Clustering on SN

Contents

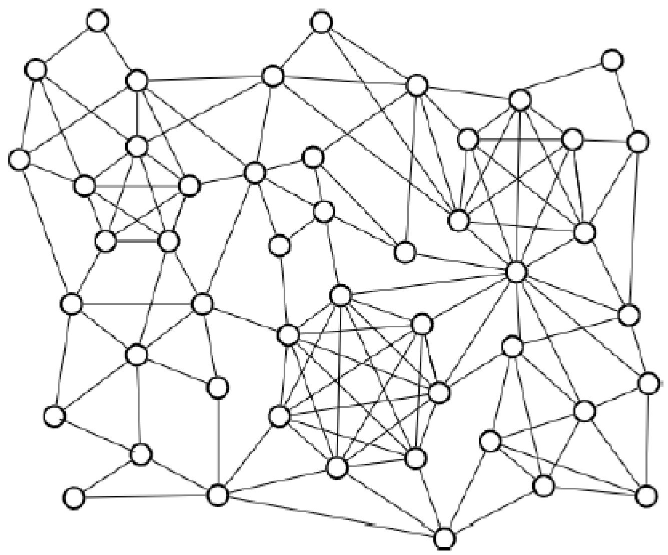
```r
library('igraph')
```

TO SAVE YOUR TIME, PLEASE START DOWNLOADING THIS NETWORK RIGHT NOW

Cohesive subgraphs

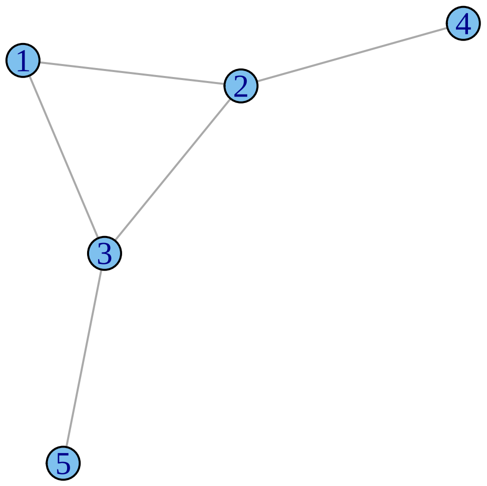
Graph cliques

Graph clique is a subset of vertices of a graph such that every two vertices in the clique are adjacent.



How many cliques can you see on this graph?

```r
plot(graph.famous("bull"))
```



There was a couple of definitions about the cliques in graph on the lecture.

A maximum clique is a clique that cannot be extended by including one more adjacent vertex (not included in larger one). Can you name maximum cliques in the given graph?

A maximal clique is a clique of the largest possible size in a given graph.

And, finally, graph clique number is the size of the maximum clique. Bull graph's clique number is 3.

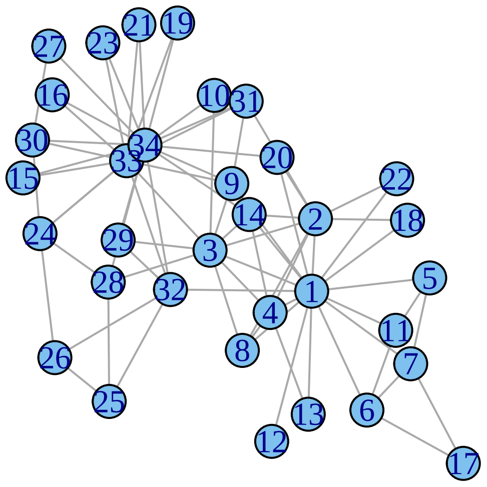
`maximal.cliques` returns lists of vertices, that form a maximum graph. Let's see maximum cliques for a bull graph:

```r
maximal.cliques(graph.famous("bull"))
```

```
## [[1]]
## [1] 4 2
##
## [[2]]
## [1] 5 3
##
## [[3]]
## [1] 1 2 3
```

Let's demonstrate some useful functions for finding cliques. Our graph today is again Zachary's Karate Club graph:

```r
g = graph.famous("Zachary")
plot(g)
```



We can define sizes of maximal cliques we interested in:

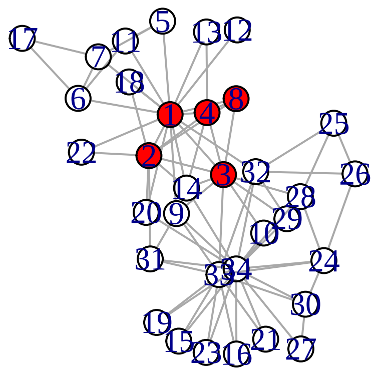
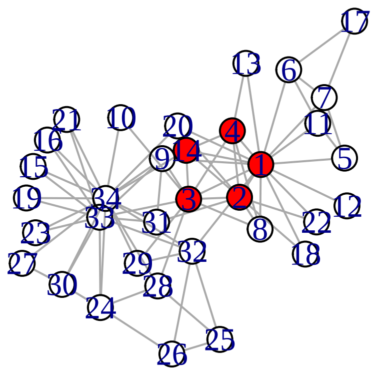
```r
maximal.cliques(g, min = 4, max = 5) # maximal cliques of sizes 4 and 5
```

```
## [[1]]
## [1] 24 34 33 30
##
## [[2]]
## [1] 34  9 33 31
##
## [[3]]
## [1] 2 1 4 3 8
##
## [[4]]
## [1]  2  1  4  3 14
```

`maximal.cliques` returns lists of vertices - maximal cliques. `clique.number` returns graph's clique number.

Let's find and show maximal cliques for Zachary Carate Club graph: `lrg = largest.cliques(g)` returns ids of nodes - largest cliques

```r
largest = largest.cliques(g)

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

labels = rep(0, vcount(g))

labels[largest[[1]]] = 2
plot(g, vertex.color = labels)
labels = rep(0, vcount(g))
labels[largest[[2]]] = 2
plot(g, vertex.color = labels)
```

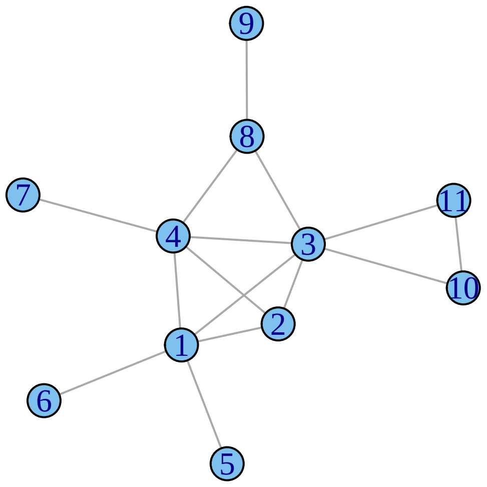
```r
par(op)
```

k-core

k-core is a maximal subset of vertices such that each is connected to at least k others in the subset.

R has a function wich calculates the *coreness* for each vertex. The coreness of a vertex is k if it belongs to the k-core but not to the (k+1)-core.

```r
# Let's make some graph
z<-graph.empty(n=11, directed = FALSE)
z <- add.edges(z,c(1,2, 1,3, 1,4, 1,6, 1,5, 2,3, 2,4, 3,10, 3,11, 3,8, 3,4, 4,8, 4,7, 8,9, 10,11))
plot(z)
```

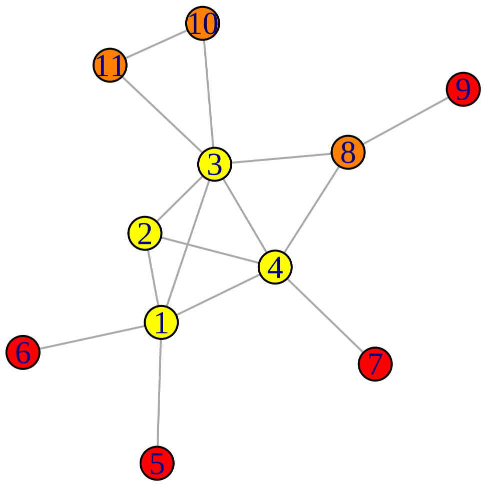


Now we find maximum k-core and pick out it on graph

```
coreness <- graph.coreness(z)
max_cor <- max(coreness)
max_cor
```

```
## [1] 3
```

```
color_bar <- heat.colors(max_cor)
plot(z, vertex.color = color_bar[coreness])
```



Network communities

Network communities are groups of vertices such that vertices inside the group connected with many more edges than between groups.

Communitiy density Graph $G(V;E)$, n = $|V|$, m = $|E|$ Community - set of nodes S n_s - number of nodes in S, m_s - number of edges in S Graph density:

$$\rho = \frac{m}{n(n-1)/2}$$

Community internal density:

$$\delta_{int}(C) = \frac{m_s}{n_s(n_s-1)/2}$$

External edges density:

$$\delta_{ext}(C) = \frac{m_{ext}}{n_c(n_c-1)/2}$$

Community (cluster):

$$\delta_{int} > \rho, \delta_{ext} < \rho$$

Community cuts

Graph cut:

$$Q = cut = c_s$$

Ratio cut:

$$Q = \frac{cut}{|S|} + \frac{cut}{|V\S|} = \frac{nc_s}{n_s(n-n_s)}$$

Normalized cut:

$$Q = \frac{cut}{Vol(S)} + \frac{cut}{Vol(V\S)} = \frac{c_s}{2m_s + c_s} + \frac{c_s}{2(m-m_s)+c_s}$$

Conductance (quotient cut):

$$Q = \frac{cut}{min(Vol(S), Vol(V\S))} = \frac{c_s}{2m_s + c_s}$$

Community modularity

Compare fraction of edges within the cluster to expected fraction in random graph with identical degree sequence:

$$Q = \frac{1}{4}(m_s - E(m_s))$$

Modularity score

$$Q = \frac{1}{2m}\sum_{ij}\left(A_{ij} - \frac{k_i k_j}{2m}\right)\delta(c_i, c_j) = \sum_u (e_{uu} - a_u^2)$$

e_{uu} - fraction of edges within community u $a_u = \sum_u e_{uv}$ fraction of ends of edges attached to nodes in u The higher the modularity score - the better are communities. Modularity score range $Q \in [-1/2, 1)$, single community $Q = 0$.

Community detection

Community detection is an assignment of vertices to communities. Consider only sparse graphs $m \ll n^2$ Each community should be connected. Combinatorial optimization problem: - optimization criterion (cut, conductance, modularity) - optimization method Exact solution NP-hard. (bi-partition: $n = n_1 + n_2, n! = (n_1!n_2!)$ combinations) Solved by greedy, approximate algorithms or heuristics Recursive top-down 2-way partition, multiway partition Balanced class partition vs communities

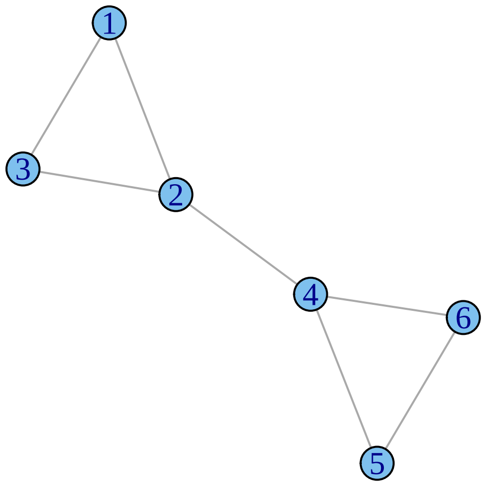
The list of community detection algorithms in igraph

- `edge.betweenness.community` [Newman and Girvan, 2004]
- `fastgreedy.community` [Clauset et al., 2004] (modularity optimization method)
- `label.propagation.community` [Raghavan et al., 2007]
- `leading.eigenvector.community` [Newman, 2006]
- `multilevel.community` [Blondel et al., 2008] (the Louvain method)
- `optimal.community` [Brandes et al., 2008]
- `spinglass.community` [Reichardt and Bornholdt, 2006]
- `walktrap.community` [Pons and Latapy, 2005]
- `infomap.community` [Rosvall and Bergstrom, 2008]

Newman-Girvan Edge-Betweenness

Edge betweenness **Edge betweenness** is equal to the number of shortest paths $\sigma_{st}(e)$ from all vertices to all others that pass through that edge e. $C_B(e) = \sum_{s \neq t} \frac{sigma_{st}(e)}{sigma_{st}}$

```r
g<-graph.empty(n=6, directed = FALSE)
g <- add.edges(g,c(1,2, 2,3, 1,3, 2,4, 4,5, 4,6, 5,6))
plot(g)
```

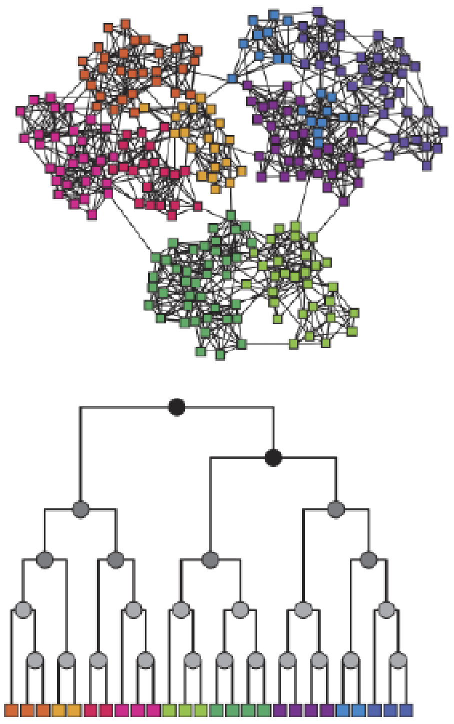


```r
betw <- edge.betweenness(g)
#E(g)
#betw
```

The algorithm The Newman-Girvan algorithm detects communities by progressively removing edges from the original network. The Girvan-Newman algorithm focuses on edges that are most likely “between” communities.

Algorithm:

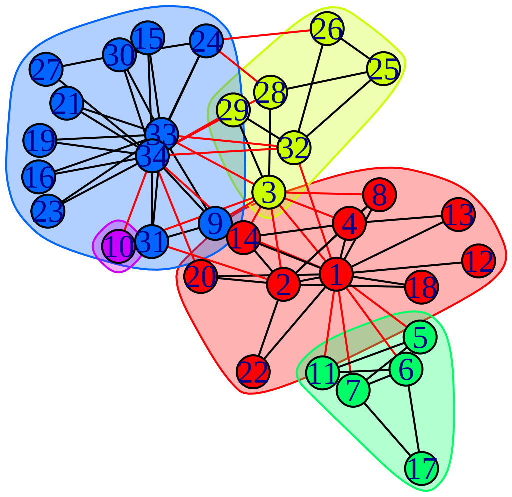
- Step 1: the betweenness of all existing edges in the network is calculated first.
- Step 2: the edge with the highest betweenness is removed.
- Step 3: the betweenness of all edges affected by the removal is recalculated.
- Step 4: steps 2 and 3 are repeated until no edges remain.



The best partition is selected based on modularity.

There is edge.betweenness.community function in R

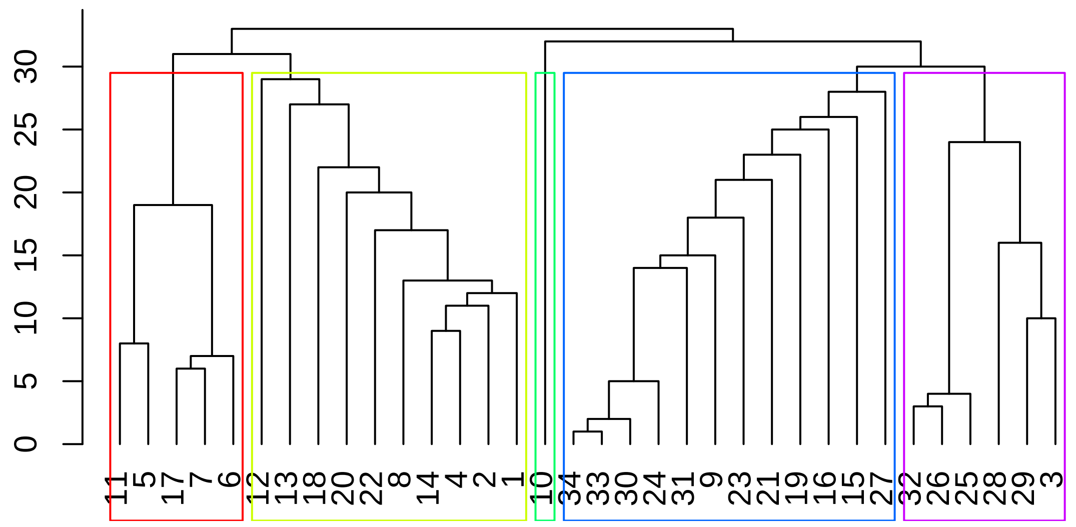
```r
g <- graph.famous("Zachary")
eb <- edge.betweenness.community(g)
plot(eb, g)
```



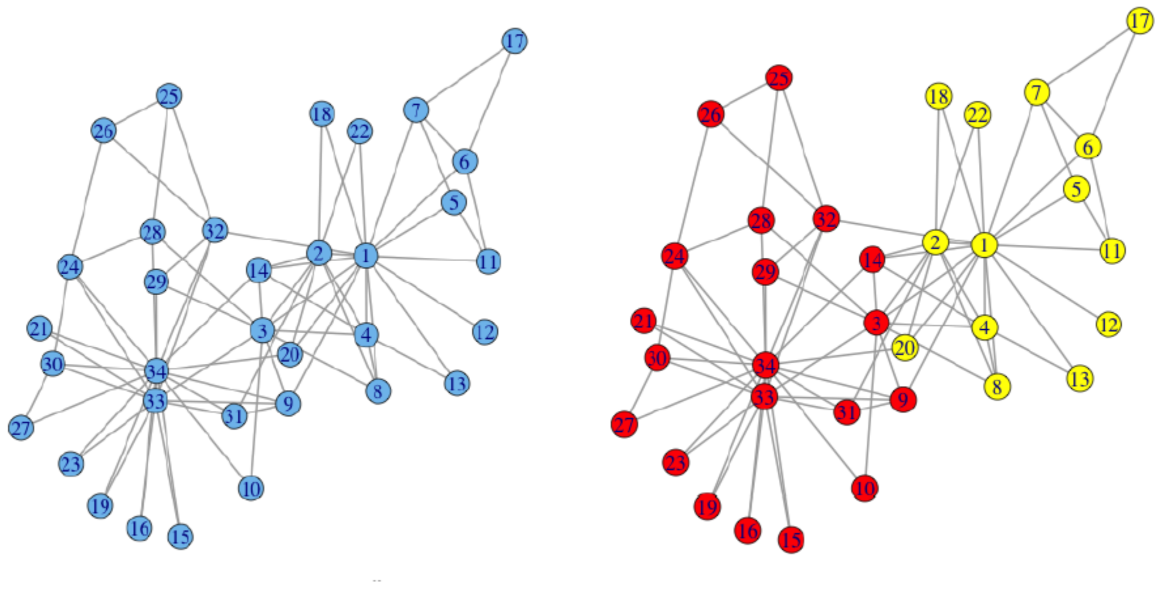
```
## A bit more hand-made way
# color_map = c("grey","blue","black","yellow","red","green")
# membership = cutat(eb, no = 4)
# membership = eb$membership
# plot(g, vertex.color = eb$membership)
```

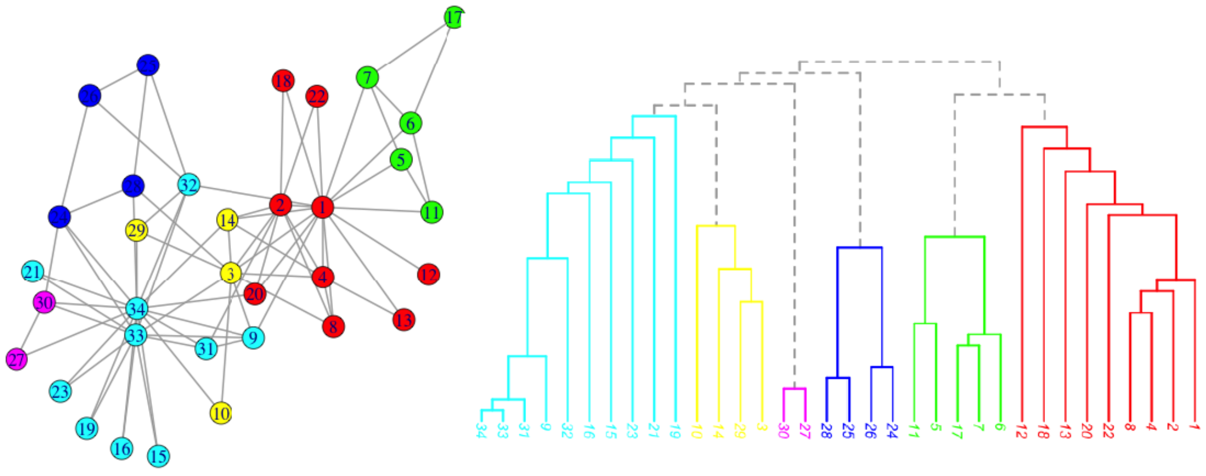
Also you can obtain dendrogram:

```
dendPlot(eb, mode="hclust", rect = 5)
```



```
## Optionally you can run this
# dend <- as.dendrogram(eb)
# plot(dend)
```





Spectral graph partitioning

Indicator vector $s_i = \pm 1$.

Integer optimization problem.

Normalized cuts:

$$Q = \frac{1}{4} s^T L s, L = D - A$$

Modularity optimization:

$$Q = \frac{1}{4m} s^T B s, B_{ij} = A_{ij} - \frac{k_i k_j}{2m}$$

Relaxation $s \to x, s \in Z_n, x \in R^n$.

Quadratic optimization problem under constraints.

Solved by finding min/max eigenvalues and eigenvectors of L or B:

$Lx = \lambda D x$; or $Bx = \lambda x$;

Eigenvector rounds up to indicator vector $s = sign(x)$.

Spectral modularity maximization

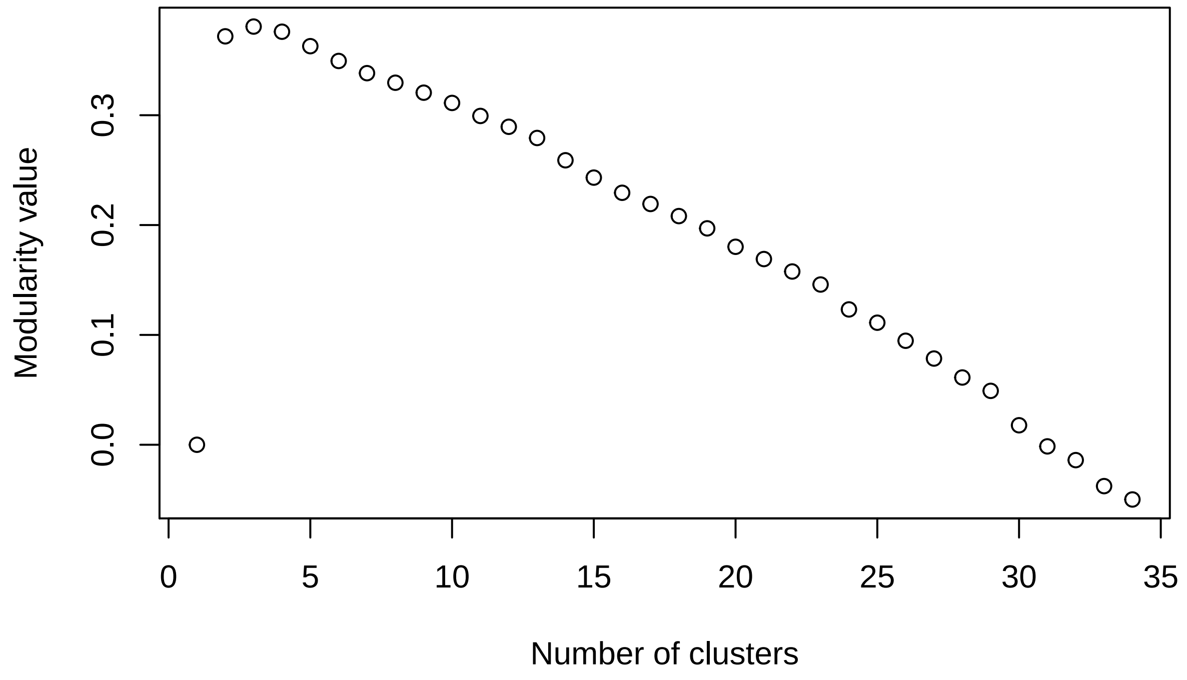
```
Algorithm: Spectral modularity maximization: two-way partition
Input: adjacency matrix $A$
Output: class indicator vector $s$
compute $k = deg(A)$;
compute $B = A - \frac{1}{2m} k k^T$
solve for maximal eigenvector $Bx = \lambda x$;
set $s = sign(x_{max})$
```

Greedy Modularity maximization

Alternatively to the previous method, this one is agglomerative. Intially consider a network s.t. * There is no edges * All clusters consist of a single vertex

Iteratively add an edge that delivers maximum modularity gain and merge correspondent communitues.

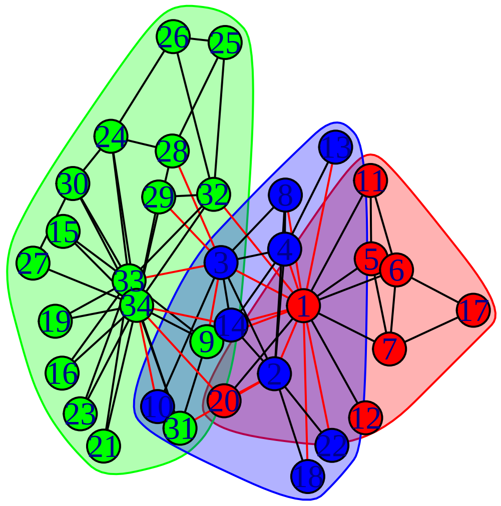
```
g <- graph.famous(name = "Zachary")
mm <- fastgreedy.community(g)

plot(rev(mm$modularity), xlab = 'Number of clusters', ylab = 'Modularity value')
```



```
which.max(rev(mm$modularity))
```

```
## [1] 3
```

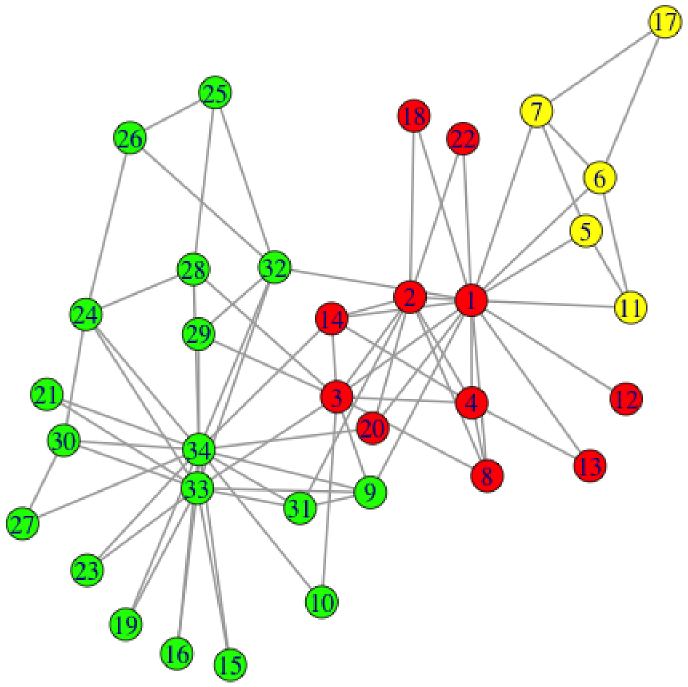
```r
plot(mm, g)
```



Label propagation

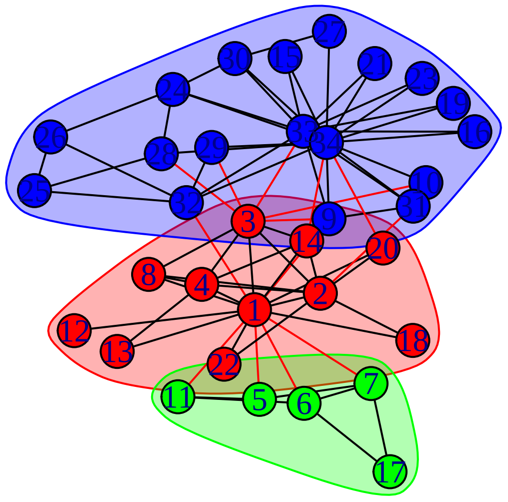
Label propagation algorithm consists of four steps:

- Step 1: Initialize labels
- Step 2: Randomize node ordering
- Step 3: For every node replace its label with occurring with the highest frequency among neighbors
- Step 4: Repeat steps 2-3 until every node will have a label that the maximum number of its neighbors have



Warning! Due to **step 2** you may get different results.

```
g <- graph.famous("Zachary")
lp <- label.propagation.community(g)
plot(lp, g)
```



Wikipedia example

Load wikipedia network in R and run some community detection algorithm. Extract article names in some communities and check whether they make sense?

```
g <- read.graph('wikipedia.gml', format = 'gml')
g <- as.undirected(g)
```

The next lines of code might be usefull for interpretation

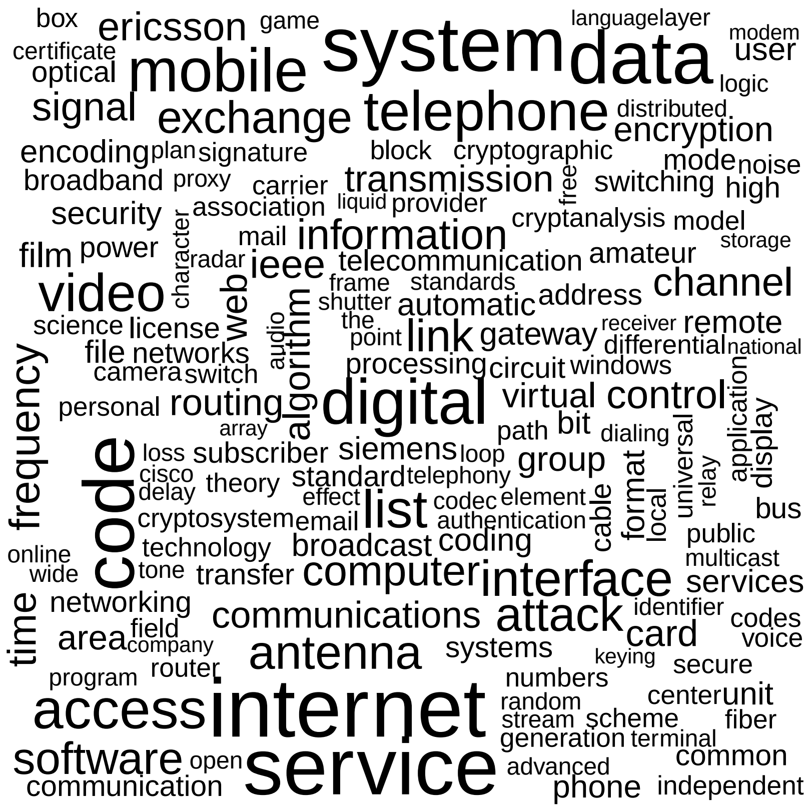
```
mm <- fastgreedy.community(g)
l <- V(g)$label[mm$membership == 2]
text <- paste(l, collapse = ' ')
```

```
#install.packages(c("tm", "SnowballC", "wordcloud", "RColorBrewer", "XML"))
library(wordcloud)
```

```
## Loading required package: RColorBrewer
```

```
wordcloud(text, type="text",
        lang="english", excludeWords = NULL,
        textStemming = FALSE,  colorPalette="Dark2",
        max.words=200)
```

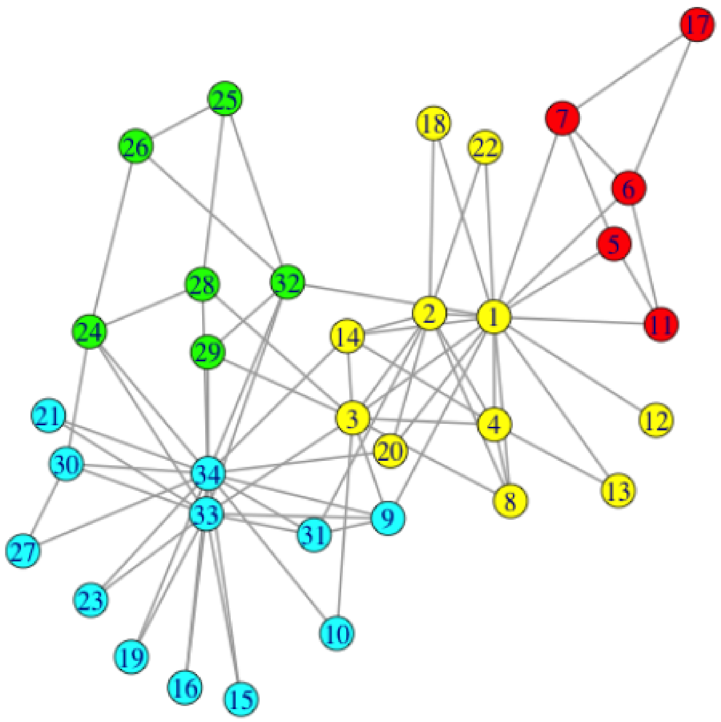
```
## Loading required package: tm
## Loading required package: NLP
```



Fast community unfolding

- Heuristic method for greedy modularity optimization
- Find partitions with high modularity
- Multi-level (multi-resolution) hierarchical scheme
- Scalable

```
Assign every node to its own community;
repeat
    repeat
        For every node evaluate modularity gain from removing node from
        its community and placing it in the community of its neighbor;
        Place node in the community maximizing modularity gain;
    until no more improvement (local max of modularity);
    Nodes from communities merged into "super nodes" ;
    Weight on the links added up
until no more changes (max modularity);
```



Walktrap community

Consider random walk on graph. At each time step walk moves to NN uniformly at random.

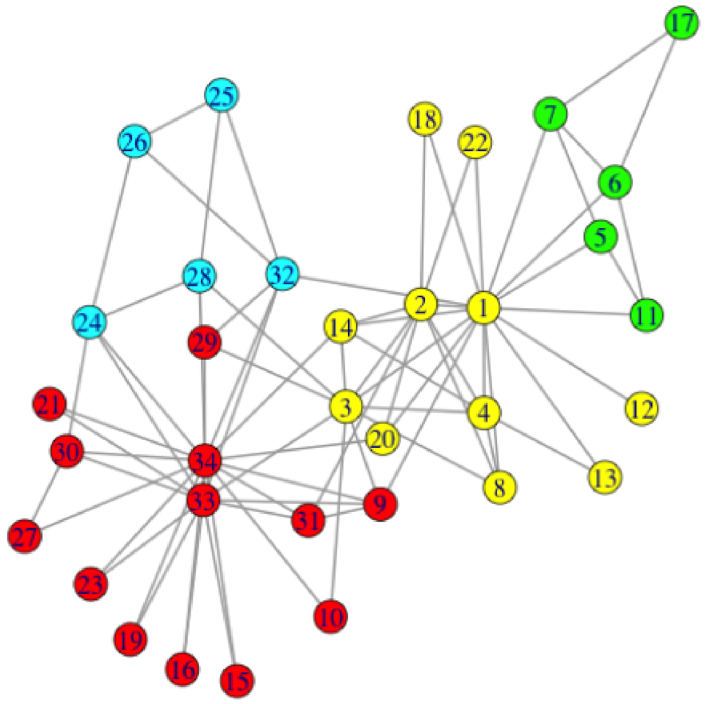
Distance between nodes $r_{ij}(t)$ is computed as probability P_{ij}^t to get from one to another in t steps.

Computations:

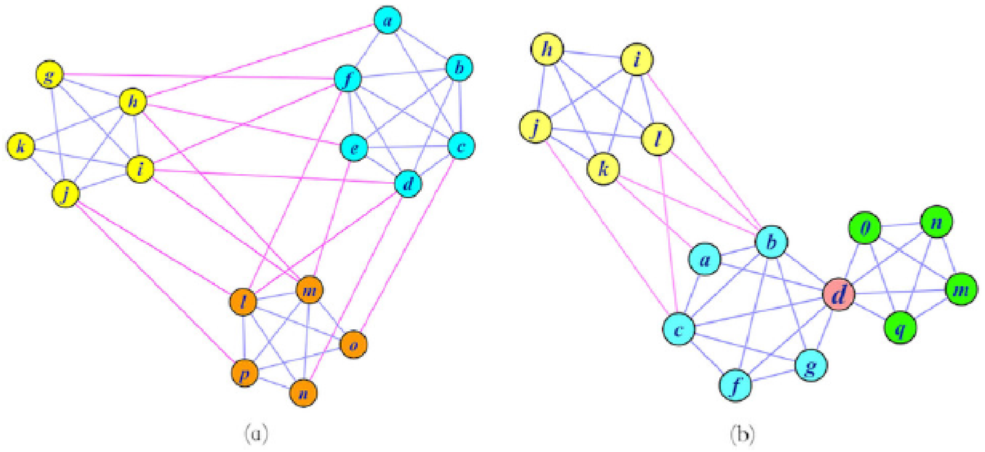
- exact, matrix multiplication
- approximate, random walk simulation

Vertex clustering (agglomerative algorithm).

```
Compute random walk distance between adjacent vertices;
for n-1 steps do
    choose two "closest" communities and merge them ;
    update distance between communities
end
```



Overlapping communities


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
(a)
h
i
j
k
l
a
b
c
d
f
g
0
n
m
q
(b)