Social Network Analysis

An individual project on community detection in a network.

by: Inderpreet Singh RANA

Table of Contents

Community Detection:

In social network analysis, a community is a subset of nodes in a graph that are more likely to be related to each other than the remainder of the network.

In the context of graphs, a community can be described as a subset of nodes that are strongly linked to one another but loosely connected to nodes in other communities in the same network.

Let me illustrate this with an example. Consider how we use social media platforms like Facebook, Instagram, and Twitter to communicate with others. We eventually become connected with individuals from various social groups after a period. These social circles can include a group of relatives, classmates, coworkers, and so on.

One of the most important tasks in network analysis is detecting communities in a network. We may have millions of nodes and edges in a huge network, such as an online community. It becomes a tremendous undertaking to detect communities in such networks.

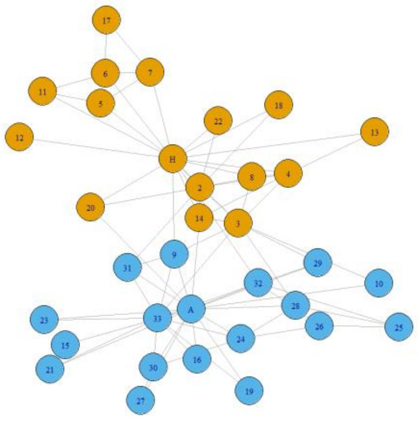
As a result, community detection algorithms that can divide the network into many communities are required.

Dataset:

Zachary's karate club network: Social network between members of a university karate club. The edge weights are the number of common activities the club members took part of.

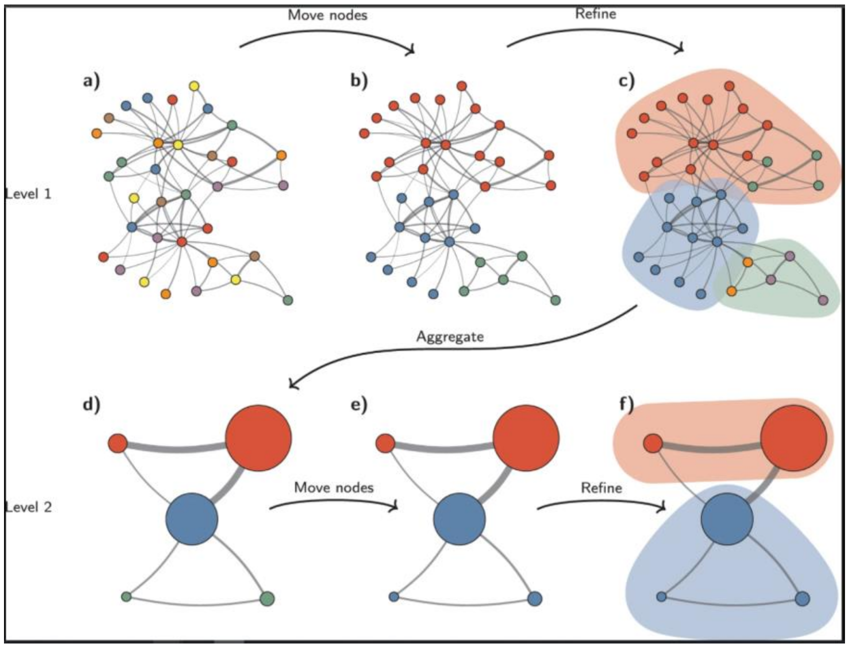
The Network:

Before we start the community detection, lets visualize the network:



Leiden algorithm:

Leiden algorithm guarantees that communities are well connected. The Leiden algorithm is partly based on the previously introduced smart local move algorithm, which itself can be seen as an improvement of the Louvain algorithm. The Leiden algorithm also takes advantage of the idea of speeding up the local moving of nodes and the idea of moving nodes to random neighbors.



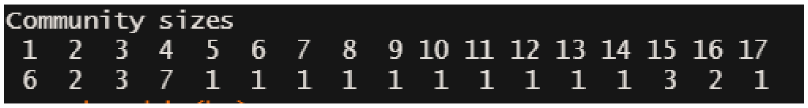
The Leiden algorithm starts from a singleton partition (a).
The algorithm moves individual nodes from one community to another to find a partition (b), which is then refined (c). An aggregate network (d) is created based on the refined partition, using the non-refined partition to create an initial partition for the aggregate network. For example, the red community in (b) is refined into two subcommunities in (c), which after aggregation become two separate nodes in (d), both belonging to the same community. The algorithm then moves individual nodes in the aggregate network (e). In this case, refinement does not change the partition (f). These steps are repeated until no further improvements can be made.

~Reference https://www.nature.com/articles/s41598-019-41695-z#Sec4

Leiden algorithm implementation:

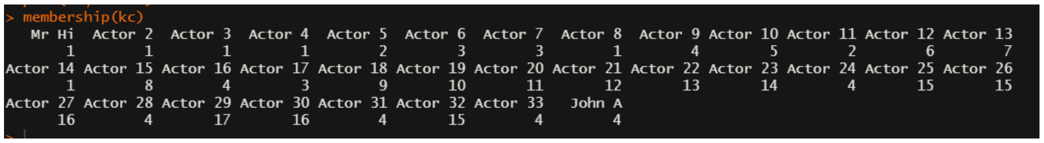
Now we implement this method on our karate club dataset:

we use the *cluster_leiden* function of the *igraph* library in R to find the community structure which gives us the community sizes in our graph:



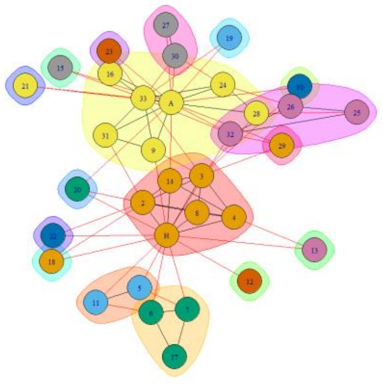
```
Community sizes
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
 6  2  3  7  1  1  1  1  1  1  1  1  1  1  3  2  1
```

We can further use the *membership* function on our *cluster_leiden* object to identify the main actors:



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

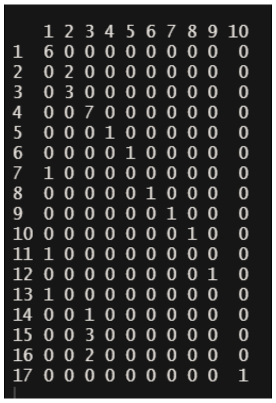
We can now simply plot the 17 communities that we identify:



We can compute the network's density and weighted density for comparison, and we get a fewer number of larger communities using Leiden with CPM at the weighted density, as expected.

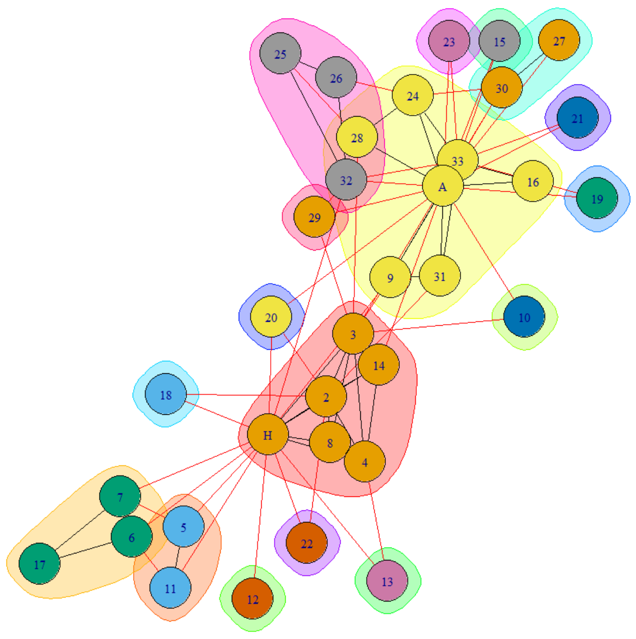
Density = 0.4117647

We can also look at the contingence table to understand the data and the communities as well:



```
   1 2 3 4 5 6 7 8 9 10
1  6 0 0 0 0 0 0 0 0  0
2  0 2 0 0 0 0 0 0 0  0
3  0 3 0 0 0 0 0 0 0  0
4  0 0 7 0 0 0 0 0 0  0
5  0 0 0 1 0 0 0 0 0  0
6  0 0 0 0 1 0 0 0 0  0
7  1 0 0 0 0 0 0 0 0  0
8  0 0 0 0 0 1 0 0 0  0
9  0 0 0 0 0 0 1 0 0  0
10 0 0 0 0 0 0 0 1 0  0
11 1 0 0 0 0 0 0 0 0  0
12 0 0 0 0 0 0 0 0 1  0
13 1 0 0 0 0 0 0 0 0  0
14 0 0 1 0 0 0 0 0 0  0
15 0 0 3 0 0 0 0 0 0  0
16 0 0 2 0 0 0 0 0 0  0
17 0 0 0 0 0 0 0 0 0  1
```

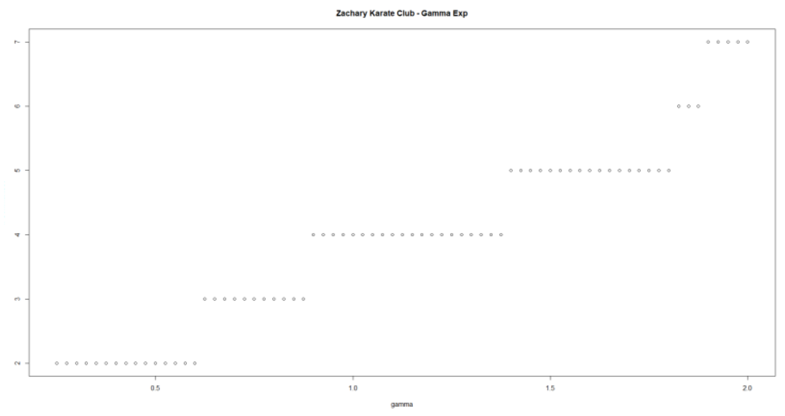
It might be instructive to check what happens with the default cluster leiden parameters on the unweighted version of the karate club network before moving from CPM to modularity:



Using modularity with resolution parameters in cluster_leiden()

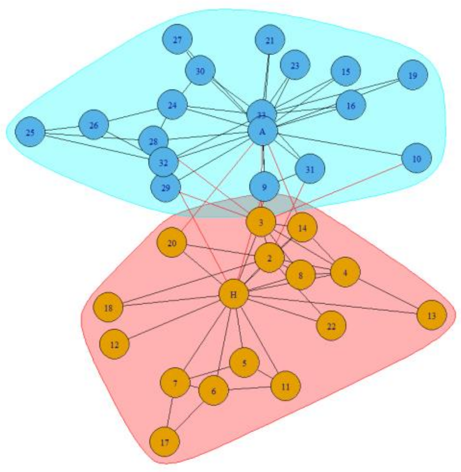
The modularity of the network has a resolution limit, which implies that increasing the total size of the network can lead to larger communities even when this isn't the case. The number of groups is indirectly determined by the default resolution parameter gamma=1, however smaller (higher) gamma will prefer to choose fewer (larger) groups on the same network data set. If you don't specify gamma, you'll still be using the default value.

We can explore the communities in our network with different gamma levels and plot the results to visualize how the gamma value will impact the community structures:

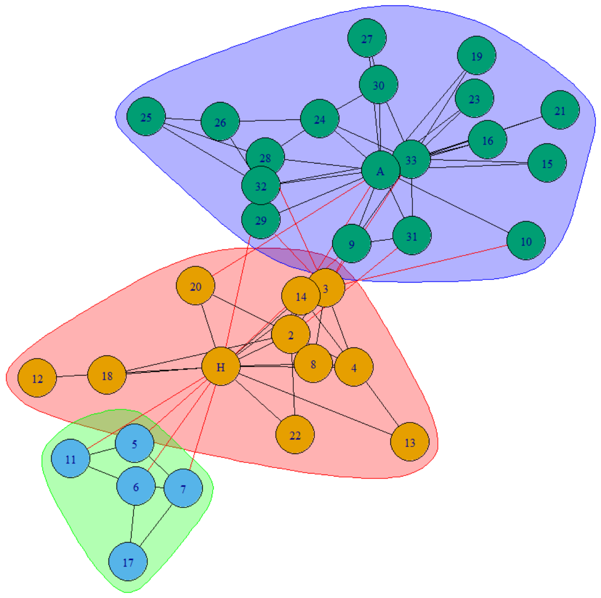


The above plot is not much helpful, so we can compare a few of the Leiden results at different resolution parameters by way of contingency tables.

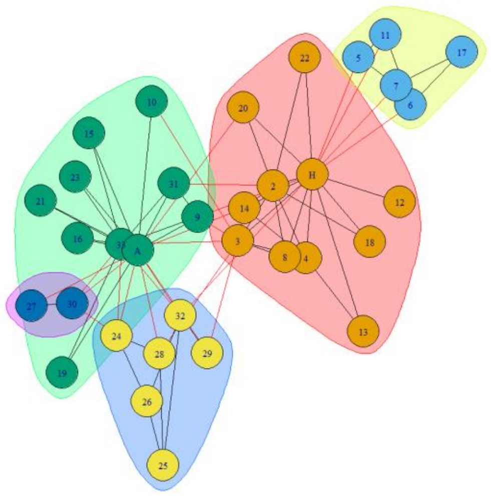
```
kc2 <- cluster_leiden(karate, objective_function = "modularity",
                    n_iterations = 3, resolution_parameter = 0.5)
plot(kc2,karate)
```



```
kc2 <- cluster_leiden(karate, objective_function = "modularity",
                   n_iterations = 3, resolution_parameter = 0.75)
plot(kc2,karate)
```



```
kc2 <- cluster_leiden(karate, objective_function = "modularity",
                   n_iterations = 3, resolution_parameter = 1.5)
plot(kc2,karate)
```



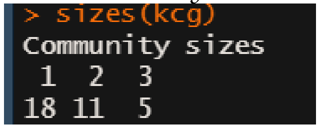
Fast-greedy community detection:

Fast-greedy is a modularity-based method - it works by trying to build larger and larger communities by adding vertices to each community one by one and assessing a modularity score at each step. The modularity score is a measure of how interconnected the edges are within a community versus between them. In igraph it can be performed by applying the function fastgreedy.community() to the graph object. Printed to console are the community memberships of each vertex.

Fast Greedy implementation:

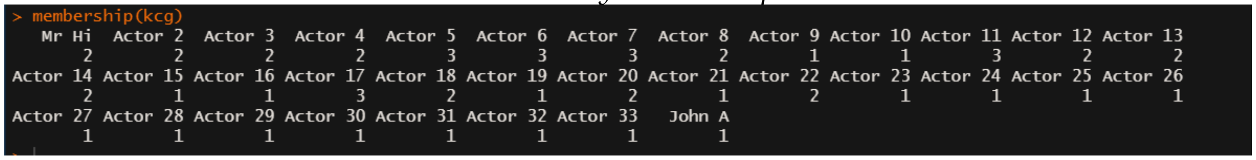
Using the fast greedy method, we get the following community sizes and memberships:

Community size:



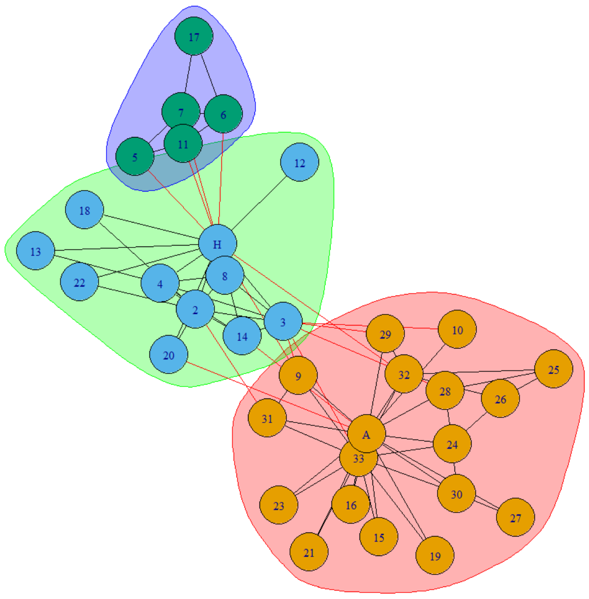
```
> sizes(kcg)
Community sizes
 1  2  3
18 11  5
```

Community memberships:



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

Lets plot the communities using fast greedy meathod:



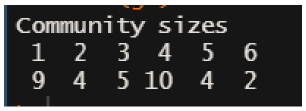
We see that the fast greedy method has detected 3 communities compared to 17 in the *cluster_leiden* method with 2 communities (Green and Blue) slightly overlapping with each other. This means that these 2 communities may be indirectly connected to each other and may have more influence over one another compared to the red community.

Edge-betweenness:

In contrast to fast-greedy, the edge-betweenness is a divisive method - it works by dividing the network into smaller and smaller pieces until it finds edges that it perceives to be 'bridges' between communities.

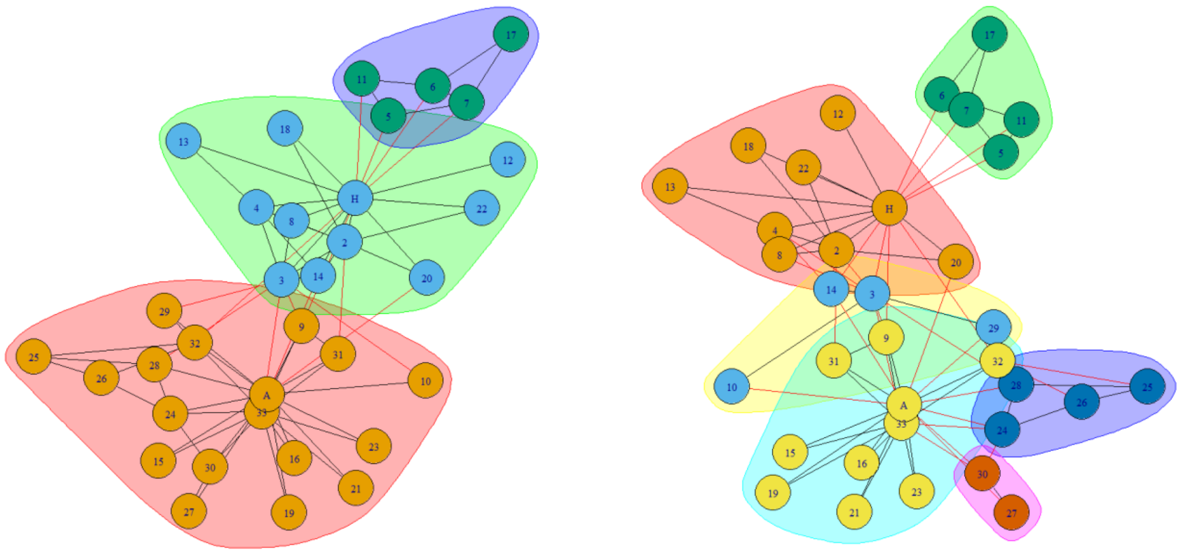
Edge-betweenness implementation:

To compare the difference in edge-betweenness and fast greedy method, we plot both communities' side by side:



```
Community sizes
 1  2  3  4  5  6
 9  4  5 10  4  2
```

Community sizes using for edge-betweenness



Left: Greedy method, *Right: edge-betweenness*

We can see the difference visually on how these 2 methods compare to each other,

In edge-betweenness method, we can see the communities are closely linked to each other and it was able to find sub-communities within the same network, whereas the greedy method took the largest communities instead of finding smaller sub-communities.

Walk-trap:

The walktrap method uses short random walks on a graph to detect communities, It assumes that random walks within a graph should get “trapped” within the communities.

A random walk process begins on a selected node and moves to another node chosen randomly and uniformly from its neighbors, and then proceeds to a next node in the same way, with the number of steps specified as the walk length. The length of the walk must be short enough to not be trivial, but long enough to gather community

information. From the network, a transition matrix $\mathbf{P}_{N\times N} = \{p_{ij}\}$ is formed where p_{ij}=$\mathrm{x}_{\mathrm{ij}}\mathrm{d}_{\mathrm{i}}$ is the transition probability from n_i to n_j at any step. For a random walk of length m starting at n_i, the probability of ending at n_j is $\mathrm{P}_{\mathrm{mij}}$. From here a distance measure, $D(i, j)$, between n_i and n_j is calculated:

$$D(i,j) = \sqrt{\sum_{n=1}^{N} \frac{\left(P_{in}^{m} - P_{jn}^{m}\right)^2}{d_n}}.$$

This is then generalized to a distance measure between communities:

where $\mathrm{P}_{\mathrm{mC_kj}}$=$1_{\mathrm{N_k}}$$\sum_{\mathrm{i\in C_k}}$$\mathrm{P}_{\mathrm{mij}}$ is probability of going from C_k to the n_j, where $n_j \notin C_k$, in m steps.

The walktrap algorithm proceeds as follows:

1. Begin with N communities (every node is its own community) and calculate the distances D(i, j) for each pair.
2. Use Ward’s criterion to merge two communities (minimize the average squared distance between each node and its community):

$$\sigma_k = \frac{1}{N}\sum_{k}\sum_{i\in C_k} D^2\left(C_k, i\right).$$

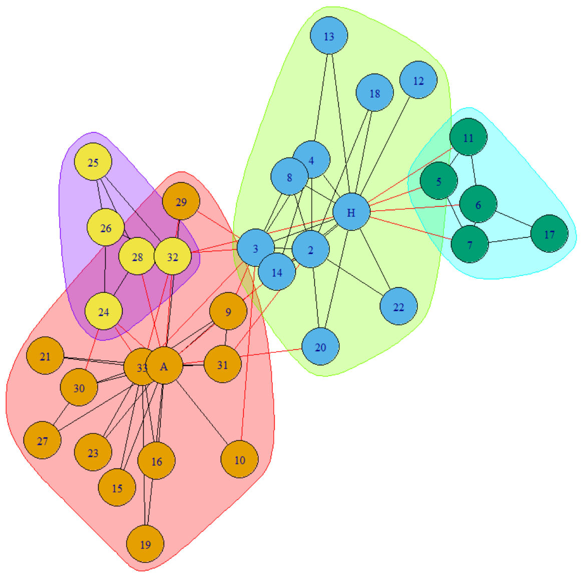
3. Update the distances between adjacent communities.
4. Repeat steps 2 and 3 until all nodes are in the same community.

Finally, the best choice of K is selected from the sequence of communities with increasing k based on the maximum modularity.

Walktrap implementation:

We use the *walktrap.community()* function to detect communities in the karate club network using the walktrap algorithm.

a <- walktrap.community(karate, weights = E(karate)$weight, steps = 4, merges = TRUE, modularity = FALSE)



#Reference: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6103523/

References:

https://people.duke.edu/~jmoody77/

https://towardsdatascience.com/community-detection-in-r-using-communities-of-friends-characters-2161e845c198

https://igraph.org/r/doc/communities.html

https://psych-networks.com/r-tutorial-identify-communities-items-networks/

https://www.nature.com/articles/s41598-019-41695-z

https://campus.datacamp.com/courses/network-analysis-in-r