Graph Theory Homework 1

Alperen Kantarcı 504191504

Istanbul Technical University, Turkey

1 Profiling a brain network using different centrality measures

For Step 1,2 and 3 a script named "Question1_except4.py" is coded. You can either give a path of adjacency matrix or choose random matrix. More details to test the script is given in README.txt file.

Step 1: network binarization Edges with weights lower than 0 are made 0. After the binarization, adjacency matrix is shown to user immediately. You can see the binarized version of thresholded matrix in Fig. 1 Following command makes this thresholding and binarization.

```
binarized_mat = np.where(adj_mat>0, 1, 0)
```

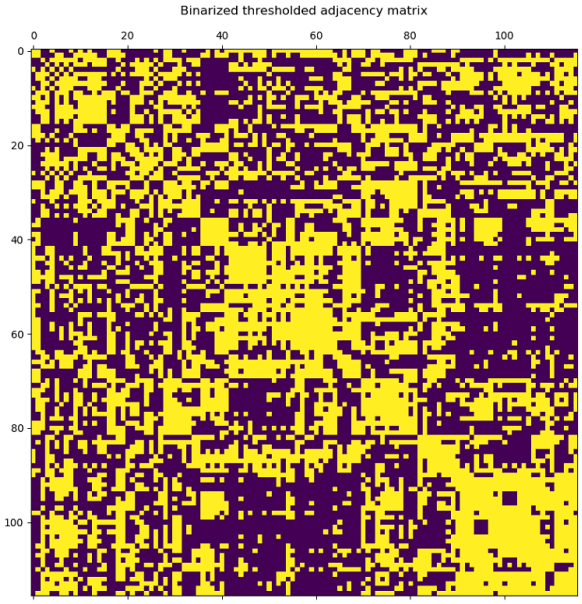

Fig. 1: *Thresholded and binarized adjacency matrix of C.mat matrix.*

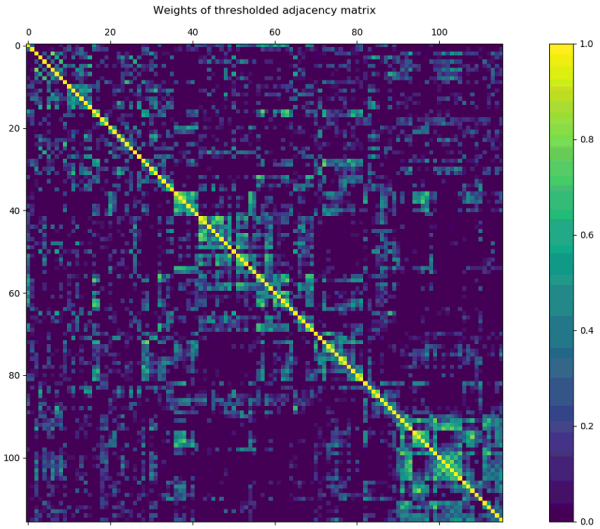

Fig. 2: *Weighted version of the C.mat matrix that thresholded at 0.*

You can also see the code screenshot in Fig. 3 to reading, random graph creation and binarization.

Step 2: node centralities

2.1 For every node in the graph GetDegreeCentr(),GetClosenessCentr and GetEigenVectorCentr() function is used to calculate the centrality measure. All the measures saved into the lists and plotted as in Fig. 4. Since we have soo many nodes please look the graph from original image. In report it looks small and texts may be unreadable.

You can also see the centrality calculation code in Fig. 5

2.2 There are some vital nodes but in overall there is no node that gets attention at first look. Nodes 28, 55, 75 and 105 shows higher centralities in all three metrics. Lesioning these nodes can affect the functionality of the brain because these nodes takes a role of junction and if you close a junction then it would affect the flow. Brain may continue to work but efficiency and functionality would hurt. Especially in degree centrality, immediate neighbors increase the centrality measure and lesioning these more central nodes according to the degree centrality would affect much more immediate neighbors (parts that make connections) than lesioning any other node. Also it can be deducted that in general if a node shows higher centrality in a metric, it also shows higher centrality in other metrics. I couldn't detect much nodes that shows incredibly high value in one measure but shows low centrality in other metric.

Step 3: centrality distributions

3.1 By using the metric values that I found in Step 2. I have created the histogram of the centrality measures. In Fig. 6

```python
def read_adj_mat(path=""):
    mat = scipy.io.loadmat(path)
    adj_mat = mat['A']
    binarized_mat = np.where(adj_mat>0, 1, 0)
    return  adj_mat , binarized_mat

# Create random binarized matrix with given arguments
def create_random_mat():
    nodes = 100
    edges = 2555
    bin_graph = s.GenRndGnm(s.PUNGraph, nodes, edges)
    adj_mat = np.zeros((nodes,nodes))
    np.fill_diagonal(adj_mat, 1)

    for edge in bin_graph.Edges():
        random_weight = np.random.uniform(-0.2,1)
        adj_mat[edge.GetSrcNId(),edge.GetDstNId()] = random_weight
        adj_mat[edge.GetDstNId(),edge.GetSrcNId()] = random_weight
    return bin_graph, adj_mat

parser = argparse.ArgumentParser()
parser.add_argument("--path", help="random graph else relative path of .mat file of the adjacency matrix")
parser.add_argument("--random", help="True if you want to test on random graph")
args = parser.parse_args()
if not any(vars(args).values()):
    raise Exception("Please give an argument. Give -h to see all the arguments.")
if(args.random):
    bin_graph, adj_mat = create_random_mat()
    binarized_mat = np.where(adj_mat>0, 1, 0)
else:
    # Step 1
    if args.path == "":
        raise Exception("Path is not given. For random graph please give \"--random True\" argument")
    adj_mat, binarized_mat  = read_adj_mat(args.path)
    num_nodes = binarized_mat.shape[0]
    bin_graph = s.TUNGraph.New()
    [bin_graph.AddNode(i) for i in range(binarized_mat.shape[0])]
    for i in range(binarized_mat.shape[0]):
        for j in range(i+1,binarized_mat.shape[1]):
            if binarized_mat[i,j]:
                bin_graph.AddEdge(i,j)
```

Fig. 3: *Adjacency matrix reading, random graph creation and binarization.*

3.2 I see three nearly Gaussian distributions in all three centralities. Also it looks like graphs have little right skeweness which shows nodes with higher probabilities are more probable. Also frequencies show us there are 23-24 nodes that give median probability in all closeness metrics but only 2-3 nodes give highest probability in each metrics. Also each centrality measure scattered at different probabilities therefore no overlapping can be seen in this graph. But more important information is distribution of the values, not the centrality metric value when one compare three different metrics.

Step 4: brain connectome percolation

4.1 All the steps implemented in Python3 environment therefore integration to the module is not possible unlike the C++. Therefore a function named "findPercolationThreshold" is implemented by taking an adjacency matrix and returning percolation threshold. This function only takes adjacency matrix therefore can be imported to any project that works with SNAP Python. Because of the usage of the weighted graphs of the SNAP. This function doesn't take any SNAP graph but it creates it's own SNAP graph via given adjacency matrix. Then this graph used for the calculations. This function can be tested by running "Question1_4.py" script with either giving path of .mat file or choosing random graph. Script testing methods detailed in README.txt file.

You can also see the function code in Fig. 7

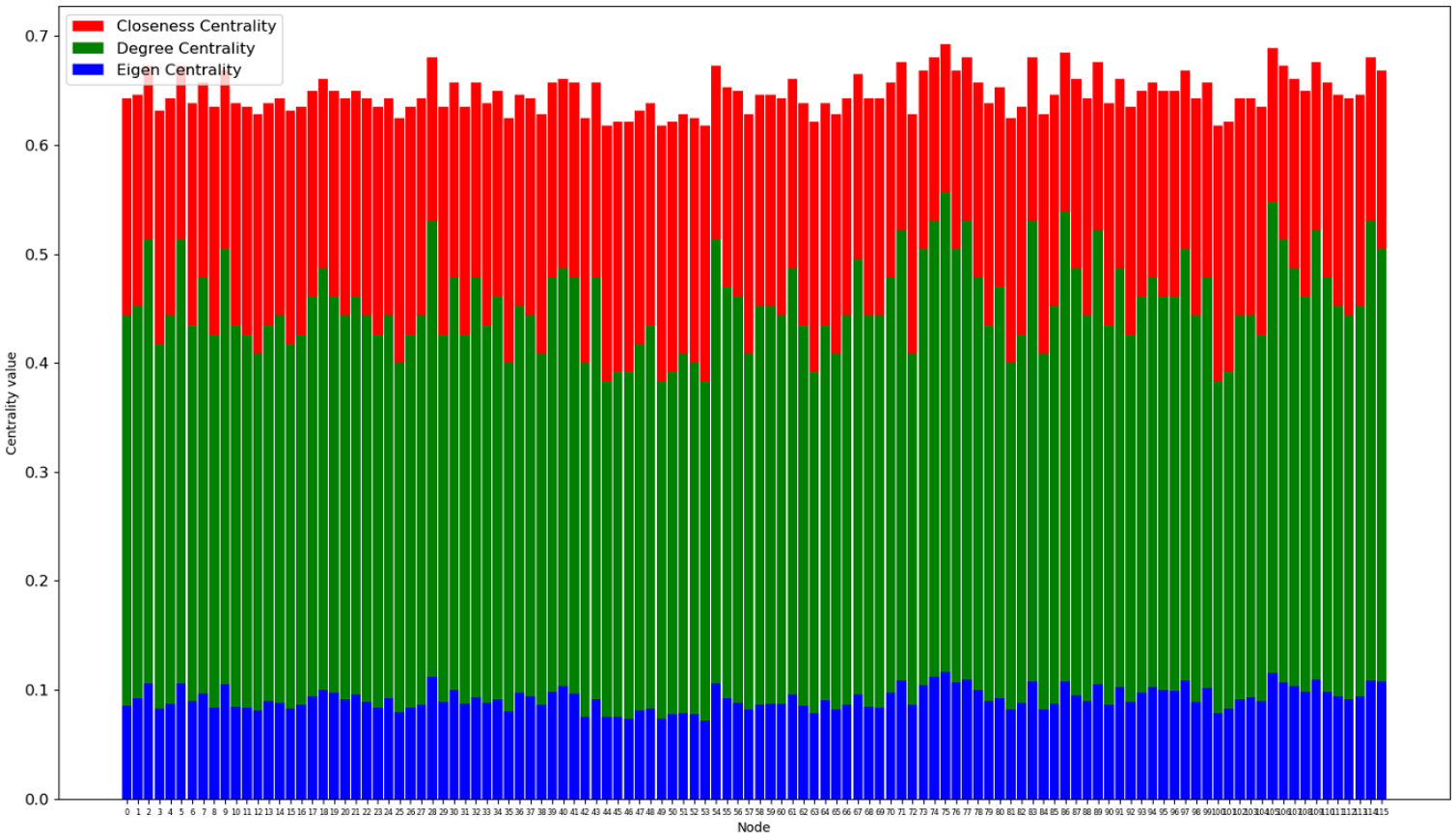

Fig. 4: *Centralities of the matrix C.mat.*

4.2 The following logic is followed to create similar plot to Figure 6.4 in FBNA textbook. First empty graph created. Then edge weights of the given graph (input of the function) sorted in decreasing order. By starting the biggest weighted edge, edges added to the empty graph. At each addition, $GetMxWccSz()$ function is used to calculate the fraction of nodes in the largest weakly connected component of the graph and saved with the connection density of a graph at given state. Until the reaching the 95% at the fraction value edges are added. 95% is set for the percolation threshold. Connection density value at 95% fraction is returned by the function.

For visualization purposes another function named "generate_connected_component_plot" is implemented. This function use same logic with percolation threshold but it continues to add all the edges. After adding all the edges, saved fractions and connection densities are plotted. You can also see the function code in Fig. 8

```
# Step 2
# Store centralities for each node
degree_cent = []
closeness_cent = []
eigen_cent = []
# Calculate eigen centralities
NIdEigenH = s.TIntFltH()
s.GetEigenVectorCentr(bin_graph, NIdEigenH)
# Calculate centralities for each nodes
for id,node in enumerate(bin_graph.Nodes()):
    degree_cent.append(s.GetDegreeCentr(bin_graph, node.GetId()))
    closeness_cent.append(s.GetClosenessCentr(bin_graph, node.GetId()))
    eigen_cent.append(NIdEigenH[id])

fig = plt.figure(figsize=(64,32))
ax = fig.add_subplot(111)
indices = np.arange(len(degree_cent))
width = 0.9
ax.bar(indices, closeness_cent,width=width, color='r', alpha=1, label='Closeness Centrality')
ax.bar(indices, degree_cent,width=width, color='g', alpha=1, label='Degree Centrality')
ax.bar(indices, eigen_cent, width=width, color='b', alpha=1, label='Eigen Centrality')
plt.xticks(indices,
           ['{}'.format(i) for i in range(len(degree_cent))] )
ax.tick_params(axis='x', which='major', labelsize=6)
ax.tick_params(axis='y', which='major', labelsize=12)
ax.set_xlabel("Node")
ax.set_ylabel("Centrality value")
ax.legend(fontsize='large',loc=2)
plt.show()
```

Fig. 5: *Centrality calculations.*

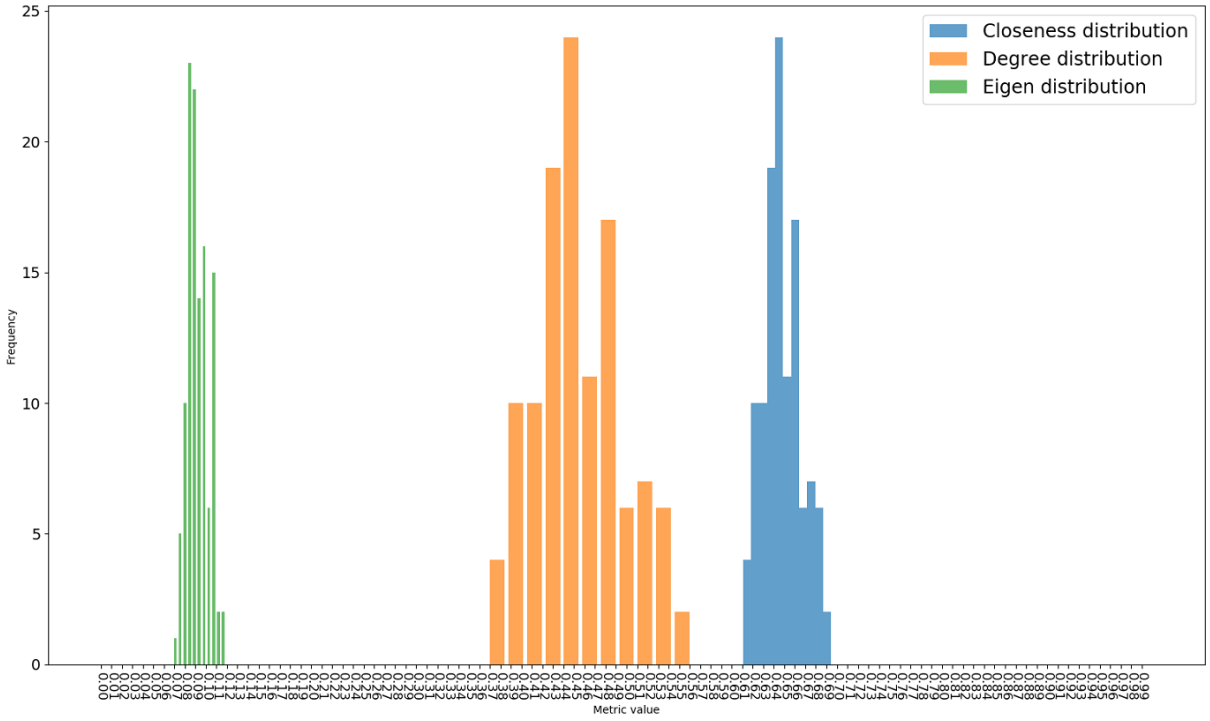

Fig. 6: *Centrality distributions of the matrix C.mat according to the three centralities.*

In test script first percolation threshold of a given graph or random graph (chosen by argument) is printed into terminal and then the plot is shown to the

```python
def get_density(edges,nodes):
    return edges / (nodes * (nodes-1))

def findPercolationThreshold(adj_matrix):
    adj_matrix = np.array(adj_matrix)
    # Remove self connections from adj matrix
    np.fill_diagonal(adj_matrix, 0)

    # Create an empty graph
    graph = s.TUNGraph.New()
    # Create all nodes but no connections between them
    [graph.AddNode() for i in range(adj_matrix.shape[0])]

    percolation_threshold = 0
    # Add every edge in order of descending weights
    while np.amax(adj_matrix) > 0 :
        result = np.where(adj_matrix == np.amax(adj_matrix))[0]
        # Add the edge with biggest weight
        graph.AddEdge(int(result[0]),int(result[1]))
        # Mark (make weight 0) the biggest weighted edge in adj_matrix
        adj_matrix[result[0]][result[1]] = 0.
        adj_matrix[result[1]][result[0]] = 0.
        # Calculate biggest connected compenent
        fraction_val = s.GetMxWccSz(graph)

        if fraction_val > 0.95 and percolation_threshold == 0:
            percolation_threshold = get_density(graph.GetEdges(),graph.GetNodes())

    return percolation_threshold
```

Fig. 7: *Percolation threshold finding.*

```python
def generate_connected_component_plot(adj_matrix):
    adj_matrix = np.array(adj_matrix)
    # Remove self connections from adj matrix
    np.fill_diagonal(adj_matrix, 0)

    # Create an empty graph
    graph = s.TUNGraph.New()
    # Create all nodes but no connections between them
    [graph.AddNode() for i in range(adj_matrix.shape[0])]
    percolation_percents = []
    connectivity_percents = []

    # Add every edge in order of descending weights
    while np.amax(adj_matrix) > 0 :
        result = np.where(adj_matrix == np.amax(adj_matrix))[0]
        # Add the edge with biggest weight
        graph.AddEdge(int(result[0]),int(result[1]))
        # Mark (make weight 0) the biggest weighted edge in adj_matrix
        adj_matrix[result[0]][result[1]] = 0.
        adj_matrix[result[1]][result[0]] = 0.
        # Calculate biggest connected compenent
        fraction_val = s.GetMxWccSz(graph)
        percolation_percents.append(fraction_val)
        connectivity_percents.append(get_density(graph.GetEdges(),graph.GetNodes()))

    plt.plot(connectivity_percents,percolation_percents)
    plt.xlabel("Connection Density")
    plt.ylabel("Proportion of nodes in largest component")
    plt.savefig("connected_component_plot.png")
    plt.show()
```

Fig. 8: *Connected component plot creation code.*

user. Also the produced plot is saved into the same folder. You can see the plot of the C.mat matrix in Fig. 9. Also the percolation threshold of the given C.mat matrix is 0.0238380809

2 Hypergraph centrality

Step 1: Finding a good paper. The paper that I found is titled "Weighted node degree centrality for hypergraphs"[1] which investigates the hypergraph centralities and proposes a new centrality measure.

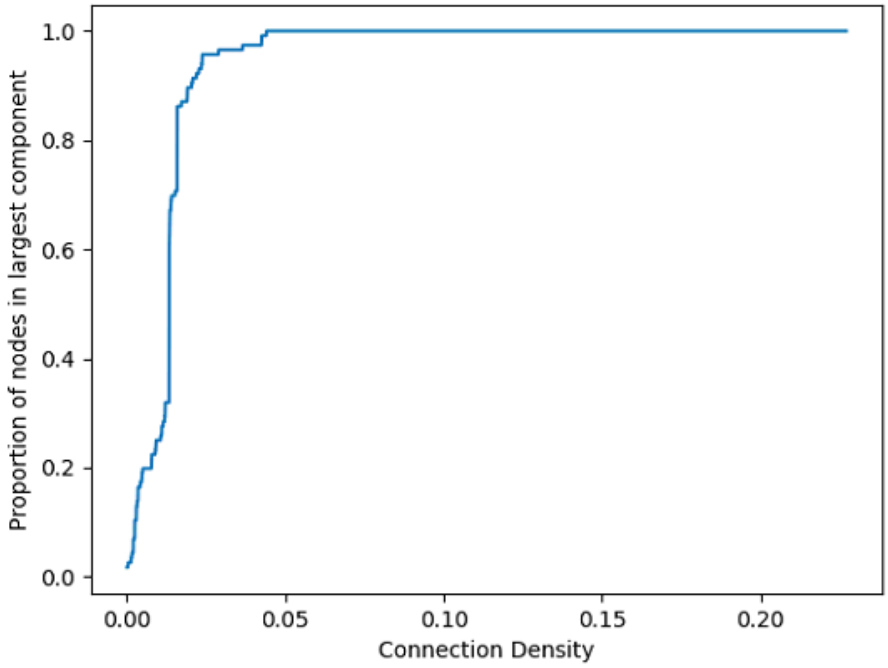

Fig. 9: *Proportion of nodes in largest component vs connection density of given C.mat matrix. Percolation threshold is 0.0238380809 where proportion is reached 95%*

Researchers generally focus on big social interaction graphs. They test their centrality measure with DBLP dataset of scientific collaborations and the group network in a popular Chinese multi-player online game. They propose the centralities that are called Weighted Node Degree (WHND), Strong and Weak Tie Degree Centralities (SWTHND). SWTHND is used to classify nodes. In this homework I will implement only WHND which is more trivial centrality measure and my knowledge in the graph theory was not enough to grasp the concept of SWTHND. Therefore i will be only implement the WHND centrality.

In graph theory degree centrality of a node in a hypergraph is defined as the number of nodes adjacent to it. But this metric calculates the centrality from local viewpoint and disregards the strength of the two adjacent nodes.The weighted degree centrality of a node in a hypergraph is defined as the sum of weights of the ties of the node with the other nodes in the hypergraph.

Given a hypergraph $H = (V; E)$, weighted node degree centrality can be defined as follows: $C_d^h(i) = \sum_{k=1}^{L} \sum_{j=1,j\neq i}^{N} x_{ik} * x_{jk}$ where x_{mn} is 1 if node belongs to edge and 0 otherwise.

But researchers argue that giving same weights to all edges miss some relations between the nodes. Therefore they propose WHND with multiple differ-

ent weights. So proposed WHND centrality measure formula becomes $C_d^h(i) = \sum_{j=1,j\neq i}^{N} \sum_{k=1,\{v_i,v_j\}\subset e_k}^{L} w_k$ where w_k is a weight of the hyper edge k. Weight calculation can be done via different methods. Authors give 5 different weight calculation methods by using methods of previous research. In this homework I will only implement 2 weight calculations. First one is constant weight that each edge gets 1 as weight. Second one is Frequency based which every edge weight is calculated from the multiplicity of the edge. Therefore frequency based weight is related to frequency of the hyperedge's occurence. Authors also gives weight calculation methods by using cardinality of the edge, but for this homework I only consider multiplicity of the edges.

Authors conclude that their centrality measures fits more into social networks. Also by their experiments they prove that their centrality measures work on the real social network datasets.

Step 2: Coding up hypergraph centrality. For the implementation I have not use any SNAP or Graph library. The test script, named "Question2.py", gets either path of the incidence matrix (.mat file) or can be used by random incidence matrix. Details of the script execution can be found in "README.txt". Another argument, which selects the weight calculation, can be given to the script. Argument weight=1 selects constant weight calculation and weight=2 selects frequency based weight calculation. Then scripts show centralities of each node as a graph, after that it shows the plot of the centrality distribution. Note that random graphs may not create meaningful results.

You can see the hypergraph edge weight calculation and centrality calculation code in Fig. 10 and Fig. 11

```
def calculate_hyperedge_weights(inc_mat,weight_type=1):
    weights = []
    # Two types of weight calculation is considered
    # normalization is different for each weights
    # because max. sum of centralities differ with different weights
    # Weight 1 = constant weight for each edge
    # Weight 2 = multiplicity of a edge is the weight of the edge
    if weight_type == 1: #Constant weights
        weights = [1 for i in range(inc_mat.shape[1])]
        normalizer = inc_mat.shape[0] * (inc_mat.shape[0]-1)

    elif weight_type == 2:  #Frequency based (multiplicity) weights
        for i in range(inc_mat.shape[1]): # For each edge
            weight = 1
            #Calculate multiplicity by comparing different edges with same vertex
            for j in range(inc_mat.shape[1]):
                if i == j: continue
                flag = 0
                for k in range(inc_mat.shape[0]):
                    if inc_mat[k][i] == inc_mat[k][j]:
                        flag+=1
                weight += flag
            weights.append(weight)
    else:
        raise Exception("Wrong type of weight. Please select weight as 1 or 2!")
    normalizer = inc_mat.shape[0]*(inc_mat.shape[0]-1)*inc_mat.shape[1]*(inc_mat.shape[1]-1)
    return weights, normalizer
```

Fig. 10: *Hypergraph edge weight calculation according to the two method.*

Step 3: Hypergraph centrality profile.

3.1 The centralities for each node can be seen in Fig. 12. Note that this calculation made with WHND measure by using frequency based weight calculation. Also, the distribution of the centralities can be seen in Fig. 14.

```
# Calculate edge weights for the chosen weight type
centralities = []
weights, normalizer = calculate_hyperedge_weights(inc_mat,int(args.weight))
for i in range(num_node):
    weight_sum = 0
    for j in range(num_node):
        if i == j: continue
        for k in range(num_hyperedge):
            if(inc_mat[i][k] == 1 and inc_mat[j][k] == 1):
                weight_sum += weights[k]
    centralities.append(weight_sum/normalizer)
```

Fig. 11: *Centrality calculation by supplied edge weights.*

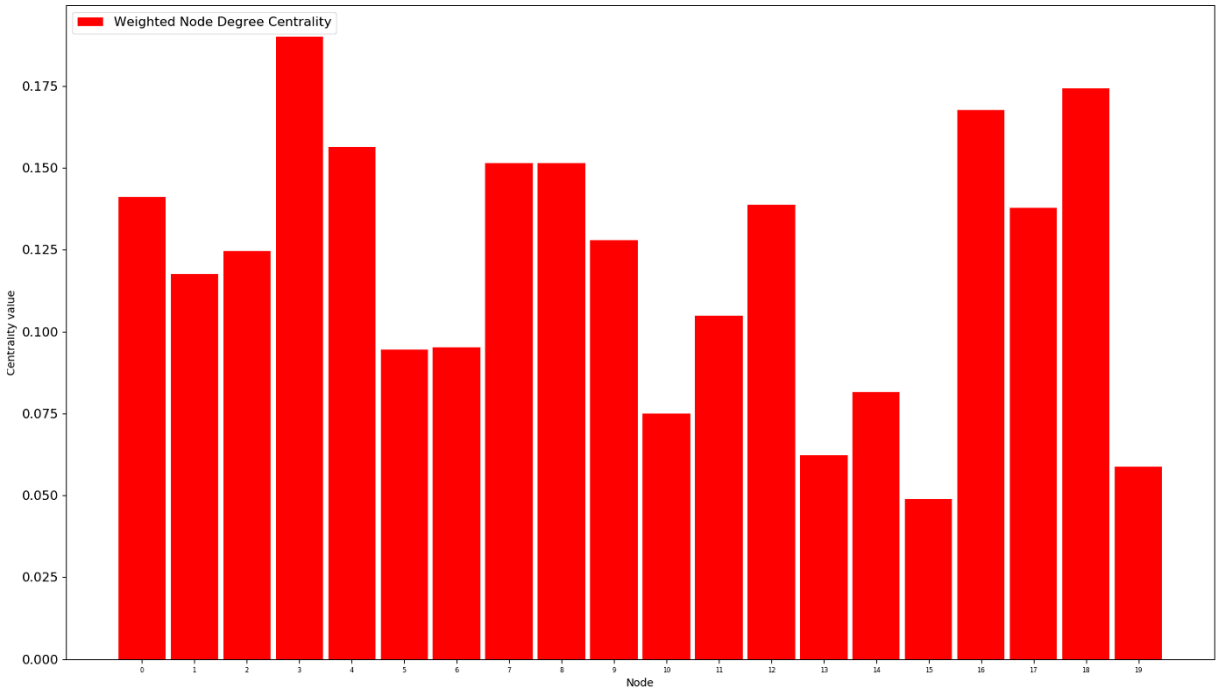

Fig. 12: *Centrality of the nodes of the matrix H.mat according to the WHND centrality.*

Fig. 13: *Epinions logo*

3.2 What conclusions can you derive about the hypergraph topology? Node 3, 15 and 18 are the nodes that looks more "popular" (central). And there are also lots of nodes with low centralities, therefore this graph topology looks like followers in the instagram topology. There are some "popular" nodes and other nodes are not as "popular" as these nodes. These popular nodes have relation

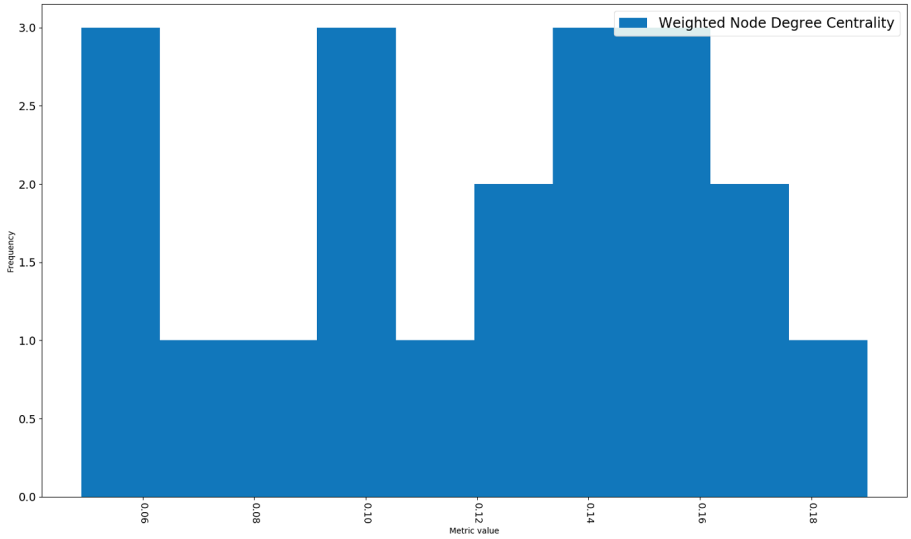

Fig. 14: *Centrality distributions of the matrix H.mat according to the WHND centrality*

more frequently because we used frequency based weight calculation which gives more centrality to the nodes that have more frequent relations. This can also be seen as ISP network topology where users have high traffic with their ISP and also different ISP companies have higher traffic between them.

3 Profile a SNAP graph of your choice

Step 1: Network selection and presentation .

I have selected the soc-Epinions1 dataset for investigation. You can access the dataset from here : http://snap.stanford.edu/data/soc-Epinions1.html

1.1 Epinions.com was a general consumer review site established in 1999. Users of the site was reviewing their online shopping experiences. As a user you would like to see reviews from more "trusted" users. Such as in daily life we get product or service recommendations from our friends and families. In Epinions.com if you believe a user is trusted source for you, then you could tag the user as "trusted". User see more reviews from their trusted contacts. This dataset shows the trust relationship among the users. I selected this dataset because trust is a really ambigous entity and learning or analyzing how a community build trust, or how some people became trustable person is really exciting. Today most of the shopping have been made on online and if one can learn how to be a "trusted identity" then one can use the trust to boost their sales.

Each node in the graph represent a user and edges between the nodes are trust between the users.There are over 75k nodes and over 500k edges. Also, it

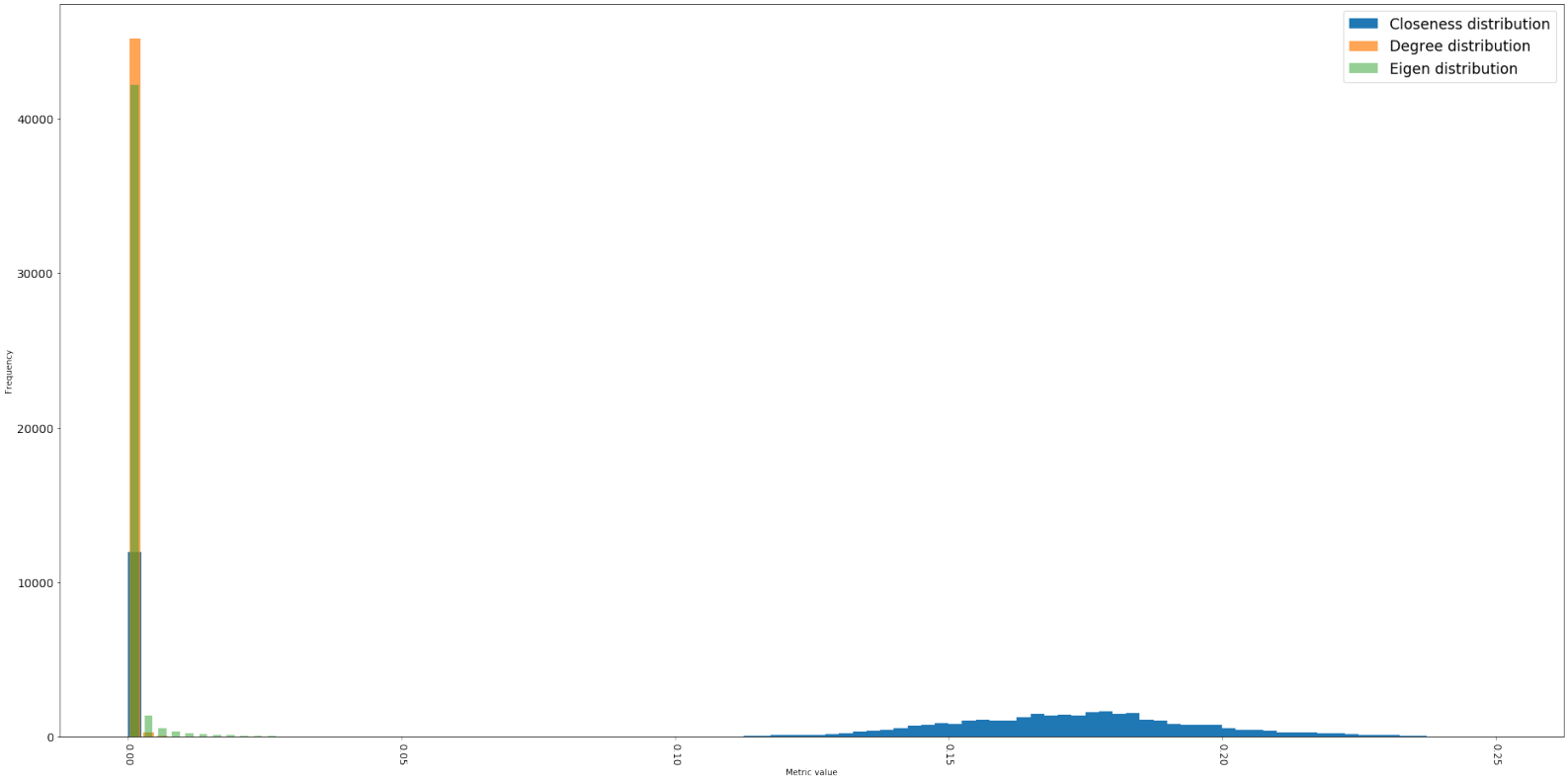

Fig. 15: *Centrality distributions of the Epinions dataset*

is a directed graph because user "A" can trust the user "B" but user "B" may not trust to "A". It is important because if trust is two way then it reveals more information about the strength of the trust relationship.

1.2 I have not removed 40% of the nodes because number of nodes were not really high and computing the graph by just removing 20% was feasible. Therefore, I have removed 20% of the nodes randomly. To remove randomly 20% of the graph, for every node basically I throw a coin with 20% chance to be succesful. So at the end of the loop nearly 20% of the nodes would be removed. Then remaining graph contains nearly 60k nodes and 260k edges. You can see the code to randomly remove 20% in Fig. 16

Step 2: Network profiling using centralities

2.1 To generate the overlapping plots of the centralities I have used my code that I have implemented in the Question 1. You can see the adapted code in Fig. 17

Here is the distribution of the centralities. The plot may be not clear enough to read or see but by using original plot image, one can see the details.

```
def delete_20_percent(graph):
    for node in ep_graph.Nodes():
        # For every node 20% possiblity to delete it
        # Since my dataset is not really big only 20%  is enough for thresholding
        # After the for loop nearly 20% of the nodes would be deleted
        should_delete = np.random.choice(2, 1, p=[0.8,0.2])[0]
        if should_delete:
            graph.DelNode(node.GetId())
    return graph
```

Fig. 16: *Randomly removing 20% of the nodes.*

```
# Step 1.2
# Remove 20% of the nodes to use thresholded version
ep_graph = delete_20_percent(ep_graph)
num_node = ep_graph.GetNodes()
num_edge = ep_graph.GetEdges()
print("After removing 20\% of the nodes, Number of nodes: {}, Number of edges {}".format(num_node,num_edge))

# Step 2
# Calculate centralities
node_ids = []
degree_cent = []
closeness_cent = []
eigen_cent = []

NIdEigenH = s.TIntFltH()
s.GetEigenVectorCentr(ep_graph, NIdEigenH)
print("Please wait for calculation, it takes time.")
# Calculate centralities for each nodes
for id,node in enumerate(ep_graph.Nodes()):
    if id % 1000 == 0:
        print("{} / {}".format(id,num_node))
    node_ids.append(node.GetId())
    degree_cent.append(s.GetDegreeCentr(ep_graph, node.GetId()))
    closeness_cent.append(s.GetClosenessCentr(ep_graph, node.GetId()))
    eigen_cent.append(NIdEigenH[node.GetId()])
```

Fig. 17: *Centrality calculation of the graph.*

2.2 The most interesting thing in the first look is lots of nodes really close to 0 centrality which means that they do not have nearly any centrality value in the graph. We can interpret this result as most of the users do not have any trust relationship between each other. Especially in Degree centrality nearly all the nodes have near 0 centrality, which can be interpreted as most of the users do not trust many people or not trusted by so many people. Eigen centrality results looking also the same but for eigen centrality there are some user nodes that shows higher centrality. So we can say that there are some users that people see them as trustworthy identities, because the eigenvector centrality metric is a measure of the influence of a node in a network. The scores are based on the concept that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. However we see in the closeness centrality, there are lots of nodes that shows closeness centrality between 0.15 and 0.25. It means that there are some group of nodes which are closer (shortest path between them is low). So from any user you can access other trusted users, which show us if you know someone trustworthy then you can find more trusted persons. It also show us trustworthy people trust more on trustworthy people and become more closer nodes in the graph. So we can deduce the advice that trust is a collabrative definition and you can find more trustable people when you trust a person who is trusted by the "society" (Epinion users in this case).

Step 3: Connected components and resilience to attacks

3.1 The dataset contains directed graph because it contains trust relationships. Therefore I have calculated strongly connected component sizes. To sim-

ulate the targeted attacks for each centrality value, top 5% most central nodes removed at each step until we remove 60% of the nodes. To achieve this thresholded graph object is copied 3 times. From these 3 exact graphs most central nodes according to the centralities removed to simulate the attack. Then for each removal **size of biggest strongly connected component** is plotted in overlapping scatter plot. Note that there are more than one strongly connected components but for the better visualization only the biggest strongly connected component size is shown. You can see the result of the attacks in Fig. 18 and you can see the code to simulate the attack in Fig. 19

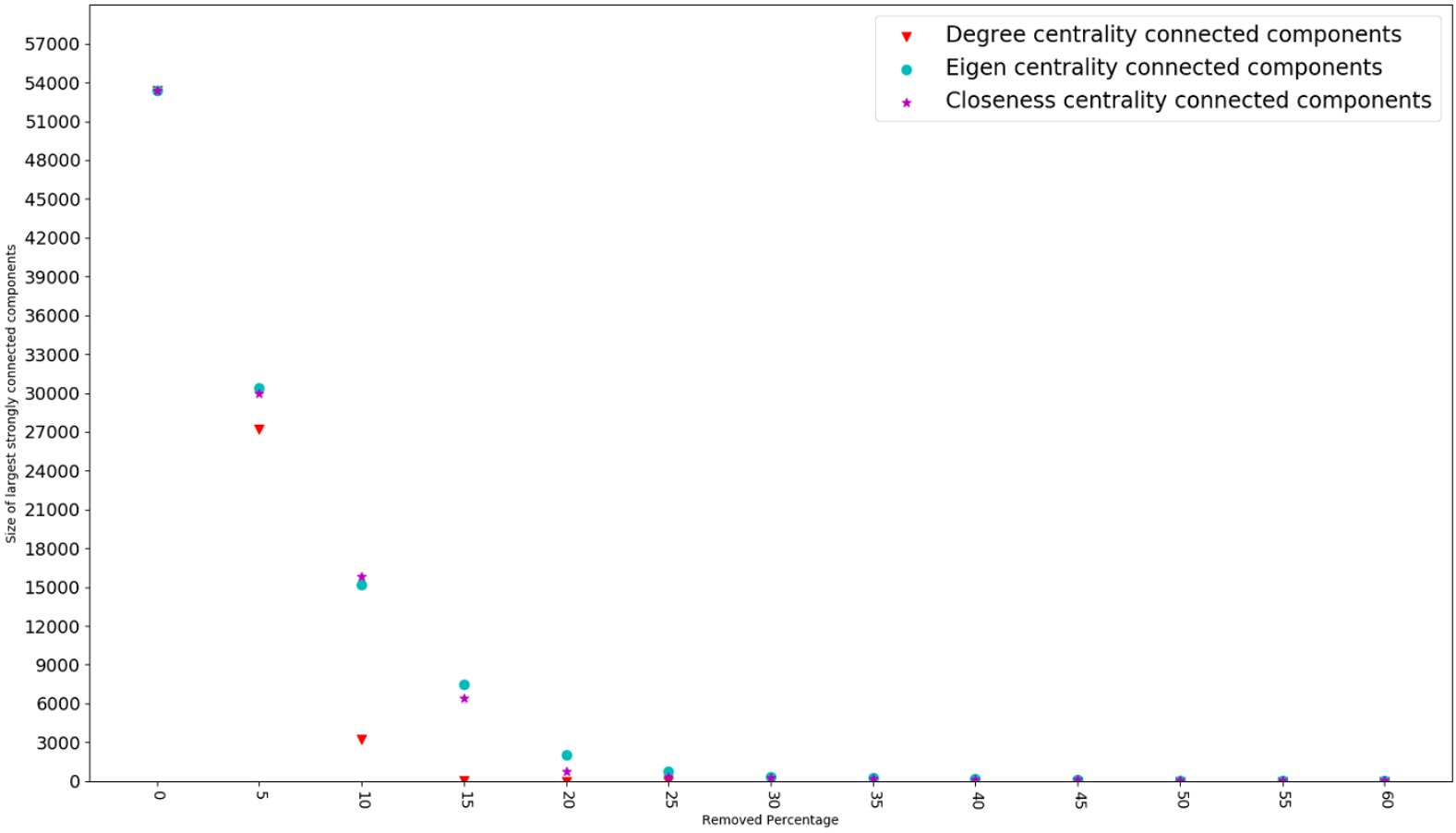

Fig. 18: *Size of the largest connected component when attacked to most central nodes according to the three different centralities*

3.2 Degree centrality shows very different behavior when attacked to the central nodes. Both eigen centrality and closeness centrality cases show simi-

```
degree_components = []
closeness_components = []
eigen_components = []

# At each attack, remove top  5% most central nodes until 60%
num_delete = int(node_ids_degree.shape[0] * 0.05)
# First loop would hold the connected compenent size of original graph
# Therefore loop stop when 60% of the most central nodes removed
for topk in range(0,65,5):
    ComponentDist = s.TIntPrV()
    s.GetSccSzCnt(ep_graph_1, ComponentDist)
    degree_components.append(ComponentDist.Last().GetVal1())

    ComponentDist = s.TIntPrV()
    s.GetSccSzCnt(ep_graph_2, ComponentDist)
    closeness_components.append(ComponentDist.Last().GetVal1())

    ComponentDist = s.TIntPrV()
    s.GetSccSzCnt(ep_graph_3, ComponentDist)
    eigen_components.append(ComponentDist.Last().GetVal1())


    # Delete top 5 percent for each centralities
    for i in range(num_delete):
        ep_graph_1.DelNode(int(node_ids_degree[-1-i]))
        ep_graph_2.DelNode(int(node_ids_closeness[-1-i]))
        ep_graph_3.DelNode(int(node_ids_eigen[-1-i]))
    n_degree_cent = n_degree_cent[:-num_delete]
    n_closeness_cent = n_closeness_cent[:-num_delete]
    n_eigen_cent = n_eigen_cent[:-num_delete]

    node_ids_degree = node_ids_degree[:-num_delete]
    node_ids_closeness = node_ids_closeness[:-num_delete]
    node_ids_eigen = node_ids_eigen[:-num_delete]
```

Fig. 19: *Attacking 60% most central nodes of the graph to simulate the attack.*

lar results. After the attack to the most central 25% nodes, graph completely fragmented which means connections between many nodes are disappear and there is no big connected component if you attack the most central 25% nodes. However, when you attack the most central nodes by calculating degree centrality, size of strongly connected component drastically decrease. After 10%-15% graph is completely fragmented. Especially in 10% and 15% difference between the degree centrality and others is significant. We can make an assumption that nodes with high degrees connect far nodes (shortest path between two nodes is high) with each other and most of the nodes connected to only a node which is a hub. These hub nodes connected to many nodes (high degree centrality) and attacking to these hubs remove the path between two unrelated nodes. This also show us the number of hub nodes (nodes with high degree centralities) is low in the graph. So there are some people which are trusted by so many users but users (nodes with low degree centralities) trust low amount of people, therefore removing some "trustworthy" nodes shear the graph quickly.

So the best deduction from this attack scenario is, Epinions users trust few amount of "trustworthy" users, but these few amount of "trustworthy" people is trusted by the community (lot of users trust these people).This deduction stand on the assumption that attacking only 10% of the highly connected nodes is enough to fragment the graph. When 15% of the most trustworth users are deleted, there are no users that is trusted by multiple people.

References

1. Kapoor, K., Sharma, D., Srivastava, J.: Weighted node degree centrality for hypergraphs. In: 2013 IEEE 2nd Network Science Workshop (NSW). (2013) 152–155