Network Topology

Satyadev Tummala
801202003

Akhil Reddy Gangula
801180579

1) Introduction:

Network Topology Properties can be measured in various ways and they are very useful for graph data mining and spatial analysis of the network.
In this project we measured 6 different characteristics using Biological Network data.

2)Degree Centrality (Average Node Degree):

Degree Centrality is used for finding the degree information of each node. It is used in identifying the core nodes in a complex network.

Formula for computing: $< K > = \frac{\Sigma a_{ij}}{N}$ **Where a** $_{ij}$ **is the adjacency matrix**

Pseudo Code:

```
def compute_node_deg(self):
        node_dict = {}
        node_sum =  0
        if self.no_nodes == 0:
            return None,None
        for i in range(self.no_nodes+1):
```

```
            node_deg = 0
            for j in range(self.no_nodes+1):
                if self.adjancey_matrix[i][j] > 0:
                    node_deg += 1
            node_dict[i]  = node_deg
            node_sum  +=  node_deg
        avg_nodes = node_sum/self.no_nodes
        return node_dict,avg_nodes
```

Time Complexity

The Time Complexity of the Algorithm is O(V^2) Where V is the number of Vertices.

Space Complexity

The Space Complexity of the Algorithm is O(V) .

3)Node Centrality (Strength Distribution):

This Measure is useful for finding how much a node is significant in the network when compared to other nodes.

Formula to Compute $< w > = \frac{\Sigma_i \Sigma_j w_{ij}}{N}$ **Where w** $_{ij}$ **is the adjacency matrix with weights and N is the number of nodes**

Pseudo code

```
def compute_strength_distibution(self):
        strength_dict = {}
        strength_sum =  0
        for i in range(self.no_nodes+1):
```

```
            strength = 0
            for j in range(self.no_nodes+1):
                strength += self.adjancey_matrix[i][j]
            strength_dict[i]  = strength
            strength_sum += strength
        avg_strength = strength_sum/self.no_nodes
        return strength_dict,avg_strength
```

Time Complexity

The Time Complexity of the Algorithm is O(V^2) Where V is the number of Vertices.

Space Complexity

The Space Complexity of the Algorithm is O(V) .

4)Average Node Weight:

This metric is used to compute the average weight with respect to the edges.It is used to find how each node is contributing to the total cost of the network .

Formula to Compute $< w > = \frac{\Sigma_i \Sigma_j w_{ij}}{E}$ **Where** w_{ij} **is the adjacency matrix with weights and E is the Number of Edges.**

Pseudo code

```
def compute_avg_weight(self):
        total_weight = 0
        for i in range(self.no_nodes+1):
            for j in range(self.no_nodes+1):
                total_weight +=
```

```
self.adjancey_matrix[i][j]
        node_weight = 0
        for i  in range(self.no_nodes+1):
            for j in range(self.no_nodes+1):
                node_weight +=
self.adjancey_matrix[i][j]
        avg_weight = node_weight/self.no_edges
        return avg_weight
```

Time Complexity:

The Time Complexity of the Algorithm is O(V^2) Where V is the number of Vertices.

Space Complexity:
The Space Complexity of the Algorithm is O(1) .

5)Characteristic Path Length:

This metric is used for determining whether the graph is in linear chain form or more complex form.

Formula to Compute $< L >$ $= \frac{\Sigma_i \Sigma_j Lij}{N(N-1)}$ **Where L** $_{ij}$ **is the shortest path between node i and j , and N is the number of nodes.**

Higher L means the network is more linear and lower L means the network is complex.

Pseudo code

```
def compute_strength_distibution(self):
        strength_dict = {}
        strength_sum =  0
        for i in range(self.no_nodes+1):
            strength = 0
            for j in range(self.no_nodes+1):
                strength += self.adjancey_matrix[i][j]
            strength_dict[i]  = strength
            strength_sum += strength
        avg_strength = strength_sum/self.no_nodes
        return strength_dict,avg_strength

    def compute_avg_weight(self):
        total_weight = 0
        for i in range(self.no_nodes+1):
            for j in range(self.no_nodes+1):
                total_weight +=
self.adjancey_matrix[i][j]
        node_weight = 0
        for i  in range(self.no_nodes+1):
            for j in range(self.no_nodes+1):
                node_weight +=
self.adjancey_matrix[i][j]
        avg_weight = node_weight/self.no_edges
        return avg_weight


    def compute_distance_between_nodes(self,beg_node):
        unvistied  = queue.Queue()
        unvistied.put(self.nodes[beg_node])
        vistied = []
```

```
        each_node_arr = np.full(self.no_nodes+1,np.inf)
        each_node_arr[beg_node] = 0
        itr_no = 1
        while len(vistied) != len(self.nodes):
            node = unvistied.get()
            neighbhors = []
            for j in range(self.no_nodes+1):
                if self.adjancey_matrix[node][j]  > 0:
                    neighbhors.append(j)
            for each in neighbhors:
                dist = each_node_arr[node]  +
self.adjancey_matrix[node][each]
                if dist  < each_node_arr[each]:
                    each_node_arr[each] = dist
            vistied.append(node)
            next_node =
self.find_min(each_node_arr,vistied)
            unvistied.put(next_node)
        each_node_arr  =
np.where(each_node_arr==np.inf,0,each_node_arr)
        return each_node_arr


    def find_min(self,array,vistied_nodes):
        array_dict = {}
        for value in np.unique(array):
            array_dict[value]  =
np.where(array==value)[0].tolist()
        sorted_array =
np.sort(np.delete(array,vistied_nodes))
        min_index =   sorted_array.min()
```

```
        for item in array_dict[min_index]:
            if item not in vistied_nodes:
                next_node = item
        return next_node

    def charestric_path_length(self):
        try:
            sum_of_distnaces = 0
            for i in self.nodes:
                sum_of_distnaces  +=
np.sum(self.compute_distance_between_nodes(i))
            char_path_length =
sum_of_distnaces/(self.no_nodes * (self.no_nodes-1))
            return char_path_length
        except Exception as e:
            print("Skipping this metric")
```

Time Complexity:

The Time Complexity of the Algorithm is $O(V^3)$ Where V is the number of Vertices.

Space Complexity:

The Space Complexity of the Algorithm is O(V) .

6)Clustering Coefficient:

This metric is used to determine how close the nodes in a graph are clustered to each other.

Formula to Compute $< C >$ $= \frac{2\,e}{K(K-1)}$ **Where e is the number of edges among neighbors and K is the degree of the current node**

Pseudo code

```
def clustering_coeffiecnt(self,node_dgerees):
        try:
            cc = 0
            for node,node_deg in node_dgerees.items():
                if node == 0:
                    continue
                k  = node_deg
                neighbors_edges  = 0
                neighbhors = []
                for j in range(self.no_nodes+1):
                    if self.adjancey_matrix[node][j] >
0:
                        neighbhors.append(j)
                for q in range(len(neighbhors)):
                    for p in range(len(neighbhors)):
                        if
self.adjancey_matrix[neighbhors[q]][neighbhors[p]] > 0:
                            neighbors_edges  += 1
                    p = 0
                if neighbors_edges == 0:
                    cc += 0
                else:
                    cc += (2 * neighbors_edges)/(k *
(k-1))
```

```
            cc_n = cc/self.no_edges
            return cc_n
        except Exception as e:
            print("Skiping this metric")
            return 0
```

Time Complexity:

The Time Complexity of the Algorithm is O(V^2) Where V is the number of Vertices.

Space Complexity:
The Space Complexity of the Algorithm is O(V) .

7)Closeness Centrality:

It is used to find central vertices . Higher centrality means the vertex is more central in the network

Formula to Compute $< C(res) > = \frac{N-1}{\Sigma dist(x,y)}$ **Where N is the number of nodes and denominator is the sum of all distance pairs.**

Pseudo code

```
def closness_centrality(self):
        closse_ness  = []
        for i in self.nodes:
            closeness = (self.no_nodes -1
)/np.sum(self.compute_distance_between_nodes(i))
            closse_ness.append(closeness)
        return closse_ness
```

Time Complexity:

The Time Complexity of the Algorithm is $O(V^3)$ Where V is the number of Vertices.

Space Complexity:
The Space Complexity of the Algorithm is O(V) .

8) Evaluation Results:

Run Command Example **python3 run_network.py --src Data/Bacillus_subtilis_168**

Creates an output.json file

Output:
A Snippet from output.json file


```
{
    "Graph_name":
"de_novo_biosynthesis_of_pyrimidine_deoxyribonucleotides.gr
p",
    "Node Average of Graph": "1.2857142857142858",
    "Strength Average of Graph": "7.571428571428571",
    "Average weight of  Graph": "5.888888888888889",
    "Characteristic Path Length of  Graph":
"9.833333333333334",
```

```
    "Clustering Coefficient of Graph":
"0.025379382522239664",
    "Closeness_Centrality": "[0.08450704225352113, 0.125,
0.21428571428571427, 0.11764705882352941,
0.08108108108108109, 0.08333333333333333,
0.08695652173913043]"
  },
{
    "Graph_name":
"2-keto_glutarate_dehydrogenase_complex.grp",
    "Node Average of Graph": "1.0",
    "Strength Average of Graph": "5.666666666666667",
    "Average weight of  Graph": "5.666666666666667",
    "Characteristic Path Length of  Graph": "8.5",
    "Clustering Coefficient of Graph": "0.0",
    "Closeness_Centrality": "[0.16666666666666666,
0.09090909090909091, 0.11764705882352941]"
  },
```