Network Analysis

Graphs, Centrality, and PageRank

DS 6030 | Fall 2022

networks.pdf

Contents

1 Preliminaries

1.1 Reading

- Network Science (Chapter 2)
- Statistical Analysis of Network Data with R (Chapter 2)
- McPherson, M., Smith-Lovin, L., & Cook, J. M. (2001). Birds of a Feather: Homophily in Social Networks.
- MMDS 5.1-5.3, 5.5
- R package `igraph`

1.2 Required R Packages

We will be using the R packages of:

- `igraph` for network modeling
- `sand` for data (supplement to book *Statistical Analysis of Network Data with R* by Kolaczyk and Csárdi)
- `igraphdata` for some network datasets
- `tidygraph` for `tbl_graph()`

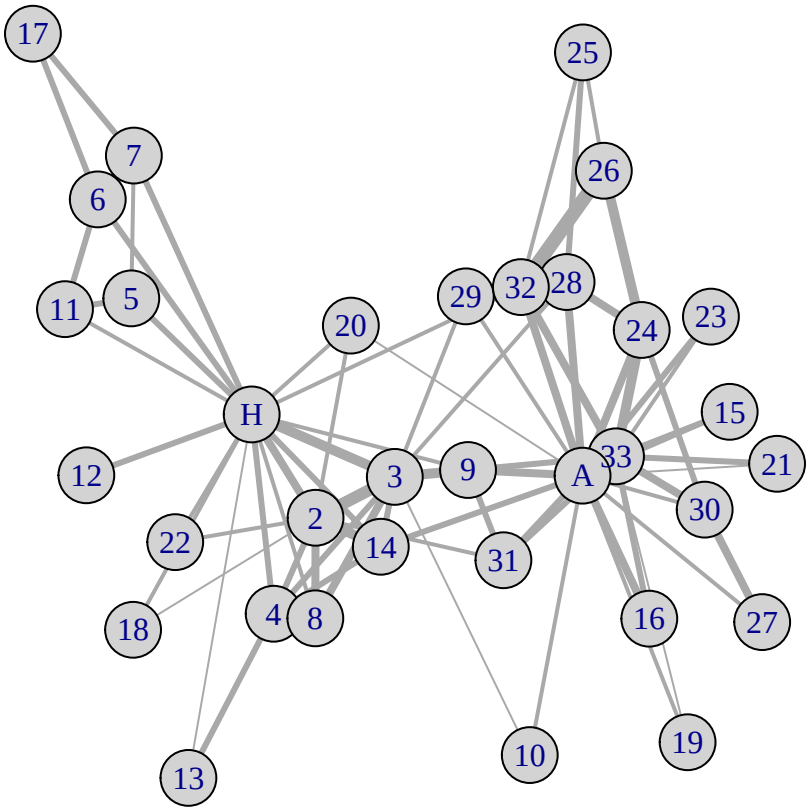
```
library(igraph)     # install.packages('igraph') if not installed
library(sand)       # install.packages('sand') if not installed
library(igraphdata)# install.packages('igraphdata') if not installed
library(tidygraph) # install.packages('tidygraph') if not installed
library(tidyverse) # load last so functions are available
```

2 Network Intro

2.1 Example: Zachary's karate club network

```
library(igraphdata)    # for karate data
data(karate)           # ?karate to see description

library(igraph)
plot(karate,
     layout=layout_with_fr(karate),    # determines coordinates of nodes
     vertex.color="lightgrey",         # color of vertices
     edge.width=E(karate)$weight)      # edge weights
```



Note

The famous karate network is based on the social network of 34 members of a university karate club. To uncover the true relationships between club members, sociologist Wayne Zachary (Wayne W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research* Vol. 33, No. 4 452-473) documented 78 pairwise links between members who regularly interacted outside the club. The edge weights represent the number of shared activities between the members.

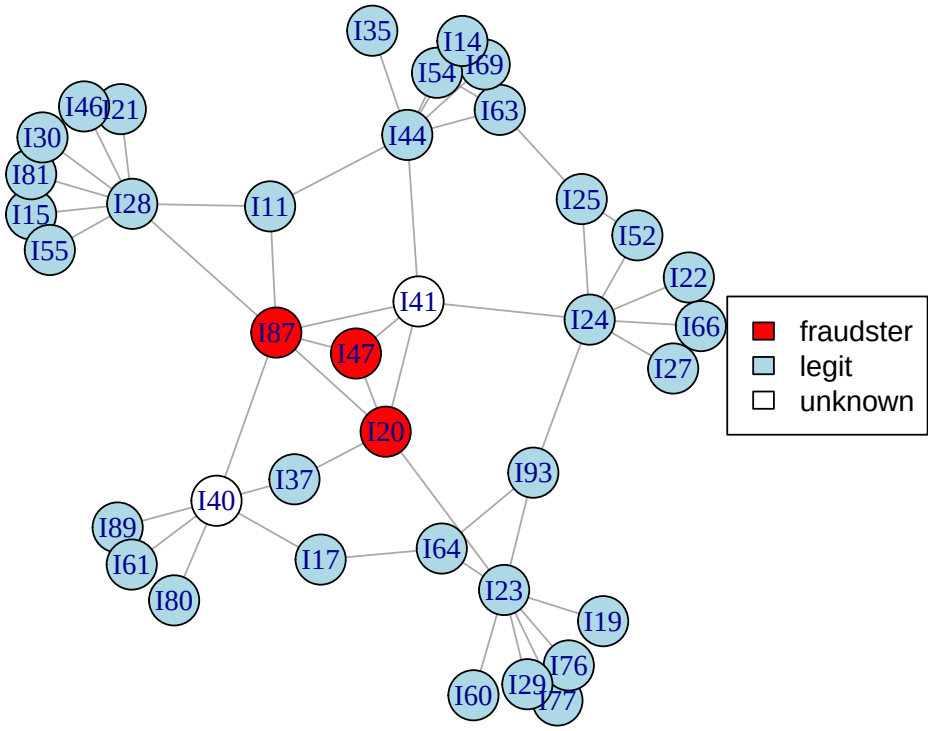
The network experienced a singular event during the study period: a conflict between the club's president, Mr. Hi (H), and the instructor, John A (A), split the club into two; About half of the members followed the instructor and the other half the president. The breakup unveiled club's underlying community structure.

Your Turn #1 : Zachary Karate

1. Do you think the graph can reveal which members will follow Mr. Hi? Why?
2. Which members will follow Mr. Hi? Why?

2.2 Example: Money Laundering Data

The DataCamp course Fraud Detection in R has some financial transaction data where some of the nodes (people) are engaged in fraudulence financial activities.



Your Turn #2 : Money Laundering

1. How would you classify node I40? Why?
2. How would you classify node I41? Why?
3. Does the graph layout inform your decision?

2.3 Community Detection for Karate

Community Detection is the name given to process of trying to discover the *community structure* of a network.

We are not going to go into details about community detection, but we can quickly run one community detection algorithm, termed *fast greedy* by igraph, that greedily optimizes something called the *modularity* score[1]. The basic idea of community detection (or network clustering) is to identify the nodes that form

[1] A Clauset, MEJ Newman, C Moore: Finding community structure in very large networks

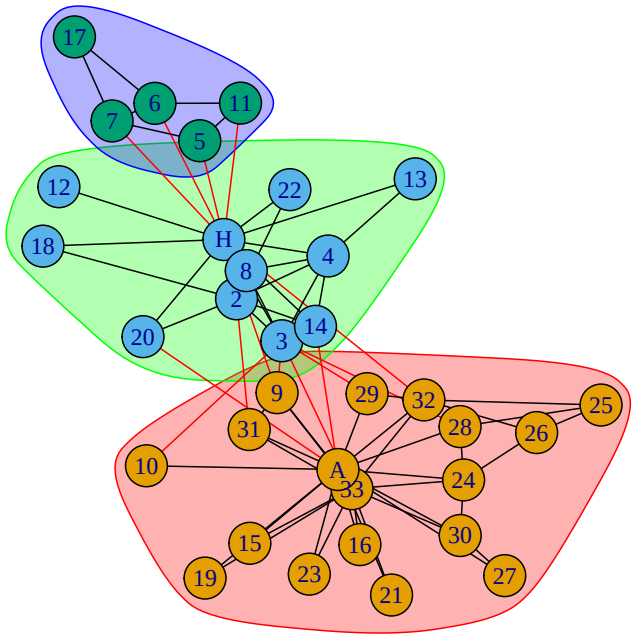
natural groups; usually based on the idea that nodes within the same community should have a higher probability of being connected to each other than to members of other communities.

Santo Fortunato is a good place to start if you are interested in learning more:

- Community detection in networks: A user guide https://arxiv.org/abs/1608.00163
- Community detection in graphs https://arxiv.org/abs/0906.0612

```
#-- Run community detection
fg = cluster_fast_greedy(karate)
membership(fg)
#>    Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 8
#>        2        2        2        2        3        3        3        2
#>  Actor 9 Actor 10 Actor 11 Actor 12 Actor 13 Actor 14 Actor 15 Actor 16
#>        1        1        3        2        2        2        1        1
#> Actor 17 Actor 18 Actor 19 Actor 20 Actor 21 Actor 22 Actor 23 Actor 24
#>        3        2        1        2        1        2        1        1
#> Actor 25 Actor 26 Actor 27 Actor 28 Actor 29 Actor 30 Actor 31 Actor 32
#>        1        1        1        1        1        1        1        1
#> Actor 33   John A
#>        1        1

#-- igraph has a built in plotting for communities
plot(fg, karate)
```



The fast-greedy community detection algorithm suggests there are 3 communities: one involving H, one A, and another that is connected to H, but tends to be more connected to each other than other nodes in H's community.

Note

Community detection is equivalent to clustering the nodes of a network. Like we discussed in the Clustering Section, there is no one best way to cluster. As with most unsupervised methods, community detection is best thought of as an exploratory, rather than confirmatory, approach.

3 Basic Network Concepts

3.1 Basic Definitions

- A graph can be represented by $G = (V, E)$ where V are the set of vertices (also called nodes) and E is a set of edges (also called links).
- There are $|V|$ nodes and $|E|$ edges in G
- The edge set E is a collection of pairs, (u, v) where $u, v \in V$
  - For undirected graphs, (u, v) is same as (v, u).
  - For directed graphs (digraph), (u, v) is distinct from (v, u)

3.2 Creating a Network

- A network needs two components:
  1. Nodes
  2. Edges
- **Nodes**: data frame of node labels and (optional) node attributes

```
nodes = tibble(node=1:7, group=c(1,1,2,2,1,2,1))
#> # A tibble: 7 x 2
#>    node group
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     3     2
#> 4     4     2
#> 5     5     1
#> 6     6     2
#> # ... with 1 more row
```

- **Edges**: data frame of edges
  - Common to use labels *from* and *to*, even for *undirected* networks
  - optional edge attributes

```
edges = tibble(from = c(1,1,2,2,3,4,4,6,4,5,6,6),
               to =   c(2,3,3,4,5,5,6,4,7,6,5,7),
               weight = c(1,1,2,2,3,3,2,2,1,1,2,2)) # edge weight
edges
#> # A tibble: 12 x 3
#>    from    to weight
#>   <dbl> <dbl>  <dbl>
#> 1     1     2      1
#> 2     1     3      1
#> 3     2     3      2
#> 4     2     4      2
#> 5     3     5      3
#> 6     4     5      3
#> # ... with 6 more rows
```

- R `igraph` package
  - The `igraph` package is one package in R to help with network data

```
library(igraph)
#-- Undirected Graph
g = graph_from_data_frame(d=edges, vertices=nodes, directed=FALSE)
g
#> IGRAPH 566d567 UNW- 7 12 --
```

```
#> + attr: name (v/c), group (v/n), weight (e/n)
#> + edges from 566d567 (vertex names):
#>  [1] 1--2 1--3 2--3 2--4 3--5 4--5 4--6 4--6 4--7 5--6 5--6 6--7

#-- Directed Graph
g_dir = graph_from_data_frame(d=edges, vertices=nodes, directed=TRUE)
g_dir
#> IGRAPH 566eb84 DNW- 7 12 --
#> + attr: name (v/c), group (v/n), weight (e/n)
#> + edges from 566eb84 (vertex names):
#>  [1] 1->2 1->3 2->3 2->4 3->5 4->5 4->6 6->4 4->7 5->6 6->5 6->7
```

- R `tidygraph` package
  - The `tidygraph` package also creates an `igraph` object, but includes some extra goodies. The interface to make the graph is slightly different

```
library(tidygraph)
tbl_graph(nodes = nodes, edges=edges, directed=FALSE, node_key='node')
#> # A tbl_graph: 7 nodes and 12 edges
#> #
#> # An undirected multigraph with 1 component
#> #
#> # Node Data: 7 x 2 (active)
#>    node group
#>   <int> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     3     2
#> 4     4     2
#> 5     5     1
#> 6     6     2
#> # ... with 1 more row
#> #
#> # Edge Data: 12 x 3
#>    from    to weight
#>   <int> <int>  <dbl>
#> 1     1     2      1
#> 2     1     3      1
#> 3     2     3      2
#> # ... with 9 more rows
```

- Node information is stored in the object `V(g)`

```
vertex_attr(g)   # get all node attributes
#> $name
#> [1] "1" "2" "3" "4" "5" "6" "7"
#>
#> $group
#> [1] 1 1 2 2 1 2 1

V(g)$name  # get vector of the names
#> [1] "1" "2" "3" "4" "5" "6" "7"
```

```
V(g)$group # get vector of group info
#> [1] 1 1 2 2 1 2 1

as_tibble(vertex_attr(g))  # make into a tibble
#> # A tibble: 7 x 2
#>   name  group
#>   <chr> <dbl>
#> 1 1         1
#> 2 2         1
#> 3 3         2
#> 4 4         2
#> 5 5         1
#> 6 6         2
#> # ... with 1 more row
```

- Edge information is stored in the object `E(g)`

```
edge_attr(g)   # get all node attributes
#> $weight
#>  [1] 1 1 2 2 3 3 2 2 1 1 2 2

E(g)$weight    # get vector of weights
#>  [1] 1 1 2 2 3 3 2 2 1 1 2 2

tibble(edge = attr(E(g), "vnames"),  # make into a tibble
       weight = E(g)$weight)
#> # A tibble: 12 x 2
#>   edge  weight
#>   <chr>  <dbl>
#> 1 1|2        1
#> 2 1|3        1
#> 3 2|3        2
#> 4 2|4        2
#> 5 3|5        3
#> 6 4|5        3
#> # ... with 6 more rows
```

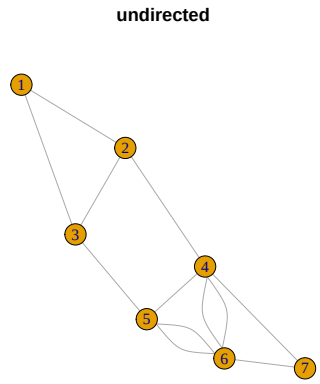
3.3 Visualizing a Network

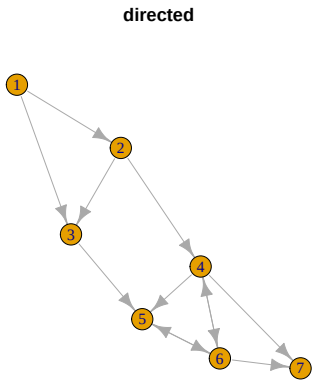
3.3.1 Graph Layout

Graph layouts are projections of the vertices and edges into some space. Different layouts reveal different aspects of a graph.

- Note: 2D layouts can be very misleading; don't trust your eyes
- Choose the layout to help reveal the structure. In `igraph`,
  - `layout.fructerman.reingold` is a spring-embedder method
  - `layout.kamada.kawai` is based on multidimensional scaling (MDS)
  - These will be a function of the *distance* between vertices

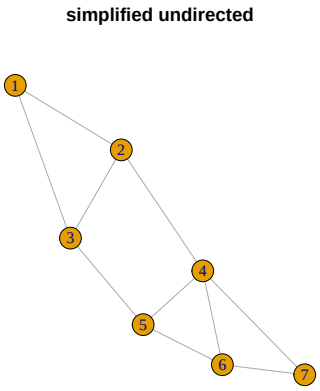
```
g.layout = layout_with_fr(g) # create layout (node coordinates)
plot(g, layout=g.layout, main="undirected")     # plot undirected graph
plot(g_dir, layout=g.layout, main="directed")   # plot directed graph
```





- Notice that we have multiple edges in the *undirected* graph. We can simplify the graph into a *proper* undirected graph (with only a single edge between nodes).

```
# Note: there is a conflict with purrr::simplify() and igraph::simplify(),
#  thus I will be specific that I want igraph's simplify function.
g = igraph::simplify(g)     # this removes multiple edges, loops, and combines edge attr.
plot(g, layout=g.layout, main="simplified undirected")     # plot undirected graph
```



Note

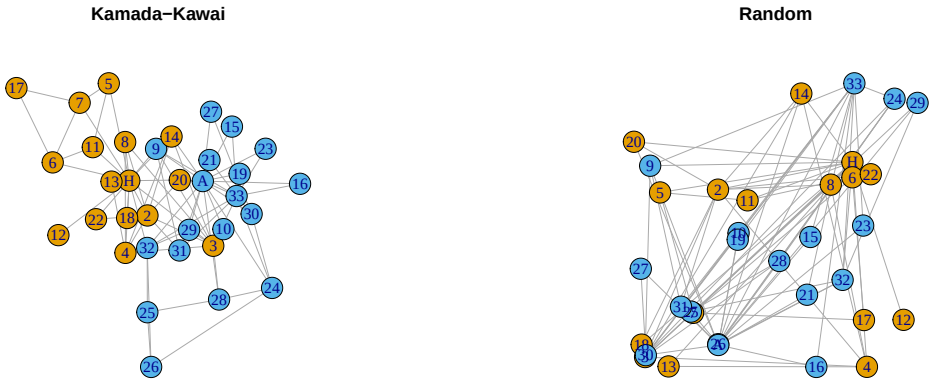
Don't miss the important information in the code comments about `igraph::simplify()`:

- if the argument `remove.multiple=TRUE` (the default setting), then all edge attributes are combined (e.g., summed).
- The graph now has single edges only, and the weights are aggregated.

```
#- Notice the difference from the previous version
tibble(edge = attr(E(g), "vnames"),
       weight = E(g)$weight)
#> # A tibble: 10 x 2
#>   edge  weight
#>   <chr>  <dbl>
#> 1 1|2        1
#> 2 1|3        1
#> 3 2|3        2
#> 4 2|4        2
#> 5 3|5        3
#> 6 4|5        3
#> # ... with 4 more rows
```

- The layout can have a **big** influence on how you perceive the network

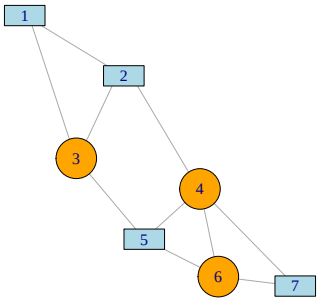
```
plot(karate, layout=layout_with_kk(karate), main="Kamada-Kawai")  # Kamada-Kawai layout
plot(karate, layout=layout_randomly(karate), main="Random")       # Random layout
```



3.3.2 Graph Decoration

- The nodes and edges can be *decorated* with color, size, shape, etc.

```
plot(g, layout=g.layout,
   vertex.size=30,
   vertex.shape=ifelse(V(g)$group==1, "rectangle", "circle"),
   vertex.color=ifelse(V(g)$group==1, "lightblue", "orange"))
```



3.4 Representations for Graphs

There are three main components of any graph:

1. The edges (with attributes)

from	to	weight
1	2	1
1	3	1
2	3	2
2	4	2
3	5	3
4	5	3
4	6	4
4	7	1
5	6	3
6	7	2

2. The nodes/vertices (with attributes)

name	group	x_coord	y_coord
1	1	0.816	0.042
2	1	0.647	0.370
3	2	0.120	0.757
4	2	0.544	0.002
5	1	0.185	0.160
6	2	0.636	0.145
7	1	0.074	0.519

3. Graph Level Attributes
   - directed vs. undirected
   - source/citation, time of collection,
   - graph level metrics (e.g., mean degree, graph assortativity, graph density)

- Other Plotting Attributes

- Nodes: layout coordinates, colors, shapes, size
- Edges: color, shape, size

3.4.1 Edge List

An edge list is usually represented as a two-column matrix (or data.frame)

```
as_edgelist(g)
#>       [,1] [,2]
#>  [1,] "1"  "2"
#>  [2,] "1"  "3"
#>  [3,] "2"  "3"
#>  [4,] "2"  "4"
#>  [5,] "3"  "5"
#>  [6,] "4"  "5"
#>  [7,] "4"  "6"
#>  [8,] "4"  "7"
#>  [9,] "5"  "6"
#> [10,] "6"  "7"
```

3.4.2 Adjacency Matrix

An adjacency matrix is the $|V| \times |V|$ matrix, $\mathbf{A}$ such that

$$A_{ij} = \begin{cases} 1, & \text{if } \{i,j\} \in E, \\ 0, & \text{otherwise} \end{cases}$$

For undirected graphs, the adjacency matrix will by symmetric.

```
as_adj(g)        # binary and symmetric
#> 7 x 7 sparse Matrix of class "dgCMatrix"
#>   1 2 3 4 5 6 7
#> 1 . 1 1 . . . .
#> 2 1 . 1 1 . . .
#> 3 1 1 . . 1 . .
#> 4 . 1 . . 1 1 1
#> 5 . . 1 1 . 1 .
#> 6 . . . 1 1 . 1
#> 7 . . . 1 . 1 .
as_adj(g_dir)    # binary and not-symmetric
#> 7 x 7 sparse Matrix of class "dgCMatrix"
#>   1 2 3 4 5 6 7
#> 1 . 1 1 . . . .
#> 2 . . 1 1 . . .
#> 3 . . . . 1 . .
#> 4 . . . . 1 1 1
#> 5 . . . . . 1 .
#> 6 . . . 1 1 . 1
#> 7 . . . . . . .
```

3.4.3 Adjacency List

The adjacency list is an array (in R, a list) of size $|V|$, where the elements of the list indicate the set of vertices that are adjacent. It is essentially the sparse representation of the adjacency matrix.

```
as_adj_list(g)
#> $`1`
```

```
#> + 2/7 vertices, named, from 5693afc:
#> [1] 2 3
#>
#> $`2`
#> + 3/7 vertices, named, from 5693afc:
#> [1] 1 3 4
#>
#> $`3`
#> + 3/7 vertices, named, from 5693afc:
#> [1] 1 2 5
#>
#> $`4`
#> + 4/7 vertices, named, from 5693afc:
#> [1] 2 5 6 7
#>
#> $`5`
#> + 3/7 vertices, named, from 5693afc:
#> [1] 3 4 6
#>
#> $`6`
#> + 3/7 vertices, named, from 5693afc:
#> [1] 4 5 7
#>
#> $`7`
#> + 2/7 vertices, named, from 5693afc:
#> [1] 4 6
```

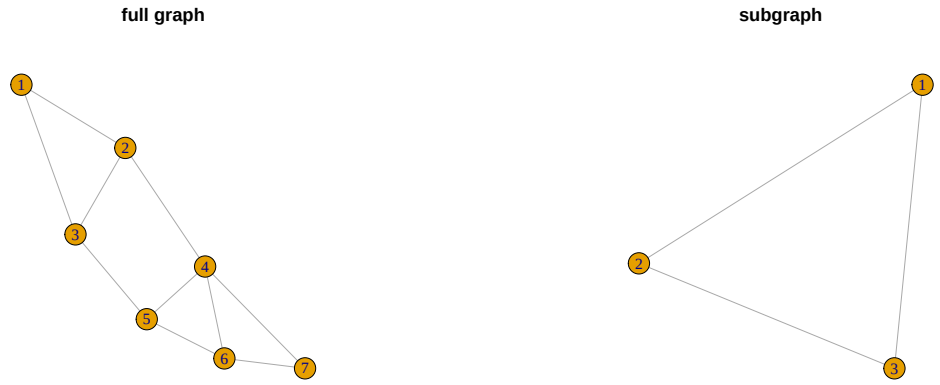
3.5 Weighted Edges

- Edges can have attributes that describe the nature of the connection between two vertices
- An example is to assign edge weights $\{w_{ij} : e_{ij} \in E\}$
  - Weights can be measurements of things like: flow rate, number of transactions, call time, travel speed, etc.
- More generally, consider the weight matrix W, which is the $|V| \times |V|$ matrix containing the edge weights. The weights will be $W_{ij} = 0$ if $A_{ij} = 0$.
  - The adjacency matrix is a special case of weight matrix with binary weights

3.6 Subgraphs

- A graph $H = (V_H, E_H)$ is a subgraph of $G = (V_G, E_G)$ if $V_H \subseteq V_G$ and $E_H \subseteq E_G$.
- An *induced subgraph* of graph G is a subgraph $G' = (V', E')$ where $V' \subseteq V$ is a pre-specified subset of vertices and $E' \subseteq E$ is the collection of edges to be found in G among that subset of vertices.

```
g3 = induced_subgraph(g, v=1:3)    # only select vertices 1:3
plot(g, layout=g.layout, main='full graph')
plot(g3, main='subgraph')
```

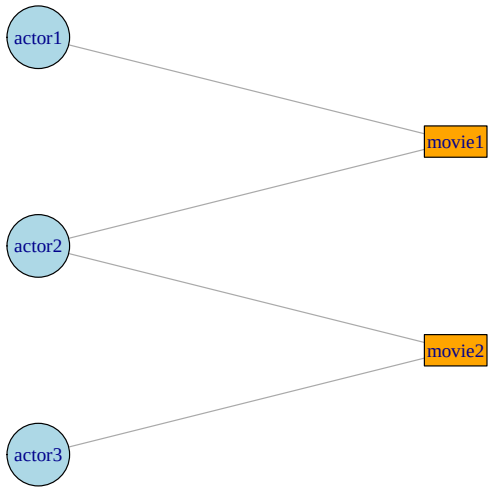


3.7 Bipartite graphs

A bipartite graph (also called *two-mode*) is a graph $G = (V, E)$ such that the vertex set V may be partitioned into two disjoint sets V_1 and V_2, and each edge in E has one endpoint in V_1 and the other in V_2.

```
g.bip <- graph.formula(actor1:actor2:actor3,
   movie1:movie2, actor1:actor2 - movie1,
   actor2:actor3 - movie2)
V(g.bip)$type <- grepl("^movie", V(g.bip)$name)
plot(g.bip, layout=-layout.bipartite(g.bip)[,2:1],
   vertex.size=30, vertex.shape=ifelse(V(g.bip)$type,
     "rectangle", "circle"),
   vertex.color=ifelse(V(g.bip)$type, "orange", "lightblue"))

get.incidence(g.bip)         # get the incidence matrix
#>        movie1 movie2
#> actor1      1      0
#> actor2      1      1
#> actor3      0      1
```



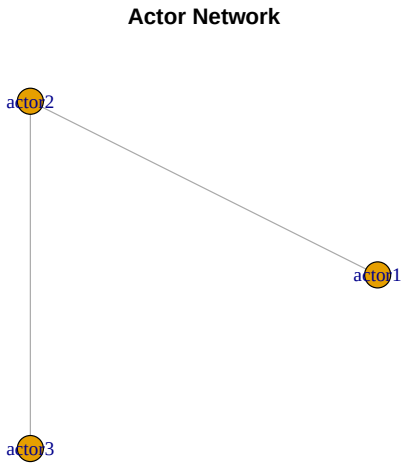
Some examples:

- Membership networks: V_1 are the members and V_2 the organizations

- Recommender data: V_1 are the movies and V_2 the reviewers
- Market basket data: V_1 are the shoppers and V_2 are the items in the store
- Travel: V_1 are the people and V_2 are the places they visit

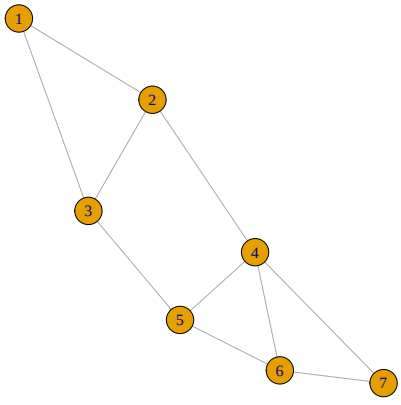
A bipartite graph can be accompanied by the induced subgraph formed by connecting the vertices, say V_1, by assigning an edge to vertices that edges in E to at least one common vertex in V_2

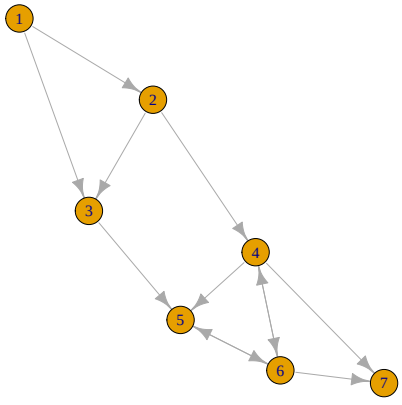
```
plot(bipartite.projection(g.bip)$proj1,
     main="Actor Network",
     layout=layout_in_circle)
```



3.8 Graphs and Matrix Notation

We will be using our example graph





3.8.1 Adjacency matrix

$$A_{ij} = \begin{cases} 1, & \text{if } \{i,j\} \in E, \\ 0, & \text{otherwise} \end{cases}$$

3.8.2 Degree

- The row sums give the vertex *degree* (for undirected graphs),

$$d_i = \sum_j A_{ij}$$

  which is the number of edges vertex i is connected to

```
A = get.adjacency(g, sparse=FALSE)
A
#>   1 2 3 4 5 6 7
#> 1 0 1 1 0 0 0 0
#> 2 1 0 1 1 0 0 0
#> 3 1 1 0 0 1 0 0
#> 4 0 1 0 0 1 1 1
#> 5 0 0 1 1 0 1 0
#> 6 0 0 0 1 1 0 1
#> 7 0 0 0 1 0 1 0

tibble(
  rowSums(A), # degree from adjacency matrix
  degree(g)   # using igraph::degree() function
)
#> # A tibble: 7 x 2
#>   `rowSums(A)` `degree(g)`
#>          <dbl>       <dbl>
#> 1            2           2
#> 2            3           3
#> 3            3           3
#> 4            4           4
#> 5            3           3
#> 6            3           3
#> # ... with 1 more row
```

- For directed graphs (digraphs),
  - row sums $d_i^{out} = \sum_j A_{ij}$ give *out-degree*
  - column sums $d_i^{in} = \sum_j A_{ji}$ given *in-degree*

```
A2 = get.adjacency(g_dir, sparse=FALSE)
A2
#>   1 2 3 4 5 6 7
#> 1 0 1 1 0 0 0 0
#> 2 0 0 1 1 0 0 0
#> 3 0 0 0 0 1 0 0
#> 4 0 0 0 0 1 1 1
#> 5 0 0 0 0 0 1 0
#> 6 0 0 0 1 1 0 1
#> 7 0 0 0 0 0 0 0

tibble(
  in.degree = degree(g_dir, mode="in"),    # colSums(A2)
  out.degree = degree(g_dir, mode="out")   # rowSums(A2)
)
#> # A tibble: 7 x 2
#>   in.degree out.degree
#>       <dbl>      <dbl>
#> 1         0          2
#> 2         1          2
```

```
#> 3         2          1
#> 4         2          3
#> 5         3          1
#> 6         2          3
#> # ... with 1 more row
```

- For weighted graphs, the graph *strength* is the respective sums of the weight matrix W. See `igraph::strength()`

3.8.3 Movement on a graph

- A *walk* on a graph G describes a sequence of adjacent vertices $(v_0, v_1, ..., v_n)$, where each v_i is connected to v_{i+1} by an edge.
- A *connected* graph is one where a walk exists between every pair of vertices
- *Geodesic distance* (also called *number of hops*) is the length of the shortest path between two vertices

```
distances(g, weights=NA)   # geodesic or shortest-path distances
#>   1 2 3 4 5 6 7
#> 1 0 1 1 2 2 3 3
#> 2 1 0 1 1 2 2 2
#> 3 1 1 0 2 1 2 3
#> 4 2 1 2 0 1 1 1
#> 5 2 2 1 1 0 1 2
#> 6 3 2 2 1 1 0 1
#> 7 3 2 3 1 2 1 0

distances(g, weights=E(g)$weight)  # use edge weights
#>   1 2 3 4 5 6 7
#> 1 0 1 1 3 4 6 4
#> 2 1 0 2 2 5 5 3
#> 3 1 2 0 4 3 6 5
#> 4 3 2 4 0 3 3 1
#> 5 4 5 3 3 0 3 4
#> 6 6 5 6 3 3 0 2
#> 7 4 3 5 1 4 2 0
```

- The matrix power, A^r gives the number of walks of length r between vertices

```
#- set r
r = 2     # walks of length 2

#- Direct method
Ar = diag(nrow(A))
for(i in 1:r) {Ar <- Ar %*% A}    # r = 2
Ar
#>      1 2 3 4 5 6 7
#> [1,] 2 1 1 1 1 0 0
#> [2,] 1 3 1 0 2 1 1
#> [3,] 1 1 3 2 0 1 0
#> [4,] 1 0 2 4 1 2 1
#> [5,] 1 2 0 1 3 1 2
#> [6,] 0 1 1 2 1 3 1
#> [7,] 0 1 0 1 2 1 2
```

```
#- eigen method
eig = eigen(A)
Ar2 = eig$vectors %*% diag(eig$values^r) %*% solve(eig$vectors)
round(Ar2)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
#> [1,]    2    1    1    1    1    0    0
#> [2,]    1    3    1    0    2    1    1
#> [3,]    1    1    3    2    0    1    0
#> [4,]    1    0    2    4    1    2    1
#> [5,]    1    2    0    1    3    1    2
#> [6,]    0    1    1    2    1    3    1
#> [7,]    0    1    0    1    2    1    2
```

Note

Let A be an $n \times n$ adjacency matrix with eigen-decomposition:

$$\begin{aligned} Ax &= \lambda x \\ AAx &= A\lambda x \qquad \text{multiple both sides by } A \\ A^2x &= \lambda^2 x \qquad \text{because } Ax = \lambda x \end{aligned}$$

Let $A = V\Lambda V^{-1}$ where Λ is a diagonal matrix of eigenvalues and V the orthogonal matrix of eigenvectors.

$$\begin{aligned} AA &= (V\Lambda V^{-1})(V\Lambda V^{-1}) \\ A^2 &= V\Lambda^2 V^{-1} \end{aligned}$$

And thus,

$$\begin{aligned} A(AA) &= (V\Lambda V^{-1})V\Lambda^2 V^{-1} \\ A^3 &= V\Lambda^3 V^{-1} \end{aligned}$$

and so on.

3.8.4 Graph Laplacian

Graph Laplacian is the $|V| \times |V|$ matrix $L = D - A$, where $D = \text{diag}[d_i : i \in V]$ is the diagonal matrix with degree along the diagonal. It is useful for calculating the sum of squared differences of node attributes for all connected nodes:

$$\mathbf{x}^\top L\mathbf{x} = \sum_{\{i,j\} \in E} (x_i - x_j)^2$$

for node attributes $\mathbf{x} \in \mathbb{R}^{|V|}$.

The graph Laplacian is used for *spectral clustering*.

4 Homophily, Assortativity, and Fraud Prediction

4.1 Homophily

"Birds of a feather flock together"

"Misery loves company"

McPherson et al (2001)[2] observed that people's personal/social networks are homogeneous with regard to many sociodemographic, behavioral, and intrapersonal characteristics. As such, contact between similar people occur at a higher rate than among dissimilar people; this principal is termed *homophily* (greek: same + love/affection).

Note

Perhaps the most basic source of homophily is **space**:

> *We are more likely to have contact with those who are closer to us in geographic location than those who are distant.*

Zipf (1949) stated the principle as a matter of effort: It takes more energy to connect to those who are far away than those who are readily available.

4.1.1 Examples:

- **Political Blogs:**[3]

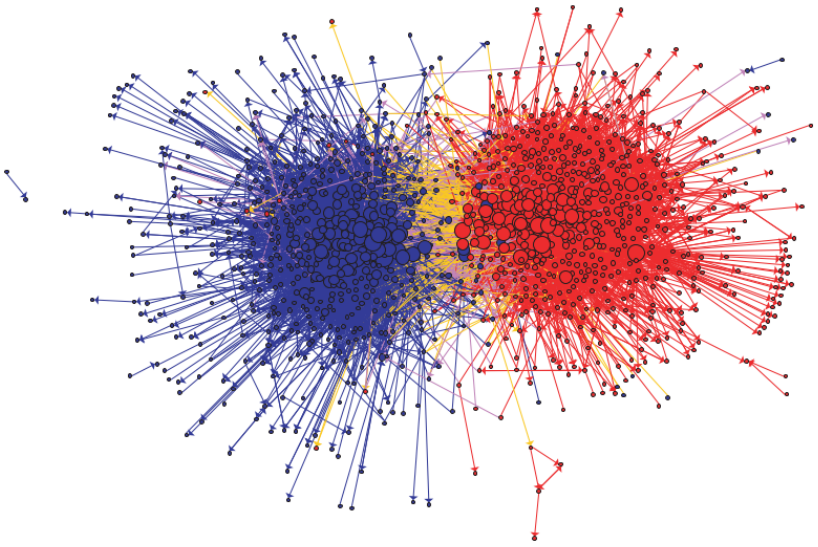
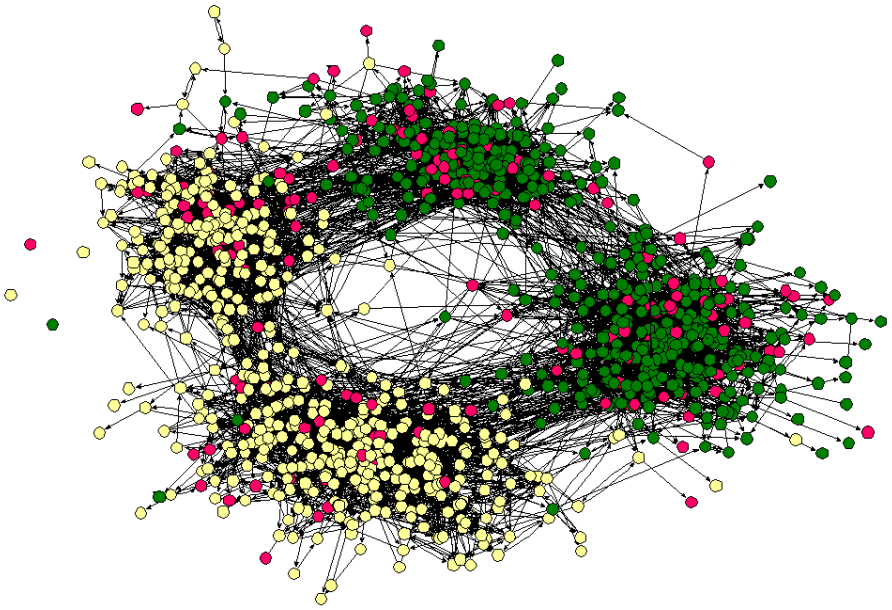


Figure 1: Community structure of political blogs (expanded set), shown using utilizing a GEM layout [11] in the GUESS[3] visualization and analysis tool. The colors reflect political orientation, red for conservative, and blue for liberal. Orange links go from liberal to conservative, and purple ones from conservative to liberal. The size of each blog reflects the number of other blogs that link to it.

[2]McPherson, M., Smith-Lovin, L., & Cook, J. M. (2001). Birds of a Feather: Homophily in Social Networks. *Annual Review of Sociology*, 27, 415–444

[3]Lada A. Adamic and Natalie Glance. 2005. The political blogosphere and the 2004 U.S. election: divided they blog. In Proceedings of the 3rd international workshop on Link discovery (LinkKDD '05). ACM, New York, NY, USA

- **School Segregation:**[4]

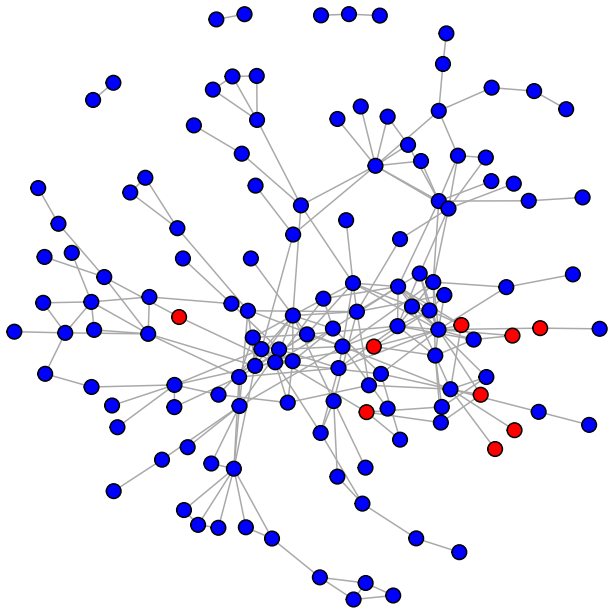


Nodes are students in a high school and two nodes are connected if one student named the other student as friend (the data was collected as part of the Add Health study). The color of the nodes corresponds to the race of the students. As we can see, "yellow" students are much more likely to be friends with other yellow students and "green" students are more likely to connect to other green students. (Interestingly, the "pink" students, who are in the vast minority seem to be distributed throughout the network.

- **Yeast protein interaction network**[5] *Notice that the red nodes are not connected to other red nodes.*

[4]Moody (2001) "Race, school integration, and friendship segregation in America," *American Journal of Sociology* 107, 679-716. Figure taken from: http://networksciencebook.com/chapter/7#summary7. Text taken from: http://social-dynamics.org/homophily

[5]X. Jiang, N. Nariai, M. Steffen, S. Kasif, E. Kolaczyk (2008) "Integration of relational and hierarchical network information for protein function prediction". *BMC Bioinform.* 9, 350. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2535605/. Data accessed from the R `sand` package: `data(ppi.CC, package="sand")`. Color indicates whether the protein contains the 'rho GTPase-activating protein domain' (IPR000198) motif.

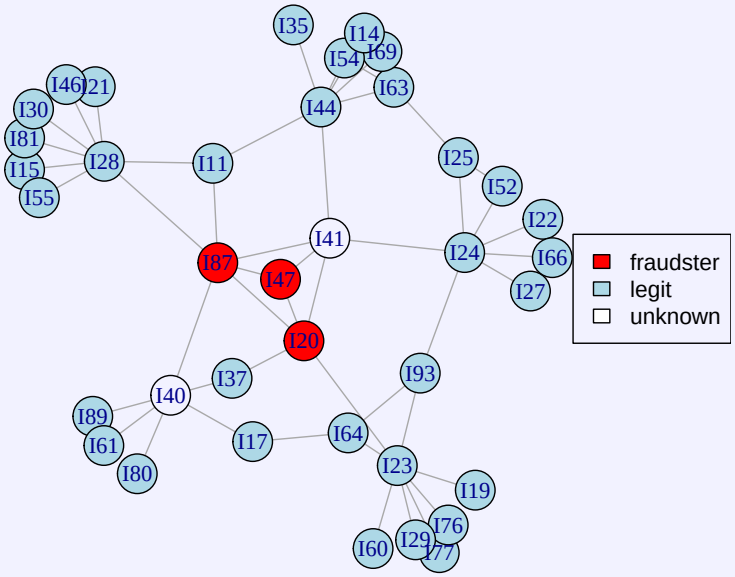


4.2 Node Prediction

- If there is homophily in the network, then we can expect **nodes with similar attributes to be connected by an edge**.
- More specifically, under homophily, we might expect that node attributes could be predicted from the attributes of its closest neighbors.

Your Turn #3

Consider the *Money Laundering* network.



1. Write an algorithm to *estimate the probability* that an unlabeled node is a fraudster.

2. What should your algorithm predict for nodes I40 and I41?
3. What if the node has no neighbors (i.e., degree of 0)?
4. How would you assess the *uncertainty* in your estimate?
5. How would your approach change if the network had weights or directed edges?

4.2.1 Testing the Node Prediction Algorithm

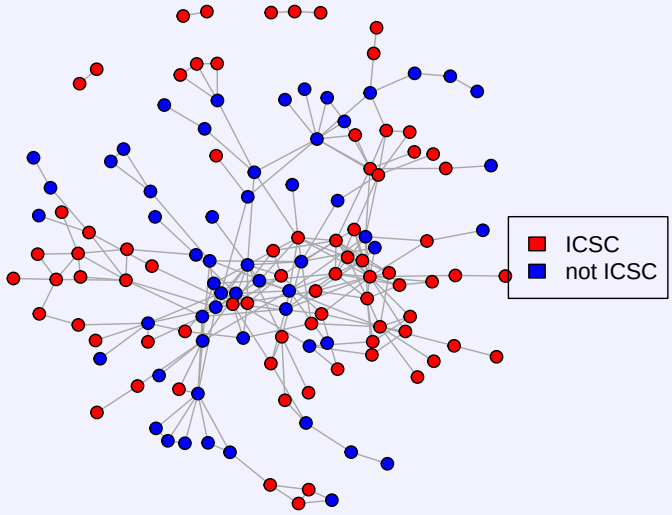
We can use a type of *resampling* to evaluate how well our algorithm might do on an actual network.

1. Take a real network with binary/categorical label
2. Randomly remove some node labels (for some fraction f)
3. Run the algorithm (using different values of k)
4. Record results
5. Evaluate effects of f and k.

Your Turn #4

Evaluate how well the simple nearest neighbor method works on the Yeast Protein Interaction Data for predicting the `ICSC` attribute which indicates whether the protein is annotated with the "intracellular signaling cascade" GO term. It takes a binary (zero or one) value.

Yeast Protein Interaction Data: ICSC label



Examine the results for different values of f and k. See the R code `node-predict.R` from the course website for help.

4.3 Link Prediction

It can also be useful to have a model for estimating the presence of an edge between two nodes.

- Based on the notion of homophily, we can use some *similarity score* between nodes i and j to estimate the probability of e_{ij}.

Your Turn #5

Think up 3 ways to measure the similarity of two nodes, when it is unknown whether an edge exists

between them or not.

5 Node Importance: Vertex Centrality

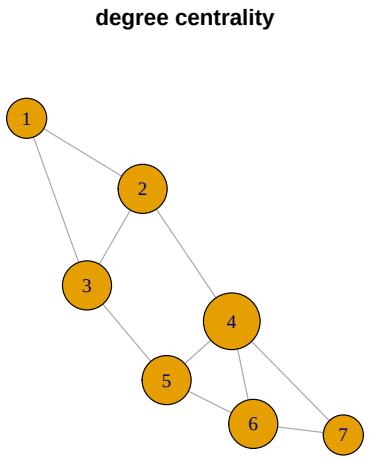
Centrality tries to assess how "important" a vertex is.

- Which actors in a social network seem to hold the 'reins of power'?
- How authoritative does a webpage seem to be considered?
- How critical is a router in the internet network?

5.1 Degree centrality

the number of edges (sum of weights) a vertex has is the most basic definition of importance

```
deg = degree(g)
cent.deg = deg/sum(deg)
plot(g, layout=g.layout, vertex.size=80*sqrt(cent.deg))
title("degree centrality")
```



Mathematically, the degree for node i can be written

$$c_i = \sum_j A_{ij}$$

5.2 Closeness centrality

Measures the importance in terms of how 'close' a vertex is to the other vertices in the graph.

The standard approach is to let the centrality vary inversely with a measure of the total distance of a vertex to all the others:

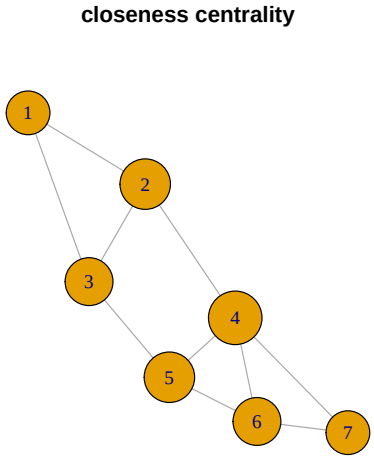
$$c(v) = \frac{1}{\sum_{u \in V} \text{dist}(v,u)}$$

Note

Closeness is only defined if graph is connected!

```
close = centr_clo(g)$res
cent.close = close/sum(close)
```

```
plot(g, layout=g.layout, vertex.size=80*sqrt(cent.close))
title("closeness centrality")
```



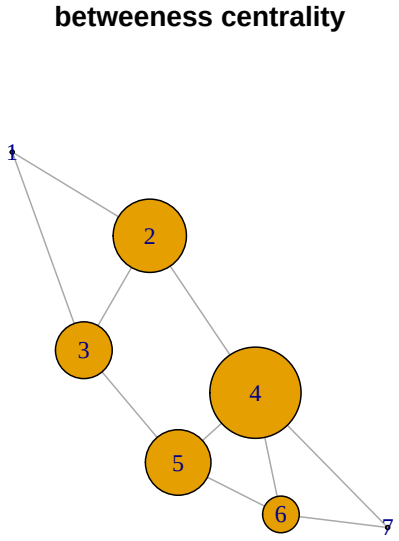
5.3 Betweenness centrality

Measures how many paths cross through a vertex. An important vertex is one in which lots of information flows.

$$c(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

where $\sigma(s,t|v)$ is the total number of *shortest paths* between s and t that pass through v, and $\sigma(s,t)$ is the total number of shortest paths between s and t (regardless of whether or note they pass through v).

```
between = centr_betw(g)$res
cent.between = between/sum(between)
plot(g,  layout=g.layout, vertex.size=80*sqrt(cent.between))
title("betweeness centrality")
```



Note

Centrality scores are commonly standardized so they can be understood relative to the other nodes in the network. Above, I list the centralities as score_i/sum(score), which makes the centralities sum to one over all nodes. In `igraph`, the `centr_<metric>()` functions have an argument named `normalized=TRUE` which instead divides the score by the theoretical maximum.
You may also see score_i / norm(score), where norm is a vector norm.

The `igraph` package has two versions of centrality calculations.

- The ones starting with `centr_` do not consider edge weights.
- The others (e.g., `betweenness()`, `closeness()`, `eigen_centrality()`) will allow weights.
  - Not only will they allow weights, but if you edges have an attribute named `weight`, it will use it without warning. Must set `weights=NA` to ignore weights.

5.4 Eigenvector centrality

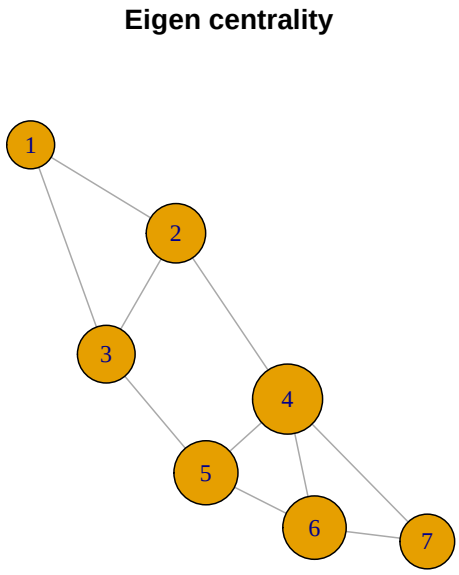
Based on the notion that an important vertex will be connected to other importance vertices.

$$c(v) = \alpha \sum_{\{u,v\} \in E} c(u)$$

- Notice that the eigenvector centrality for vertex v is the sum of the centrality of the vertices that it is connected to (scaled by α).
- In a more standard form, $A\mathbf{c} = \lambda\mathbf{c}$, is seen to be the eigen equations for A where $\mathbf{c}$ are the eigenvectors and λ the eigenvalues.
  - If A is not a stochastic matrix (rows sum to one, non-negative), then use the eigenvector corresponding to the largest magnitude eigenvalue
  - Standardize c to either have a maximum value of 1 or norm (sum of squares) of 1.

```
#eigen = eigen_centrality(g)$vector  # first eigenvalue (max of 1)
eigen = centr_eigen(g)$vector
```

```
cent.eigen = eigen/sum(eigen)
plot(g,layout=g.layout,vertex.size=80*sqrt(cent.eigen))
title("Eigen centrality")
```



5.4.1 Power Method

We can use the *power method* to solve $\mathbf{c} = A\mathbf{c}$

$$c^{\text{new}} = Ac^{\text{old}}/||Ac^{\text{old}}||$$

```
A = as_adj(g, sparse=FALSE)
n = nrow(A)
y = matrix(1/n, n, 1) # initialize
for (i in 1:50){      # run until converges
  y = A %*% y
  y = y/sum(y)
}
tibble(cent.eigen, y)
#> # A tibble: 7 x 2
#>   cent.eigen  y[,1]
#>        <dbl>  <dbl>
#> 1     0.0912 0.0912
#> 2     0.140  0.140
#> 3     0.132  0.132
#> 4     0.195  0.195
#> 5     0.163  0.163
#> 6     0.160  0.160
#> # ... with 1 more row
```

5.5 Other Centrality and Node Importance Measures

See the `netrankr` package for a more detailed description (and periodic table!) of centrality.

6 PageRank

6.1 Random Surfer

The pagerank algorithm is based on the idea of a random (internet) surfer who randomly clicks on links from the current page.

- Consider transforming the adjacency matrix A into the appropriate transition matrix (markov chain)
  - For directed networks
  - Let $P_{ij} = A_{ij}/d_i^{out}$ be the row-standardized transition probability

$$P_{ij} = \begin{cases} \frac{1}{d_i^{out}}, & \text{if } \{i,j\} \in E, \\ 0, & \text{otherwise} \end{cases}$$

  - P_{ij} is the probability of a move from $i \to j$ if all edges are equally likely (random walk)
  - $P = D^{-1}A$ where $D = diag(d^{out})$

```
P = sweep(A, 1, rowSums(A), '/')
round(P, 2)
#>      1    2    3    4    5    6    7
#> 1 0.00 0.50 0.50 0.00 0.00 0.00 0.00
#> 2 0.33 0.00 0.33 0.33 0.00 0.00 0.00
#> 3 0.33 0.33 0.00 0.00 0.33 0.00 0.00
#> 4 0.00 0.25 0.00 0.00 0.25 0.25 0.25
#> 5 0.00 0.00 0.33 0.33 0.00 0.33 0.00
#> 6 0.00 0.00 0.00 0.33 0.33 0.00 0.33
#> 7 0.00 0.00 0.00 0.50 0.00 0.50 0.00
```

6.2 PageRank Details

Consider the directed graph representation of the www: $G = (V, E)$, where $n = |V|$ are the number of webpages

- Webpages link (hyperlink) to other webpages with directed edges
- An important webpage is one that many (important) pages **link to it**
- The (naive) PageRank score of page i is

$$\begin{aligned} r_i &= \sum_{\{j,i\} \in E} \frac{r_j}{d_j^{out}} \\ &= \sum_j P_{ji} r_j \end{aligned}$$

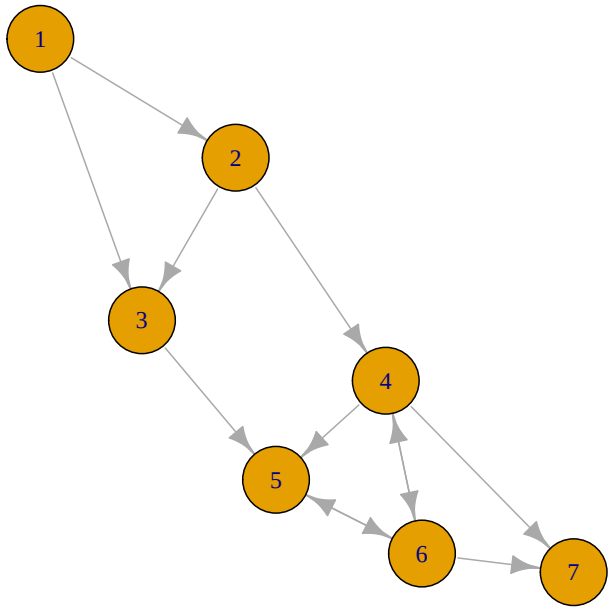
- This gives the system of equations

$$\mathbf{r} = P^{\mathsf{T}} \mathbf{r}$$

This is equivalent to eigenvector centrality! But, this naive version has three main problems for webpage centrality.

1. Friendless (nodes with only out links, no in links)
2. Dead Ends (nodes with no out links)
3. Spider Traps (communities with no out links)

They all pertain to pages that cannot be reached by other pages.



The approach taken in PageRank is to add a dampening factor. Or alternatively the idea that the websurfer will randomly click links, but occasionally will pick another webpage (from the full set of vertices) at random and starts again. This can be modeled by

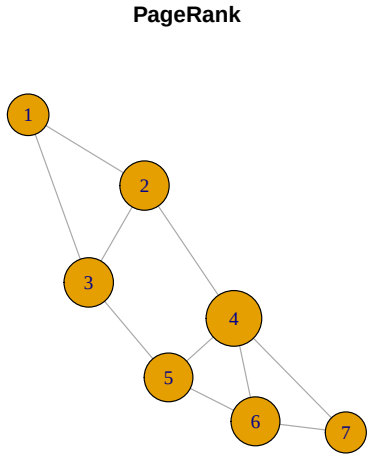
$$\mathbf{r} = \frac{1-d}{n}\mathbf{e} + dP^{\mathsf{T}}\mathbf{r}$$

- $0 \leq d \leq 1$ is the *dampening factor* or the probability the surfer keeps clicking on the links (and thus $1-d$ the probability of random selection)
  - Original method used $d = 0.85$
- $\mathbf{e}$ is a column vector of ones

- Equivalently,

$$\begin{aligned} r_i &= \frac{1-d}{n} + d \sum_{\{j,i\} \in E} \frac{r_j}{d_j^{out}} \\ &= \frac{1-d}{n} + d \sum_{j=1}^{n} P_{ji} r_j \end{aligned}$$

```
pr = page_rank(g, weights = NA)$vector
cent.pr = pr/sum(pr)
plot(g,layout=g.layout,vertex.size=80*sqrt(cent.pr))
title("PageRank")
```



- The power iteration method can also be used to solve this equation, which finds the eigenvector with eigenvalue of 1. This is a very fast approach which can be parallel processed.

```
P = sweep(A, 1, rowSums(A), '/')
d = 0.85

y = matrix(1/n, n, 1) # initialize
for (i in 1:50){     # run until converges
  y = (1-d)/n + d*crossprod(P,y)
  y = y/sum(y)         # additional sum to control roundoff error
}
tibble(cent.pr, y)
#> # A tibble: 7 x 2
#>   cent.pr y[,1]
#>     <dbl> <dbl>
#> 1   0.107 0.107
#> 2   0.150 0.150
#> 3   0.151 0.151
#> 4   0.192 0.192
#> 5   0.147 0.147
#> 6   0.148 0.148
#> # ... with 1 more row
```

7 More Resources

- https://github.com/briatte/awesome-network-analysis
- Two nice R packages to help put graph analysis in the *tidyverse* are:
  - `ggraph`
  - `tidygraph`
- Tutorials
  - https://www.jessesadler.com/post/network-analysis-with-r/
  - https://rviews.rstudio.com/2019/03/06/intro-to-graph-analysis/
  - https://www.data-imaginist.com/2018/tidygraph-1-1-a-tidy-hope/
  - https://www.data-imaginist.com/2017/introducing-tidygraph/