London Tube

Simon Schmetz - Paola Carolina Suarez
Network Analysis
Master in Statistics for Data Science
Universidad Carlos III de Madrid

Data Source: https://github.com/jaron/railgraph

Introduction

The "London Underground Network" refers to the metro system of London, UK, encompassing its stations and the connections between them. The network consists of 309 vertices (representing stations), and 370 edges (representing the metro line connections between stations). Notably, multiple metro lines can connect to the same station, resulting in multiple edges. As an undirected network, each edge signifies a bidirectional connection, meaning travel is possible in both directions between stations.

Data loading and cleanning

We beginn by loading the Network and enriching the nodes with usage data from a seperate dataset.

In [1]:
```
# Load Libraries
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import contextily as ctx
import scipy
```

We begin by loading the Network and extracting the labels of the Stations for later Use.

In [2]:
```
#Read the Network
G = nx.read_graphml('data/london_tube/tubeDLR.graphml')

# Extract labels from node data
labels = nx.get_node_attributes(G, 'label')

# get node position seperately
pos_geo = {node: (float(data['longitude']), float(data['latitude'])
```

We remove node attributes that were delivered with the network.

In [3]:
```
# List of attributes to remove
attributes_to_remove = [
    'Betweenness Centrality', 'Closeness Centrality', 'Clustering C
    'Degree', 'Eccentricity', 'Eigenvector Centrality', 'In-Degree'
    'Number of triangles', 'Out-Degree', 'PageRank', 'b','g',
    'latitude_copy', 'longitude_copy', 'r', 'size', 'x', 'y','displ
]

# Remove the attributes from each node
for node, data in G.nodes(data=True):
    for attr in attributes_to_remove:
        data.pop(attr, None)
```

To further enrich our Network, we take a seperate Dataset that contains average entry and exit values from most of the stations in our Network and add theses to our nodes. To do so, we first remove all stations for which no numbers are available, which as it turns our all correspond to the Docklands Light Railway (DLR) and thus can be knowingly discarded.

In [4]:
```
### Add Usage Data

usage_data = pd.read_csv('data/london_tube/2017_Entry_Exit.csv')

# Feature Engineering
usage_data['Avg_Exit_Weekend'] = usage_data[['Exit_Saturday', 'Exit_
usage_data['Avg_Entry_Weekend'] = usage_data[['Entry_Saturday', 'En

usage_data.drop(columns=['Entry_Saturday', 'Entry_Sunday', 'Exit_Sa

# Define mapping from non-standard to standard names
station_name_corrections = {
    'Edgware Road (Cir)': 'Edgware Road',
    'Kew Gardens': 'Kew Gardens (London)',
    "Shepherd's Bush (H&C)": "Shepherd's Bush Market",
    "Shepherd's Bush (Cen)": "Shepherd's Bush",
    'Hammersmith (Dis)': 'Hammersmith (Piccadilly and District line
    'Hammersmith (H&C)': 'Hammersmith (Hammersmith & City and Circl
    "St. John's Wood": "St John's Wood",
    'Bank & Monument': 'Bank-Monument',
    'Totteridge & Whetstone': 'Totteridge and Whetstone',
    'Paddington': 'London Paddington',
    'Richmond': 'Richmond (London)',
    'Heathrow Terminals 123': 'Heathrow Terminals 1, 2, 3',
    'Edgware Road (Bak)': 'Edgware Road (Bakerloo line)',
    'Victoria': 'London Victoria'
}

# Apply corrections to df.Stations
usage_data['Station'] = usage_data['Station'].replace(station_name_


# Filter nodes in the graph that are available in usage_data
```

```
stations_in_usage_data = set(usage_data['Station'])
nodes_to_keep = [node for node, data in G.nodes(data=True) if data.

# Identify stations to be dropped
stations_to_drop = [labels[node] for node, data in G.nodes(data=Tru



# Create a subgraph with only the nodes to keep
G = G.subgraph(nodes_to_keep).copy()

# Add Entry_Week, Exit_Week, Avg_Exit_Weekend, Avg_Entry_Weekend to
for node, data in G.nodes(data=True):
    station_label = data.get('label')
    if station_label in stations_in_usage_data:
        station_data = usage_data[usage_data['Station'] == station_
        data['Entry_Week'] = station_data['Entry_Week']
        data['Exit_Week'] = station_data['Exit_Week']
        data['Avg_Exit_Weekend'] = station_data['Avg_Exit_Weekend']
        data['Avg_Entry_Weekend'] = station_data['Avg_Entry_Weekend

        # Remove edges connected to nodes that are not in the filte
        G.remove_edges_from([(u, v) for u, v in G.edges() if u not
```

Then is added the Euclidean Distance between Nodes as a attribute of the arcs using longitude/latitude of the nodes.

In [5]:
```
# Add "dist" attribute to each edge based on the euclidean distance
for u, v, data in G.edges(data=True):
    lon1, lat1 = pos_geo[u]
    lon2, lat2 = pos_geo[v]
    distance = np.sqrt((lon2 - lon1)**2 + (lat2 - lat1)**2) * 111
    data['dist'] = distance
```

This yields us the Network with the following attributes of nodes and edges.

In [6]:
```
# Attributes

# Nodes
unique_keys = set().union(*(data.keys() for _, data in G.nodes(data=
print("Unique Node Attributes:")
for key in sorted(unique_keys):
    print(f" - {key}")

# Edges
unique_edge_keys = set().union(*(data.keys() for _, _, data in G.edg
print("\nUnique Edge Attributes:")
for key in sorted(unique_edge_keys):
    print(f" - {key}")
```

```
Unique Node Attributes:
 - Avg_Entry_Weekend
 - Avg_Exit_Weekend
 - Entry_Week
 - Exit_Week
 - displayName
 - label
 - latitude
 - longitude
 - stationReference

Unique Edge Attributes:
 - Neo4j Relationship Type
 - dist
 - line
 - weight
```

Utils functions

In [7]:

```python
def plot_shortest_path(G, start_id, stop_id):

    """
    Plot the shortest path between two nodes in a graph.
    """

    # Calculate shortest path
    try:
        shortest_path = nx.dijkstra_path(G, source=start_id, target=
    except nx.NetworkXNoPath:
        print(f"No path exists between {start_id} and {stop_id}.")
        return

    # Extract edges on shortest path
    path_edges = list(zip(shortest_path[:-1], shortest_path[1:]))

    # Plot
    fig, ax = plt.subplots(figsize=(25, 25))  # More balanced size

    ax.set_xlim(-0.68, 0.29) # london coords
    ax.set_ylim(51.39, 51.73)

    ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))  # Ad

    # map and network
    ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.Cartol
    nx.draw(G, pos=pos_geo, with_labels=False, node_size=50, node_c
            edge_color='lightgray', ax=ax)

    # shortest path
    nx.draw_networkx_edges(G, pos=pos_geo, edgelist=path_edges, edg
    nx.draw_networkx_nodes(G, pos=pos_geo, nodelist=shortest_path,

    # start and stop
    nx.draw_networkx_labels(G, pos=pos_geo, labels={start_id: label
```

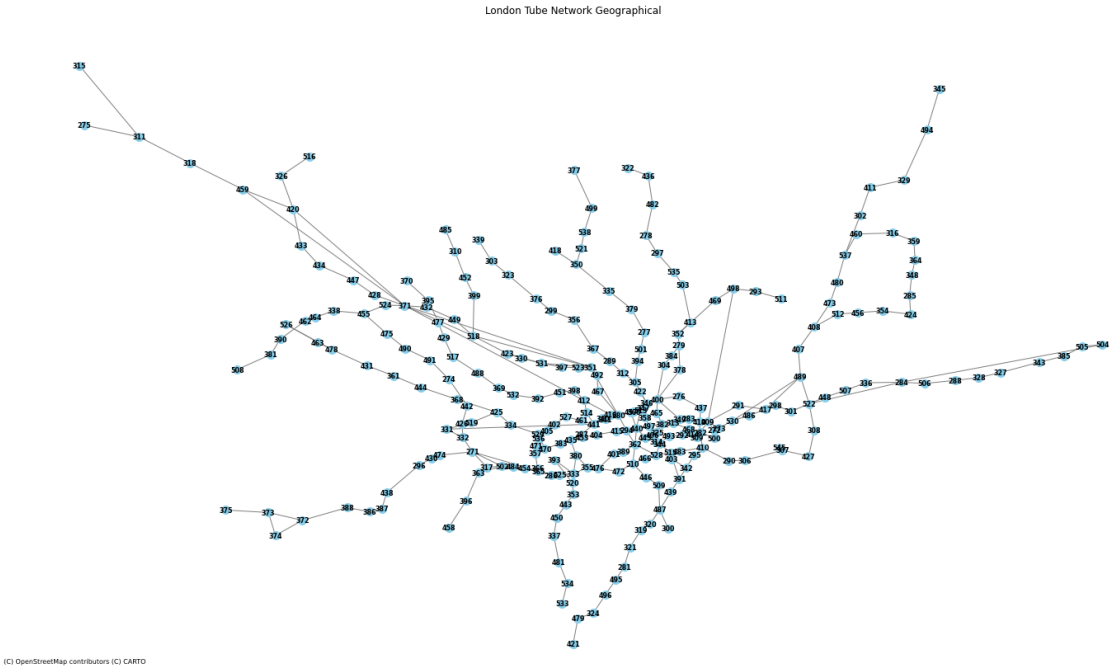
```
    plt.title(f"Shortest Path from {labels[start_id]} to {labels[st
    plt.show()
```

In [8]:
```
def plot_highlighted_stations(G, pos_geo, station_nodes, labels):
    """
    plot network with highlighted stations
    """

    # Plot the network
    plt.figure(figsize=(12, 8))
    nx.draw(G, pos=pos_geo, with_labels=False, node_size=10, node_c
    nx.draw_networkx_nodes(G, pos=pos_geo, nodelist=station_nodes,

    # Add labels
    nx.draw_networkx_labels(G, pos=pos_geo, labels={node: labels[no

    # cosmetics
    plt.legend(scatterpoints=1, loc='upper left', fontsize=10)
    plt.title("London Tube Network with Highlighted Stations")
    plt.show()
```

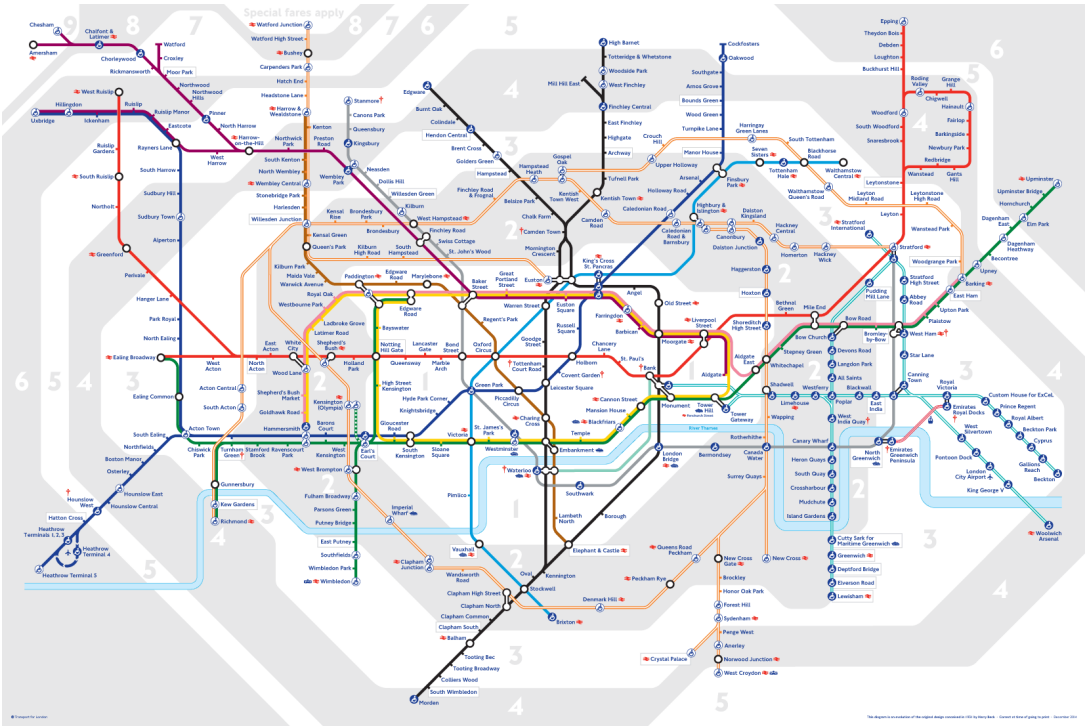
Network properties

We beginn by analysing basic properties of the network to get a general idea of the characteristics of our network. We start by plotting the network with the node position corresponding to the longitude and latitude values onto a map of the City of London to get a general feel for the geography underlying the network. We see how the Network consist out of arms corresponding to the metro lines that meet in the city center at hub stations where changes between lines are possible.

In [9]:
```
# Plot network with map of the city of London
fig, ax = plt.subplots(figsize=(25, 25))  # More balanced size

# Add map of London
ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.CartoDB.P

# Overlay Network
nx.draw(G, pos=pos_geo, with_labels=True, node_size=100, node_color
        font_size=8, font_weight='bold', edge_color='gray', ax=ax)

# Cosmetics and plot
ax.set_xlim(-0.68, 0.29)
ax.set_ylim(51.39, 51.73)
ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))  # Adjust

plt.title("London Tube Network Geographical")
plt.show()
```



We can compare our Network to an official map of the City of London Transportation System and recognize the same tree structure with nodes that connect the trees (lines).



The London Underground metro system, after data cleaning, can be represented as an undirected network with 269 nodes (stations) and 324 edges (connections between stations).

In [10]:
```
# Check if the graph is directed
if isinstance(G, nx.DiGraph):
    print("The graph is directed.")
else:
    print("The graph is undirected.")
```

```
#Count nodes
print("Number of Nodes (Order):", G.number_of_nodes())

#Count edges
print("Number of Edges:", len(G.edges(data=True)))
```

```
The graph is undirected.
Number of Nodes (Order): 269
Number of Edges: 324
```

Then, it is checked if there are any isolated vertices (stations) in the network.It is found the isolated vertex '545' in the London Underground network, whichc refers to a station that has no connections to any other stations, making it unreachable within the system. In this case, the isolated station is produced as a error of the initial enrichment of with usage data wich left the node isolated. We thus delete the node to correct for that error.

In [11]:
```
# Find isolated vertices
print("Isolated Vertices: ",list(nx.isolates(G)))
print(labels["545"])

# Remove node '545' from labels and the graph
labels.pop("545", None)
G.remove_node("545")

print("Isolated Vertices: ",list(nx.isolates(G)))
```

```
Isolated Vertices:  ['545']
Canary Wharf
Isolated Vertices:  []
```

In network analysis, specifically in the context of the London Underground metro system, it's important to identify adjacent vertices (stations) and edges (metro lines). This helps to understand the connections between stations and the metro lines serving them. The following code allows us to find these adjacent vertices, revealing the connections between different metro lines at a given station. Additionally, the adjacent edges provide information about possible line changes that a station can offer, indicating where passengers can switch between different metro lines.

In [12]:
```
# Get the adjacent vertices
chosen_vertex = '487'

adjacent_vertices = list(G.neighbors(chosen_vertex))
print(f"Adjacent vertices of vertex {chosen_vertex}: {adjacent_vert

# Print the labels of the adjacent vertices
adjacent_labels = [labels[vertex] for vertex in adjacent_vertices]
print(f"Adjacent vertices of vertex {labels[chosen_vertex]}: {adjac
```

```
Adjacent vertices of vertex 487: ['300', '320', '439', '509']
Adjacent vertices of vertex Stockwell: ['Brixton', 'Clapham North',
'Oval', 'Vauxhall']
```

In [13]:
```
# Get the adjacent edges (edges that involve the chosen vertex)
adjacent_edges = G.edges(chosen_vertex)
print(f"Adjacent edges of vertex {chosen_vertex}: {adjacent_edges}"

# Print the details of the adjacent edges, including the metro line
for edge in adjacent_edges:
    # Assuming the edge attribute 'line' stores the metro line's na
    line_name = G[edge[0]][edge[1]].get('line', 'Unknown Line')  # 
    print(f"Edge: {edge} (Connecting {labels[edge[0]]} and {labels[
```

```
Adjacent edges of vertex 487: [('487', '300'), ('487', '320'), ('48
7', '439'), ('487', '509')]
Edge: ('487', '300') (Connecting Stockwell and Brixton) - Metro Lin
e: Victoria_line
Edge: ('487', '320') (Connecting Stockwell and Clapham North) - Metr
o Line: Northern_line
Edge: ('487', '439') (Connecting Stockwell and Oval) - Metro Line: N
orthern_line
Edge: ('487', '509') (Connecting Stockwell and Vauxhall) - Metro Lin
e: Victoria_line
```

The presence of self-loops in the network was examined, and it was determined that the system does not contain any self-loops. This finding is consistent with the design of most metro systems, where a station typically does not have a direct connection to itself via a metro line. Self-loops are generally avoided in urban transit networks for practical reasons, as they would not contribute to effective transportation or improve connectivity.

In [14]:
```
# Check for self-loops by iterating through the edges
self_loops = [edge for edge in G.edges() if edge[0] == edge[1]]

# If there are any self-loops, print them
if self_loops:
    print(f"The graph has {len(self_loops)} self-loop(s):")
    for loop in self_loops:
        print(loop)
else:
    print("The graph does not have any self-loops.")
```

```
The graph does not have any self-loops.
```

Similarly, it was determined that the network does not contain any multi-edges, which means there are no multiple metro lines connecting the same pair of stations. This is consistent with the design principles of the metro system, where each pair of stations typically has a unique connection served by a single metro line. Having multiple lines between the same two stations is not a common practice, as it can lead to inefficiencies in the network design and complicate operations.

In [15]:
```
# Check if the graph allows multi-edges (MultiGraph or MultiDiGraph
if isinstance(G, nx.MultiGraph) or isinstance(G, nx.MultiDiGraph):
    # Iterate over the edges and check for multi-edges
    multi_edges = [edge for edge in G.edges()]
```

```
    if multi_edges:
        print(f"The graph has multi-edges between the following nod
        for edge in multi_edges:
            print(edge)
    else:
        print("The graph does not have any multi-edges.")
else:
    print("The graph is not a MultiGraph or MultiDiGraph, so multi-
```

```
The graph is not a MultiGraph or MultiDiGraph, so multi-edges are no
t allowed.
```

Subnetworks

With the basic properties of the network defined, we continue with looking at subnetworks and components that can be found in the network. From the official transportation map shown above, we can further derive some of the main subnetworks corresponding to individual metro, light rail or regional train lines.

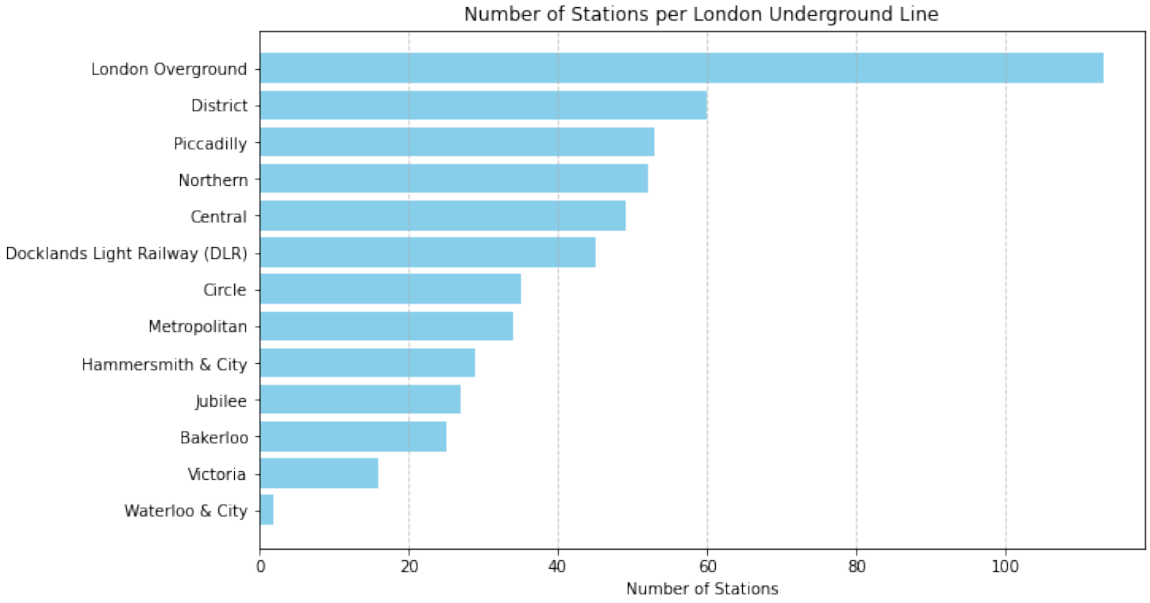
We can identify the following components 11 Tube Lines + other Lines in our Network:

- **Bakerloo**: 25 stations
- **Central**: 49 stations
- **Circle**: 35 stations
- **District**: 60 stations
- **Hammersmith & City**: 29 stations
- **Jubilee**: 27 stations
- **Metropolitan**: 34 stations
- **Northern**: 52 stations
- **Piccadilly**: 53 stations
- **Victoria**: 16 stations
- **Waterloo & City**: 2 stations
- +Light Rail, Regional Lines etc

The average order of these lines thus corresponds to 41.62 stations, but this mean has a great variance due to significant outliers as like the Waterloo & City line as can be seen in the barplot below.

In [16]:
```
import matplotlib.pyplot as plt

# Data: London Underground lines and their respective number of sta
lines = [
    "Waterloo & City", "Victoria", "Jubilee", "Bakerloo", "Hammersm
    "Metropolitan", "Circle", "Northern", "Piccadilly", "Central", "
    "Docklands Light Railway (DLR)", "London Overground"
]
stations = [2, 16, 27, 25, 29, 34, 35, 52, 53, 49, 60, 45, 113]
```

```
# Combine lines and stations into a list of tuples and sort by numb
lines_stations = sorted(zip(stations, lines))

# Unzip the sorted tuples
stations_sorted, lines_sorted = zip(*lines_stations)

# Create the bar plot
plt.figure(figsize=(10, 6))
plt.barh(lines_sorted, stations_sorted, color='skyblue')
plt.xlabel('Number of Stations')
plt.title('Number of Stations per London Underground Line')
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.show()
```



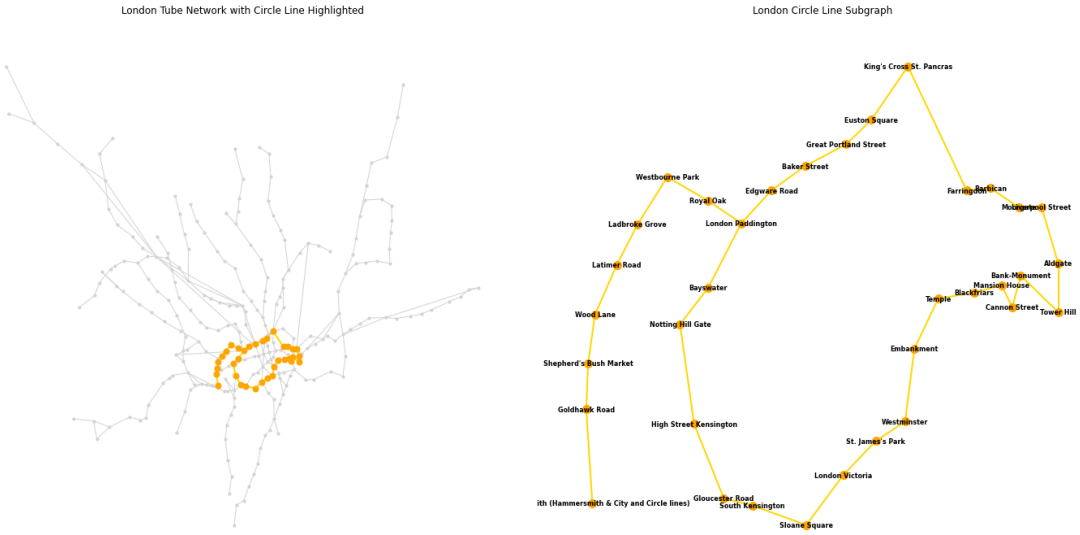
One of the most central subgraphs that also corresponds to a circle, is the Circle Line, which can be isolated in our Network as shown below.

In [17]:
```
# Define the Circle Line stations
circle_line_stations = [
    "Hammersmith (Hammersmith & City and Circle lines)", "Goldhawk
    "Shepherd's Bush Market", "Wood Lane", "Latimer Road", "Ladbrok
    "Westbourne Park", "Royal Oak", "London Paddington", "Edgware R
    "Baker Street", "Great Portland Street", "Euston Square",
    "King's Cross St. Pancras", "Farringdon", "Barbican", "Moorgate
    "Liverpool Street", "Aldgate", "Tower Hill", "Bank–Monument",
    "Cannon Street", "Mansion House", "Blackfriars", "Temple", "Emb
    "Westminster", "St. James's Park", "London Victoria", "Sloane S
    "South Kensington", "Gloucester Road", "High Street Kensington"
    "Notting Hill Gate", "Bayswater"
]

# Get station IDs and edges for the Circle Line
circle_line_station_ids = [node for node, name in labels.items() if
circle_line_edges = [(u, v) for u, v in G.edges() if u in circle_li
circle_line_edges = [edge for edge in circle_line_edges if edge not
circle_line_nodes = set(circle_line_station_ids)
```

```
# Create a 1x2 subplot layout
fig, axes = plt.subplots(1, 2, figsize=(20, 10))

# Plot the entire network with the Circle Line highlighted
nx.draw(
    G,
    pos=pos_geo,
    with_labels=False,
    node_size=10,
    node_color='lightgray',
    edge_color='lightgray',
    ax=axes[0]
)
nx.draw_networkx_edges(
    G,
    pos=pos_geo,
    edgelist=circle_line_edges,
    edge_color='gold',
    width=2,
    ax=axes[0]
)
nx.draw_networkx_nodes(
    G,
    pos=pos_geo,
    nodelist=circle_line_nodes,
    node_color='orange',
    node_size=50,
    ax=axes[0]
)

axes[0].set_title("London Tube Network with Circle Line Highlighted"

# Extract the subgraph of the Circle Line
circle_line_subgraph = G.subgraph(circle_line_nodes)

# Plot the Circle Line subgraph
nx.draw(
    circle_line_subgraph,
    pos=pos_geo,
    with_labels=True,
    labels={node: labels[node] for node in circle_line_nodes},
    edgelist=circle_line_edges,
    node_size=100,
    node_color='orange',
    font_size=8,
    font_weight='bold',
    edge_color='gold',
    width=2,
    ax=axes[1]
)
axes[1].set_title("London Circle Line Subgraph")


# Calculate the order (number of nodes) and size (number of edges) 
circle_line_order = len(circle_line_nodes)
circle_line_size = len(circle_line_edges)
```

```
print(f"Order of the Circle Line (number of stations): {circle_line

# Adjust layout
plt.tight_layout()
plt.show()
```

```
Order of the Circle Line (number of stations): 35
```



Number of components of the network

As mentioned before, the network is undirected, therefore the number of components in the London Undergroung metro system network reveals that there is only one connected component, indicating that all stations are part of a single, cohesive network. This suggests that the metro system is designed to ensure complete connectivity, with no isolated or disconnected stations. This is ensuring that every station can be reached from any other station, either directly or via transfer.

The fact that there is a single connected component also reflects the network's design efficiency, and confirms that the network is a weakly connected graph, you can travel between any two vertices by following some set of edges. This means that commuters can travel seamlessly between any two points within the system without encountering unreachable stations. This networks helps in the flexibility of route planning, including multiple tranfer points in more central lines to be able to reach any part of the city that has a metro line.

Additionally, it will enfasises the importance of central lines that can have more traffic, and can serve as critical nodes in maintaining overall network cohesion.

In [18]:
```
#Find the connected components
connected_components = list(nx.connected_components(G))
```

```
#Calculate the number of connected components
num_components = len(connected_components)

print(f"Number of connected components: {num_components}")
```

```
Number of connected components: 1
```

Diameter

Diameter without weights

The longest shortest path between any two stations in the London Underground network spans 35 vertices (stations). It will indicates, the furthest distance between any two stations, based on the shortest possible route, requires traveling through 35 stations across the city.

In [19]:
```
if nx.is_connected(G):
    diameter = nx.diameter(G)
    print(f"The diameter of the connected graph is: {diameter}")
else:
    print("The graph is disconnected.")

# Find the pair of nodes that defines the diameter
if nx.is_connected(G):
    diameter_path = nx.diameter(G)
    longest_shortest_path = nx.periphery(G)
    # Get the two nodes that define the diameter
    diameter_nodes = longest_shortest_path[:2]
    print(f"The diameter lies between stations: {diameter_nodes[0]}
```

```
The diameter of the connected graph is: 35
The diameter lies between stations: 345 and 370
```

The following code provides both the sequence of station names along the shortest route and the total distance of travel between the two specified stations, making it useful for analyzing travel routes in the London Undergroung metro system network.

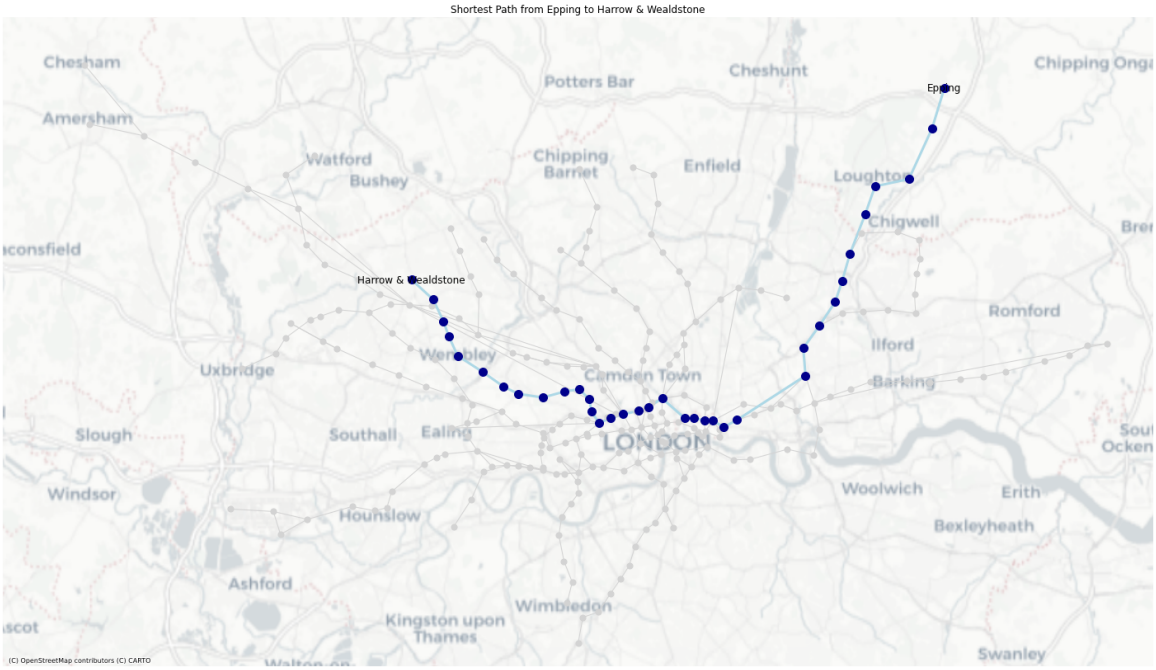
The code calculates the total distance (sum of the weights) of the shortest path using Dijkstra's algorithm, which reflects the total travel distance between the two stations. If no path exists between the nodes or if a node is not found in the graph, appropriate error messages are displayed.

In [20]:
```
def find_shortest_path(graph, initial_vertex, final_vertex):

    # Find the shortest path
    path = nx.dijkstra_path(graph, source=initial_vertex, target=fi
    path_station_names = [graph.nodes[node].get('label', 'No Label'
    path_length = len(path) - 1
    path_distance = nx.dijkstra_path_length(graph, source=initial_v


    print("Shortest Path (Node IDs):", path)
```

```
    print("Shortest Path (Station Names):", path_station_names)
    print("Number of Stations in Path:", path_length)
    print("Total Distance (km):", path_distance)


find_shortest_path(G, '345', '370')
```

```
Shortest Path (Node IDs): ['345', '494', '329', '411', '302', '537',
'480', '473', '408', '407', '489', '530', '273', '409', '419', '28
3', '349', '400', '347', '360', '280', '341', '441', '514', '412', '
398', '451', '392', '532', '369', '488', '517', '429', '477', '395',
'370']
Shortest Path (Station Names): ['Epping', 'Theydon Bois', 'Debden',
'Loughton', 'Buckhurst Hill', 'Woodford', 'South Woodford', 'Snaresb
rook', 'Leytonstone', 'Leyton', 'Stratford', 'Whitechapel', 'Aldgate
East', 'Liverpool Street', 'Moorgate', 'Barbican', 'Farringdon', "Ki
ng's Cross St. Pancras", 'Euston Square', 'Great Portland Street', '
Baker Street', 'Edgware Road', 'London Paddington', 'Warwick Avenu
e', 'Maida Vale', 'Kilburn Park', "Queen's Park", 'Kensal Green', 'W
illesden Junction', 'Harlesden', 'Stonebridge Park', 'Wembley Centra
l', 'North Wembley', 'South Kenton', 'Kenton', 'Harrow & Wealdston
e']
Number of Stations in Path: 35
Total Distance (km): 64.2756406417444
```

In [21]:
```
plot_shortest_path(G, '345', '370')
```



Diameter with weights

we can also calculate the weighted diameter which corresponds to the longest shortest path distance wise.

In [22]:
```
# Calculate the shortest path lengths using inverse distance as wei
shortest_path_lengths = dict(nx.all_pairs_dijkstra_path_length(G, w

# Find the diameter (maximum shortest path length)
weighted_diameter = max(max(lengths.values()) for lengths in shorte
```

```
print(f"The diameter of the graph with weights (inverse distance) i

# Find the pair of stations corresponding to the weighted diameter
for source, lengths in shortest_path_lengths.items():
    for target, length in lengths.items():
        if length == weighted_diameter:
            print(f"The two stations corresponding to the weighted 
            break
```

```
The diameter of the graph with weights (inverse distance) is: 23.681
40839665368
The two stations corresponding to the weighted diameter are: 421 and
366
```

In [23]:
```
find_shortest_path(G, '421', '366')
```

```
Shortest Path (Node IDs): ['421', '479', '324', '496', '495', '281',
'321', '319', '320', '487', '509', '446', '510', '362', '294', '28
0', '341', '441', '461', '527', '402', '405', '536', '471', '357', '
366']
Shortest Path (Station Names): ['Morden', 'South Wimbledon', 'Collie
rs Wood', 'Tooting Broadway', 'Tooting Bec', 'Balham', 'Clapham Sout
h', 'Clapham Common', 'Clapham North', 'Stockwell', 'Vauxhall', 'Pim
lico', 'London Victoria', 'Green Park', 'Bond Street', 'Baker Stree
t', 'Edgware Road', 'London Paddington', 'Royal Oak', 'Westbourne Pa
rk', 'Ladbroke Grove', 'Latimer Road', 'Wood Lane', "Shepherd's Bush
Market", 'Goldhawk Road', 'Hammersmith (Hammersmith & City and Circl
e lines)']
Number of Stations in Path: 25
Total Distance (km): 29.120318862949997
```

In [24]:
```
plot_shortest_path(G, '421', '366')
```

Adjecency Matrix

Using the Adjecency Matrix, we can calculate the mean giving us an average

0.009 Arcs per nodes^2, showing that overall there is very low connectivity in the network.

In [25]:
```
import scipy.io

# Generate the adjacency matrix
adj_matrix = nx.adjacency_matrix(G)
adj_matrix_dense = adj_matrix.todense()

# Calculate the mean
mean_adj_matrix = np.mean(adj_matrix_dense)
print(f"Mean of the adjacency matrix: {mean_adj_matrix}")
```

```
Mean of the adjacency matrix: 0.009022053909556694
```

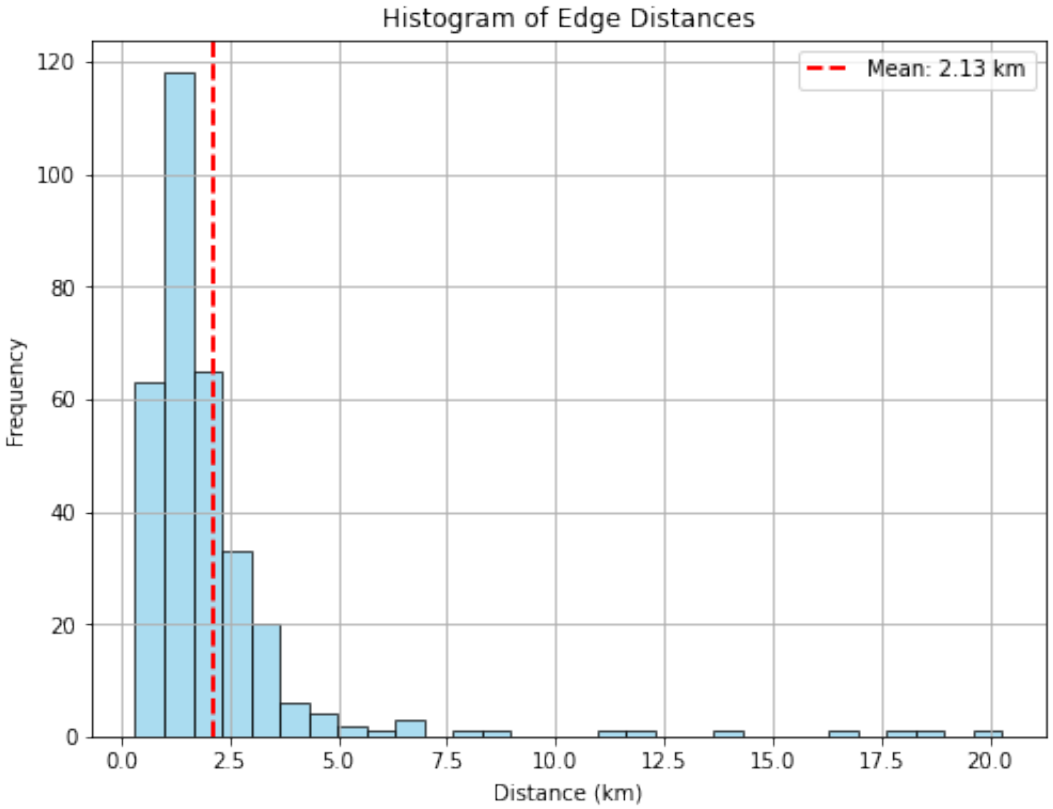
Attribute Analysis

With the main characteristics of the Network analyzed, we additionally do a brief Exploratory Data Analysis into the Attributes of the Network. This knowlegde might be useful in later interpretations of properties of the Network.

Edges

A histogram can be plotted to show the distribution of the distances and the mean distance (km) between stations. It has to be remembered as the distance is a linear approximation (e.g. euclidean distance) and the effective distances that are traveled between stations are slightly higher.

In [26]:
```
# Plot the histogram
distances = [data['dist'] for _, _, data in G.edges(data=True) if '
mean_distance = np.mean(distances)

plt.figure(figsize=(8, 6))
plt.hist(distances, bins=30, edgecolor='black', color='skyblue', al
plt.axvline(mean_distance, color='red', linestyle='dashed', linewid
plt.title('Histogram of Edge Distances')
plt.xlabel('Distance (km)')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)
plt.show()
```

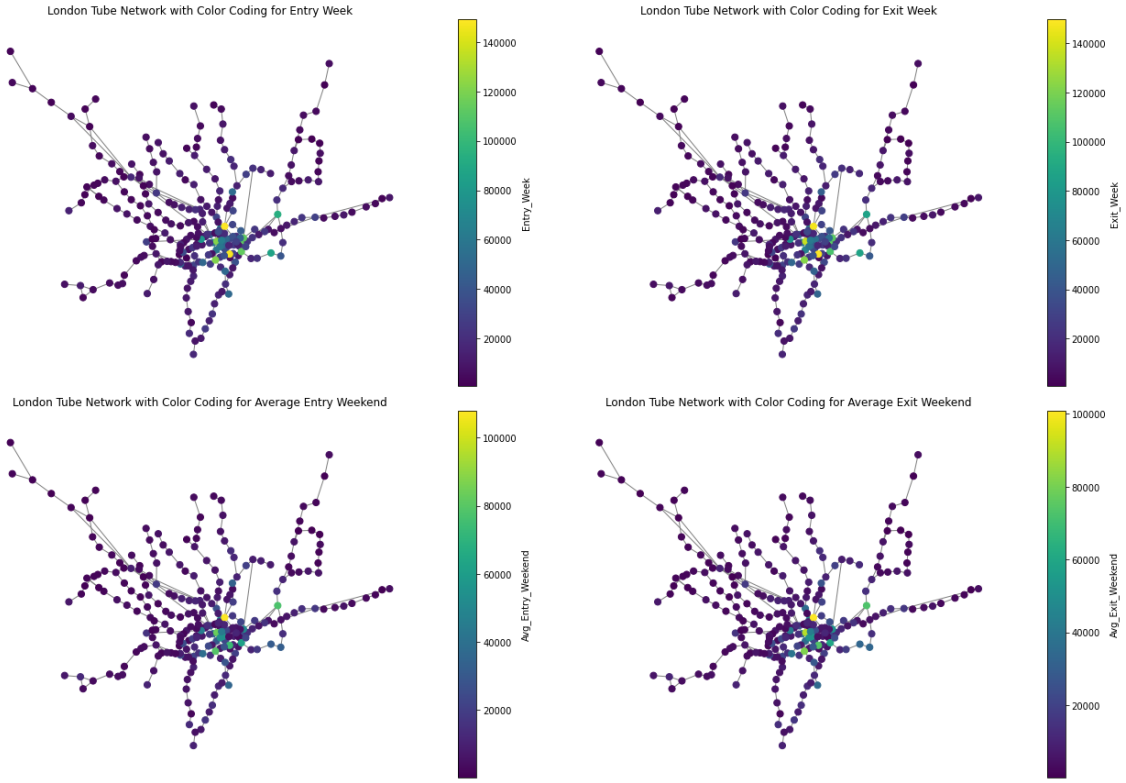


Nodes

The main attributes of the nodes correspond to the average entries and exits of the stations on weekdays and weekends. We can begin the analysis by plotting a colormapped network plot with the color corresponding to the entires/exits.

In [27]:
```
# Define the categories to visualize and their corresponding titles
categories = ['Entry_Week', 'Exit_Week', 'Avg_Entry_Weekend', 'Avg_
titles = ['Entry Week', 'Exit Week', 'Average Entry Weekend', 'Aver

# Create a 2x2 subplot
fig, axes = plt.subplots(2, 2, figsize=(18, 12))

# Iterate over the categories and plot each one
for i, (category, title) in enumerate(zip(categories, titles)):
    # Extract values for the current category
    values = [data[category] for _, data in G.nodes(data=True) if c

    # Normalize the values for the colormap
    norm = plt.Normalize(min(values), max(values))
    cmap = plt.cm.viridis

    # Create a color map for the nodes
    node_colors = [cmap(norm(data[category])) if category in data e

    # Plot the network on the corresponding subplot
    ax = axes[i // 2, i % 2]
    nx.draw(
        G,
```

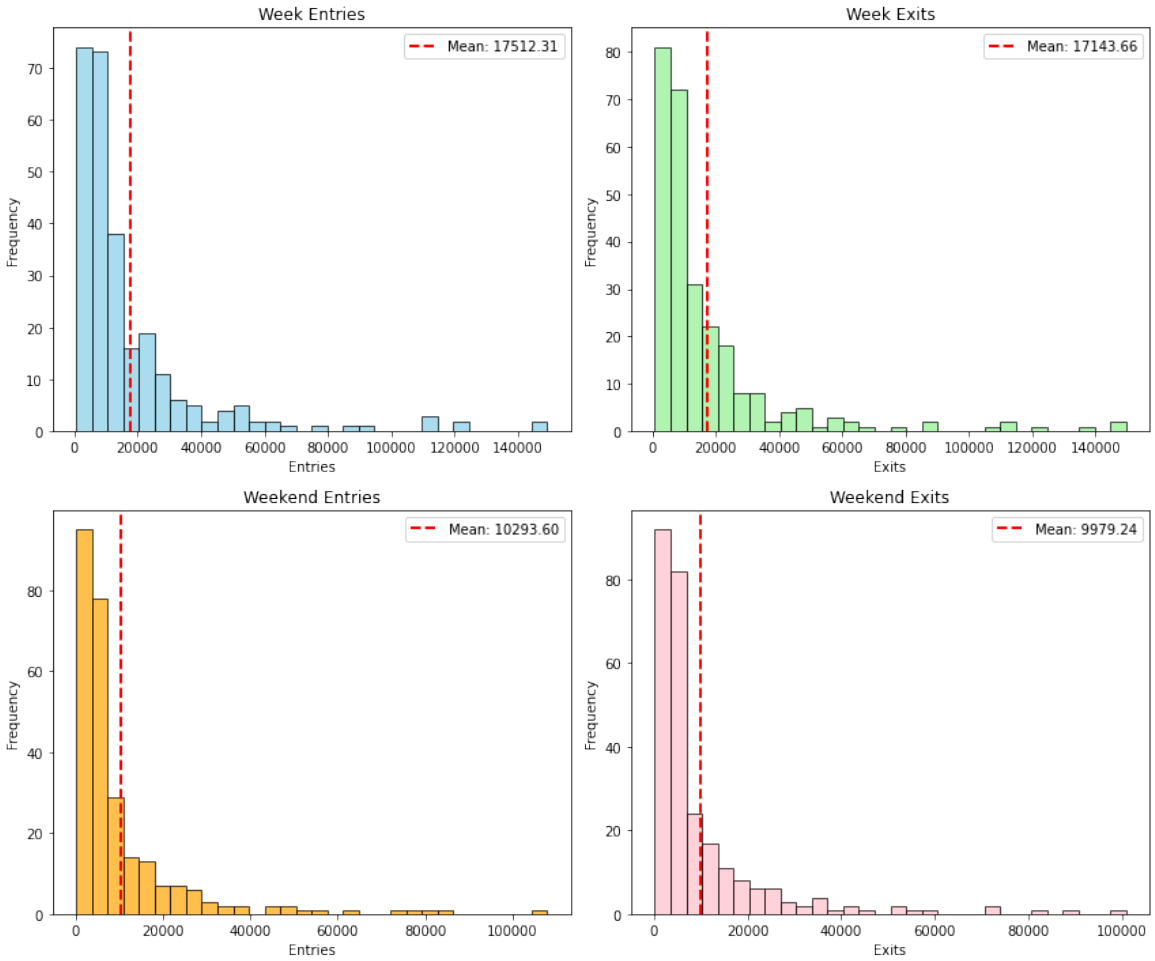
```
        pos=pos_geo,
        with_labels=False,
        node_size=50,
        node_color=node_colors,
        edge_color='gray',
        ax=ax
    )

    # Add a colorbar
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])
    cbar = plt.colorbar(sm, ax=ax)
    cbar.set_label(category)

    # Set the title for the subplot
    ax.set_title(f"London Tube Network with Color Coding for {title}

# Adjust layout
plt.tight_layout()
plt.show()
```



Furthermore, histograms for the exits and entires on weekdays and weekends of the stations can be plotted, showing a similar distribution to that of the degrees.

In [28]:
```
# Extract data for histograms directly from the graph
entry_week = [data['Entry_Week'] for _, data in G.nodes(data=True) :
exit_week = [data['Exit_Week'] for _, data in G.nodes(data=True) if
avg_entry_weekend = [data['Avg_Entry_Weekend'] for _, data in G.nod
avg_exit_weekend = [data['Avg_Exit_Weekend'] for _, data in G.nodes

# Calculate means
mean_entry_week = np.mean(entry_week)
```

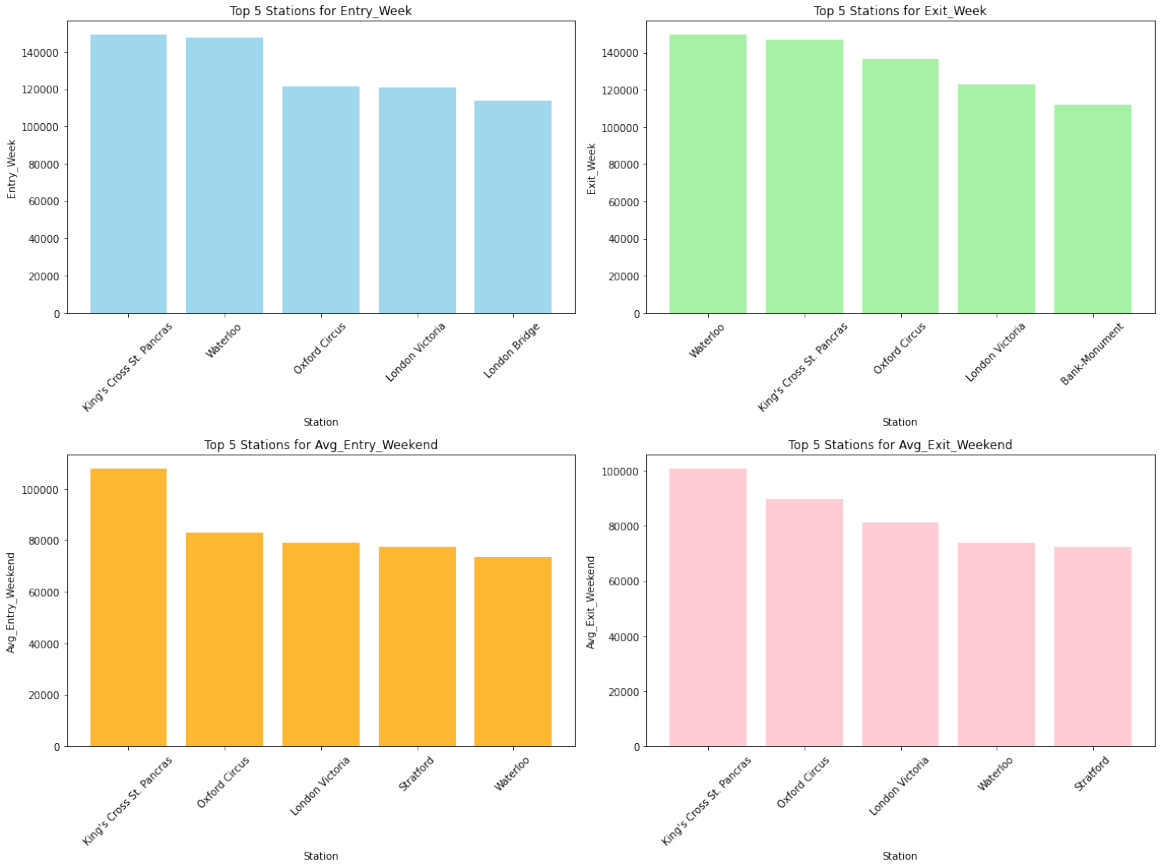
```
mean_exit_week = np.mean(exit_week)
mean_avg_entry_weekend = np.mean(avg_entry_weekend)
mean_avg_exit_weekend = np.mean(avg_exit_weekend)

# Create a 4x4 subplot
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Plot histograms with mean lines
axes[0, 0].hist(entry_week, bins=30, color='skyblue', edgecolor='bl
axes[0, 0].axvline(mean_entry_week, color='red', linestyle='dashed'
axes[0, 0].set_title('Week Entries')
axes[0, 0].set_xlabel('Entries')
axes[0, 0].set_ylabel('Frequency')
axes[0, 0].legend()

axes[0, 1].hist(exit_week, bins=30, color='lightgreen', edgecolor='
axes[0, 1].axvline(mean_exit_week, color='red', linestyle='dashed',
axes[0, 1].set_title('Week Exits')
axes[0, 1].set_xlabel('Exits')
axes[0, 1].set_ylabel('Frequency')
axes[0, 1].legend()

axes[1, 0].hist(avg_entry_weekend, bins=30, color='orange', edgecol
axes[1, 0].axvline(mean_avg_entry_weekend, color='red', linestyle='
axes[1, 0].set_title('Weekend Entries')
axes[1, 0].set_xlabel('Entries')
axes[1, 0].set_ylabel('Frequency')
axes[1, 0].legend()

axes[1, 1].hist(avg_exit_weekend, bins=30, color='pink', edgecolor=
axes[1, 1].axvline(mean_avg_exit_weekend, color='red', linestyle='da
axes[1, 1].set_title('Weekend Exits')
axes[1, 1].set_xlabel('Exits')
axes[1, 1].set_ylabel('Frequency')
axes[1, 1].legend()

# Adjust layout
plt.tight_layout()
plt.show()
```



We can also plot barplots of the top exitede/entered stations, showing that those do not correspond well to the stations with the highest degrees in the network (only Kings Cross Coincides). We thus conclude that the Stations with high degrees are more used to change between lines than as start or end of journeys. The two most used stations on weekdays are the two major train stations Kings Cross and Waterloo, which probably corresponds to people coming to the city for work, while Waterlpoos significance reduces on weekends.

In [29]:
```
# Define the categories to analyze and their corresponding colors
categories = ['Entry_Week', 'Exit_Week', 'Avg_Entry_Weekend', 'Avg_
colors = ['skyblue', 'lightgreen', 'orange', 'pink']

# Create a 2x2 subplot
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Plot bar diagrams for the top 5 most used stations in each catego
for i, category in enumerate(categories):
    row, col = divmod(i, 2)
    top_stations = usage_data.nlargest(5, category)[['Station', cat

    # Create a bar plot in the corresponding subplot
    axes[row, col].bar(top_stations['Station'], top_stations[catego
    axes[row, col].set_title(f"Top 5 Stations for {category}")
    axes[row, col].set_xlabel("Station")
    axes[row, col].set_ylabel(category)
```

```
    axes[row, col].tick_params(axis='x', rotation=45)

# Adjust layout
plt.tight_layout()
plt.show()
```



These top 5 Stations marked on the map however still appera to be very much in the city center, as could be expected.

In [30]:
```
# Identify the top 5 most exited and most entered stations
top_5_exited = usage_data.nlargest(5, 'Exit_Week')[['Station', 'Exi
top_5_entered = usage_data.nlargest(5, 'Entry_Week')[['Station', 'E

# Combine the top 5 exited and entered stations into a single set
top_5_combined_stations = set(top_5_exited['Station']).union(set(to
top_5_combined_nodes = [node for node, label in labels.items() if l

# Example usage
stations_to_highlight = ['Oxford Circus', 'Bank-Monument', 'London
plot_highlighted_stations(G, pos_geo, top_5_combined_nodes, labels)
```

Advanced Network Visualization

With the properties of the Network defined and a general understanding of the Node and Edge attributes, we continue with advanced visualization tecniques to gain further understanding of the Networks.

Energy based Visualizations

Following graphs layouts are based on energy functions, where the algorithm simulates physical forces to determine node positions. The layout process aims to minimize a global energy function.

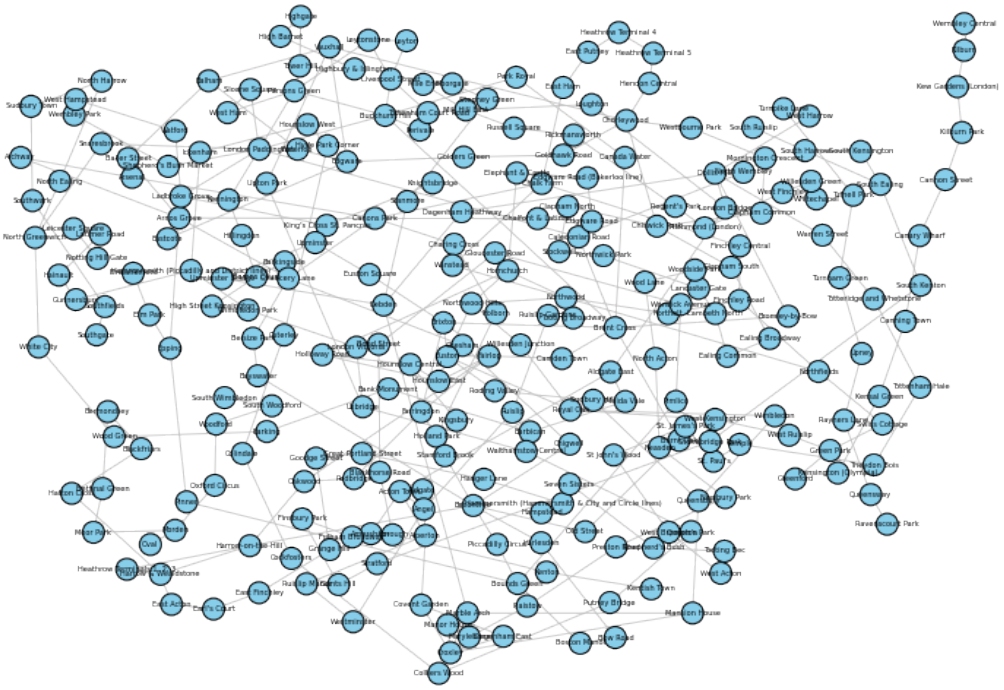
The main focus in the London Underground Network with the corresponding energy layouts is to reveal clusters, central hubs or important vertex in a clearer visually way.

Davidson and Harel layout:

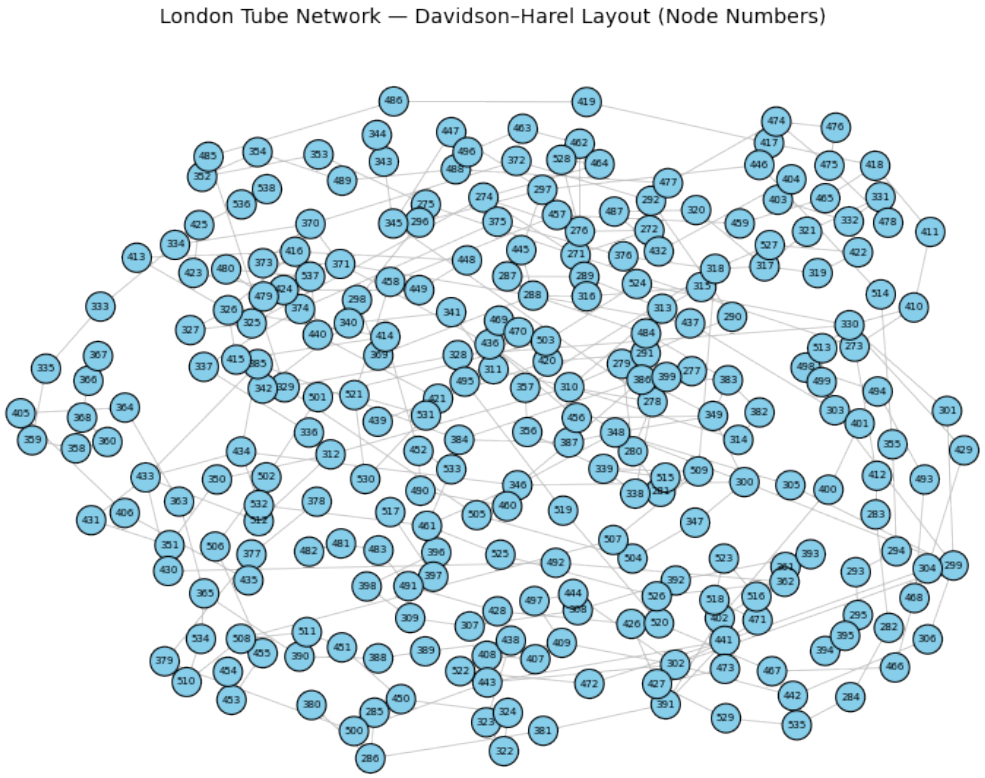
The Davidson and Harel layout will aim for a result with a minimal edge crossing. It is plotted the layouts with the stations names and node number for easier visualization.

In [31]:
```
import igraph as ig
import random
#set seed
# Set the random seed for igraph (global setting)
ig.set_random_number_generator(random.Random(1))
```

```
#Convert NetworkX graph to igraph
edges_i = list(G.edges())
ig_graph = ig.Graph.TupleList(edges_i, directed=False)

#Add station names as vertex labels
if 'label' in list(G.nodes(data=True))[0][1]:
    station_labels = [data['label'] for _, data in G.nodes(data=Tru
else:
    # Fallback to node IDs
    station_labels = [str(n) for n in G.nodes()]

ig_graph.vs["name"] = station_labels

#Compute Davidson–Harel layout
layout_dh = ig_graph.layout("dh")

#Plot the graph
fig, ax = plt.subplots(figsize=(14, 10))
ig.plot(
    ig_graph,
    layout=layout_dh,
    target=ax,
    vertex_label=ig_graph.vs["name"],
    vertex_size=15,
    vertex_color="skyblue",
    edge_color="gray",
    edge_width=0.8,
    vertex_label_size=5,
)
plt.title("London Tube Network – Davidson–Harel Layout", fontsize=1
plt.axis('off')
plt.show()
```



In [32]:
```
# Use original NetworkX node numbers as labels
# These are the unique node identifiers in G
node_ids = list(G.nodes())
ig_graph.vs["name"] = [str(n) for n in node_ids]  # Cast to str for

# Compute Davidson–Harel layout
layout_dh = ig_graph.layout("dh")

# Plot the graph with node number labels
fig, ax = plt.subplots(figsize=(14, 10))
ig.plot(
    ig_graph,
    layout=layout_dh,
    target=ax,
    vertex_label=ig_graph.vs["name"],
    vertex_size=20,
    vertex_color="skyblue",
    edge_color="gray",
    edge_width=0.8,
    vertex_label_size=7,
)
plt.title("London Tube Network – Davidson–Harel Layout (Node Number
plt.axis('off')
plt.show()
```



The Davidson and Harel layout applied to the London Underground metro system produces a circular, evenly distributed visualization where no particular emphasis is placed on important stations or clusters. All nodes (stations) are positioned based on force-directed principles, without automatically highlighting central hubs or high-connectivity areas. As a result, key stations with higher degrees are not visually distinguished. To address this, we will manually highlight the stations with the highest degrees to better illustrate their significance within the network.

Let's called again the top 5 nodes with the highest unweighted degrees.

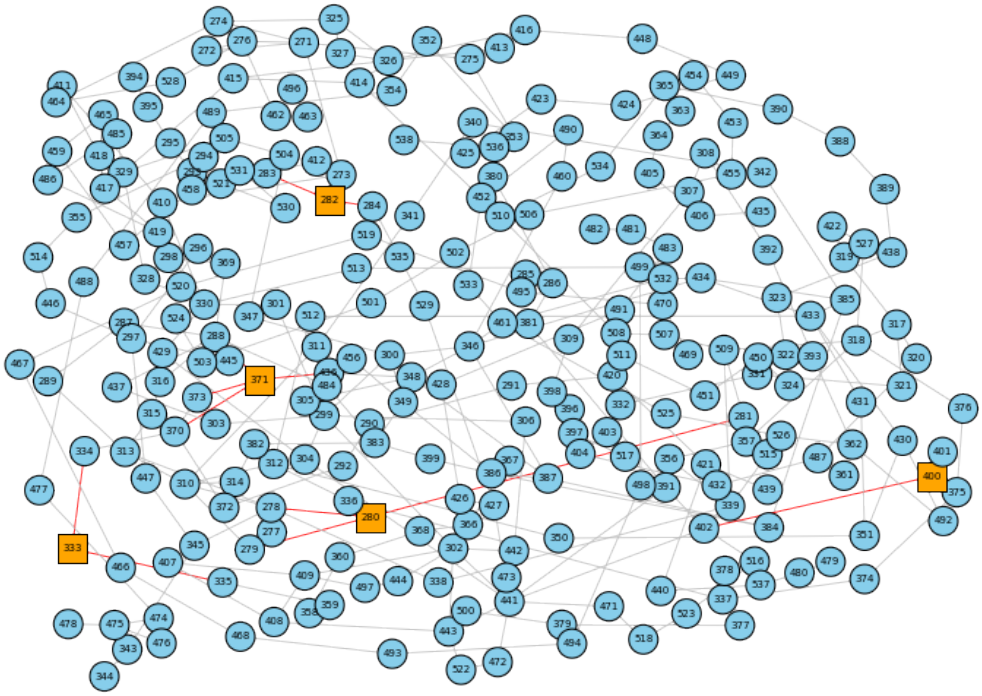
In [33]:
```
# Find the top 5 nodes with the highest degrees
degrees = dict(G.degree())  # {node: degree}
degree_sequence = [degree for node, degree in degrees.items()]
sorted_nodes_by_degree = sorted(degrees.items(), key=lambda x: x[1]

# Print the top 5 nodes with their degrees and labels
print("\nTop 5 nodes with the highest degrees:")
for node, degree in sorted_nodes_by_degree:
    label = G.nodes[node].get('label', 'No Label')
    print(f"Node: {node}, Label: {label}, Degree: {degree}")
```

```
Top 5 nodes with the highest degrees:
Node: 371, Label: Harrow-on-the-Hill, Degree: 8
Node: 280, Label: Baker Street, Degree: 7
Node: 400, Label: King's Cross St. Pancras, Degree: 7
Node: 282, Label: Bank-Monument, Degree: 6
Node: 333, Label: Earl's Court, Degree: 6
```

In [34]:
```
highlight_labels =  ['371','280','400','282','333']
```

```
vertex_colors = ['orange' if name in highlight_labels else 'skyblue
vertex_shapes = ['square' if name in highlight_labels else 'circle'

# Highlight edges connected to top nodes
edge_colors = []
for edge in ig_graph.es:
    src = ig_graph.vs[edge.source]["name"]
    tgt = ig_graph.vs[edge.target]["name"]
    if src in highlight_labels or tgt in highlight_labels:
        edge_colors.append('red')
    else:
        edge_colors.append('gray')

#  Layout and plot
layout_dh = ig_graph.layout("dh")

fig, ax = plt.subplots(figsize=(14, 10))
ig.plot(
    ig_graph,
    layout=layout_dh,
    target=ax,
    vertex_label=ig_graph.vs["name"],
    vertex_size=20,
    vertex_color=vertex_colors,
    vertex_shape=vertex_shapes,
    edge_color=edge_colors,
    edge_width=0.8,
    vertex_label_size=7,
)
plt.title("London Tube Network Davidson–Harel Layout– Top 5 Nodes b
plt.axis('off')
plt.show()
```

London Tube Network Davidson-Harel Layout— Top 5 Nodes by Degree Highlighted



Kamada and Kawai layout:

The Kamada and Kawait layout (another type of energy-based graph visualization method) aims to position nodes so their shortest path distances are preserved as accurately as possible, related nodes are placed closer together and overall structure is more readable.

The Kamada and Kawai layout produces a visualization that closely resembles the actual geographic structure of the London Underground, though not an exact match. By approximating ideal distances between all pairs of stations based on their shortest path lengths, the layout naturally forms a tree-like structure, with several "arms" of the network radiating outward from the central hub stations. This helps to visually emphasize how the main lines converge in central London, reflecting the real-world connectivity and flow of the metro system.

In [35]:
```
# Define layouts
layouts = {
    "London Tube Network - Kamada-Kawai (Node Numbers)": nx.kamada_
}

plt.figure(figsize=(30, 20),dpi = 300)
for i, (name, layout_func) in enumerate(layouts.items(), 1):
    pos = layout_func(G)
    plt.subplot(2, 3, i)
    nx.draw(G, pos, with_labels=True, node_size=40, node_color='sky
    plt.title(f"{name} layout")

plt.tight_layout()
plt.show()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent cal
l last)
Input In [35], in <module>
      6 plt.figure(figsize=(30, 20),dpi = 300)
      7 for i, (name, layout_func) in enumerate(layouts.items(), 1):
----> 8     pos = layout_func(G)
      9     plt.subplot(2, 3, i)
     10     nx.draw(G, pos, with_labels=True, node_size=40, node_col
or='skyblue', font_size=8)

File /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.
10/site-packages/networkx/drawing/layout.py:710, in kamada_kawai_lay
out(G, dist, pos, weight, scale, center, dim)
    707         pos = dict(zip(G, np.linspace(0, 1, len(G))))
    708 pos_arr = np.array([pos[n] for n in G])
--> 710 pos = _kamada_kawai_solve(dist_mtx, pos_arr, dim)
    712 pos = rescale_layout(pos, scale=scale) + center
    713 return dict(zip(G, pos))

File /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.
10/site-packages/networkx/drawing/layout.py:727, in _kamada_kawai_so
lve(dist_mtx, pos_arr, dim)
    724 meanwt = 1e-3
    725 costargs = (np, 1 / (dist_mtx + np.eye(dist_mtx.shape[0]) *
1e-3), meanwt, dim)
--> 727 optresult = sp.optimize.minimize(
    728     _kamada_kawai_costfn,
    729     pos_arr.ravel(),
    730     method="L-BFGS-B",
    731     args=costargs,
    732     jac=True,
    733 )
    735 return optresult.x.reshape((-1, dim))

AttributeError: module 'scipy' has no attribute 'optimize'
```

<Figure size 9000x6000 with 0 Axes>

The following code highlights stations with more than three connections (a degree greater than 3) in the unweighted graph of the London Underground network. As seen in the resulting layout, these high-degree stations are predominantly located near the center of the network. This distribution aligns with the real-world structure of London, where central stations act as major interchange hubs, facilitating connectivity between multiple lines. These central nodes play a crucial role in ensuring efficient passenger flow across the broader metro system.

In [36]:
```
# Define layouts
layouts = {
    "London Tube Network - Kamada-Kawai (Node Numbers)": nx.kamada_
}

# Identify nodes with degree > 3
```

```
high_degree_nodes = [n for n in G.nodes if G.degree(n) > 3]
low_degree_nodes = [n for n in G.nodes if G.degree(n) <= 3]

# Identify edges that connect to high-degree nodes
high_degree_edges = [e for e in G.edges if e[0] in high_degree_node
low_degree_edges = list(set(G.edges) - set(high_degree_edges))

plt.figure(figsize=(30, 20),dpi = 300)
for i, (name, layout_func) in enumerate(layouts.items(), 1):
    pos = layout_func(G)
    plt.subplot(2, 3, i)

    # Draw low-degree nodes
    nx.draw_networkx_nodes(G, pos,
                           nodelist=low_degree_nodes,
                           node_color='skyblue',
                           node_shape='o',
                           node_size=40)

    # Draw high-degree nodes with a different shape and color
    nx.draw_networkx_nodes(G, pos,
                           nodelist=high_degree_nodes,
                           node_color='orange',
                           node_shape='s',  # square
                           node_size=80)

    # Draw edges not connected to high-degree nodes
    nx.draw_networkx_edges(G, pos,
                           edgelist=low_degree_edges,
                           edge_color='gray',
                           width=0.5)

    # Draw edges connected to high-degree nodes
    nx.draw_networkx_edges(G, pos,
                           edgelist=high_degree_edges,
                           edge_color='red',
                           width=1.2)

    # Draw labels
    nx.draw_networkx_labels(G, pos, font_size=6)

    plt.title(f"{name} layout")

plt.tight_layout()
plt.show()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent cal
l last)
Input In [36], in <module>
     14 plt.figure(figsize=(30, 20),dpi = 300)
     15 for i, (name, layout_func) in enumerate(layouts.items(), 1):
---> 16     pos = layout_func(G)
     17     plt.subplot(2, 3, i)
     19     # Draw low-degree nodes

File /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.
10/site-packages/networkx/drawing/layout.py:710, in kamada_kawai_lay
out(G, dist, pos, weight, scale, center, dim)
    707         pos = dict(zip(G, np.linspace(0, 1, len(G))))
    708 pos_arr = np.array([pos[n] for n in G])
--> 710 pos = _kamada_kawai_solve(dist_mtx, pos_arr, dim)
    712 pos = rescale_layout(pos, scale=scale) + center
    713 return dict(zip(G, pos))

File /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.
10/site-packages/networkx/drawing/layout.py:727, in _kamada_kawai_so
lve(dist_mtx, pos_arr, dim)
    724 meanwt = 1e-3
    725 costargs = (np, 1 / (dist_mtx + np.eye(dist_mtx.shape[0]) *
1e-3), meanwt, dim)
--> 727 optresult = sp.optimize.minimize(
    728     _kamada_kawai_costfn,
    729     pos_arr.ravel(),
    730     method="L-BFGS-B",
    731     args=costargs,
    732     jac=True,
    733 )
    735 return optresult.x.reshape((-1, dim))

AttributeError: module 'scipy' has no attribute 'optimize'
<Figure size 9000x6000 with 0 Axes>
```

Fruchterman-Reingold

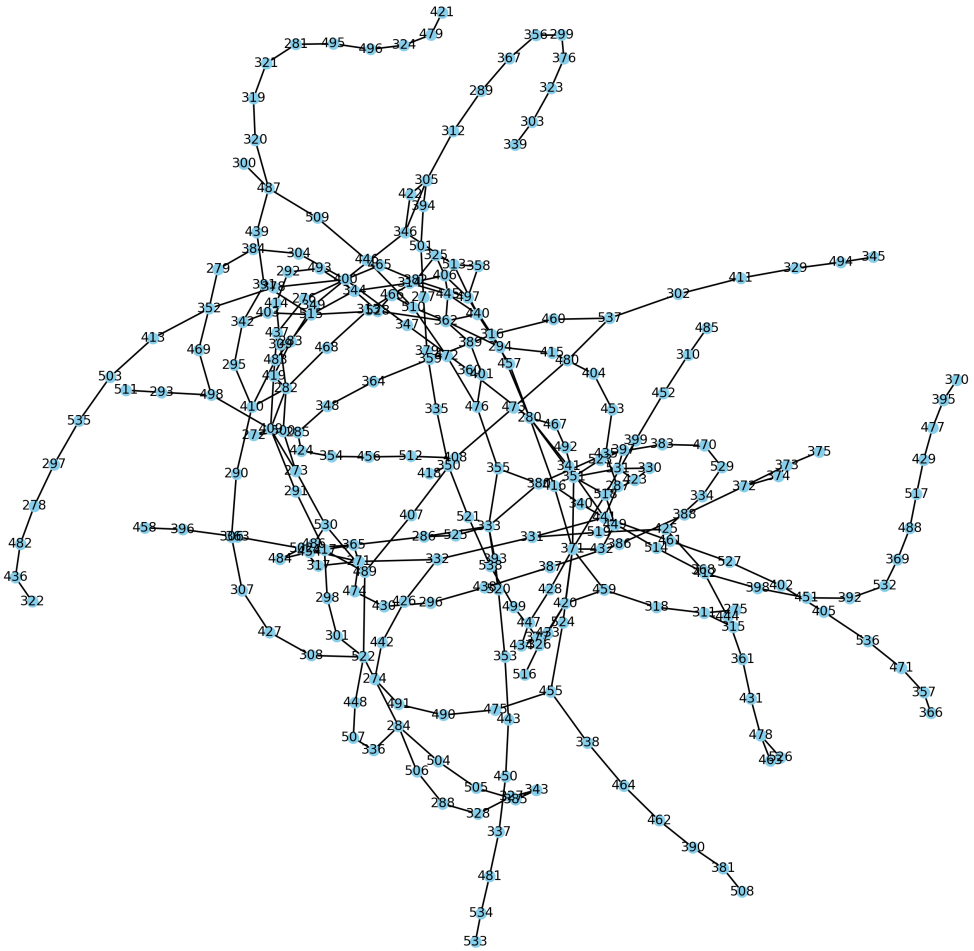
The Fruchterman-Reingol layout, treats nodes like charged particles that repel each other, while edges act like springs that pull connected nodes together, resulting in a layout where related nodes are placed closer together and the overall structure is more visually intuitive.

When applied to the London Underground network, the Fruchterman-Reingold layout reveals a general structure where some "arms" of the network, representing individual metro lines, extend outward with stations connected in a linear fashion. However, the central area appears denser and more tangled, lacking clear clustering or emphasis on specific stations. While the layout does not explicitly highlight important hubs, the visual density at the center suggests the presence of key interchange stations where multiple lines

converge. This reflects the real-world role of central London stations as critical connection points within the network.

In [37]:
```
# Define layouts
layouts = {
    "London Tube Network - Fruchterman-Reingold (Node Numbers)":nx.
}

plt.figure(figsize=(30, 20),dpi = 300)
for i, (name, layout_func) in enumerate(layouts.items(), 1):
    pos = layout_func(G, seed = 1)
    plt.subplot(2, 3, i)
    nx.draw(G, pos, with_labels=True, node_size=40, node_color='skyl
    plt.title(f"{name} layout")

plt.tight_layout()
plt.show()
```

London Tube Network - Fruchterman-Reingold (Node Numbers) layout

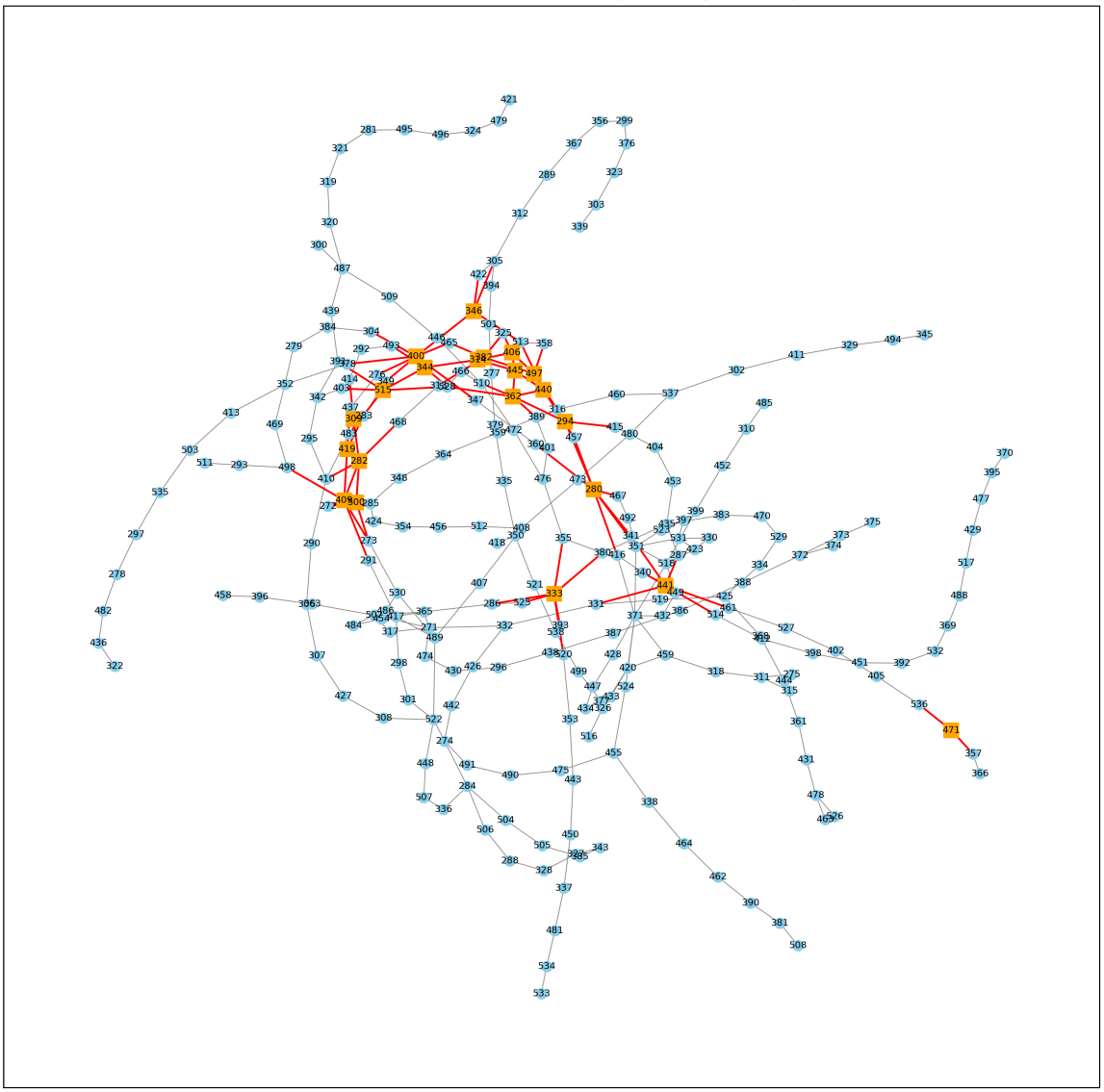


The following code applies the Fruchterman-Reingold force-directed algorithm to visualize the London Underground network. It highlights stations (nodes) whose weighted degree calculated as the sum of the inverse

distances to neighboring stations exceeds a value of 4. Stations located in central London, where multiple lines intersect and distances between stations are shorter, are more likely to meet this threshold. As expected, several central nodes are visually emphasized in the layout. Notably, station 471, which appears along one of the network's outer branches, is also highlighted. This prompts a closer investigation into its structure and connections to understand why it stands out in terms of weighted connectivity.

In [38]:
```
# Calculate the weighted degree for each node using inverse of the
def weighted_degree(graph):
    weighted_degrees = {}
    for node in graph.nodes:
        w_degree = 0
        for neighbor in graph.neighbors(node):
            weight = graph[node][neighbor].get('dist', 1)  # Defaul
            if weight > 0:
                w_degree += 1 / weight
        weighted_degrees[node] = w_degree
    return weighted_degrees

# Calculate weighted degrees
weighted_degrees = weighted_degree(G)

# Threshold for highlighting
threshold = 4

# Classify nodes based on weighted degree
high_degree_nodes = [n for n, d in weighted_degrees.items() if d >
low_degree_nodes = [n for n in G.nodes if n not in high_degree_node

# Classify edges connected to high-degree nodes
high_degree_edges = [e for e in G.edges if e[0] in high_degree_node
low_degree_edges = list(set(G.edges) - set(high_degree_edges))

# Define layout
layouts = {
    "London Tube Network - Fruchterman-Reingold (Weighted Degree > 
}

# Plot
plt.figure(figsize=(30, 20),dpi = 300)
for i, (name, layout_func) in enumerate(layouts.items(), 1):
    pos = layout_func(G, seed = 1)
    plt.subplot(2, 3, i)

    # Draw low-degree nodes
    nx.draw_networkx_nodes(G, pos,
                           nodelist=low_degree_nodes,
                           node_color='skyblue',
                           node_size=40,
                           node_shape='o')

    # Draw high-degree nodes
    nx.draw_networkx_nodes(G, pos,
```

```
                            nodelist=high_degree_nodes,
                            node_color='orange',
                            node_size=80,
                            node_shape='s')

    # Draw edges not connected to high-degree nodes
    nx.draw_networkx_edges(G, pos,
                            edgelist=low_degree_edges,
                            edge_color='gray',
                            width=0.5)

    # Draw edges connected to high-degree nodes
    nx.draw_networkx_edges(G, pos,
                            edgelist=high_degree_edges,
                            edge_color='red',
                            width=1.2)

    # Draw labels
    nx.draw_networkx_labels(G, pos, font_size=6)

    plt.title(f"{name}")

plt.tight_layout()
plt.show()
```



Upon further analysis of node 471, which corresponds to Shepherd's Bush Market station, we observe that its degree is 2. This means the station has 2 edges connecting it to other stations in the network. However, the weighted degree of the station is significantly higher, at 4.315. The weighted degree is calculated based on the inverse of the distance to neighboring stations, which is used to measure the strength of its connections. Specifically, the inverse of the distance (smaller distances contribute more heavily) reflects how "close" a node is to its neighbors. Lower distances between stations lead to higher weighted degrees, meaning that Shepherd's Bush Market is relatively close to its connected stations compared to other nodes with higher distances, making it a more strongly connected node in terms of proximity.

This closer relationship is further emphasized in the plot of "London Tube Network - Fruchterman-Reingold (Weighted Degree > 4)". In this plot, stations with weighted degrees greater than 4 are highlighted, despite Shepherd's Bush Market station having only 2 direct connections (degree without weights), its strong weighted degree indicates that those connections are particularly close, contributing to its prominence in the network.

In [39]:
```
# Get the degree
node = '471'
degree_ = G.degree[node]
print(f"Degree of node {node}:", degree_)

label_ = G.nodes[node].get('label', 'No label found')
print(f"Label of node {node}:", label_)

# Retrieve weighted degree of node 471
weighted_degree_ = weighted_degrees.get(node, "Node not found")
print(f"Weighted degree of node {node}:", weighted_degree_)
```

```
Degree of node 471: 2
Label of node 471: Shepherd's Bush Market
Weighted degree of node 471: 4.315239636255769
```

Overall, energy functions do not seem to be fit better to visualizing the problem than the simple geographical representation used initially. As the metro network is well defined and the nodes have the exact location as attribute, the energy function based visualizations do not add significant insights into the already well understood structure of the network. They do however give a first glance at the centrality of the nodes, with different focuses depending on what energy function is used. It thus might be of interest to review theses plots again once the centrality measures have been calculated.

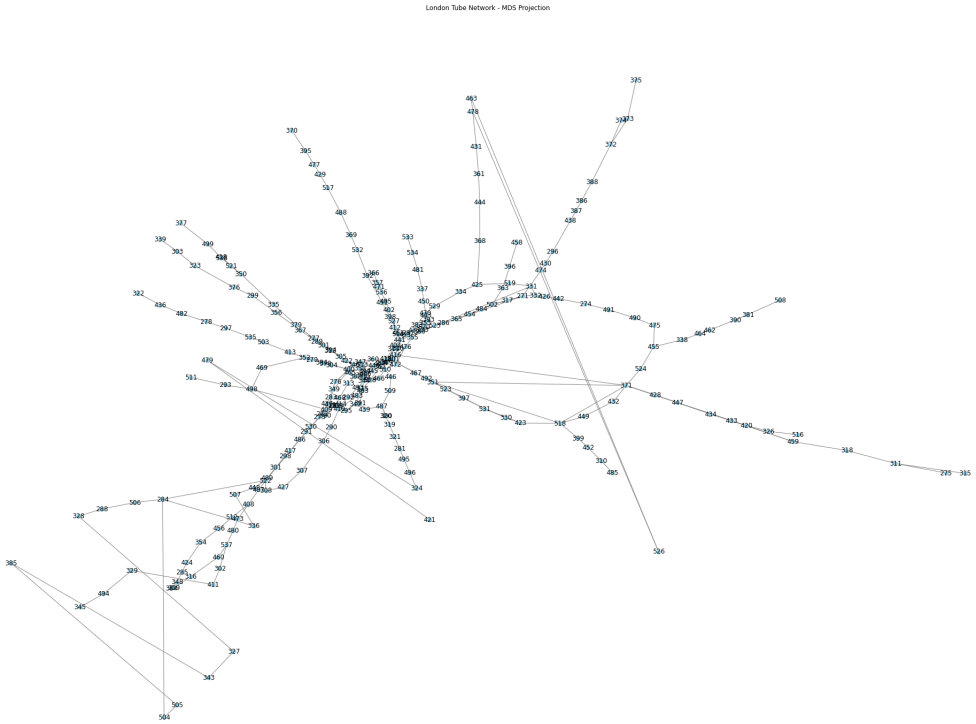
Multi Dimensional Scaling (MDS)

Multi Dimensional Scaling is a distance matrix based dimension reduction technique. To apply it to our dataset, we first generate the distance matrix by solving shortest path problems for each pair of nodes.

In [40]:
```
# Generate distance Matrix (Compute the shortest path distances betu
distance_matrix = dict(nx.all_pairs_dijkstra_path_length(G, weight=

# Convert to df
distance_df = pd.DataFrame.from_dict(distance_matrix, orient='index
distance_df = distance_df.reindex(index=G.nodes(), columns=G.nodes(
```

With the distance matrix set up, the MDS methods is import and then is applied them to distance matrix. The resulting visualization has a similar tree like structure, with most nodes close distances in the network also remain close in the MDS Visualization. However, it is also identified some clearly distorted node, like node 526, with no clear explanation on why they appear to be so distorted. In general the first MDS Coordinate appears to have been in some ways the longitude but flipped (e.g. stations previously to the right now more to the left and vice versa), while the second MDS coordinate is not very insightful, with some stations previously in the North remaining, while others are moved to the "south".

In [41]:
```
from sklearn.manifold import MDS

# Run MDS
distance_array = distance_df.values
mds = MDS(n_components=2, dissimilarity='precomputed', random_state
mds_coordinates = mds.fit_transform(distance_array)
mds_df = pd.DataFrame(mds_coordinates, columns=['MDS1', 'MDS2'], in


# Plot MDS (first two components)
plt.figure(figsize=(30, 20))
mds_positions = {str(node): (row['MDS1'], row['MDS2']) for node, ro
nx.draw(
    G,
    pos=mds_positions,
    with_labels=True,
    node_size=50,
    node_color='skyblue',
    edge_color='gray'
)
plt.title("London Tube Network – MDS Projection")
plt.show()
```



In some way, the result can be seen as an attempt to reconstruct the attributes longitude and latitude via first and second MDS coordinate from the nodes pairwise distances. As however the precise geographical positions of the nodes are known already through their longitude and latitude, MDS appears not to be able to add significant insights.

Centrality Measures

The following section contains a investigation into the centrality of nodes and edges.

Degree distribution

Degrees without weights

The average degree is calculated to be 2.42, which aligns with expectations. As most stations in the metro network are typically connected by a single line, with each station having two adjacent connections: one to the previous station and one to the next. Such a structure is common in metro systems, where the majority of stations are part of a linear sequence of stops

In [42]:
```
# Calculate the average degree of the graph
average_degree = sum(dict(G.degree()).values()) / len(G.nodes())

print(f"Average degree of the network: {average_degree:.2f}")
```

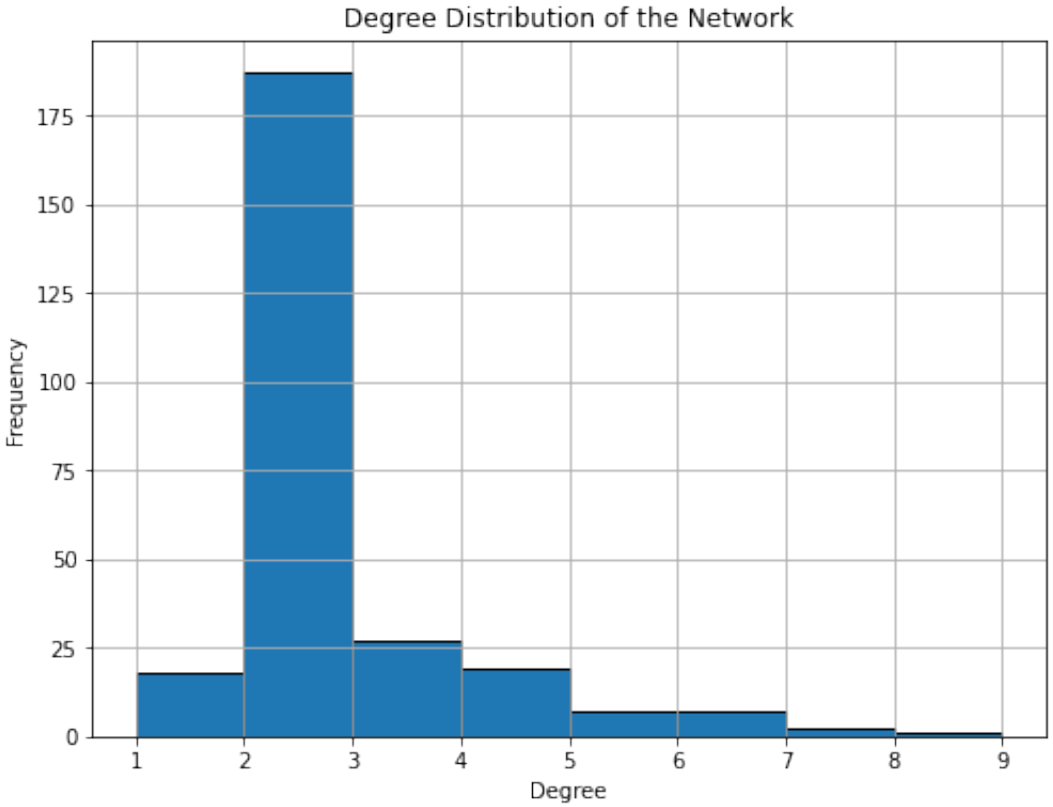
```
Average degree of the network: 2.42
```

In general, we observe that that stations have a degree of two, being a pass through station on one of the lines. Then, there is a few nodes having degree 1 corresponding to end stops of lines and other stations with degree 3+ connecting two or more lines.

The top 5 nodes with the highest degrees in the network are Harrow-on-the-Hill, Baker Street, Bank-Monument, King's Cross St. Pancras, and Earl's court. Harrow-on-the-Hill stands out with the highest degree of 8, indicating it has the most connections in the network. The other two stations—Baker Street and King's Cross St. Pancras have a degree of 7, showing they are all well-connected with a similar number of edges. Following the other two reamingn stations Bank-monument and Earl's Court have a degree of 6, in general, these stations are central hubs in the network, with varying but significant levels of connectivity. These stations are all above the average degree of the network, which is 2.42. Their higher degrees suggest they are significantly more connected than the average station, reflecting their importance in facilitating movement and providing multiple transport links. Some initial assumptions about their locations and roles in London transportation help provide context, a more detailed analysis would involve evaluating commuter traffic, station facilities, and transport links to further understand their importance within the network.

In [43]:
```
# Degree of each node in the graph
degrees = dict(G.degree())  # {node: degree}

# Degree distribution (histogram)
degree_sequence = [degree for node, degree in degrees.items()]
```

```
# Plot the degree distribution
plt.figure(figsize=(8, 6))
plt.hist(degree_sequence, bins=range(min(degree_sequence), max(degr
plt.title('Degree Distribution of the Network')
plt.xlabel('Degree')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()

# Find the top 5 nodes with the highest degrees
sorted_nodes_by_degree = sorted(degrees.items(), key=lambda x: x[1]

# Print the top 5 nodes with their degrees and labels
print("\nTop 5 nodes with the highest degrees:")
for node, degree in sorted_nodes_by_degree:
    label = G.nodes[node].get('label', 'No Label')
    print(f"Node: {node}, Label: {label}, Degree: {degree}")
```




```
Top 5 nodes with the highest degrees:
Node: 371, Label: Harrow-on-the-Hill, Degree: 8
Node: 280, Label: Baker Street, Degree: 7
Node: 400, Label: King's Cross St. Pancras, Degree: 7
Node: 282, Label: Bank-Monument, Degree: 6
Node: 333, Label: Earl's Court, Degree: 6
```

Overall, these stations indeed are more central to the network in a connectivity sense, as they represent the hubs at which passengers of the metro system can switch between lines and will thus see a substantial percentage of all passangers moving through those hubs.

In [44]:
```
#Get top 5 nodes by degree
sorted_nodes_by_degree = sorted(degrees.items(), key=lambda x: x[1]
```

```
top_5_degree_nodes = [node for node, degree in sorted_nodes_by_degr

#Plot highlighting these top degree nodes
plot_highlighted_stations(G, pos_geo, top_5_degree_nodes, labels)
```

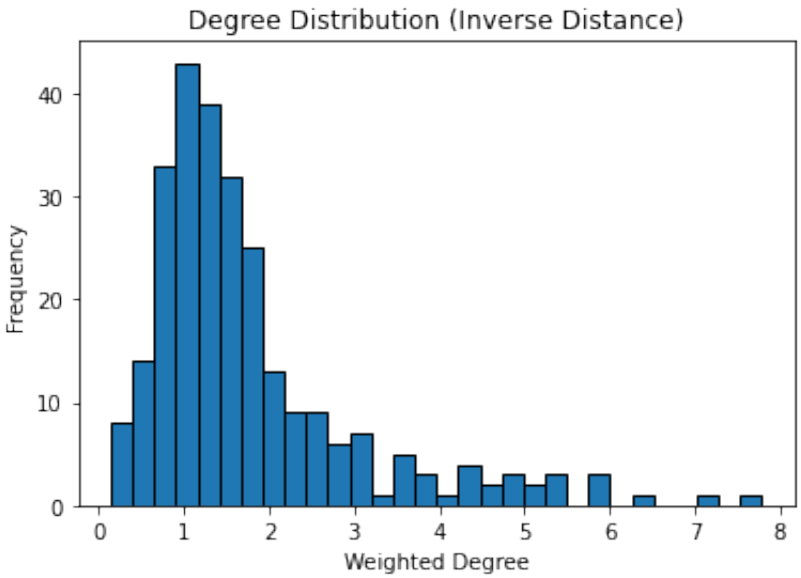
Degrees with weights

The network's degree distribution reveals an average weighted degree of 1.77, indicating that, on average, each station has a relatively modest number of strong connections when considering the inverse of the distance as the weight. The arcs weighted by their inverse distance can in this case be interpreted as the geographical closeness of the station to other stations in the network, as the inverse of a small distance yields a large weight. The a large sum of these weights thus shows that a stations has either a few stations that are very close to it or many stations connected to it but that might be a bit further away.

Notably, the top 5 stations with the highest weighted degrees, which are Bank-Monument (7.79), Leicester Square (7.18), Embankment (6.36), Charing Cross (5.92), and Waterloo (5.77), are all located in central and highly trafficked areas of the metro system. These stations exhibit significantly higher connectivity strength, suggesting they serve as major hubs where multiple metro lines converge. Their elevated weighted degrees reflect their crucial roles in the network, as they facilitate efficient transfers and high passenger flow. In the metro system is expected that the central stations are designed to handle high volumes of passengers and connections across various lines.

In [45]:
```
# Step 1: Calculate the weighted degree for each node (using inverse
def weighted_degree(graph, labels):
    weighted_degrees = {}
    for node in graph.nodes:
        weighted_degree = 0
        for neighbor in graph.neighbors(node):
            # Get the distance (weight) of the edge (inverse of dist
            weight = graph[node][neighbor].get('dist', 1)  # Default
            weighted_degree += 1 / weight  # Inverse of the distance
        weighted_degrees[node] = weighted_degree
    return weighted_degrees

# Calculate the weighted degree of each station
weighted_degrees = weighted_degree(G, labels)

# Step 2: Calculate the average degree
average_degree = sum(weighted_degrees.values()) / len(weighted_degre
print(f"Average weighted degree: {average_degree:.2f}")

# Step 3: Identify the top 5 stations based on the weighted degree
top_5_stations = sorted(weighted_degrees.items(), key=lambda x: x[1

# Display the top 5 stations and their weighted degrees along with
print("Top 5 stations with highest weighted degrees:")
for node, degree in top_5_stations:
    station_name = labels.get(node, "Unknown Station")  # Get the st
    print(f"Station: {station_name} (ID: {node}), Weighted Degree: {

#Plotting the degree distribution (using the inverse distance)
# Get all weighted degrees for plotting
all_degrees = list(weighted_degrees.values())

# Plot the degree distribution
plt.hist(all_degrees, bins=30, edgecolor='black')
plt.title("Degree Distribution (Inverse Distance)")
plt.xlabel("Weighted Degree")
plt.ylabel("Frequency")
plt.show()
```

```
Average weighted degree: 1.77
Top 5 stations with highest weighted degrees:
Station: Bank–Monument (ID: 282), Weighted Degree: 7.79
Station: Leicester Square (ID: 406), Weighted Degree: 7.18
Station: Embankment (ID: 344), Weighted Degree: 6.36
Station: Charing Cross (ID: 314), Weighted Degree: 5.92
Station: Waterloo (ID: 515), Weighted Degree: 5.77
```

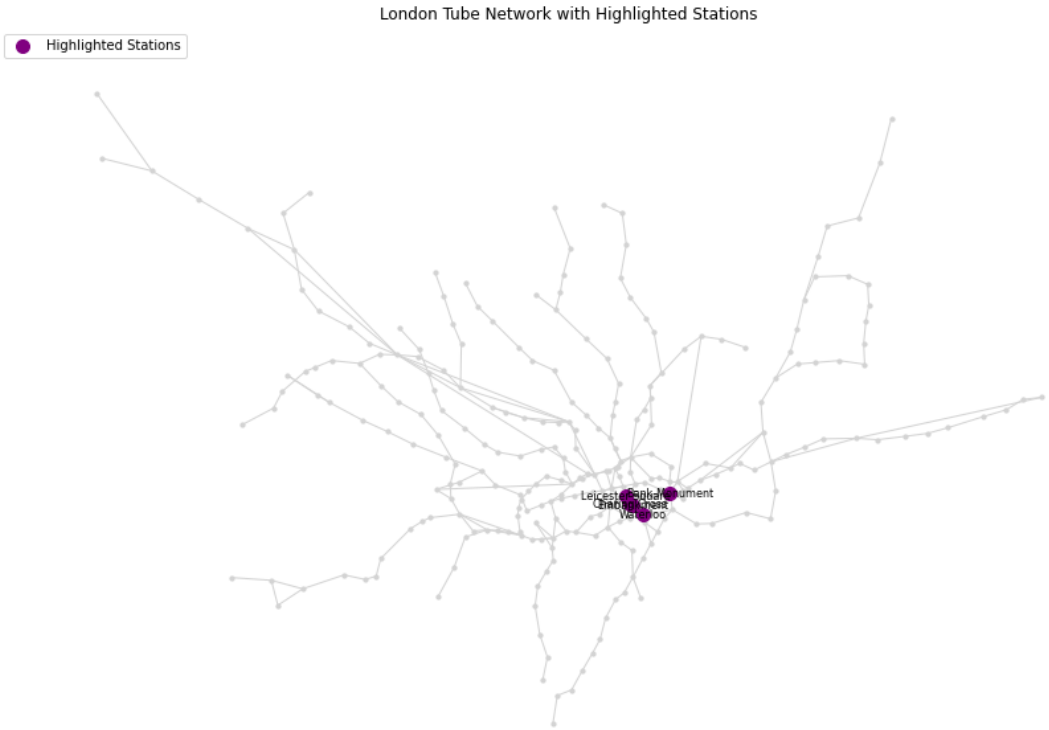


In [46]:
```
print("Top 5 nodes with the highest weighted degrees:")
for node, degree in sorted(weighted_degrees.items(), key=lambda x: x
    print(f"Node: {node} ({labels[node]}), Weighted Degree: {degree
```

```
Top 5 nodes with the highest weighted degrees:
Node: 282 (Bank–Monument), Weighted Degree: 7.79
Node: 406 (Leicester Square), Weighted Degree: 7.18
Node: 344 (Embankment), Weighted Degree: 6.36
Node: 314 (Charing Cross), Weighted Degree: 5.92
Node: 515 (Waterloo), Weighted Degree: 5.77
```

Meanwhile the top stations with a weighted degree have less significance as the closeness of a station to other stations doesnt necessarily make it more central, even though the spacing of stations in the center becomes less dense, yielding higher weights for stations in the geographical center of the city.

In [47]:
```
#Get top 5 nodes by degree weighted
top_5_stations_g = sorted(weighted_degrees.items(), key=lambda x: x
top_5_degree_nodes = [node for node, degree in top_5_stations_g]


#Plot highlighting these top degree nodes
plot_highlighted_stations(G, pos_geo, top_5_degree_nodes, labels)
```



Closeness Centrality

The closeness centrality, being 1 over the sum of (shortest) distances of the one node to every other node in the system, gives a metric that defines centrality in the sense of closeness to all other nodes in the system. The top stations with that metric thus are very close to each other and in a sense in the "center of gravity" of the network, roughly corresponding to the city center. In the context of public transport, these however not necessarily coincide with the most important stations as in this case, the top 5 stations yielded do not correspond with the hub stations that the degree analysis enfasizes.

In [48]:
```
# Calculate closeness centrality for all nodes
closeness_centrality = nx.closeness_centrality(G, distance='dist')

# Sort nodes by closeness centrality in descending order
top_closeness_stations = sorted(closeness_centrality.items(), key=la

# Display the top 5 stations with their closeness centrality
print("Top 5 stations with highest closeness centrality:")
for node, centrality in top_closeness_stations:
    station_name = labels.get(node, "Unknown Station")
    print(f"Node: {node} ({labels[node]}), Closeness Centrality: {c

top_closeness_ids = [station[0] for station in top_closeness_statio
plot_highlighted_stations(G, pos_geo, top_closeness_ids, labels)
```

```
Top 5 stations with highest closeness centrality:
Node: 440 (Oxford Circus), Closeness Centrality: 0.0637
Node: 362 (Green Park), Closeness Centrality: 0.0633
Node: 294 (Bond Street), Closeness Centrality: 0.0630
Node: 280 (Baker Street), Closeness Centrality: 0.0626
Node: 445 (Piccadilly Circus), Closeness Centrality: 0.0626
```

Betweenness Centrality: Nodes

With hubs as the most important stations in the network, betweenness centrality as the percentage of shortest paths from all pairs of nodes going through a specifif node is interesting as hubs are expected to have a high value for this measure by connecting different metro lines. As expected, the top 5 stations are similiar to those from the top 5 stations by degree and represent known hub stations where multiple lines connect.

In [49]:
```
# Calculate betweenness centrality for all nodes
betweenness_centrality = nx.betweenness_centrality(G, weight='dist'

# Sort nodes by betweenness centrality in descending order
top_betweenness_stations = sorted(betweenness_centrality.items(), k

# Display the top 5 stations with their betweenness centrality
print("Top 5 stations with highest betweenness centrality:")
for node, centrality in top_betweenness_stations:
    station_name = labels.get(node, "Unknown Station")
    print(f"Node: {node} ({labels[node]}), Betweenness Centrality:



plot_highlighted_stations(G, pos_geo, [station[0] for station in to
```

```
Top 5 stations with highest betweenness centrality:
Node: 280 (Baker Street), Betweenness Centrality: 0.3052
Node: 409 (Liverpool Street), Betweenness Centrality: 0.2675
Node: 400 (King's Cross St. Pancras), Betweenness Centrality: 0.2555
Node: 333 (Earl's Court), Betweenness Centrality: 0.2218
Node: 441 (London Paddington), Betweenness Centrality: 0.2008
```

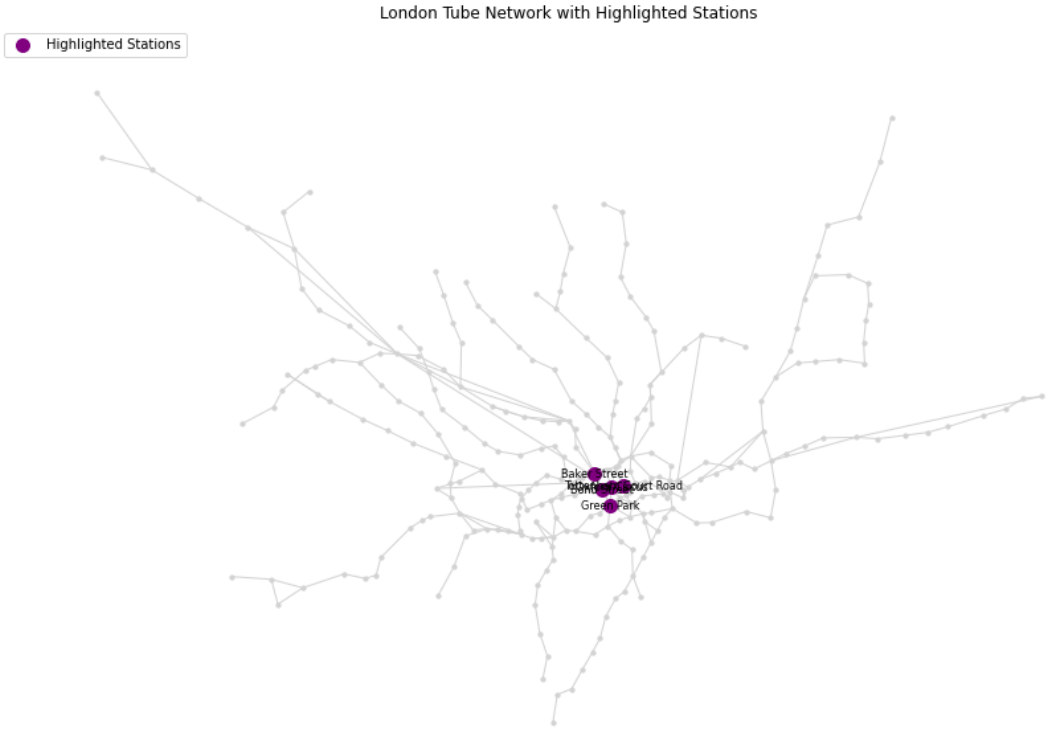
Eigenvector Centrality

Eigenvector centrality as a metric drawn from the Eigenvector-decomposition of the adjecency matrix with the idea of quantifying the importance a node by taking to account the importance of neighboring nodes (For Social Networks: Having few very influencial friends can be as good as having many friends) yields dificult to interprete results for the metro network. The resulting top 5 Stations are not significant as stations in the network and appear to have the main characteristic of being geographically central with 3 stations identical to the stations yielded from closeness centrality. This metric, designed for social network analysis appears to not be fit to make meaningful interpretations to the network at hand.

In [50]:
```
# Calculate eigenvector centrality and get top 5 nodes
eigenvector_centrality = nx.eigenvector_centrality(G, max_iter=1000
top_eigenvector_stations = sorted(eigenvector_centrality.items(), k

print("Top 5 stations with highest eigenvector centrality:")
for node, centrality in top_eigenvector_stations:
    station_name = labels.get(node, "Unknown Station")
    print(f"Node: {node} ({station_name}), Eigenvector Centrality:

plot_highlighted_stations(
    G,
```

```
    pos_geo,
    [node for node, _ in top_eigenvector_stations],
    labels,
    #title="Top 5 Stations by Eigenvector Centrality"
)
```

```
Top 5 stations with highest eigenvector centrality:
Node: 440 (Oxford Circus), Eigenvector Centrality: 0.3744
Node: 294 (Bond Street), Eigenvector Centrality: 0.3369
Node: 362 (Green Park), Eigenvector Centrality: 0.3289
Node: 497 (Tottenham Court Road), Eigenvector Centrality: 0.2764
Node: 280 (Baker Street), Eigenvector Centrality: 0.2738
```



Pagerank

Pagerank, being similar to the degree in this case of an undirected graph similar to degree (in-degree), for undirected graphs similar to the degree, correspondingly yields 3 identical stations to the degree analysis although with a less well interpretable metric. Overall the metrics appear to be low due to low overall connectivity in the network as shown in the analysis of the adjecency matrix.

In [51]:
```
# Calculate PageRank for all nodes
pagerank = nx.pagerank(G)
top_pagerank_stations = sorted(pagerank.items(), key=lambda x: x[1]

print("Top 5 stations with highest PageRank:")
for node, rank in top_pagerank_stations:
    station_name = labels.get(node, "Unknown Station")
    print(f"Node: {node} ({station_name}), PageRank: {rank:.4f}")

# Plot
```

```
top_pagerank_ids = [station[0] for station in top_pagerank_stations
plot_highlighted_stations(G, pos_geo, top_pagerank_ids, labels)
```

```
Top 5 stations with highest PageRank:
Node: 371 (Harrow-on-the-Hill), PageRank: 0.0097
Node: 400 (King's Cross St. Pancras), PageRank: 0.0088
Node: 280 (Baker Street), PageRank: 0.0079
Node: 441 (London Paddington), PageRank: 0.0079
Node: 333 (Earl's Court), PageRank: 0.0073
```

Betweenness Centrality: Edges

As done previously for nodes, for Edges the betweenness centrality can be evaluated finding the most used edges when going from all nodes to every other nodes via the shortest path. While this might give insights into the most used network sections for a evenly used transportation network, overall the meaning of this network is not as significant as for the nodes as the traffic on specific edges will depend more on the points of interest in the city of london than on the theoretical shortest paths to all other points. None the less, unsurprisingly, many of the top 5 Edges start/end in a previously as central identified node.

In [52]:
```
# Calculate top 5 Edges
edge_betweenness_centrality = nx.edge_betweenness_centrality(G)
top_betweenness_edges = sorted(edge_betweenness_centrality.items(),

print("Top 5 edges with highest betweenness centrality:")
for edge, centrality in top_betweenness_edges:
    station_1 = labels.get(edge[0], "Unknown Station")
    station_2 = labels.get(edge[1], "Unknown Station")
    print(f"Edge: {edge} (Connecting {station_1} and {station_2}), |
```

```
Top 5 edges with highest betweenness centrality:
Edge: ('280', '341') (Connecting Baker Street and Edgware Road), Bet
weenness Centrality: 0.2545
Edge: ('341', '441') (Connecting Edgware Road and London Paddingto
n), Betweenness Centrality: 0.2515
Edge: ('280', '294') (Connecting Baker Street and Bond Street), Betw
eenness Centrality: 0.2094
Edge: ('331', '441') (Connecting Ealing Broadway and London Paddingt
on), Betweenness Centrality: 0.1911
Edge: ('280', '360') (Connecting Baker Street and Great Portland Str
eet), Betweenness Centrality: 0.1722
```

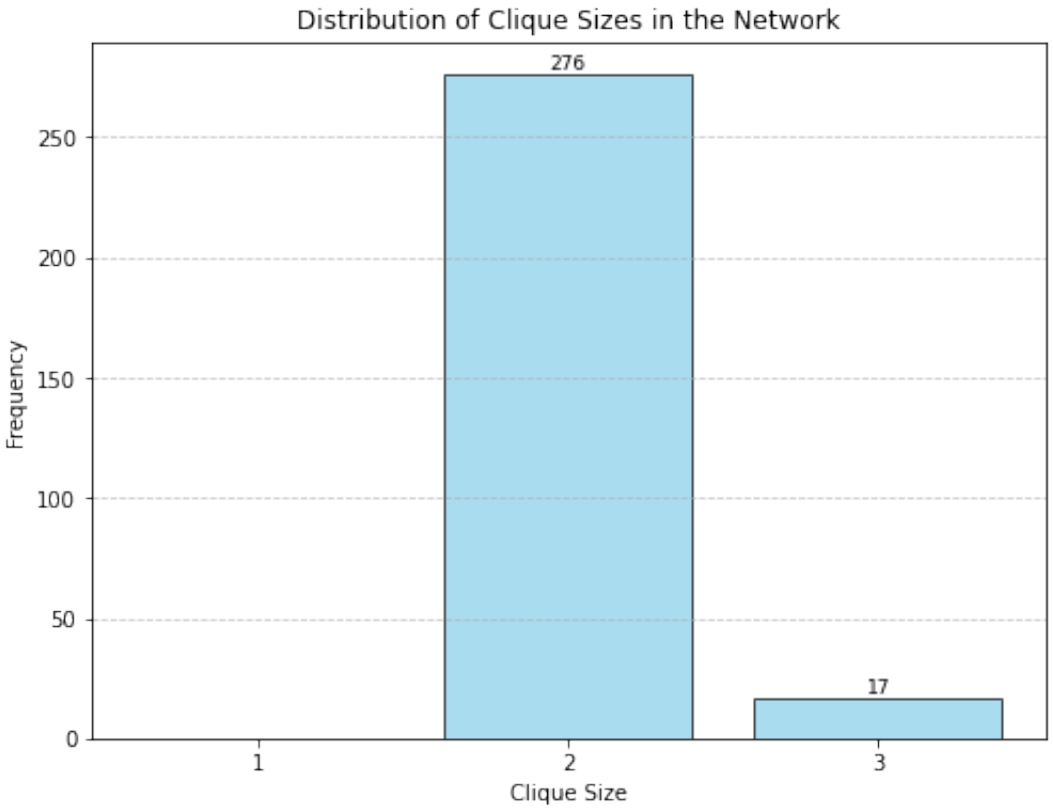
Network Cohesion

Local Density

Distribution of Clique Sizes

Looking at the distribution of Clique sizes, we find that the large majority are 2-cliques, in line with the known tree sturcture of the network. In addition to that, we find a small number of 3 cliques which correspond to central stations where Network density increases.

In [53]:
```
# Get cliques
cliques = list(nx.find_cliques(G))
clique_sizes = [len(clique) for clique in cliques]

# plot dist
plt.figure(figsize=(8, 6))
bars = plt.bar(range(1, max(clique_sizes) + 1),
               [clique_sizes.count(i) for i in range(1, max(clique_
               width=0.8,
               edgecolor='black',
               color='skyblue',
               alpha=0.7)

# cosmetics
plt.title('Distribution of Clique Sizes in the Network')
plt.xlabel('Clique Size')
plt.ylabel('Frequency')
plt.xticks(range(1, max(clique_sizes) + 1))
plt.grid(axis='y', linestyle='--', alpha=0.7)

for bar in bars:
    height = bar.get_height()
    if height > 0:
        plt.text(bar.get_x() + bar.get_width() / 2, height + 0.5, s
plt.show()
```

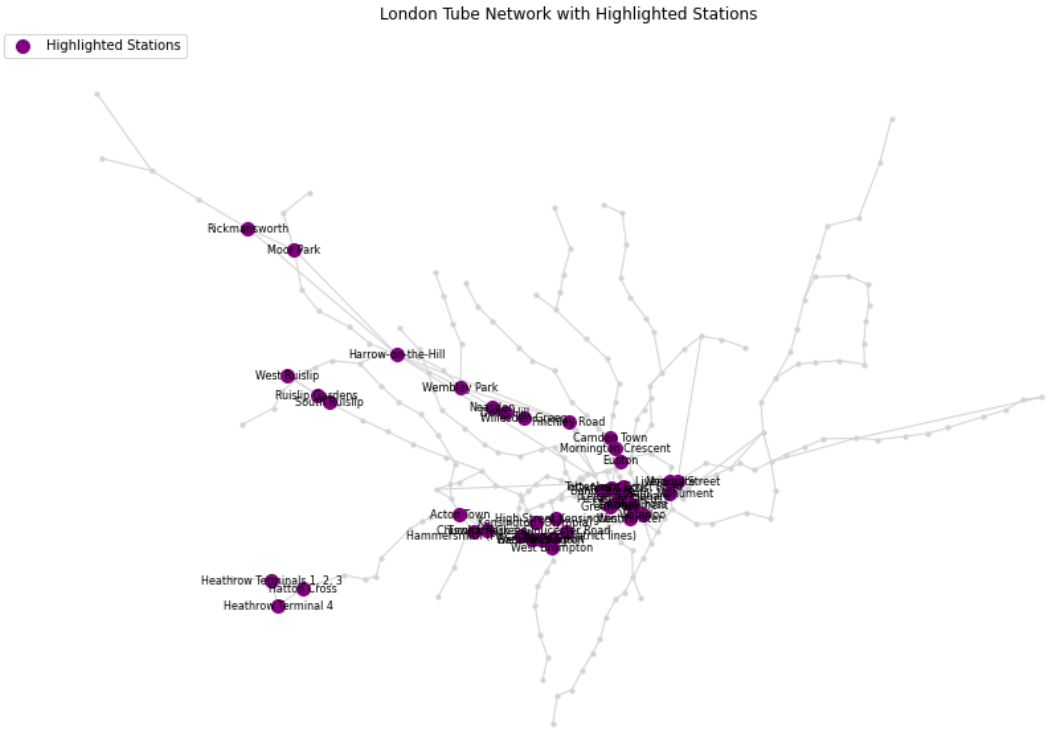


In [54]:
```
### CHECK IF CORRECT ?!?

# Find all maximal cliques in the graph
maximal_cliques = list(nx.find_cliques(G))

# Calculate the size of each clique
clique_sizes = [len(clique) for clique in maximal_cliques]

# Find the largest clique(s)
max_clique_size = max(clique_sizes)
largest_cliques = [clique for clique in maximal_cliques if len(cliq

# Print the results
print(f"Number of maximal cliques: {len(largest_cliques)}")
print(f"Size of the largest clique: {max_clique_size}")
print(f"Largest clique(s): {largest_cliques}")
```

```
Number of maximal cliques: 17
Size of the largest clique: 3
Largest clique(s): [['330', '531', '423'], ['344', '528', '515'], ['
393', '333', '520'], ['346', '422', '305'], ['445', '362', '440'],
['445', '314', '406'], ['380', '355', '333'], ['317', '271', '502'],
['463', '526', '478'], ['294', '440', '362'], ['294', '440', '497'],
['374', '373', '372'], ['502', '365', '271'], ['409', '282', '419'],
['525', '333', '286'], ['371', '459', '420'], ['371', '351', '518']]
```

As discussed before, we find the 3 cliques in centric locations and in areas of higher density like the airport London-Heathrow.

In [55]:
```
nodes_in_largest_cliques = set(node for clique in largest_cliques f
plot_highlighted_stations(G, pos_geo, list(nodes_in_largest_cliques
```

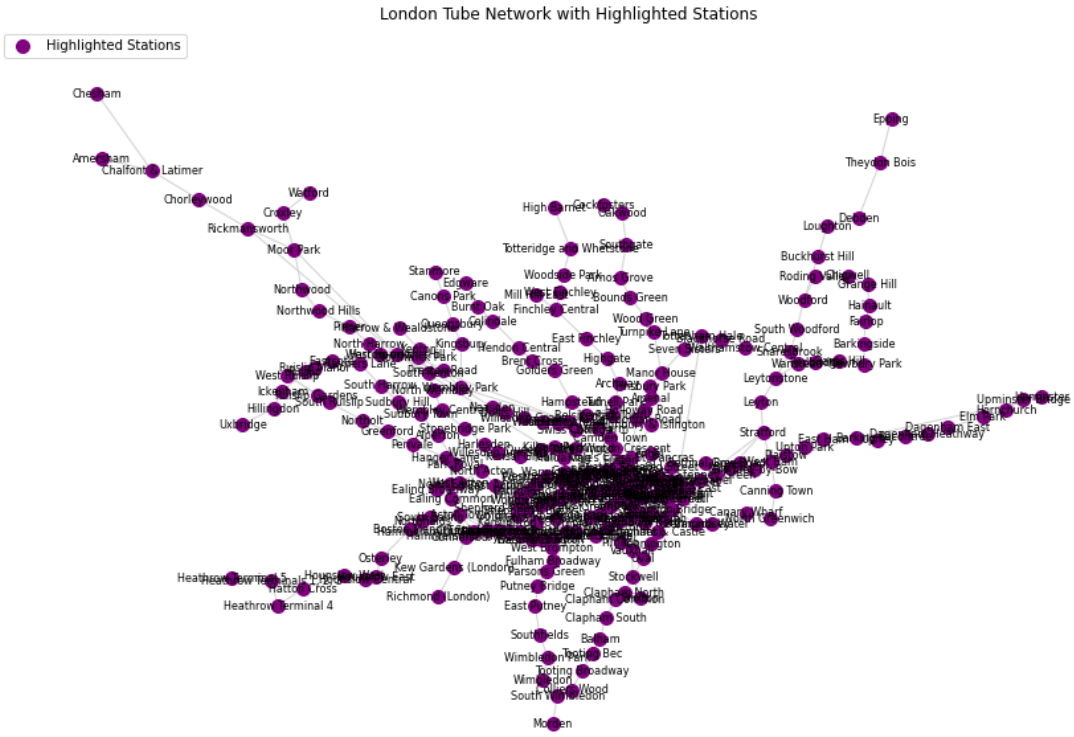


Graph Coreness (K-Cores)

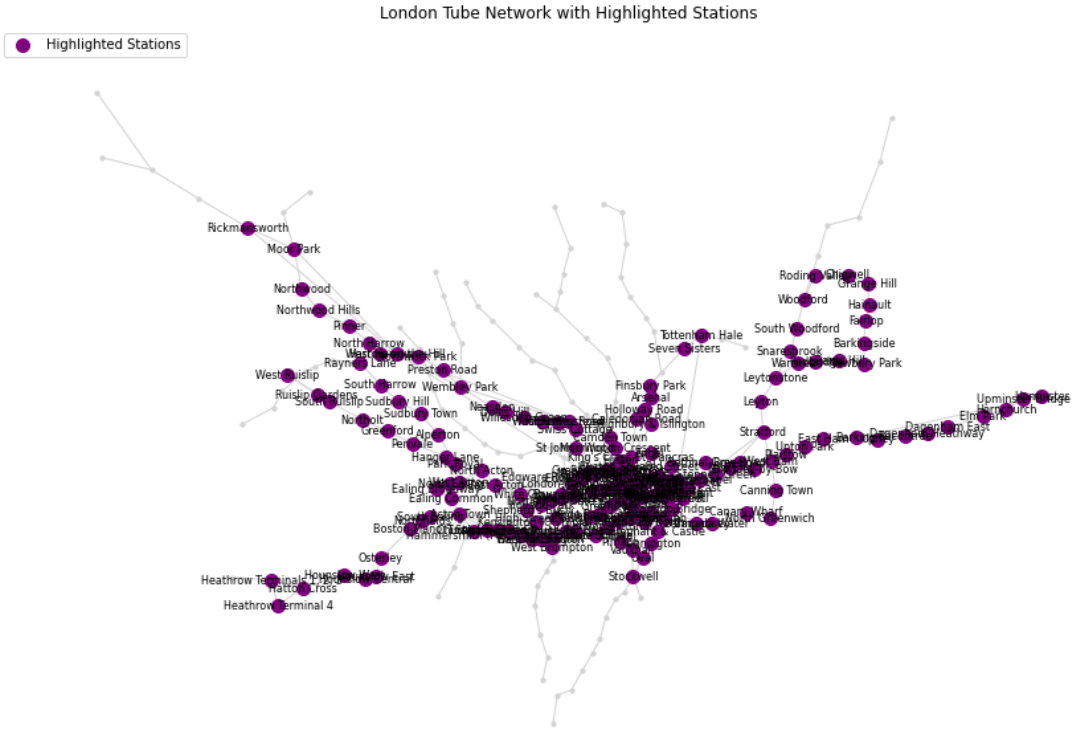
When evaluating the Graph Coreness, we find as expected that only subgraphs up to k=2 can be created, in line with the tree structure of the Network. The existing stations with higher degree (hubs) are isolated as they connect to each other by lower degree stations, meaning they can not generate a subgraph. This gives a intuitive explonation of why metro systems are highly sensitive to break downs, as there is a large amount of critical points that, if cut, will reduce connectivity significantly.

In [56]:
```
# Calculate the k-core for different values of k
k_core_dict = {}
for k in range(1, 4):
    k_core = nx.k_core(G, k=k)
    k_core_dict[k] = k_core

# Print the number of nodes in each k-core
print("Number of nodes in each k-core:")
for k, subgraph in k_core_dict.items():
    print(f"k = {k}: {len(subgraph.nodes())} nodes")

    # plot
    stations_kn = list(k_core_dict[k].nodes())
    plot_highlighted_stations(G, pos_geo, stations_kn, labels)
```

```
Number of nodes in each k-core:
k = 1: 268 nodes
```

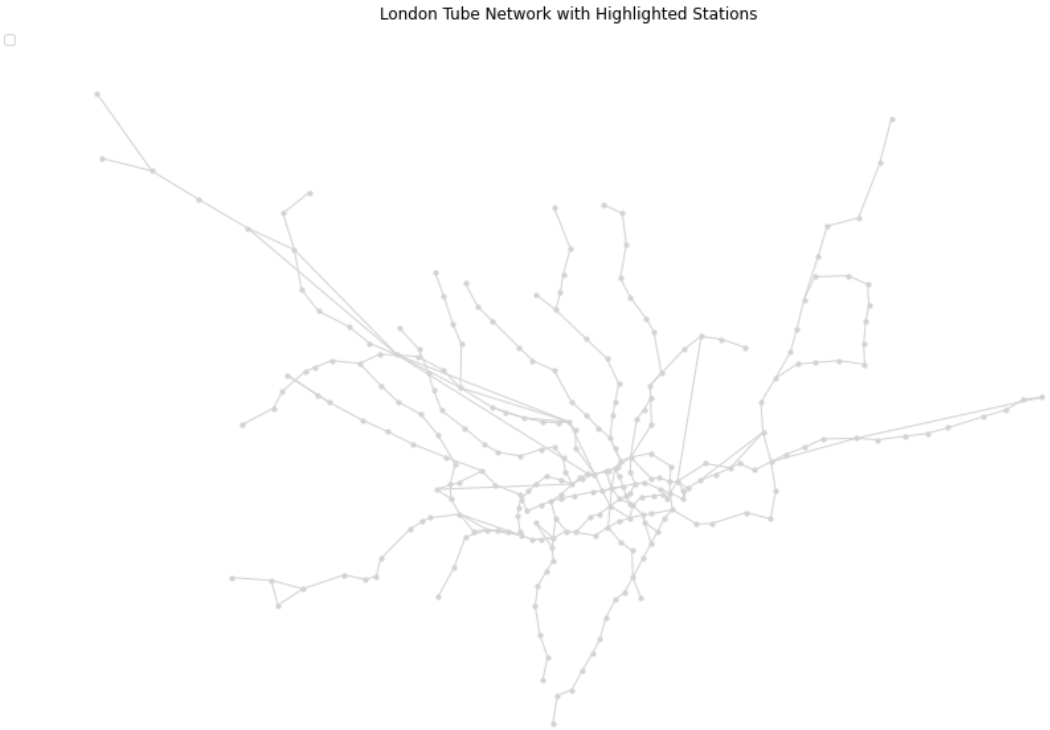


k = 2: 175 nodes



k = 3: 0 nodes

```
/var/folders/2j/t8d8dcsn1tg5sc8l4kz4_4ch0000gn/T/ipykernel_4188/180384376.py:15: UserWarning: No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
  plt.legend(scatterpoints=1, loc='upper left', fontsize=10)
```



Density The the density of the network is small, in line with the tree like structure and the majority of nodes having degree 2 as discussed in relation to the adjecency Matrix.

In [57]:
```
# Calculate the density of the network
density = nx.density(G)
print(f"Density of the network: {density:.4f}")
```

```
Density of the network: 0.0091
```

Clustering Coefficent

With low density, the clustering coefficient as existing triangles devided by possible triangles in the network is very low, with only a very low number of 3-cliques in the network. This is in line with the low connectivity already identified

In [58]:
```
# Calculate the global clustering coefficient (transitivity) of the
global_clustering_coefficient = nx.transitivity(G)
print(f"Global clustering coefficient of the network: {global_clust
```

```
Global clustering coefficient of the network: 0.0813
```

Small World Property

With a low clustering coefficient but a fairly long average shortest path, we consider the network not to hold the small world property. This again is in line with the tree structure, meaning that many stations have to be passed to reach the outer areas of the metro network, with no direct connections available. This might however change if additional transportation networks like

regional trains or express busses would be considered.

In [59]:
```
actual_avg_shortest_path_length = nx.average_shortest_path_length(G
print(f"Average Shortest Path Length: {actual_avg_shortest_path_len
```

```
Average Shortest Path Length: 12.4380
```

Community Detection

The objective of the following section is to define the optimal graph partition. This will be measured using modularity, which quantifies the quality of the partition. A higher modularity value indicates a better partition, with values approaching 1 representing strong community structure, while values closer to -1 suggest poor or weak partitions.

Optimal Modularity: *Alternatively* **Greddy Modularity Algorithm**

As previously mentioned, the goal is to identify the partition that maximizes modularity. The following code implements the optimal algorithm. However, due to the size of the London Underground network, running this algorithm has an exponential time complexity, making it computationally expensive.

In [60]:
```
import igraph as ig
import networkx as nx

# #If G is a networkx graph
# G_nx = G

# #Convert to igraph
# G_ig = ig.Graph.TupleList(G_nx.edges(), directed=False)

# #Run community detection
# clusters = G_ig.community_optimal_modularity()

# #rint community membership
# print(clusters.membership)
```

As previously mentioned, the goal is to identify the partition that maximizes modularity. The following code implements the optimal algorithm. However, due to the size of the London Underground network, running this algorithm has an exponential time complexity, making it computationally expensive.

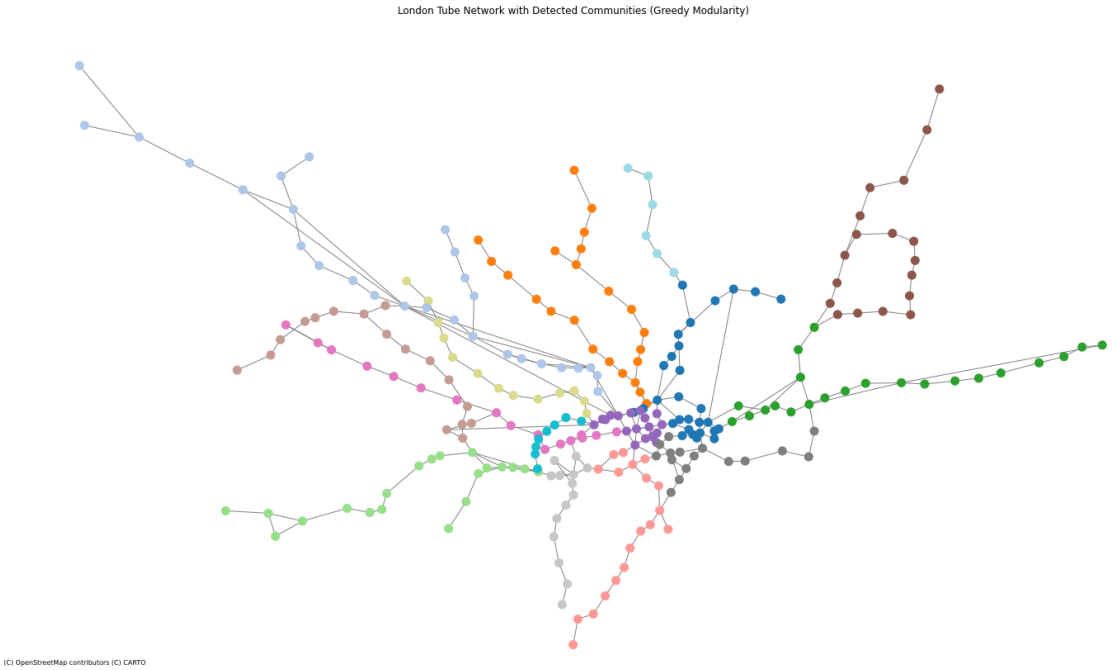
Therefore, the next algorithm to be used is Greedy Modularity, which aims to efficiently approximate the division of the network into communities by maximizing modularity in a computationally feasible manner. Although it does not guarantee the same result as the optimal modularity algorithm, it is suitable for large networks like ours and provides an acceptable level of accuracy.

The algorithm detected 15 communities, a reasonable number considering that the London Underground transportation network is based on metro lines. As mentioned in the introduction, the network consists of 11 metro lines, as well as other lines (such as Light Rail and Regional Lines). The modularity value of 0.8192 further suggests a good construction of communities within the network.

Interestingly, when analyzing the graph, we observe that the main difference between the communities and the real metro lines is that, once reaching the center of the map, the communities diverge into different groups. This contrasts with the real metro lines, which extend from one extreme of the city to the other. This is an intriguing feature of the algorithm, as it identifies communities that are not necessarily aligned with the physical metro line structure.

In [61]:
```
from networkx.algorithms.community import greedy_modularity_communi
from networkx.algorithms.community.quality import modularity
import matplotlib.pyplot as plt
import contextily as ctx
import numpy as np

# Compute communities using greedy modularity
communities = list(greedy_modularity_communities(G))

#Assign community labels to nodes
community_mapping = {node: idx for idx, community in enumerate(comm
nx.set_node_attributes(G, community_mapping, 'community')

# Print modularity info
print(f"Number of communities detected: {len(communities)}")
print(f"Modularity: {modularity(G, communities):.4f}")

# Get community colors for plotting
colors = [community_mapping[node] for node in G.nodes()]

# Plot network with basemap
fig, ax = plt.subplots(figsize=(25, 25))

# Add map of London
ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.CartoDB.P

# Overlay network with community colors
nx.draw(G, pos=pos_geo, with_labels=False, node_size=100,
        node_color=colors, cmap=plt.cm.tab20, edge_color='gray', ax

# Adjust map settings
ax.set_xlim(-0.68, 0.29)
ax.set_ylim(51.39, 51.73)
ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))  # Latitu

plt.title("London Tube Network with Detected Communities (Greedy Mo
plt.show()
```

```
Number of communities detected: 15
Modularity: 0.8192
```



Fast Greedy Algorithm

To simplify the integer programming proposed by the optimal modularity algorithm, which cannot be run on our network due to its computational complexity, the Fast Greedy algorithm will be applied to the London Underground transportation network.

In this case, the algorithm yields a slightly better modularity value than the previous Greedy Modularity algorithm. However, both algorithms converge to detect 15 communities, with a similar pattern in the layout map. The communities are primarily located in the parts of the metro lines outside the city center, while the stations in the city center form distinct communities, possibly based on their connectivity to other stations.

In [62]:
```
# Convert NetworkX graph to iGraph
edges = list(G.edges())
ig_graph = ig.Graph.TupleList(edges, directed=False)

# Run Fast Greedy community detection
communities = ig_graph.community_fastgreedy()
clusters = communities.as_clustering()

# Map igraph vertex IDs back to NetworkX node labels
# Assumes node labels in NetworkX are integers or strings matching
ig_to_nx = {i: v["name"] for i, v in enumerate(ig_graph.vs)}
nx_community_mapping = {ig_to_nx[i]: clusters.membership[i] for i i
nx.set_node_attributes(G, nx_community_mapping, 'community')

# Print results
print(f"Number of communities detected: {len(clusters)}")
print(f"Modularity: {clusters.modularity:.4f}")
```

```
# Plot over map using pos_geo and community colors
colors = [nx_community_mapping[node] for node in G.nodes()]

fig, ax = plt.subplots(figsize=(25, 25))
ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.CartoDB.P

nx.draw(
    G, pos=pos_geo, with_labels=False, node_size=100,
    node_color=colors, cmap=plt.cm.tab20, edge_color='gray', ax=ax
)

ax.set_xlim(-0.68, 0.29)
ax.set_ylim(51.39, 51.73)
ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))

plt.title("London Tube Network with Fast Greedy Communities (iGraph
plt.show()
```

```
Number of communities detected: 15
Modularity: 0.8196
```

Louvain Algorithm

As an alternative, the Louvain Algorithm can also be used to find the partition that maximizes modularity without the need to run an optimization algorithm.

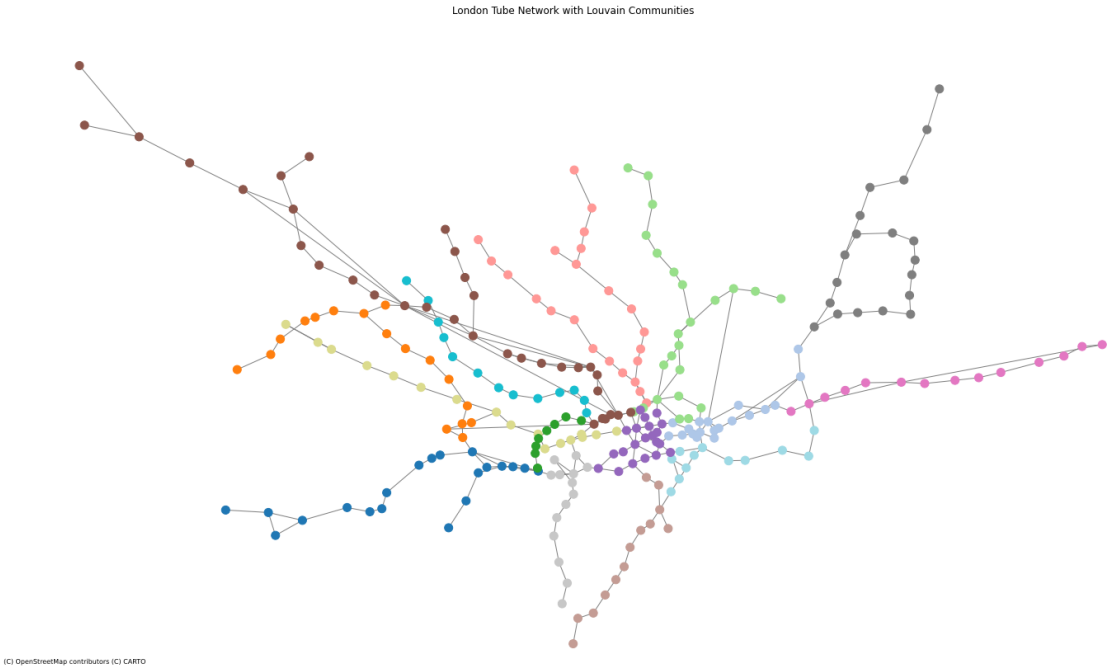
Once again, the Louvain algorithm detects 15 communities, consistent with the results from the previous two algorithms. However, the modularity increases to 0.823, which is a positive indicator of the algorithm's ability to accurately detect communities. This value surpasses the modularity obtained by the Greedy Modularity and Fast Greedy algorithms.

Similarly, the communities identified by the Louvain algorithm appear to represent the "arms" of the network. These can be interpreted as the metro

lines or stations located in the outer areas of the city center. Within the city center, one to three communities are identified, possibly based on their connectivity to key junction stations that serve as transfer points for passengers.

In [63]:
```
import community.community_louvain as community_louvain
import matplotlib.pyplot as plt
import contextily as ctx
import numpy as np

# Perform Louvain community detection
partition = community_louvain.best_partition(G, weight='weight')  #

# Assign community info to graph nodes
nx.set_node_attributes(G, partition, 'community')

# Print stats
num_communities = len(set(partition.values()))
print(f"Number of communities detected: {num_communities}")

# Compute modularity
modularity_value = community_louvain.modularity(partition, G, weigh
print(f"Modularity of the detected community structure: {modularity_

# Get node colors by community
community_colors = [partition[node] for node in G.nodes()]

# Plot on London map
fig, ax = plt.subplots(figsize=(25, 25))
ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.CartoDB.P

nx.draw(
    G,
    pos=pos_geo,
    node_color=community_colors,
    with_labels=False,
    node_size=100,
    cmap=plt.cm.tab20,
    edge_color='gray',
    ax=ax
)

# Adjust map settings
ax.set_xlim(-0.68, 0.29)
ax.set_ylim(51.39, 51.73)
ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))

plt.title("London Tube Network with Louvain Communities")
plt.show()
```

```
Number of communities detected: 15
Modularity of the detected community structure: 0.8209
```



Label Propagation Algorithm

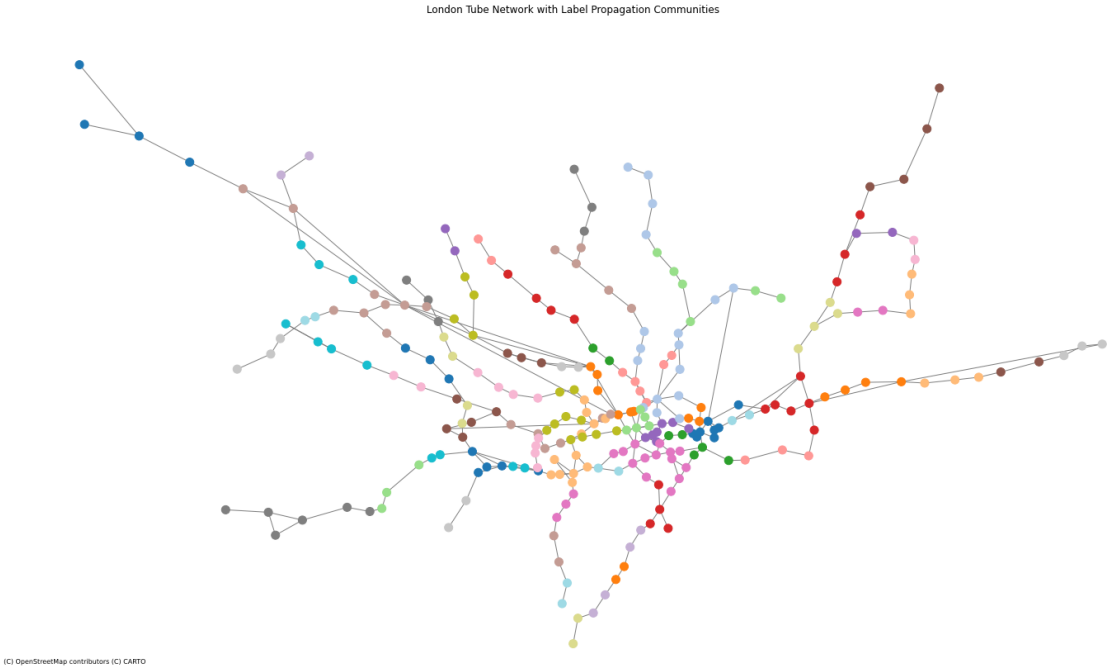
The Label Propagation Algorithm takes into account the neighbors of each vertex for community detection, making it a useful option for large networks, such as the one in this analysis of the London Underground transport network.

The result of applying this algorithm yields 77 communities with a significantly reduced modularity of 0.62, which indicates worse community detection compared to the previous algorithms. Additionally, the large number of communities, 77, is not well-defined, as there are no clear breaks in the network that would correspond to such a fragmented division of metro lines. This suggests that the algorithm is identifying communities based on smaller groups of stations, with each community containing roughly 4 stations. This is likely due to the algorithm's tendency to over-segment the network, possibly focusing on local neighborhoods in the city rather than larger areas of London.

In [64]:
```
from networkx.algorithms.community import label_propagation_communi
from networkx.algorithms.community.quality import modularity

# Detect communities using Label Propagation
communities = list(label_propagation_communities(G))

# Assign community labels to each node
community_mapping = {node: idx for idx, community in enumerate(comm
nx.set_node_attributes(G, community_mapping, 'community')

# Generate color list for nodes
community_colors = [community_mapping[node] for node in G.nodes()]

# Print results
print(f"Number of communities detected: {len(communities)}")

# Compute modularity (optional, but informative)
```

```
modularity_value = modularity(G, communities)
print(f"Modularity of the detected community structure: {modularity_

# Plot over London map
fig, ax = plt.subplots(figsize=(25, 25))
ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.CartoDB.P

nx.draw(
    G,
    pos=pos_geo,
    node_color=community_colors,
    with_labels=False,
    node_size=100,
    cmap=plt.cm.tab20,
    edge_color='gray',
    ax=ax
)

# Adjust plot bounds
ax.set_xlim(-0.68, 0.29)
ax.set_ylim(51.39, 51.73)
ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))

plt.title("London Tube Network with Label Propagation Communities")
plt.show()
```

```
Number of communities detected: 77
Modularity of the detected community structure: 0.6271
```



Edge Betweenness Algorithm

The Edge Betweenness Algorithm results in 2 communities in the network, with a much lower modularity of 0.4194. This is not optimal when compared to the results of the other algorithms. Additionally, the communities identified are quite large and seem to correspond to a division between the South West and North East parts of the network, according to the map layout.

This result stems from the algorithm's method of creating communities by iteratively removing central edges. As determined by the earlier centrality measures, these central edges are predominantly located in downtown London. These edges have the highest edge betweenness centrality. In contrast to other algorithms, which identified this central area as being split into one to three communities, the Edge Betweenness Algorithm focuses on this central region as a point of separation, resulting in the identified two large communities.

In [65]:
```
from networkx.algorithms.community import girvan_newman
import matplotlib.pyplot as plt
import contextily as ctx
import numpy as np


# Run Girvan-Newman and get first level of split
communities_generator = girvan_newman(G)
first_level_communities = next(communities_generator)
communities = [set(community) for community in first_level_communit

# Assign each node to a community
community_mapping = {node: idx for idx, community in enumerate(comm
nx.set_node_attributes(G, community_mapping, 'community')

# Print number of communities
print(f"Number of communities detected: {len(communities)}")

from networkx.algorithms.community.quality import modularity

# Compute and print modularity
modularity_value = modularity(G, communities)
print(f"Modularity of the detected community structure: {modularity_

# Get node colors
community_colors = [community_mapping[node] for node in G.nodes()]

# Plot over London map
fig, ax = plt.subplots(figsize=(25, 25))
ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.CartoDB.P

nx.draw(
    G,
    pos=pos_geo,
    node_color=community_colors,
    with_labels=False,
    node_size=100,
    cmap=plt.cm.tab20,
    edge_color='gray',
    ax=ax
)

#Map settings
ax.set_xlim(-0.68, 0.29)
```

```
ax.set_ylim(51.39, 51.73)
ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))

plt.title("London Tube Network with Girvan–Newman Communities (Edge
plt.show()
```

```
Number of communities detected: 2
Modularity of the detected community structure: 0.4194
```

Walktrap Algorithm

The Walktrap Algorithm detects communities by simulating a large number of random walks between vertices. In this case, the algorithm identifies 33 communities with a modularity of 0.77, which is an acceptable value but still lower than the modularity found in previous algorithms.

The relatively high number of communities can be attributed to the algorithm's tendency to create clusters with small sizes. This is a characteristic feature of the Walktrap algorithm, which tends to identify finer-grained communities rather than larger, more broadly defined ones, unlike the Label Propagation Algorithm which resulted in a much higher number of communities. While the communities detected here are smaller and more numerous, they are not as detailed as those produced by the Label Propagation Algorithm.

One interesting aspect of the Walktrap Algorithm is its reliance on random walks, which simulate the likelihood of vertices being part of the same community based on their proximity to each other in the graph, also it allows the algorithm to capture finer distinctions between groups.

In [66]:
```
import matplotlib.pyplot as plt
import contextily as ctx
import numpy as np
import igraph as ig
```

```
from networkx.algorithms.community.quality import modularity

#Convert NetworkX graph to iGraph
edges = list(G.edges())
ig_graph = ig.Graph.TupleList(edges, directed=False)

# Run Walktrap community detection
walktrap_communities = ig_graph.community_walktrap(weights=None)
clusters = walktrap_communities.as_clustering()

# Map iGraph indices back to NetworkX node IDs
ig_to_nx = {i: v["name"] for i, v in enumerate(ig_graph.vs)}
nx_community_mapping = {ig_to_nx[i]: clusters.membership[i] for i i
nx.set_node_attributes(G, nx_community_mapping, 'community')

# Format communities as list of sets for modularity calculation
num_communities = len(clusters)
print(f"Number of communities detected: {num_communities}")
communities = [[] for _ in range(num_communities)]
for node, com_id in nx_community_mapping.items():
    communities[com_id].append(node)
communities = [set(c) for c in communities]

# Compute modularity
modularity_value = modularity(G, communities)
print(f"Modularity of the detected community structure: {modularity_

# Get node colors
community_colors = [nx_community_mapping[node] for node in G.nodes(

# Plot on London map
fig, ax = plt.subplots(figsize=(25, 25))
ctx.add_basemap(ax, crs="EPSG:4326", source=ctx.providers.CartoDB.P

nx.draw(
    G,
    pos=pos_geo,
    node_color=community_colors,
    with_labels=False,
    node_size=100,
    cmap=plt.cm.tab20,
    edge_color='gray',
    ax=ax
)

# Map settings
ax.set_xlim(-0.68, 0.29)
ax.set_ylim(51.39, 51.73)
ax.set_aspect(1 / np.cos(np.radians((51.4 + 51.73) / 2)))

plt.title("London Tube Network with Walktrap Communities")
plt.show()
```

```
Number of communities detected: 33
Modularity of the detected community structure: 0.7719
```

Summary Community Detection Methods

After running several community detection algorithms, the resulting communities were visually reviewed to assess their characteristics. The first three algorithms listed in the table below all identified the same number of communities, 15, which is a reasonable figure given the number of metro lines in the London Underground network and its overall structure.

Among these, the Louvain algorithm produced the highest modularity value (0.8235),by a small margin, indicating the most optimal partitioning of the network. This high modularity suggests a strong alignment between the detected communities and the underlying structure of the network.

The layout maps produced by these three algorithms reveal that the detected communities largely correspond to the "arms" of the network metro lines extending outward from the city center. These algorithms effectively capture the tree-like structure of the network, identifying each arm as a distinct community. However, they fall short in representing the full extent of individual metro lines; instead of grouping opposite arms of the same line into a single community, they treat each arm as a separate entity. Despite this limitation, the algorithms demonstrate a good ability to capture the core structure of the network.

Additionally, communities in the city center represent major interchanging metro stations, which act as key transfer points for passengers. This partitioning reflects a more natural and practical division of the network, aligning with real-world patterns of connectivity and transit usage.

In contrast, the remaining three algorithms struggled to effectively model the network's tree-like structure. They produced inconsistent community counts,

ranging from as few as 2 to as many as 77, indicating difficulty in identifying a coherent partitioning according to the metro lines in the transportation network.

Algorithm	Communities	Modularity
Greedy Modularity Algorithm	15	0.8192
Fast Greedy Algorithm	15	0.8196
Louvain Algorithm	**15**	**0.8235**
Label Propagation Algorithm	77	0.6271
Edge Betweenness Algorithm	2	0.4194
Walktrap Algorithm	33	0.7719

Assortativity

The (degree) assortativity in the Network is low, again due to the fact that high degree stations (hubs) tend to connect to low degree stations rather than directly to other hubs. This is, like discussed previous, due to the tree like structure and the goal of hubs to distribute traffic between arms, rather than connect directly to other hubs.

In [67]:
```
# Calculate the assortativity coefficient of the network
assortativity_coefficient = nx.degree_assortativity_coefficient(G)
print(f"Assortativity Coefficient of the network: {assortativity_co
```

```
Assortativity Coefficient of the network: 0.1679
```