In [5]:

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

In [52]:

```
df = pd.read_csv("train_stations_europe.csv")
df.drop(columns=['entur_id', 'entur_is_enabled'], inplace=True)
bool_columns = ['is_city', 'is_main_station', 'is_airport']
for col in bool_columns:
    df[col] = df[col].astype(str).str.upper().map({'TRUE': True, 'FALSE': False})

df_cleaned = df.dropna(subset=['latitude', 'longitude'])

df_cleaned['uic'] = df_cleaned['uic'].fillna(-1)
df_cleaned['parent_station_id'] = df_cleaned['parent_station_id'].fillna(-1)

df_cleaned['uic'] = df_cleaned['uic'].astype(int)
df_cleaned['parent_station_id'] = df_cleaned['parent_station_id'].astype(int)

df_cleaned = df_cleaned.reset_index(drop=True)

df_cleaned
```

Out[52]:

	id	name	name_norm	uic	latitude	longitude	parent_station_id	country	time_zone
0	1	Château-Arnoux—St-Auban	Chateau-Arnoux-St-Auban	-1	44.081790	6.001625	-1	FR	Europe/Paris
1	2	Château-Arnoux—St-Auban	Chateau-Arnoux-St-Auban	8775123	44.061565	5.997373	1	FR	Europe/Paris
2	3	Château-Arnoux Mairie	Chateau-Arnoux Mairie	8775122	44.063863	6.011248	1	FR	Europe/Paris
3	4	Digne-les-Bains	Digne-les-Bains	-1	44.350000	6.350000	-1	FR	Europe/Paris
4	6	Digne-les-Bains	Digne-les-Bains	8775149	44.088710	6.222982	4	FR	Europe/Paris
...	...	...	...	...	...	...	...	...	...
62137	68175	Bari Villaggio del Lavoratore	Bari Villaggio del Lavoratore	8311003	41.104234	16.822596	-1	IT	Europe/Rome
62138	68176	Baucca-Garavelle	Baucca-Garavelle	-1	43.442030	12.250215	-1	IT	Europe/Rome
62139	68177	Bibbiano Fossa	Bibbiano Fossa	-1	44.674312	10.478726	-1	IT	Europe/Rome
62140	68178	Bibbiano Via Monti	Bibbiano Via Monti	-1	44.661871	10.466608	-1	IT	Europe/Rome
62141	68179	Bivio Barco	Bivio Barco	-1	44.694544	10.498249	-1	IT	Europe/Rome

62142 rows × 12 columns

In [53]:

```
import networkx as nx
import numpy as np
import pandas as pd
from math import radians, cos, sin, asin, sqrt

# Haversine formula to calculate distance between two coordinates
def haversine(lat1, lon1, lat2, lon2):
    # Convert decimal degrees to radians
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])
    # Haversine calculation
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371  # Radius of earth in kilometers
    return c * r

def generate_station_graph(df, distance_weighted=False):
    G = nx.DiGraph()

    # Add nodes with attributes
    print("Adding nodes...")
    for _, row in tqdm(df.iterrows(), total=len(df), desc="Nodes", ncols=100):
        G.add_node(row['id'], name=row['name'], latitude=row['latitude'], longitude=rov

    # Add edges from child to parent
    print("Adding edges (child → parent)...")
    for _, row in tqdm(df.iterrows(), total=len(df), desc="Edges", ncols=100):
        parent_id = row['parent_station_id']
        if parent_id != -1 and not pd.isna(parent_id) and G.has_node(parent_id):
            child = row['id']
            parent = int(parent_id)

            if distance_weighted:
                lat1, lon1 = row['latitude'], row['longitude']
                lat2, lon2 = df.loc[df['id'] == parent, ['latitude', 'longitude']].valu
                dist = haversine(lat1, lon1, lat2, lon2)
            else:
                dist = 1

            G.add_edge(child, parent, weight=dist)

    return G
```

In [1]:

```
import pickle
try:
  with open("european_network.gpickle", "rb") as f:
    railway_network = pickle.load(f)
except:
  railway_network = generate_station_graph(df_cleaned, distance_weighted=True)
  with open("european_network.gpickle", "wb") as f:
    pickle.dump(railway_network, f)
```

In [55]:

```
# 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}")
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: 62142
Number of edges: 3429
Is directed: True
Is multigraph: False
Number of weakly connected components: 58713
Number of strongly connected components: 62142
Average in-degree: 0.06
Average out-degree: 0.06
```

In [56]:

```
def get_top_degree_nodes(graph, top_k=10, degree_type='total'):
    """
    Get top_k nodes by degree.

    degree_type: 'total' for degree, 'in' for in-degree, 'out' for out-degree (only mat
    """
    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]
# top_nodes = get_top_degree_nodes(railway_network, top_k=5)
def get_top_degree_nodes_in_largest_component(graph, top_k=5):
    # Work with undirected version for connected components
    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, id_col='id', name_col='name'):
    name_map = dict(zip(station_df[id_col], station_df[name_col]))
    for node_id in top_nodes:
        name = name_map.get(node_id, "Unknown")
        print(f"{node_id}: {name}")

top_nodes = get_top_degree_nodes_in_largest_component(railway_network, 10)
print_station_names(top_nodes, df_cleaned)
# print(top_nodes)
```

```
7527: Berlin
7630: Berlin Hbf
7550: Berlin-Spandau
14255: Berlin-Schöneweide
29222: Berlin Ostkreuz
7720: Berlin Wannsee
23600: Berlin-Schöneweide
29232: Berlin Hbf (Washingtonplatz)
14950: Berlin-Hohenschönhausen
16637: Berlin Poelchaustraße
```

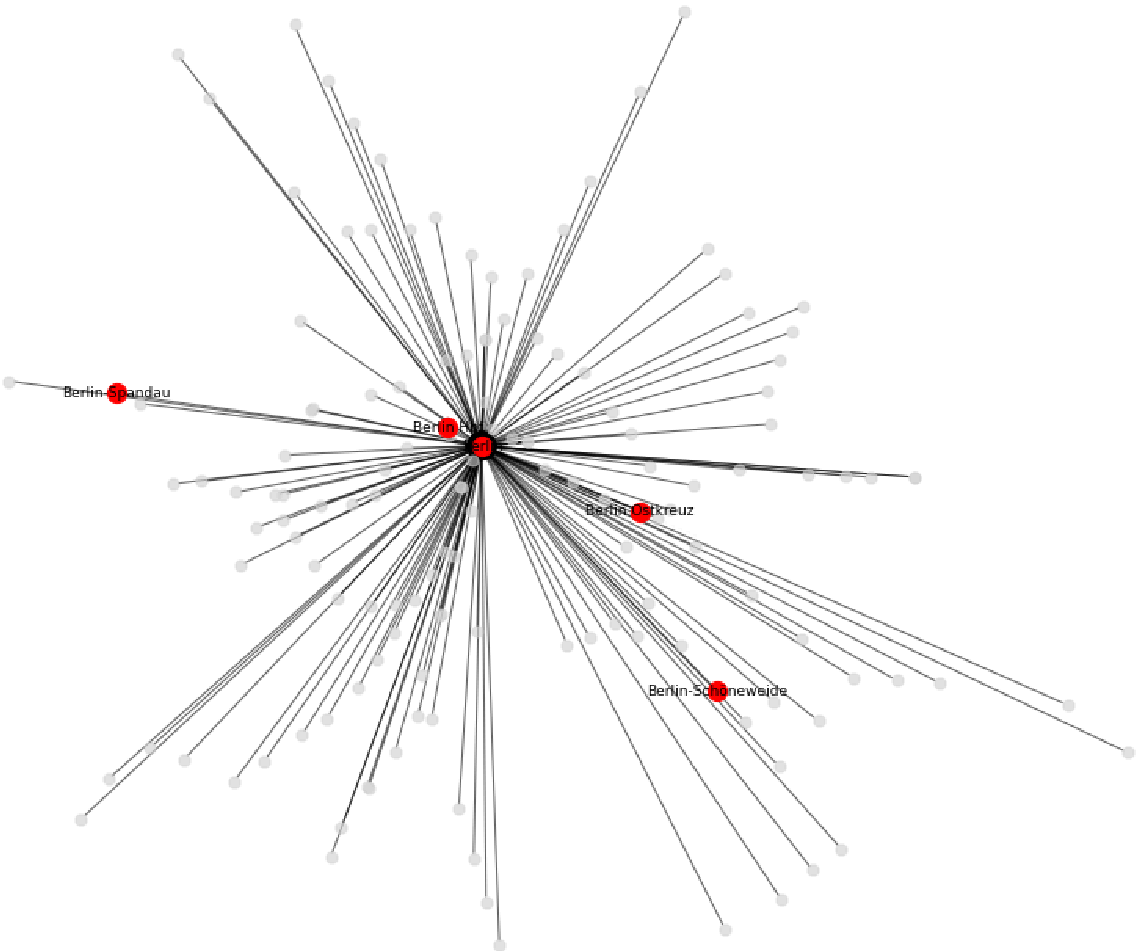
In [57]:

```
import matplotlib.pyplot as plt
from tqdm import tqdm
import networkx as nx
```

```python
def get_subgraph(railway_data, graph, subgraph_nodes, distance_weighted=False, plot=Tru
    # Step 1: Generate local graph with only relevant nodes
    subgraph_station_ids = set(subgraph_nodes)

    # Step 2: Add neighbors of each queried node
    for node in subgraph_nodes:
        if node in graph:
            neighbors = list(graph.successors(node)) + list(graph.predecessors(node))
            subgraph_station_ids.update(neighbors)

    # Step 3: Add nodes in shortest paths between all pairs
    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 source, target in tqdm(node_pairs, desc="Finding shortest paths", ncols=100, le
        try:
            path = nx.shortest_path(graph, source=source, target=target, weight='weigh
            subgraph_station_ids.update(path)
        except nx.NetworkXNoPath:
            continue

    # Step 4: Induce subgraph
    subgraph = graph.subgraph(subgraph_station_ids).copy()

    # Step 5: Plot if requested
    if plot:
        print("Plotting...")
        plt.figure(figsize=(12, 10))
        pos = {
            node: (graph.nodes[node]['longitude'], graph.nodes[node]['latitude'])
            for node in subgraph.nodes
        }

        # Base graph
        nx.draw(subgraph, pos, node_color='lightgray', edge_color='black', node_size=5(

        # Highlight subgraph_nodes in red
        nx.draw_networkx_nodes(subgraph, pos, nodelist=subgraph_nodes, node_color='red

        # Add labels
        labels = {node: graph.nodes[node].get('name', node) for node in subgraph_nodes}
        nx.draw_networkx_labels(subgraph, pos, labels, font_size=9, font_color='black'

        plt.title("Queried Stations and Their Connecting Paths")
        plt.axis("off")
        plt.show()

    return subgraph

subgraph = get_subgraph(df_cleaned, railway_network, subgraph_nodes=top_nodes[:5], dis
subgraph = get_subgraph(df_cleaned, railway_network, subgraph_nodes=top_nodes[5:], dis
```

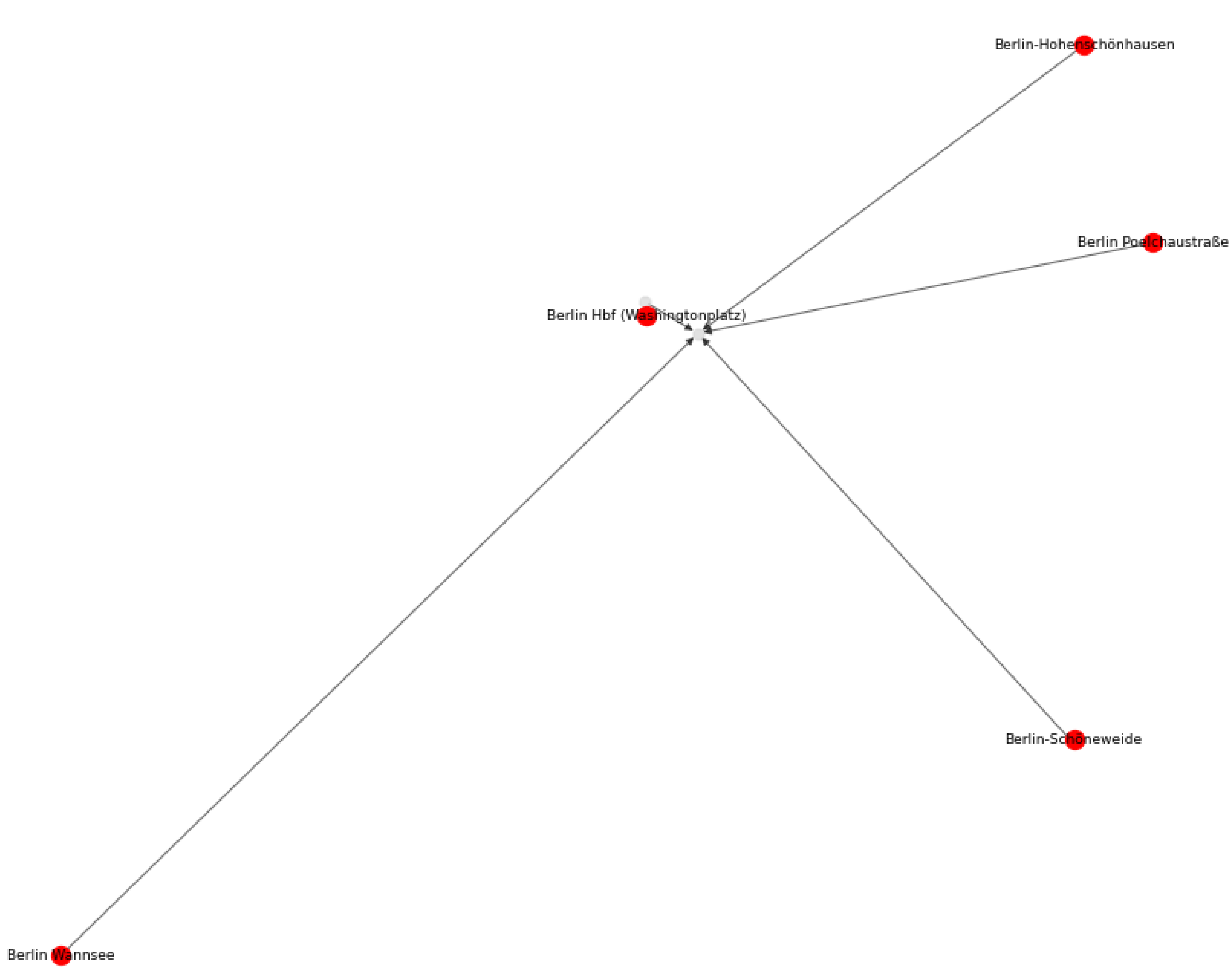
```
Plotting...
```

Queried Stations and Their Connecting Paths




```
Plotting...
```

Queried Stations and Their Connecting Paths



In [58]:

```
import pickle
def _single_source_shortest_path_basic(G, s):
    S = []
    P = {}
    for v in G:
        P[v] = []
    sigma = dict.fromkeys(G, 0.0)
    D = {}
    sigma[s] = 1.0
    D[s] = 0
    Q = [s]
    while Q:
        v = Q.pop(0)
        S.append(v)
        for w in G[v]:
            if w not in D:
                Q.append(w)
                D[w] = D[v] + 1
            if D[w] == D[v] + 1:
                sigma[w] += sigma[v]
                P[w].append(v)
    return S, P, sigma

def _accumulate_basic(betweenness, S, P, sigma, s):
    delta = dict.fromkeys(S, 0)
    while S:
        w = S.pop()
        for v in P[w]:
            delta[v] += (sigma[v] / sigma[w]) * (1 + delta[w])
        if w != s:
            betweenness[w] += delta[w]
```

```python
    return betweenness

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)
    print("Computing Betweenness Centrality (with progress bar)...")
    centrality = {v: 0.0 for v in Railway_Network}
    nodes = list(Railway_Network.nodes())

    for s in tqdm(nodes, desc="Processing nodes"):
        S, P, sigma = _single_source_shortest_path_basic(Railway_Network, s)
        centrality = _accumulate_basic(centrality, S, P, sigma, s)

    n = len(Railway_Network)
    if n <= 2:
        scale = None
    else:
        scale = 1.0 / ((n - 1) * (n - 2))
    for v in centrality:
        centrality[v] *= scale
  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['id'] == station_code]['name'].to_l

  return stations

with open("european_network.gpickle", "rb") as f:
    Railway_Network = pickle.load(f)
```

In [114…

```python
DegreeCentrality_stations = compute_centrality(Railway_Network, "Degree", df_cleaned)
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
         Berlin                  0.0023012181973254437
         Göteborg                0.0019471846285061393
         Hamburg                 0.001013823401618899
         Wien            0.0008046217473165865
         München                 0.0006919747026922643
         Dortmund                0.0006919747026922643
         Leipzig                 0.0006276049629069375
         Stockholm               0.0006276049629069375
         Chemnitz                0.0005149579182826153
         Frankfurt (Main)                0.00048277304838995186
```

In [60]:

```python
closenessCentrality_stations = compute_centrality(Railway_Network, "Closeness", df_clea
print("Top Stations in the European Railway System acc to the Closeness Centrality:\n\
for item in closenessCentrality_stations:
  print("\t",item[0],"\t\t",item[1])
```

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

        STATION NAME             BETWEENNESS CENTRALITY
```

```
        Berlin                  0.0023049557951194243
        Göteborg                0.001947315461310581
        Hamburg                 0.0010140709775411502
        Wien            0.0008316659782680606
        München                 0.0006970233097342508
        Dortmund                0.0006919747026922643
        Stockholm               0.0006330832386333482
        Leipzig                 0.0006308234498962038
        Chemnitz                0.0005149579182826153
        Frankfurt (Main)                0.0004895488104726179
```

In [61]:

```
%pip install scipy
eigenCentrality_stations = compute_centrality(Railway_Network, "Eigen Vector", df_clean
print("Top Stations in the European Railway System acc to the Eigen Vector Centrality:\
for item in eigenCentrality_stations:
  print("\t",item[0],"\t\t",item[1])
```

```
Requirement already satisfied: scipy in ./venv/lib/python3.6/site-packages
Requirement already satisfied: numpy>=1.14.5 in ./venv/lib/python3.6/site-packages (fro
m scipy)
Note: you may need to restart the kernel to use updated packages.
Top Stations in the European Railway System acc to the Eigen Vector Centrality:

        STATION NAME            BETWEENNESS CENTRALITY
         Berlin                 0.4570738990109226
         Orly           0.33983776617990996
         Zürich                 0.19346481668902413
         Köln           0.17963856431323355
         Évry           0.17132656779321118
         Göteborg               0.1664931208124981
         Oslo           0.16513920663913115
         Nürnberg               0.1629894996341007
         Kraków                 0.149344878266345
         Wrocław                0.13520974301872213
```

In [ ]:

```
betweennessCentrality_stations = compute_centrality(Railway_Network, "Betweenness", df_
print("Top Stations in the European Railway System acc to the Betweenness Centrality:\r
for item in betweennessCentrality_stations:
  print("\t",item[0],"\t\t",item[1])
```

Geographical Analysis:

In [64]:

```
num_unique_stations = df['id'].nunique()
print(f"Number of unique stations: {num_unique_stations}")
```

```
Number of unique stations: 64037
```

In [65]:

```
north = df_cleaned.loc[df_cleaned['latitude'].idxmax()]
south = df_cleaned.loc[df_cleaned['latitude'].idxmin()]
east = df_cleaned.loc[df_cleaned['longitude'].idxmax()]
west = df_cleaned.loc[df_cleaned['longitude'].idxmin()]

print("North Extreme:\n", north)
```

```
North Extreme:
 id                          65383
name                   Narvik stn
name_norm              Narvik stn
uic                            -1
latitude                  68.4417
longitude                 17.4414
parent_station_id              -1
country                        NO
time_zone             Europe/Oslo
is_city                      True
is_main_station             False
is_airport                  False
Name: 59353, dtype: object
```

In [66]:

```
print("South Extreme:\n", south)
```

```
South Extreme:
 id                               38705
name                         Marrakech
name_norm                    Marrakech
uic                                 -1
latitude                       31.6295
longitude                     -7.98108
parent_station_id                   -1
country                             MA
time_zone            Africa/Casablanca
is_city                           True
is_main_station                  False
is_airport                       False
Name: 33967, dtype: object
```

In [67]:

```
print("East Extreme:\n", east)
```

```
East Extreme:
 id                           37765
name                    Volzhskiy
name_norm               Volzhskiy
uic                            -1
latitude                  48.8099
longitude                 44.7532
parent_station_id              -1
country                        RU
time_zone           Europe/Moscow
is_city                      True
is_main_station             False
is_airport                  False
Name: 33037, dtype: object
```

In [68]:

```
print("West Extreme:\n", west)
```

```
West Extreme:
 id                           33339
name                      Funchal
name_norm                 Funchal
uic                            -1
latitude                  32.6474
longitude                  -16.92
parent_station_id              -1
country                        PT
time_zone           Europe/Lisbon
is_city                      True
is_main_station             False
is_airport                  False
Name: 28617, dtype: object
```

In [ ]:

```
merged = df_cleaned.merge(df_cleaned, left_on='parent_station_id', right_on='id', suffi
merged['parent_distance_km'] = merged.apply(
    lambda row: haversine(row['latitude'], row['longitude'], row['latitude_parent'], ro
    axis=1
)
avg_parent_distance = merged['parent_distance_km'].mean()

print("Average Parent Distance:",avg_parent_distance)
```

```
Average Parent Distance: 4.7528879231377745
```

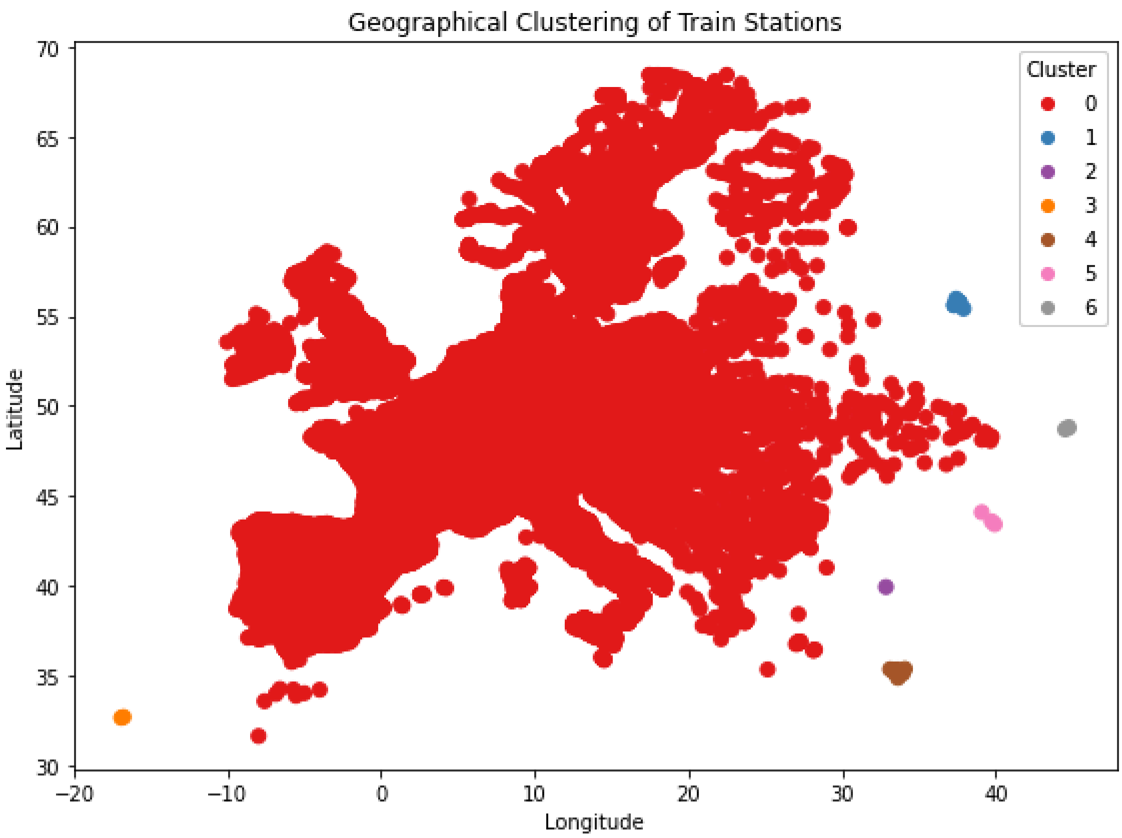
In [78]:

```
import folium
from folium.plugins import HeatMap  # Import HeatMap
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
```

```
# Assuming df_cleaned is already defined and contains the relevant data
# Convert relevant columns
df_cleaned = df_cleaned.dropna(subset=["latitude", "longitude"])
df_cleaned["latitude"] = df_cleaned["latitude"].astype(float)
df_cleaned["longitude"] = df_cleaned["longitude"].astype(float)

# Generate heatmap
m = folium.Map(location=[df_cleaned["latitude"].mean(), df_cleaned["longitude"].mean()]
heat_data = df_cleaned[["latitude", "longitude"]].values.tolist()
HeatMap(heat_data).add_to(m)

# Clustering
coords_rad = np.radians(df_cleaned[['latitude', 'longitude']])
db = DBSCAN(eps=0.05, min_samples=1, metric='haversine').fit(coords_rad)
df_cleaned['cluster'] = db.labels_

# Plot clustering
fig, ax = plt.subplots(figsize=(8, 6))
scatter = ax.scatter(df_cleaned['longitude'], df_cleaned['latitude'], c=df_cleaned['clu
legend = ax.legend(*scatter.legend_elements(), title="Cluster", loc="upper right")
ax.add_artist(legend)
ax.set_title("Geographical Clustering of Train Stations")
ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")
plt.tight_layout()

# Save outputs
heatmap_path = "station_density_heatmap.html"
m.save(heatmap_path)
cluster_plot_path = "cluster_scatter_plot.png"
plt.savefig(cluster_plot_path)
cluster_plot_path, heatmap_path
```

Out[78]:

```
('cluster_scatter_plot.png', 'station_density_heatmap.html')
```



In [3]:

```
import networkx as nx
import numpy as np
import pandas as pd

# Load the graph from the GPickle file
```

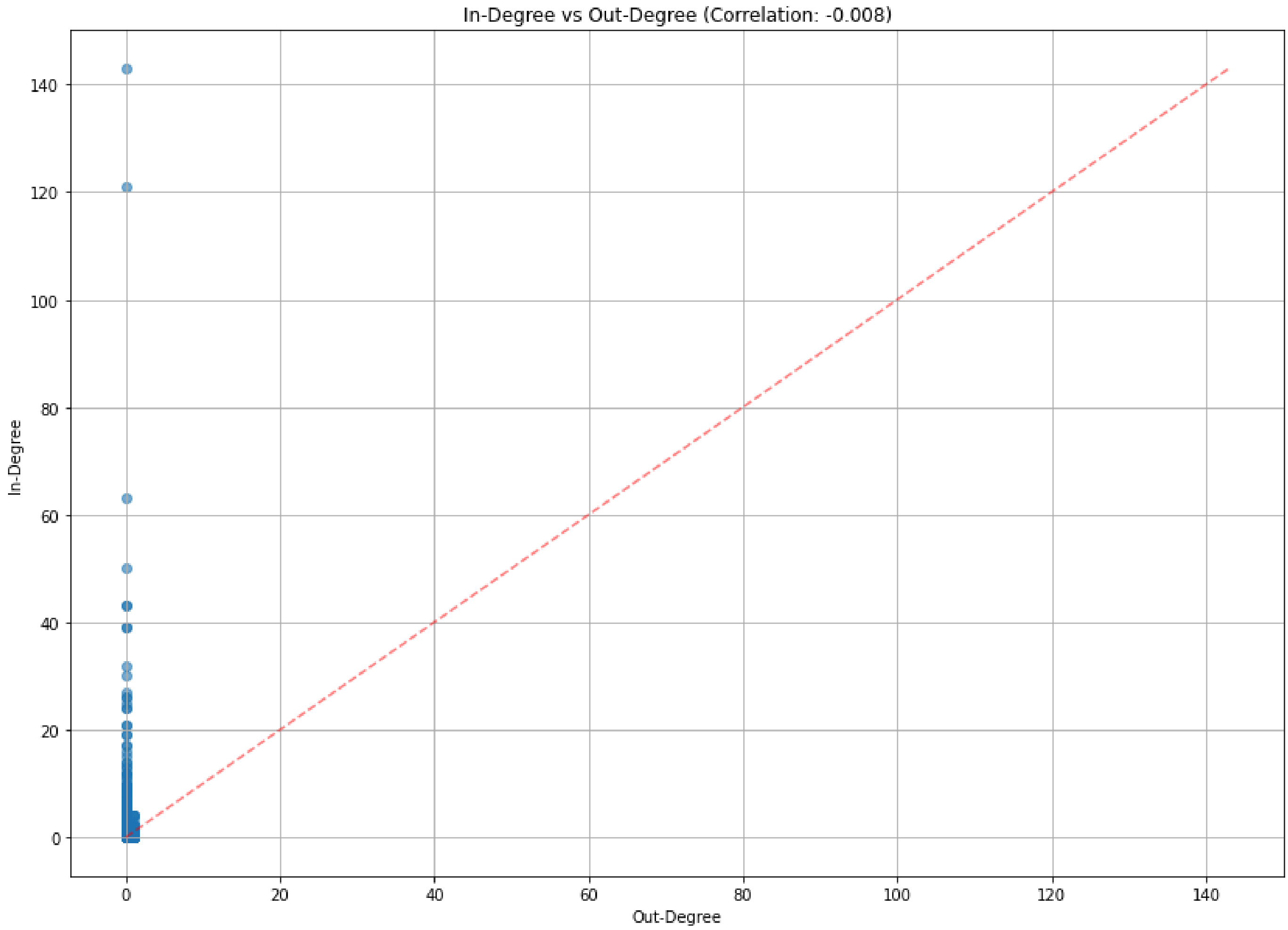
```python
G = nx.read_gpickle('european_network.gpickle')  # Replace with your actual file path
num_nodes = G.number_of_nodes()
num_edges = G.number_of_edges()

print(f"Number of nodes (stations): {num_nodes}")
print(f"Number of edges (routes): {num_edges}")

# 2. Check node attributes
print("\nNode attributes (first 5 nodes):")
for node in list(G.nodes(data=True))[:5]:  # Display attributes for the first 5 nodes
    print(node)

# 3. Check edge attributes
print("\nEdge attributes (first 5 edges):")
for edge in list(G.edges(data=True))[:5]:  # Display attributes for the first 5 edges
    print(edge)

# 4. Check if the graph is weighted
is_weighted = any('weight' in data for _, _, data in G.edges(data=True))
print(f"\nIs the graph weighted? {'Yes' if is_weighted else 'No'}")
unique_stations = G.number_of_nodes()

# 2. Find number of unique trains (assuming each edge represents a unique train route)
unique_trains = G.number_of_edges()

# 3. Calculate route distances
route_lengths = [data['weight'] for _, _, data in G.edges(data=True)]

# 4. Compute metrics
longest_route = max(route_lengths) if route_lengths else 0
shortest_route = min(route_lengths) if route_lengths else 0
max_consecutive_distance = max(route_lengths) if route_lengths else 0
min_consecutive_distance = min(route_lengths) if route_lengths else 0
average_total_route_distance = np.mean(route_lengths) if route_lengths else 0
average_consecutive_distance = np.mean(route_lengths) if route_lengths else 0

# Print results
print("Number of unique stations:", unique_stations)
print("Number of unique trains (routes):", unique_trains)
print("Longest train route distance:", longest_route)
print("Shortest train route distance:", shortest_route)
print("Maximum distance between any two consecutive stations:", max_consecutive_distan
print("Minimum distance between any two consecutive stations:", min_consecutive_distan
print("Average total train route distance:", average_total_route_distance)
print("Average distance between consecutive stops:", average_consecutive_distance)
```

```
Number of nodes (stations): 62142
Number of edges (routes): 3429

Node attributes (first 5 nodes):
(1, {'name': 'Château-Arnoux–St-Auban', 'latitude': 44.081790000000005, 'longitude': 6.
001625})
(2, {'name': 'Château-Arnoux–St-Auban', 'latitude': 44.0615651, 'longitude': 5.99737340
0000001})
(3, {'name': 'Château-Arnoux Mairie', 'latitude': 44.063863, 'longitude': 6.011248})
(4, {'name': 'Digne-les-Bains', 'latitude': 44.35, 'longitude': 6.35})
(6, {'name': 'Digne-les-Bains', 'latitude': 44.088710133980605, 'longitude': 6.22298240
6616211})

Edge attributes (first 5 edges):
(2, 1, {'weight': 2.2744118474539916})
(3, 1, {'weight': 2.1364959736670643})
(6, 4, {'weight': 30.76682649593988})
(9, 5768, {'weight': 0.9555541907702116})
(11, 10, {'weight': 0.915710932014366})

Is the graph weighted? Yes
Number of unique stations: 62142
Number of unique trains (routes): 3429
Longest train route distance: 1734.7779952405094
Shortest train route distance: 0.0
```

```
Maximum distance between any two consecutive stations: 1734.7779952405094
Minimum distance between any two consecutive stations: 0.0
Average total train route distance: 4.7528879231377745
Average distance between consecutive stops: 4.7528879231377745
```

In [109…

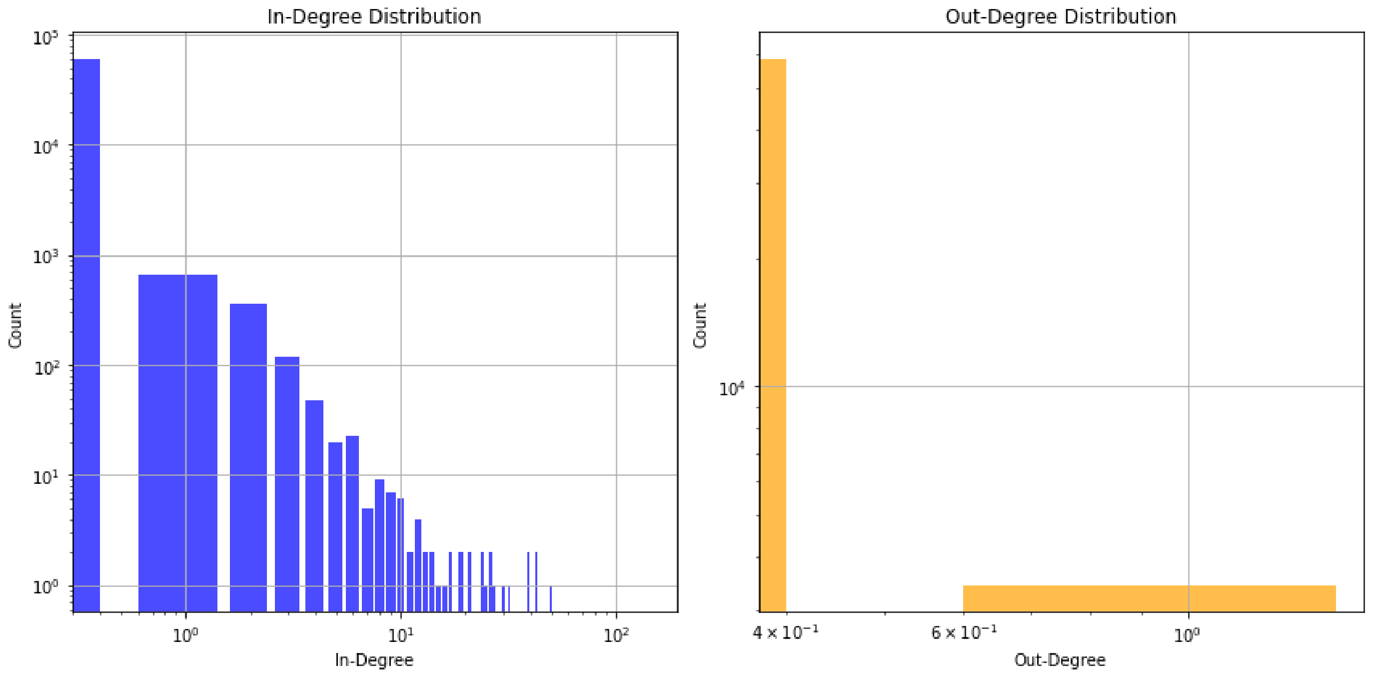
```python
import collections
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import random
from scipy.stats import pearsonr

in_degrees = [d for n, d in G.in_degree()]
out_degrees = [d for n, d in G.out_degree()]

corr, p_value = pearsonr(in_degrees, out_degrees)

# Plot the degree sequences
plt.figure(figsize=(14, 10))
plt.scatter(out_degrees, in_degrees, alpha=0.6)
plt.title(f'In-Degree vs Out-Degree (Correlation: {corr:.3f})')
plt.xlabel('Out-Degree')
plt.ylabel('In-Degree')

# Add a diagonal line for reference
max_degree = max(max(in_degrees), max(out_degrees))
plt.plot([0, max_degree], [0, max_degree], 'r--', alpha=0.5)

plt.grid(True)
plt.show()

print(f"Pearson correlation coefficient: {corr:.3f}")
print(f"P-value: {p_value:.4f}")
```




```
Pearson correlation coefficient: -0.008
P-value: 0.0504
```

In [125…

```
in_degree_sequence = [d for n, d in G.in_degree()]
out_degree_sequence = [d for n, d in G.out_degree()]

# Count the occurrences of each degree
in_degree_count = collections.Counter(in_degree_sequence)
out_degree_count = collections.Counter(out_degree_sequence)

# Prepare data for plotting
in_degrees, in_counts = zip(*sorted(in_degree_count.items()))
out_degrees, out_counts = zip(*sorted(out_degree_count.items()))

# Create subplots
plt.figure(figsize=(12, 6))

# Plot In-Degree Distribution
plt.subplot(1, 2, 1)
plt.bar(in_degrees, in_counts, color='blue', alpha=0.7)
plt.xscale('log')
plt.yscale('log')
plt.title('In-Degree Distribution')
plt.xlabel('In-Degree')
plt.ylabel('Count')
plt.grid(True)

# Plot Out-Degree Distribution
plt.subplot(1, 2, 2)
plt.bar(out_degrees, out_counts, color='orange', alpha=0.7)
plt.xscale('log')
plt.yscale('log')
plt.title('Out-Degree Distribution')
plt.xlabel('Out-Degree')
plt.ylabel('Count')
plt.grid(True)

# Show the plots
plt.tight_layout()
plt.show()
```



AFTER CONVERTING INTO UNDIRECTED NETWORK

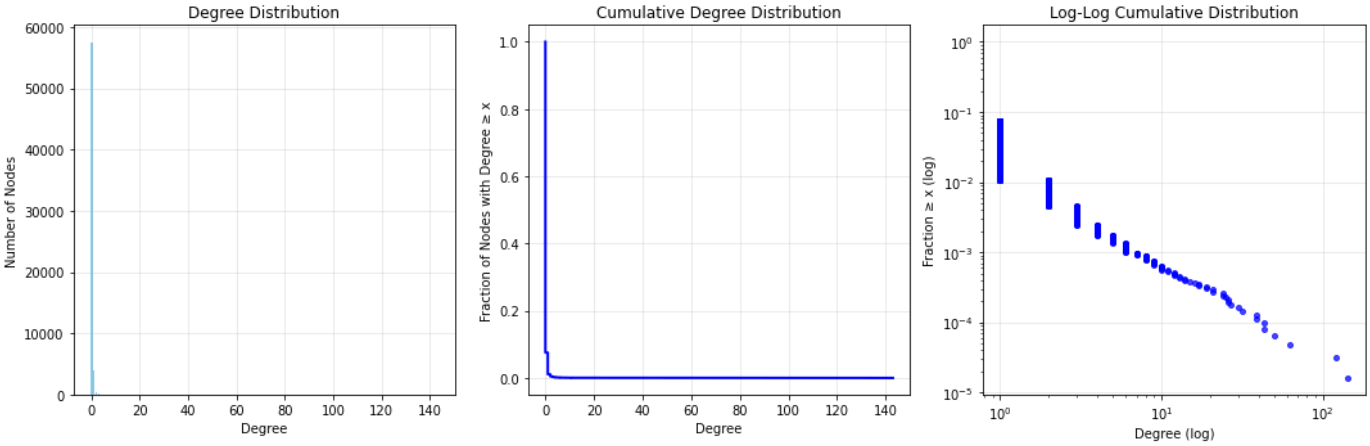
In [155…

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
```

```python
# Convert to undirected graph
G_Undirected = G.to_undirected()

# Get degree sequence
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}")

## 2. Degree Distribution Visualization
plt.figure(figsize=(15, 5))

# Plot 1: Degree histogram
plt.subplot(1, 3, 1)
degree_counts = nx.degree_histogram(G_Undirected)
plt.bar(range(len(degree_counts)), degree_counts, color='skyblue')
plt.title('Degree Distribution')
plt.xlabel('Degree')
plt.ylabel('Number of Nodes')
plt.grid(True, alpha=0.3)

# Plot 2: Cumulative distribution (linear scale)
plt.subplot(1, 3, 2)
sorted_degrees = np.sort(degrees)[::-1]  # Sort in descending order
plt.plot(sorted_degrees, np.arange(1, len(sorted_degrees)+1)/len(sorted_degrees),
         'b-', linewidth=2)
plt.title('Cumulative Degree Distribution')
plt.xlabel('Degree')
plt.ylabel('Fraction of Nodes with Degree ≥ x')
plt.grid(True, alpha=0.3)

# Plot 3: Log-log cumulative distribution
plt.subplot(1, 3, 3)
plt.loglog(sorted_degrees, np.arange(1, len(sorted_degrees)+1)/len(sorted_degrees),
         'bo', markersize=4, alpha=0.7)
plt.title('Log-Log Cumulative Distribution')
plt.xlabel('Degree (log)')
plt.ylabel('Fraction ≥ x (log)')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

## 3. Connectivity Analysis
print("\n=== Connectivity Analysis ===")
print(f"Is connected: {nx.is_connected(G_Undirected)}")
if not nx.is_connected(G_Undirected):
    print(f"Number of connected components: {nx.number_connected_components(G_Undirecte
    print(f"Size of largest component: {len(max(nx.connected_components(G_Undirected),
    print(f"Size of smallest component: {len(min(nx.connected_components(G_Undirected)

    # Giant component analysis
    giant = G_Undirected.subgraph(max(nx.connected_components(G_Undirected), key=len))
    print(f"\nGiant component properties:")
    print(f"  Nodes: {giant.number_of_nodes()}")
    print(f"  Edges: {giant.number_of_edges()}")
    print(f"  Average degree: {2*giant.number_of_edges()/giant.number_of_nodes():.2f}"

# Average shortest path length (for connected graphs or giant component)
try:
    if nx.is_connected(G_Undirected):
        print(f"\nAverage shortest path length: {nx.average_shortest_path_length(G_Und
```

```
    else:
        print(f"\nAverage shortest path length (giant component): {nx.average_shortest_
except nx.NetworkXError:
    print("Graph is too large to compute shortest paths")

# Clustering coefficient
print(f"Average clustering coefficient: {nx.average_clustering(G_Undirected):.8f}")
```

```
=== Degree Analysis ===
Number of nodes: 62142
Number of edges: 3429
Average degree: 0.11
Median degree: 0.00
Maximum degree: 143
Minimum degree: 0
Density: 0.0000
```




```
=== Connectivity Analysis ===
Is connected: False
Number of connected components: 58713
Size of largest component: 150
Size of smallest component: 1

Giant component properties:
  Nodes: 150
  Edges: 149
  Average degree: 1.99

Average shortest path length (giant component): 2.06
Average clustering coefficient: 0.00000000
```

In [ ]:

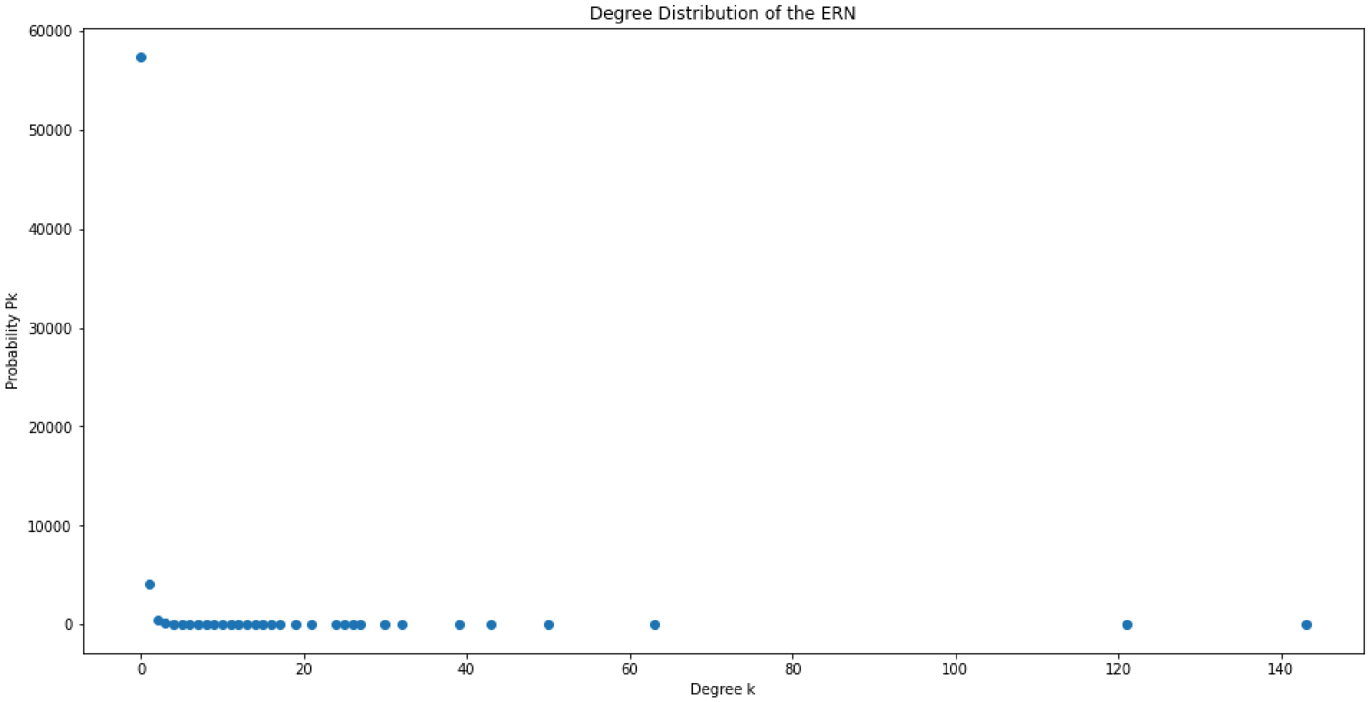
```
def Calculate_Network_Degree(european_undirected):

    node_degree_values = european_undirected.degree()
    weighted_node_degree_values = european_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_value

    return average_degree, weighted_average_degree, node_degree_values, weighted_node_(

G_Undirected = G.to_undirected()
average_degree, weighted_average_degree, node_degree_values, weighted_node_degree_value

print("The Average Degree of the European Railway Network: ", average_degree)
print("\nThe Weighted Average Degree of the European Railway Network: ", weighted_avera

print("The 5 stations having the highest degree (that are directly accessible the most)
highest_degree = sorted(G_Undirected.degree, key=lambda x: x[1], reverse=True)[:5]
highest_degree_nodes = [x for (x, y) in highest_degree]
highest_degree_values = [y for (x, y) in highest_degree]

for k in range(len(highest_degree_nodes)):
    highest_degree_node = highest_degree_nodes[k]
```

```
    highest_degree_station = df_cleaned.loc[df_cleaned['id'] == highest_degree_node, 'n
    print(highest_degree_station,  highest_degree_values[k])


print("\n\nThe 5 stations having highest degree (that are directly accessible the most)
highest_weighted_degree = sorted(G_Undirected.degree(weight='weight'), key=lambda x: x
highest_weighted_degree_nodes = [x for (x, y) in highest_weighted_degree]
highest_weighted_degree_values = [y for (x, y) in highest_weighted_degree]

for k in range(len(highest_weighted_degree_nodes)):
    highest_weighted_degree_node = highest_weighted_degree_nodes[k]
    highest_weighted_degree_station = df_cleaned.loc[df_cleaned['id'] == highest_weigh
    print(highest_weighted_degree_station,  highest_weighted_degree_values[k])
```

```
The Average Degree of the European Railway Network:  0.11036014289852274

The Weighted Average Degree of the European Railway Network:  0.5245293903781477
The 5 stations having the highest degree (that are directly accessible the most) by dif
ferent stations in the European Railway Network are:
Berlin 143
Göteborg 121
Hamburg 63
Wien 50
München 43


The 5 stations having highest degree (that are directly accessible the most) in the Eur
opean Railway Network considering weighted degree distribution are:
Loulé 1734.7779952405094
Luzern Bahnhofquai 1734.7779952405094
Kłodzko Główne 1343.5231747902178
Bessines 1343.5231747902178
Berlin 1237.1220974957348
```

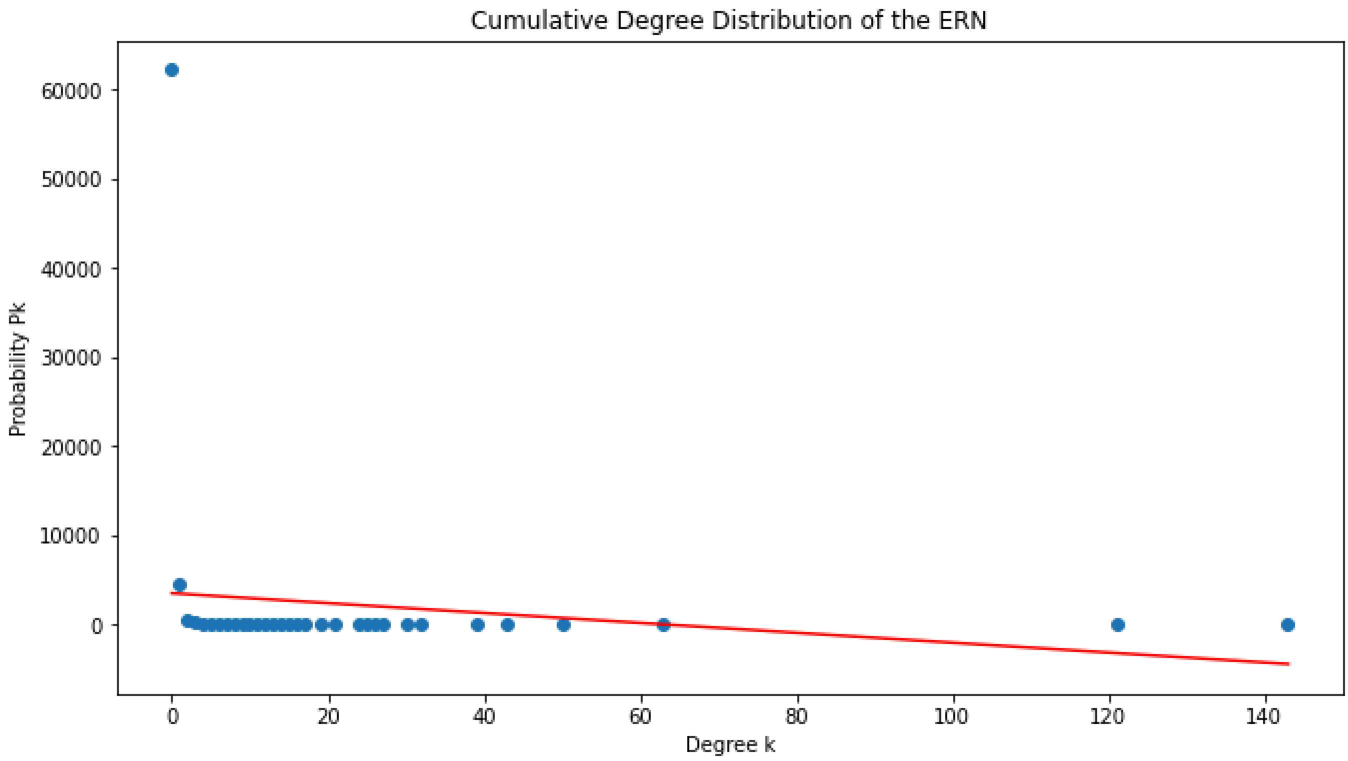
In [158…

```
def Calculate_Degree_Distribution(european_undirected):

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

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

    plt.figure(figsize=(16, 8))
    plt.scatter(degree, count)

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

    plt.title("Degree Distribution of the ERN")

    plt.show()
Calculate_Degree_Distribution(G_Undirected)
```



In [159…

```
def Calculate_Cumulative_Degree_Distribution(european_undirected):

    degree_sequence = sorted([d for n, d in european_undirected.degree()])
    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("Probability 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 ERN")

    plt.show()
Calculate_Cumulative_Degree_Distribution(G_Undirected)
```

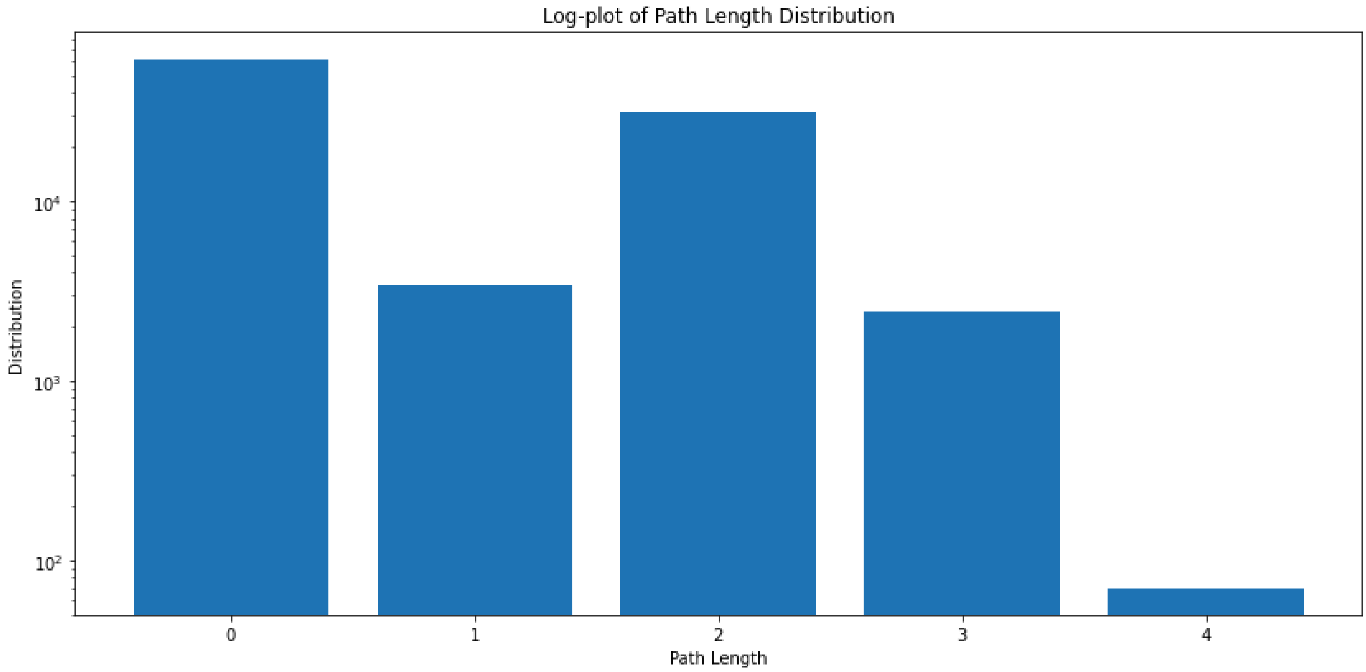


In [167…

```
def analyze_path_length(european_undirected):

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

    for node_path_lengths in 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_Lengths = analyze_path_length(G_Undirected)
def Calculate_Path_Length_Distribution(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 = Calculate_Path_Length_Distributio
path_length_distribution = {shortest_path_lengths[i]: shortest_path_length_values[i] fo
print(path_length_distribution, "\n")

plt.figure(figsize=(14, 6.5))
plt.bar(path_length_distribution.keys(), path_length_distribution.values())
plt.xlabel("Path Length")
plt.ylabel("Distribution")
plt.yscale('log')

plt.title("Log-plot of Path Length Distribution")
```

```
{0: 62142, 1: 3429, 2: 31131, 3: 2419, 4: 70}
```

Out[167… Text(0.5, 1.0, 'Log-plot of Path Length Distribution')



In [169…

```
average_path_length_sum = 0
Diameter = []
Path_Length = []
count = 0
for C in (G_Undirected.subgraph(c).copy() for c in nx.connected_components(G_Undirected
    if(nx.average_shortest_path_length(C) > 0):
      count = count + 1
      average_path_length_sum = average_path_length_sum + nx.average_shortest_path_leng
    Diameter.append(nx.diameter(C))
    Path_Length.append(nx.average_shortest_path_length(C))
average_path_length = average_path_length_sum / count
average_diameter = np.sum(Diameter) / count

print("The average path length of Indain Railway Network: ", average_path_length)
print("The Diameter of the Indian Railway Network: ",average_degree)
```

```
The average path length of Indain Railway Network:  1.2330580950669778
The Diameter of the Indian Railway Network:  0.11036014289852274
```

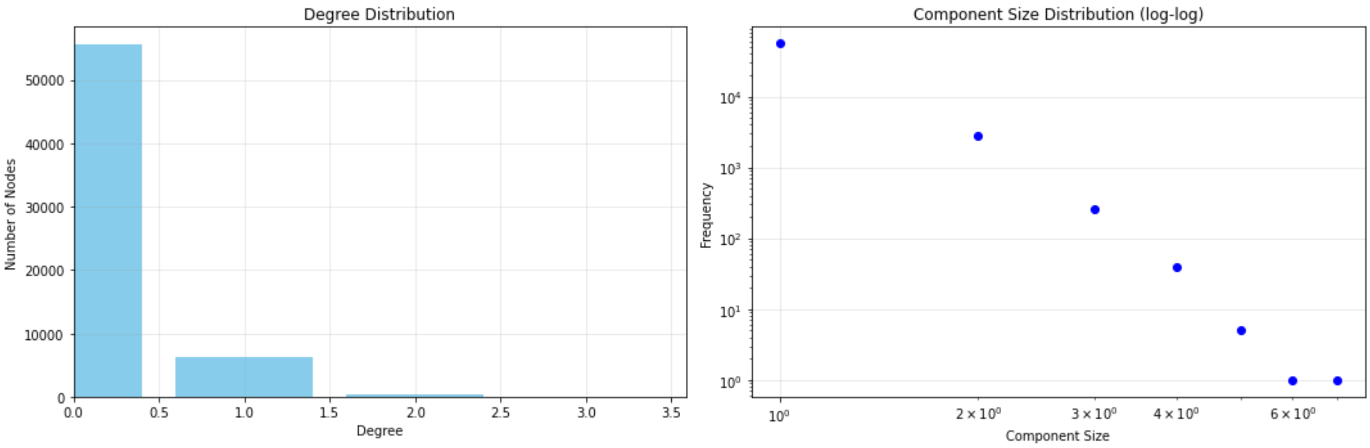
In [177…

```
import networkx as nx
import random
import matplotlib.pyplot as plt
from collections import Counter
from tqdm import tqdm  # For progress bars
import numpy as np

# ======================
# PARAMETERS
# ======================
n = 62142  # Number of nodes
target_edges = 3429  # Target number of edges
seed = 42  # Random seed for reproducibility

# ======================
# GRAPH GENERATION
# ======================
print("Generating Erdős-Rényi graph...")

# Method 1: Probabilistic ER model (faster but edge count may vary slightly)
p = target_edges / (n * (n - 1) / 2)  # Edge probability
G = nx.erdos_renyi_graph(n, p, seed=seed, directed=False)

# Method 2: Exact edge count (slower but precise) - uncomment to use
"""
```

```python
G = nx.empty_graph(n)
all_possible_edges = [(i, j) for i in range(n) for j in range(i+1, n)]
random.seed(seed)
edges = random.sample(all_possible_edges, target_edges)
with tqdm(total=target_edges, desc="Adding edges") as pbar:
    for edge in edges:
        G.add_edge(*edge)
        pbar.update(1)
"""

# ======================
# ANALYSIS
# ======================
print("\nAnalyzing graph properties...")

# Basic properties
actual_edges = G.number_of_edges()
density = nx.density(G)
avg_degree = 2 * actual_edges / n

print(f"Nodes: {n}")
print(f"Edges: {actual_edges} (target: {target_edges})")
print(f"Density: {density:.2e}")
print(f"Average degree: {avg_degree:.4f}")

# Connectivity analysis
is_connected = nx.is_connected(G)
components = list(nx.connected_components(G))

print(f"\nConnectivity:")
print(f"Connected: {'Yes' if is_connected else 'No'}")
print(f"Number of components: {len(components)}")
print(f"Largest component: {len(max(components, key=len))} nodes")
print(f"Smallest component: {len(min(components, key=len))} nodes")

# Degree distribution
degrees = [d for _, d in G.degree()]
degree_counts = Counter(degrees)

# ======================
# VISUALIZATION
# ======================
print("\nGenerating visualizations...")

plt.figure(figsize=(15, 5))

# Degree histogram
plt.subplot(1, 2, 1)
plt.bar(degree_counts.keys(), degree_counts.values(), color='skyblue')
plt.title("Degree Distribution")
plt.xlim(left=0)
plt.xlabel("Degree")
plt.ylabel("Number of Nodes")
plt.grid(True, alpha=0.3)

# Component size distribution (log scale)
plt.subplot(1, 2, 2)
component_sizes = [len(c) for c in components]
size_counts = Counter(component_sizes)
plt.loglog(size_counts.keys(), size_counts.values(), 'bo')
plt.title("Component Size Distribution (log-log)")
plt.xlabel("Component Size")
plt.ylabel("Frequency")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# ======================
# SAVE RESULTS
```

```
# =======================
output_file = f"ER_graph_{n}n_{actual_edges}e.gml"
nx.write_gml(G, output_file)
print(f"\nGraph saved to {output_file}")
```

```
Generating Erdős-Rényi graph...

Analyzing graph properties...
Nodes: 62142
Edges: 3468 (target: 3429)
Density: 1.80e-06
Average degree: 0.1116

Connectivity:
Connected: No
Number of components: 58674
Largest component: 7 nodes
Smallest component: 1 nodes

Generating visualizations...
```

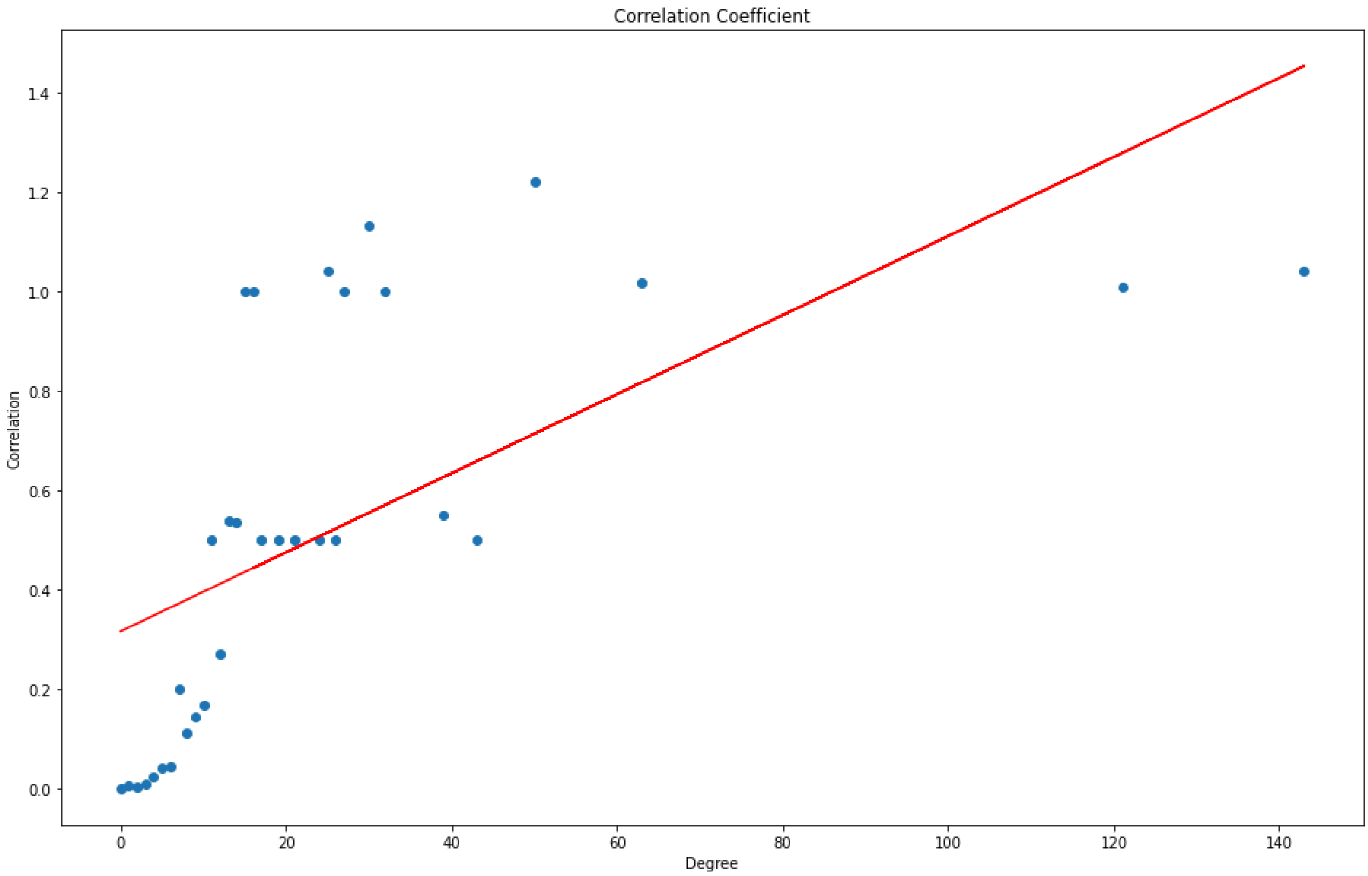



```
Graph saved to ER_graph_62142n_3468e.gml
```

In [179…

```
def Calculate_Degree_Correlation(european_undirected):

    node_degree_values = european_undirected.degree()
    unique_degrees = list(set([y for (x,y) in node_degree_values]))
    Degree_Correlation = {}

    for degree in unique_degrees:

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

        final_degree_sum = 0

        for node in nodes_kdegree:

            Neighbours = []

            for n in european_undirected.neighbors(node):
              Neighbours.append(n)

            degree_sum = 0
            for neighbour in Neighbours:
                degree_sum = degree_sum + european_undirected.degree[neighbour]

            if(len(Neighbours) > 0):
                node_average_degree = degree_sum / len(Neighbours)
            else:
                node_average_degree = 0

        final_degree_sum = final_degree_sum + node_average_degree

        correlation = final_degree_sum / count_nodes_kdegree
        #print(correlation)
```

```
        Degree_Correlation[degree] = correlation

    return Degree_Correlation
Degree_Correlation = Calculate_Degree_Correlation(G_Undirected)
Degree_Values = (list(Degree_Correlation.keys()))
Degree_Correlation_Values = (list(Degree_Correlation.values()))

plt.figure(figsize=(16, 10))

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 [181…

```
assortativity_coefficient = nx.degree_pearson_correlation_coefficient(G)
print("The Assotativity Coefficient of the European Railway Network: ", assortativity_c
```

```
The Assotativity Coefficient of the European Railway Network:  0.021195065159220657
```

MOTIF ANALYSIS

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

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} secon

    # 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

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({
            'triad_id': triad_id,
            'original_count': original_count,
            'mean_random': mean_random,
            'std_random': std_random,
            'z_score': z_score
        })
```

```python
    # 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']}, 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
```

```python
        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(G, num_random=5)  # Reduced number for efficien

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

# motifs, anti_motifs, neutral = classify_triads(results)
```

```
Enumerating connected triads...
Processed 50/62142 nodes in 0.01 seconds
Processed 100/62142 nodes in 0.01 seconds
Processed 150/62142 nodes in 0.01 seconds
Processed 200/62142 nodes in 0.01 seconds
Processed 250/62142 nodes in 0.01 seconds
Processed 300/62142 nodes in 0.01 seconds
Processed 350/62142 nodes in 0.01 seconds
Processed 400/62142 nodes in 0.01 seconds
Processed 450/62142 nodes in 0.01 seconds
Processed 500/62142 nodes in 0.01 seconds
Processed 550/62142 nodes in 0.01 seconds
Processed 600/62142 nodes in 0.01 seconds
Processed 650/62142 nodes in 0.01 seconds
Processed 700/62142 nodes in 0.01 seconds
Processed 750/62142 nodes in 0.01 seconds
Processed 800/62142 nodes in 0.01 seconds
Processed 850/62142 nodes in 0.01 seconds
Processed 900/62142 nodes in 0.02 seconds
Processed 950/62142 nodes in 0.02 seconds
Processed 1000/62142 nodes in 0.02 seconds
Processed 1050/62142 nodes in 0.02 seconds
Processed 1100/62142 nodes in 0.02 seconds
Processed 1150/62142 nodes in 0.02 seconds
Processed 1200/62142 nodes in 0.02 seconds
Processed 1250/62142 nodes in 0.02 seconds
Processed 1300/62142 nodes in 0.02 seconds
Processed 1350/62142 nodes in 0.02 seconds
```

```
Processed 55950/62142 nodes in 3.51 seconds
Processed 56000/62142 nodes in 3.51 seconds
Processed 56050/62142 nodes in 3.51 seconds
Processed 56100/62142 nodes in 3.51 seconds
Processed 56150/62142 nodes in 3.51 seconds
Processed 56200/62142 nodes in 3.52 seconds
Processed 56250/62142 nodes in 3.52 seconds
Processed 56300/62142 nodes in 3.52 seconds
Processed 56350/62142 nodes in 3.53 seconds
Processed 56400/62142 nodes in 3.53 seconds
Processed 56450/62142 nodes in 3.53 seconds
Processed 56500/62142 nodes in 3.53 seconds
Processed 56550/62142 nodes in 3.53 seconds
Processed 56600/62142 nodes in 3.53 seconds
Processed 56650/62142 nodes in 3.53 seconds
Processed 56700/62142 nodes in 3.53 seconds
Processed 56750/62142 nodes in 3.53 seconds
Processed 56800/62142 nodes in 3.53 seconds
Processed 56850/62142 nodes in 3.53 seconds
Processed 56900/62142 nodes in 3.54 seconds
Processed 56950/62142 nodes in 3.54 seconds
Processed 57000/62142 nodes in 3.54 seconds
Processed 57050/62142 nodes in 3.54 seconds
Processed 57100/62142 nodes in 3.54 seconds
Processed 57150/62142 nodes in 3.54 seconds
Processed 57200/62142 nodes in 3.54 seconds
Processed 57250/62142 nodes in 3.54 seconds
Processed 57300/62142 nodes in 3.54 seconds
Processed 57350/62142 nodes in 3.54 seconds
Processed 57400/62142 nodes in 3.54 seconds
Processed 57450/62142 nodes in 3.54 seconds
Processed 57500/62142 nodes in 3.54 seconds
Processed 57550/62142 nodes in 3.54 seconds
Processed 57600/62142 nodes in 3.54 seconds
Processed 57650/62142 nodes in 3.54 seconds
Processed 57700/62142 nodes in 3.54 seconds
Processed 57750/62142 nodes in 3.54 seconds
Processed 57800/62142 nodes in 3.54 seconds
Processed 57850/62142 nodes in 3.54 seconds
Processed 57900/62142 nodes in 3.54 seconds
Processed 57950/62142 nodes in 3.54 seconds
Processed 58000/62142 nodes in 3.54 seconds
Processed 58050/62142 nodes in 3.54 seconds
Processed 58100/62142 nodes in 3.54 seconds
Processed 58150/62142 nodes in 3.54 seconds
Processed 58200/62142 nodes in 3.54 seconds
Processed 58250/62142 nodes in 3.55 seconds
Processed 58300/62142 nodes in 3.55 seconds
Processed 58350/62142 nodes in 3.55 seconds
Processed 58400/62142 nodes in 3.55 seconds
Processed 58450/62142 nodes in 3.55 seconds
Processed 58500/62142 nodes in 3.55 seconds
Processed 58550/62142 nodes in 3.55 seconds
Processed 58600/62142 nodes in 3.55 seconds
Processed 58650/62142 nodes in 3.55 seconds
Processed 58700/62142 nodes in 3.55 seconds
Processed 58750/62142 nodes in 3.55 seconds
Processed 58800/62142 nodes in 3.55 seconds
Processed 58850/62142 nodes in 3.55 seconds
Processed 58900/62142 nodes in 3.55 seconds
Processed 58950/62142 nodes in 3.55 seconds
Processed 59000/62142 nodes in 3.55 seconds
Processed 59050/62142 nodes in 3.55 seconds
Processed 59100/62142 nodes in 3.55 seconds
Processed 59150/62142 nodes in 3.55 seconds
Processed 59200/62142 nodes in 3.55 seconds
Processed 59250/62142 nodes in 3.55 seconds
Processed 59300/62142 nodes in 3.55 seconds
Processed 59350/62142 nodes in 3.55 seconds
Processed 59400/62142 nodes in 3.55 seconds
Processed 59450/62142 nodes in 3.55 seconds
Processed 59500/62142 nodes in 3.55 seconds
Processed 59550/62142 nodes in 3.55 seconds
Processed 59600/62142 nodes in 3.55 seconds
```

```
Processed 59650/62142 nodes in 3.55 seconds
Processed 59700/62142 nodes in 3.55 seconds
Processed 59750/62142 nodes in 3.55 seconds
Processed 59800/62142 nodes in 3.55 seconds
Processed 59850/62142 nodes in 3.55 seconds
Processed 59900/62142 nodes in 3.55 seconds
Processed 59950/62142 nodes in 3.55 seconds
Processed 60000/62142 nodes in 3.55 seconds
Processed 60050/62142 nodes in 3.55 seconds
Processed 60100/62142 nodes in 3.56 seconds
Processed 60150/62142 nodes in 3.56 seconds
Processed 60200/62142 nodes in 3.56 seconds
Processed 60250/62142 nodes in 3.56 seconds
Processed 60300/62142 nodes in 3.56 seconds
Processed 60350/62142 nodes in 3.56 seconds
Processed 60400/62142 nodes in 3.56 seconds
Processed 60450/62142 nodes in 3.56 seconds
Processed 60500/62142 nodes in 3.56 seconds
Processed 60550/62142 nodes in 3.56 seconds
Processed 60600/62142 nodes in 3.56 seconds
Processed 60650/62142 nodes in 3.56 seconds
Processed 60700/62142 nodes in 3.56 seconds
Processed 60750/62142 nodes in 3.56 seconds
Processed 60800/62142 nodes in 3.56 seconds
Processed 60850/62142 nodes in 3.56 seconds
Processed 60900/62142 nodes in 3.56 seconds
Processed 60950/62142 nodes in 3.56 seconds
Processed 61000/62142 nodes in 3.56 seconds
Processed 61050/62142 nodes in 3.56 seconds
Processed 61100/62142 nodes in 3.56 seconds
Processed 61150/62142 nodes in 3.56 seconds
Processed 61200/62142 nodes in 3.56 seconds
Processed 61250/62142 nodes in 3.56 seconds
Processed 61300/62142 nodes in 4.62 seconds
Processed 61350/62142 nodes in 4.62 seconds
Processed 61400/62142 nodes in 4.62 seconds
Processed 61450/62142 nodes in 4.62 seconds
Processed 61500/62142 nodes in 4.62 seconds
Processed 61550/62142 nodes in 4.62 seconds
Processed 61600/62142 nodes in 4.62 seconds
Processed 61650/62142 nodes in 4.62 seconds
Processed 61700/62142 nodes in 4.62 seconds
Processed 61750/62142 nodes in 4.62 seconds
Processed 61800/62142 nodes in 4.62 seconds
Processed 61850/62142 nodes in 4.62 seconds
Processed 61900/62142 nodes in 4.62 seconds
Processed 61950/62142 nodes in 4.62 seconds
Processed 62000/62142 nodes in 4.62 seconds
Processed 62050/62142 nodes in 4.62 seconds
Processed 62100/62142 nodes in 4.62 seconds
Processed 62142/62142 nodes in 4.62 seconds
Found 3 unique connected triad patterns
Triad counting for random network 5 completed in 4.62 seconds

Automatically determined Z-score thresholds:
Motif threshold: > 1.41, Anti-motif threshold: < -1.41
```

In [19]:

```python
print_results(motifs, anti_motifs, neutral, upper_thresh, lower_thresh)

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

print('''\nINFERENCE:\nWe are getting only 3 triads because graph is very sparse (Avera
```

```
===== AUTOMATIC MOTIF ANALYSIS =====
Motif Z-score threshold: > 1.41
Anti-motif Z-score threshold: < -1.41

=== TOP 1 MOTIFS (Z > 1.41) ===
Motif 1: Z = 1.57, Count = 7, Random Mean = 5.40

=== TOP 1 ANTI-MOTIFS (Z < -1.41) ===
Anti-motif 1: Z = -1.57, Count = 5, Random Mean = 6.60
```

```
Neutral triads: 1 (Z between -1.41 and 1.41)

===== SUMMARY TABLE =====
         Type   Z-score  Original Count  Random Mean  Standard Deviation
0       Motif  1.568929               7          5.4            1.019804
1  Anti-motif -1.568929               5          6.6            1.019804
Results saved to motif_analysis_results.csv

Total execution time: 552.01 seconds
```

INFERENCE:
We are getting only 3 triads because graph is very sparse (Average degree is 0.11) and hence only 1 Motif and 1-Anti motif is detected