Social Network Analysis for Computer Scientists – Assignment 1

Benard Adala Wanyande (s4581733)

Leiden University, Leiden, The Netherlands

Global Assumptions for 1.1 - 1.5

- The network $G = (V, E)$ is finite: $|V| = n < \infty$, $|E| = m < \infty$.
- Self-loops (v, v), if present, are counted as one edge and considered trivially reciprocated.
- Multiple edges between the same ordered pair of nodes are ignored (i.e., G is a simple directed graph).

Reference

All definitions and notation follow the assignment instructions [3] by F. Takes.

1.1

Let $G = (V, E)$ be a directed network. The neighborhoods of a node $v \in V$ are:

$$N(v) = \{w \in V : (v, w) \in E\}, \qquad N'(v) = \{u \in V : (u, v) \in E\}.$$

The outdegree and indegree are:

$$\deg^+(v) = |N(v)|, \qquad \deg^-(v) = |N'(v)|.$$

1.2

The combined neighborhood of v is:

$$N^*(v) = N(v) \cup N'(v),$$

and the combined degree is:

$$\deg^*(v) = |N^*(v)| = |N(v) \cup N'(v)|.$$

1.3

Assuming 'incident edges' refers to the set of unique neighbors, let $R(v) = N(v) \cap N'(v)$ be the set of reciprocated neighbors of v. The condition that at least half of a node's incident edges are reciprocated is:

$$\forall v \in V : \quad |R(v)| \geq \frac{1}{2}\Big(|N(v)| + |N'(v)| - |R(v)|\Big).$$

1.4

For a set of nodes $W \subseteq V$, the k-neighborhood is recursively defined:

$$N_0(W) = W, \qquad N_k(W) = N(N_{k-1}(W)) \cup N_{k-1}(W), \ k > 0$$

The reversed k-neighborhood applies the same recursion to the reverse neighborhood N':

$$N'_0(W) = W, \qquad N'_k(W) = N'(N'_{k-1}(W)) \cup N'_{k-1}(W), \ k > 0$$

For a single node v, define:

$$N'_k(v) := N'_k(\{v\}).$$

Interpretation: $N'_k(v)$ measures all nodes that can reach v within k steps via incoming edges, capturing potential influence or information flow towards v.

1.5

Two nodes $u, v \in V$ are in the same *strongly connected component (SCC)* if and only if u is reachable from v and v is reachable from u. Using the previously introduced concepts of k-neighborhood and reversed k-neighborhood, this condition can be expressed formally as:

$$v \in \bigcup_{k=0}^{n-1} N_k(u) \quad \wedge \quad u \in \bigcup_{k=0}^{n-1} N'_k(v),$$

where $N_k(u)$ denotes the k-neighborhood of u, and $N'_k(v)$ the reversed k-neighborhood of v. Thus, u and v are in the same SCC if they belong to each other's reachability sets.

Pseudocode In practice, the dominance relation can be checked by computing neighborhoods iteratively. This can be done efficiently using a standard graph traversal such as Breadth-First Search (BFS), which explores nodes layer by layer corresponding to increasing k-neighborhoods [1]. The pseudocode below illustrates how to construct the union of all forward and reversed k-neighborhoods to verify dominance:

```
// Helper: compute the union of all k-neighborhoods of a start node using BFS
FUNCTION ComputeNeighborhoodUnion(G, start_node):
    q ← new Queue()
    q.enqueue(start_node)

    union_set ← {start_node}

    WHILE q is not empty:
        current ← q.dequeue()
        FOR EACH neighbor  N(current):
            IF neighbor  union_set:
                union_set.add(neighbor)
                q.enqueue(neighbor)

    RETURN union_set


// Main procedure: check if u and v are in the same SCC
ALGORITHM AreInSameSCC(G, u, v):
    // Step 1: Forward reachability from u
    forward_union ← ComputeNeighborhoodUnion(G, u)
    IF v  forward_union:
        RETURN False

    // Step 2: Backward reachability to u (using reversed graph)
    G_rev ← ReverseGraph(G)
    reverse_union ← ComputeNeighborhoodUnion(G_rev, u)

    // Step 3: SCC membership check
    RETURN (v  reverse_union)
```

Global Assumptions for 1.6 - 1.10

- The network is undirected: for every $(u, v) \in E$, we also have $(v, u) \in E$.
- There is a single connected component.
- Self-loops are ignored.
- Since the network is undirected, forward and reversed neighborhoods are identical: $N_k(v) = N'_k(v)$.
- A triangle involving v is a set of three distinct nodes $\{v, u, w\}$ where $(v, u), (v, w), (u, w) \in E$ [4].

1.6

Let $G = (V, E)$ be an undirected network with symmetric edges, modeling a single connected component. We are interested in counting the number of unique triangles involving a given node $v \in V$ using the notion of the (reversed) k-neighborhood.

Formal Definition A pair of distinct nodes $\{u, w\}$ forms a triangle with v if and only if

$$u \in N_1(v), \quad w \in N_1(v), \quad w \in N_1(u).$$

The total number of unique triangles involving v, denoted $T(v)$, can be expressed as

$$T(v) = \frac{1}{2} \sum_{u \in N_1(v)} |N_1(v) \cap N_1(u)|,$$

where the factor $1/2$ corrects for double-counting each triangle.

Pseudocode

```
// Computes the number of unique triangles involving node v
ALGORITHM CountTrianglesForNode(Graph G, Node v):
    neighbors_of_v = G.get_neighbors(v)  // 1-neighborhood N_1(v)
    triangle_count = 0
    neighbor_list = convert_to_list(neighbors_of_v)
    n = length(neighbor_list)

    FOR i from 0 to n-2:
        u = neighbor_list[i]
        FOR j from i+1 to n-1:
            w = neighbor_list[j]
            IF G.has_edge(u, w):
                triangle_count = triangle_count + 1

    RETURN triangle_count

// Helper functions:
// G.get_neighbors(node): returns the set N_1(node)
// G.has_edge(u, w): returns True if an edge exists between u and w
// convert_to_list(set): converts a set to an ordered list
```

Explanation The algorithm begins by retrieving the 1-neighborhood $N_1(v)$ of the node v, which consists of all nodes directly connected to v [3]. It then iterates over all unique pairs of neighbors (u, w) to avoid double-counting any potential triangle, following standard combinatorial reasoning in undirected graphs [4]. For each pair, the algorithm checks whether an edge exists between u and w. If such an edge is present, the triplet $\{v, u, w\}$ forms a triangle in the network. The algorithm counts all such triangles and returns the total number of unique triangles involving v.

Note that in directed networks, the function `ReverseGraph(G)` returns a graph in which the direction of every edge is reversed; however, for undirected graphs, forward and reversed neighborhoods coincide [3].

1.7

Let $G = (V, E)$ be an undirected network with symmetric edges, modeling a single connected component. The network center is the set of nodes with minimal eccentricity, where the eccentricity of a node $v \in V$ is the smallest k such that its k-neighborhood contains all nodes in the network.

Formal Definition The eccentricity of a node v is

$$\varepsilon(v) = \min\{k \geq 0 \mid N_k(v) = V\},$$

where $N_k(v)$ denotes the k-neighborhood of v.

The network radius is

$$r(G) = \min_{v \in V} \varepsilon(v),$$

and the network center is

$$C(G) = \{v \in V \mid \varepsilon(v) = r(G)\}.$$

Pseudocode

```
// Helper: compute the eccentricity of a node using BFS and k-neighborhoods
ALGORITHM ComputeEccentricity(Graph G, Node v):
    q = new Queue()
    q.enqueue((v, 0))  // (node, distance)
    visited = Set({v})
    max_k = 0

    WHILE q is not empty:
        current_node, k = q.dequeue()
        max_k = max(max_k, k)

        FOR EACH neighbor IN G.get_neighbors(current_node):
            IF neighbor NOT IN visited:
                visited.add(neighbor)
                q.enqueue((neighbor, k + 1))

    RETURN max_k  // eccentricity of v

// Main algorithm: compute the network center
ALGORITHM FindNetworkCenter(Graph G):
    eccentricities = Map()
    min_eccentricity = Infinity

    FOR EACH node v IN G.V:
        ecc_v = ComputeEccentricity(G, v)
        eccentricities[v] = ecc_v
        min_eccentricity = min(min_eccentricity, ecc_v)

    center_nodes = Set()
    FOR EACH node v IN G.V:
        IF eccentricities[v] == min_eccentricity:
            center_nodes.add(v)

    RETURN center_nodes
```

Explanation The algorithm computes the eccentricity of each node by performing a BFS starting from that node, which effectively constructs the k-neighborhood layer by layer. The maximum k reached in BFS is the eccentricity $\varepsilon(v)$. After computing all eccentricities, the minimum eccentricity is identified as the network radius $r(G)$. Finally, all nodes whose eccentricity equals the radius are collected to form the network center $C(G)$.

1.8

Let $G = (V, E)$ be an undirected network with symmetric edges. A node $u \in V$ is said to *dominate* a node $v \in V$ if every neighbor of v is also a neighbor of u.

Formal Definition Using the neighborhood function $N(\cdot)$, the dominance relation can be defined as:

$$u \text{ dominates } v \quad \iff \quad N(v) \subseteq N(u).$$

Equivalently, in words: u dominates v if and only if for all $w \in V$,

$$w \in N(v) \implies w \in N(u).$$

Explanation This definition captures the idea that u "covers" all the immediate connections of v. In terms of the neighborhood function, dominance is simply the set inclusion of v's neighbors within u's neighbors. Since the network is undirected, the forward and reversed neighborhoods are identical.

1.9

Let $G = (V, E)$ be an undirected graph with $n = |V|$ nodes and $m = |E|$ edges. We wish to find the set of all non-dominated nodes in the network.

Formal Definition

A node $v \in V$ is said to **dominate** a node $u \in V$ if every neighbor of u is also a neighbor of v [2]. Using the neighborhood function $N(\cdot)$, which denotes the set of adjacent nodes (the 1-neighborhood), this is formally expressed as:

$$v \text{ dominates } u \iff N(u) \subseteq N(v).$$

A node u is **non-dominated** if there exists no other node $v \in V$ $(v \neq u)$ that dominates it.

Pseudocode

The algorithm works by a process of elimination. We begin by assuming all nodes are non-dominated. We then iterate through all ordered pairs of distinct nodes (u, v) and check if v dominates u. If it does, we remove u from the set of non-dominated candidates. The nodes that remain at the end are the non-dominated set.

```
// Finds all non-dominated nodes in G
ALGORITHM FindNonDominatedNodes(Graph G):
    non_dominated_set = Set of all nodes in G

    FOR each node u in G.V:
        FOR each node v in G.V:
            IF u == v:
                CONTINUE
            ENDIF

            // Check if N(u) is a subset of N(v)
            IF IsSubset(G, u, v):
                non_dominated_set.remove(u)
                BREAK // u is dominated, no need to check other nodes
            ENDIF
        ENDFOR
    ENDFOR

    RETURN non_dominated_set

// Helper function: checks if N(u)  N(v)
FUNCTION IsSubset(Graph G, Node u, Node v):
    neighbors_u = G.get_neighbors(u)
    neighbors_v = G.get_neighbors(v)

    IF size(neighbors_u) > size(neighbors_v):
        RETURN False
    ENDIF

    FOR each w in neighbors_u:
        IF w NOT IN neighbors_v:
            RETURN False
        ENDIF
    ENDFOR

    RETURN True
```

Complexity Analysis

The algorithm's time complexity is dictated by its nested loops, which check the dominance condition for every ordered pair of distinct nodes (u, v). The core operation, checking if $N(u) \subseteq N(v)$, costs $O(d_u)$ time, where d_u is the degree of node u. By summing this cost over all pairs, the total time complexity is derived as $O\left(n \cdot \sum_{u \in V} d_u\right)$, which simplifies to $O(n \cdot m)$ since $\sum d_u = 2m$.

The space complexity is determined by the storage for the graph itself. Using an adjacency list, this requires $O(n + m)$ space. The set of non-dominated nodes adds at most $O(n)$ space.

The performance implications are summarized in Table 1.

1.10

Let $G = (V, E)$ be an undirected graph. We have access only to an arbitrary node $v \in V$ and the neighborhood function $N(\cdot)$. We aim to find all non-dominated nodes in the connected component of v.

Table 1. Summary of Algorithm Complexity

Graph Type	Time Complexity	Space Complexity
General	$O(n \cdot m)$	$O(n+m)$
Sparse ($m = O(n)$)	$O(n^2)$	$O(n+m)$
Dense ($m = O(n^2)$)	$O(n^3)$	$O(n+m)$

Pseudocode

```
// Main algorithm to find non-dominated nodes from a single start node
ALGORITHM FindNonDominatedNodes_FromSingleNode(Graph G, Node start_node):
    // Phase 1: Discover all nodes in the connected component
    q = new Queue()
    q.enqueue(start_node)
    discovered_nodes = Set(start_node)

    WHILE q is not empty:
        current_node = q.dequeue()
        FOR EACH neighbor OF current_node:
            IF neighbor NOT IN discovered_nodes:
                discovered_nodes.add(neighbor)
                q.enqueue(neighbor)

    // Phase 2: Dominance check
    non_dominated_set = Set of all discovered_nodes
    FOR EACH u IN discovered_nodes:
        FOR EACH v IN discovered_nodes:
            IF u == v: CONTINUE
            IF IsSubset(G, u, v):
                non_dominated_set.remove(u)
                BREAK
    RETURN non_dominated_set

    // IsSubset() function defined in solution 1.9
```

Explanation

The algorithm consists of two phases. First, it performs a breadth-first search (BFS) starting from the given node to discover all nodes in its connected component. Second, it iteratively checks all pairs of discovered nodes (u, v) to identify dominated nodes by testing if $N(u) \subseteq N(v)$. Any node found to be dominated is removed from the candidate set. The remaining nodes form the non-dominated set for that component.

Limitation: Only nodes reachable from the start node are considered. For disconnected graphs, non-dominated nodes in other components are not discovered.

Question 2 Answer Table

Table 2. Summary of key network statistics for `medium.tsv` and `large.tsv`.

Measure	medium.tsv	large.tsv
Number of directed links (Q2.1)	16,206	149,668
Number of users / nodes (Q2.2)	5,883	41,761
# Weakly connected components (Q2.4)	198	647
# Strongly connected components (Q2.4)	1,834	19,259
Largest WCC: nodes (Q2.4)	5,258	39,654
Largest WCC: edges (Q2.4)	15,508	147,776
Largest SCC: nodes (Q2.4)	3,636	21,213
Largest SCC: edges (Q2.4)	13,002	120,529
Average clustering coefficient (Q2.5)	0.1991	0.2997
Average distance in largest WCC (Q2.7)	5.9887	5.8115

2.1

For both datasets, the number of directed links corresponds to the total number of edges in the network. We loaded the edge lists using pandas and constructed directed graphs in NetworkX. The results are shown in Table 2.
Method: Each line in the TSV file represents a directed edge from `source` to `target`. Counting all lines gives the number of links. See Appendix A for the Python code.

2.2

We define a user as a node that appears at least once as a source or a target in the dataset. Using NetworkX, we automatically collect all unique nodes from the edge list, which yields the total number of users per network. The results are also listed in Table 2.
Method: After creating a directed graph from the edge list, the `number_of_nodes()` function returns the total number of distinct users. Refer to Appendix A for the code.

2.3

We computed the indegree and outdegree distributions for both networks using NetworkX. Each node's indegree and outdegree was collected from the directed graphs constructed from the TSV edge lists. The distributions were visualized with both histograms (with log-scaled y-axis) and log-log scatter plots to highlight the heavy-tailed nature of the networks.
Method: For each graph, we extracted in-degrees and out-degrees via `G.in_degree()` and `G.out_degree()`, respectively, and plotted both histogram (log y-axis) and log-log scatter plots. See Appendix B for the full Python code.
These plots reveal the heavy-tailed nature of the social networks, where a few nodes have very high degrees while most nodes have low degrees.

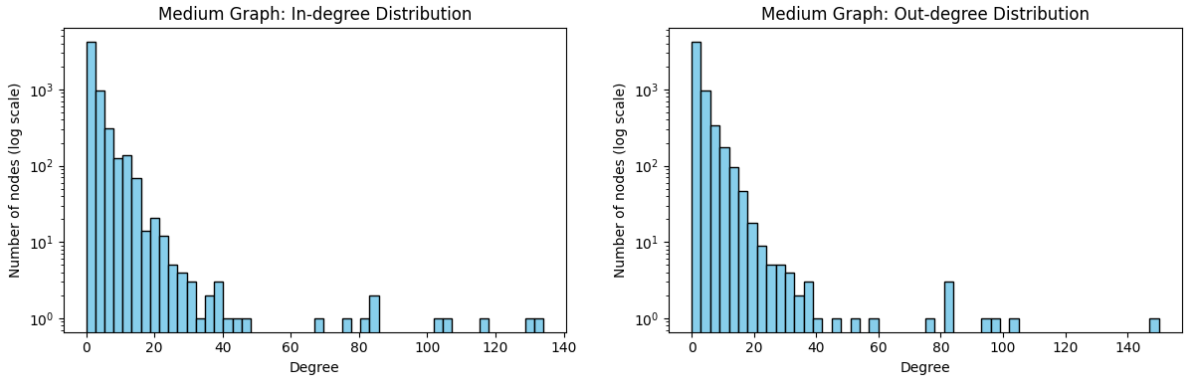

Fig. 1. Medium network: in-degree (left) and out-degree (right) distributions (histogram, log-scale y-axis).

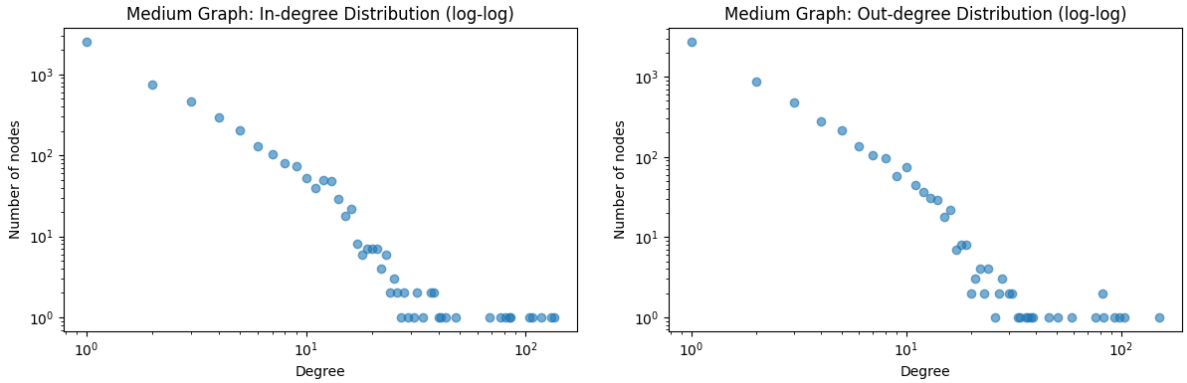

Fig. 2. Medium network: in-degree (left) and out-degree (right) distributions (log-log scatter plots).

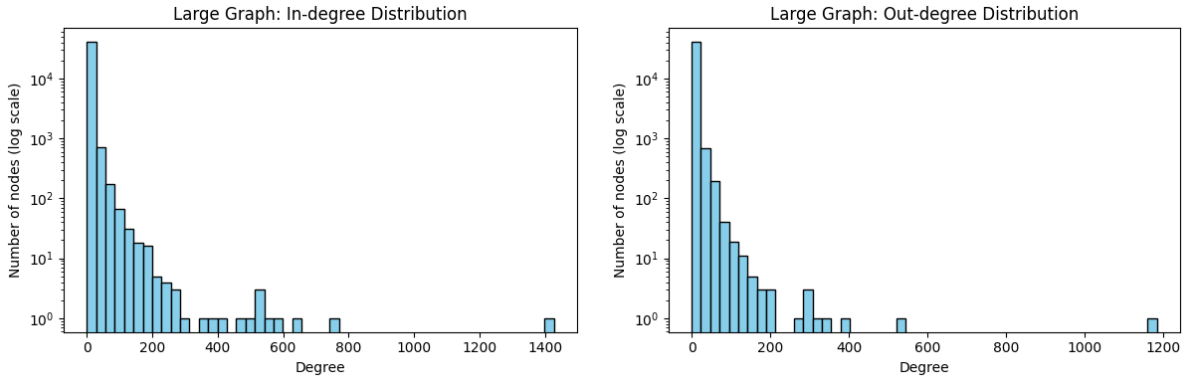

Fig. 3. Large network: in-degree (left) and out-degree (right) distributions (histogram, log-scale y-axis).

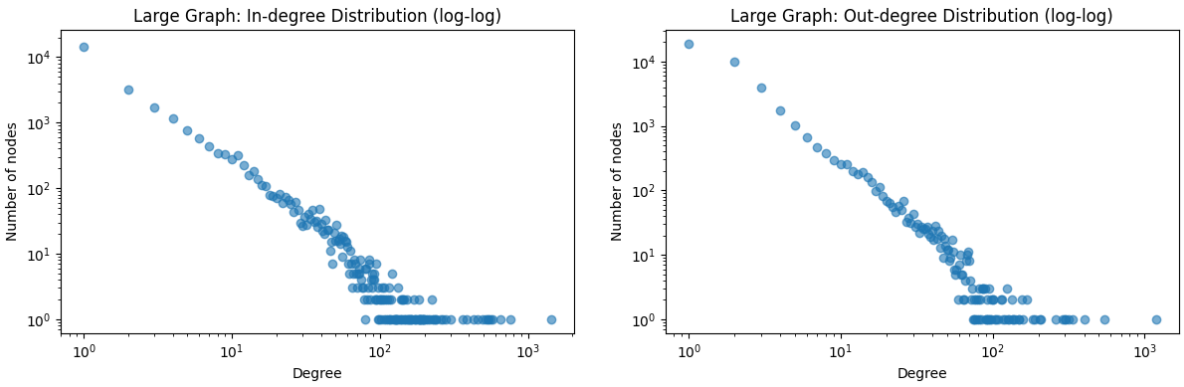

Fig. 4. Large network: in-degree (left) and out-degree (right) distributions (log-log scatter plots).

2.4

We computed the number of weakly connected components (WCCs) and strongly connected components (SCCs) for both datasets using `NetworkX`'s `weakly_connected_components` and `strongly_connected_components` functions. For each type of component, we also measured the size of the largest component in terms of nodes and edges. The results are reported in Table 2, and the implementation is listed in Appendix B.

2.5

Directionality is accounted for by ignoring the orientation of edges and treating the graph as undirected. In practice, this means that an undirected edge is considered present whenever there is at least one directed edge between two nodes. Triangles are then counted regardless of edge orientation, providing a consistent measure of clustering.

The implementation used for this calculation is provided in Appendix B.

2.6

The distance distribution was computed on the largest weakly connected component (LWCC) of each network. For the large network, a sample of 1000 nodes was used to approximate the distribution. Directionality was ignored by converting the directed graph to undirected before computing shortest paths. The y-axis in both plots uses a logarithmic scale to better visualize rare long-distance pairs.

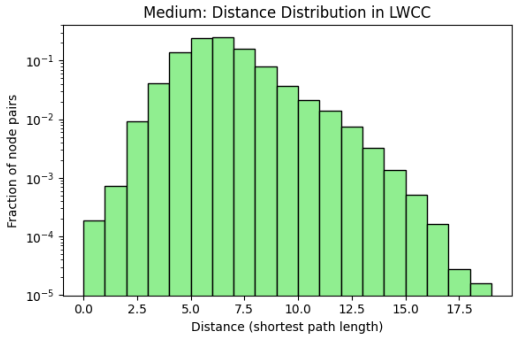

Fig. 5. Distance distribution in the largest weakly connected component of the medium graph (log-scaled y-axis).

For the source code used to generate these distributions, see Appendix B.

2.7

The average distance between any two node pairs in the largest weakly connected component (LWCC) of each network was computed using shortest path lengths on the undirected version of the LWCC.

For the `medium.tsv` network, the computation was exact, while for `large.tsv`, a sample of 1000 nodes was used to approximate the average distance due to the network's size. Directionality was ignored by treating the LWCC as undirected, consistent with common practice for distance measures in directed networks.

The implementation details and plotting of the distance distributions are provided in Appendix B.

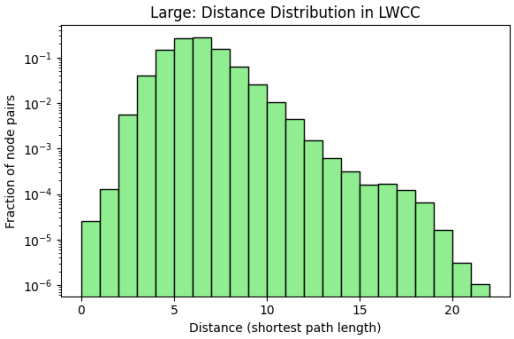

Fig. 6. Approximate distance distribution in the largest weakly connected component of the large graph (sample of 1000 nodes, log-scaled y-axis).

2.8

The social network contained in `medium.tsv` was visualized using Gephi. Several node-level centrality measures and a force-based layout were employed to produce an interpretable visualization.

Node size was determined by betweenness centrality. In the context of a social network, nodes with high betweenness centrality act as intermediaries that bridge different parts of the graph. Enlarging these nodes emphasizes their role in maintaining communication across otherwise distant communities, making them visually stand out as structural brokers.

Node color indicates community membership, as detected by the modularity optimization algorithm. Each distinct color represents a separate community, highlighting the modular structure of the network.

To further reveal modular organization, community detection was performed using modularity optimization. Nodes were grouped into color-coded clusters that represent densely connected subcommunities. The resulting modularity score of 0.843 indicates a clear community structure.

For layout, the Fruchterman–Reingold force-directed algorithm was applied. This algorithm positions nodes by balancing attractive forces along edges with repulsive forces between nodes, producing an intuitive spatialization of the network. Parameters such as gravity were increased to prevent nodes from drifting too far apart, and scaling was adjusted to reduce overlap in dense regions.

Labels were enabled only for the most central nodes (by betweenness centrality) to avoid clutter while still allowing identification of the key actors in the network. This ensures that important nodes are annotated directly on the visualization without overwhelming the readability of the graph.

The final visualization, shown in Figure 7, is included as a full-page vector image. In order to achieve a vector image a the pdf export of the network has been embedding into this report on the next page. It combines structural, positional, and community-level information, yielding a clear picture of both local and global organization in the network.

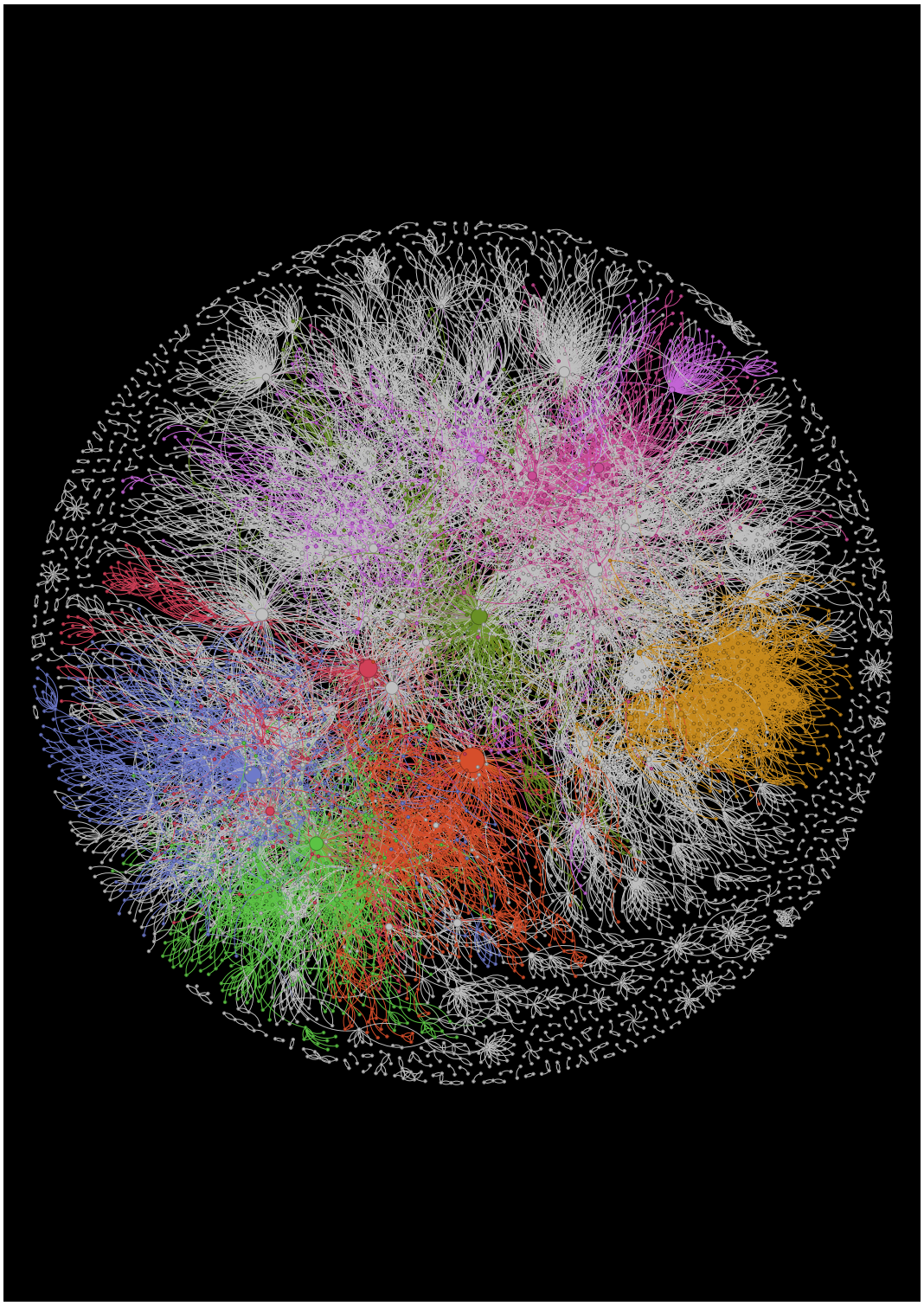

Fig. 7. Visualization of the `medium.tsv` social network. Node size reflects betweenness centrality, node color reflects eigenvector centrality and community membership, and layout is based on the Fruchterman–Reingold algorithm.

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.
2. Reinhard Diestel. *Graph Theory*. Springer, Berlin, Heidelberg, 5th edition, 2018.
3. Frank W. Takes. Social network analysis for computer scientists – assignment 1. https://liacs.leidenuniv.nl/ takesfw/SNACS/snacs2025-assignment1.pdf, 2025.
4. Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 2001.

A Python Code for 2.1 and 2.2

```
import networkx as nx
import pandas as pd

# Load data
medium_edges = pd.read_csv('snacs2025-student4581733-medium.tsv', sep='\t', header=None, names=['source', 'target'])
large_edges = pd.read_csv('snacs2025-student4581733-large.tsv', sep='\t', header=None, names=['source', 'target'])

# Create directed graphs
G_medium = nx.from_pandas_edgelist(medium_edges, source='source', target='target', create_using=nx.DiGraph())
G_large = nx.from_pandas_edgelist(large_edges, source='source', target='target', create_using=nx.DiGraph())

# Test outputs
print("Medium graph: nodes =", G_medium.number_of_nodes(), ", edges =", G_medium.number_of_edges())
print("Large graph: nodes =", G_large.number_of_nodes(), ", edges =", G_large.number_of_edges())
```

B Python Code for 2.3

```
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt
import collections
import os

# Load data
medium_edges = pd.read_csv('snacs2025-student4581733-medium.tsv', sep='\t', header=None, names=['source','target'])
large_edges = pd.read_csv('snacs2025-student4581733-large.tsv', sep='\t', header=None, names=['source','target'])

# Create directed graphs
G_medium = nx.from_pandas_edgelist(medium_edges, 'source', 'target', create_using=nx.DiGraph())
G_large = nx.from_pandas_edgelist(large_edges, 'source', 'target', create_using=nx.DiGraph())

# Function to extract degrees
def get_degrees(G):
    return [d for n,d in G.in_degree()], [d for n,d in G.out_degree()]

in_med, out_med = get_degrees(G_medium)
in_large, out_large = get_degrees(G_large)

# Function to plot histogram
def plot_hist(degrees, title, filename):
    plt.figure(figsize=(6,4))
    plt.hist(degrees, bins=50, color='skyblue', edgecolor='black', log=True)
    plt.xlabel("Degree")
    plt.ylabel("Number of nodes (log scale)")
    plt.title(title)
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()

# Function to plot log-log scatter
def plot_loglog(degrees, title, filename):
    degree_count = collections.Counter(degrees)
    deg, cnt = zip(*degree_count.items())
    plt.figure(figsize=(6,4))
    plt.scatter(deg, cnt, alpha=0.6)
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel("Degree")
    plt.ylabel("Number of nodes")
    plt.title(title)
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()

# Create output folder
output_dir = 'plots'
os.makedirs(output_dir, exist_ok=True)

# Medium network plots
plot_hist(in_med, "Medium Graph: In-degree Distribution", os.path.join(output_dir, "medium_in_deg_hist.png"))
plot_hist(out_med, "Medium Graph: Out-degree Distribution", os.path.join(output_dir, "medium_out_deg_hist.png"))
plot_loglog(in_med, "Medium Graph: In-degree Distribution (log-log)", os.path.join(output_dir, "medium_in_deg_loglog.png"))
plot_loglog(out_med, "Medium Graph: Out-degree Distribution (log-log)", os.path.join(output_dir, "medium_out_deg_loglog.png"))

# Large network plots
plot_hist(in_large, "Large Graph: In-degree Distribution", os.path.join(output_dir, "large_in_deg_hist.png"))
```

```
plot_hist(out_large, "Large Graph: Out-degree Distribution", os.path.join(output_dir, "large_out_deg_hist.png"))
plot_loglog(in_large, "Large Graph: In-degree Distribution (log-log)", os.path.join(output_dir, "large_in_deg_loglog.png"))
plot_loglog(out_large, "Large Graph: Out-degree Distribution (log-log)", os.path.join(output_dir, "large_out_deg_loglog.png"))

print(f"All plots saved in '{output_dir}' folder.")
```

Python Code for 2.4

```
import networkx as nx
import pandas as pd

medium_file = 'snacs2025-student4581733-medium.tsv'
large_file = 'snacs2025-student4581733-large.tsv'

medium_edges = pd.read_csv(medium_file, sep='\t', header=None, names=['source', 'target'])
large_edges = pd.read_csv(large_file, sep='\t', header=None, names=['source', 'target'])

G_medium = nx.from_pandas_edgelist(medium_edges, source='source', target='target', create_using=nx.DiGraph())
G_large = nx.from_pandas_edgelist(large_edges, source='source', target='target', create_using=nx.DiGraph())

def components_stats(G, name):
    wccs = list(nx.weakly_connected_components(G))
    num_wcc = len(wccs)
    largest_wcc = max(wccs, key=len)
    G_lwcc = G.subgraph(largest_wcc)

    sccs = list(nx.strongly_connected_components(G))
    num_scc = len(sccs)
    largest_scc = max(sccs, key=len)
    G_lscc = G.subgraph(largest_scc)

    print(f"{name} Graph:")
    print(f"  Number of weakly connected components: {num_wcc}")
    print(f"  Largest WCC: nodes = {G_lwcc.number_of_nodes()}, edges = {G_lwcc.number_of_edges()}")
    print(f"  Number of strongly connected components: {num_scc}")
    print(f"  Largest SCC: nodes = {G_lscc.number_of_nodes()}, edges = {G_lscc.number_of_edges()}")
    print()

components_stats(G_medium, "Medium")
components_stats(G_large, "Large")
```

Python Code for 2.5

```
import networkx as nx
import pandas as pd

medium_file = 'snacs2025-student4581733-medium.tsv'
large_file = 'snacs2025-student4581733-large.tsv'

medium_edges = pd.read_csv(medium_file, sep='\t', header=None,
                           names=['source', 'target'])
large_edges = pd.read_csv(large_file, sep='\t', header=None,
                          names=['source', 'target'])

G_medium = nx.from_pandas_edgelist(medium_edges, source='source',
                                   target='target',
                                   create_using=nx.DiGraph())
G_large = nx.from_pandas_edgelist(large_edges, source='source',
                                  target='target',
                                  create_using=nx.DiGraph())

def compute_avg_clustering(G, name):
    G_undir = G.to_undirected()
    avg_clust = nx.average_clustering(G_undir)
    print(f"{name} Graph: Average clustering coefficient = {avg_clust:.4f}")
    print("  (Direction ignored: treated graph as undirected for clustering)\n")

compute_avg_clustering(G_medium, "Medium")
compute_avg_clustering(G_large, "Large")
```

Python Code for 2.6

```
import networkx as nx
import pandas as pd
```

```
import matplotlib.pyplot as plt
import random
import os

# Load data and create directed graphs
medium_edges = pd.read_csv('snacs2025-student4581733-medium.tsv', sep='\t', header=None, names=['source','target'])
large_edges = pd.read_csv('snacs2025-student4581733-large.tsv', sep='\t', header=None, names=['source','target'])
G_medium = nx.from_pandas_edgelist(medium_edges, create_using=nx.DiGraph())
G_large = nx.from_pandas_edgelist(large_edges, create_using=nx.DiGraph())

# Get largest weakly connected component
def get_lwcc(G):
    largest_wcc_nodes = max(nx.weakly_connected_components(G), key=len)
    return G.subgraph(largest_wcc_nodes).copy()

# Distance distribution function
def distance_distribution(G, filename, sample_size=None):
    G_undir = G.to_undirected()
    distances = []
    if sample_size:
        nodes = list(G_undir.nodes())
        for _ in range(sample_size):
            sp_lengths = nx.single_source_shortest_path_length(G_undir, random.choice(nodes))
            distances.extend(sp_lengths.values())
    else:
        for sp_lengths in nx.all_pairs_shortest_path_length(G_undir):
            distances.extend(sp_lengths[1].values())
    plt.hist(distances, bins=range(max(distances)+1), density=True, edgecolor='black')
    plt.xlabel("Distance")
    plt.ylabel("Fraction of node pairs")
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()

# Compute and plot distance distributions
distance_distribution(get_lwcc(G_medium), "plots_distance/medium_distance_distribution.png")
distance_distribution(get_lwcc(G_large), "plots_distance/large_distance_distribution.png", sample_size=1000)
```

Python Code for 2.7

```
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt
import random
import os

medium_file = 'snacs2025-student4581733-medium.tsv'
large_file = 'snacs2025-student4581733-large.tsv'

medium_edges = pd.read_csv(medium_file, sep='\t', header=None, names=['source','target'])
large_edges = pd.read_csv(large_file, sep='\t', header=None, names=['source','target'])

G_medium = nx.from_pandas_edgelist(medium_edges, source='source', target='target', create_using=nx.DiGraph())
G_large = nx.from_pandas_edgelist(large_edges, source='source', target='target', create_using=nx.DiGraph())

output_dir = 'plots_average_distance'
os.makedirs(output_dir, exist_ok=True)

def get_lwcc(G):
    wccs = list(nx.weakly_connected_components(G))
    largest_wcc_nodes = max(wccs, key=len)
    return G.subgraph(largest_wcc_nodes).copy()

def avg_distance_and_plot(G, name, filename, sample_size=None):
    G_undir = G.to_undirected()
    if sample_size:
        nodes = list(G_undir.nodes())
        distances = []
        for _ in range(sample_size):
            source = random.choice(nodes)
            sp_lengths = nx.single_source_shortest_path_length(G_undir, source)
            distances.extend(sp_lengths.values())
    else:
        distances = []
        for sp_lengths in nx.all_pairs_shortest_path_length(G_undir):
            distances.extend(sp_lengths[1].values())
    avg_dist = sum(distances)/len(distances)
    print(f"{name} LWCC: Average distance = {avg_dist:.4f}")
    plt.figure(figsize=(6,4))
```

```
    plt.hist(distances, bins=range(max(distances)+1), color='lightcoral', edgecolor='black', density=True)
    plt.xlabel("Distance (shortest path length)")
    plt.ylabel("Fraction of node pairs")
    plt.title(f"{name}: Distance Distribution in LWCC")
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()
    print(f"{name} LWCC: Distance distribution plot saved as '{filename}'")

G_medium_lwcc = get_lwcc(G_medium)
avg_distance_and_plot(G_medium_lwcc, "Medium", os.path.join(output_dir,"medium_avg_distance_distribution.png"))

G_large_lwcc = get_lwcc(G_large)
avg_distance_and_plot(G_large_lwcc, "Large", os.path.join(output_dir,"large_avg_distance_distribution.png"), sample_size=1000)
```