3803ICT
Big Data Analysis

Lab 09 – Network Data Analytics

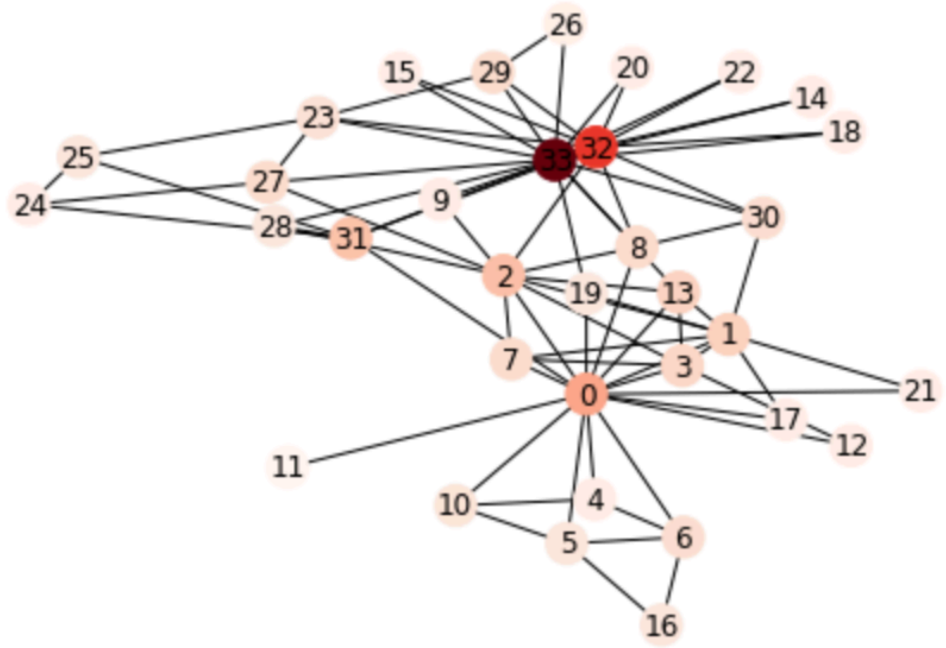
Table of Contents

Complete the code with TODO tag in the Jupyter notebooks.

1. Centrality Analysis

In this exercise, you will implement the pagerank centrality.

```
Name: Zachary's Karate Club
Type: Graph
Number of nodes: 34
Number of edges: 78
Average degree:   4.5882
```

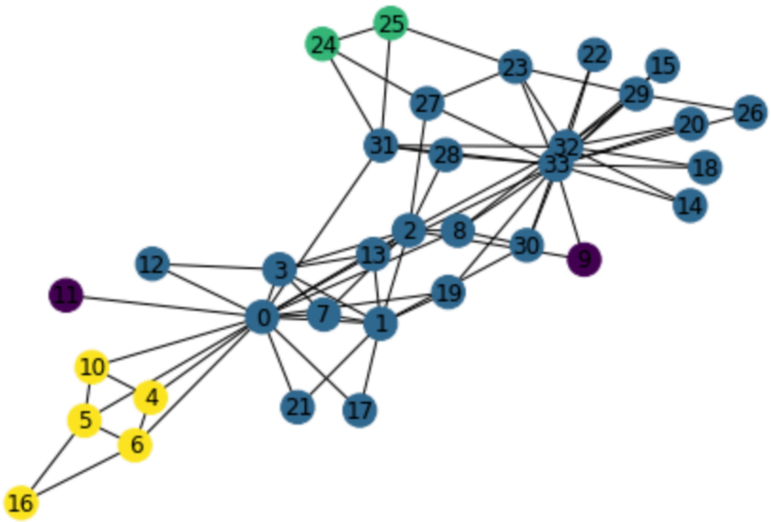


2. Community Analysis

2.1. Clique Percolation Method

One well-known algorithm for detecting overlapping communities is called the Clique Percolation Method (CPM).

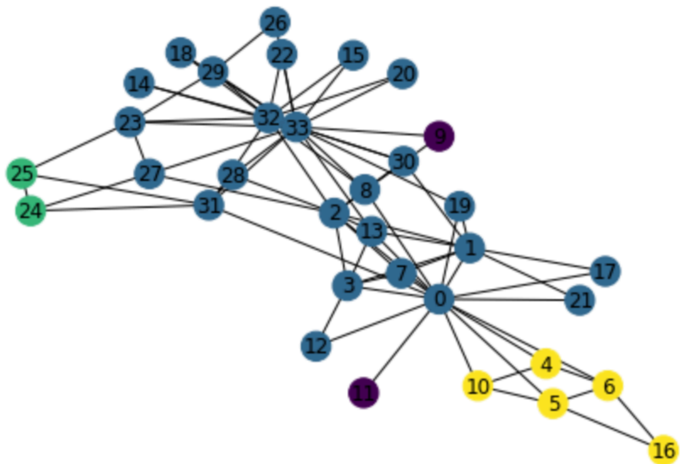
```
Name: Zachary's Karate Club
Type: Graph
Number of nodes: 34
Number of edges: 78
Average degree:   4.5882
---0.000205993652344 seconds---
```



2.2.Efficient Implementation

That implementation is correct but expensive---it requires O(N^2) clique comparisons, where N is the number of cliques (which is often much larger than the number of nodes!). If we use a python dictionary to index which nodes belong to which cliques, then we can easily compare only those cliques that share at least one node in common. This implementation is a bit longer but should be more efficient.

```
---0.0001540184021 seconds---
```



2.3. Test with large dataset

Now we test with a real large-scale network data at https://snap.stanford.edu/data/com-Amazon.html

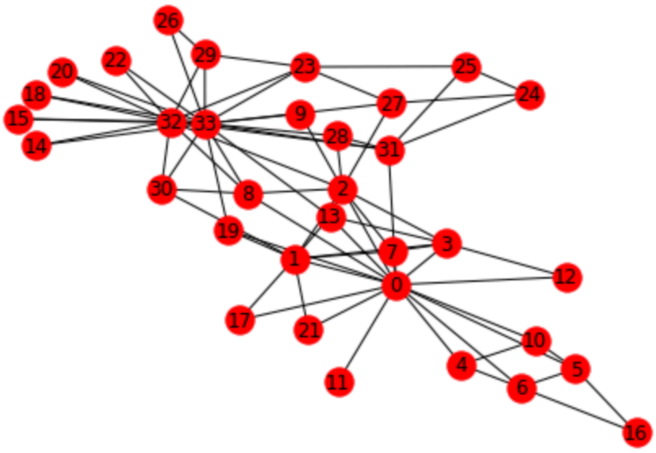
```
Name:
Type: Graph
Number of nodes: 334863
Number of edges: 925872
Average degree:   5.5299
---0.0001220703125 seconds---
---0.000126123428345 seconds---
```

3. Information Diffusion

It is also known as graph activation process, e.g. http://ncase.me/crowds/

Further readings:

- https://stackoverflow.com/questions/31815454/animate-graph-diffusion-with-networkx
- https://stackoverflow.com/questions/27475211/animating-a-network-graph-to-show-the-progress-of-an-algorithm/



3.1. Diffusion process

Now we implement the diffusion process. Each active node will cause other nodes in the graph to become active over time. The diffusion rule is that a node gets active if at least a certain percentage of its neighbours become active. The process continues until convergence (i.e. has no new node activated).

OPTIONAL: Can you implement a data visualization to illustrate the diffusion process?

```
diffusion(G, {0,1})
```

```
{0, 1, 2, 3, 7, 9, 11, 12, 13, 17, 19, 21}
```

3.2. Influence maximization

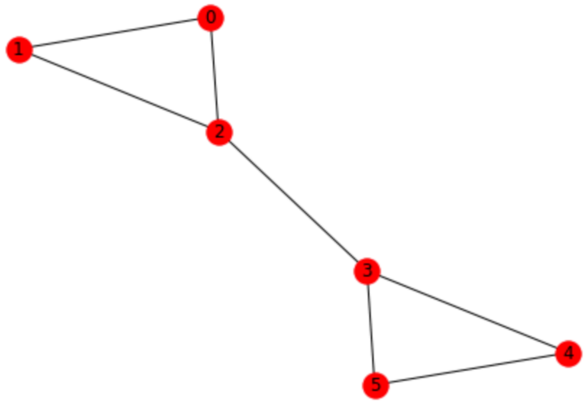
Now we find a minimal set of seeds that maximize the influence (i.e. the number of active nodes). The influence maximization problem is NP-hard in general. Here, we use a greedy algorithm that iteratively chooses a seed such that the gain of influence is maximal.

```
seeds = greedy(G,3)
print(seeds)
print(utility(G, seeds))
```

```
set([0, 33, 4])
34
```

4. Graph Modularity and Louvain Algorithm (OPTIONAL)

In this exercise, we compute the modularity measure of a graph. If you haven't installed networkx package, please install. First we create a small dataset and manually assign the community label to each node.



4.1. Compute Modularity

Now we compute the modularity of the graph given the current community assignment.

```
def compute_modularity(G):
    '''
    TODO: compute the modularity of a networkx graph
    HINTS:
    + The community label of a node can be accessed by G[node_id]['community']
    + The degree of a node: G.degree[node_id]
    + The neighbors of a node: G.neighbors(node_id)
    + Number of edges between 2 nodes: G.number_of_edges(node_1, node_2)
    '''
    m = len(G.edges)
    Q = 0
    ...
    return Q/(2*m)

compute_modularity(G)
```

```
0.35714285714285715
```

4.2. Naïve Louvain algorithm

Now we implement phase 1 of Louvain algorithm, in which we partition the nodes to maximize the modularity.

```
array([1, 1, 1, 5, 5, 5])
```

Now we implement the phase 2 of Louvain algorithm, in which we merge the nodes within the same community to a single node and create edges between communities.

```
(array([[0, 1],
        [1, 0]]), array([1, 1]))
```

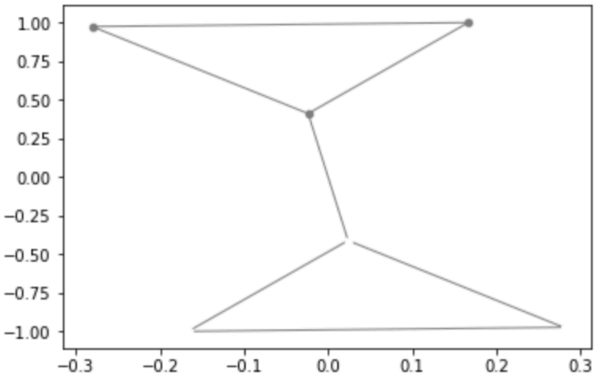
Now we can implement the full Louvain algorithm:

```
Level 0 partition: [0 1 2 3 4 5]
Level 1 partition: [1 1 1 5 5 5]
Level 2 partition: [1 1]
```

4.3. Efficient Louvain algorithm

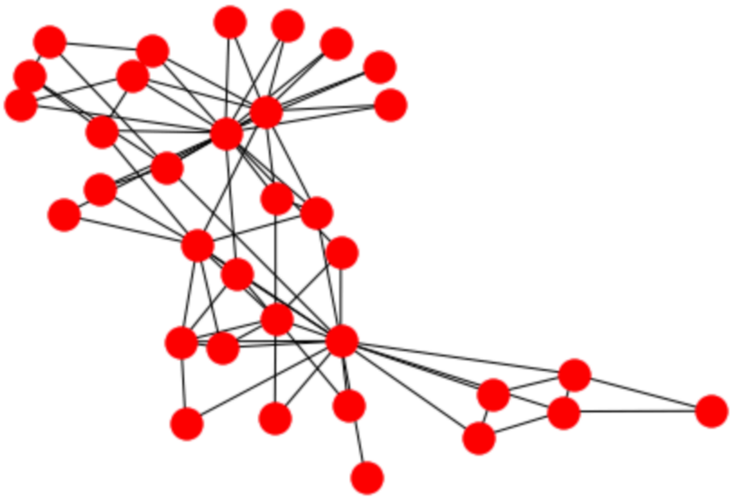
The naive Louvain algorithm is not efficient. It takes O(n3)O(n3) . There are some improvements in the literature

http://www.ijcee.org/vol8/927-A023.pdf
https://www.cs.upc.edu/~CSN/slides/07communities.pdf
https://en.wikipedia.org/wiki/Louvain_Modularity
https://www.quora.com/Is-there-a-simple-explanation-of-the-Louvain-Method-of-community-detection
http://arxiv.org/abs/0803.0476
https://stackoverflow.com/questions/22070196/community-detection-in-networkx
For simplicity, we will demo the existing implementations. You can try to improve the final visualization further.

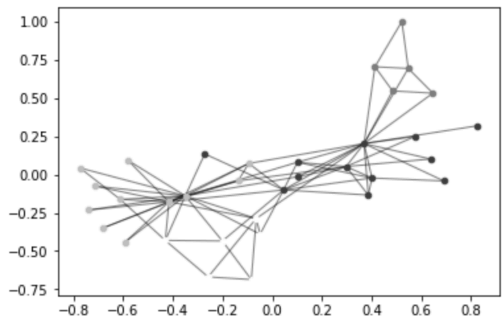


4.4. Efficiency comparison

We load a larger network and compare the running time of the two implementations.



```
--- 0.00814485549927 seconds ---
{0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 0, 8: 2, 9: 0, 10: 1, 11: 0, 12: 0, 13: 0, 14: 2, 15: 2, 16: 1, 17: 0,
18: 2, 19: 0, 20: 2, 21: 0, 22: 2, 23: 3, 24: 3, 25: 3, 26: 2, 27: 3, 28: 3, 29: 2, 30: 2, 31: 3, 32: 2, 33: 2}
```



Now we load an even larger network: https://snap.stanford.edu/data/email-Eu-core.html

```
Name:
Type: Graph
Number of nodes: 1005
Number of edges: 16706
Average degree:  33.2458
```

```
--- 0.617053985596 seconds ---
```