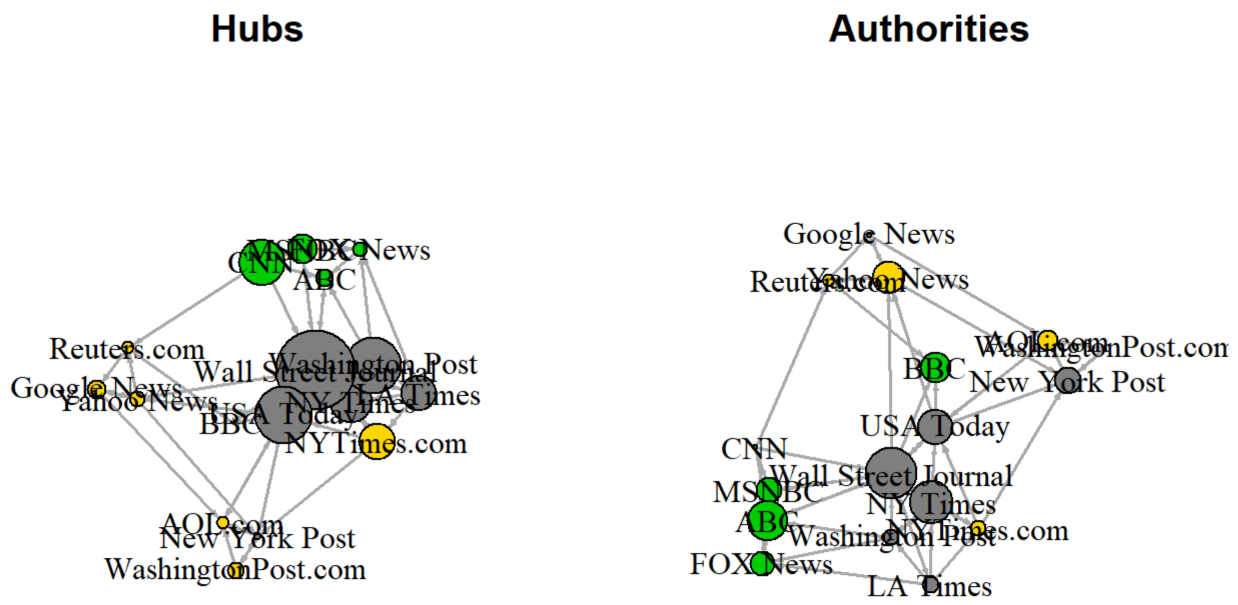
```
centr_betw(net, directed=T, normalized=T)
```

```
## $res
##  [1]  24.0000000   5.8333333 127.0000000  93.5000000  16.500000
0  20.3333333
##  [7]   1.8333333  19.5000000   0.8333333  15.0000000   0.000000
0  33.5000000
## [13]  20.0000000   4.0000000   5.6666667   0.0000000  58.500000
0
##
## $centralization
## [1] 0.4460938
##
## $theoretical_max
## [1] 3840
```

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")
```



5.7 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
```

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

```
distances(net) # with edge weights
```

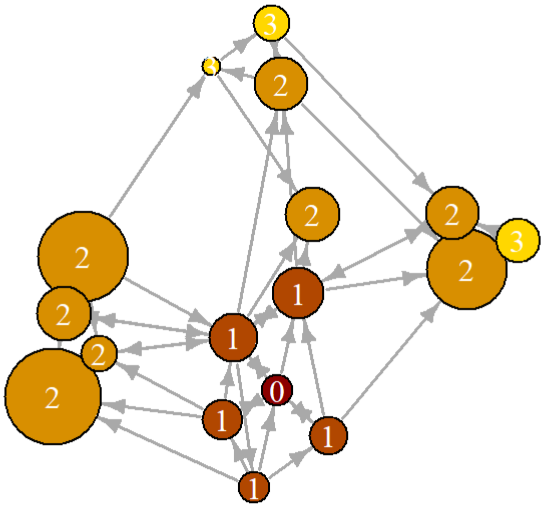
```
##     s01 s02 s03 s04 s05 s06 s07 s08 s09 s10 s11 s12 s13 s14 s15
s16 s17
## s01   0   4   2   6   1   5   3   4   3   4   3   3   9   4   7
26   8
## s02   4   0   4   8   3   7   5   6   1   5   5   5  11   6   9
28  10
## s03   2   4   0   4   1   3   1   2   3   2   1   1   7   2   5
24   6
## s04   6   8   4   0   5   1   5   6   7   6   5   3   3   6   1
22   2
## s05   1   3   1   5   0   4   2   3   2   3   2   2   8   3   6
25   7
## s06   5   7   3   1   4   0   4   5   6   5   4   2   4   5   2
21   3
## s07   3   5   1   5   2   4   0   3   4   3   2   2   8   3   6
25   7
## s08   4   6   2   6   3   5   3   0   5   4   3   3   9   4   7
26   8
## s09   3   1   3   7   2   6   4   5   0   5   4   4  10   5   8
27   9
## s10   4   5   2   6   3   5   3   4   5   0   3   3   9   4   7
26   8
## s11   3   5   1   5   2   4   2   3   4   3   0   2   8   1   6
25   7
## s12   3   5   1   3   2   2   2   3   4   3   2   0   6   3   4
23   5
## s13   9  11   7   3   8   4   8   9  10   9   8   6   0   9   4
22   1
## s14   4   6   2   6   3   5   3   4   5   4   1   3   9   0   7
26   8
## s15   7   9   5   1   6   2   6   7   8   7   6   4   4   7   0
23   3
## s16  26  28  24  22  25  21  25  26  27  26  25  23  22  26  23
0  21
## s17   8  10   6   2   7   3   7   8   9   8   7   5   1   8   3
21   0
```

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

```
##       s01 s02 s03 s04 s05 s06 s07 s08 s09 s10 s11 s12 s13 s14 s15
s16 s17
## s01    0   1   1   1   1   2   2   2   2   2   2   2   3   3   1
3   2
## s02    1   0   1   2   1   3   2   2   1   1   2   2   3   3   2
4   3
## s03    1   1   0   1   1   2   1   1   2   1   1   1   2   2   2
3   2
## s04    1   2   1   0   2   1   2   2   3   2   1   1   2   2   1
2   1
## s05    1   1   1   2   0   2   2   2   1   2   2   2   3   3   1
3   3
## s06    2   3   2   1   2   0   3   3   3   3   2   1   2   2   1
1   1
## s07    2   2   1   2   2   3   0   1   2   1   2   2   2   1   3
4   3
## s08    2   2   1   2   2   3   1   0   1   2   2   2   3   2   3
4   3
## s09    2   1   2   3   1   3   2   1   0   1   3   3   4   3   2
4   4
## s10    2   1   1   2   2   3   1   2   1   0   2   2   3   2   3
4   3
## s11    2   2   1   1   2   2   2   2   3   2   0   2   2   1   2
3   2
## s12    2   2   1   1   2   1   2   2   3   2   2   0   1   1   2
2   2
## s13    3   3   2   2   3   2   2   3   4   3   2   1   0   1   3
2   1
## s14    3   3   2   2   3   2   1   2   3   2   1   1   1   0   3
3   2
## s15    1   2   2   1   1   1   3   3   2   3   2   2   3   3   0
2   2
## s16    3   4   3   2   3   1   4   4   4   4   3   2   2   3   2
0   1
## s17    2   3   2   1   3   1   3   3   4   3   2   2   1   2   2
1   0
```

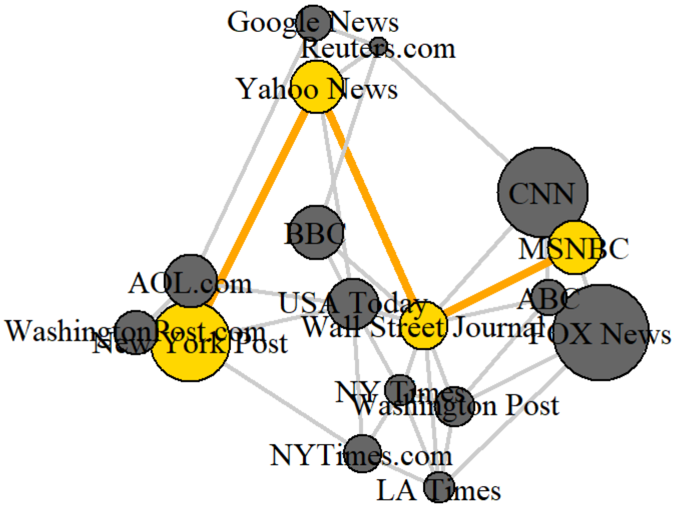
We can extract the distances to a node or set of nodes we are interested in. Here we will get the distance of every media from the New York Times.

```
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 an
d 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)
```



5.8 Groups

##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)

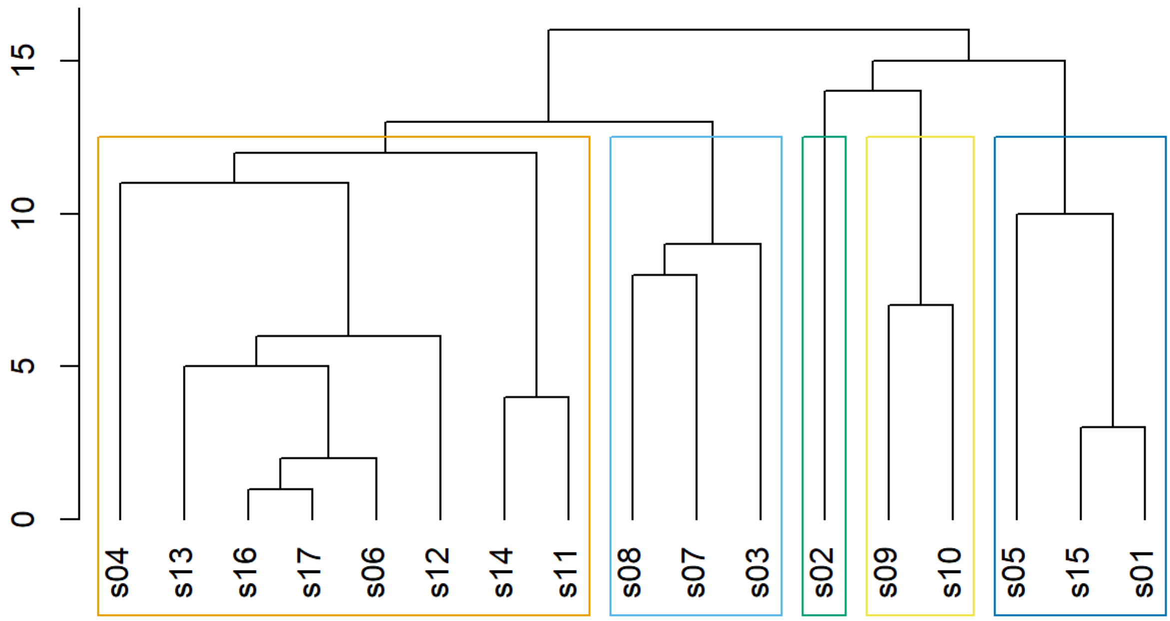
High-betweenness edges are removed sequentially (recalculating at each step) and the best partitioning of the network is selected.

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

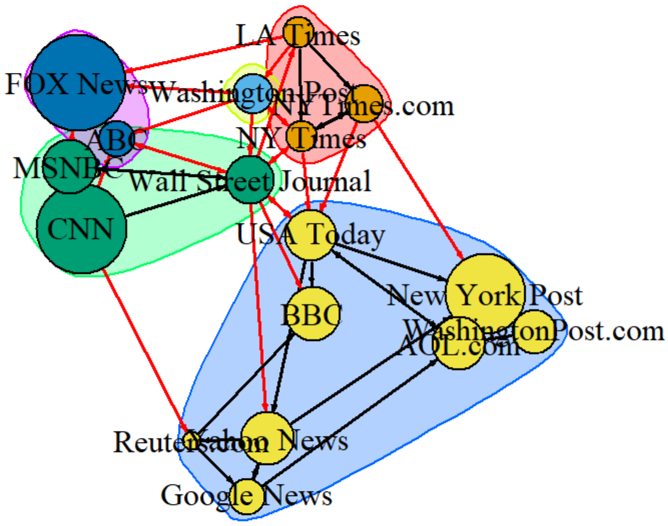
```
## Warning in cluster_edge_betweenness(net): At community.c:460 :M
embership vector
## will be selected based on the lowest modularity score.
```

```
## Warning in cluster_edge_betweenness(net): At community.c:467 :M
odularity
## calculation with weighted edge betweenness community detection
might not make
## sense -- modularity treats edge weights as similarities while e
dge 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)
```

```
## [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
```

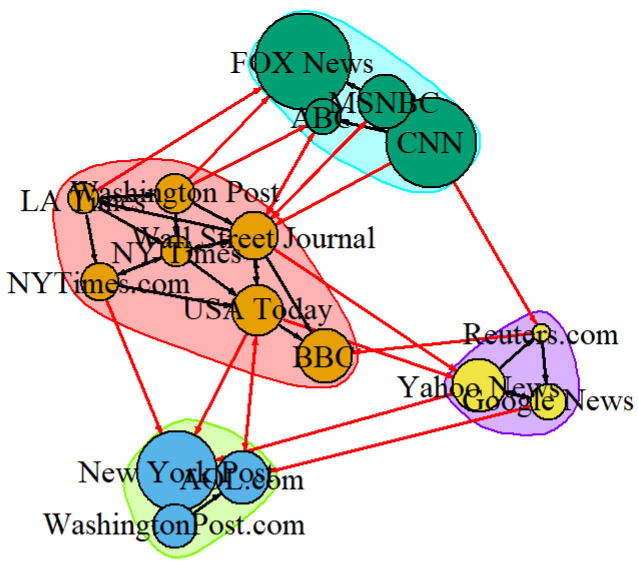
```
modularity(ceb) # how modular the graph partitioning is
```

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

Community detection based on based on propagating labels

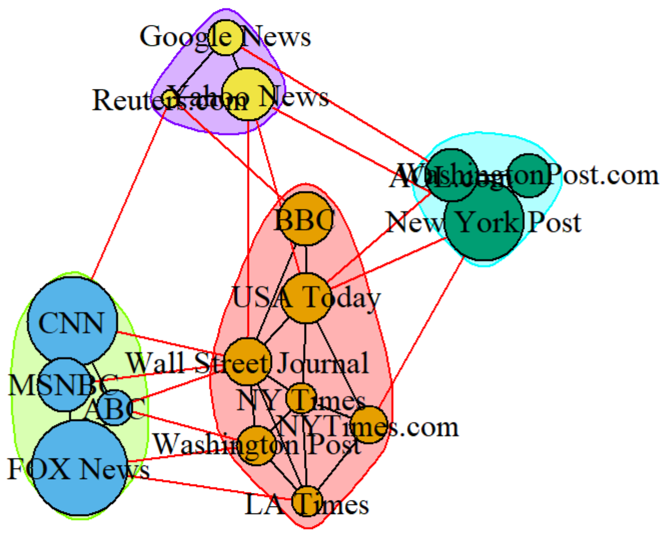
Assigns node labels, randomizes, than replaces each vertex’s label with the label that appears most frequently among neighbors. Those steps are repeated until each vertex has the most common label of its neighbors.

```
clp <- cluster_label_prop(net)

plot(clp, net)
```



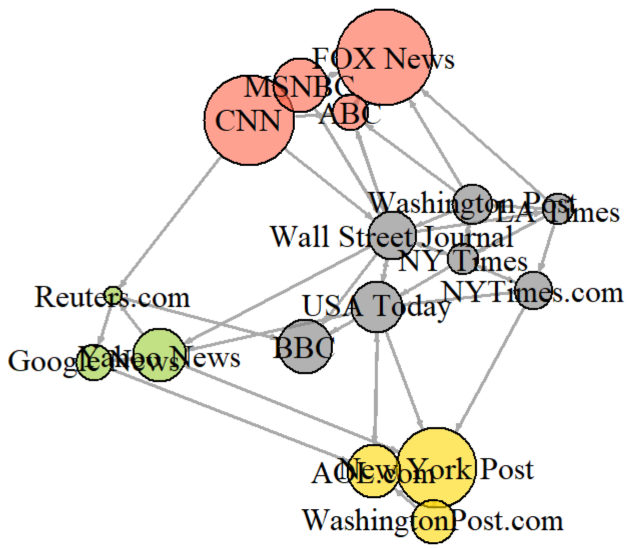
Community detection based on greedy optimization of modularity

```
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])
```


FOX News
MSNBC
CNN
ABC
Washington Post
LA Times
Wall Street Journal
NY Times
NYTimes.com
Reuters.com
USA Today
BBC
Google News
Yahoo News
AOL.com
New York Post
WashingtonPost.com