Social Network Analysis for Computer Scientists
Fall 2024 — Assignment 1

Jiameng Ma(4255445)

September 2024

Exercise 1:Neighborhoods

Question 1.1: For $G = (V, E)$, the reverse neighbourhood of v, denoted as $N_{in}(v) = \{u \in V : (u, v) \in E\}$, which consists of all nodes that have incoming edges to v; and $N_{out}(v) = \{u \in V : (v, u) \in E\}$ was consists of all nodes that have outgoing edges to v.
So the combined degree is $k_{combined}(v) = |N_{out}(v) \cup N_{in}(v)|$

Question 1.2: For $G = (V, E)$, $\forall v \in V, \quad N_{out}(v) = N_{in}(v)$

Question 1.3: For $G = (V, E)$, $\forall v \in V, \quad \forall u, w \in N(v), \quad (u, w) \notin E$
(u, w) means the edges between node u and w

Question 1.4:
K-neighborhood: The k-neighborhood means node v in a graph is the set of all nodes that can be reached from v within k steps.
Reversed Neighborhood: The reversed neighborhood of a node v, is the set of all vertices that have an edge pointing to v.
Based on above, The reversed k-neighborhood of a node v is the set of vertices that can reach v within k steps.

Question 1.5:

```
def is_same_weakly_connected_component(G, u, v):
    N_u = set(nx.descendants(G, u)).union(set(nx.ancestors(G, u)), {u})
    N_v = set(nx.descendants(G, v)).union(set(nx.ancestors(G, v)), {v})

    return bool(N_u.intersection(N_v))
```

nx.descendants(G, u) and nx.ancestors(G, u) give the out-neighborhood and in-neighborhood of u respectively, and intersection() check if the neighborhoods overlap.

Question 1.6:

```
def is_dependent_on(G, a, b):
 return nx.has_path(G, a, b)
```

Question 1.7:

```
import networkx as nx
from itertools import combinations

def count_open_wedges(G):
    open_wedge_count = 0

    for v in G.nodes():

        neighbors = list(G.neighbors(v))

        for u, w in combinations(neighbors, 2):
            if not G.has_edge(u, w):
                # Count as an open wedge (u, v, w)
                open_wedge_count += 1

    return open_wedge_count
```

We need looking over all nodes(V), For each node, finding the set of neighbors, takes $O(\deg(v))$, $\deg(v)$ s the degree (number of neighbors) of node v. For each node v, we check all pairs of neighbors, so is $O(\deg(v)^2)$. ¡br/¿ Thus, the time complexity for each node is: $O(\deg(v)^2)$¡br/¿ So the overall time complexity of the algorithm is: $O\left(\sum_{v \in V} \deg(v)^2\right)$

Question 1.8:

```
def friendship_paradox_count(G):
count = 0

for u in G.nodes():
    deg_u = G.degree(u)

    neighbors = G.neighbors(u)

    if deg_u > 0:
        avg_deg_neighbors = sum(G.degree(v) for v in neighbors) / deg_u

        if avg_deg_neighbors > deg_u:
            count += 1

return count
```

For $G = (V, E)$, each node u, the degree can be obtained in constant time,$O(1)$, summing the degrees of its neighbors involves iterating over the neighbors, which is $O(E)$, this algorithm iterates over all the nodes, and for each node, it performs a calculation proportional to the number of its neighbors. So the time complexity is $O(|V| + |E|)$.

Question 1.9: The diameter of a graph G is the greatest distance between any two nodes in the graph. This distance is the length of the longest shortest path between any pair of nodes. eccentricity measures the maximum distance from a node to any other node, the diameter is the maximum eccentricity among all nodes in the graph.
Eccentricity: $e(v) = \max_{u \in V} dist(v, u)$
Diameter: $D(G) = \max_{v \in V} e(v)$

Exercise 2:Mining An Online Social Network

	Medium	**Large**
2.1	8888	889007
2.2	2158	100312
2.4	1, 443, 1716, 8387, 2158, 8888	304, 38951, 60575, 705767, 99634, 888602
2.5	0.2751	0.2607
2.7	3.0767	1.57

Table 1: Answers to Exercise2 questions

Question 2.3:

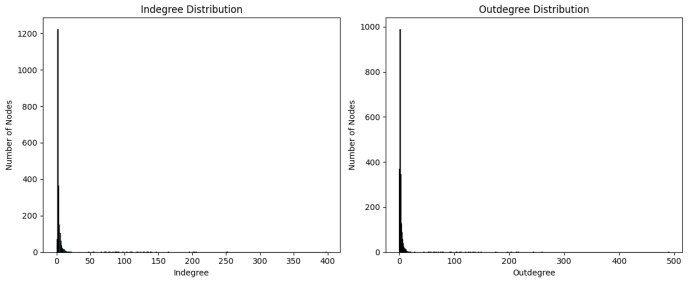

Figure 1: degree distributions for medium

Due to the data is huge of large graph, I take 1000 nodes randomly for a sample.

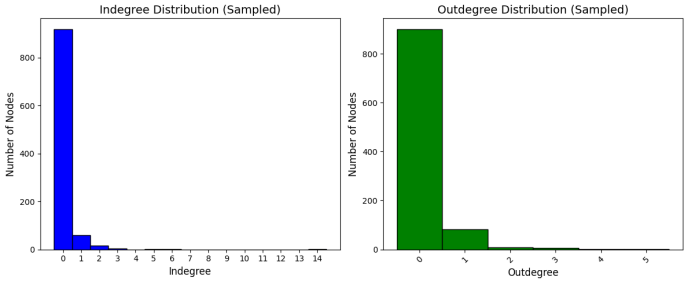

Figure 2: degree distributions for large

Question 2.4:

Get number of strongly and weakly connected components:

```
strongly_connected_components = list(nx.strongly_connected_components(G))
num_strongly_connected_components = len(strongly_connected_components)

weakly_connected_components = list(nx.weakly_connected_components(G))
num_weakly_connected_components = len(weakly_connected_components)
```

Get number of nodes and links of largest strongly and largest weakly connected component:

```
strongly_connected_components = list(nx.strongly_connected_components(G))
largest_scc = max(strongly_connected_components, key=len)

# Create a subgraph
largest_scc_subgraph = G.subgraph(largest_scc)

weakly_connected_components = list(nx.weakly_connected_components(G))
largest_wcc = max(weakly_connected_components, key=len)

largest_wcc_subgraph = G.subgraph(largest_wcc)

# Count the number
scc_nodes_num = largest_scc_subgraph.number_of_nodes()
scc_edges_num = largest_scc_subgraph.number_of_edges()

wcc_nodes_num = largest_wcc_subgraph.number_of_nodes()
wcc_edges_num = largest_wcc_subgraph.number_of_edges()
```

Question 2.5:

```
avg_clustering_coefficient = nx.average_clustering(G.to_undirected(), count_zeros=True)
avg_clustering_coefficient = round(avg_clustering_coefficient, 4)
```

I convert the directed graph into an undirected graph before calculating the average clustering coefficient.

Question 2.6:

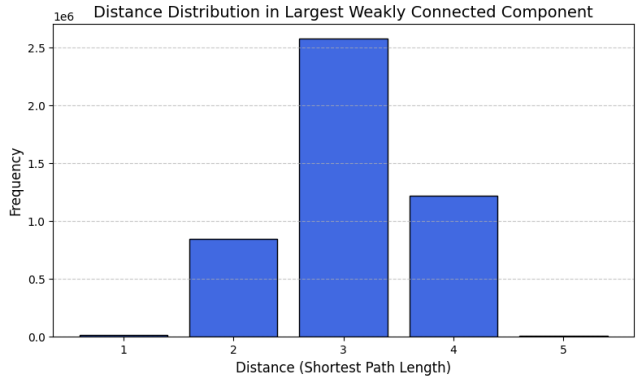

Figure 3: distance distribution in largest weakly connected for medium

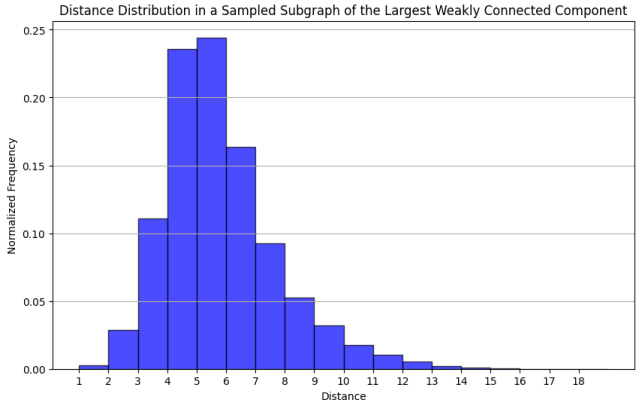

Figure 4: distance distribution in largest weakly connected for large

Due to the data is huge of large graph, I take 10000 nodes randomly for a sample.

Question 2.7:

```
    largest_wcc_subgraph = G.subgraph(largest_wcc).to_undirected()
if nx.is_connected(largest_wcc_subgraph):
    avg_path_length = nx.average_shortest_path_length(largest_wcc_subgraph)
    avg__path_length = round(avg_path_length, 4)
```

Question 2.8:

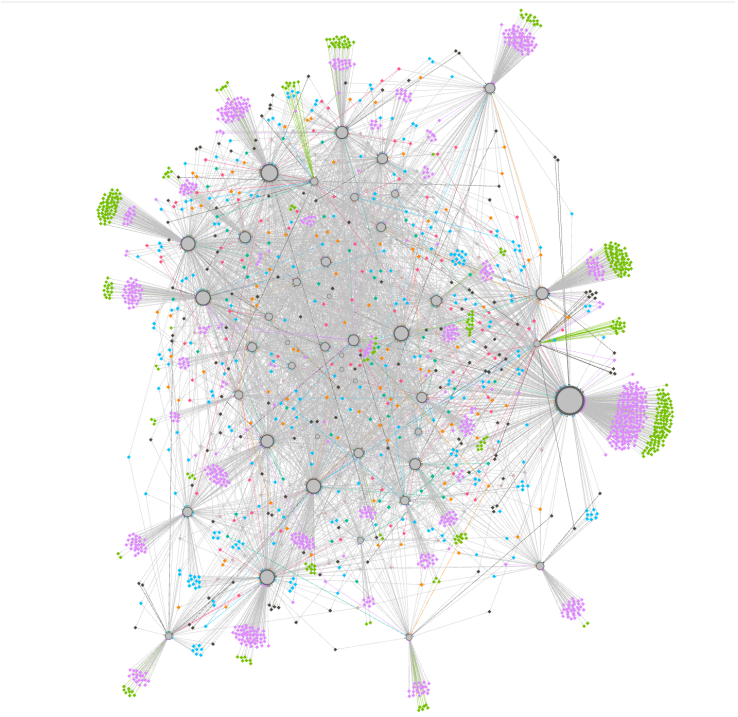

Figure 5: medium graph visualization

I use ForceAtlas2 which is a widely used graph layout algorithm for visualizing large networks. It is a force-directed layout algorithm that positions nodes in a way that reflects the graph's structure. The ForceAtlas2 layout algorithm places nodes closer together when they are more interconnected, forming clusters. The nodes are colored based on their degree. In the "Appearance" panel, the nodes are colored according to a degree partition, with different degrees corresponding to specific colours, and larger nodes represent nodes with a higher degree.