In [1]:

```
import pandas as pd
import numpy as np
import csv
import matplotlib.pyplot as plt
from itertools import combinations
from tqdm import tqdm
import warnings
warnings.filterwarnings('ignore')
import networkx as nx
```

In [2]:

```
railway_data = pd.read_csv("Train_details_22122017.csv")
modified_railway_data = railway_data.dropna()
modified_railway_data
```

Out[2]:

	Train No	Train Name	SEQ	Station Code	Station Name	Arrival time	Departure Time	Distance	Source Station	Source Station Name	Des
0	107	SWV-MAO-VLNK	1	SWV	SAWANTWADI R	00:00:00	10:25:00	0	SWV	SAWANTWADI ROAD	
1	107	SWV-MAO-VLNK	2	THVM	THIVIM	11:06:00	11:08:00	32	SWV	SAWANTWADI ROAD	
2	107	SWV-MAO-VLNK	3	KRMI	KARMALI	11:28:00	11:30:00	49	SWV	SAWANTWADI ROAD	
3	107	SWV-MAO-VLNK	4	MAO	MADGOAN JN.	12:10:00	00:00:00	78	SWV	SAWANTWADI ROAD	
4	108	VLNK-MAO-SWV	1	MAO	MADGOAN JN.	00:00:00	20:30:00	0	MAO	MADGOAN JN.	
...	...	...	...	...	...	...	...	...	...	...	
186119	99908	EMU	8	AKRD	AKURDI	23:30:00	23:31:00	19	PUNE	PUNE JN.	
186120	99908	EMU	9	DEHR	DEHU ROAD	23:35:00	23:36:00	24	PUNE	PUNE JN.	
186121	99908	EMU	10	BGWI	BEGDAEWAI	23:39:00	23:40:00	28	PUNE	PUNE JN.	
186122	99908	EMU	11	GRWD	GHORAWADI	23:41:00	23:42:00	31	PUNE	PUNE JN.	
186123	99908	EMU	12	TGN	TALEGAON	23:50:00	00:00:00	34	PUNE	PUNE JN.	

186114 rows × 12 columns

In [ ]:

```
STATION_CODE = 'Station Code'
SOURCE_STATION = 'Source Station'
DESTINATION_STATION = 'Destination Station'
TRAIN_NAME = 'Train Name'
TRAIN_NO = 'Train No'
DISTANCE = 'Distance'

import networkx as nx
import numpy as np
from tqdm import tqdm

STATION_CODE = 'Station Code'
SOURCE_STATION = 'Source Station'
DESTINATION_STATION = 'Destination Station'
TRAIN_NAME = 'Train Name'
TRAIN_NO = 'Train No'
DISTANCE = 'Distance'
```

```python
def generate_graph(railway_data, filter_nodes=None, distance_weighted=False):
    # Make an empty directed graph
    graph = nx.DiGraph()
    stations = None

    # Add all stations if there are none to filter
    if filter_nodes is None:
        stations = np.unique(railway_data[STATION_CODE])
    else:
        stations = filter_nodes

    graph.add_nodes_from(stations)

    # Find all unique trains
    trains = np.unique(railway_data[TRAIN_NAME].astype('str'))

    print("Generating graph from train routes...")
    for train_name in tqdm(trains, desc="Processing trains", unit="train"):
        # Get the train route
        train_route = railway_data.loc[railway_data[TRAIN_NAME] == train_name]
        stations_in_route = train_route[STATION_CODE].to_list()
        station_distances = train_route[DISTANCE].to_list()

        # Make a connected graph out of all stations in the route
        for i in range(len(stations_in_route)):
            for j in range(i + 1, len(stations_in_route)):
                src = stations_in_route[i]
                dst = stations_in_route[j]

                # Only add edge if node is present in filter (if applied)
                if filter_nodes is None or src in filter_nodes or dst in filter_nodes:
                    if distance_weighted:
                        # Distance weight
                        try:
                            distance = int(station_distances[j]) - int(station_distance
                        except ValueError:
                            distance = 1  # fallback if distance is not clean
                    else:
                        distance = 1

                    # Add or update the edge
                    if graph.has_edge(src, dst):
                        graph[src][dst]['weight'] += distance
                    else:
                        graph.add_edge(src, dst, weight=distance, label=train_name)

    print("Graph generation complete.")
    return graph
```

In [32]:

```python
import pickle
try:
  with open("train_count_weighted_graph.gpickle", "rb") as f:
    railway_network = pickle.load(f)
except:
  railway_network = generate_graph(modified_railway_data)
  with open("train_count_weighted_graph.gpickle", "wb") as f:
    pickle.dump(railway_network, f)
```

In [4]:

```python
# print(f"Graph name: {railway_network.name}")
print(f"Number of nodes: {railway_network.number_of_nodes()}")
print(f"Number of edges: {railway_network.number_of_edges()}")
print(f"Is directed: {railway_network.is_directed()}")
print(f"Is multigraph: {railway_network.is_multigraph()}")
num_weakly_connected = nx.number_weakly_connected_components(railway_network)
print(f"Number of weakly connected components: {num_weakly_connected}")
```

```python
num_strongly_connected = nx.number_strongly_connected_components(railway_network)
print(f"Number of strongly connected components: {num_strongly_connected}")
# Calculate average in-degree
avg_in_degree = sum(dict(railway_network.in_degree()).values()) / railway_network.numbe

# Calculate average out-degree
avg_out_degree = sum(dict(railway_network.out_degree()).values()) / railway_network.nur

print(f"Average in-degree: {avg_in_degree:.2f}")
print(f"Average out-degree: {avg_out_degree:.2f}")
```

```
Number of nodes: 8147
Number of edges: 902602
Is directed: True
Is multigraph: False
Number of weakly connected components: 7
Number of strongly connected components: 9
Average in-degree: 110.79
Average out-degree: 110.79
```

In [5]:

```python
def get_top_degree_nodes(graph, top_k=10, degree_type='total'):
    if degree_type == 'total':
        degree_dict = dict(graph.degree())
    elif degree_type == 'in':
        degree_dict = dict(graph.in_degree())
    elif degree_type == 'out':
        degree_dict = dict(graph.out_degree())
    else:
        raise ValueError("Invalid degree_type. Choose from 'total', 'in', 'out'.")
    top_nodes = sorted(degree_dict.items(), key=lambda x: x[1], reverse=True)[:top_k]
    return [node for node, _ in top_nodes]

def get_top_degree_nodes_in_largest_component(graph, top_k=5):
    G_undirected = graph.to_undirected()
    largest_cc = max(nx.connected_components(G_undirected), key=len)
    subgraph = graph.subgraph(largest_cc)
    degrees = dict(subgraph.degree())
    top_nodes = sorted(degrees.items(), key=lambda x: x[1], reverse=True)[:top_k]
    return [node for node, _ in top_nodes]

def print_station_names(top_nodes, station_df):
    """
    Map station codes to their station names and print them.
    Assumes 'Station Code' and 'Station Name' columns exist.
    """
    # Build unique mapping from code to name
    code_to_name = dict(station_df[['Station Code', 'Station Name']].drop_duplicates()
    print("Top Stations by Degree:")
    for code in top_nodes:
        name = code_to_name.get(code, "Unknown Station")
        print(f"{code} - {name}")

top_nodes = get_top_degree_nodes_in_largest_component(railway_network, 10)
print_station_names(top_nodes, railway_data)
```

```
Top Stations by Degree:
HWH - HOWRAH JN.
BZA - VIJAYWADA JN
CNB - KANPUR CENTR
BSB - VARANASI JN.
GZB - GHAZIABAD JN
KYN - KALYAN JN
ET - ITARSI
LKO - LUCKNOW JN.
ADI - AHMEDABAD
MTJ - MATHURA JN.
```

In [ ]:

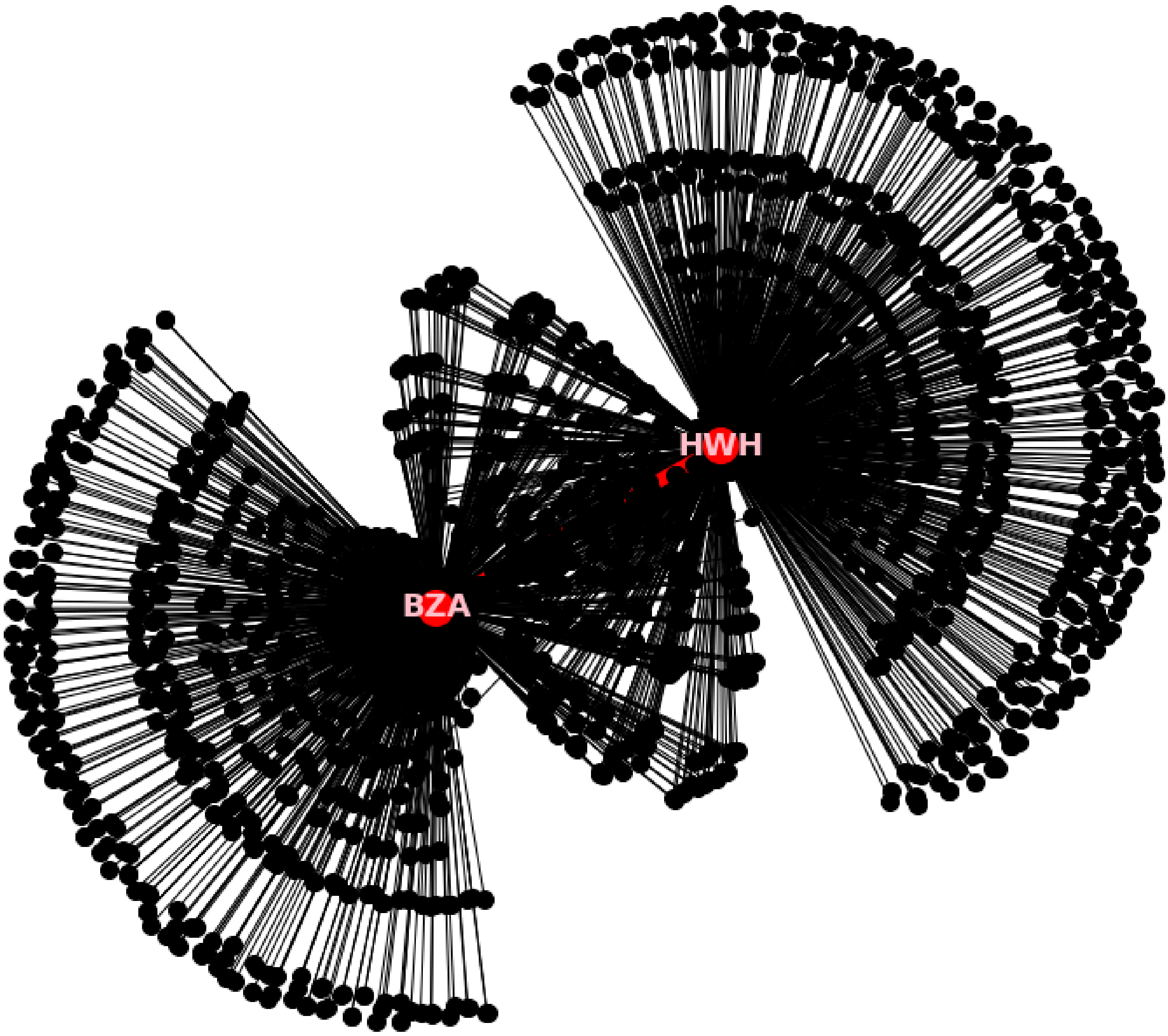
```python
from tqdm import tqdm
import networkx as nx
```

```python
import matplotlib.pyplot as plt

def get_subgraph(railway_data, graph, subgraph_nodes, distance_weighted=False, plot=Tru
    # Initialize an empty directed graph
    sub_graph = nx.empty_graph(0, create_using=nx.DiGraph())

    # Iterate over all node pairs with a single clean progress bar
    node_pairs = [
        (subgraph_nodes[i], subgraph_nodes[j])
        for i in range(len(subgraph_nodes))
        for j in range(i + 1, len(subgraph_nodes))
    ]
    for node1, node2 in tqdm(node_pairs, desc="Building subgraph", ncols=100, leave=Fal
        sub_sub_graph = generate_graph(railway_data, [node1, node2], distance_weighted
        sub_sub_graph = nx.compose(sub_sub_graph, graph.subgraph([node1, node2]))
        sub_graph = nx.compose(sub_graph, sub_sub_graph)

    # Convert to undirected
    sub_graph = sub_graph.to_undirected()

    # Plot the graph if enabled
    if plot:
        plt.figure(figsize=(10, 10))
        pos = nx.spring_layout(sub_graph)
        nx.draw(sub_graph, pos, node_color='k', node_size=100)

        # Compute shortest paths with a clean progress bar
        path_edges = []
        path_nodes = []
        for node1, node2 in tqdm(node_pairs, desc="Finding shortest paths", ncols=100,
            try:
                path = nx.shortest_path(graph, source=node1, target=node2)
                path_edges.extend([(path[i], path[i + 1]) for i in range(len(path) - 1
                path_nodes.extend(path)
            except:
                continue

        nx.draw_networkx_nodes(sub_graph, pos, nodelist=path_nodes, node_color='b')
        nx.draw_networkx_nodes(sub_graph, pos, nodelist=subgraph_nodes, node_color='r'
        nx.draw_networkx_edges(sub_graph, pos, edgelist=path_edges, edge_color='r', wid

        nx.draw_networkx_labels(sub_graph.subgraph(subgraph_nodes), pos, font_color='pi
        connecting_nodes = [station for station in path_nodes if station not in subgrap
        nx.draw_networkx_labels(sub_graph.subgraph(connecting_nodes), pos, font_color=

        plt.axis('equal')
        plt.show()

    return sub_graph
```

In [ ]:

```python
# subgraph = get_subgraph(modified_railway_data, railway_network, ['DAA', 'SWV'])
subgraph = get_subgraph(modified_railway_data, railway_network, ['HWH', 'BZA'])
```

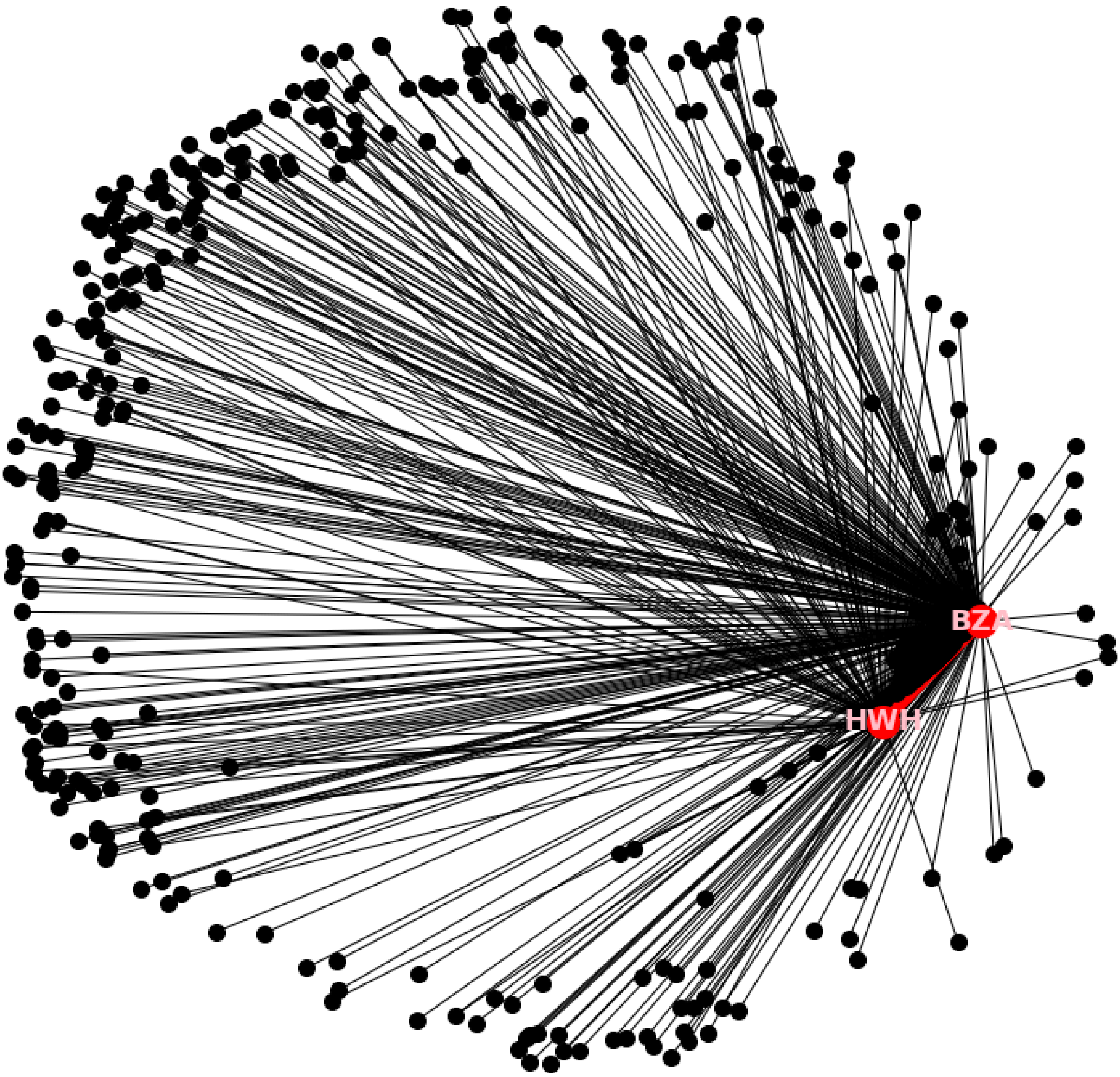
```
Building subgraph:   0%|                                                     | 0/1 [0
0:00<?, ?it/s]
Generating graph from train routes...
Processing trains: 100%|██████████| 7580/7580 [01:42<00:00, 74.01train/s]

Graph generation complete.
```



In [ ]:

```
subgraph = get_subgraph(modified_railway_data, railway_network, ['HWH', 'BZA'], distanc
```

```
Building subgraph:   0%|                                                  | 0/1 [0
0:00<?, ?it/s]
Generating graph from train routes...
Processing trains: 100%|██████████| 7580/7580 [01:41<00:00, 74.70train/s]

Graph generation complete.
```



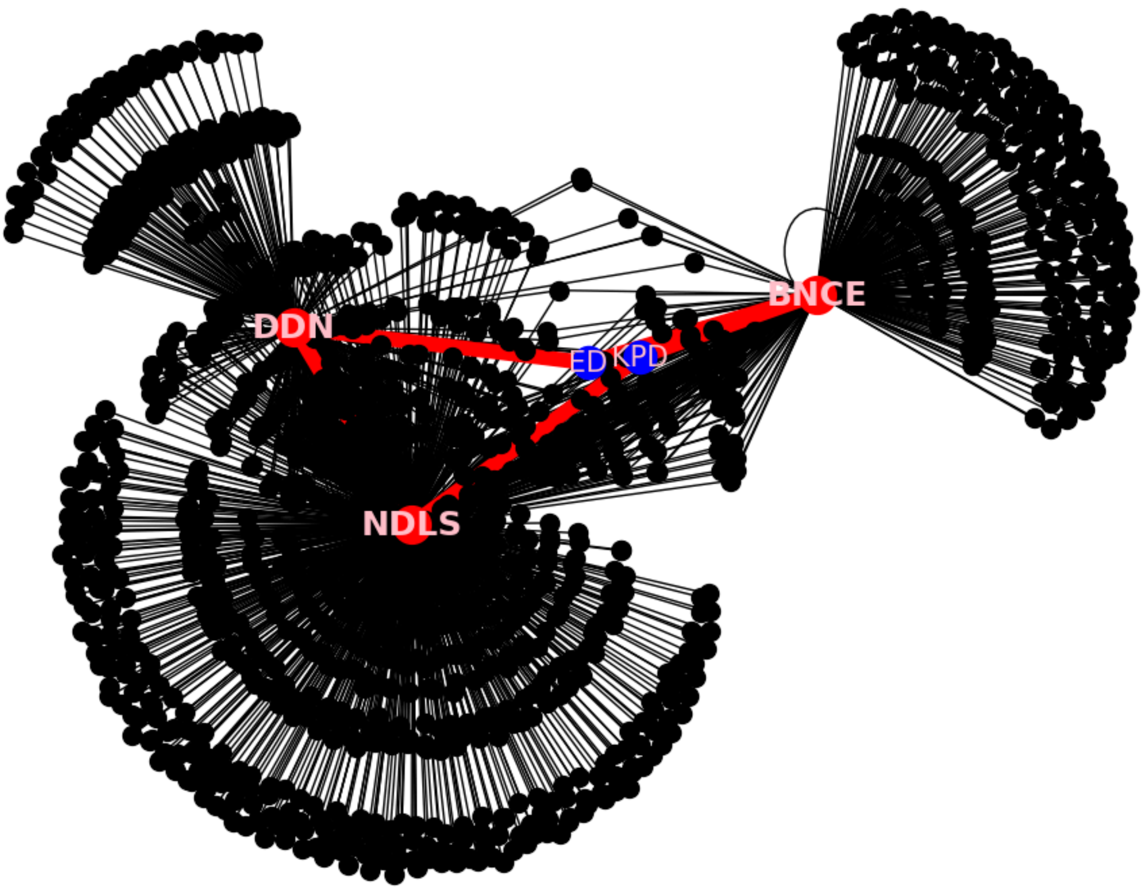
In [ ]:

```
subgraph = get_subgraph(modified_railway_data, railway_network, ['NDLS', 'BNCE', 'DDN']
```

```
Building subgraph:   0%|                                                      | 0/3 [0
0:00<?, ?it/s]
Generating graph from train routes...
Processing trains:   0%|          | 0/7580 [00:00<?, ?train/s]
Processing trains:   0%|          | 8/7580 [00:00<01:40, 75.48train/s]
Processing trains:   0%|          | 16/7580 [00:00<01:40, 75.60train/s]
Processing trains:   0%|          | 24/7580 [00:00<01:40, 74.92train/s]
Processing trains:   0%|          | 32/7580 [00:00<01:39, 75.51train/s]
Processing trains:   1%|          | 40/7580 [00:00<01:40, 75.33train/s]
Processing trains:   1%|          | 48/7580 [00:00<01:38, 76.59train/s]
Processing trains:   1%|          | 57/7580 [00:00<01:36, 77.73train/s]
Processing trains:   1%|          | 65/7580 [00:00<01:36, 77.49train/s]
Processing trains:   1%|          | 73/7580 [00:00<01:37, 77.35train/s]
Processing trains:   1%|          | 81/7580 [00:01<01:38, 75.84train/s]
Processing trains:   1%|          | 89/7580 [00:01<01:37, 76.71train/s]
Processing trains:   1%|▏         | 97/7580 [00:01<01:37, 76.72train/s]
Processing trains:   1%|▏         | 106/7580 [00:01<01:35, 78.12train/s]
Processing trains:   2%|▏         | 115/7580 [00:01<01:32, 80.29train/s]
Processing trains:   2%|▏         | 124/7580 [00:01<01:32, 80.96train/s]
Processing trains:   2%|▏         | 133/7580 [00:01<01:33, 79.78train/s]
Processing trains:   2%|▏         | 141/7580 [00:01<01:36, 76.73train/s]
Processing trains:   2%|▏         | 149/7580 [00:01<01:35, 77.60train/s]
Processing trains:   2%|▏         | 157/7580 [00:02<01:43, 71.87train/s]
Processing trains:   2%|▏         | 165/7580 [00:02<01:42, 72.26train/s]
```

```
Processing trains:  98%|██████████| 7436/7580 [01:38<00:01, 77.31train/s]
Processing trains:  98%|██████████| 7444/7580 [01:38<00:01, 76.06train/s]
Processing trains:  98%|██████████| 7452/7580 [01:38<00:01, 75.89train/s]
Processing trains:  98%|██████████| 7460/7580 [01:38<00:01, 75.97train/s]
Processing trains:  99%|██████████| 7468/7580 [01:38<00:01, 75.66train/s]
Processing trains:  99%|██████████| 7476/7580 [01:38<00:01, 75.23train/s]
Processing trains:  99%|██████████| 7484/7580 [01:39<00:01, 75.70train/s]
Processing trains:  99%|██████████| 7492/7580 [01:39<00:01, 75.60train/s]
Processing trains:  99%|██████████| 7500/7580 [01:39<00:01, 75.84train/s]
Processing trains:  99%|██████████| 7508/7580 [01:39<00:00, 75.72train/s]
Processing trains:  99%|██████████| 7516/7580 [01:39<00:00, 75.86train/s]
Processing trains:  99%|██████████| 7524/7580 [01:39<00:00, 75.41train/s]
Processing trains:  99%|██████████| 7532/7580 [01:39<00:00, 74.60train/s]
Processing trains:  99%|██████████| 7540/7580 [01:39<00:00, 75.13train/s]
Processing trains: 100%|██████████| 7548/7580 [01:39<00:00, 74.69train/s]
Processing trains: 100%|██████████| 7556/7580 [01:40<00:00, 74.35train/s]
Processing trains: 100%|██████████| 7564/7580 [01:40<00:00, 74.52train/s]
Processing trains: 100%|██████████| 7572/7580 [01:40<00:00, 73.64train/s]
Processing trains: 100%|██████████| 7580/7580 [01:40<00:00, 75.55train/s]
```

```
Graph generation complete.
```



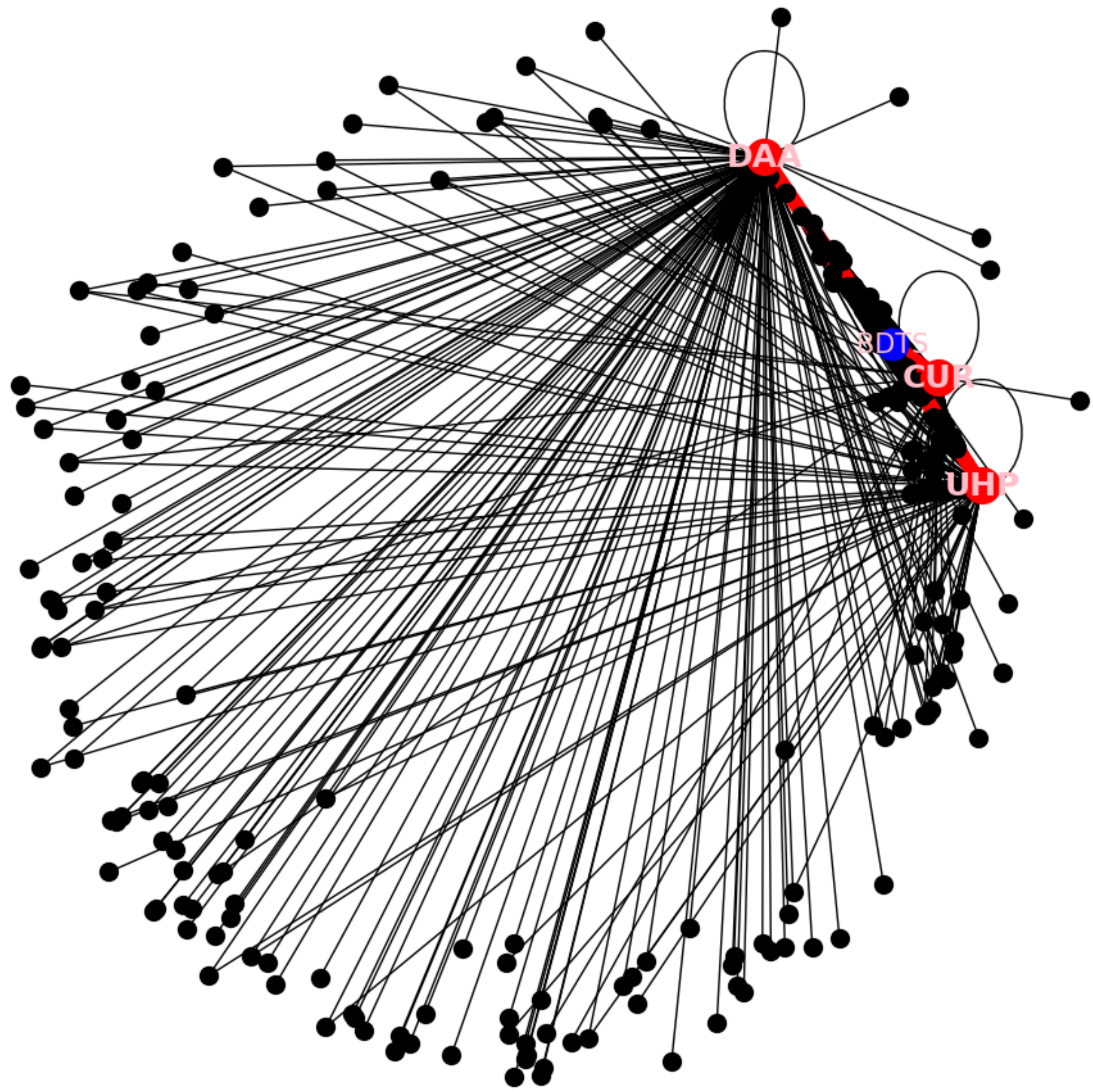
In [ ]:

```
subgraph = get_subgraph(modified_railway_data, railway_network, ['DAA', 'UHP', 'CUR'],
```

```
Building subgraph:   0%|                                                  | 0/3 [0
0:00<?, ?it/s]
```

```
Processing trains:  94%|█████████▍| 7111/7580 [01:34<00:05, 80.69train/s]
Processing trains:  94%|█████████▍| 7120/7580 [01:34<00:05, 80.12train/s]
Processing trains:  94%|█████████▍| 7129/7580 [01:34<00:05, 80.02train/s]
Processing trains:  94%|█████████▍| 7138/7580 [01:34<00:05, 79.89train/s]
Processing trains:  94%|█████████▍| 7146/7580 [01:34<00:05, 78.05train/s]
Processing trains:  94%|█████████▍| 7154/7580 [01:34<00:05, 76.36train/s]
Processing trains:  94%|█████████▍| 7162/7580 [01:34<00:05, 74.85train/s]
Processing trains:  95%|█████████▍| 7170/7580 [01:34<00:05, 75.09train/s]
Processing trains:  95%|█████████▍| 7178/7580 [01:35<00:05, 74.15train/s]
Processing trains:  95%|█████████▍| 7186/7580 [01:35<00:05, 74.52train/s]
Processing trains:  95%|█████████▍| 7194/7580 [01:35<00:05, 74.27train/s]
Processing trains:  95%|█████████▌| 7202/7580 [01:35<00:05, 73.90train/s]
Processing trains:  95%|█████████▌| 7210/7580 [01:35<00:05, 73.67train/s]
Processing trains:  95%|█████████▌| 7218/7580 [01:35<00:04, 75.11train/s]
Processing trains:  95%|█████████▌| 7226/7580 [01:35<00:04, 74.57train/s]
Processing trains:  95%|█████████▌| 7234/7580 [01:35<00:04, 73.89train/s]
Processing trains:  96%|█████████▌| 7242/7580 [01:35<00:04, 73.52train/s]
Processing trains:  96%|█████████▌| 7250/7580 [01:36<00:04, 72.12train/s]
Processing trains:  96%|█████████▌| 7258/7580 [01:36<00:04, 69.29train/s]
Processing trains:  96%|█████████▌| 7266/7580 [01:36<00:04, 71.63train/s]
Processing trains:  96%|█████████▌| 7274/7580 [01:36<00:04, 73.35train/s]
Processing trains:  96%|█████████▌| 7282/7580 [01:36<00:04, 74.05train/s]
Processing trains:  96%|█████████▌| 7290/7580 [01:36<00:03, 74.83train/s]
Processing trains:  96%|█████████▋| 7298/7580 [01:36<00:03, 74.57train/s]
Processing trains:  96%|█████████▋| 7306/7580 [01:36<00:03, 70.87train/s]
Processing trains:  96%|█████████▋| 7314/7580 [01:36<00:03, 71.75train/s]
Processing trains:  97%|█████████▋| 7322/7580 [01:37<00:03, 73.70train/s]
Processing trains:  97%|█████████▋| 7330/7580 [01:37<00:03, 74.96train/s]
Processing trains:  97%|█████████▋| 7338/7580 [01:37<00:03, 74.28train/s]
Processing trains:  97%|█████████▋| 7346/7580 [01:37<00:03, 73.88train/s]
Processing trains:  97%|█████████▋| 7354/7580 [01:37<00:03, 74.53train/s]
Processing trains:  97%|█████████▋| 7362/7580 [01:37<00:02, 74.03train/s]
Processing trains:  97%|█████████▋| 7370/7580 [01:37<00:02, 73.62train/s]
Processing trains:  97%|█████████▋| 7378/7580 [01:37<00:02, 74.63train/s]
Processing trains:  97%|█████████▋| 7387/7580 [01:37<00:02, 76.71train/s]
Processing trains:  98%|█████████▊| 7396/7580 [01:37<00:02, 78.40train/s]
Processing trains:  98%|█████████▊| 7404/7580 [01:38<00:02, 78.77train/s]
Processing trains:  98%|█████████▊| 7413/7580 [01:38<00:02, 79.37train/s]
Processing trains:  98%|█████████▊| 7421/7580 [01:38<00:02, 78.88train/s]
Processing trains:  98%|█████████▊| 7429/7580 [01:38<00:01, 77.98train/s]
Processing trains:  98%|█████████▊| 7437/7580 [01:38<00:01, 77.53train/s]
Processing trains:  98%|█████████▊| 7445/7580 [01:38<00:01, 77.80train/s]
Processing trains:  98%|█████████▊| 7453/7580 [01:38<00:01, 76.81train/s]
Processing trains:  98%|█████████▊| 7461/7580 [01:38<00:01, 74.97train/s]
Processing trains:  99%|█████████▊| 7469/7580 [01:38<00:01, 74.47train/s]
Processing trains:  99%|█████████▊| 7478/7580 [01:39<00:01, 76.25train/s]
Processing trains:  99%|█████████▉| 7487/7580 [01:39<00:01, 78.99train/s]
Processing trains:  99%|█████████▉| 7496/7580 [01:39<00:01, 80.38train/s]
Processing trains:  99%|█████████▉| 7505/7580 [01:39<00:00, 79.54train/s]
Processing trains:  99%|█████████▉| 7513/7580 [01:39<00:00, 79.06train/s]
Processing trains:  99%|█████████▉| 7521/7580 [01:39<00:00, 78.12train/s]
Processing trains:  99%|█████████▉| 7529/7580 [01:39<00:00, 77.80train/s]
Processing trains:  99%|█████████▉| 7537/7580 [01:39<00:00, 77.62train/s]
Processing trains: 100%|█████████▉| 7545/7580 [01:39<00:00, 77.30train/s]
Processing trains: 100%|█████████▉| 7553/7580 [01:39<00:00, 77.39train/s]
Processing trains: 100%|█████████▉| 7561/7580 [01:40<00:00, 77.09train/s]
Processing trains: 100%|█████████▉| 7569/7580 [01:40<00:00, 76.15train/s]
Processing trains: 100%|██████████| 7580/7580 [01:40<00:00, 75.53train/s]
```

```
Graph generation complete.
```



In [ ]:

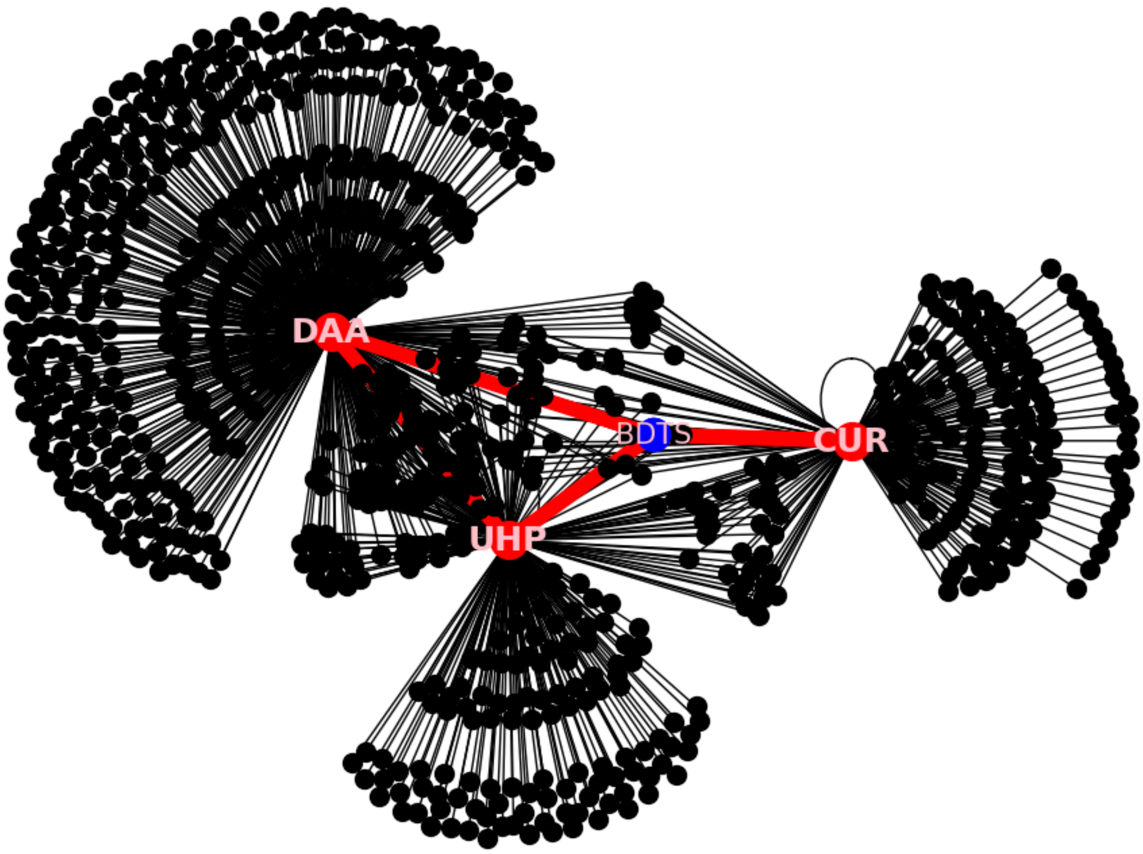
```
subgraph = get_subgraph(modified_railway_data, railway_network, ['DAA', 'UHP', 'CUR'])
```

```
Building subgraph:   0%|                                                       | 0/3 [00:00<?, ?it/s]
Generating graph from train routes...
Processing trains:   0%|          | 0/7580 [00:00<?, ?train/s]
Processing trains:   0%|          | 8/7580 [00:00<01:36, 78.68train/s]
Processing trains:   0%|          | 17/7580 [00:00<01:33, 80.76train/s]
Processing trains:   0%|          | 26/7580 [00:00<01:33, 81.16train/s]
Processing trains:   0%|          | 35/7580 [00:00<01:32, 81.99train/s]
Processing trains:   1%|          | 44/7580 [00:00<01:32, 81.39train/s]
Processing trains:   1%|          | 53/7580 [00:00<01:33, 80.45train/s]
Processing trains:   1%|          | 62/7580 [00:00<01:33, 80.42train/s]
Processing trains:   1%|          | 71/7580 [00:00<01:33, 80.26train/s]
Processing trains:   1%|          | 80/7580 [00:00<01:34, 79.19train/s]
Processing trains:   1%|          | 88/7580 [00:01<01:34, 79.28train/s]
Processing trains:   1%|▏         | 97/7580 [00:01<01:33, 79.93train/s]
Processing trains:   1%|▏         | 105/7580 [00:01<01:34, 78.97train/s]
Processing trains:   1%|▏         | 113/7580 [00:01<01:35, 78.38train/s]
Processing trains:   2%|▏         | 121/7580 [00:01<01:35, 77.80train/s]
Processing trains:   2%|▏         | 129/7580 [00:01<01:35, 78.22train/s]
Processing trains:   2%|▏         | 138/7580 [00:01<01:34, 79.16train/s]
Processing trains:   2%|▏         | 147/7580 [00:01<01:33, 79.71train/s]
Processing trains:   2%|▏         | 155/7580 [00:01<01:33, 79.44train/s]
Processing trains:   2%|▏         | 164/7580 [00:02<01:32, 79.81train/s]
Processing trains:   2%|▏         | 172/7580 [00:02<01:33, 79.29train/s]
```

```
Processing trains:  99%|█████████▉| 7485/7580 [01:38<00:01, 75.61train/s]
Processing trains:  99%|█████████▉| 7493/7580 [01:38<00:01, 76.66train/s]
Processing trains:  99%|█████████▉| 7501/7580 [01:38<00:01, 76.34train/s]
Processing trains:  99%|█████████▉| 7509/7580 [01:38<00:00, 75.31train/s]
Processing trains:  99%|█████████▉| 7517/7580 [01:38<00:00, 75.06train/s]
Processing trains:  99%|█████████▉| 7526/7580 [01:38<00:00, 76.81train/s]
Processing trains:  99%|█████████▉| 7534/7580 [01:38<00:00, 75.57train/s]
Processing trains:  99%|█████████▉| 7542/7580 [01:38<00:00, 74.96train/s]
Processing trains: 100%|█████████▉| 7550/7580 [01:38<00:00, 75.10train/s]
Processing trains: 100%|█████████▉| 7558/7580 [01:39<00:00, 75.96train/s]
Processing trains: 100%|█████████▉| 7566/7580 [01:39<00:00, 76.98train/s]
Processing trains: 100%|██████████| 7580/7580 [01:39<00:00, 76.28train/s]
Building subgraph:  33%|███████████████                               | 1/3 [01:39<03:1
8, 99.48s/it]
Graph generation complete.
Generating graph from train routes...
Processing trains:   0%|          | 0/7580 [00:00<?, ?train/s]
Processing trains:   0%|          | 9/7580 [00:00<01:31, 82.80train/s]
Processing trains:   0%|          | 18/7580 [00:00<01:35, 79.43train/s]
Processing trains:   0%|          | 26/7580 [00:00<01:36, 78.48train/s]
Processing trains:   0%|          | 34/7580 [00:00<01:40, 74.76train/s]
Processing trains:   1%|          | 43/7580 [00:00<01:37, 77.16train/s]
Processing trains:   1%|          | 52/7580 [00:00<01:36, 78.27train/s]
Processing trains:   1%|          | 61/7580 [00:00<01:35, 79.12train/s]
Processing trains:   1%|          | 69/7580 [00:00<01:36, 78.23train/s]
Processing trains:   1%|          | 77/7580 [00:00<01:37, 76.94train/s]
Processing trains:   1%|          | 85/7580 [00:01<01:37, 77.12train/s]
Processing trains:   1%|          | 94/7580 [00:01<01:34, 79.35train/s]
Processing trains:   1%|▏         | 103/7580 [00:01<01:33, 80.36train/s]
Processing trains:   1%|▏         | 112/7580 [00:01<01:31, 81.48train/s]
Processing trains:   2%|▏         | 121/7580 [00:01<01:32, 80.44train/s]
Processing trains:   2%|▏         | 130/7580 [00:01<01:32, 80.51train/s]
Processing trains:   2%|▏         | 139/7580 [00:01<01:32, 80.63train/s]
Processing trains:   2%|▏         | 148/7580 [00:01<01:32, 80.74train/s]
Processing trains:   2%|▏         | 157/7580 [00:01<01:31, 81.12train/s]
Processing trains:   2%|▏         | 166/7580 [00:02<01:31, 81.24train/s]
Processing trains:   2%|▏         | 175/7580 [00:02<01:32, 80.04train/s]
Processing trains:   2%|▏         | 184/7580 [00:02<01:32, 79.93train/s]
Processing trains:   3%|▎         | 193/7580 [00:02<01:32, 80.28train/s]
Processing trains:   3%|▎         | 202/7580 [00:02<01:33, 78.97train/s]
Processing trains:   3%|▎         | 210/7580 [00:02<01:33, 78.57train/s]
Processing trains:   3%|▎         | 218/7580 [00:02<01:33, 78.71train/s]
Processing trains:   3%|▎         | 226/7580 [00:02<01:34, 77.56train/s]
Processing trains:   3%|▎         | 234/7580 [00:02<01:34, 77.38train/s]
Processing trains:   3%|▎         | 242/7580 [00:03<01:34, 77.58train/s]
Processing trains:   3%|▎         | 250/7580 [00:03<01:33, 78.27train/s]
Processing trains:   3%|▎         | 258/7580 [00:03<01:34, 77.14train/s]
Processing trains:   4%|▎         | 266/7580 [00:03<01:35, 76.46train/s]
Processing trains:   4%|▎         | 274/7580 [00:03<01:35, 76.49train/s]
Processing trains:   4%|▎         | 282/7580 [00:03<01:34, 76.93train/s]
Processing trains:   4%|▍         | 290/7580 [00:03<01:37, 74.70train/s]
Processing trains:   4%|▍         | 299/7580 [00:03<01:34, 76.64train/s]
Processing trains:   4%|▍         | 307/7580 [00:03<01:36, 75.76train/s]
Processing trains:   4%|▍         | 315/7580 [00:04<01:35, 76.21train/s]
Processing trains:   4%|▍         | 323/7580 [00:04<01:34, 76.57train/s]
Processing trains:   4%|▍         | 331/7580 [00:04<01:34, 76.57train/s]
Processing trains:   4%|▍         | 339/7580 [00:04<01:34, 76.34train/s]
Processing trains:   5%|▍         | 347/7580 [00:04<01:36, 75.09train/s]
Processing trains:   5%|▍         | 355/7580 [00:04<01:35, 75.27train/s]
Processing trains:   5%|▍         | 363/7580 [00:04<01:36, 74.91train/s]
Processing trains:   5%|▍         | 371/7580 [00:04<01:36, 74.41train/s]
Processing trains:   5%|▌         | 380/7580 [00:04<01:34, 76.51train/s]
Processing trains:   5%|▌         | 388/7580 [00:04<01:33, 76.77train/s]
Processing trains:   5%|▌         | 397/7580 [00:05<01:31, 78.69train/s]
Processing trains:   5%|▌         | 406/7580 [00:05<01:29, 79.89train/s]
Processing trains:   5%|▌         | 415/7580 [00:05<01:29, 80.43train/s]
Processing trains:   6%|▌         | 424/7580 [00:05<01:29, 80.28train/s]
Processing trains:   6%|▌         | 433/7580 [00:05<01:28, 80.48train/s]
Processing trains:   6%|▌         | 442/7580 [00:05<01:29, 79.98train/s]
Processing trains:   6%|▌         | 450/7580 [00:05<01:30, 79.08train/s]
Processing trains:   6%|▌         | 458/7580 [00:05<01:30, 78.90train/s]
Processing trains:   6%|▌         | 466/7580 [00:05<01:31, 78.07train/s]
Processing trains:   6%|▋         | 474/7580 [00:06<01:31, 77.87train/s]
Processing trains:   6%|▋         | 482/7580 [00:06<01:31, 77.42train/s]
```

```
Processing trains:  91%|█████████ | 6896/7580 [01:29<00:08, 78.67train/s]
Processing trains:  91%|█████████ | 6904/7580 [01:30<00:08, 78.73train/s]
Processing trains:  91%|█████████ | 6913/7580 [01:30<00:08, 79.91train/s]
Processing trains:  91%|█████████▏| 6922/7580 [01:30<00:08, 80.67train/s]
Processing trains:  91%|█████████▏| 6931/7580 [01:30<00:08, 80.59train/s]
Processing trains:  92%|█████████▏| 6940/7580 [01:30<00:08, 76.85train/s]
Processing trains:  92%|█████████▏| 6949/7580 [01:30<00:08, 77.46train/s]
Processing trains:  92%|█████████▏| 6957/7580 [01:30<00:08, 71.92train/s]
Processing trains:  92%|█████████▏| 6966/7580 [01:30<00:08, 74.76train/s]
Processing trains:  92%|█████████▏| 6975/7580 [01:30<00:07, 76.80train/s]
Processing trains:  92%|█████████▏| 6984/7580 [01:31<00:07, 78.50train/s]
Processing trains:  92%|█████████▏| 6992/7580 [01:31<00:07, 78.52train/s]
Processing trains:  92%|█████████▏| 7000/7580 [01:31<00:07, 77.64train/s]
Processing trains:  92%|█████████▏| 7008/7580 [01:31<00:07, 73.43train/s]
Processing trains:  93%|█████████▎| 7016/7580 [01:31<00:07, 74.08train/s]
Processing trains:  93%|█████████▎| 7025/7580 [01:31<00:07, 76.62train/s]
Processing trains:  93%|█████████▎| 7034/7580 [01:31<00:07, 77.93train/s]
Processing trains:  93%|█████████▎| 7043/7580 [01:31<00:06, 79.48train/s]
Processing trains:  93%|█████████▎| 7052/7580 [01:31<00:06, 81.22train/s]
Processing trains:  93%|█████████▎| 7061/7580 [01:32<00:06, 81.84train/s]
Processing trains:  93%|█████████▎| 7070/7580 [01:32<00:06, 81.27train/s]
Processing trains:  93%|█████████▎| 7079/7580 [01:32<00:06, 81.87train/s]
Processing trains:  94%|█████████▎| 7088/7580 [01:32<00:05, 83.03train/s]
Processing trains:  94%|█████████▎| 7097/7580 [01:32<00:05, 82.57train/s]
Processing trains:  94%|█████████▎| 7106/7580 [01:32<00:05, 82.51train/s]
Processing trains:  94%|█████████▍| 7115/7580 [01:32<00:05, 82.06train/s]
Processing trains:  94%|█████████▍| 7124/7580 [01:32<00:05, 82.17train/s]
Processing trains:  94%|█████████▍| 7133/7580 [01:32<00:05, 81.86train/s]
Processing trains:  94%|█████████▍| 7142/7580 [01:33<00:05, 80.91train/s]
Processing trains:  94%|█████████▍| 7151/7580 [01:33<00:05, 80.07train/s]
Processing trains:  94%|█████████▍| 7160/7580 [01:33<00:05, 79.08train/s]
Processing trains:  95%|█████████▍| 7168/7580 [01:33<00:05, 78.38train/s]
Processing trains:  95%|█████████▍| 7176/7580 [01:33<00:05, 78.14train/s]
Processing trains:  95%|█████████▍| 7184/7580 [01:33<00:05, 77.37train/s]
Processing trains:  95%|█████████▍| 7192/7580 [01:33<00:05, 77.51train/s]
Processing trains:  95%|█████████▍| 7200/7580 [01:33<00:04, 77.14train/s]
Processing trains:  95%|█████████▌| 7209/7580 [01:33<00:04, 78.59train/s]
Processing trains:  95%|█████████▌| 7217/7580 [01:33<00:04, 78.09train/s]
Processing trains:  95%|█████████▌| 7225/7580 [01:34<00:04, 76.79train/s]
Processing trains:  95%|█████████▌| 7233/7580 [01:34<00:04, 76.21train/s]
Processing trains:  96%|█████████▌| 7241/7580 [01:34<00:04, 76.02train/s]
Processing trains:  96%|█████████▌| 7249/7580 [01:34<00:04, 76.67train/s]
Processing trains:  96%|█████████▌| 7257/7580 [01:34<00:04, 77.18train/s]
Processing trains:  96%|█████████▌| 7265/7580 [01:34<00:04, 77.75train/s]
Processing trains:  96%|█████████▌| 7273/7580 [01:34<00:03, 77.98train/s]
Processing trains:  96%|█████████▌| 7282/7580 [01:34<00:03, 79.54train/s]
Processing trains:  96%|█████████▌| 7291/7580 [01:34<00:03, 80.19train/s]
Processing trains:  96%|█████████▋| 7300/7580 [01:35<00:03, 79.37train/s]
Processing trains:  96%|█████████▋| 7309/7580 [01:35<00:03, 80.99train/s]
Processing trains:  97%|█████████▋| 7318/7580 [01:35<00:03, 80.36train/s]
Processing trains:  97%|█████████▋| 7327/7580 [01:35<00:03, 79.33train/s]
Processing trains:  97%|█████████▋| 7335/7580 [01:35<00:03, 78.07train/s]
Processing trains:  97%|█████████▋| 7343/7580 [01:35<00:03, 78.30train/s]
Processing trains:  97%|█████████▋| 7351/7580 [01:35<00:02, 78.21train/s]
Processing trains:  97%|█████████▋| 7360/7580 [01:35<00:02, 79.54train/s]
Processing trains:  97%|█████████▋| 7368/7580 [01:35<00:02, 79.41train/s]
Processing trains:  97%|█████████▋| 7376/7580 [01:36<00:02, 78.54train/s]
Processing trains:  97%|█████████▋| 7384/7580 [01:36<00:02, 78.79train/s]
Processing trains:  98%|█████████▊| 7392/7580 [01:36<00:02, 79.09train/s]
Processing trains:  98%|█████████▊| 7401/7580 [01:36<00:02, 80.20train/s]
Processing trains:  98%|█████████▊| 7410/7580 [01:36<00:02, 80.34train/s]
Processing trains:  98%|█████████▊| 7419/7580 [01:36<00:02, 79.62train/s]
Processing trains:  98%|█████████▊| 7427/7580 [01:36<00:01, 78.51train/s]
Processing trains:  98%|█████████▊| 7435/7580 [01:36<00:01, 77.58train/s]
Processing trains:  98%|█████████▊| 7443/7580 [01:36<00:01, 77.24train/s]
Processing trains:  98%|█████████▊| 7451/7580 [01:36<00:01, 77.26train/s]
Processing trains:  98%|█████████▊| 7459/7580 [01:37<00:01, 76.70train/s]
Processing trains:  99%|█████████▊| 7467/7580 [01:37<00:01, 76.52train/s]
Processing trains:  99%|█████████▊| 7475/7580 [01:37<00:01, 76.15train/s]
Processing trains:  99%|█████████▊| 7483/7580 [01:37<00:01, 76.70train/s]
Processing trains:  99%|█████████▉| 7491/7580 [01:37<00:01, 77.34train/s]
Processing trains:  99%|█████████▉| 7500/7580 [01:37<00:01, 78.45train/s]
Processing trains:  99%|█████████▉| 7508/7580 [01:37<00:00, 77.29train/s]
Processing trains:  99%|█████████▉| 7516/7580 [01:37<00:00, 77.20train/s]
```

```
Processing trains:  99%|██████████| 7524/7580 [01:37<00:00, 77.65train/s]
Processing trains:  99%|██████████| 7532/7580 [01:38<00:00, 77.21train/s]
Processing trains:  99%|██████████| 7540/7580 [01:38<00:00, 77.29train/s]
Processing trains: 100%|██████████| 7548/7580 [01:38<00:00, 76.99train/s]
Processing trains: 100%|██████████| 7556/7580 [01:38<00:00, 77.64train/s]
Processing trains: 100%|██████████| 7564/7580 [01:38<00:00, 78.28train/s]
Processing trains: 100%|██████████| 7580/7580 [01:38<00:00, 76.86train/s]

Graph generation complete.
```



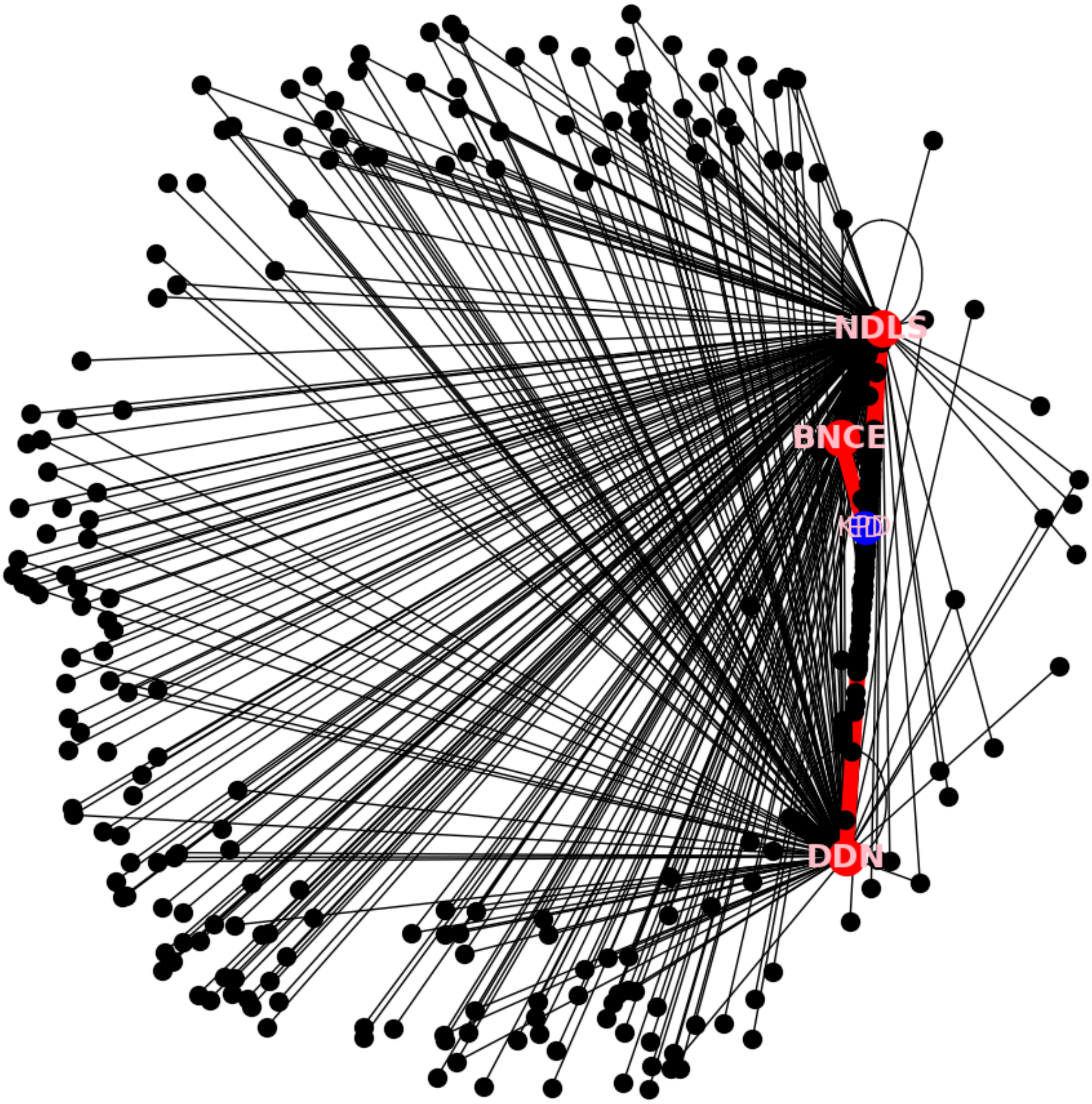
In [ ]:

```
subgraph = get_subgraph(modified_railway_data, railway_network, ['NDLS', 'BNCE', 'DDN'
```

```
Building subgraph:   0%|                                                          | 0/3 [0
0:00<?, ?it/s]
Generating graph from train routes...
Processing trains:   0%|          | 0/7580 [00:00<?, ?train/s]
Processing trains:   0%|          | 9/7580 [00:00<01:31, 82.97train/s]
Processing trains:   0%|          | 18/7580 [00:00<01:33, 81.27train/s]
Processing trains:   0%|          | 27/7580 [00:00<01:32, 81.95train/s]
Processing trains:   0%|          | 36/7580 [00:00<01:32, 81.67train/s]
Processing trains:   1%|          | 45/7580 [00:00<01:32, 81.65train/s]
Processing trains:   1%|          | 54/7580 [00:00<01:32, 81.61train/s]
Processing trains:   1%|          | 63/7580 [00:00<01:32, 81.23train/s]
Processing trains:   1%|          | 72/7580 [00:00<01:31, 81.77train/s]
Processing trains:   1%|          | 81/7580 [00:00<01:32, 81.27train/s]
```

```
Processing trains:  89%|████████▉ | 6781/7580 [01:28<00:09, 82.51train/s]
Processing trains:  90%|████████▉ | 6790/7580 [01:28<00:09, 83.11train/s]
Processing trains:  90%|████████▉ | 6799/7580 [01:28<00:09, 82.66train/s]
Processing trains:  90%|████████▉ | 6808/7580 [01:28<00:09, 81.53train/s]
Processing trains:  90%|████████▉ | 6817/7580 [01:28<00:09, 81.38train/s]
Processing trains:  90%|█████████ | 6826/7580 [01:29<00:09, 81.67train/s]
Processing trains:  90%|█████████ | 6835/7580 [01:29<00:09, 81.57train/s]
Processing trains:  90%|█████████ | 6844/7580 [01:29<00:09, 81.66train/s]
Processing trains:  90%|█████████ | 6853/7580 [01:29<00:08, 81.51train/s]
Processing trains:  91%|█████████ | 6862/7580 [01:29<00:08, 81.64train/s]
Processing trains:  91%|█████████ | 6871/7580 [01:29<00:08, 82.24train/s]
Processing trains:  91%|█████████ | 6880/7580 [01:29<00:08, 81.56train/s]
Processing trains:  91%|█████████ | 6889/7580 [01:29<00:08, 81.87train/s]
Processing trains:  91%|█████████ | 6898/7580 [01:29<00:08, 81.93train/s]
Processing trains:  91%|█████████ | 6907/7580 [01:29<00:08, 82.88train/s]
Processing trains:  91%|█████████ | 6916/7580 [01:30<00:08, 82.62train/s]
Processing trains:  91%|█████████▏| 6925/7580 [01:30<00:07, 82.48train/s]
Processing trains:  91%|█████████▏| 6934/7580 [01:30<00:07, 83.27train/s]
Processing trains:  92%|█████████▏| 6943/7580 [01:30<00:07, 83.00train/s]
Processing trains:  92%|█████████▏| 6952/7580 [01:30<00:07, 81.81train/s]
Processing trains:  92%|█████████▏| 6961/7580 [01:30<00:07, 81.11train/s]
Processing trains:  92%|█████████▏| 6970/7580 [01:30<00:07, 80.70train/s]
Processing trains:  92%|█████████▏| 6979/7580 [01:30<00:07, 80.19train/s]
Processing trains:  92%|█████████▏| 6988/7580 [01:31<00:07, 79.73train/s]
Processing trains:  92%|█████████▏| 6996/7580 [01:31<00:07, 78.32train/s]
Processing trains:  92%|█████████▏| 7004/7580 [01:31<00:07, 77.96train/s]
Processing trains:  93%|█████████▎| 7012/7580 [01:31<00:07, 74.55train/s]
Processing trains:  93%|█████████▎| 7021/7580 [01:31<00:07, 76.89train/s]
Processing trains:  93%|█████████▎| 7029/7580 [01:31<00:07, 77.40train/s]
Processing trains:  93%|█████████▎| 7037/7580 [01:31<00:06, 77.86train/s]
Processing trains:  93%|█████████▎| 7046/7580 [01:31<00:06, 79.23train/s]
Processing trains:  93%|█████████▎| 7054/7580 [01:31<00:06, 79.14train/s]
Processing trains:  93%|█████████▎| 7062/7580 [01:31<00:06, 79.13train/s]
Processing trains:  93%|█████████▎| 7071/7580 [01:32<00:06, 80.26train/s]
Processing trains:  93%|█████████▎| 7080/7580 [01:32<00:06, 81.06train/s]
Processing trains:  94%|█████████▎| 7089/7580 [01:32<00:06, 81.25train/s]
Processing trains:  94%|█████████▎| 7098/7580 [01:32<00:05, 82.19train/s]
Processing trains:  94%|█████████▍| 7107/7580 [01:32<00:05, 82.85train/s]
Processing trains:  94%|█████████▍| 7116/7580 [01:32<00:05, 83.27train/s]
Processing trains:  94%|█████████▍| 7125/7580 [01:32<00:05, 83.72train/s]
Processing trains:  94%|█████████▍| 7134/7580 [01:32<00:05, 83.17train/s]
Processing trains:  94%|█████████▍| 7143/7580 [01:32<00:05, 82.76train/s]
Processing trains:  94%|█████████▍| 7152/7580 [01:33<00:05, 82.14train/s]
Processing trains:  94%|█████████▍| 7161/7580 [01:33<00:05, 82.06train/s]
Processing trains:  95%|█████████▍| 7170/7580 [01:33<00:05, 81.29train/s]
Processing trains:  95%|█████████▍| 7179/7580 [01:33<00:05, 78.83train/s]
Processing trains:  95%|█████████▍| 7187/7580 [01:33<00:05, 77.54train/s]
Processing trains:  95%|█████████▍| 7195/7580 [01:33<00:05, 76.51train/s]
Processing trains:  95%|█████████▌| 7203/7580 [01:33<00:04, 77.14train/s]
Processing trains:  95%|█████████▌| 7211/7580 [01:33<00:04, 77.53train/s]
Processing trains:  95%|█████████▌| 7219/7580 [01:33<00:04, 76.79train/s]
Processing trains:  95%|█████████▌| 7227/7580 [01:34<00:04, 75.69train/s]
Processing trains:  95%|█████████▌| 7235/7580 [01:34<00:04, 70.76train/s]
Processing trains:  96%|█████████▌| 7243/7580 [01:34<00:04, 72.74train/s]
Processing trains:  96%|█████████▌| 7251/7580 [01:34<00:04, 74.28train/s]
Processing trains:  96%|█████████▌| 7259/7580 [01:34<00:04, 75.56train/s]
Processing trains:  96%|█████████▌| 7267/7580 [01:34<00:04, 76.80train/s]
Processing trains:  96%|█████████▌| 7276/7580 [01:34<00:03, 78.78train/s]
Processing trains:  96%|█████████▌| 7285/7580 [01:34<00:03, 79.54train/s]
Processing trains:  96%|█████████▌| 7293/7580 [01:34<00:03, 76.70train/s]
Processing trains:  96%|█████████▋| 7301/7580 [01:35<00:03, 73.23train/s]
Processing trains:  96%|█████████▋| 7309/7580 [01:35<00:03, 73.95train/s]
Processing trains:  97%|█████████▋| 7318/7580 [01:35<00:03, 76.78train/s]
Processing trains:  97%|█████████▋| 7327/7580 [01:35<00:03, 78.19train/s]
Processing trains:  97%|█████████▋| 7336/7580 [01:35<00:03, 78.79train/s]
Processing trains:  97%|█████████▋| 7344/7580 [01:35<00:03, 78.14train/s]
Processing trains:  97%|█████████▋| 7353/7580 [01:35<00:02, 79.02train/s]
Processing trains:  97%|█████████▋| 7361/7580 [01:35<00:02, 78.54train/s]
Processing trains:  97%|█████████▋| 7369/7580 [01:35<00:02, 77.38train/s]
Processing trains:  97%|█████████▋| 7377/7580 [01:35<00:02, 76.99train/s]
Processing trains:  97%|█████████▋| 7385/7580 [01:36<00:02, 77.21train/s]
Processing trains:  98%|█████████▊| 7393/7580 [01:36<00:02, 77.62train/s]
Processing trains:  98%|█████████▊| 7401/7580 [01:36<00:02, 77.83train/s]
Processing trains:  98%|█████████▊| 7409/7580 [01:36<00:02, 78.35train/s]
```

```
Processing trains:  98%|█████████▊| 7417/7580 [01:36<00:02, 75.25train/s]
Processing trains:  98%|█████████▊| 7425/7580 [01:36<00:02, 71.85train/s]
Processing trains:  98%|█████████▊| 7433/7580 [01:36<00:02, 71.45train/s]
Processing trains:  98%|█████████▊| 7441/7580 [01:36<00:01, 72.70train/s]
Processing trains:  98%|█████████▊| 7449/7580 [01:36<00:01, 69.09train/s]
Processing trains:  98%|█████████▊| 7457/7580 [01:37<00:01, 70.95train/s]
Processing trains:  98%|█████████▊| 7465/7580 [01:37<00:01, 70.92train/s]
Processing trains:  99%|█████████▊| 7473/7580 [01:37<00:01, 72.11train/s]
Processing trains:  99%|█████████▊| 7482/7580 [01:37<00:01, 75.27train/s]
Processing trains:  99%|█████████▉| 7490/7580 [01:37<00:01, 74.64train/s]
Processing trains:  99%|█████████▉| 7498/7580 [01:37<00:01, 76.10train/s]
Processing trains:  99%|█████████▉| 7506/7580 [01:37<00:00, 75.86train/s]
Processing trains:  99%|█████████▉| 7514/7580 [01:37<00:00, 76.27train/s]
Processing trains:  99%|█████████▉| 7522/7580 [01:37<00:00, 76.21train/s]
Processing trains:  99%|█████████▉| 7530/7580 [01:38<00:00, 76.37train/s]
Processing trains:  99%|█████████▉| 7538/7580 [01:38<00:00, 77.24train/s]
Processing trains: 100%|█████████▉| 7546/7580 [01:38<00:00, 77.04train/s]
Processing trains: 100%|█████████▉| 7554/7580 [01:38<00:00, 77.72train/s]
Processing trains: 100%|█████████▉| 7562/7580 [01:38<00:00, 77.84train/s]
Processing trains: 100%|█████████▉| 7570/7580 [01:38<00:00, 77.75train/s]
Processing trains: 100%|██████████| 7580/7580 [01:38<00:00, 76.82train/s]
Building subgraph:  33%|███████████████                               | 1/3 [01:38<03:1
7, 98.79s/it]
Graph generation complete.
Generating graph from train routes...
Processing trains:   0%|          | 0/7580 [00:00<?, ?train/s]
Processing trains:   0%|          | 8/7580 [00:00<01:35, 79.36train/s]
Processing trains:   0%|          | 17/7580 [00:00<01:33, 80.55train/s]
Processing trains:   0%|          | 26/7580 [00:00<01:34, 80.21train/s]
Processing trains:   0%|          | 35/7580 [00:00<01:33, 80.54train/s]
Processing trains:   1%|          | 44/7580 [00:00<01:32, 81.37train/s]
Processing trains:   1%|          | 53/7580 [00:00<01:32, 81.49train/s]
Processing trains:   1%|          | 62/7580 [00:00<01:32, 81.35train/s]
Processing trains:   1%|          | 71/7580 [00:00<01:32, 81.47train/s]
Processing trains:   1%|          | 80/7580 [00:00<01:33, 79.91train/s]
Processing trains:   1%|          | 88/7580 [00:01<01:35, 78.63train/s]
Processing trains:   1%|▏         | 96/7580 [00:01<01:34, 78.86train/s]
Processing trains:   1%|▏         | 105/7580 [00:01<01:33, 80.29train/s]
Processing trains:   2%|▏         | 114/7580 [00:01<01:32, 80.91train/s]
Processing trains:   2%|▏         | 123/7580 [00:01<01:31, 81.54train/s]
Processing trains:   2%|▏         | 132/7580 [00:01<01:30, 82.00train/s]
Processing trains:   2%|▏         | 141/7580 [00:01<01:31, 81.62train/s]
Processing trains:   2%|▏         | 150/7580 [00:01<01:30, 81.72train/s]
Processing trains:   2%|▏         | 159/7580 [00:01<01:30, 81.55train/s]
Processing trains:   2%|▏         | 168/7580 [00:02<01:31, 81.14train/s]
Processing trains:   2%|▏         | 177/7580 [00:02<01:31, 80.54train/s]
Processing trains:   2%|▏         | 186/7580 [00:02<01:31, 80.81train/s]
Processing trains:   3%|▎         | 195/7580 [00:02<01:31, 80.69train/s]
Processing trains:   3%|▎         | 204/7580 [00:02<01:31, 80.62train/s]
Processing trains:   3%|▎         | 213/7580 [00:02<01:31, 80.59train/s]
Processing trains:   3%|▎         | 222/7580 [00:02<01:30, 81.21train/s]
Processing trains:   3%|▎         | 231/7580 [00:02<01:30, 81.05train/s]
Processing trains:   3%|▎         | 240/7580 [00:02<01:30, 81.13train/s]
Processing trains:   3%|▎         | 249/7580 [00:03<01:30, 81.00train/s]
Processing trains:   3%|▎         | 258/7580 [00:03<01:31, 80.26train/s]
Processing trains:   4%|▎         | 267/7580 [00:03<01:31, 79.92train/s]
Processing trains:   4%|▎         | 276/7580 [00:03<01:31, 79.97train/s]
Processing trains:   4%|▍         | 285/7580 [00:03<01:30, 80.68train/s]
Processing trains:   4%|▍         | 294/7580 [00:03<01:32, 78.78train/s]
Processing trains:   4%|▍         | 302/7580 [00:03<01:33, 78.24train/s]
Processing trains:   4%|▍         | 311/7580 [00:03<01:31, 79.41train/s]
Processing trains:   4%|▍         | 320/7580 [00:03<01:30, 80.21train/s]
Processing trains:   4%|▍         | 329/7580 [00:04<01:30, 79.89train/s]
Processing trains:   4%|▍         | 337/7580 [00:04<01:31, 79.14train/s]
Processing trains:   5%|▍         | 346/7580 [00:04<01:30, 79.57train/s]
Processing trains:   5%|▍         | 354/7580 [00:04<01:31, 79.20train/s]
Processing trains:   5%|▍         | 363/7580 [00:04<01:30, 79.62train/s]
Processing trains:   5%|▍         | 371/7580 [00:04<01:31, 78.72train/s]
Processing trains:   5%|▌         | 380/7580 [00:04<01:30, 79.90train/s]
Processing trains:   5%|▌         | 388/7580 [00:04<01:30, 79.74train/s]
Processing trains:   5%|▌         | 396/7580 [00:04<01:30, 79.57train/s]
Processing trains:   5%|▌         | 404/7580 [00:05<01:31, 78.07train/s]
Processing trains:   5%|▌         | 412/7580 [00:05<01:32, 77.46train/s]
Processing trains:   6%|▌         | 420/7580 [00:05<01:36, 74.05train/s]
```



In [ ]:

```
import operator
def compute_centrality(Railway_Network, description, railway_data):
  centrality = {}
  if description == "Degree":
    centrality = nx.degree_centrality(Railway_Network)
  elif description == "Betweenness":
    centrality = nx.betweenness_centrality(Railway_Network)
  elif description == "Closeness":
    centrality = nx.closeness_centrality(Railway_Network)
  elif description == "Eigen Vector":
    centrality = nx.eigenvector_centrality_numpy(Railway_Network)
  else:
    print("Incorrect input centrality measure")

  centrality = sorted(centrality.items(), key=operator.itemgetter(1), reverse=True)[:1
  stations = []
  for item in centrality:
    station_code = item[0]
    stations.append((railway_data.loc[railway_data['Station Code'] == station_code]['S

  return stations
```

In [ ]:

```
DegreeCentrality_stations = compute_centrality(railway_network, "Degree", railway_data)
print("Top Stations in the European Railway System acc to the Degree Centrality:\n\n \
for item in DegreeCentrality_stations:
  print("\t",item[0],"\t\t",item[1])
```

```
Top Stations in the European Railway System acc to the Degree Centrality:

        STATION NAME                 BETWEENNESS CENTRALITY
         HOWRAH JN.                  0.30211146575006137
         VIJAYWADA JN                0.2704394794991407
         KANPUR CENTR                0.263810459121041
         VARANASI JN.                0.25767247728946724
         GHAZIABAD JN                0.25202553400441935
         KALYAN JN                   0.24797446599558065
         ITARSI                      0.24441443653326786
         LUCKNOW JN.                 0.243677878713479
         AHMEDABAD                   0.23852197397495703
         MATHURA JN.                 0.2363123005155905
```

In [ ]:

```
BetweennessCentrality_stations = compute_centrality(railway_network, "Betweenness", rai
print("Top stations in the Indian Railway System acc to the Betweenness Centrality:\n\n
for item in BetweennessCentrality_stations:
  print("\t",item[0],"\t\t",item[1])
```

```
Top stations in the Indian Railway System acc to the Betweenness Centrality:

        STATION NAME                 BETWEENNESS CENTRALITY
         HOWRAH JN.                  0.03406351792928565
         SEALDAH                     0.021795409939440933
         KANPUR CENTR                0.020930718809083173
         VIJAYWADA JN                0.015177400281457004
         AHMEDABAD                   0.014301664879802252
         YESVANTPUR J                0.014256569821548529
         VADODARA JN.                0.012967968029171826
         VARANASI JN.                0.012623744668081234
         KOLKATA                     0.011898350401287686
         PILIBHIT JN.                0.01184539309233258
```

In [ ]:

```
ClosenessCentrality_stations = compute_centrality(railway_network, "Closeness", railway
print("Top stations in the Indian Railway System acc to the Closeness Centrality:\n\n \
for item in ClosenessCentrality_stations:
  print("\t",item[0],"\t\t",item[1])
```

```
Top stations in the Indian Railway System acc to the Closeness Centrality:

        STATION NAME                 BETWEENNESS CENTRALITY
         HOWRAH JN.                  0.5124082033874587
         AHMEDABAD                   0.5107660799308908
         VADODARA JN.                0.5063522924820026
         KANPUR CENTR                0.5062879529276847
         VARANASI JN.                0.5052287059583946
         NEW DELHI                   0.5051966767517281
         KALYAN JN                   0.5036640360941574
         MUGHAL SARAI                0.5035049206471066
         VIJAYWADA JN                0.5021406666088064
         ITARSI                      0.4998101090743702
```

In [ ]:

```
EigenVectorCentrality_stations = compute_centrality(railway_network, "Eigen Vector", ra
print("Top stations in the Indian Railway System acc to the Eigen Vector Centrality:\n\
for item in EigenVectorCentrality_stations:
  print("\t",item[0],"\t\t",item[1])
```

```
Top stations in the Indian Railway System acc to the Eigen Vector Centrality:

        STATION NAME                 BETWEENNESS CENTRALITY
         VARANASI JN.                0.07072646825994844
         HOWRAH JN.                  0.0696564673531263
         KANPUR CENTR                0.06676069453032213
         ITARSI                      0.06674728443587287
```

```
NEW DELHI              0.06624320025503437
LUDHIANA JN.           0.06593208791866617
MATHURA JN.            0.06569071466478023
KALYAN JN              0.06527068019550918
PATNA JN.              0.06473312502358503
MUGHAL SARAI           0.06457963445769374
```

In [ ]:

```
print('Number of trains:', len(np.unique(modified_railway_data['Train Name'].astype('s
print('Number of stations:', len(np.unique(modified_railway_data['Station Code'].astype
```

```
Number of trains: 7580
Number of stations: 8147
```

In [ ]:

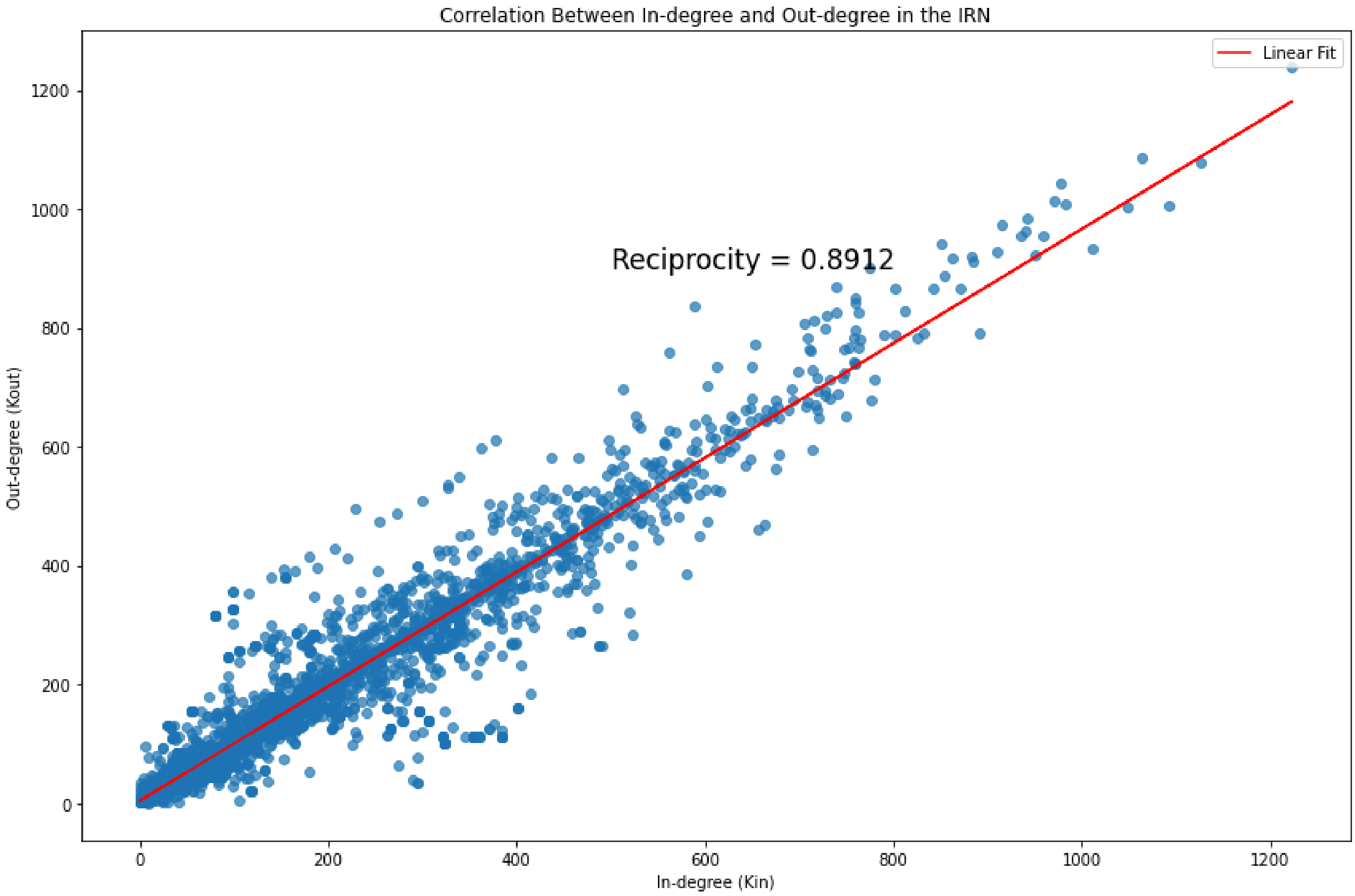
```
distances = modified_railway_data['Distance'].astype('int')
longest_route_df = modified_railway_data[modified_railway_data['Distance']==str(distanc
distances = distances.replace(0, distances.max())
shortest_route_df = modified_railway_data[modified_railway_data['Distance']==str(distar
print('Longest train route:', distances.max(), 'km. Train = ', longest_route_df['Train
print('Shortest train route:', distances.min(), 'km. Train = ', shortest_route_df['Tra

trains = np.unique(modified_railway_data['Train Name'].astype('str'))
max_distance_between_stations = 0
min_distance_between_stations = distances.max()
min_train_name = ''
max_train_name = ''
average_train_route_distance = 0
average_distance_between_stops = 0
# iterate over all trains
for train_name in trains:
  train_route = modified_railway_data.loc[modified_railway_data['Train Name'] == train_
  station_distances = train_route['Distance'].to_list()
  for station_itr in range(len(station_distances)-1):
    distance = int(station_distances[station_itr+1]) - int(station_distances[station_i
    if distance < min_distance_between_stations and distance > 0:
      min_distance_between_stations = distance
      min_train_name = train_name
    if distance > max_distance_between_stations:
      max_distance_between_stations = distance
      max_train_name = train_name
  average_train_route_distance+=int(station_distances[-1])

average_distance_between_stops = average_train_route_distance
average_train_route_distance/=len(trains)
average_distance_between_stops/=modified_railway_data['Station Code'].shape[0]
print("Maximum distance between any two consecutive stations:", max_distance_between_s
print("Minimum distance between any two consecutive stations:", min_distance_between_s
print("Average total train route distance:", round(average_train_route_distance, 2), 'l
print("Average distance between consecutive stops", round(average_distance_between_stop
```

```
Longest train route: 4260 km. Train =  15905 CAPE - DBRG . Starting station:  KANNIYAKU
MARI . Ending Station:  DIBRUGARH
Shortest train route: 1 km. Train =  3308 PLJE-SZE MEM . Starting station:  PHULWARTANR
. Ending Station:  SONARDIH
Maximum distance between any two consecutive stations: 1301  km with train RSD-PJP BSF
Minimum distance between any two consecutive stations: 1  km with train  LGL KNJ EMU
Average total train route distance: 439.7 km
Average distance between consecutive stops 17.91 km
```

In [ ]:

```
# import collections
# indegree_sequence = [d for n, d in railway_network.in_degree()]
# #print("Indegree Distribution", Railway_Network.in_degree())
# indegreeCount = collections.Counter(indegree_sequence)


# outdegree_sequence = [d for n, d in railway_network.out_degree()]
# #print("Outdegree Distribution", Railway_Network.out_degree())
# outdegreeCount = collections.Counter(outdegree_sequence)
```

```
# plt.figure(figsize=(11, 6.5))
# plt.scatter(indegree_sequence, outdegree_sequence)
# plt.xlabel("Kin")
# plt.ylabel("Kout")
# plt.text(500, 900, "R =" + str(round(nx.reciprocity(railway_network), 4)), fontsize=1

# m, b = np.polyfit(np.array(indegree_sequence), np.array(outdegree_sequence), 1)
# plt.plot(np.array(indegree_sequence), m*np.array(indegree_sequence) + b, color = 'r',
# plt.title("The correlation between Kin and Kout in the IRN")

# plt.show()

import matplotlib.pyplot as plt
import collections
import numpy as np
import networkx as nx

# Extract in-degree and out-degree for all nodes
in_degrees = [deg for _, deg in railway_network.in_degree()]
out_degrees = [deg for _, deg in railway_network.out_degree()]

# Count frequency of each degree value
in_deg_freq = collections.Counter(in_degrees)
out_deg_freq = collections.Counter(out_degrees)

# Create a scatter plot to visualize the relationship
plt.figure(figsize=(12, 8))
plt.scatter(in_degrees, out_degrees, alpha=0.7)

# Label axes
plt.xlabel("In-degree (Kin)")
plt.ylabel("Out-degree (Kout)")

# Calculate and display reciprocity of the graph
reciprocal_ratio = round(nx.reciprocity(railway_network), 4)
plt.text(500, 900, f"Reciprocity = {reciprocal_ratio}", fontsize=17)

# Fit and plot a linear regression line for Kin vs Kout
slope, intercept = np.polyfit(in_degrees, out_degrees, 1)
regression_line = slope * np.array(in_degrees) + intercept
plt.plot(in_degrees, regression_line, color='red', label='Linear Fit')

plt.title("Correlation Between In-degree and Out-degree in the IRN")
plt.legend()
plt.tight_layout()
plt.show()
```



In [ ]:
```
G_Undirected = railway_network.to_undirected()
degrees = [d for n, d in G_Undirected.degree()]

## 1. Basic Statistics
print("=== Degree Analysis ===")
print(f"Number of nodes: {G_Undirected.number_of_nodes()}")
print(f"Number of edges: {G_Undirected.number_of_edges()}")
print(f"Average degree: {np.mean(degrees):.2f}")
print(f"Median degree: {np.median(degrees):.2f}")
print(f"Maximum degree: {max(degrees)}")
print(f"Minimum degree: {min(degrees)}")
print(f"Density: {nx.density(G_Undirected):.4f}")
```

```
=== Degree Analysis ===
Number of nodes: 8147
Number of edges: 500412
Average degree: 122.85
Median degree: 68.00
Maximum degree: 1284
Minimum degree: 1
Density: 0.0151
```

In [ ]:
```
print('Num connected components in the undirected graph is:', nx.number_connected_compo
```

```
Num connected components in the undirected graph is: 7
```

In [ ]:
```
def Compute_Network_Degree(indian_undirected):

    node_degree_values = indian_undirected.degree()
    weighted_node_degree_values = indian_undirected.degree(weight='weight')

    degree_values = [val for (node, val) in node_degree_values]
    weighted_degree_values = [val for (node, val) in weighted_node_degree_values]

    average_degree = np.sum(degree_values)/ len(degree_values)
    weighted_average_degree = np.sum(weighted_degree_values)/ len(weighted_degree_valu

    return average_degree, weighted_average_degree, node_degree_values, weighted_node_
```

```
average_degree, weighted_average_degree, node_degree_values, weighted_node_degree_value

print("The Average Degree of the Indian Railway Network is: ", average_degree)
print("\nThe Weighted Average Degree of the Indian Railway Network is: ", weighted_ave
```

```
The Average Degree of the Indian Railway Network is:  122.84571007732907

The Weighted Average Degree of the Indian Railway Network is:  1513.2014238369952
```

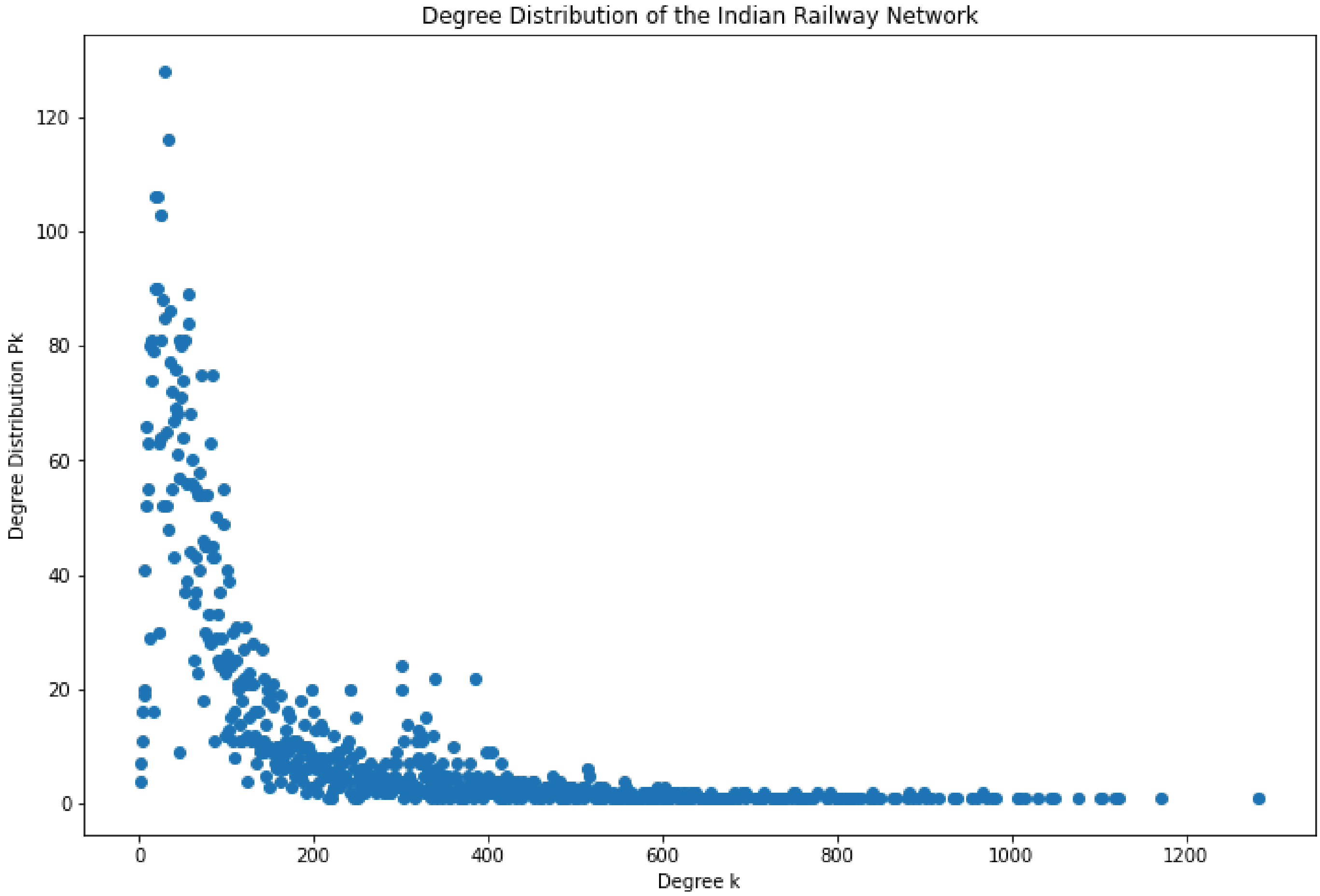
In [ ]:

```
print("Top 5 stations with the highest number of direct connections in the Indian Rail
top_degrees = sorted(G_Undirected.degree, key=lambda x: x[1], reverse=True)[:5]
top_degree_nodes = [node for node, _ in top_degrees]
top_degree_counts = [deg for _, deg in top_degrees]

for i in range(len(top_degree_nodes)):
    node = top_degree_nodes[i]
    station_name = modified_railway_data.loc[modified_railway_data['Station Code'] == n
    print(f"{station_name}: {top_degree_counts[i]} connections")

print("\nTop 5 stations with the highest weighted connectivity (based on cumulative di
top_weighted = sorted(G_Undirected.degree(weight='weight'), key=lambda x: x[1], revers
top_weighted_nodes = [node for node, _ in top_weighted]
top_weighted_counts = [weight for _, weight in top_weighted]

for i in range(len(top_weighted_nodes)):
    node = top_weighted_nodes[i]
    station_name = modified_railway_data.loc[modified_railway_data['Station Code'] == n
    print(f"{station_name}: weighted degree = {top_weighted_counts[i]}")
```

```
Top 5 stations with the highest number of direct connections in the Indian Railway Netw
ork:
HOWRAH JN.: 1284 connections
VIJAYWADA JN: 1171 connections
VARANASI JN.: 1124 connections
KANPUR CENTR: 1119 connections
LUCKNOW JN.: 1104 connections

Top 5 stations with the highest weighted connectivity (based on cumulative distance or
train traffic):
CHENNAI BEAC: weighted degree = 289635
TAMBARAM: weighted degree = 231587
PALLAVARAM: weighted degree = 195255
ST. THOMAS M: weighted degree = 194981
CHENNAI EGMO: weighted degree = 194449
```

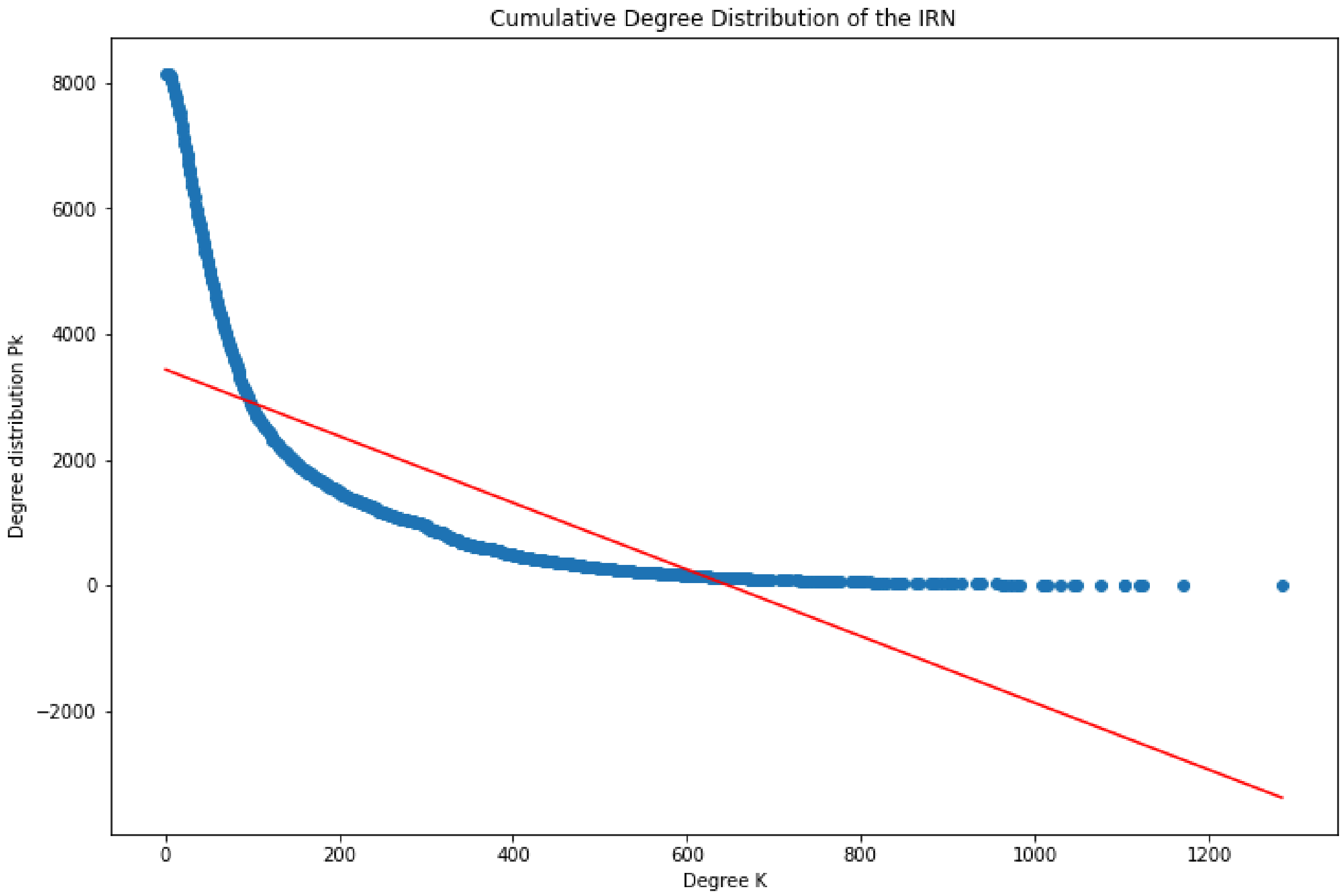
In [ ]:

```
def Compute_Degree_Distribution(indian_undirected):

    degree_sequence = sorted([d for n, d in G_Undirected.degree()])
    degreeCount = collections.Counter(degree_sequence)

    degree, count = zip(*degreeCount.items())

    plt.figure(figsize=(12, 8))
    plt.scatter(degree, count)

    plt.xlabel("Degree k")
    plt.ylabel("Degree Distribution Pk")

    plt.title("Degree Distribution of the Indian Railway Network")

    plt.show()
Compute_Degree_Distribution(G_Undirected)
```



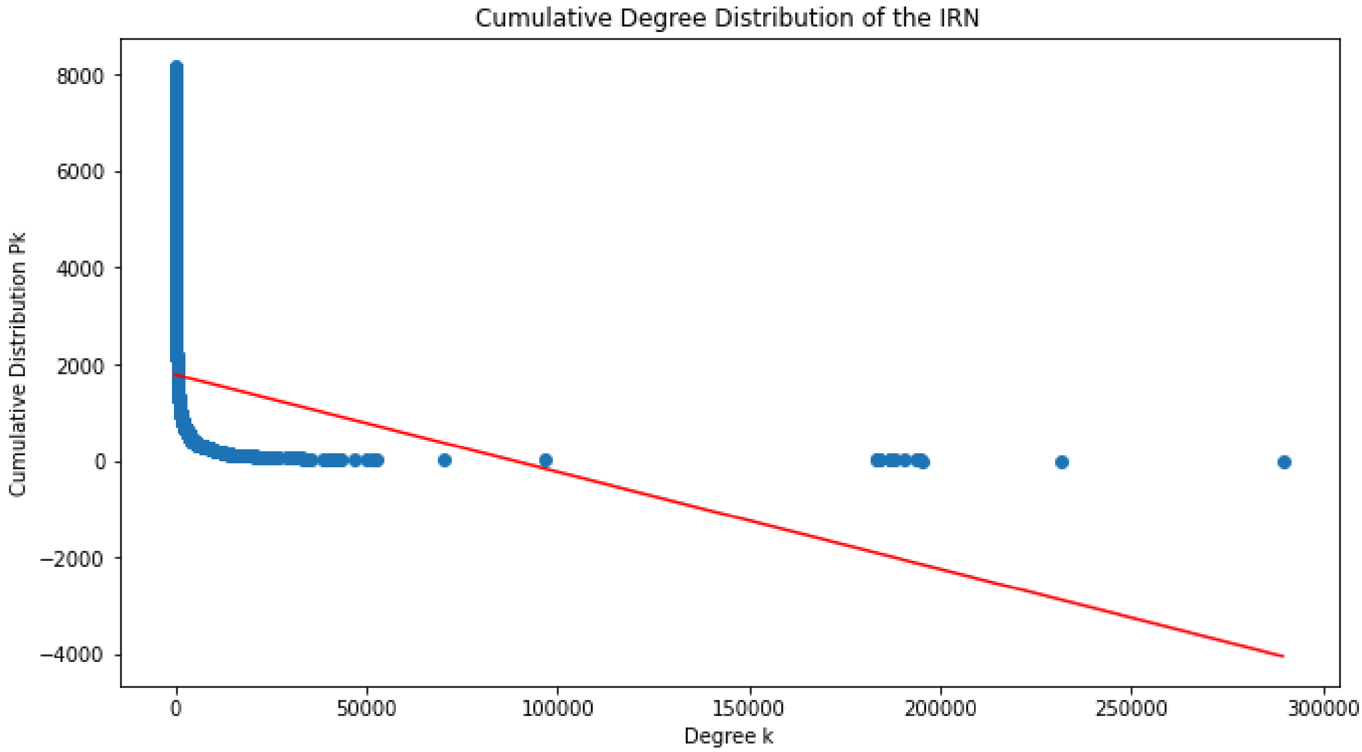
In [ ]:

```
def Compute_Cumulative_Degree_Distribution(indian_undirected):

    degree_sequence = sorted([d for n, d in indian_undirected.degree()])
    degreeCount = collections.Counter(degree_sequence)

    degree, count = zip(*degreeCount.items())

    cumulative_count = np.cumsum(count[::-1])[::-1]

    plt.figure(figsize=(12, 8))
    plt.scatter(degree, cumulative_count)

    plt.xlabel("Degree K")
    plt.ylabel("Degree distribution Pk")

    m, b = np.polyfit(np.array(degree), np.array(cumulative_count), 1)
    plt.plot(np.array(degree), m*np.array(degree) + b, color = 'r')

    plt.title("Cumulative Degree Distribution of the IRN")

    plt.show()
Compute_Cumulative_Degree_Distribution(G_Undirected)
```



In [ ]:

```
def Compute_Cumulative_Strength_Distribution(indian_undirected):

    degree_sequence = sorted([d for n, d in indian_undirected.degree(weight = 'weight')
    degreeCount = collections.Counter(degree_sequence)

    degree, count = zip(*degreeCount.items())

    cumulative_count = np.cumsum(count[::-1])[::-1]

    plt.figure(figsize=(11, 6))
    plt.scatter(degree, cumulative_count)

    plt.xlabel("Degree k")
    plt.ylabel("Cumulative Distribution Pk")

    m, b = np.polyfit(np.array(degree), np.array(cumulative_count), 1)
    plt.plot(np.array(degree), m*np.array(degree) + b, color = 'r')

    plt.title("Cumulative Degree Distribution of the IRN")

    plt.show()

Compute_Cumulative_Strength_Distribution(G_Undirected)
Clustering_Coefficient = nx.average_clustering(G_Undirected)
print("The Clustering Coefficent of the Graph: ", Clustering_Coefficient)
```




```
The Clustering Coefficent of the Graph:  0.7635363552259165
```

In [ ]:

```python
def Compute_Clustering_Coefficient(indian_undirected):

    node_clustering_values = nx.clustering(indian_undirected)

    node_degree_values = indian_undirected.degree()
    unique_degrees = list(set([y for (x,y) in node_degree_values]))

    Degree_Clustering = {}

    for degree in unique_degrees:
        nodes_kdegree = [x for (x, y) in node_degree_values if y == degree]
        count_nodes_kdegree = len(nodes_kdegree)

        clustering_sum  = 0
        for node in nodes_kdegree:
            clustering_sum = clustering_sum + node_clustering_values[node]

        average_clustering = clustering_sum / count_nodes_kdegree
        Degree_Clustering[degree] = average_clustering

    return Degree_Clustering
Degree_Clustering = Compute_Clustering_Coefficient(G_Undirected)
Degree_Clustering
```

Out[ ]:

```
{1: 0.0,
 2: 0.75,
 3: 0.6363636363636364,
 4: 0.9895833333333334,
 5: 0.9842105263157894,
 6: 0.9916666666666666,
 7: 1.0,
 8: 0.9958791208791209,
 9: 1.0,
 10: 1.0,
 11: 0.9810405643738976,
 12: 0.9791013584117031,
 13: 0.9877272727272729,
 14: 0.9924129924129924,
 15: 0.9967304300637634,
 16: 1.0,
 17: 0.9909318157642804,
 18: 0.9873504747811074,
 19: 0.975797882505829,
```

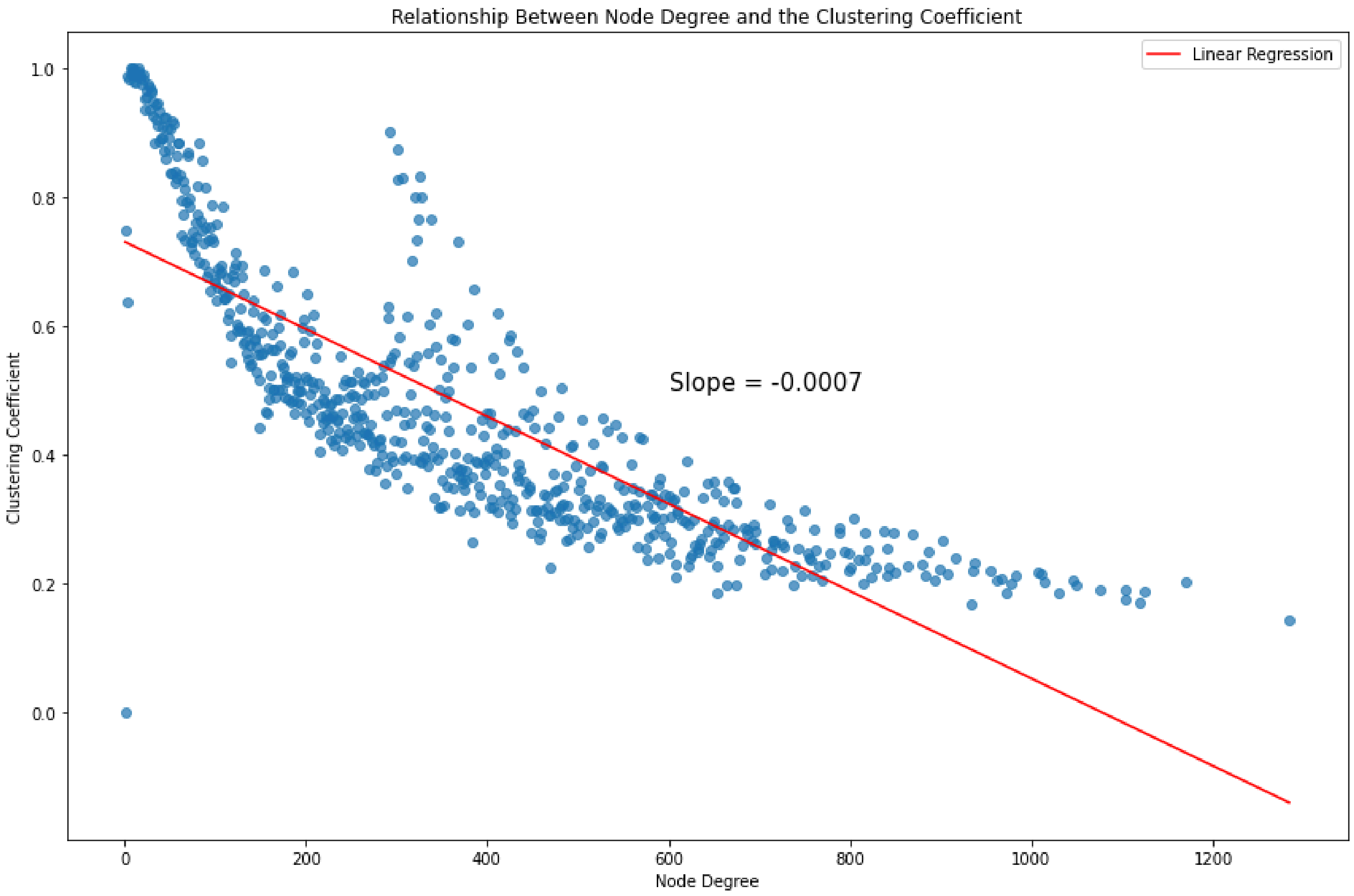
```
20: 0.9835802973953567,
21: 0.9921841641139886,
22: 0.9551627987718214,
23: 0.9365167199949808,
24: 0.9556817310440497,
25: 0.9666800477602108,
26: 0.9756382115141087,
27: 0.9712095312095311,
28: 0.9360834037757115,
29: 0.9644335699023197,
30: 0.9663764662953304,
31: 0.9491264204834836,
32: 0.9277346624454519,
33: 0.8840371965623985,
34: 0.9454532938162624,
35: 0.9207522755402372,
36: 0.9133018496654859,
37: 0.9474352153273724,
38: 0.9350937950937945,
39: 0.8861950642856395,
40: 0.8923798503924192,
41: 0.892630810923494,
42: 0.9090065101778838,
43: 0.9248506492784296,
44: 0.872261162986733,
45: 0.9249744895739729,
46: 0.8604352705801981,
47: 0.9078610077928446,
48: 0.8747865273923325,
49: 0.8917594262105831,
50: 0.9080822568389063,
51: 0.8389819711668448,
52: 0.9206276850721298,
53: 0.8389225646717322,
54: 0.9137909283189455,
55: 0.839863883495959,
56: 0.8224642944656421,
57: 0.8655917827709668,
58: 0.8308261927260442,
59: 0.8860281893064219,
60: 0.8853018724485088,
61: 0.834315438835181,
62: 0.7956630554058158,
63: 0.7418468434438817,
64: 0.8249225289343413,
65: 0.7746792077437237,
66: 0.8125170662670663,
67: 0.7347426367705406,
68: 0.7939610076133339,
69: 0.8645420884417716,
70: 0.8712456890832149,
71: 0.7852619965316195,
72: 0.7978045046086142,
73: 0.7212277384925143,
74: 0.7309976501260297,
75: 0.7469147503668051,
76: 0.7321669181798959,
77: 0.7116899182162341,
78: 0.7606072355195161,
79: 0.7382512155766768,
80: 0.7739505571784052,
81: 0.8170559252046593,
82: 0.8842730883049684,
83: 0.7000307716021961,
84: 0.7628081906496377,
85: 0.8582140600387476,
86: 0.7487371576830929,
87: 0.6971728226174192,
88: 0.7293215417536848,
89: 0.8162587662634576,
90: 0.752613660823657,
91: 0.6765305363875334,
92: 0.734159662542662,
93: 0.6855035359944196,
```

```
750: 0.3125729298656301,
753: 0.24680692410119842,
755: 0.23897318527309203,
757: 0.2369771813023697,
759: 0.2112488030585785,
760: 0.28489419769050867,
762: 0.2520837667290757,
765: 0.22780294665001738,
769: 0.20388002491821583,
773: 0.22932942543837487,
777: 0.24624989580728515,
788: 0.27789825124390205,
789: 0.28596047088340754,
793: 0.24841172046280147,
798: 0.2206251382699662,
799: 0.24645183256306627,
801: 0.22451937101828412,
804: 0.300624842388411,
810: 0.23707166255659023,
814: 0.2008285094725845,
816: 0.23760694609403096,
817: 0.27793069142762394,
820: 0.2517649100860983,
824: 0.20849595917387556,
828: 0.22323868222173307,
837: 0.2813883025316274,
840: 0.21183451524509342,
841: 0.25346830099476303,
843: 0.22383217258365892,
848: 0.2798047197392533,
849: 0.21786530684016178,
863: 0.2272155146800637 3,
868: 0.27702145269593775,
880: 0.22965535333490908,
883: 0.21173769476834176,
886: 0.249048133932552,
893: 0.20581848447016987,
899: 0.2223781653129479,
902: 0.26547521937955754,
908: 0.21451587330625785,
916: 0.23868459715128668,
933: 0.16870748299319727,
935: 0.2183482144910736,
938: 0.2326111796700032,
955: 0.2185358928461206,
961: 0.2036023955015772,
967: 0.20794401565153828,
972: 0.18595640100858574,
978: 0.19914039512400167,
982: 0.21175918784265493,
1008: 0.21729721175434952,
1010: 0.213438470389811,
1015: 0.2021721572131461,
1030: 0.18421680767146956,
1045: 0.20482036352394078,
1049: 0.19799993060387414,
1076: 0.19065916234091923,
1103: 0.17507389976054827,
1104: 0.1906516267178328,
1119: 0.17065680923364235,
1124: 0.18731524724073395,
1171: 0.20270369241946634,
1284: 0.142069195648388}
```

In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np

# Prepare data for plotting: degree vs clustering coefficient
degree_values = list(Degree_Clustering.keys())
clustering_scores = list(Degree_Clustering.values())

# Initialize figure
plt.figure(figsize=(12, 8))
```

```
plt.scatter(degree_values, clustering_scores, alpha=0.7)

# Axis labels
plt.xlabel("Node Degree")
plt.ylabel("Clustering Coefficient")

# Fit and overlay a linear trend line
slope, intercept = np.polyfit(degree_values, clustering_scores, 1)
trend_line = slope * np.array(degree_values) + intercept
plt.plot(degree_values, trend_line, color='red', label='Linear Regression')

# Display the slope on the plot
plt.text(600, 0.5, f"Slope = {round(slope, 4)}", fontsize=15)

# Plot title
plt.title("Relationship Between Node Degree and the Clustering Coefficient")
plt.legend()
plt.tight_layout()
plt.show()
```



In [ ]:

```
def analyze_path_length_of_IRN(indian_undirected):

    shortest_path_lengths = list(nx.shortest_path_length(indian_undirected))
    Path_Lengths = {}

    for node_path_lengths in tqdm(shortest_path_lengths):

        source_station = node_path_lengths[0]
        destination_stations = node_path_lengths[1]

        for station in destination_stations:
            path = (station, source_station)
            if(path not in Path_Lengths):
                Path_Lengths[(source_station, station)] =  destination_stations[statio

    return Path_Lengths
Path_length = analyze_path_length_of_IRN(G_Undirected)
# print(Path_length)
```

```
100%|██████████| 8147/8147 [00:58<00:00, 138.73it/s]
```

In [ ]:

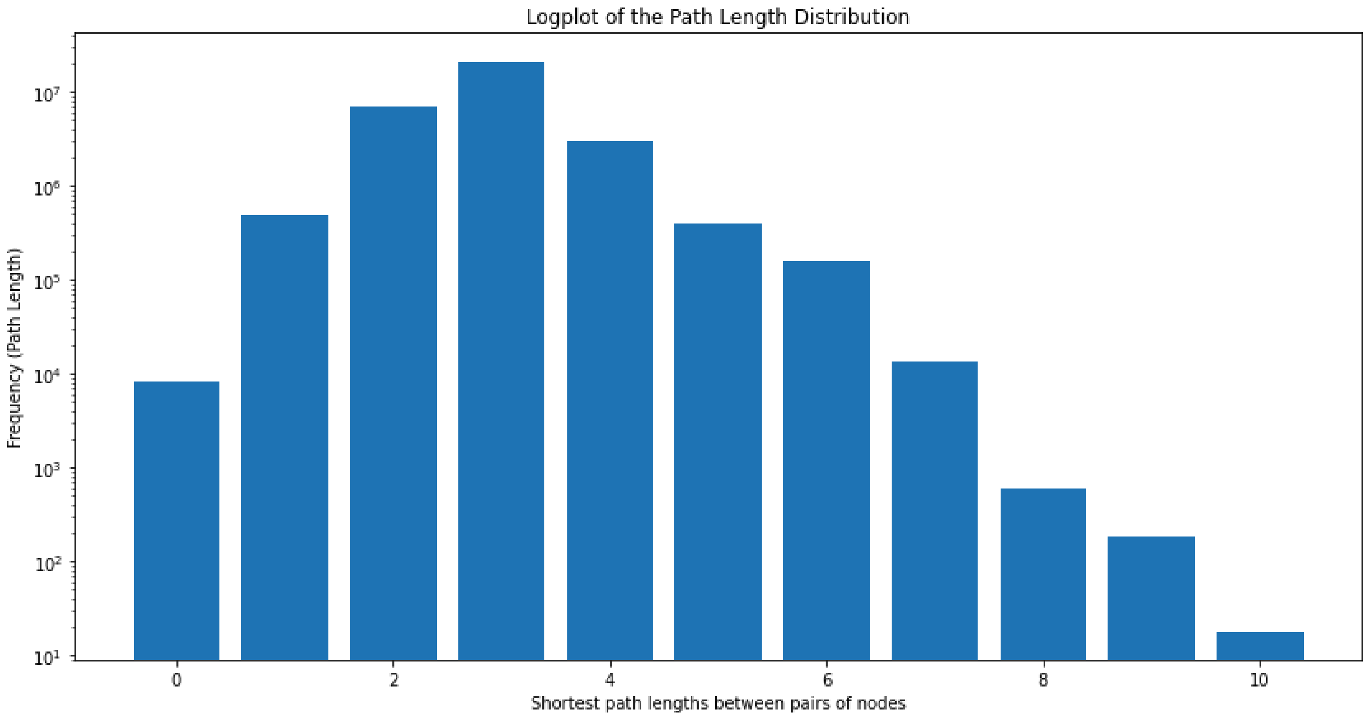
```python
def analyze_path_length_distribution_of_IRN(Path_Lengths):

    path_length_sequence = Path_Lengths.values()
    path_lengthCount = collections.Counter(path_length_sequence)

    path_length, count = zip(*path_lengthCount.items())

    return path_length, count
shortest_path_lengths, shortest_path_length_values = analyze_path_length_distribution_c
Path_length_distribution = {shortest_path_lengths[i]: shortest_path_length_values[i] fo
```

In [ ]:

```python
print(Path_length_distribution, "\n")

plt.figure(figsize=(14, 7))
plt.bar(Path_length_distribution.keys(), Path_length_distribution.values())
plt.xlabel("Shortest path lengths between pairs of nodes")
plt.ylabel("Frequency (Path Length)")
plt.yscale('log')

plt.title("Logplot of the Path Length Distribution")
```

```
{0: 8147, 1: 495316, 2: 7114945, 3: 21252418, 4: 3044950, 5: 394421, 6: 154422, 7: 1324
4, 8: 583, 9: 186, 10: 18}
```

Out[ ]: Text(0.5, 1.0, 'Logplot of the Path Length Distribution')



In [ ]:

```python
import numpy as np
import networkx as nx

# Initialize accumulators for statistics
total_path_length = 0
diameters = []
path_lengths = []
component_count = 0

# Iterate over each connected component of the undirected graph
for component in tqdm(G_Undirected.subgraph(nodes).copy() for nodes in nx.connected_cor
    if nx.average_shortest_path_length(component) > 0:
        component_count += 1
        total_path_length += nx.average_shortest_path_length(component)
```

```
    diameters.append(nx.diameter(component))
    path_lengths.append(nx.average_shortest_path_length(component))

# Compute average values
avg_path_length = total_path_length / component_count
avg_diameter = np.mean(diameters)

# Output the results
print("Mean shortest path length across Indian Railway Network:", avg_path_length)
print("Mean diameter of connected components in the network:", avg_diameter)
```

```
0it [00:00, ?it/s]7it [19:14, 164.98s/it]
Mean shortest path length across Indian Railway Network: 1.3796888795757478
Mean diameter of connected components in the network: 2.857142857142857
```

In [ ]:

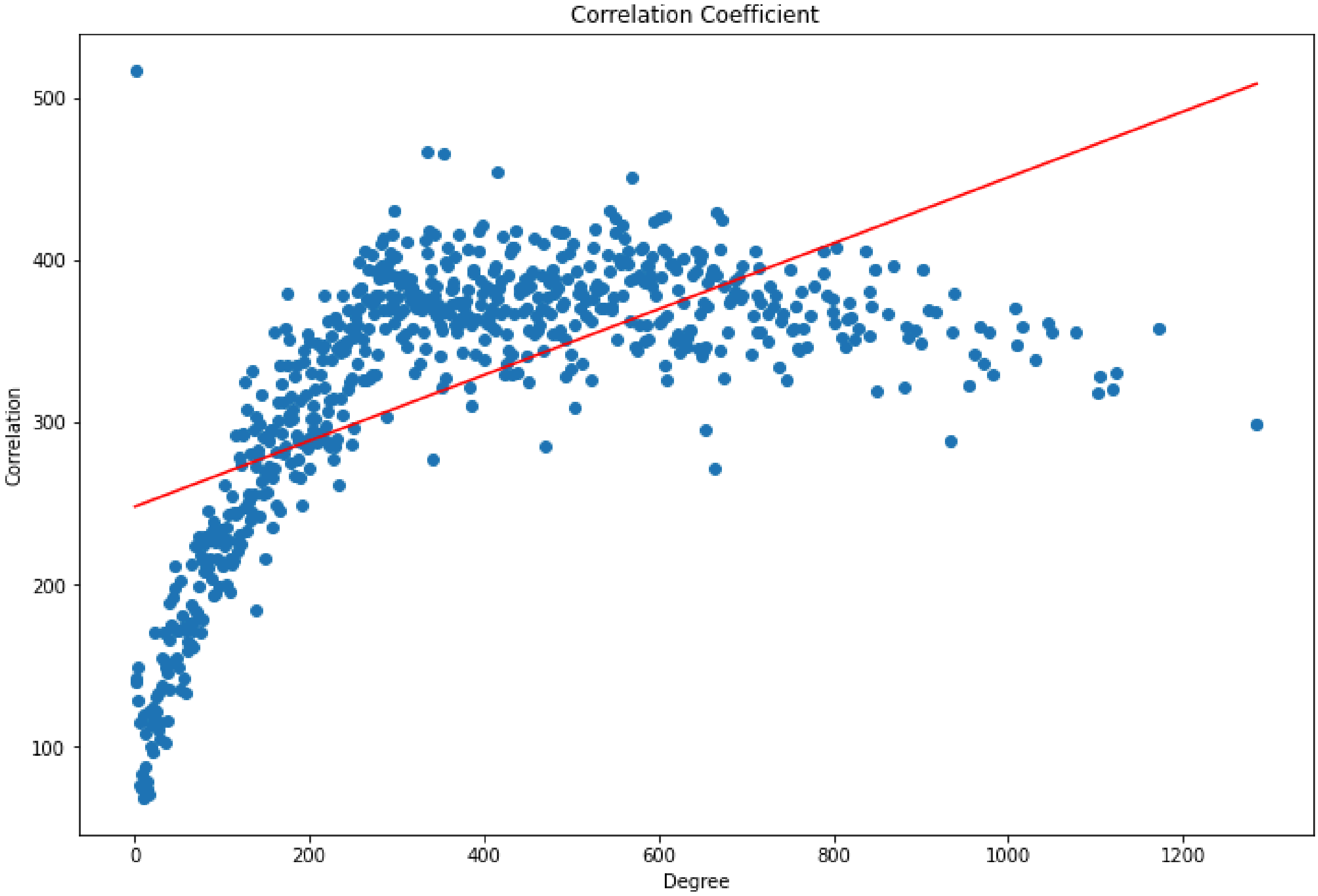
```
import networkx as nx

# Generate a Barabási–Albert scale-free network
ba_graph = nx.barabasi_albert_graph(n=8147, m=30)

# Calculate clustering coefficient and path length
ba_clustering = nx.average_clustering(ba_graph)
ba_path_length = nx.average_shortest_path_length(ba_graph)

# Display results
print("Barabási–Albert Model - Avg. Clustering Coefficient:", ba_clustering)
print("Barabási–Albert Model - Avg. Shortest Path Length:", ba_path_length)

# ----------------------------

# Generate an Erdős–Rényi random graph with similar size and density
# p is chosen so that expected number of edges is roughly equal to BA model
# BA model has ~n*m edges, so p ≈ 2*m / (n - 1)
n = 8147
m = 30
p = 2 * m / (n - 1)
er_graph = nx.erdos_renyi_graph(n=n, p=p)

# Compute metrics for the ER model
er_clustering = nx.average_clustering(er_graph)
er_path_length = nx.average_shortest_path_length(er_graph)

# Display results
print("Erdős–Rényi Model - Avg. Clustering Coefficient:", er_clustering)
print("Erdős–Rényi Model - Avg. Shortest Path Length:", er_path_length)
```

```
Barabási–Albert Model - Avg. Clustering Coefficient: 0.02736096103553356
Barabási–Albert Model - Avg. Shortest Path Length: 2.5331992414970306
Erdős–Rényi Model - Avg. Clustering Coefficient: 0.007374898133090904
Erdős–Rényi Model - Avg. Shortest Path Length: 2.629876124421465
```

In [ ]:

```
def compute_degree_correlation(undirected_graph):
    # Get the degree of all nodes
    node_degrees = dict(undirected_graph.degree())
    unique_k_values = set(node_degrees.values())
    degree_correlation = {}

    for k in unique_k_values:
        # Find nodes with degree exactly equal to k
        nodes_with_k = [node for node, deg in node_degrees.items() if deg == k]
        if not nodes_with_k:
            continue

        # Sum of average neighbor degrees for all such nodes
        average_neighbor_degree_sum = 0
```

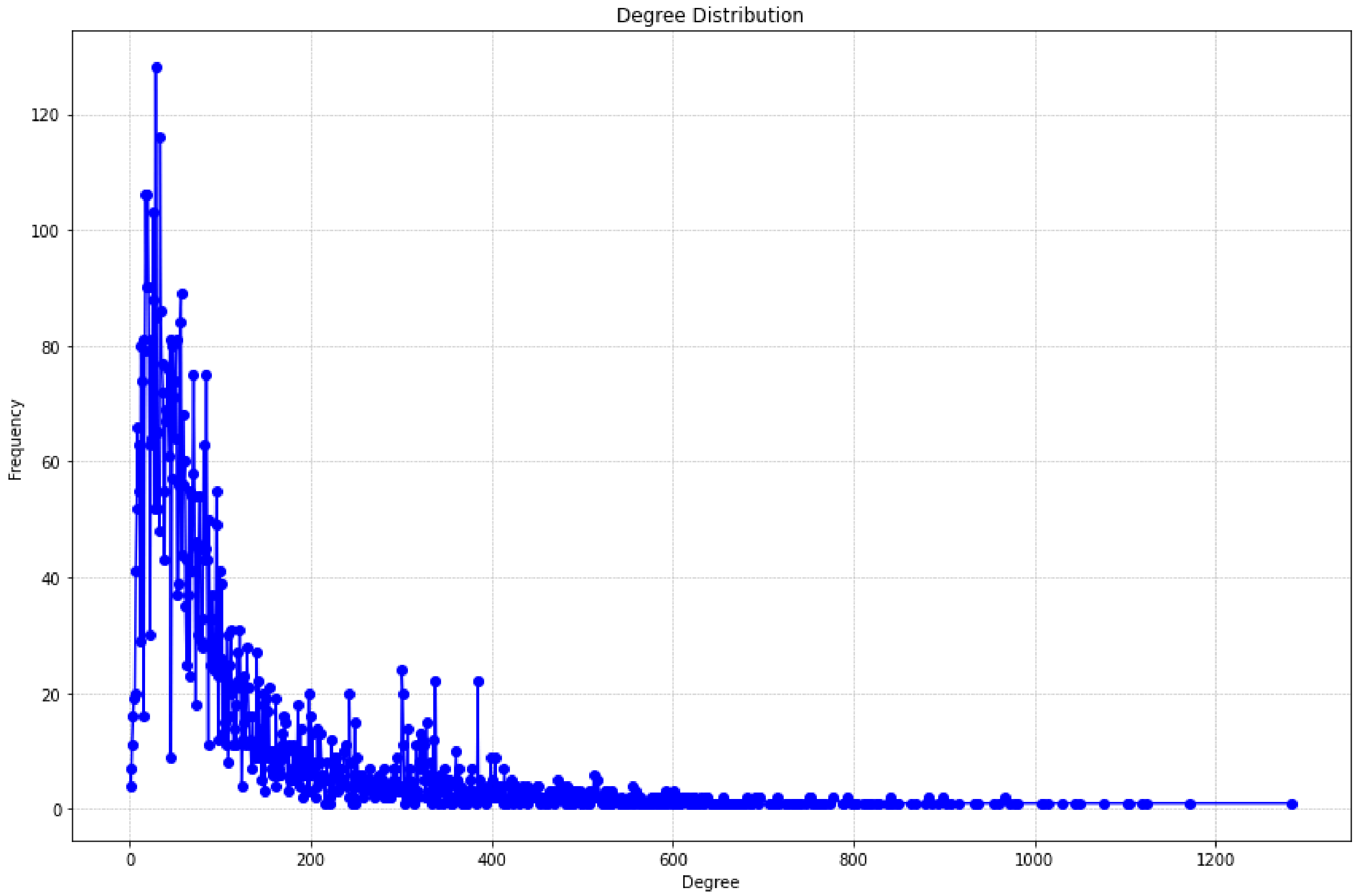
```
        for node in nodes_with_k:
            neighbors = list(undirected_graph.neighbors(node))
            if neighbors:
                neighbor_degrees = [node_degrees[neighbor] for neighbor in neighbors]
                avg_neighbor_deg = sum(neighbor_degrees) / len(neighbor_degrees)
            else:
                avg_neighbor_deg = 0
            average_neighbor_degree_sum += avg_neighbor_deg

        # Final correlation value for current degree k
        degree_correlation[k] = average_neighbor_degree_sum / len(nodes_with_k)

    return degree_correlation

Degree_Correlation = compute_degree_correlation(G_Undirected)
Degree_Values = (list(Degree_Correlation.keys()))
Degree_Correlation_Values = (list(Degree_Correlation.values()))

plt.figure(figsize=(12, 8))

plt.scatter(Degree_Values, Degree_Correlation_Values)

plt.xlabel("Degree")
plt.ylabel("Correlation")

m, b = np.polyfit(np.array(Degree_Values), np.array(Degree_Correlation_Values), 1)
plt.plot(np.array(Degree_Values), m*np.array(Degree_Values) + b, color = 'r')

plt.title("Correlation Coefficient")

plt.show()
```

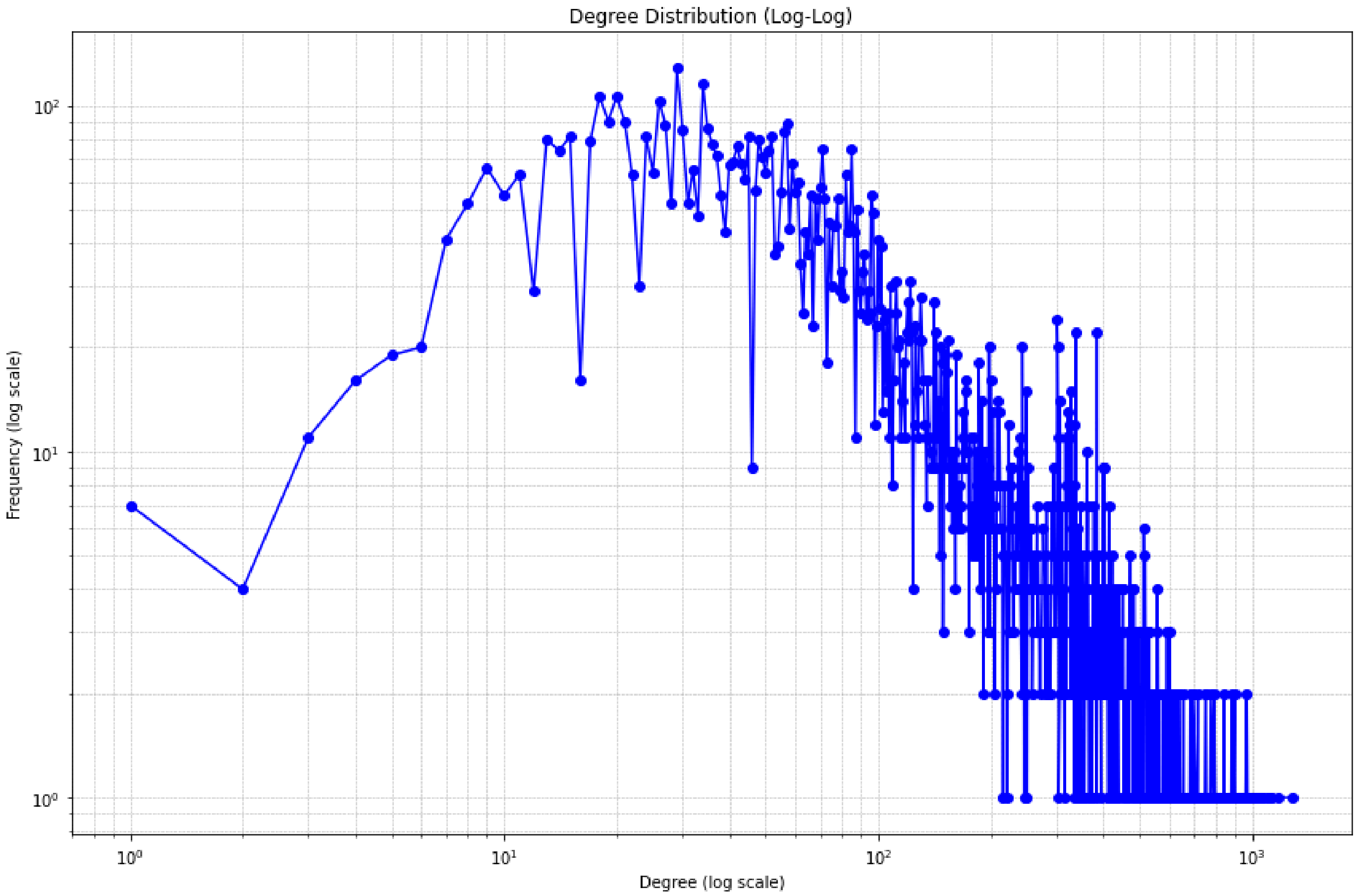


In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np

def plot_degree_distribution(graph, loglog=False, title="Degree Distribution"):
    """
    Plots the degree distribution of a given graph.
```

```
    Parameters:
    - graph: NetworkX graph (can be directed or undirected)
    - loglog: If True, plots the degree distribution on a log-log scale
    - title: Title of the plot
    """

    # Get degree of all nodes
    if graph.is_directed():
        degrees = [deg for _, deg in graph.degree()]
    else:
        degrees = list(dict(graph.degree()).values())

    # Count frequency of each degree
    degree_counts = dict()
    for deg in degrees:
        degree_counts[deg] = degree_counts.get(deg, 0) + 1

    # Sort the data for plotting
    x = sorted(degree_counts.keys())
    y = [degree_counts[k] for k in x]

    # Plot the degree distribution
    plt.figure(figsize=(12, 8))
    if loglog:
        plt.loglog(x, y, 'bo-')
        plt.xlabel("Degree (log scale)")
        plt.ylabel("Frequency (log scale)")
        plt.title(f"{title} (Log-Log)")
    else:
        plt.plot(x, y, 'bo-')
        plt.xlabel("Degree")
        plt.ylabel("Frequency")
        plt.title(title)

    plt.grid(True, which="both", linestyle='--', linewidth=0.5)
    plt.tight_layout()
    plt.show()

plot_degree_distribution(G_Undirected, loglog=False)
plot_degree_distribution(G_Undirected.to_undirected(), loglog=True)
```





In [ ]:

```
assortativity_coefficient = nx.degree_pearson_correlation_coefficient(railway_network)
print("The Assotativity Coefficient of the Indian Railway Network is: ", assortativity_
```

```
The Assotativity Coefficient of the Indian Railway Network is:  0.24250767571456786
```

In [ ]:

```
import networkx as nx
import numpy as np
import random
from collections import Counter, defaultdict
```

```python
from itertools import combinations
import matplotlib.pyplot as plt
import time
import pandas as pd

def get_triad_id(G, nodes):
    """
    Generate a canonical ID for a 3-node subgraph based on its adjacency pattern.
    This implementation is more efficient for large networks.
    """
    # Create a 3x3 adjacency matrix
    adj = np.zeros((3, 3), dtype=int)

    # Map nodes to indices 0, 1, 2
    node_to_idx = {node: i for i, node in enumerate(nodes)}

    # Fill the adjacency matrix
    for u, v in G.subgraph(nodes).edges():
        adj[node_to_idx[u]][node_to_idx[v]] = 1

    # Return as a hashable tuple
    return tuple(map(tuple, adj))

def enumerate_connected_triads(G):
    """
    Enumerate all connected 3-node subgraphs.
    This approach is more memory-efficient for large networks.
    """
    triad_counts = Counter()
    print("Enumerating connected triads...")

    # Process in batches of nodes to avoid memory issues
    nodes = list(G.nodes())
    nodes_count = len(nodes)
    batch_size = min(50, nodes_count)  # Adjust batch size based on memory constraints

    total_processed = 0
    start_time = time.time()

    for i in range(0, nodes_count, batch_size):
        batch_nodes = nodes[i:min(i+batch_size, nodes_count)]

        # Process triplets involving at least one node from this batch
        for node1 in batch_nodes:
            neighbors = set(G.successors(node1)).union(set(G.predecessors(node1)))
            neighbors = list(neighbors)

            # Consider triplets with node1 and two of its neighbors
            for j, node2 in enumerate(neighbors):
                for node3 in neighbors[j+1:]:
                    triplet = (node1, node2, node3)
                    subgraph = G.subgraph(triplet)

                    if nx.is_weakly_connected(subgraph):
                        # Get canonical ID and increment count
                        triad_id = get_triad_id(G, triplet)
                        triad_counts[triad_id] += 1

        total_processed += len(batch_nodes)
        elapsed = time.time() - start_time
        print(f"Processed {total_processed}/{nodes_count} nodes in {elapsed:.2f} second

    # Divide by 6 because each triad is counted multiple times
    # (once for each node in the triad)
    for triad_id in triad_counts:
        triad_counts[triad_id] = triad_counts[triad_id] // 6

    print(f"Found {len(triad_counts)} unique connected triad patterns")
    return triad_counts
```

```python
def generate_random_graph(G, preserving_method='configuration'):
    """
    Generate a random graph with the same degree sequence as G.
    """
    if preserving_method == 'configuration':
        # Configuration model preserves degree sequence
        in_degrees = [d for n, d in G.in_degree()]
        out_degrees = [d for n, d in G.out_degree()]

        try:
            R = nx.directed_configuration_model(in_degrees, out_degrees)
            R = nx.DiGraph(R)  # Remove parallel edges
            R.remove_edges_from(nx.selfloop_edges(R))  # Remove self-loops
        except Exception as e:
            print(f"Configuration model failed: {e}. Using edge swapping instead.")
            R = generate_random_graph(G, 'edge_swap')
    else:
        # Edge swapping preserves exact degree sequence
        R = G.copy()
        try:
            n_swaps = min(10 * len(G.edges()), 100000)  # Cap the number of swaps
            nx.algorithms.swap.directed_edge_swap(R, nswaps=n_swaps, max_tries=n_swaps
        except Exception as e:
            print(f"Edge swapping warning: {e}")

    return R

def calculate_motif_significance(G, num_random=10):
    """
    Calculate the significance of each triad motif using Z-scores.
    """
    # Count triads in the original network
    original_counts = enumerate_connected_triads(G)

    # Initialize arrays for random networks
    random_counts = defaultdict(list)

    # Generate random networks and count triads
    print(f"Generating {num_random} random networks...")
    for i in range(num_random):
        start_time = time.time()
        R = generate_random_graph(G)
        print(f"Random network {i+1} generated in {time.time() - start_time:.2f} secon

        start_time = time.time()
        r_counts = enumerate_connected_triads(R)
        print(f"Triad counting for random network {i+1} completed in {time.time() - sta

        for triad_id, count in original_counts.items():
            random_counts[triad_id].append(r_counts.get(triad_id, 0))

    # Calculate z-scores
    results = []
    for triad_id, original_count in original_counts.items():
        random_values = random_counts[triad_id]
        mean_random = np.mean(random_values)
        std_random = np.std(random_values)

        # Calculate z-score with proper handling of zero std
        if std_random > 0:
            z_score = (original_count - mean_random) / std_random
        else:
            if original_count == mean_random:
                z_score = 0
            else:
                z_score = float('inf') if original_count > mean_random else float('-in

        results.append({
```

```python
            'triad_id': triad_id,
            'original_count': original_count,
            'mean_random': mean_random,
            'std_random': std_random,
            'z_score': z_score
        })

    # Sort by absolute z-score
    results.sort(key=lambda x: abs(x['z_score']), reverse=True)
    return results

def visualize_triad(triad_id, index):
    """
    Visualize a 3-node subgraph from its adjacency matrix.
    """
    # Create a directed graph from the adjacency matrix
    G = nx.DiGraph()
    G.add_nodes_from([0, 1, 2])

    for i in range(3):
        for j in range(3):
            if triad_id[i][j] == 1:
                G.add_edge(i, j)

    # Position nodes in a triangle
    pos = {0: (0, 0), 1: (1, 0), 2: (0.5, 0.866)}

    plt.figure(figsize=(4, 4))
    nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=500,
            arrowsize=20, font_weight='bold', font_size=16)
    plt.title(f"Triad {index+1}")
    plt.savefig(f"triad_{index+1}.png")
    plt.close()

def classify_triads(results, threshold=0):
    """
    Classify triads as motifs or anti-motifs based on z-scores.
    """
    motifs = [r for r in results if r['z_score'] > threshold]
    anti_motifs = [r for r in results if r['z_score'] < -threshold]
    neutral = [r for r in results if abs(r['z_score']) <= threshold]

    return motifs, anti_motifs, neutral
def classify_triads_auto_threshold(results, percentile=95):
    """
    Automatically determine thresholds for motifs and anti-motifs based on Z-score dis
    """
    z_scores = np.array([r['z_score'] for r in results if np.isfinite(r['z_score'])])

    upper_thresh = np.percentile(z_scores, percentile)
    lower_thresh = np.percentile(z_scores, 100 - percentile)

    print(f"\nAutomatically determined Z-score thresholds:")
    print(f"Motif threshold: > {upper_thresh:.2f}, Anti-motif threshold: < {lower_thres

    motifs = [r for r in results if r['z_score'] > upper_thresh]
    anti_motifs = [r for r in results if r['z_score'] < lower_thresh]
    neutral = [r for r in results if lower_thresh <= r['z_score'] <= upper_thresh]

    return motifs, anti_motifs, neutral, upper_thresh, lower_thresh

def print_results(motifs, anti_motifs, neutral, upper_thresh, lower_thresh):
    """
    Print and visualize motifs and anti-motifs based on auto-computed Z-score threshol
    """
    print(f"\n===== AUTOMATIC MOTIF ANALYSIS =====")
    print(f"Motif Z-score threshold: > {upper_thresh:.2f}")
    print(f"Anti-motif Z-score threshold: < {lower_thresh:.2f}")
```

```
    print(f"\n=== TOP {len(motifs)} MOTIFS (Z > {upper_thresh:.2f}) ===")
    for i, r in enumerate(motifs[:10]):  # visualize top 10
        print(f"Motif {i+1}: Z = {r['z_score']:.2f}, Count = {r['original_count']}, Ran
        visualize_triad(r['triad_id'], i)

    print(f"\n=== TOP {len(anti_motifs)} ANTI-MOTIFS (Z < {lower_thresh:.2f}) ===")
    for i, r in enumerate(anti_motifs[:10]):  # visualize top 10
        print(f"Anti-motif {i+1}: Z = {r['z_score']:.2f}, Count = {r['original_count']}
        visualize_triad(r['triad_id'], i + len(motifs))

    print(f"\nNeutral triads: {len(neutral)} (Z between {lower_thresh:.2f} and {upper_

    # Create a summary table
    data = []
    for r in motifs:
        data.append({
            'Type': 'Motif',
            'Z-score': r['z_score'],
            'Original Count': r['original_count'],
            'Random Mean': r['mean_random'],
            'Standard Deviation': r['std_random']
        })
    for r in anti_motifs:
        data.append({
            'Type': 'Anti-motif',
            'Z-score': r['z_score'],
            'Original Count': r['original_count'],
            'Random Mean': r['mean_random'],
            'Standard Deviation': r['std_random']
        })

    df = pd.DataFrame(data)
    print("\n===== SUMMARY TABLE =====")
    print(df)

    # Save to CSV
    df.to_csv('motif_analysis_results.csv', index=False)
    print("Results saved to motif_analysis_results.csv")


start_time = time.time()

# Calculate motif significance
results = calculate_motif_significance(railway_network, num_random=5)

# Classify triads
motifs, anti_motifs, neutral, upper_thresh, lower_thresh = classify_triads_auto_thresho
```

```
Enumerating connected triads...
Processed 50/8147 nodes in 1593.72 seconds
Processed 100/8147 nodes in 2614.48 seconds
Processed 150/8147 nodes in 3761.80 seconds
Processed 200/8147 nodes in 5437.24 seconds
Processed 250/8147 nodes in 6457.09 seconds
Processed 300/8147 nodes in 7052.18 seconds
Processed 350/8147 nodes in 8577.03 seconds
Processed 400/8147 nodes in 9471.46 seconds
```

In [ ]:

```
import networkx as nx
import numpy as np
import random
from collections import Counter, defaultdict
from itertools import combinations
import matplotlib.pyplot as plt
import time
import pandas as pd
import multiprocessing as mp
from functools import lru_cache
from scipy.sparse import csr_matrix
```

```python
def get_triad_id_fast(G, nodes):
    """
    More efficient triad ID calculation using bitwise operations.
    """
    # Create a binary representation for fast comparison
    triad_id = 0
    node_to_idx = {node: i for i, node in enumerate(nodes)}

    for u, v in G.subgraph(nodes).edges():
        i, j = node_to_idx[u], node_to_idx[v]
        # Set a bit for each edge
        triad_id |= (1 << (i * 3 + j))

    return triad_id

def process_node_batch(args):
    """
    Process a batch of nodes for parallel execution.
    """
    G, batch_nodes, all_nodes_dict = args
    local_triad_counts = Counter()

    # Pre-compute neighborhoods for faster access
    neighborhoods = {node: set(G.successors(node)).union(set(G.predecessors(node)))
                     for node in batch_nodes}

    for node1 in batch_nodes:
        neighbors = neighborhoods[node1]
        neighbors_list = list(neighbors)

        # Process pairs of neighbors to form triads
        for i, node2 in enumerate(neighbors_list):
            for node3 in neighbors_list[i+1:]:
                # Early filtering: only process if this node is responsible for the tri
                if node1 <= min(node2, node3) or (node1 not in all_nodes_dict.get(node2
                                                  node1 not in all_nodes_dict.get(node3
                    triplet = (node1, node2, node3)
                    subgraph = G.subgraph(triplet)

                    if nx.is_weakly_connected(subgraph):
                        triad_id = get_triad_id_fast(G, triplet)
                        local_triad_counts[triad_id] += 1

    return local_triad_counts

def enumerate_connected_triads_parallel(G, num_processes=None):
    """
    Parallelized implementation of triad enumeration.
    """
    if num_processes is None:
        num_processes = max(1, mp.cpu_count() - 1)  # Leave one core free

    print(f"Enumerating connected triads using {num_processes} processes...")

    # Create a node-to-neighborhood mapping for filtering
    all_nodes = list(G.nodes())
    all_nodes_dict = {}

    # Pre-compute neighborhoods to avoid redundant calculations
    for node in all_nodes:
        all_nodes_dict[node] = set(G.successors(node)).union(set(G.predecessors(node)))

    # Process in batches
    batch_size = max(1, len(all_nodes) // (num_processes * 4))  # Create more batches t
    node_batches = [all_nodes[i:i+batch_size] for i in range(0, len(all_nodes), batch_s

    # Prepare arguments for parallel processing
    args_list = [(G, batch, all_nodes_dict) for batch in node_batches]
```

```python
    start_time = time.time()

    # Use multiprocessing to distribute the workload
    with mp.Pool(processes=num_processes) as pool:
        results = pool.map(process_node_batch, args_list)

    # Combine results from all processes
    triad_counts = Counter()
    for local_counts in results:
        triad_counts.update(local_counts)

    print(f"Triad enumeration completed in {time.time() - start_time:.2f} seconds")
    print(f"Found {len(triad_counts)} unique connected triad patterns")

    return triad_counts

def generate_random_graph(G, preserving_method='configuration'):
    """
    Generate a random graph with the same degree sequence as G.
    Optimized implementation with error handling.
    """
    if preserving_method == 'configuration':
        # Configuration model preserving degree sequence
        in_degrees = [d for n, d in G.in_degree()]
        out_degrees = [d for n, d in G.out_degree()]

        try:
            R = nx.directed_configuration_model(in_degrees, out_degrees, seed=random.ra
            R = nx.DiGraph(R)  # Remove parallel edges
            R.remove_edges_from(nx.selfloop_edges(R))  # Remove self-loops
        except Exception as e:
            print(f"Configuration model failed: {e}. Using edge swapping instead.")
            R = generate_random_graph(G, 'edge_swap')
    else:
        # Edge swapping preserves exact degree sequence
        R = G.copy()
        try:
            n_swaps = min(10 * len(G.edges()), 100000)  # Cap the number of swaps
            nx.algorithms.swap.directed_edge_swap(R, nswaps=n_swaps, max_tries=n_swaps
        except Exception as e:
            print(f"Edge swapping warning: {e}")

    return R

def calculate_motif_significance_early_stop(G, min_random=5, max_random=20,
                                            convergence_threshold=0.1, num_processes=Non
    """
    Calculate motif significance with early stopping based on convergence.
    """
    # Count triads in the original network
    original_counts = enumerate_connected_triads_parallel(G, num_processes)

    # Initialize arrays for random networks
    random_counts = defaultdict(list)
    z_scores = {}

    # Generate random networks and count triads with early stopping
    print(f"Generating random networks (min={min_random}, max={max_random})...")

    for i in range(max_random):
        start_time = time.time()
        R = generate_random_graph(G)
        print(f"Random network {i+1} generated in {time.time() - start_time:.2f} second

        start_time = time.time()
        r_counts = enumerate_connected_triads_parallel(R, num_processes)
        print(f"Triad counting for random network {i+1} completed in {time.time() - sta
```

```
        # Update random counts and recalculate z-scores
        for triad_id, original_count in original_counts.items():
            random_value = r_counts.get(triad_id, 0)
            random_counts[triad_id].append(random_value)

            # Calculate z-score with at least min_random samples
            if i + 1 >= min_random:
                values = random_counts[triad_id]
                mean_random = np.mean(values)
                std_random = np.std(values) if len(values) > 1 else 1e-6

                # Calculate z-score with proper handling of zero std
                if std_random > 0:
                    new_z_score = (original_count - mean_random) / std_random
                else:
                    new_z_score = 0 if original_count == mean_random else \
                                  float('inf') if original_count > mean_random else flo

                z_scores[triad_id] = new_z_score

        # Check for convergence after minimum iterations
        if i + 1 >= min_random and i + 1 < max_random:
            # Check if z-scores have stabilized
            if i > 0 and check_convergence(z_scores, original_counts, random_counts, co
                print(f"Z-scores converged after {i+1} random networks")
                break

    # Prepare final results
    results = []
    for triad_id, original_count in original_counts.items():
        random_values = random_counts[triad_id]
        mean_random = np.mean(random_values)
        std_random = np.std(random_values) if len(random_values) > 1 else 1e-6

        # Calculate final z-score
        if std_random > 0:
            z_score = (original_count - mean_random) / std_random
        else:
            z_score = 0 if original_count == mean_random else \
                      float('inf') if original_count > mean_random else float('-inf')

        results.append({
            'triad_id': triad_id,
            'original_count': original_count,
            'mean_random': mean_random,
            'std_random': std_random,
            'z_score': z_score
        })

    # Sort by absolute z-score
    results.sort(key=lambda x: abs(x['z_score']), reverse=True)
    return results

def check_convergence(z_scores, original_counts, random_counts, threshold):
    """
    Check if z-scores have stabilized based on most significant triads.
    """
    # Sort triads by absolute z-score
    sorted_triads = sorted(z_scores.items(), key=lambda x: abs(x[1]), reverse=True)

    # Take top triads for convergence check
    top_n = min(10, len(sorted_triads))
    top_triads = [t[0] for t in sorted_triads[:top_n]]

    # Calculate relative changes in z-scores if we had one less random network
    changes = []
    for triad_id in top_triads:
        values = random_counts[triad_id][:-1]  # All but the last
        if len(values) <= 1:  # Need at least 2 values for std
```

```
            continue

        mean_prev = np.mean(values)
        std_prev = np.std(values)

        if std_prev > 0:
            z_prev = (original_counts[triad_id] - mean_prev) / std_prev
            z_current = z_scores[triad_id]
            relative_change = abs((z_current - z_prev) / (z_prev + 1e-10))
            changes.append(relative_change)

    # If all top triads have stabilized
    return changes and max(changes) < threshold

def visualize_triad(triad_id, index):
    """
    Visualize a 3-node subgraph using its binary ID representation.
    """
    # Create a directed graph
    G = nx.DiGraph()
    G.add_nodes_from([0, 1, 2])

    # Convert binary ID back to edges
    for i in range(3):
        for j in range(3):
            if (triad_id & (1 << (i * 3 + j))) != 0:
                G.add_edge(i, j)

    # Position nodes in a triangle
    pos = {0: (0, 0), 1: (1, 0), 2: (0.5, 0.866)}

    plt.figure(figsize=(4, 4))
    nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=500,
            arrowsize=20, font_weight='bold', font_size=16)
    plt.title(f"Triad {index+1}")
    plt.savefig(f"triad_{index+1}.png")
    plt.close()

def classify_triads_auto_threshold(results, percentile=95):
    """
    Automatically determine thresholds for motifs and anti-motifs based on Z-score dis
    """
    z_scores = np.array([r['z_score'] for r in results if np.isfinite(r['z_score'])])

    upper_thresh = np.percentile(z_scores, percentile)
    lower_thresh = np.percentile(z_scores, 100 - percentile)

    print(f"\nAutomatically determined Z-score thresholds:")
    print(f"Motif threshold: > {upper_thresh:.2f}, Anti-motif threshold: < {lower_thres

    motifs = [r for r in results if r['z_score'] > upper_thresh]
    anti_motifs = [r for r in results if r['z_score'] < lower_thresh]
    neutral = [r for r in results if lower_thresh <= r['z_score'] <= upper_thresh]

    return motifs, anti_motifs, neutral, upper_thresh, lower_thresh

def print_results(motifs, anti_motifs, neutral, upper_thresh, lower_thresh):
    """
    Print and visualize motifs and anti-motifs based on auto-computed Z-score threshol
    """
    print(f"\n===== AUTOMATIC MOTIF ANALYSIS =====")
    print(f"Motif Z-score threshold: > {upper_thresh:.2f}")
    print(f"Anti-motif Z-score threshold: < {lower_thresh:.2f}")

    print(f"\n=== TOP {len(motifs)} MOTIFS (Z > {upper_thresh:.2f}) ===")
    for i, r in enumerate(motifs[:10]):  # visualize top 10
        print(f"Motif {i+1}: Z = {r['z_score']:.2f}, Count = {r['original_count']}, Ra
        visualize_triad(r['triad_id'], i)
```

```
    print(f"\n=== TOP {len(anti_motifs)} ANTI-MOTIFS (Z < {lower_thresh:.2f}) ===")
    for i, r in enumerate(anti_motifs[:10]):  # visualize top 10
        print(f"Anti-motif {i+1}: Z = {r['z_score']:.2f}, Count = {r['original_count']}
        visualize_triad(r['triad_id'], i + len(motifs))

    print(f"\nNeutral triads: {len(neutral)} (Z between {lower_thresh:.2f} and {upper_

    # Create a summary table
    data = []
    for r in motifs:
        data.append({
            'Type': 'Motif',
            'Z-score': r['z_score'],
            'Original Count': r['original_count'],
            'Random Mean': r['mean_random'],
            'Standard Deviation': r['std_random']
        })
    for r in anti_motifs:
        data.append({
            'Type': 'Anti-motif',
            'Z-score': r['z_score'],
            'Original Count': r['original_count'],
            'Random Mean': r['mean_random'],
            'Standard Deviation': r['std_random']
        })

    df = pd.DataFrame(data)
    print("\n===== SUMMARY TABLE =====")
    print(df)

    # Save to CSV
    df.to_csv('motif_analysis_results.csv', index=False)
    print("Results saved to motif_analysis_results.csv")

def analyze_network_motifs(network, min_random=5, max_random=10, percentile=95):
    """
    Main function to analyze network motifs with optimized performance.
    """
    start_time = time.time()
    print(f"Starting motif analysis on network with {network.number_of_nodes()} nodes a

    # Determine number of processes based on system resources
    num_processes = max(1, mp.cpu_count() - 1)

    # Calculate motif significance with early stopping
    results = calculate_motif_significance_early_stop(
        network,
        min_random=min_random,
        max_random=max_random,
        num_processes=num_processes
    )

    # Classify triads
    motifs, anti_motifs, neutral, upper_thresh, lower_thresh = classify_triads_auto_thr

    # Print results
    print_results(motifs, anti_motifs, neutral, upper_thresh, lower_thresh)

    total_time = time.time() - start_time
    print(f"\nTotal analysis time: {total_time:.2f} seconds")

    return results, motifs, anti_motifs, neutral

analyze_network_motifs(railway_network, min_random=5, max_random=10, percentile=95)
```