Implementation of Girvan-Newman Algorithm (Python 3): Readme

MSBD5008: Introduction to Social Computing (Assignment 1)

MSc in Big Data Technology, The Hong Kong University of Science and Technology

For unweighted and undirected networks, the Girvan-Newman algorithm is considered to be one of the most accurate algorithms to identify and breakdown networks in a hierarchical fashion. When juxtaposed with the network modularity (Q) – a measure of how well a network is partitioned into communities – one can identify the optimal number of clusters/communities in the network.

For this assignment, the following two functions are performed with respect to undirected and unweighted graphs:

1) Implementation of the Girvan-Newman algorithm and output the resultant hierarchical decomposition of the network.
2) Calculation of the modularity and output the corresponding optimal cluster structure.

Getting Started

For this assignment it has been assumed that the provided network is *undirected* and *unweighted*. Also, the node names in the network is assumed to be numeric, each pertaining to its corresponding row in the input adjacent matrix (starting from 0).

Prerequisites

This Python3 implementation requires $pandas$, sys and $networkx$. While the first two come along with $Anaconda$, $networkx$ has to be installed separately via the following command:

$$\boldsymbol{pip\ install\ networkx}$$

Implementation

Required User Details

To run the code, create an input file ($input.txt$) whose first line contains the number of nodes in the network and its remaining lines contain the network's adjacency matrix (see example). It should be kept in mind that all inputs are space-separated by default. To use another input text file name or delimiter, please change the variables accordingly in the main function of the program.

By default, the required outputs are printed onto the console. If you require them to be saved onto a file ($output.txt$), please uncomment the lines of code right after library importation.

Table 1: Example of required input format (space separated by default)

```
9
0 1 1 1 0 0 0 0 0
1 0 1 1 0 0 0 0 0
1 1 0 1 0 0 0 0 0
1 1 1 0 0 0 0 0 1
0 0 0 0 0 1 1 1 1
0 0 0 0 1 0 1 1 0
0 0 0 0 1 1 0 1 0
0 0 0 0 1 1 1 0 0
0 0 0 1 1 0 0 0 0
```

Main Function and Global Variables

The main part of the program follows the below steps:

1) Read the input file as a pandas DataFrame. The first line is omitted as the number of nodes is taken from the adjacent matrix directly.
2) The adjacent matrix is converted into a graph using *networkx*.
3) The following information are stored as global variables:
    a. NUM_NODES – number of nodes in the original network
    b. $NETWORK\_DEGREE$ – degree of each node in the original network.
        i. This is calculated using the $get\_node\_degree()$ function which obtains the row-wise count of the number of non-zero elements in the adjacency matrix.
    c. NUM_EDGES – the total number of edges in the original network.
    d. $ALL\_NODE\_COMBINATIONS$ – a pair-wise combination of all nodes in the network
        i. This is used to calculate the shortest paths between every combination of node pair.
        ii. This is obtained by performing a cartesian cross product of all nodes via the $get\_cartesian\_product()$ function.
        iii. Nodes crossed with themselves (i.e., $[0,0]$ etc.) are removed.
        iv. As the network is considered to be undirected, duplicate rows ($[0,1]$ = $[1,0]$) are removed.
    e. $CLUSTER\_MODULARITY$ – tracks the number of clusters in each hierarchical decomposition and its corresponding modularity.
4) Once the global variables are declared, the $GN\_hierarchical\_decomposition()$ function is called.
5) The required hierarchical breakdown (along with cluster numbers and modularity) is then printed in the required format.
6) The index of the highest modularity is found to retrieve the corresponding optimal community structure.

Function: $get\_shortest\_path(graph, start\_node, end\_node)$

Created with help from: https://pythoninwonderland.wordpress.com/2017/03/18/how-to-implement-breadth-first-search-in-python/

This function returns the shortest path between any two nodes in a network. It uses the concept of Breadth First Search (BFS) to do so. By maintaining a list of the visited nodes and appending necessary neighbours, it returns the path of all nodes traversed to reach the destination node. The major difference between this function and normal BFS is that here the queue maintains a list of all possible paths instead of nodes. If no path exists between two nodes, it returns a null set.

After every hierarchical decomposition of the Girvan-Newman algorithm, this function is called to find new paths between every node pairs in the $ALL\_NODE\_COMBINATIONS$ global variable.

Function: $get\_edge\_betweenness(graph)$

Given a network, the edge betweenness is calculated as the number of times a shortest path passes over a given edge, or node pair. By using the $get\_shortest\_path()$ function for all node combinations, every required shortest path is obtained. The betweenness of every existing edge is then the number of times the nodal pair occurs within the list of shortest paths.

For example, if the list of shortest paths is $[[1,2], [\mathbf{2}, \mathbf{3}, 4,7,5], [5, \mathbf{3}, 1, \mathbf{2}]]$, then the edge betweenness between nodes $(2,3)$ is 2. Through this function a dictionary of all the existing network node pairs and their corresponding edge betweenness is obtained.

Function: $get\_max\_edge\_betweenness(betweenness)$

This is a simple function to extract the node pairs (key of the betweenness dictionary) corresponding to the maximum betweenness value. If more than one edge has the highest betweenness, all those node pairs are returned.

Function: $get\_community\_modularity(adjacent\ matrix, clusters)$

Modularity (Q) is a measure of how well a network is partitioned into communities. It is positive if the number of edges within groups exceeds the expected number, thus having a possible community structure. Usually, $0.3 < Q < 0.7$ suggests a significant community structure.

Given a cluster/community S, the modularity of the network G and cluster S is shown below where m is the total number of edges in the original network (NUM_EDGES) and k_i and k_j are degrees of node i and node j in the original network ($NETWORK\_DEGREE$):

$$Q(G, S) = \frac{1}{2m} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left(A_{ij} - \frac{k_i k_j}{2m} \right)$$

Given the updated network adjacency matrix, this function returns the modularity for a given cluster number by directly updating the same in the $CLUSTER\_MODULARITY$ global variable.

Function: $GN\_hierarchical\_decomposition(graph)$

This function accumulates all the previously described functions into a wholesome structure to get all the necessary outputs. Till there are edges remaining in the network, the function performs the following steps:

1) Get the edge betweenness for every present edge in the network.
2) Extract all the edges having the maximum edge betweenness.
3) Remove all the edges obtained in the previous step.
4) Obtain a list of all connected communities (i.e., clusters) in the updated network after removing edges (via $nx.connected\_components$).
5) Output the obtained clusters from the previous step in the desired format.
6) Calculate the modularity and cluster numbers for the network – directly appended to the global variable.
7) Repeat loop till no edges are present in the network.

Results

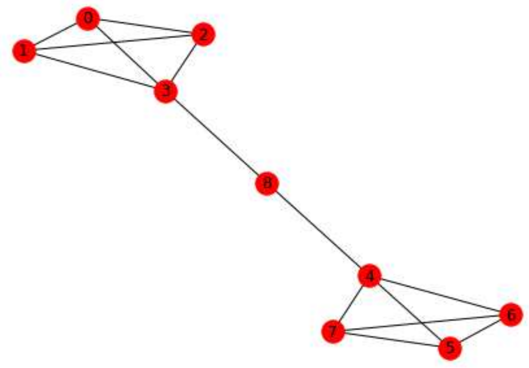
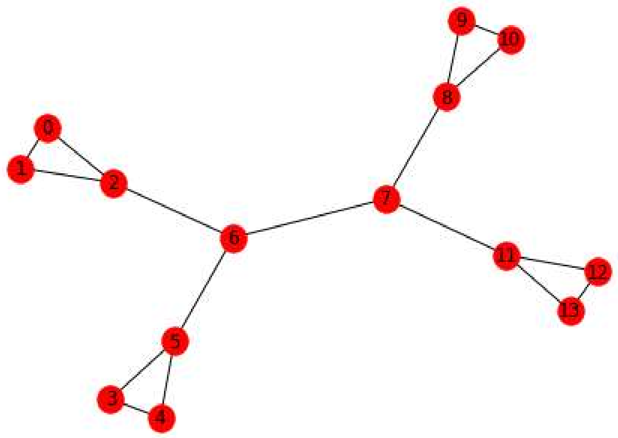
The implementation was tested on two networks having 9 nodes and 14 nodes respectively. The output was verified to be correct. Table 2 shows the structure of the input network and the resultant output.

Running the Code

Place the $input.txt$ file in the required format in the same folder as the code. Open the python3 command prompt and run the following command in the proper directory.

$$\boldsymbol{python3 < codefile\ name >.py}$$

Table 2: Tested outputs for different network structures

Original Network Structure	Required Output
	network decomposition: ([0, 1, 2, 3], [4, 5, 6, 7], [8]) ([0], [1], [2], [3], [4], [5], [6], [7], [8]) 3 clusters: modularity 0.4209 9 clusters: modularity -0.1148 optimal structure: ([0, 1, 2, 3], [4, 5, 6, 7], [8])
	network decomposition: ([0, 1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12, 13]) ([0, 1, 2], [3, 4, 5], [8, 9, 10], [11, 12, 13], [6], [7]) ([0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]) 2 clusters: modularity 0.4412 6 clusters: modularity 0.5208 14 clusters: modularity -0.0744 optimal structure: ([0, 1, 2], [3, 4, 5], [8, 9, 10], [11, 12, 13], [6], [7])

Author

Rohini BANERJEE – HKUST Student ID: 20543577

References

1) https://neo4j.com/docs/graph-algorithms/current/algorithms/betweenness-centrality/
2) https://github.com/ZwEin27/Community-Detection/blob/master/communities.py
3) https://github.com/kjahan/community/blob/master/cmty.py
4) https://pythoninwonderland.wordpress.com/2017/03/18/how-to-implement-breadth-first-search-in-python/
5) https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/modularity.pdf