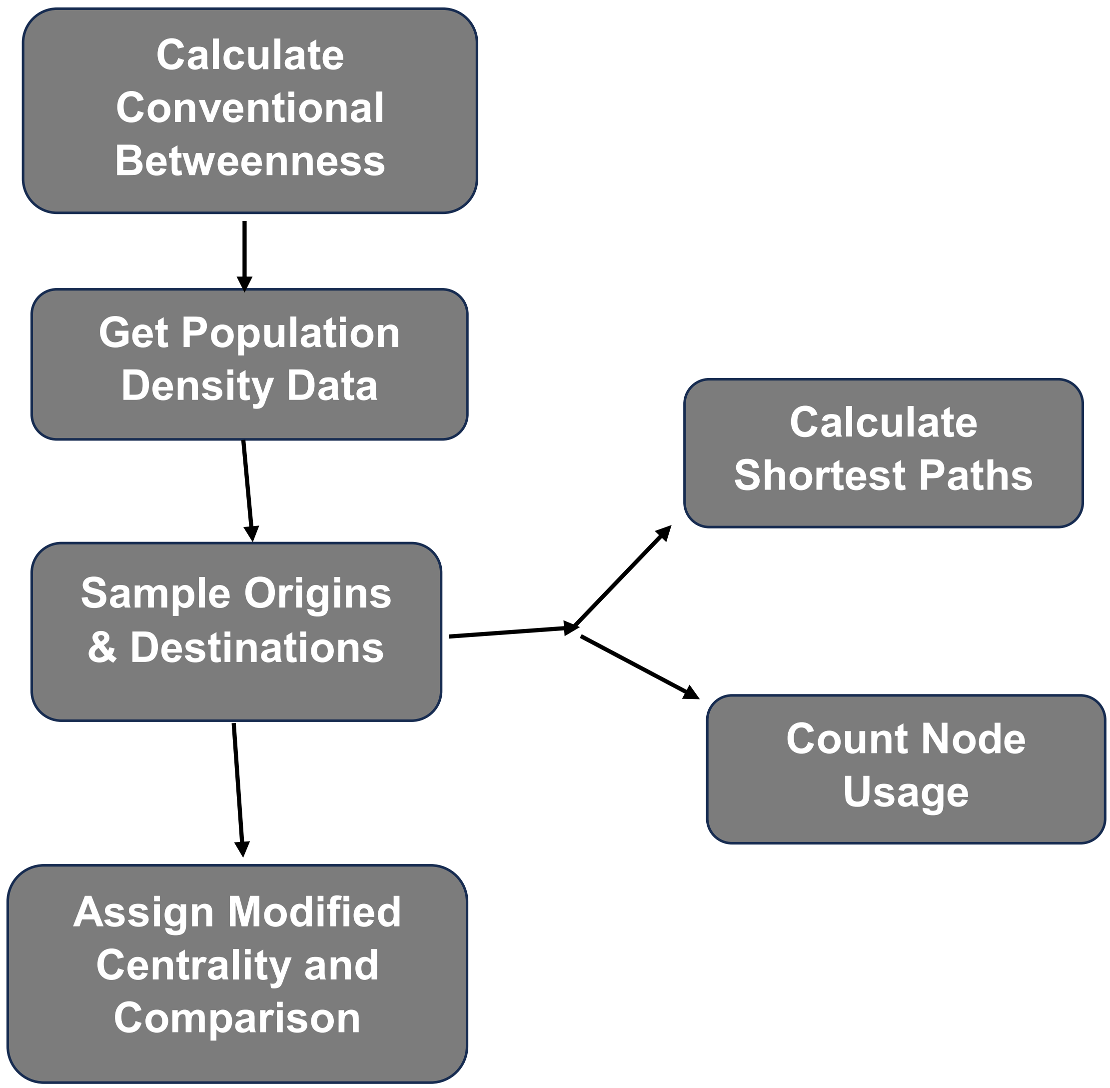
Lab 2

Hsu-Chieh (Jasmine) Ma

Workflow


Calculate Conventional Betweenness
Get Population Density Data
Sample Origins & Destinations
Calculate Shortest Paths
Count Node Usage
Assign Modified Centrality and Comparison

Steps Breakdown

1. Calculate Conventional Betweenness Centrality

- Use OSMnx to extract the road network for the given bounding box.
- Apply NetworkX to compute the conventional betweenness centrality.
- Visualize the results using Matplotlib.

2. Get Population Density Data

- Obtain population density data for the bounding box area.

3. Sample Origins and Destinations

- Based on population density, randomly sample origin and destination points for your network analysis.

4. Calculate Shortest Paths and Count Node Usage

- Using the sampled origins and destinations, calculate the shortest paths between them.
- Count how often each node is used in these paths to reflect the "new" betweenness centrality.

5. Assign Modified Betweenness Centrality and Comparison

- Assign the modified centrality scores based on the node usage counts.
- Compare these results with the conventional betweenness centrality calculated earlier.

Calculate Conventional Betweenness Centrality

Use OSMnx package in Python to extract the road network within the provided bounding box coordinates: (33.86, -84.40, 33.82, -84.34), representing north, south, east, and west, separately. This step collects the necessary nodes and edges to represent the road network in the area of interest.

The extracted graph will contain nodes representing intersections and edges that displaying road segments.

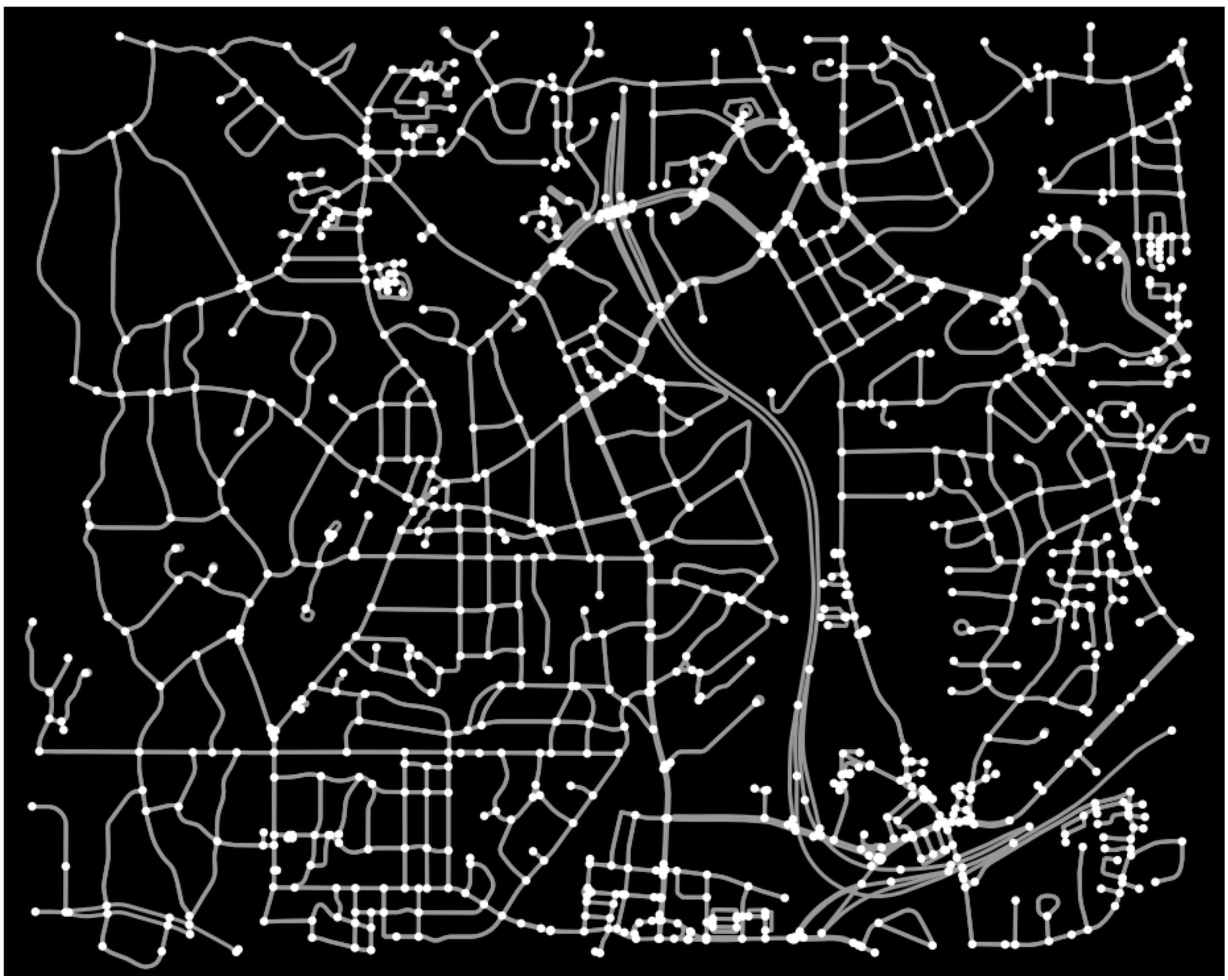


Figure 1: Visualized road network, which nodes are colored based on their betweenness centrality values

Conventional betweenness centrality values are displaying as such (partially).

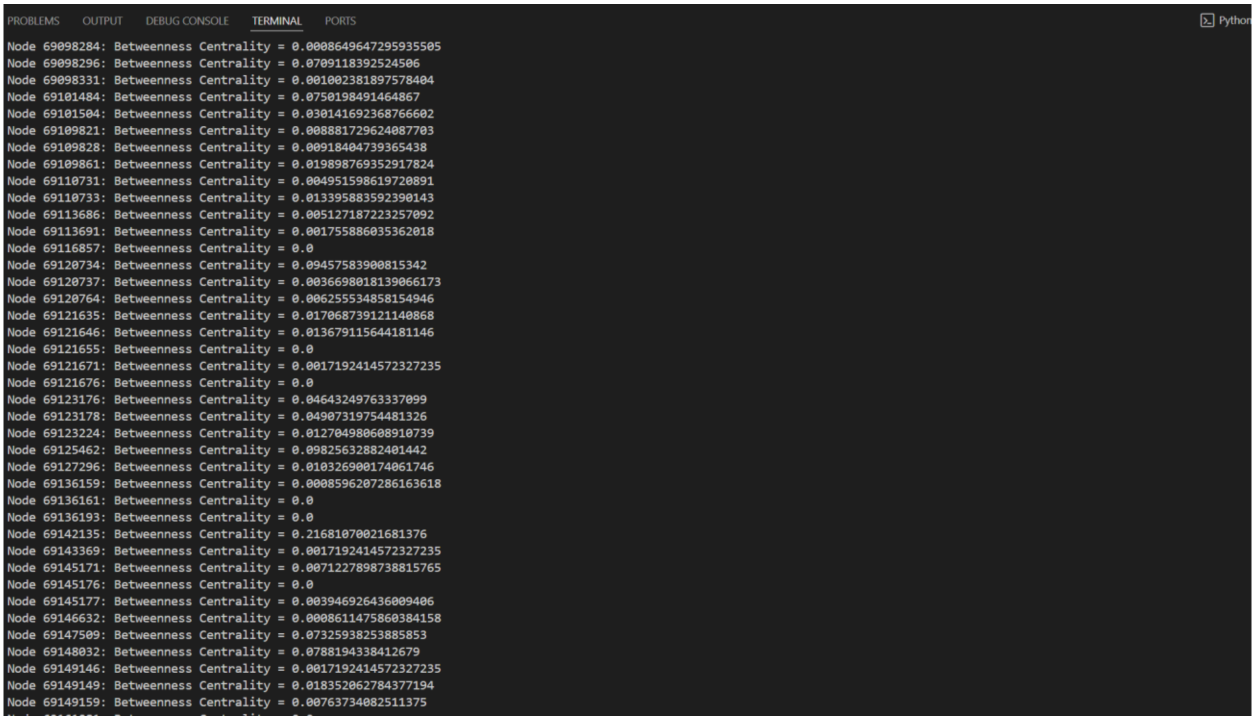


Figure 2: Centrality scores for each node in terminal.

Get Population Density Data

In this step, we will retrieve population density data for the defined bounding box area (33.86, -84.40, 33.82, -84.34) using the U.S. Census Bureau API, as it provides more comprehensive and accurate population data, comparing to OpenStreetMap(OSM). This data will help prioritize areas for selecting trip origins and destinations based on population distribution. Higher population density areas will have a higher probability of being chosen.

```python

import requests
import pandas as pd

# Define Census API key and the bounding box area
API_KEY = '71bbb84546221d6b8404af5a411d009ba4fb40fc'
bounding_box = {"N": 33.86, "S": 33.82, "W": -84.40, "E": -84.34}

# Define the base URL for the Census API ACS 5-Year data
base_url = "https://api.census.gov/data/2020/acs/acs5"

# Define the fields that need to be queried (total population, area for density)
fields = ["B01003_001E"]  # Total population

# Construct the query URL
query_url = f"{base_url}?get={','.join(fields)}&for=tract:*&in=state:13&key={API_KEY}"

# Send the request to the Census API
response = requests.get(query_url)
if response.status_code == 200:
    data = response.json()
    columns = data[0]
    rows = data[1:]

    # Convert the response to a pandas DataFrame
    df = pd.DataFrame(rows, columns=columns)

    # Output the DataFrame for inspection
    print(df.head())
else:
    print(f"Failed to retrieve data. Status code: {response.status_code}")

```

Figure 3: Retrieving population density data from U.S. Census Bureau with Census API

```
...
  B01003_001E state county   tract
0        3025    13    059  001200
1        2100    13    059  001700
1        2100    13    059  001700
2        2200    13    059  001800
3        3239    13    059  001900
4        2066    13    059  002000
```

Figure 4: The population data from Census, tract within the specific bounding box area.

We are aiming to calculate population density for census tracts within a specific bounding box area using U.S. Census data and shapefiles from TIGER/Line. The goal is to integrate population density into the calculation of betweenness centrality, adjusting it to reflect real-world travel patterns. We obtained population data using the Census API (variable B01003_001E for total population, Figure 4) and now have the corresponding land area data (ALAND) from a shapefile. By combining these datasets, we can calculate population density, which we will use to modify our centrality calculations.

```python
import geopandas as gpd

gdf = gpd.read_file("C:/Users/jasmi/Downloads/tl_2024_13_tract/tl_2024_13_tract.shp")
# Calculate population density in people per square kilometer
gdf["Population Density"] = gdf["B01003_001E"] / (gdf["ALAND"] / 1e6)
print(gdf[["B01003_001E", "ALAND", "Population Density"]].head())

# Merge df and gdf on common columns: state, county, and tract
merged_df = df.merge(gdf, left_on=["state", "county", "tract"], right_on=["STATEFP", "COUNTYFP", "TRACTCE"], how="inner")

# Ensure population and land area columns are numeric
merged_df["B01003_001E"] = pd.to_numeric(merged_df["B01003_001E"], errors="coerce")
merged_df["ALAND"] = pd.to_numeric(merged_df["ALAND"], errors="coerce")

# Calculate Population Density in people per square kilometer
merged_df["Population Density"] = merged_df["B01003_001E"] / (merged_df["ALAND"] / 1e6)  # Convert ALAND to sq. km
print(merged_df[["B01003_001E", "ALAND", "Population Density"]].head())
```

ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
>> # Calculate Population Density in people per square kilometer
>>
>> merged_df["Population Density"] = merged_df["B01003_001E"] / (merged_df["ALAND"] / 1e6)  # Convert ALAND to sq. km
>> print(merged_df[["B01003_001E", "ALAND", "Population Density"]].head())
  B01003_001E    ALAND  Population Density
0        3025  3470225          871.701403
1        2100  3481084          603.260364
2        2200  2084584         1055.366442
3        3239  2972604         1089.617050
4        2066  1993638         1036.296459
>>
```

Figure 5: Population Density of the Bounding Box Area

In this section, we merged two datasets: the Census population data (df) and the TIGER/Line shapefile data (gdf). The df contains total population data (B01003_001E), while the gdf includes geographic details such as land area (ALAND). By merging these tables on the common identifiers (state, county, and tract in df with STATEFP, COUNTYFP, and TRACTCE in gdf), we created a unified dataset (merged_df) that associates population counts with corresponding land areas.

To ensure accurate calculations, we converted the B01003_001E and ALAND columns to numeric data types, allowing us to compute population density in people per square kilometer by dividing population by land area (converted to square kilometers). This step integrates population density into our analysis, supporting more refined centrality calculations that consider spatial distribution.

Sample Origins and Destinations

Select multiple origins and destinations to simulate realistic travel patterns across the network. By sampling trip origins and destinations based on population density, more densely populated areas will have a higher likelihood of being chosen as starting points or destinations.

```python
# Ensure latitude and longitude columns are floats
origins["INTPTLAT"] = pd.to_numeric(origins["INTPTLAT"], errors="coerce")
origins["INTPTLON"] = pd.to_numeric(origins["INTPTLON"], errors="coerce")
destinations["INTPTLAT"] = pd.to_numeric(destinations["INTPTLAT"], errors="coerce")
destinations["INTPTLON"] = pd.to_numeric(destinations["INTPTLON"], errors="coerce")

# Drop any rows with NaN values in coordinates to avoid issues
origins.dropna(subset=["INTPTLAT", "INTPTLON"], inplace=True)
destinations.dropna(subset=["INTPTLAT", "INTPTLON"], inplace=True)

# Convert origins and destinations to coordinate lists
origin_coords = list(zip(origins["INTPTLAT"], origins["INTPTLON"]))
destination_coords = list(zip(destinations["INTPTLAT"], destinations["INTPTLON"]))

shortest_paths = []

# Calculate shortest paths for each origin-destination pair
for origin, destination in zip(origin_coords, destination_coords):
    origin_node = ox.distance.nearest_nodes(G, origin[1], origin[0])
    destination_node = ox.distance.nearest_nodes(G, destination[1], destination[0])
    path = nx.shortest_path(G, origin_node, destination_node, weight="length")
    shortest_paths.append(path)

# Skipping nodes that are not connected via try-except methods
for origin, destination in zip(origin_coords, destination_coords):
    try:
        origin_node = ox.distance.nearest_nodes(G, origin[1], origin[0])
        destination_node = ox.distance.nearest_nodes(G, destination[1], destination[0])
        path = nx.shortest_path(G, origin_node, destination_node, weight="length")
        shortest_paths.append(path)
    except nx.NetworkXNoPath:
        print(f"No path between {origin} and {destination}")
        continue
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
...         print(f"No path between {origin} and {destination}")
...         continue
...
No path between (34.3718242, -82.8857632) and (33.7114788, -84.3908033)
No path between (33.7800638, -84.4234369) and (33.7681004, -84.3912905)
>>> 
```

Figure 6: Formatting issues

In this section, I calculated the shortest paths between sampled origin and destination pairs, chosen probabilistically. First, we converted latitude and longitude coordinates to numeric values and removed any rows with N/A values to ensure compatibility for node mapping. Using OSMnx's `nearest_nodes`, we identified the closest network nodes for each origin-destination pair.

By applying a block with “try-except” method to handle disconnected pairs, skipping paths where no connection exists and printing feedback for these cases. This approach ensures we only keep connected paths, preparing our data for accurate centrality calculations based on reachable origin-destination pairs.

Calculate Shortest Paths

For each origin-destination pair, we calculated the shortest paths using Dijkstra's algorithm, focusing on edge weights representing road lengths. This method simulates the most efficient travel routes between sampled nodes, reflecting real-world travel dynamics. To account for disconnected nodes, we implemented a try-except block, which skips and logs cases where no path exists. After testing all results could successfully print out, these calculated paths would be stored in shortest_paths for further analysis.

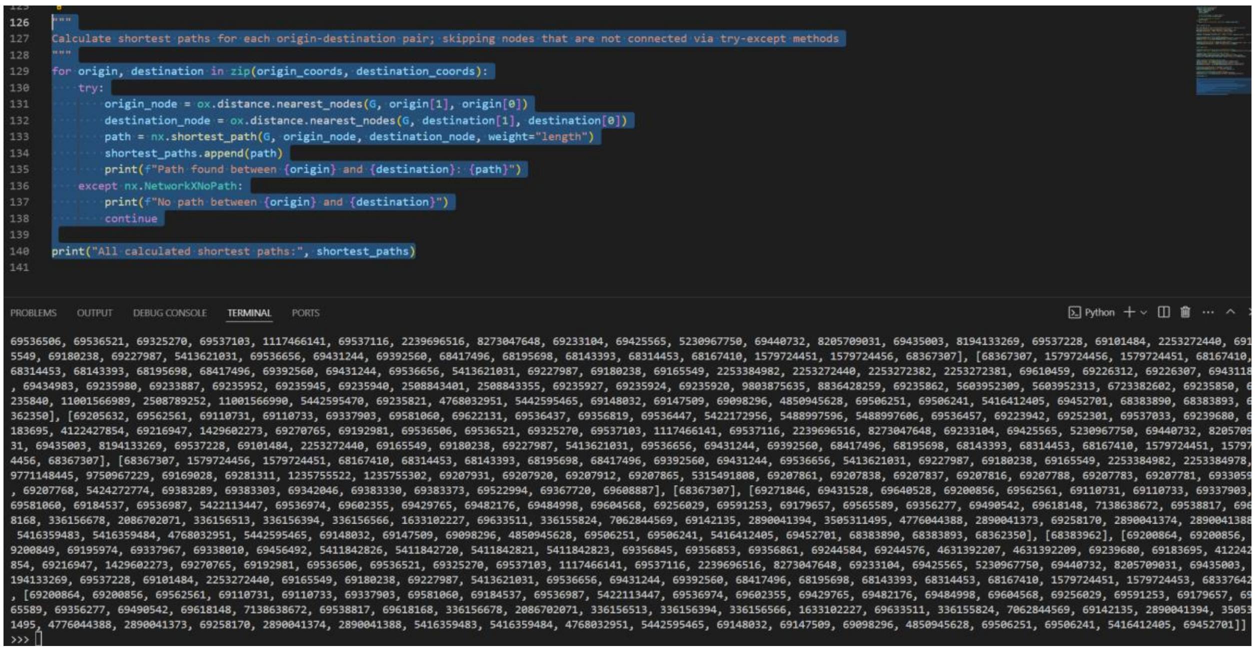


Figure 7: Shortest paths printed out in the terminal

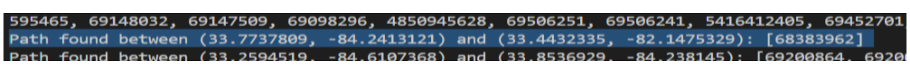


Figure 8: Selected two calculated coordinates.

The selected line of output could be interpreted as such:

- **Coordinates**: (33.737789, -84.241321) as **origin**, and (33.443235, -82.147529) as destination.
- **Path Nodes**: [68383962, ...] - The numbers within the brackets represent the unique identifiers of nodes along the path from the start coordinate to the end coordinate.

This output indicates that the algorithm found the most efficient path (based on road length) between the given origin and destination points within the network. The path is represented as a sequence of nodes that the algorithm traversed to reach the destination from the origin.

```
... # Print a summary
...
No path between (34.3718242, -82.8857632) and (33.7114788, -84.3908033)
No path between (33.7800638, -84.4234369) and (33.7681004, -84.3912905)
>>> print(f"Total paths calculated: {len(shortest_paths_dict)}")
Total paths calculated: 98
```

Figure 9: Sum of calculated paths.

To prevent terminal clutter, I stored the calculated shortest paths in a dictionary, with each origin-destination pair as the key and the corresponding path as the value. This approach allowed efficient storage and retrieval of paths while providing a concise summary. After processing all pairs, a total of **98 paths** were successfully computed, as indicated by the printed summary.

Count node usage

For each shortest path, we counted the frequency of node traversal, identifying nodes that are commonly passed through as critical connectors within the network. Higher frequencies indicate nodes that act as central "bridges" in the network, serving as important links in travel routes. This frequency count will inform our modified centrality calculations, allowing us to assess node importance based on observed travel patterns.

```python
from collections import Counter

# Initialize a counter to track node usage
node_usage = Counter()

# Iterate through each path in shortest_paths_dict and count nodes
for path in shortest_paths_dict.values():
    node_usage.update(path)

# Print out the node usage or process it as needed
print("Node usage counts:", node_usage)

# Sort nodes by frequency of usage in descending order
sorted_node_usage = node_usage.most_common()
# Print the sorted node usage
print("Node usage frequency (sorted):", sorted_node_usage)
```

OBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Pytho

```
, 1929996336: 1, 8194133272: 1, 2253272378: 1, 5559796337: 1, 2253330515: 1, 7138737903: 1, 7138737911: 1, 2253330489: 1, 7138737907: 1, 2255101947: 1, 2253330443: 1, 225333
253330389: 1, 68402232: 1, 7191836135: 1, 68179134: 1, 68337377: 1, 2508825150: 1, 336156027: 1, 69255843: 1, 7062844579: 1, 69109828: 1, 69149159: 1, 1930011434: 1, 6920839
>


, 1929996336: 1, 8194133272: 1, 2253272378: 1, 5559796337: 1, 2253330515: 1, 7138737903: 1, 7138737911: 1, 2253330489: 1, 7138737907: 1, 2255101947: 1, 2253330443: 1, 225333
253330389: 1, 68402232: 1, 7191836135: 1, 68179134: 1, 68337377: 1, 2508825150: 1, 336156027: 1, 69255843: 1, 7062844579: 1, 69109828: 1, 69149159: 1, 1930011434: 1, 6920839
>
, 1929996336: 1, 8194133272: 1, 2253272378: 1, 5559796337: 1, 2253330515: 1, 7138737903: 1, 7138737911: 1, 2253330489: 1, 7138737907: 1, 2255101947: 1, 2253330443: 1, 225333
253330389: 1, 68402232: 1, 7191836135: 1, 68179134: 1, 68337377: 1, 2508825150: 1, 336156027: 1, 69255843: 1, 7062844579: 1, 69109828: 1, 69149159: 1, 1930011434: 1, 6920839
, 1929996336: 1, 8194133272: 1, 2253272378: 1, 5559796337: 1, 2253330515: 1, 7138737903: 1, 7138737911: 1, 2253330489: 1, 7138737907: 1, 2255101947: 1, 2253330443: 1, 225333
, 1929996336: 1, 8194133272: 1, 2253272378: 1, 5559796337: 1, 2253330515: 1, 7138737903: 1, 7138737911: 1, 2253330489: 1, 7138737907: 1, 2255101947: 1, 2253330443: 1, 225333
253330389: 1, 68402232: 1, 7191836135: 1, 68179134: 1, 68337377: 1, 2508825150: 1, 336156027: 1, 69255843: 1, 7062844579: 1, 69109828: 1, 69149159: 1, 1930011434: 1, 6920839
>
```

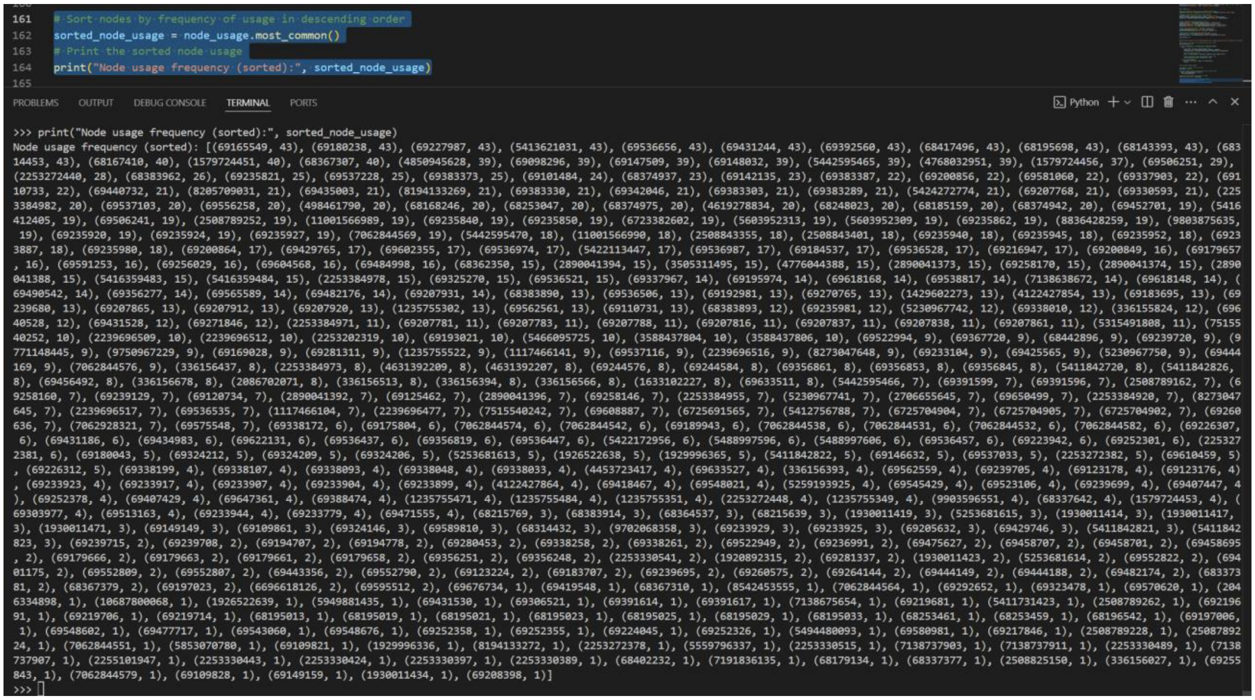


Figure 10 and 11: Calculation and sorting process for node usage frequency

According to Figure 11, we could interpret as follows:

Node Frequency:

- Each entry in the list represents a node ID and its frequency of appearance across all calculated shortest paths.
- (69165549, 43) indicates that node 69165549 was traversed **43 times**, making it the most frequently used node in all paths.
- (69208398, 1) indicates that node 69208398 was only traversed **once**, making it one of the least frequently used nodes.

Significance of Node Frequency:

- **High-frequency nodes** (e.g. 69165549 appearing 43 times) suggest **key points in the network** that are heavily relied upon to connect various origin-destination pairs. These nodes might represent important intersections or hubs in the network.
- **Low-frequency nodes** (e.g. 69208398 appearing only once) are less central in connecting paths. They might represent peripheral or less essential nodes in the network's connectivity.

Comparison between Conventional & Modified Centrality

After completing steps above, assigning the final node betweenness centrality scores based on the number of times each node is used in the shortest paths. The result is a modified betweenness centrality that reflects actual travel patterns, as it considers population density and travel demands.

	Conventional Betweenness	**Modified Betweenness Centrality**
Purpose	Provides a theoretical view of node importance based on network structure alone.	Incorporates actual travel data, prioritizing nodes based on real travel demand.
Data Basis	Provides a theoretical view of node importance based on network structure alone.	Considers population distribution and realistic origin-destination pairs.
Node Importance	Identifies nodes critical to network-wide connectivity.	Highlights nodes essential for daily travel flow.
Practical Application	Useful for understanding structural influence, independent of real-world usage.	Reflects nodes frequently used in actual trips, relevant for daily functionality
Interpretation	High centrality suggests nodes are theoretically central to connectivity.	High centrality indicates nodes are practically essential for travel needs.

Recap of Key Questions

Inputs

- OSM road network: Extracted using OSMnx from OpenStreetMap data within the bounding box, with the coordinates: (33.86, -84.40, 33.82, -84.34) defining the geographical area.
- Graph structure: The road network is converted into a graph of nodes (intersections) and edges (road segments).

Outputs

- In the "Graph Visualization" section, describe that node colors correspond to centrality scores, visually indicating node importance.
- For "Betweenness Centrality Values," clarify how these values directly indicate a node's role in the network, referencing "network's structure and flow" as context for visual analysis.

Required Spatial Operations

- Extract road network: Using OSMnx, extract the road network data within the bounding box.
- Construct a graph: Build a graph from the road network, with nodes representing intersections and edges representing roads.
- Calculate betweenness centrality: Use the shortest path algorithm from Networkx to compute the betweenness centrality for each node in the graph.
- Visualize the results: Color nodes based on their centrality scores to visually analyze their importance.