Social Network Analysis for Computer Scientists Assignment

S3977226 Zhipei Qin

October 2023

1 Exercise 1

1.1 Question 1.1

The combined degree $C(v)$ of a node $v \in V$ is defined as

$$C(v) = |N(v) \cup N'(v)|$$

Where V represents the set of all nodes. $N(v)$ represents the neighborhood of a node $v \in V$, and $N'(v)$ represents the reversed neighborhood of a node $v \in V$. $|S|$ represents the cardinality of set S.

1.2 Question 1.2

If a directed network is complete, then there are two opposite links connected between every two vertices in this network. This also means that every node's neighborhood contains all the other nodes in the network expect the node itself. Thus, the neighborhood of a node $v \in V$ in a completed directed network G is:

$$N(v) = V \setminus \{v\}$$

This equation holds true only if G is complete.

1.3 Question 1.3

The reversed k-neighborhood denotes the set of nodes that can link to nodes in W by following reversed edges in at most k steps. For the case $k = 0$ we have $N'_0(W) = W$. Then for $k > 0$ we have

$$N'_k(W) = N'(N'_{k-1}(W)) \cup N'_{k-1}(W)$$

1.4 Question 1.4

The *reciprocity* of a directed network $G = (V, E)$ can be defined as:

$$R = \frac{\sum_{v \in V} |N(v) \cap N'(v)|}{n}$$

Where V represents the set of all nodes. $N(v)$ represents the neighborhood of a node $v \in V$ and N' represents the reversed neighborhood of v. n represents the total notes in the network.

1.5 Question 1.5

If two nodes in a directed network can reach each other (i.e. there is a bidirectional path), then they must be in the same strongly connected component. Using the concept of k-neighborhood N_k, it can be expressed as two given nodes $u, v \in V$ are in the same strongly connected component only if

$$\exists k \leq n \ s.t. (u \in N_k(v) \wedge v \in N_k(u))$$

Design the following algorithm to determine whether two nodes u, v can reach each other and whether they are in a strongly connected component:

1. Initialize an empty stack to track the search process. Place the start node u on the stack and mark it as visited.

2. Loop from the starting node u until the stack is empty:
The node at the top of the stack (the current node) is popped.

3. Use depth-first search (DFS) algorithm. For the current node, iterate over all its neighborhood nodes:
If the neighborhood node is the target node v, it means that a path from u to v has been found, and the traversal stops.
If the neighborhood node is not the target node v and have not been visited, mark it as visited and push it onto the stack so that the search can continue in subsequent iterations.

4. If the stack is empty and the target node v is not found, then there is no path from u to v, indicating that the two nodes are not in a strongly connected component, return "False".

5. If there is a path from u to v, then set the starting node to v and perform another DFS algorithm to find if there is a path from v to u.

6. If node u is found in the second search, then node u and node v can reach each other, indicating that the two nodes are in a strongly connected component, return "True". If node u is not found, it means that the two nodes can not reach each other, indicating that the two nodes are not in a strongly connected

component, return "False".
Therefore, if the two points u, v are in a strongly connected component of a given network, then the algorithm must return "True", and otherwise return "False".

1.6 Question 1.6

Based on depth-first search (DFS) and the concepts of k-neighborhood and reverse k-neighborhood, algorithms can be designed to count the number of triangles in a given network as follows:
1. Set a counter $NodeCount$ and assign its initial value to $NodeCount = 0$.
2. Traverse each node v in the directed network $G = (V, E)$ and do the following for each node:

2.1 Initializes an empty list $NodeList$.

2.2 Use depth-first search (DFS) algorithm. For node v, traverse all of its 1-neighborhood nodes.

2.3 For each 1-neighborhood node u of node v, DFS is performed again to obtain 1-neighborhood nodes of each u, which are also 2-neighborhood nodes of node v.

2.4 Add all nodes obtained in 2.3 to the List without removing duplicate values.
3. For each value in the $NodeList$, determine whether its corresponding nodes are reverse 1-neighborhood nodes of node v. If true, then $NodeCount \rightarrow NodeCount + 1$.
After traversing all nodes in G, the $NodeCount/3$ is the total number of triangles in the network.

For each node v,
In step 2.2, DFS needs to visit all nodes and edges in the worst case, and the time complexity is $O(n+m)$, which n is the number of nodes $|V|$, and m is the number of links $|E|$.
In step 2.3, DFS is performed again, and the time complexity is $O(n+m)$.
In step 3, it is necessary to determine whether each node stored in $NodeList$ is the reverse 1-neighborhood node of node v, so the time complexity is $O(n)$.
Because steps 2 and 3 are executed for every node in G, the time complexity of my algorithm is $O(n * (n+m))$.

1.7 Question 1.7

Design the following algorithm to compute the number of nodes for which this paradox does not hold:

1. Set a counter $NodeCount$ and assign its initial value to $NodeCount = 0$.

2. Initialize an empty list $DegreeList$, used to store the degree of the node.

3. Traverse each node v in the undirected network $G = (V, E)$, and perform the following operations for each node:

   3.1 Iterate through the edges of the network. For each edge (v, u) which is not visited, increase the degree of node v by 1 and the degree of node u by 1 in the list.

   3.2 Mark edge (v, u) as visited.

   3.3 After traversing all edges in E, the degree of each node in G would be obtained in $DegreeList$.

4. Traverse each node v. For each node v, traverse all its 1-neighborhood nodes. If the degree of v is greater than the average degree of all its 1-neighborhood nodes, then set $NodeCount \rightarrow NodeCount + 1$, otherwise $NodeCount \rightarrow NodeCount$.

5. After traversing all nodes in G, the value of $NodeCount$ is the number of nodes for which the $FriendshipParadox$ does not hold.

In step 3, in the worst case, all nodes and edges need to be visited, and the time complexity is $O(n+m)$, which n is the number of nodes $|V|$, and m is the number of links $|E|$.

In step 4, all 1-neighborhood nodes of each node need to be traversed, and the time complexity is also $O(n + m)$, because in the worst case, the number of 1-neighborhood nodes of each node will be the same as the number of edges.

Therefore, the time complexity of my algorithm is $O(n + m)$.

1.8 Question 1.8

(a) When an edge $\{v, w\} \in E$ is deleted, there are two situations that cause the *state* of a node to change:
1. The *state* of node v and node w will change because they are no longer directly connected. It can be expressed as: before deleting the edge, $v \in N_1(w)$ and $w \in N_1(v)$; After deleting the edge, $v \notin N_1(w)$ and $w \notin N_1(v)$.
2. Other nodes that are no longer reachable to v or w in the k-neighborhood due to edge deletion: These nodes can originally reach v or w in the k-neighborhood through the edges $\{v, w\}$, but because the edge is deleted, the node cannot reach v or w, or requires more steps to reach. Assuming that the node is u, the situation where its *state* is affected in the k-neighborhood can be expressed as, setting a specific int value k, before deleting the edge, $u \in N_k(v)$ and $u \in N_k(w)$; After deleting the edge, $u \notin N_k(v)$ or $u \notin N_k(w)$.

(b) When an edge $\{v, w\} \in E$ is added, there are two situations that cause the *state* of a node to change:
1. The *state* of node v and node w will change because they become directly connected. It can be expressed as: before adding the edge, $v \notin N_1(w)$ and $w \notin N_1(v)$, after adding the edge, $v \in N_1(w)$ and $w \in N_1(v)$.
2. Other nodes that are reachable to v or w in the k-neighborhood due to the addition of edges: These nodes are originally unable to reach v or w or require more steps to reach, but since edges are added, these nodes can reach v or w within the k-neighborhood. Assuming that the node is u, the situation where its *state* is affected in the k-neighborhood can be expressed as, setting a specific int value k, before adding an edge, $u \notin N_k(v)$ or $u \notin N_k(w)$; before adding an edge After adding an edge, $u \in N_k(v)$ and $u \in N_k(w)$.

1.9 Question 1.9

Design the following algorithm to compute the network radius using the notion of a (k-)neighborhood.
1. Traverse each node v in the network, and perform the following operations for each node v:

1.1 Initialize eccentricity=0.

1.2 Set k as an integer value, traverse each node u in $N_k(v)$, and calculate the shortest path distance S_{vu} from v to u. If $eccentricity < S_{vu}$, then

$eccentricity \rightarrow S_{vu}$; Otherwise $eccentricity \rightarrow eccentricity$.

1.3 After traversing all nodes u, the eccentricity of v within its k-neighborhood can be obtained. The eccentricity can be represented symbolically as $max(S_{vu})$ where $u \in N_k(v)$.

2. After calculating the eccentricity of all nodes v, find the minimum eccentricity value among all nodes. This minimum eccentricity value $min(eccentricity)$ represents the k-neighborhood network radius.

2 Exercise 2

2.1 Question 2.1

In a directed network, a directed link corresponds to a directed edge of the network. Therefore, the number of directed links of this network is equal to the number of directed links of this network.

By using the $.number\_of\_edges()$ function in $NETWORKX$, it can be obtained that the number of the directed links for the network in *medium.tsv* is 16329, and the number of the directed links for the network in *large.tsv* is 149755.

2.2 Question 2.2

By using the $.number\_of\_nodes()$ function in $NETWORKX$, it can be obtained that the number of the nodes for the network in *medium.tsv* is 5895, and the number of the nodes for the network in *large.tsv* is 41767.

2.3 Question 2.3

The indegree and outdegree of each node in a network can be obtained by function $.in\_degree()$ and $.out\_degree()$. The indegree and outdegree distributions can be drawn using $Matplotlib$:

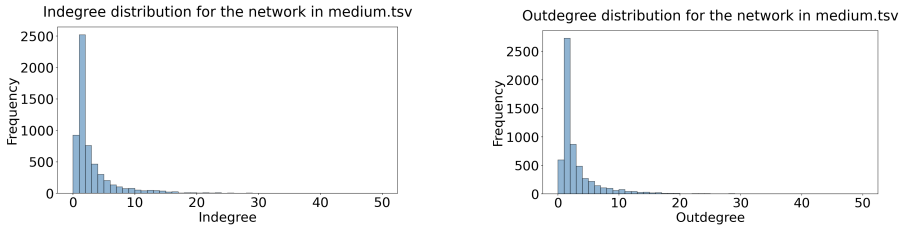

Figure 1: Indegree and outdegree distributions for the network in medium.tsv

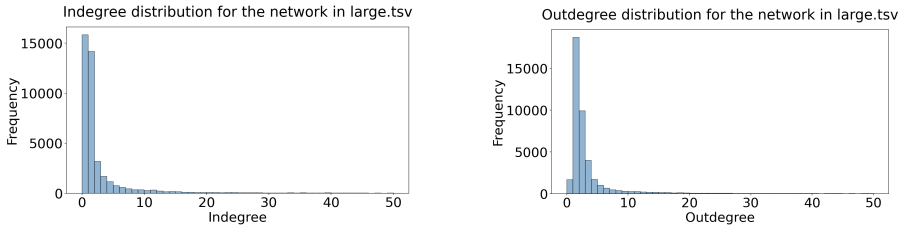

Figure 2: Indegree and outdegree distributions for the network in large.tsv

2.4 Question 2.4

The number of weakly connected components and strongly connected components can be obtained by function *number_weakly_connected_components* and *number_strongly_connected_components*.

For the network in *medium.tsv*, the number of weakly connected components is 200 and the number of strongly connected components is 1804. For the network in *large.tsv*, the number of weakly connected components is 647 and the number of strongly connected components is 19250.

In order to calculate the number of nodes and links of the largest weakly connected component, we need to first use function *networkx.weakly_connected_components* to calculate the largest weakly connected component. Similarly, to calculate the number of nodes and links of the largest strongly connected component, we need to first use function *networkx.strongly_connected_components* to calculate the largest strongly connected component.

For the network in *medium.tsv*, the largest strongly connected component

has 3677 notes and 13166 links and the largest weakly connected component has 5265 notes and 15623 links.

For the network in *large.tsv*, the largest strongly connected component has 21226 notes and 120614 links and the largest weakly connected component has 39660 notes and 147862 links.

2.5 Question 2.5

The average clustering coefficient of a network can be obtained by function *networkx.average_clustering*(). The average clustering coefficient for the network in *medium.tsv* is 0.164, while the average clustering coefficient for the network in *large.tsv* 0.254 (both takes to three decimal places).

Two types of closed triplets, directed closed triplets and bidirectional closed triplets, are considered in the calculation of clustering coefficients of directed networks. A directed closed triplet consists of three nodes with a directed edge between each point and the other two points. In a bidirectional closed triplet, two nodes have a directed edge between each other.

To calculate a node's clustering coefficient, divide the number of closed triplets (including directed and bidirectional closed triplets) owned by the node by the total number of possible closed triplets. Finally, the average clustering coefficients of all nodes are averaged to obtain the average clustering coefficients of the whole network.

2.6 Question 2.6

The distance distributions of the largest weakly connected component for two networks are shown as below:

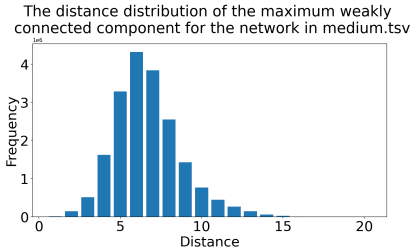

Figure 3: The distance distribution of the maximum weakly connected component

2.7 Question 2.7

2.8 Question 2.8

The visualization of the social network in *medium.tsv* using Gephi is shown as below.

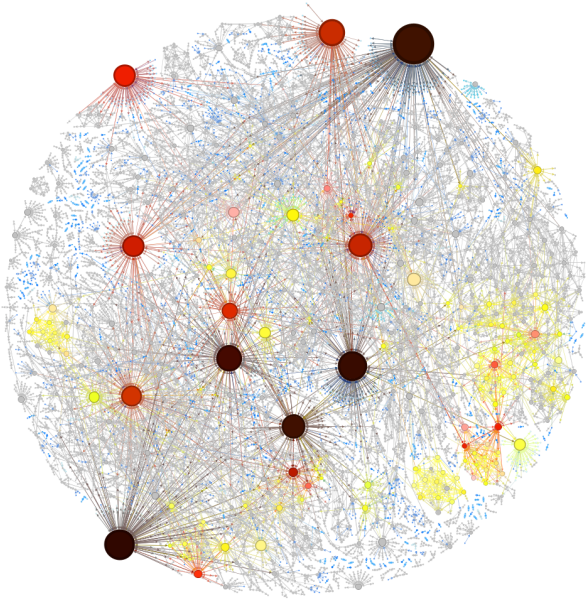

Figure 4: Visualization for the network in medium.tsv

1. Algorithms and parameters

This visualization choose ForceAtlas 2 to be the visualization algorithm. This algorithm improves performance by reducing the number of interactions between compute nodes and is therefore suitable for networks containing a large number of nodes and edges.

In the visualization attempt, several parameters of the algorithm are adjusted. Implement this algorithm with enabling parameter "strong gravity" and set "scaling =50.0", which gets a better layout. When the image stabilizes, enable parameter "Prevent overlap". Run "Expansion" to make some more space between the nodes, and finally run "Label Adjust" to prevent label overlap.

2. Measurements

The size and color of the node can reflect different influence of the node. The size of a node is related to its degree. The larger the node size, the greater its

degree, that is, the greater the number of edges directly connected to the node. The degree of a node can reflect how many social connections a person has in a social network. Individuals with more social connections are generally thought to be more influential in social networks, or they may be more important social connections.

The color of a node reflects the Eigenvector centrality of this node. In the visualization, the nodes with the largest Eigenvector centrality are colored dark red, the nodes with the second largest centrality are colored light red, and the nodes with the second largest centrality are colored yellow. Let the nodes with the smallest value of Eigenvector centrality be colored blue. Eigenvector centrality can be used to discover opinion leaders in social networks. These people who has high value of Eigenvector centrality are often at the center of the conversation and are also more likely to be the starting point for information dissemination because they have more social connections and are therefore more likely to spread the information to more people.

3. Analysis

In the resulting visualizations, several nodes were larger than others, suggesting that they had established social relationships with more people. It can also be seen that nodes with a relatively large size tend to be dark or light red, that is, these nodes also have relatively large Eigenvector centrality. This result is intuitive. This indicates that these nodes are important central nodes in their own right and that these nodes also have extensive connections to other important nodes. From the perspective of these two indicators, these nodes are likely to have greater influence and appeal in the entire social network, can radiate a wider interpersonal circle, and are also more important social connection points. The person corresponding to this node is also likely to be a key decision-maker or leader in the network.

It can also be noticed that some points converge on a partition or sub-graph of the network (for example, some points painted yellow on the right side of the viewable). These partitions are relatively independent from the rest of the network, and there are more internal connections between nodes within the partition. At the same time, these points don't radiate much outward. These points may belong to the same fixed social community or group, and there is more interconnection between them. The social structure of social networks often leads to the phenomenon of aggregation of points, and the nodes in the

same community usually have a high degree of interconnection. These users may have similar interests or personalities, so they are more likely to connect and be attracted to each other. Some red nodes in a group of yellow nodes have higher Eigenvector centrality, in some societies, these nodes may be more active or central, they will become the organization and leader of the community or group, and attract nodes belonging to the community to connect with them.

3 Appendix source code

```
import pandas as pd
from pandas import read_csv
import networkx as nx
import matplotlib as mpl
from matplotlib import rcParams
import matplotlib.pyplot as plt

#Import Data
names = ['userA','userB'] #custom column name
medium = pd.read_csv(r'E:\SNA\medium.tsv', sep='\t', header=None, names=names)
large = pd.read_csv(r'E:\SNA\large.tsv', sep='\t', header=None, names=names)

#Q2.1
#Calculate the number of the directed links for the network in medium.tsv
G1 = nx.DiGraph()
edges1 = medium[['userA','userB']].values.tolist()
G1.add_edges_from(edges1)
num_edges1 = G1.number_of_edges()

#Calculate the number of the directed links for the network in large.tsv
G2 = nx.DiGraph()
edges2 = large[['userA','userB']].values.tolist()
```

```
G2.add_edges_from(edges2)
num_edges2 = G2.number_of_edges()
print(num_edges1,num_edges2)

#Q2.2
#Calculate the number of the nodes for the network in medium.tsv and large.tsv
num_nodes1 = G1.number_of_nodes()
num_nodes2 = G2.number_of_nodes()
print(num_nodes1,num_nodes2)

#Q2.3
#Draw the indegree and outdegree distributions
Indegree1 = G1.in_degree()
Indegree2 = G2.in_degree()
Outdegree1 = G1.out_degree()
Outdegree2 = G2.out_degree()

in_degrees1 = dict(Indegree1)
indegree_values1 = list(in_degrees1.values())
in_degrees2 = dict(Indegree2)
indegree_values2 = list(in_degrees2.values())
out_degrees1 = dict(Outdegree1)
outdegree_values1 = list(out_degrees1.values())
out_degrees2 = dict(Outdegree2)
outdegree_values2 = list(out_degrees2.values())

rcParams['axes.titlepad'] = 20
fig = plt.figure(figsize=(12,6), dpi=150)
plt.hist(indegree_values1, bins=50, facecolor="steelblue", edgecolor="black",
         range=(0,50),alpha=0.6)
plt.xlabel("Indegree",fontsize=26)
plt.ylabel("Frequency",fontsize=26)
plt.tick_params(axis='x', labelsize=26)
plt.tick_params(axis='y', labelsize=26)
plt.title("Indegree distribution for the network in medium.tsv",fontsize=28)
plt.show()
```

```
rcParams['axes.titlepad'] = 20
fig = plt.figure(figsize=(12,6), dpi=150)
plt.hist(indegree_values2, bins=50, facecolor="steelblue", edgecolor="black",
         range=(0,50),alpha=0.6)
plt.xlabel("Indegree",fontsize=26)
plt.ylabel("Frequency",fontsize=26)
plt.tick_params(axis='x', labelsize=26)
plt.tick_params(axis='y', labelsize=26)
plt.title("Indegree distribution for the network in large.tsv",fontsize=28)
plt.show()

rcParams['axes.titlepad'] = 20
fig = plt.figure(figsize=(12,6), dpi=150)
plt.hist(outdegree_values1, bins=50, facecolor="steelblue", edgecolor="black",
         range=(0,50),alpha=0.6)
plt.xlabel("Outdegree",fontsize=26)
plt.ylabel("Frequency",fontsize=26)
plt.tick_params(axis='x', labelsize=26)
plt.tick_params(axis='y', labelsize=26)
plt.title("Outdegree distribution for the network in medium.tsv",fontsize=28)
plt.show()

rcParams['axes.titlepad'] = 20
fig = plt.figure(figsize=(12,6), dpi=150)
plt.hist(outdegree_values2, bins=50, facecolor="steelblue", edgecolor="black",
         range=(0,50),alpha=0.6)
plt.xlabel("Outdegree",fontsize=26)
plt.ylabel("Frequency",fontsize=26)
plt.tick_params(axis='x', labelsize=26)
plt.tick_params(axis='y', labelsize=26)
plt.title("Outdegree distribution for the network in large.tsv",fontsize=28)
plt.show()

#Q2.4
#Compute weakly connected components and strongly connected components
weakly_connected_components1 = nx.number_weakly_connected_components(G1)
print(weakly_connected_components1)
```

```
weakly_connected_components2 = nx.number_weakly_connected_components(G2)
print(weakly_connected_components2)
strongly_connected_components1 = nx.number_strongly_connected_components(G1)
print(strongly_connected_components1)
strongly_connected_components2 = nx.number_strongly_connected_components(G2)
print(strongly_connected_components2)

# Calculate the number of nodes and links of the largest strongly and largest weakly connect
largest_weakly_connected_component1 = max(nx.weakly_connected_components(G1), key=len)
largest_weakly_connected_component2 = max(nx.weakly_connected_components(G2), key=len)
largest_strongly_connected_component1 = max(nx.strongly_connected_components(G1), key=len)
largest_strongly_connected_component2 = max(nx.strongly_connected_components(G2), key=len)
num_nodes_in_largest_weakly_connected1 = len(largest_weakly_connected_component1)
num_edges_in_largest_weakly_connected1 = G1.subgraph(largest_weakly_connected_component1).nu
num_nodes_in_largest_strongly_connected1 = len(largest_strongly_connected_component1)
num_edges_in_largest_strongly_connected1 = G1.subgraph(largest_strongly_connected_component1

num_nodes_in_largest_weakly_connected2 = len(largest_weakly_connected_component2)
num_edges_in_largest_weakly_connected2 = G2.subgraph(largest_weakly_connected_component2).nu
num_nodes_in_largest_strongly_connected2 = len(largest_strongly_connected_component2)
num_edges_in_largest_strongly_connected2 = G2.subgraph(largest_strongly_connected_component2
print(num_nodes_in_largest_weakly_connected1,num_edges_in_largest_weakly_connected1,
      num_nodes_in_largest_strongly_connected1,num_edges_in_largest_strongly_connected1)
print(num_nodes_in_largest_weakly_connected2,num_edges_in_largest_weakly_connected2,
      num_nodes_in_largest_strongly_connected2,num_edges_in_largest_strongly_connected2)


#Q2.5
#Compute the average clustering coefficient
Coefficient1 = nx.average_clustering(G1)
Coefficient2 = nx.average_clustering(G2)
print(Coefficient1,Coefficient2)

#Q2.6
#Compute The distance distribution of the maximum weakly connected component
largest_wcc = max(nx.weakly_connected_components(G2), key=len)
```

```
distance_distribution = {}
for source_node in largest_wcc:
    shortest_paths = nx.single_source_shortest_path_length(G2, source_node)
    for target_node, distance in shortest_paths.items():
        if target_node != source_node:
            if distance in distance_distribution:
                distance_distribution[distance] += 1
            else:
                distance_distribution[distance] = 1

distances = list(distance_distribution.keys())
frequencies = [distance_distribution[distance] for distance in distances]

rcParams['axes.titlepad'] = 20
fig = plt.figure(figsize=(12,6), dpi=150)
plt.bar(distances, frequencies)
plt.xlabel("Distance",fontsize=26)
plt.ylabel("Frequency",fontsize=26)
plt.tick_params(axis='x', labelsize=26)
plt.tick_params(axis='y', labelsize=26)
plt.title('The distance distribution of the maximum weakly \n connected component for the ne
plt.show()
```