Complex Network Modeling Project

1 Network Construction

I chose to generate a scale-free network, which is a network with a "power-law" degree distribution, that is, a few nodes have a large number of connections, while most nodes have only a few connections. This type of network simulates many complex networks in the real world, such as social networks, the Internet, citation networks, etc.

The steps of its construction are:

1. Initialize the network: first build a small complete graph;
2. Add nodes: each time a new node is added, this node will be connected to the existing nodes in the network;
3. Prioritize connection: new nodes are more likely to connect to nodes with higher degrees;

Node: represents an individual;

Edge: represents the relationship or connection between individuals;

```
def create_scale_free_network(n, m):

    edges = []  # 网络中的边列表
    nodes = list(range(m))  # 初始节点集

    # 初始完全图
    for i in range(m):
        for j in range(i + 1, m):
            edges.append((i, j))

    # 逐步添加节点
    for new_node in range(m, n):
        target_nodes = set()
        while len(target_nodes) < m:
            # 根据度数优先选择已有节点
            potential_target = random.choices(nodes, weights=[degree_count(n,
edges, node) for node in nodes])[0]
            target_nodes.add(potential_target)

        # 将新节点连接到目标节点
        for target in target_nodes:
            edges.append((new_node, target))
        nodes.append(new_node)

    return edges
```

Network Evolution

To make the generated scale-free network evolve further, the following methods can be considered:

1. Node addition: Continue to add new nodes and connect them to existing nodes in the same way, selecting based on degree.

2. Edge reconnection: Randomly select some edges and reconnect them to other nodes to introduce more randomness and change the network structure.
3. Node or edge deletion: Randomly delete some nodes or edges and observe how the network adapts to the changes.
4. Introduce dynamics: Set rules, such as adding connections based on the activity of nodes (such as activity frequency).
5. Propagation process: Introduce information propagation models, such as epidemic propagation models, and observe the evolution of the network.

```
def evolve_network(original_edges, new_nodes, m, rewire_probability=0.1):
    edges = original_edges.copy()
    current_nodes = list(set([node for edge in edges for node in edge]))

    for new_node in range(max(current_nodes) + 1, max(current_nodes) + new_nodes
+ 1):
        target_nodes = set()
        while len(target_nodes) < m:
            potential_target = random.choices(current_nodes, weights=
[degree_count(len(edges), edges, node) for node in current_nodes])[0]
            target_nodes.add(potential_target)
        for target in target_nodes:
            edges.append((new_node, target))
        current_nodes.append(new_node)

    for i, (u, v) in enumerate(edges):
        if random.random() < rewire_probability:
            new_target = random.choice(current_nodes)
            edges[i] = (u, new_target)

    return edges
```

2 Network Analysis

Calculate the node degree distribution of the network

Just traverse every edge of the graph.

```
def degree_distribution(n, edges):

    degree_dict = {i: 0 for i in range(n)}
    for edge in edges:
        degree_dict[edge[0]] += 1
        degree_dict[edge[1]] += 1

    degree_freq = {}
    for degree in degree_dict.values():
        if degree not in degree_freq:
            degree_freq[degree] = 0
        degree_freq[degree] += 1

    return degree_freq
```

Calculate the average shortest path length

Use breadth-first search to find the shortest path between each pair of nodes, and then sum them up to get the average shortest path length.

```
def bfs_shortest_path(n, edges, start):

    distances = {i: float('inf') for i in range(n)}
    distances[start] = 0
    queue = deque([start])

    while queue:
        node = queue.popleft()
        for edge in edges:
            if node in edge:
                neighbor = edge[0] if edge[1] == node else edge[1]
                if distances[neighbor] == float('inf'):
                    distances[neighbor] = distances[node] + 1
                    queue.append(neighbor)
    return distances

def average_shortest_path_length(n, edges):

    total_path_length = 0
    num_pairs = 0

    for node in range(n):
        distances = bfs_shortest_path(n, edges, node)
        for target, distance in distances.items():
            if distance != float('inf') and target != node:
                total_path_length += distance
                num_pairs += 1

    return total_path_length / num_pairs if num_pairs > 0 else float('inf')
```

Calculate the clustering coefficient

Steps:

1. Find the neighbors of the current node;
2. Calculate the number of edges between neighbors;
3. Finally, calculate the clustering coefficient;

```
def clustering_coefficient(n, edges):
    clustering_coeffs = []

    for node in range(n):

        neighbors = set()
        for edge in edges:
            if edge[0] == node:
                neighbors.add(edge[1])
            elif edge[1] == node:
                neighbors.add(edge[0])


        if len(neighbors) < 2:
            clustering_coeffs.append(0)
            continue
```

```
        links_between_neighbors = 0
        for neighbor1 in neighbors:
            for neighbor2 in neighbors:
                if neighbor1 != neighbor2 and (neighbor1, neighbor2) in edges or
(neighbor2, neighbor1) in edges:
                    links_between_neighbors += 1


        clustering_coeffs.append(links_between_neighbors / (len(neighbors) *
(len(neighbors) - 1)))

    return sum(clustering_coeffs) / n
```

To further visualize it:

1. Calculate the local clustering function of each node;
2. Calculate the clustering coefficient;
3. Then draw its scatter plot and bar chart;

```
def calculate_local_clustering_coefficients(n, edges):
    local_clustering_coeffs = []

    for node in range(n):
        neighbors = set()
        for edge in edges:
            if edge[0] == node:
                neighbors.add(edge[1])
            elif edge[1] == node:
                neighbors.add(edge[0])

        if len(neighbors) < 2:
            local_clustering_coeffs.append(0)
            continue

        links_between_neighbors = 0
        neighbors_list = list(neighbors)
        for i in range(len(neighbors_list)):
            for j in range(i + 1, len(neighbors_list)):
                if (neighbors_list[i], neighbors_list[j]) in edges or
(neighbors_list[j], neighbors_list[i]) in edges:
                    links_between_neighbors += 1


        local_clustering_coeffs.append(links_between_neighbors / (len(neighbors)
* (len(neighbors) - 1) / 2))

    return local_clustering_coeffs

def plot_local_clustering_coefficients(local_clustering_coefficients):
    nodes = list(range(len(local_clustering_coefficients)))
    clustering_values = local_clustering_coefficients

    plt.figure(figsize=(8, 6))
    plt.scatter(nodes, clustering_values, color='purple', alpha=0.7)
    plt.title("Local Clustering Coefficients of Nodes")
    plt.xlabel("Node")
```

```
    plt.ylabel("Local Clustering Coefficient")
    plt.ylim(0, 1)
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.show()

def plot_average_clustering_coefficient(avg_clustering_coefficient):

    plt.figure(figsize=(6, 4))
    plt.bar(['Network'], [avg_clustering_coefficient], color='skyblue',
edgecolor='black')
    plt.title("Average Clustering Coefficient of the Network")
    plt.ylabel("Clustering Coefficient")
    plt.ylim(0, 1)
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()
```

3 Simulating Dynamic Behavior

Two types of attacks are simulated: random attacks and intentional attacks.

Simulated random attacks on the network

Random attacks randomly shuffle the order of nodes and then remove nodes and related edges.

```
def simulate_random_attack(n, edges):

    nodes = list(range(n))
    random.shuffle(nodes)
    sizes = []

    for node in nodes:
        edges = [(u, v) for u, v in edges if u != node and v != node]
        if not edges:
            break
        largest_cc_size = get_largest_connected_component_size(n, edges)
        sizes.append(largest_cc_size)

    return sizes
```

Simulating intentional attacks on the network

Intentional attacks first sort the degrees from large to small, and then remove the corresponding nodes and related edges from the beginning.

```
def simulate_targeted_attack(n, edges):
    nodes = list(range(n))
    degrees = {node: degree_count(n, edges, node) for node in nodes}
    nodes_sorted_by_degree = sorted(nodes, key=lambda x: degrees[x],
reverse=True)
    sizes = []

    for node in nodes_sorted_by_degree:
        edges = [(u, v) for u, v in edges if u != node and v != node]
        if not edges:
            break
```

```
        largest_cc_size = get_largest_connected_component_size(n, edges)
        sizes.append(largest_cc_size)

    return sizes
```

Calculate the size of the largest connected subgraph

We measure the connectivity and impact of the network after being attacked by calculating the size of the largest connected subgraph in the current graph. We perform a depth-first search by building a stack, recording the longest path each time and recording it in `largest_size`, and then traversing each node violently to get the size of the largest connected subgraph.

```
def get_largest_connected_component_size(n, edges):
    visited = [False] * n

    def dfs(node):
        stack = [node]
        size = 0
        while stack:
            current = stack.pop()
            if not visited[current]:
                visited[current] = True
                size += 1
                for edge in edges:
                    if edge[0] == current and not visited[edge[1]]:
                        stack.append(edge[1])
                    elif edge[1] == current and not visited[edge[0]]:
                        stack.append(edge[0])
        return size

    largest_size = 0
    for node in range(n):
        if not visited[node]:
            component_size = dfs(node)
            if component_size > largest_size:
                largest_size = component_size
    return largest_size
```

4 Calculate the coreness of nodes in the constructed graph

Calculate the core degree of the nodes and arrange them according to the size of the degree. First remove the nodes with small degrees, then update the core degree, then remove the current node and update the degree of its neighbors. Remember to update the queue every time.

```
def calculate_core_number(n, edges):
    degree = {i: 0 for i in range(n)}
    for u, v in edges:
        degree[u] += 1
        degree[v] += 1

    core_number = {i: 0 for i in range(n)}

    nodes = sorted(degree.keys(), key=lambda x: degree[x])
```

```
    while nodes:
        node = nodes.pop(0)
        current_degree = degree[node]

        core_number[node] = current_degree

        for u, v in edges:
            if u == node or v == node:
                neighbor = v if u == node else u
                if degree[neighbor] > current_degree:
                    degree[neighbor] -= 1

        nodes = sorted(nodes, key=lambda x: degree[x])

    return core_number
```

5 Whether is a Small World

- Calculate the **clustering coefficient** and **average path length** of the target networ
- Creating **random networks** and comparing them
- Determining small-world propertie

```
def small_world_test(edges, num_randomizations=100):
    G = nx.Graph()
    G.add_edges_from(edges)

    if not nx.is_connected(G):
        return False, None, None, None, None

    target_clustering = nx.average_clustering(G)
    target_avg_path_length = nx.average_shortest_path_length(G)

    random_clustering = []
    random_avg_path_length = []

    for _ in range(num_randomizations):
        random_graph = nx.gnm_random_graph(len(G.nodes()), len(G.edges()))

        if nx.is_connected(random_graph):
            random_clustering.append(nx.average_clustering(random_graph))

 random_avg_path_length.append(nx.average_shortest_path_length(random_graph))

    if random_clustering and random_avg_path_length:
        avg_random_clustering = np.mean(random_clustering)
        avg_random_path_length = np.mean(random_avg_path_length)

        is_small_world = (target_clustering > avg_random_clustering) and (
                    target_avg_path_length < avg_random_path_length)
        return is_small_world, target_clustering, target_avg_path_length,
avg_random_clustering, avg_random_path_length
    else:
        return False, None, None, None, None
```

Result

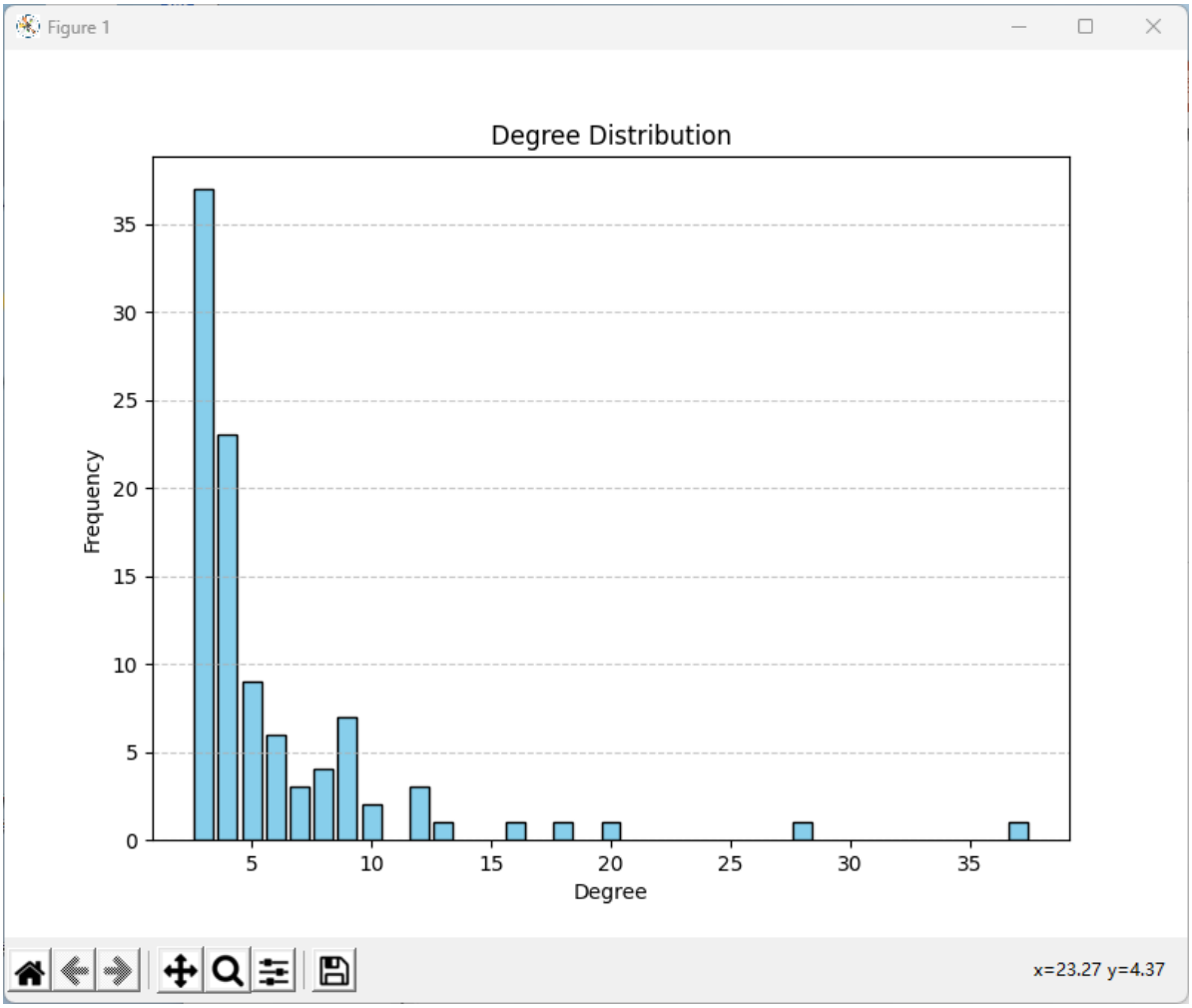
Figure 1
Degree Distribution
Frequency
Degree
0
5
10
15
20
25
30
35
x=23.27 y=4.37

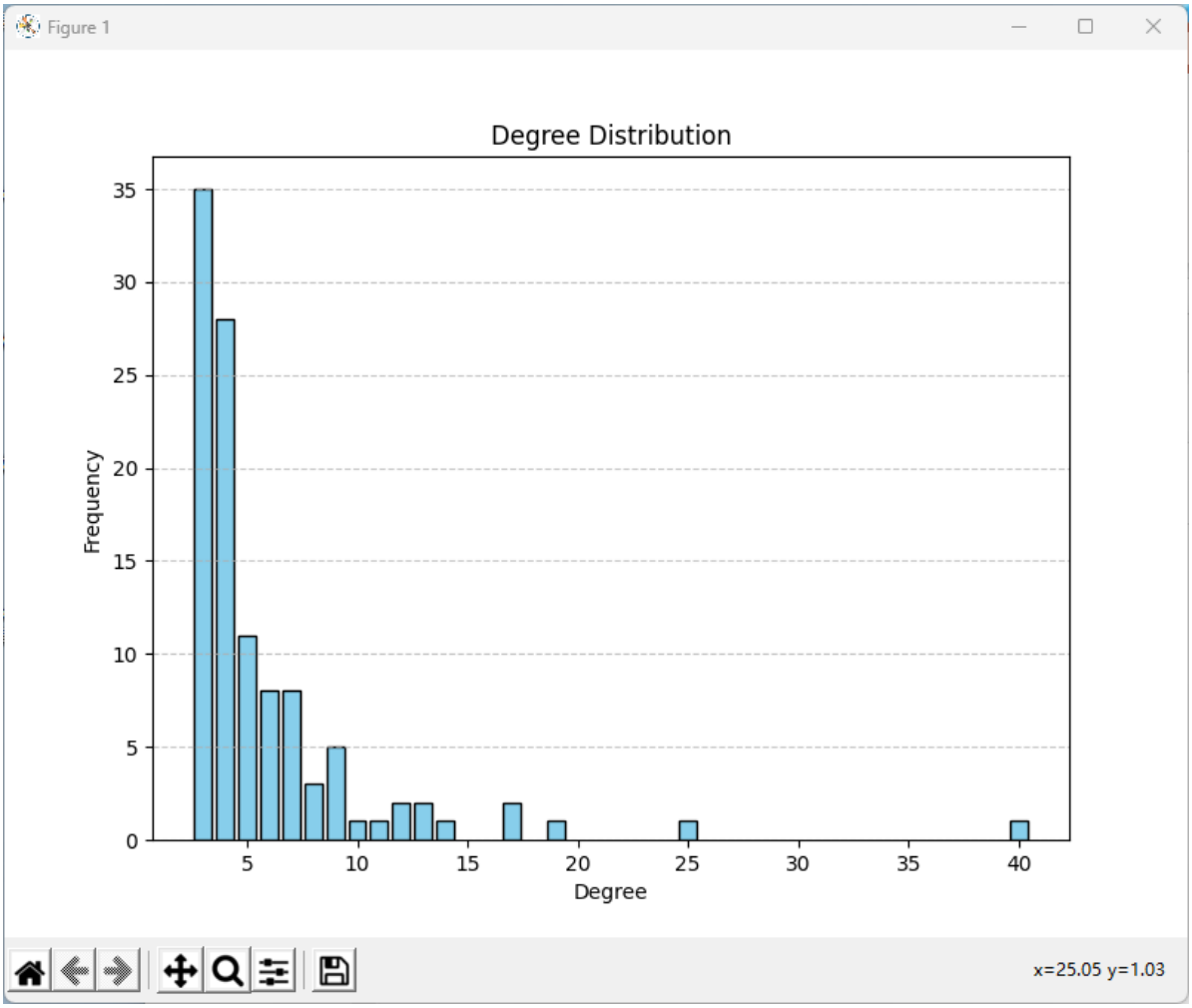
Figure 1
Degree Distribution
Frequency
Degree
0
5
10
15
20
25
30
35
40
x=25.05 y=1.03

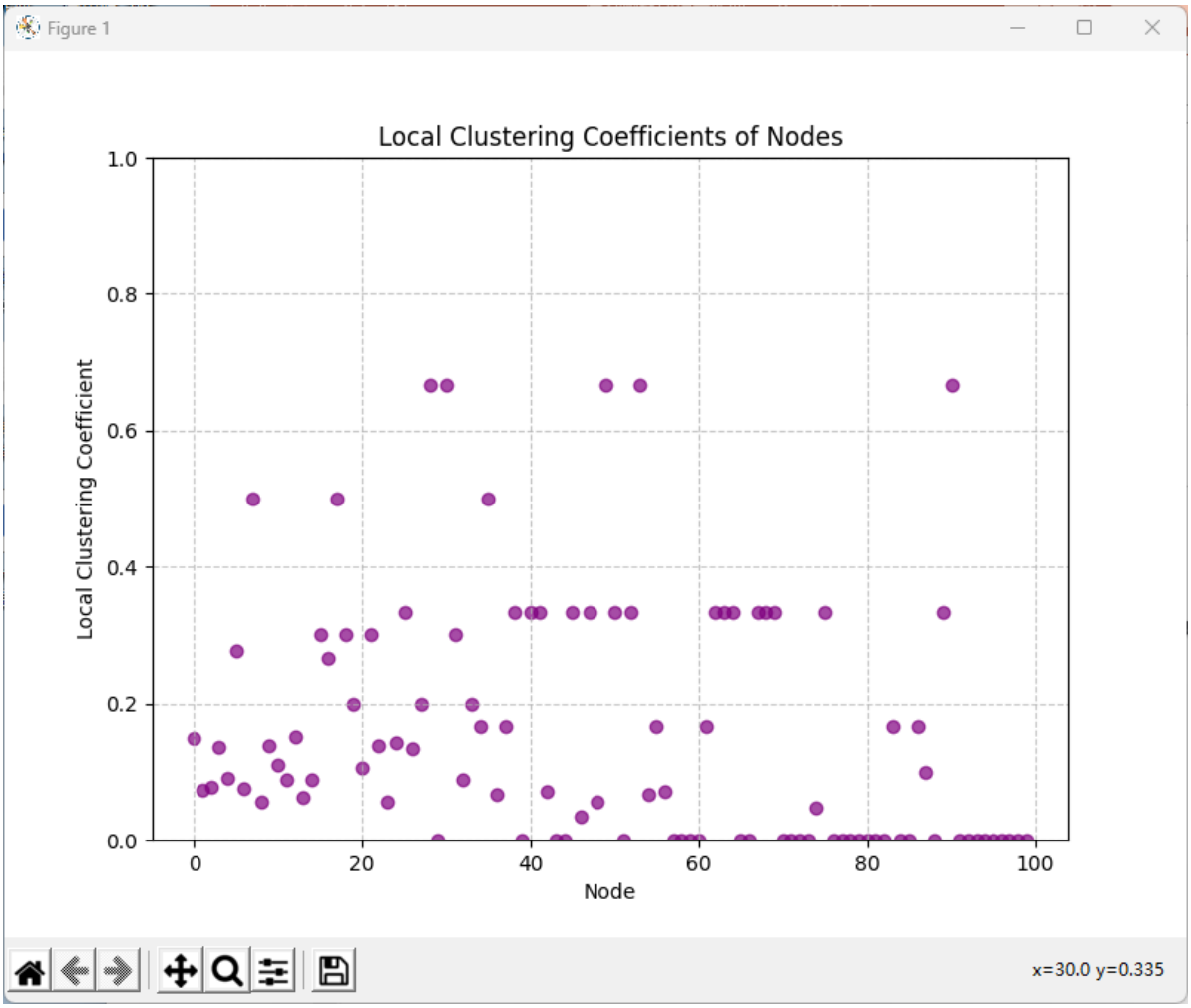
Figure 1
Local Clustering Coefficients of Nodes
Local Clustering Coefficient
Node
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100
x=30.0 y=0.335

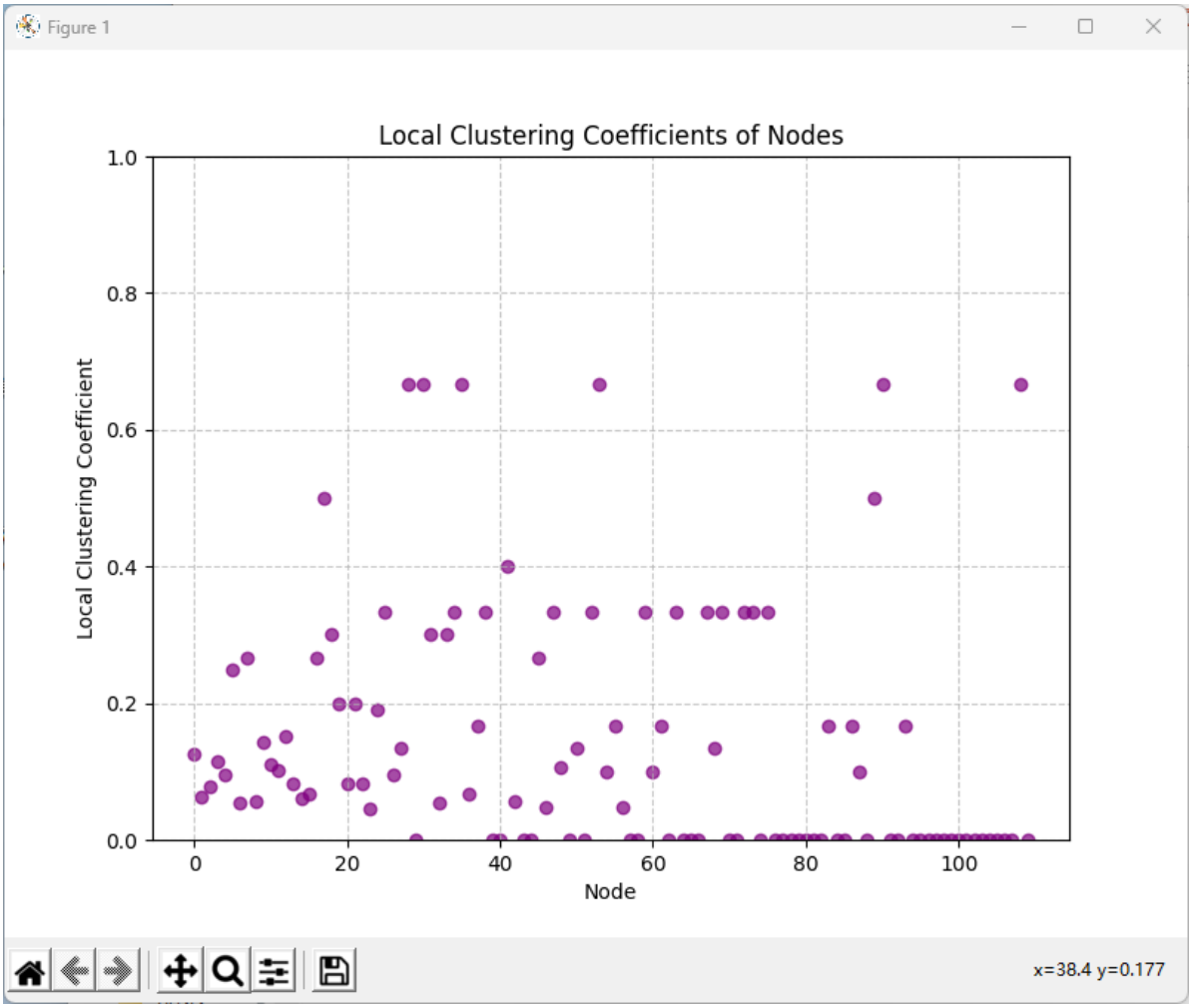
Figure 1
Local Clustering Coefficients of Nodes
Local Clustering Coefficient
Node
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
x=38.4 y=0.177

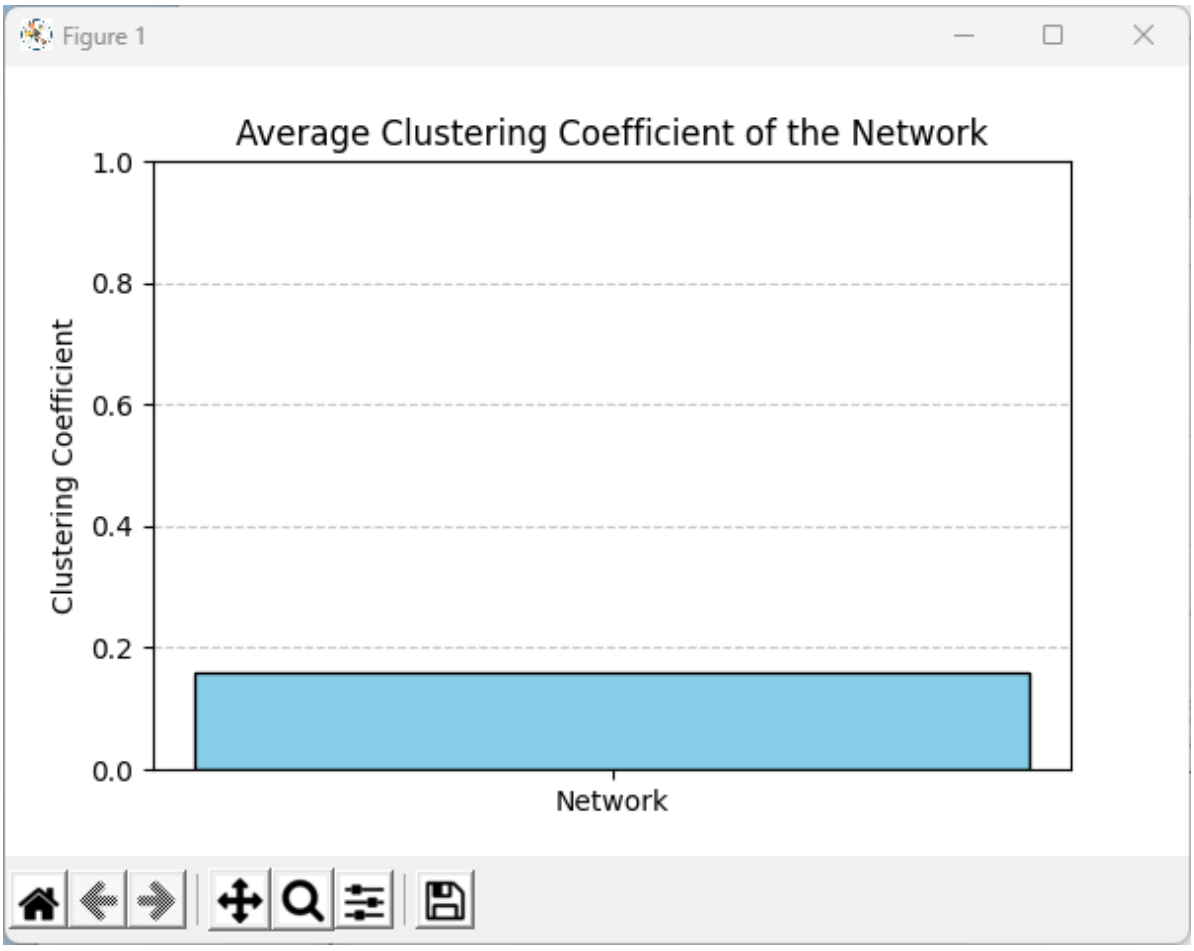
Figure 1
Average Clustering Coefficient of the Network
Clustering Coefficient
Network
1.0
0.8
0.6
0.4
0.2
0.0

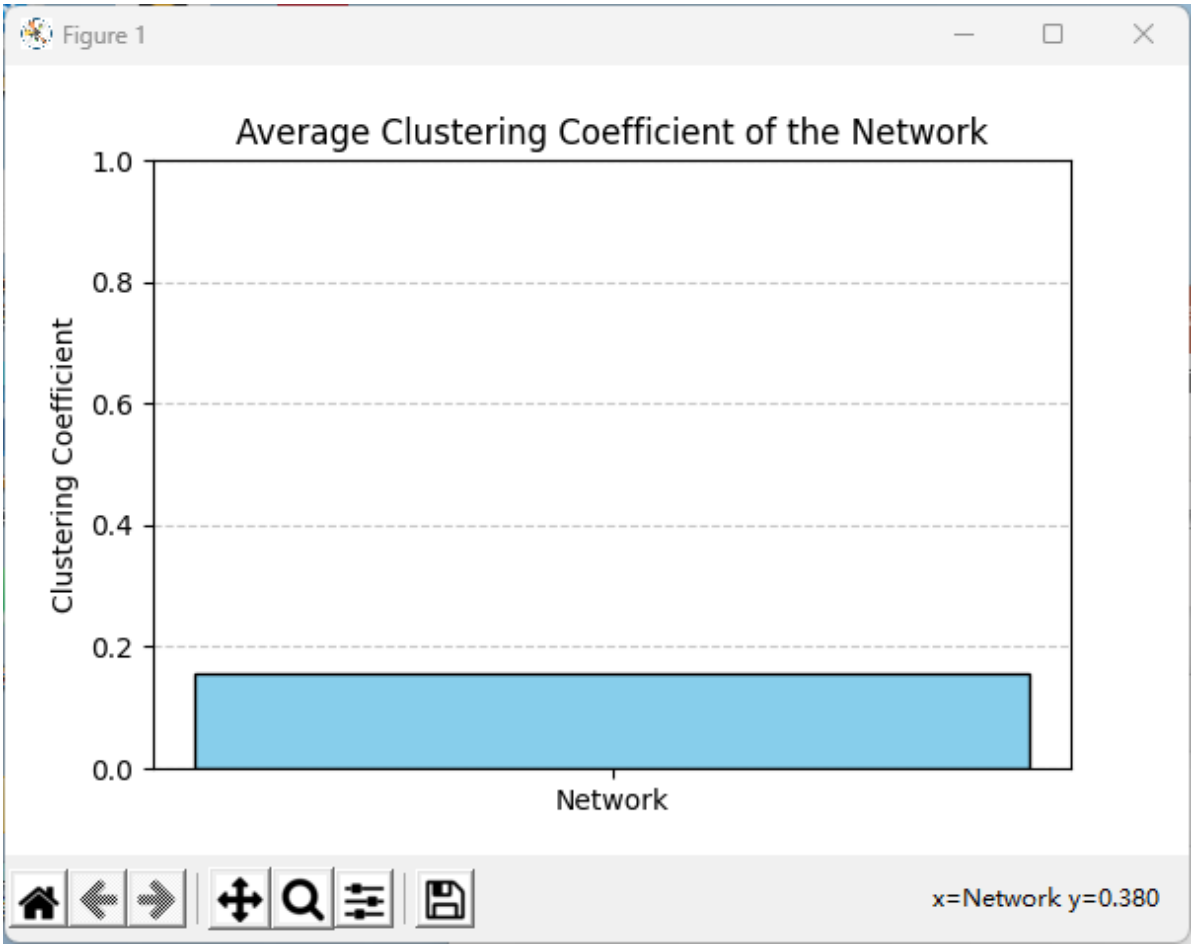
Figure 1
Average Clustering Coefficient of the Network
Clustering Coefficient
1.0
0.8
0.6
0.4
0.2
0.0
Network
x=Network y=0.380

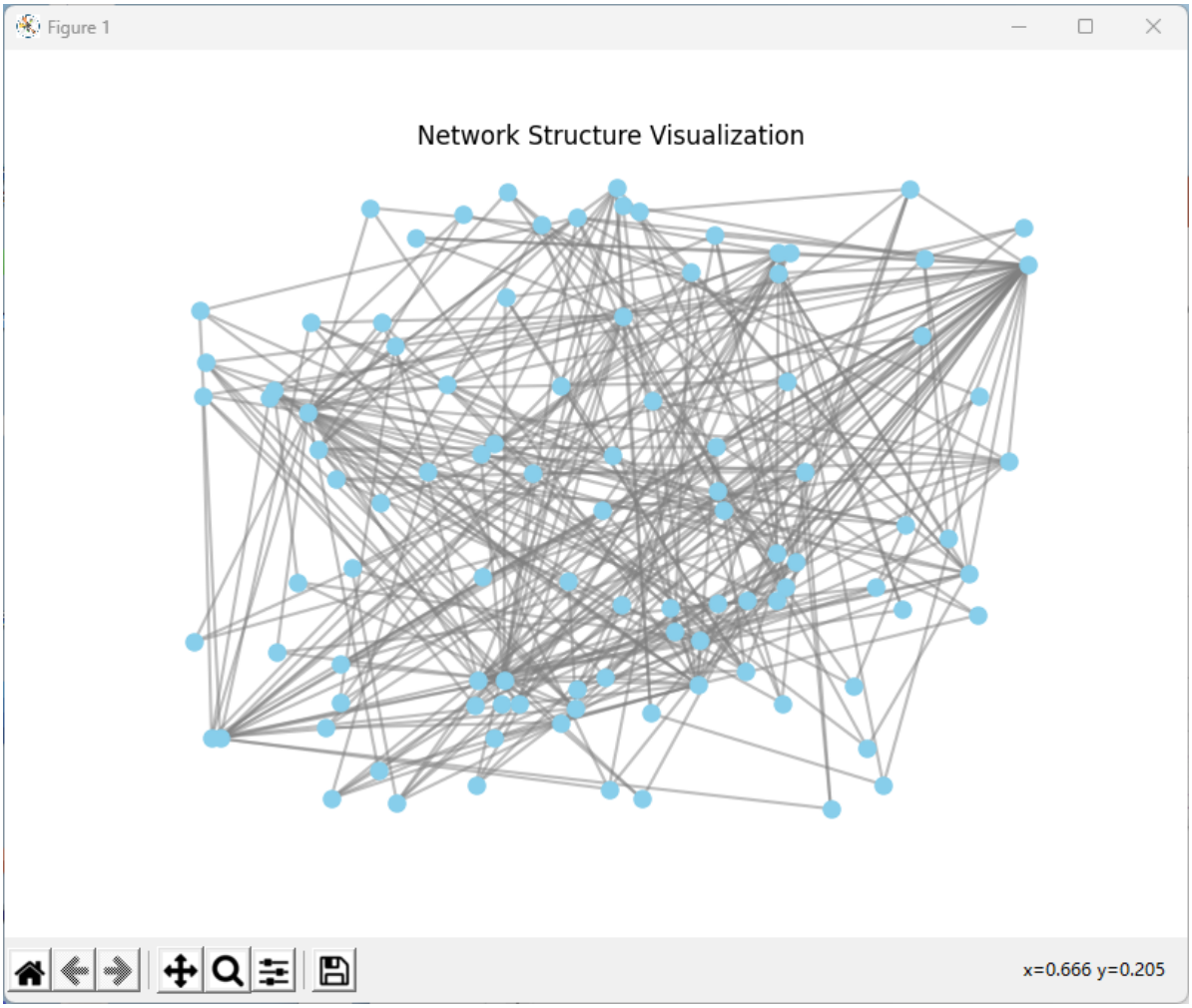
Figure 1
Network Structure Visualization
x=0.666 y=0.205

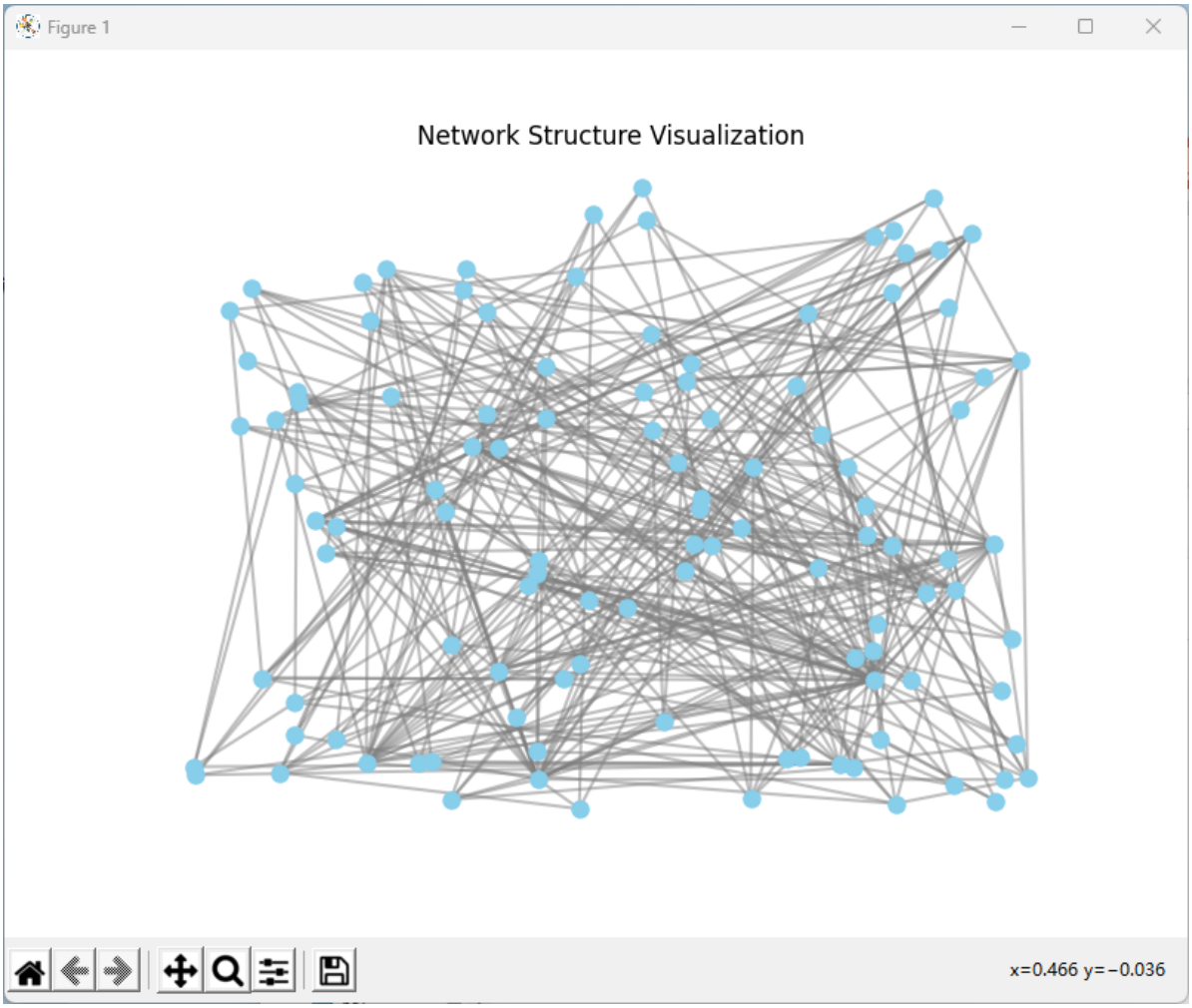
Figure 1
Network Structure Visualization
x=0.466 y=−0.036

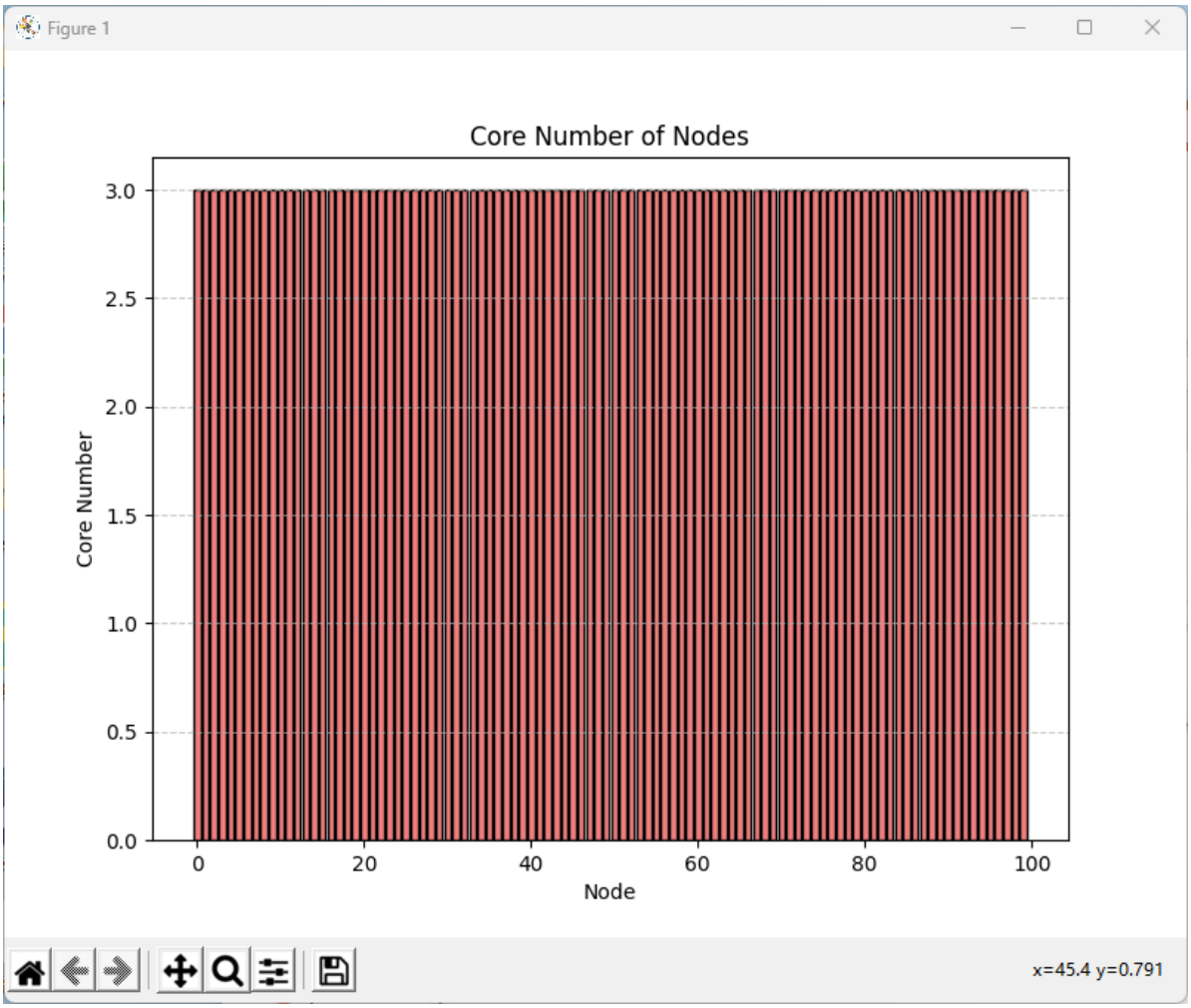
Figure 1
Core Number of Nodes
Core Number
Node
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
20
40
60
80
100
x=45.4 y=0.791

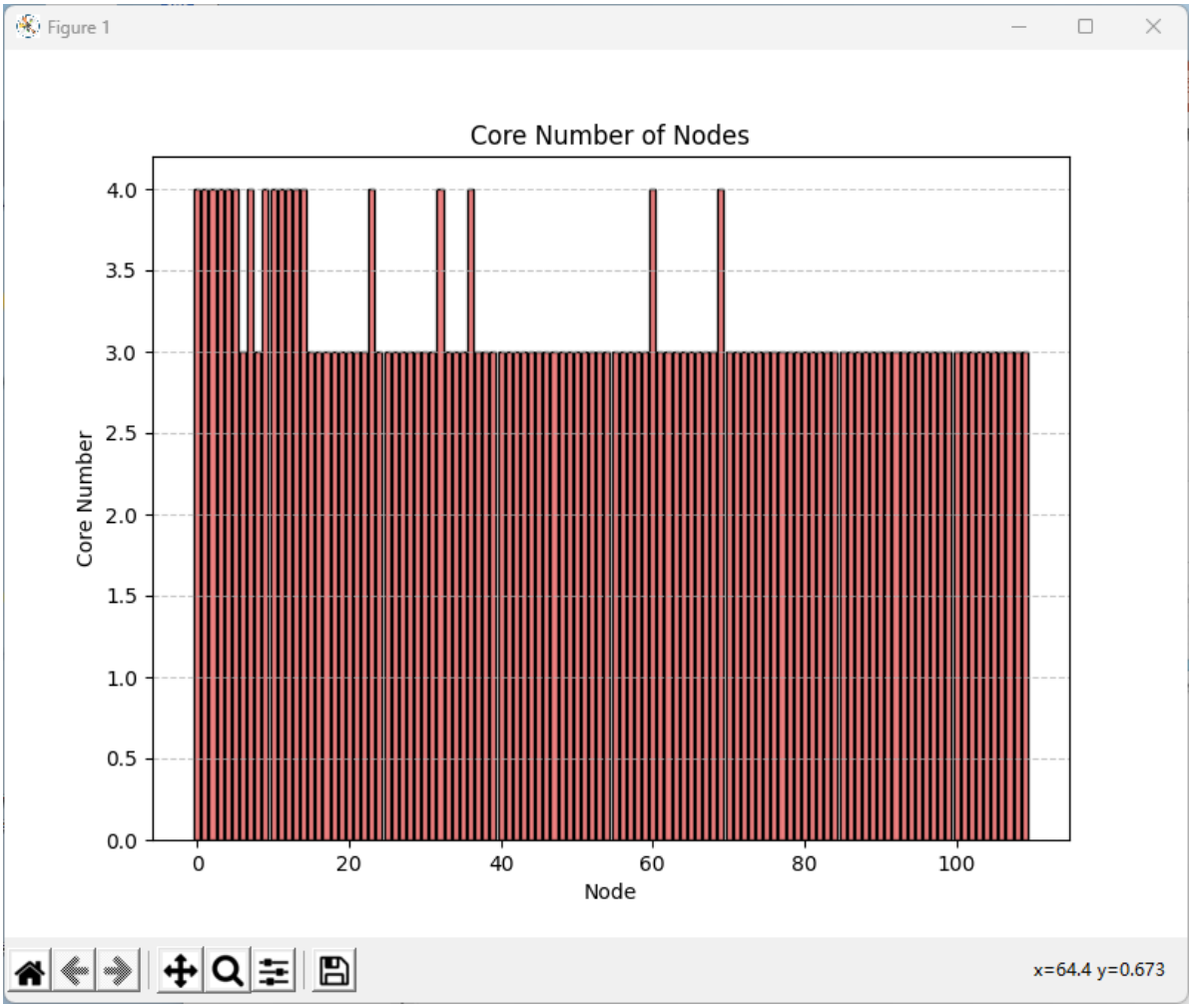
Figure 1
Core Number of Nodes
Core Number
Node
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
20
40
60
80
100
x=64.4 y=0.673

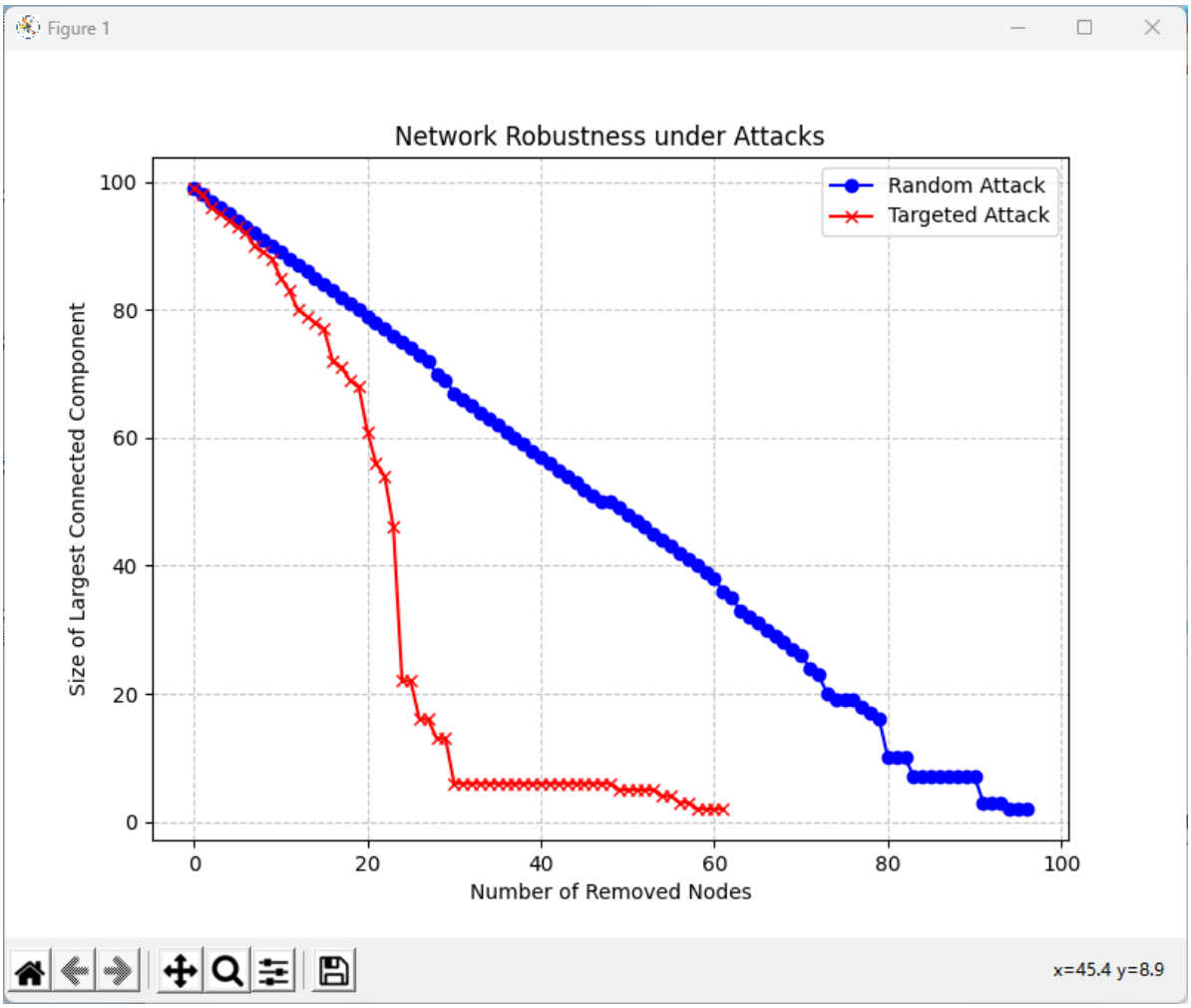
Figure 1
Network Robustness under Attacks
Random Attack
Targeted Attack
Size of Largest Connected Component
Number of Removed Nodes
0
20
40
60
80
100
x=45.4 y=8.9

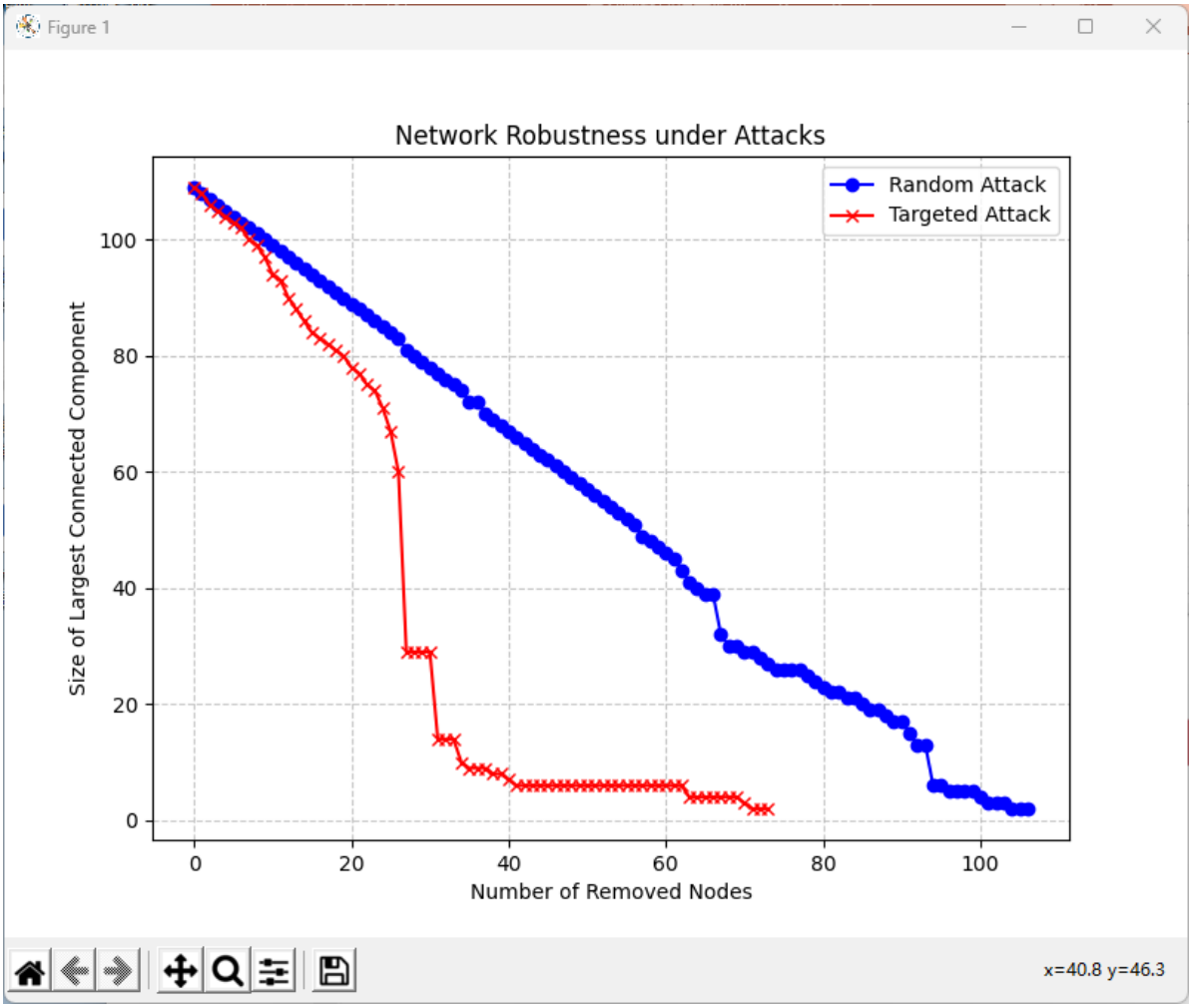
Figure 1
Network Robustness under Attacks
Random Attack
Targeted Attack
Size of Largest Connected Component
Number of Removed Nodes
0
20
40
60
80
100
x=40.8 y=46.3

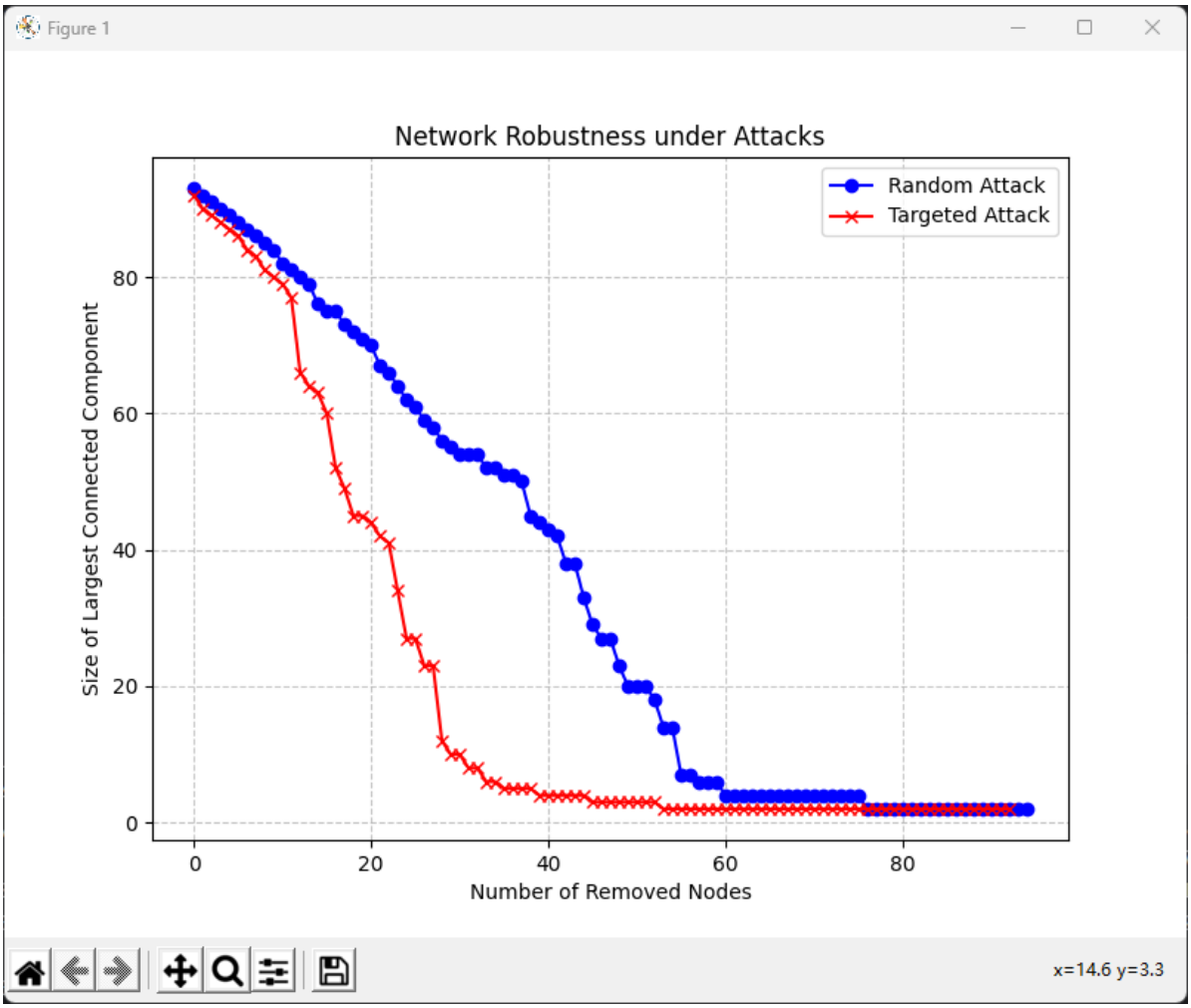
Figure 1
Network Robustness under Attacks
Random Attack
Targeted Attack
Size of Largest Connected Component
Number of Removed Nodes
0
20
40
60
80
x=14.6 y=3.3

```
Is Small World: True
Target Clustering: 0.08967054478661325, Random Clustering: 0.052521077783512754
Target Average Path Length: 2.653878231859883, Random Average Path Length: 2.838915991636866
```

```
Degree Distribution: {16: 1, 28: 1, 37: 1, 12: 3, 18: 1, 9: 7, 4: 23, 13: 1, 20:
1, 10: 2, 5: 9, 6: 6, 8: 4, 7: 3, 3: 37}
New Degree Distribution: {17: 2, 25: 1, 40: 1, 13: 2, 8: 3, 14: 1, 6: 8, 9: 5,
12: 2, 19: 1, 4: 28, 5: 11, 10: 1, 7: 8, 3: 35, 11: 1}
Total distance 25406
Average Shortest Path Length: 2.5662626262626262
Total distance 31574
New Average Shortest Path Length: 2.6333611342785654
Average Clustering Coefficient: 0.15997494855544395
Average Clustering Coefficient: 0.1420899588887205
Core Number of each node: {0: 3, 1: 3, 2: 3, 3: 3, 4: 3, 5: 3, 6: 3, 7: 3, 8: 3,
9: 3, 10: 3, 11: 3, 12: 3, 13: 3, 14: 3, 15: 3, 16: 3, 17: 3, 18: 3, 19: 3, 20:
3, 21: 3, 22: 3, 23: 3, 24: 3, 25: 3, 26: 3, 27: 3, 28: 3, 29: 3, 30: 3, 31: 3,
32: 3, 33: 3, 34: 3, 35: 3, 36: 3, 37: 3, 38: 3, 39: 3, 40: 3, 41: 3, 42: 3, 43:
3, 44: 3, 45: 3, 46: 3, 47: 3, 48: 3, 49: 3, 50: 3, 51: 3, 52: 3, 53: 3, 54: 3,
55: 3, 56: 3, 57: 3, 58: 3, 59: 3, 60: 3, 61: 3, 62: 3, 63: 3, 64: 3, 65: 3, 66:
3, 67: 3, 68: 3, 69: 3, 70: 3, 71: 3, 72: 3, 73: 3, 74: 3, 75: 3, 76: 3, 77: 3,
78: 3, 79: 3, 80: 3, 81: 3, 82: 3, 83: 3, 84: 3, 85: 3, 86: 3, 87: 3, 88: 3, 89:
3, 90: 3, 91: 3, 92: 3, 93: 3, 94: 3, 95: 3, 96: 3, 97: 3, 98: 3, 99: 3}
Core Number of each node: {0: 4, 1: 4, 2: 4, 3: 4, 4: 4, 5: 4, 6: 3, 7: 4, 8: 3,
9: 4, 10: 4, 11: 4, 12: 4, 13: 4, 14: 4, 15: 3, 16: 3, 17: 3, 18: 3, 19: 3, 20:
3, 21: 3, 22: 3, 23: 4, 24: 3, 25: 3, 26: 3, 27: 3, 28: 3, 29: 3, 30: 3, 31: 3,
32: 4, 33: 3, 34: 3, 35: 3, 36: 4, 37: 3, 38: 3, 39: 3, 40: 3, 41: 3, 42: 3, 43:
3, 44: 3, 45: 3, 46: 3, 47: 3, 48: 3, 49: 3, 50: 3, 51: 3, 52: 3, 53: 3, 54: 3,
55: 3, 56: 3, 57: 3, 58: 3, 59: 3, 60: 4, 61: 3, 62: 3, 63: 3, 64: 3, 65: 3, 66:
3, 67: 3, 68: 3, 69: 4, 70: 3, 71: 3, 72: 3, 73: 3, 74: 3, 75: 3, 76: 3, 77: 3,
78: 3, 79: 3, 80: 3, 81: 3, 82: 3, 83: 3, 84: 3, 85: 3, 86: 3, 87: 3, 88: 3, 89:
3, 90: 3, 91: 3, 92: 3, 93: 3, 94: 3, 95: 3, 96: 3, 97: 3, 98: 3, 99: 3, 100: 3,
101: 3, 102: 3, 103: 3, 104: 3, 105: 3, 106: 3, 107: 3, 108: 3, 109: 3}
Random Attack Sizes: [99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85,
84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 70, 69, 67, 66, 65, 64, 63,
62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 50, 49, 48, 47, 46, 45, 44,
43, 42, 41, 40, 39, 38, 36, 35, 33, 32, 31, 30, 29, 28, 27, 26, 24, 23, 20, 19,
19, 19, 18, 17, 16, 10, 10, 10, 7, 7, 7, 7, 7, 7, 7, 7, 3, 3, 3, 2, 2, 2]
New Random Attack Sizes: [109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99,
98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 81, 80, 79, 78,
77, 76, 75, 74, 72, 72, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57,
56, 55, 54, 53, 52, 51, 49, 48, 47, 46, 45, 43, 41, 40, 39, 39, 32, 30, 30, 29,
29, 28, 27, 26, 26, 26, 26, 25, 24, 23, 22, 22, 21, 21, 20, 19, 19, 18, 17, 17,
15, 13, 13, 6, 6, 5, 5, 5, 5, 4, 3, 3, 3, 2, 2, 2]
Targeted Attack Sizes: [99, 98, 96, 95, 94, 93, 92, 90, 89, 88, 85, 83, 80, 79,
78, 77, 72, 71, 69, 68, 61, 56, 54, 46, 22, 22, 16, 16, 13, 13, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 5, 5, 5, 4, 4, 3, 3, 2, 2, 2, 2]
Targeted Attack Sizes: [109, 108, 106, 105, 104, 103, 102, 100, 99, 97, 94, 93,
90, 88, 86, 84, 83, 82, 81, 80, 78, 77, 75, 74, 71, 67, 60, 29, 29, 29, 29, 14,
14, 14, 10, 9, 9, 9, 8, 8, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4, 3, 2, 2, 2]
Is Small World: True
Target Clustering: 0.1420899588887205, Random Clustering: 0.05351145786203375
Target Average Path Length: 2.6333611342785654, Random Average Path Length:
2.830809862919954
```