Task 2.

(a)

Explanation of Cell 1. The code begins by instantiating a *RoadNetwork* object, which loads road data for Berlin through *ox.graph_from_address*. After retrieving a directed graph of the city streets, the code retains only the largest strongly connected component to ensure that most node pairs remain reachable. It then computes a *travel_time* attribute for each edge by dividing the edge's length (in meters) by the corresponding road speed (in km/h, defaulting to 30 if missing), converted into minutes. This processed graph is stored as *road_network.G*.

Once the road network is ready, the *TrafficSimulation* class is used to create a specified number of *Car* objects, each with a random start node and destination node. A *Car* tracks its current location and the path it must follow, but begins with an empty *path*. When *simulate_traffic* is called, the code updates car positions at each discrete time step by invoking *move_cars*. If a car lacks a path, a route is computed using *nx.shortest_path* (with *travel_time* as the weight). The car then attempts to move to the next node along its path. However, if more than a set *jam_threshold* of cars want to move onto the same edge simultaneously, none of them proceed on that edge during this step. Otherwise, each car advances by one node.

After each step, the code computes how many cars occupy each node by constructing a *node_counts* dictionary. These counts may be printed out for diagnostic purposes (though they can be commented out to avoid excessive console output). Additionally, the code can visualize traffic through the *plot_network_with_traffic* method, which indicates how many cars are poised to move onto each edge in the next step by adjusting edge line width and color.

(b)

By loading a real street network from OSMnx data, the code captures the actual geometry and connectivity of roads in Berlin. Each edge is assigned a different travel time based on distance and speed limit, reflecting variations in road length and speed restrictions. Furthermore, the use of *nx.shortest_path* to decide routes simulates the idea of cars choosing the fastest way to their destinations. The concept of a *jam_threshold* also provides a straightforward mechanism for modeling congestion at a high level: when too many cars try to use the same road segment, they stall for that time step, mirroring the buildup of traffic at bottlenecks.

(c)

Although the streets are realistic, the simulation abstracts away many complexities of real-world traffic. Cars jump discretely from node to node each step, with no notion of traveling partway along a road or dealing with multiple lanes. The *jam_threshold* is a single integer, ignoring how road capacity might depend on other factors such as lane count, intersection design, or traffic signals. All cars behave in the same way, and none of them re-route based on evolving traffic

conditions. Other realistic considerations—like speed changes over time, stops at traffic lights, or random driver behavior—are similarly not included. As a result, while the code is suitable for demonstrating basic congestion ideas on a city map, it does not capture the full complexity of urban traffic flow.

(d)

(How the Original Code Determines Congestion)

In **Cell 1**, the simulation treats a road as "congested" if more than a fixed number of cars, by default five, attempt to enter the same edge in a single step. Specifically, the code checks how many cars want to move from *current_node* to *next_node*, and if that count exceeds *jam_threshold=5*, none of them proceeds. This simple mechanism helps illustrate the idea of traffic piling up on busy streets, but it relies on an arbitrary cutoff that ignores important real-world factors such as the actual length of the road, the number of lanes, or variable driver behavior. In practice, a short one-lane street might "jam" with fewer than five cars, and a long multi-lane highway could easily handle more than five cars without significant slowdown.

(Improvements to the Threshold-Based Model)

Cell 2. To improve the threshold-based approach, we can maintain the same overall logic but allow for smaller slowdowns whenever multiple cars enter an edge, without necessarily blocking them altogether. In the first enhanced code cell, each car accumulates additional *travel_time_spent* based on how many vehicles share its target edge. This makes congestion more gradual: a road carrying two or three cars at once moves more slowly than an empty road but does not instantly "jam" until a higher cutoff is reached. While this revised threshold model still depends on an integer limit (e.g., five cars), it at least captures partial congestion by scaling travel time upward, making the code more realistic than an all-or-nothing jam approach.

(Session 9-Inspired Continuous Congestion)

Cell 3. A more sophisticated idea, drawn from CS166 Session 9's traffic modeling, is to abandon the fixed threshold altogether and instead treat congestion as a continuous phenomenon. In this second enhanced code cell, each edge's base travel time increases in proportion to a computed "density," reflecting how many cars share that segment relative to its capacity. This density-based slowdown is further combined with a random probability of reduced speed (*p_slow*), capturing the stochastic elements often seen in real traffic flows. By injecting these ideas into the *move_cars_one_step_improved* method, cars can coexist on a single road without abruptly blocking each other, but they face higher travel times when crowding intensifies. This approach removes the arbitrary cutoff and more closely mirrors how real congestion evolves, wherein speeds deteriorate gradually rather than halting at a strict numerical threshold.

Task 3

Figure 1. The original plot from ChatGPT has several faults. Every edge is drawn on the same black canvas, but only those that happen to carry traffic are colored. Edges with zero cars fade into the background, so the street layout is incomplete. The red scale is linear yet unlabeled, there is no colour-bar, legend, or numeric annotation, so the user cannot tell whether a dark segment represents three cars or thirty. Line thickness grows one pixel per extra car, but on high-resolution screens that difference is almost invisible. Nodes and intersections are rendered as plain white dots with no indication of how busy they are, and the plot shows a single static snapshot, giving no clue about flow direction or change over time. In short, the picture looks attractive but does not answer the key question: *which* roads are truly congested and *how* congested are they?

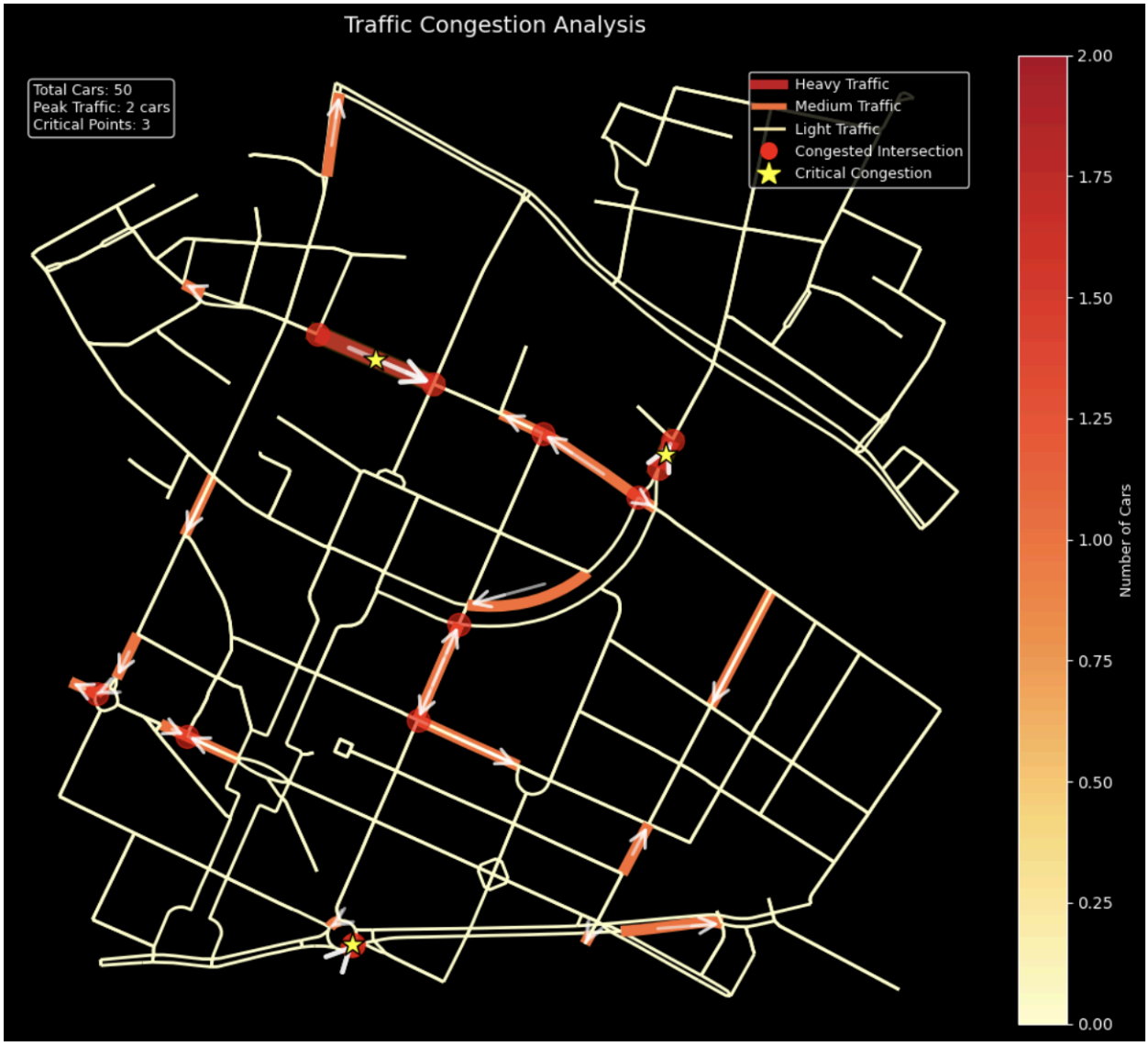


Figure 2. The enhanced visualization (Cell 5) tackles those gaps one by one and then adds several layers of quantitative detail. All roads stay on the map, and unused links are drawn in faint grey, so the viewer never loses the underlying street grid. Active links are colored with a calibrated yellow-to-deep-red gradient, and their widths grow according to a logarithmic rule, so even a two-or-three-car difference produces a noticeable visual jump without making the thickest roads overwhelming. Any edge whose load reaches 80 % or more of the step's peak is over-painted with a subtle golden glow and stamped with a bright yellow star, instantly flagging it as a critical hotspot. Intersections where the sum of incoming and outgoing cars exceeds 70 % of the peak are marked by semi-transparent red circles, revealing multi-edge choke points. White flow arrows, with both count and arrow-head size proportional to the traffic level, trace the direction and relative strength of movement along every busy street, turning the static image into a snapshot of network flow. A legend decodes heavy, medium, and light colors, explains the star and circle symbols and a color bar on the right converts hue into an exact car count. In the

upper-left corner, a statistics panel reports the total cars in the run, the highest single-edge load, and the number of critical edges so the user can gauge overall stress at a glance. Taken together, these elements transform the picture from a simple heat map into a diagnostic dashboard that shows exactly where, and how severely, congestion is building throughout the network.

Figure 3. While Figure 2 showed a strong static snapshot of congestion, I wanted to better illustrate how congestion develops over time, so I created an animation with 30 frames (see Cell 6). Each frame shows live traffic (left) and cumulative congestion (right) as cars move through the network. The left side updates dynamically with car positions, traffic levels (color-coded), and intersections marked red when overloaded. The right side builds up a congestion count – roads turn from blue to red based on how often they were congested. In this case, a road is marked "congested" if the number of cars trying to enter it exceeds 50% of the peak traffic in that step. This dual view makes it easier to see both short-term traffic jams and long-term structural bottlenecks.

Task 4.

(a)

I ran multiple simulation batches (50 runs × 30 steps) using 100 and 150 cars on a real Berlin street network, applying a congestion rule where a road segment is marked as congested if it carries at least 50% of the peak load observed in that step. This 50% rule is a design choice that simplifies congestion detection. For each simulation, I tracked three core metrics: (1) how frequently each road became congested over all runs (i.e., congestion frequency), (2) the number of cars still in motion at each time step, and (3) the number of congested roads per step. Congestion frequency served as a key output: it quantifies which streets are consistently bottlenecks and provides a normalized, step-aggregated view of stress on the network. This metric later becomes the basis for comparison with theoretical network predictors. I visualized the empirical data using four plots: a histogram of congestion frequencies across roads, a time-series of average cars in motion, a bar chart capturing congestion volatility per step, and a network heatmap showing the top 10 most frequently congested roads. The time-series of cars in motion also helps confirm that the system typically reaches a steady state before the simulation ends. I saved the results to CSVs for transparency and reproducibility. To understand how the system responds to increased traffic, I then compared the 100- and 150-car cases by plotting congestion shifts road-by-road and summarizing the overall change using basic statistics.

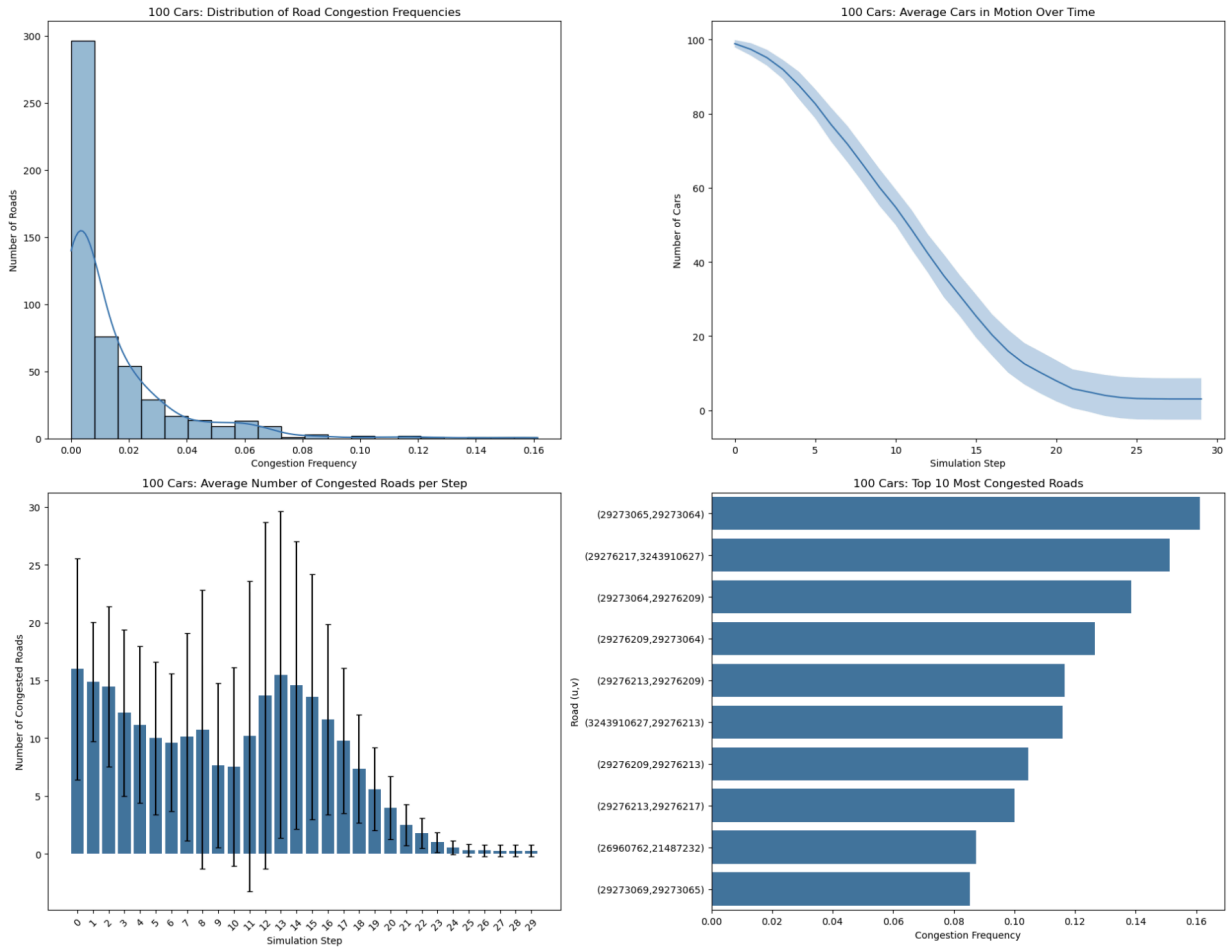


Figure 4: This panel contains four plots summarizing congestion patterns across 50 runs of a 30-step simulation with 100 cars. The top-left histogram shows that most roads have very low congestion frequency, though a few exhibit values above 10%, indicating consistent overuse. The top-right time series plot illustrates how the number of cars in motion drops steadily over time, suggesting that most cars reach their destinations before the simulation ends. The bottom-left bar chart captures the average number of congested roads per simulation step, with error bars indicating volatility; congestion is highest in the early steps and tapers off. The bottom-right bar chart identifies the top 10 most congested roads based on their empirical congestion frequency. I repeated this exact analysis for the 150-car scenario to study the effect of increased demand.

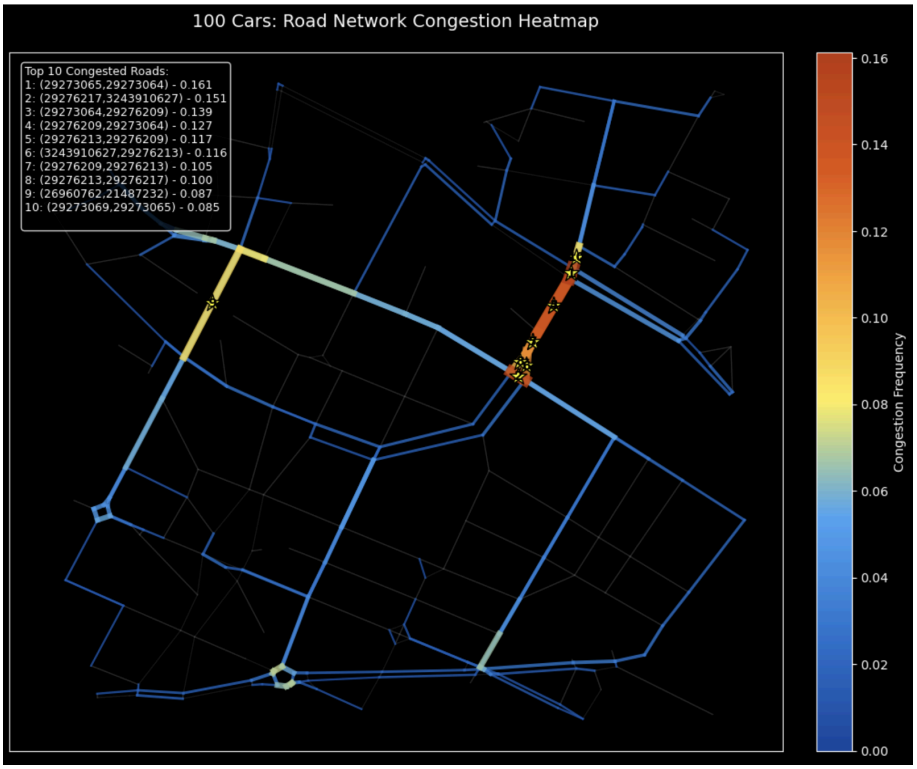


Figure 5: This map visualizes congestion frequency across all roads in the Berlin street network used in the 100-car simulation. Roads are color-coded from blue (low congestion) to red (high congestion), and the 10 most congested roads are labeled directly on the map with yellow star markers and rank annotations. This spatial representation complements the prior plots by revealing where geographically congestion tends to accumulate. I also created a corresponding heatmap for the 150-car experiment to visualize how congestion hotspots shift under higher traffic load.

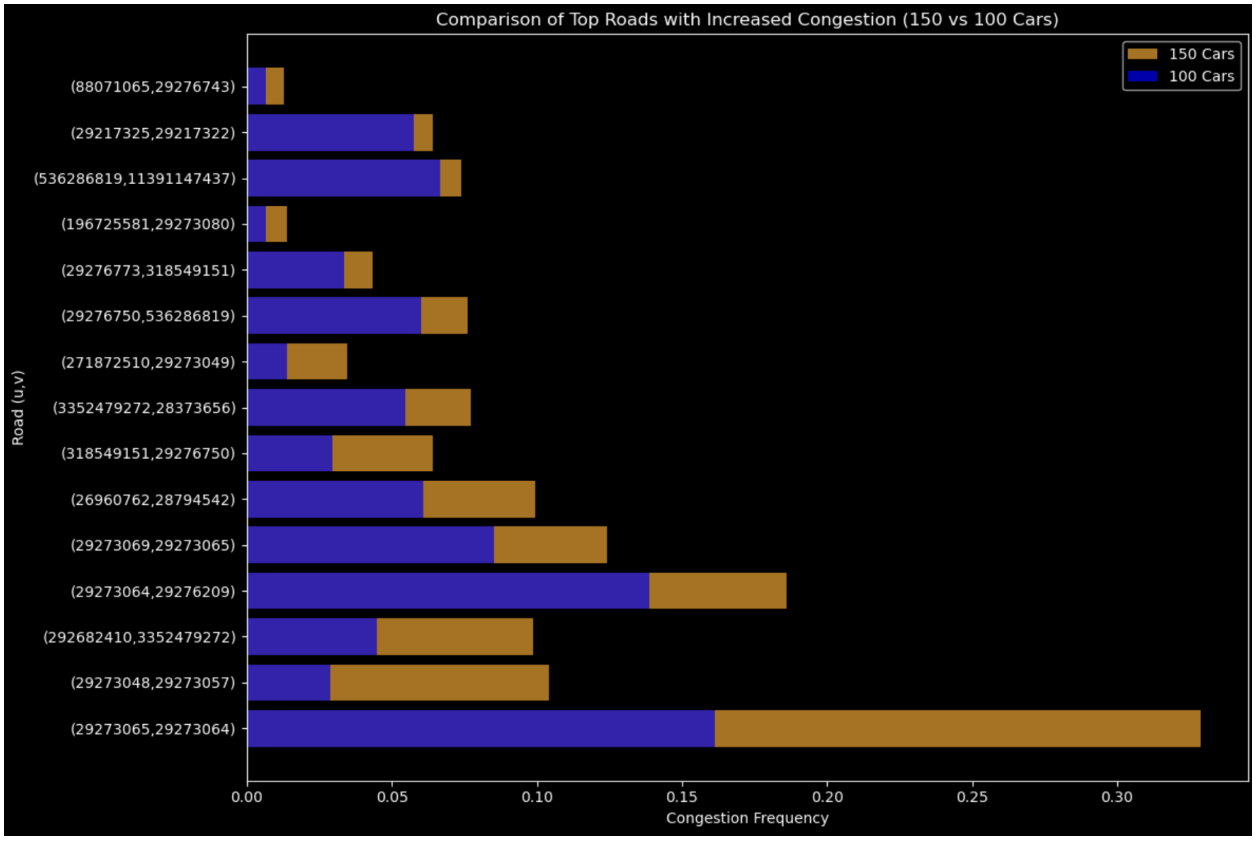


Figure 6: This horizontal bar chart compares congestion frequencies for the top roads whose congestion increased most when moving from 100 to 150 cars. Blue bars show the original congestion levels, and orange bars represent the updated frequencies under heavier demand. Several roads show a significant rise, while others experience relatively small changes, revealing that congestion is concentrated rather than evenly distributed. This visualization helps quantify which parts of the network are most sensitive to load increases.

Summary of Changes in Congestion Frequency:

Metric	Value
Average change	-0.003336
Maximum increase	0.167333
Maximum decrease	-0.051333
Roads with >10% increase	1.000000
Roads with any increase	97.000000
Roads with any decrease	300.000000

Figure 7: This table reports the numerical summary of congestion frequency changes between the 100-car and 150-car simulations. The average change is slightly negative, but this is due to many minor decreases across roads with very low initial congestion. The maximum increase is 0.167, and one road experienced a rise of more than 10%. In total, 97 roads showed any increase, while 300 saw a decrease. This summary supports the conclusion that congestion shifts are not uniform but highly localized. It is useful for documenting the statistical impact of scaling demand.

(b)

To investigate which network metric best predicts road congestion in our simulation, I calculated several measures – betweenness centrality, average node degree, and average node closeness – for each edge in the Berlin street network. Building on the empirical results saved in CSV files (from the 100- and 150-car runs), I merged these centrality metrics with each edge's observed congestion frequency. I then computed correlation coefficients (Spearman's ρ and Pearson's r) and fit simple regression models to see which metric aligned best with empirical congestion. This approach allowed me to systematically compare the relative performance of each theoretical predictor under the same data. The betweenness calculation factored in travel_time weights so that routes favored short or fast edges. Once I identified betweenness as the strongest single predictor, I dove deeper into its explanatory power by looking at R^2 values, residual plots, and regression-based composite scores that might further refine predictions.

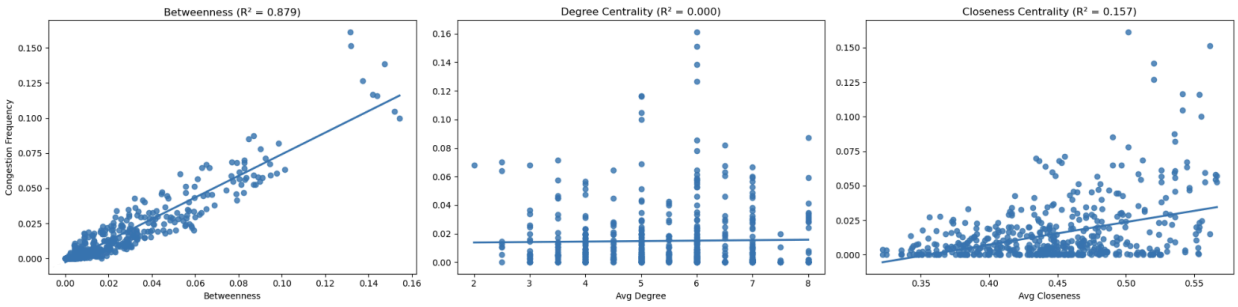


Figure 8: This correlation comparison (see the first image) shows three scatter plots: one each for betweenness, average node degree, and average node closeness on the x-axis, versus empirical congestion frequency on the y-axis. A fitted line and an R^2 value quantify how well each metric explains road congestion. The results reveal that betweenness centrality outperforms both degree and closeness by a wide margin, with $R^2 \approx 0.879$, implying that roads lying on many shortest paths are the ones most likely to experience steady congestion in the simulation. By contrast, average degree and closeness show considerably lower correlations, suggesting that simple connectivity or proximity alone does not capture our traffic conditions as accurately.

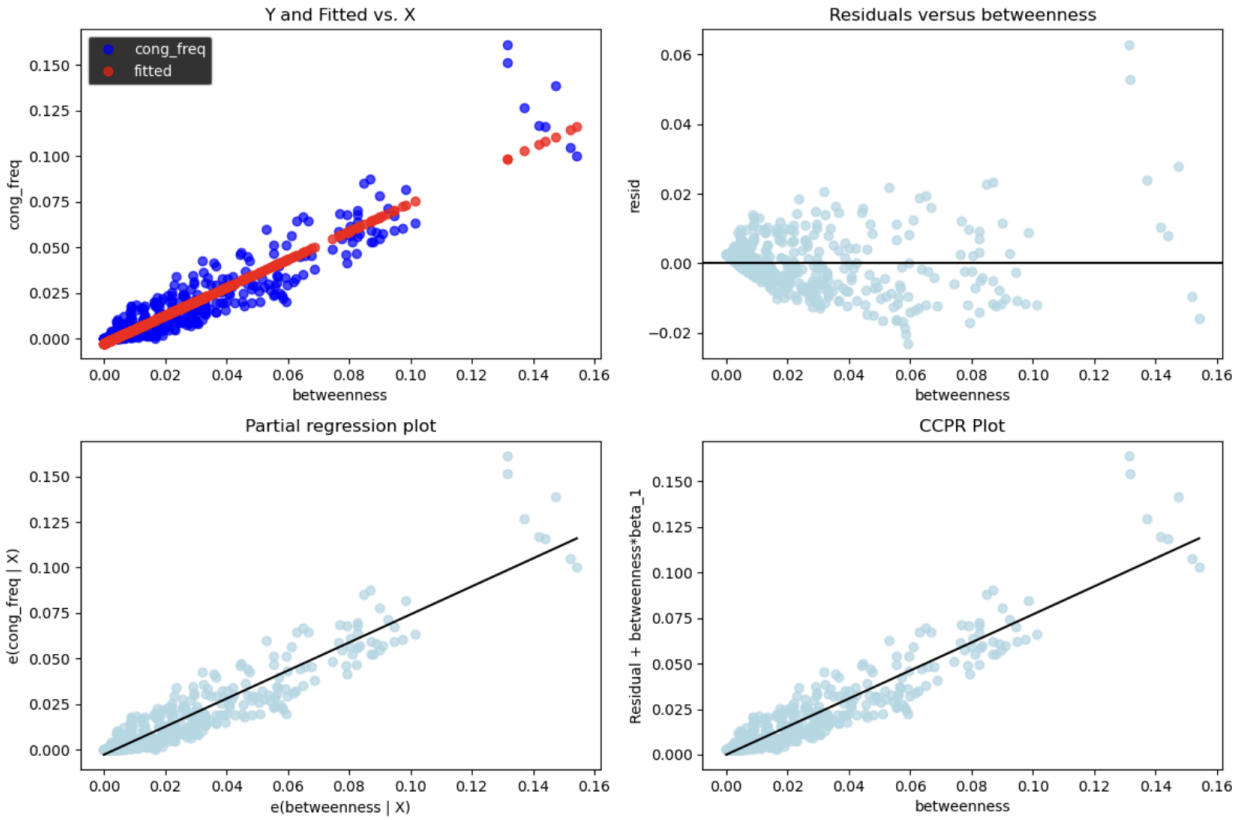


Figure 9: Having confirmed that betweenness offers the strongest theoretical match, I examined it in more detail using two additional plots. The second image provides a deeper dive: it shows a regression plot of betweenness against empirical congestion frequency, with R^2 displayed in the title, and a residuals analysis that checks how well a linear model fits the data. We see that the main regression line fits very closely (indicating high R^2), and the residuals appear relatively unpatterned, suggesting no strong nonlinearities or overlooked factors. In other words, roads that rank highly on betweenness also consistently appear in the higher congestion-frequency range, aligning closely with the idea that more cars pass along these crucial corridors.

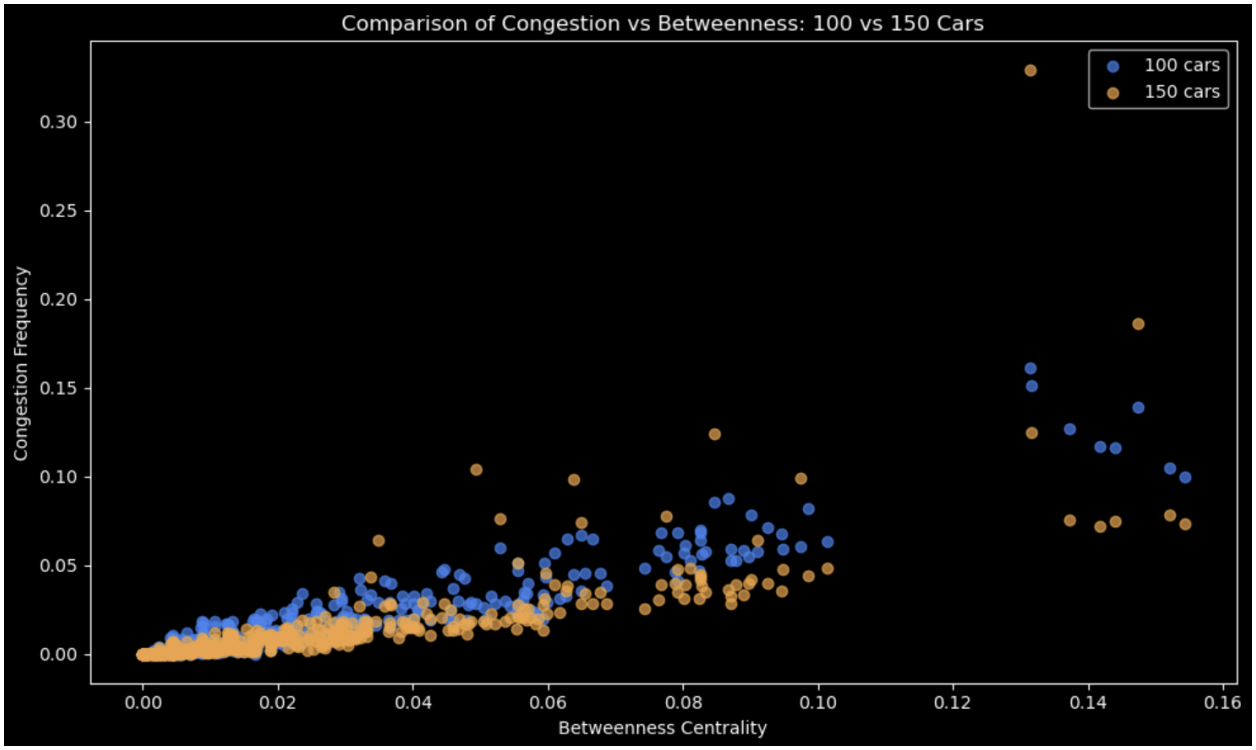


Figure 10: Sensitivity check. To further validate the betweenness metric under different demand levels, I plotted a side-by-side comparison of 100-car and 150-car data overlaid on the same axis (third image). Each dot represents a road's betweenness value (x-axis) versus its congestion frequency (y-axis), color-coded by scenario (blue for 100 cars, orange for 150 cars). Most data points shift upward slightly from the 100-car cluster to the 150-car cluster, indicating that adding more cars intensifies congestion – especially on high-betweenness roads. This visual underscores the stability of betweenness as a predictor even under a heavier traffic load, since both scenarios show a similar trend line with similarly strong correlations.

(c)

To evaluate the robustness of my findings, I replicated the full simulation and theoretical analysis pipeline for a new location in Buenos Aires (Avenida Santa Fe 730), building directly on my earlier workflow from Berlin (See Cell 9). I constructed a real street network around this address and reused my prior logic for measuring congestion frequency: I ran 50 simulations of 30 steps each with 100 cars, and marked roads as congested if they carried at least 50% of the peak load at any given step. I tracked congestion frequency across all roads, exported the data, and then computed betweenness centrality as the primary theoretical metric of interest. The resulting datasets were merged and analyzed to assess whether betweenness could still explain congestion patterns in a new, more grid-like urban context. I also examined residual patterns to check the goodness of fit and plotted the top 10 most congested roads to compare their theoretical scores to

the rest of the network. This allowed me to test the generalizability of my original conclusion – that betweenness centrality is a strong predictor of congestion hotspots.

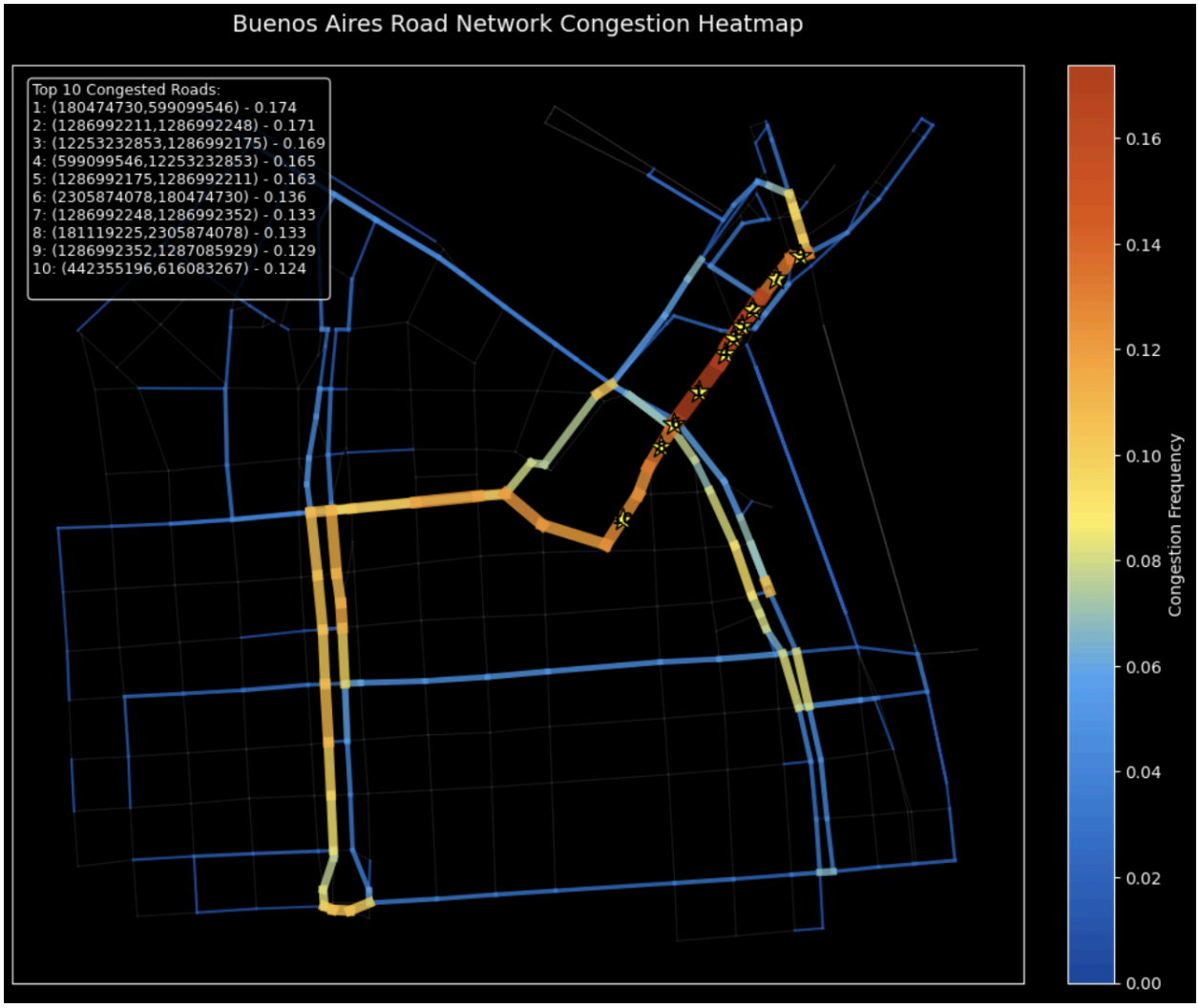


Figure 11: This shows the road network of Buenos Aires centered around Avenida Santa Fe 730, color-coded by congestion frequency. As before, roads that were congested in more steps across the simulations are shown in warmer colors (yellow to red), while those with lower congestion frequency remain in cooler blue tones. The top 10 most frequently congested roads are marked with yellow stars and labeled directly on the map. The heatmap clearly reveals that congestion is concentrated along a north-south corridor that runs through a few highly central edges. This spatial pattern confirms that even in a different urban grid, congestion tends to cluster along a predictable set of high-traffic roads.

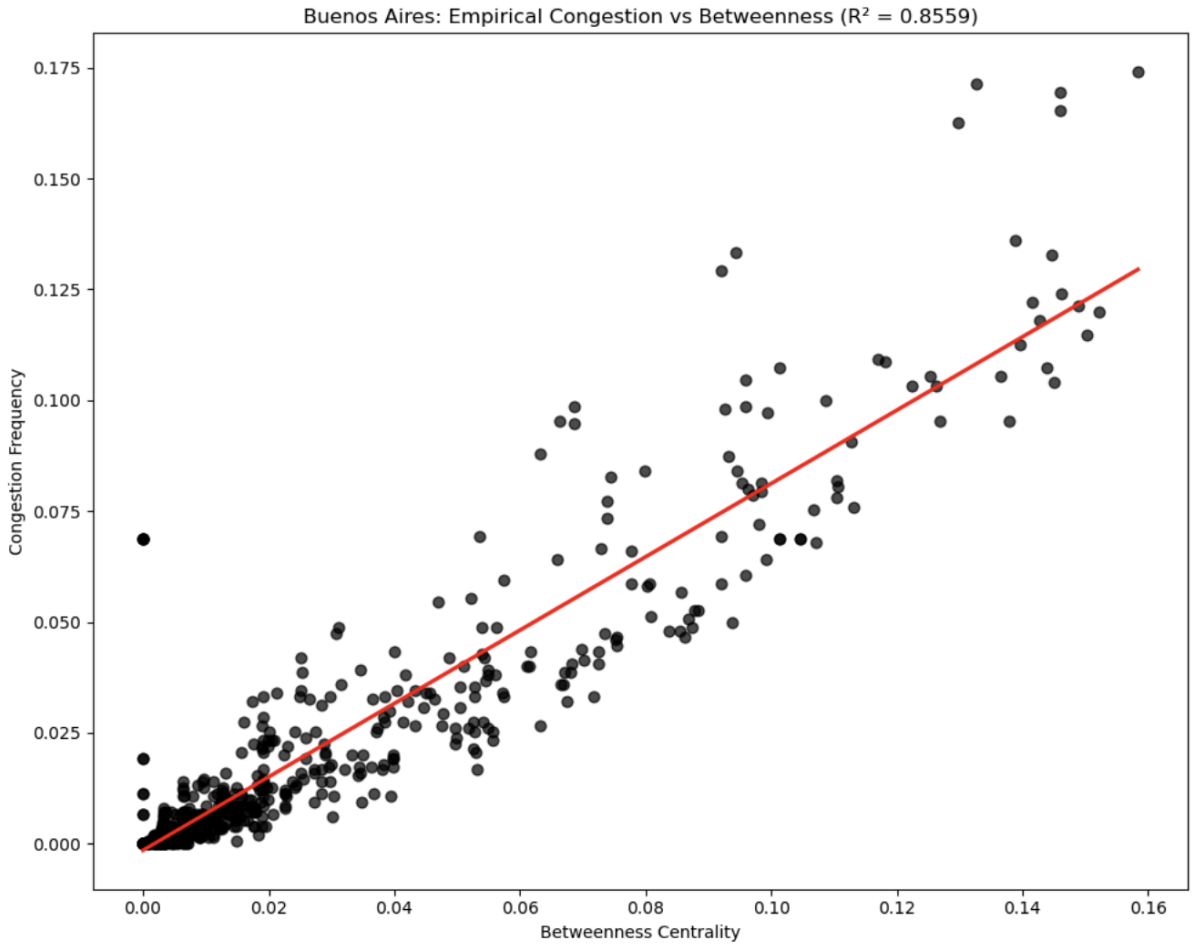


Figure 12: This plots betweenness centrality against congestion frequency for all road segments in Buenos Aires and shows a clear, tight linear relationship with an R^2 of 0.856. This confirms that betweenness remains a very strong predictor of congestion, even in a completely different network structure. The intuition here is consistent with what we saw in Berlin: roads that are more "central" in terms of shortest-path flow tend to carry higher traffic volume and are more likely to become bottlenecks during the simulation. This supports the broader claim that betweenness centrality generalizes well as a network-based congestion predictor across urban contexts.

Top 10 Most Congested Roads in Buenos Aires:

	u	v	Congestion Freq	Betweenness	Betweenness Rank
39	180474730	599099546	0.1740	0.1584	1.0000
401	1286992211	1286992248	0.1713	0.1326	7.0000
578	12253232853	1286992175	0.1693	0.1459	3.5000
359	599099546	12253232853	0.1653	0.1459	3.5000
400	1286992175	1286992211	0.1627	0.1297	8.0000
458	2305874078	180474730	0.1360	0.1388	6.0000
402	1286992248	1286992352	0.1333	0.0943	9.0000
47	181119225	2305874078	0.1327	0.1446	5.0000
412	1286992352	1287085929	0.1293	0.0920	10.0000
357	442355196	616083267	0.1240	0.1462	2.0000

Figure 13: Table showing the 10 most congested roads in Buenos Aires along with their congestion frequencies, betweenness centrality values, and ranks. We observe that nearly all top congestion hotspots also appear in the top tier of betweenness rankings, with the most congested road *(180474730, 599099546)* also having the highest betweenness. This alignment further validates that betweenness is not only a good overall predictor but also accurately identifies specific high-stress roads.

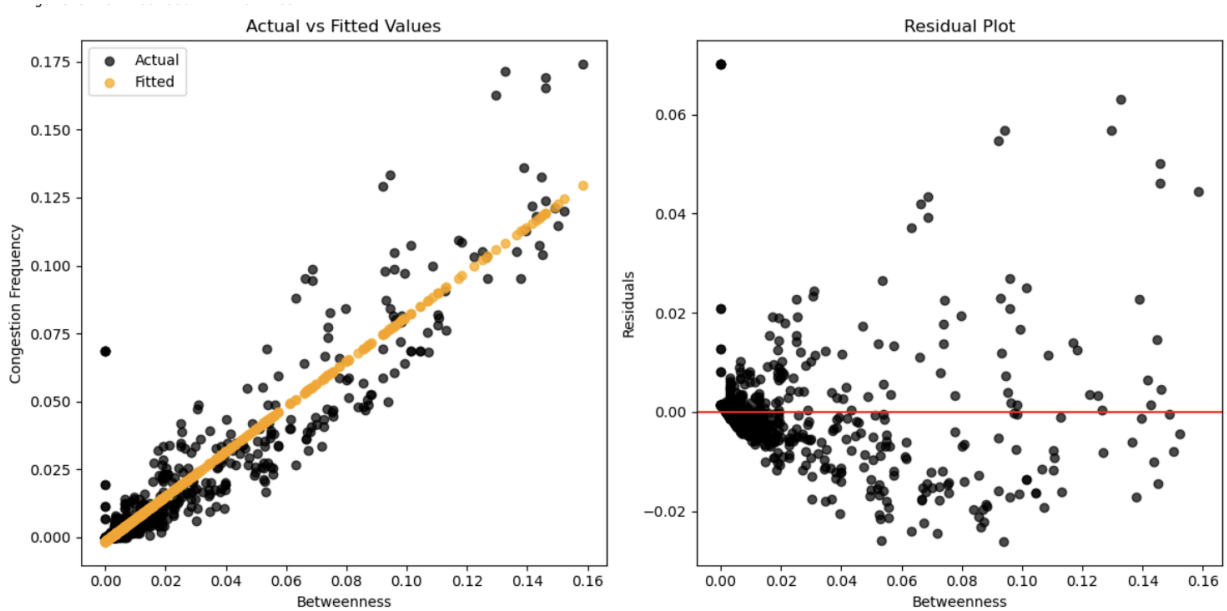


Figure 14: This deepens our insight by visualizing the linear regression fit. The left panel shows the actual vs. fitted congestion frequencies, with a close alignment along the diagonal line, indicating high predictive accuracy of the betweenness-based model. The right panel presents residuals plotted against betweenness values. Most residuals are tightly clustered around zero, with no strong non-linear trends, suggesting that the linear model is appropriate and well-specified. Together, these plots confirm that betweenness captures the underlying structure of traffic dynamics in Buenos Aires as effectively as it did in Berlin.

In both study areas I quantify "how pivotal is this road segment to shortest-path flow?" with weighted edge betweenness centrality. Formally, for an edge e let σ_{st} be the number of shortest-travel-time paths between every ordered origin-destination pair (s, t) in the network and $\sigma_{st}(e)$ the subset of those paths that traverse e. The centrality is

$$B_e = \frac{1}{(n-1)(n-2)} \sum_{s \neq t;\, s,t \xi V} \frac{\sigma_{st}(e)}{\sigma_{st}},$$

where the denominator normalises by the total number of pairs (n is the node count). Because I supply travel-time as the edge weight, each σ_{st} is computed on the fastest, not merely the geometrically shortest, route. Intuitively, B_e estimates the fraction of ideal drivers who would have to use edge e if everyone pursued the quickest path in an uncongested network. My simulation does exactly that at every step, so, before feedback effects set in, the expected load on each edge is proportional to B_e.

This mechanistic link explains why betweenness predicts congestion in both Berlin and Buenos Aires despite their contrasting geometries. Berlin's irregular, radial street pattern funnels many origin-destination pairs through a handful of curving arterials; those edges accumulate very high B_e values and become chronic bottlenecks in the empirical runs ($R^2 \approx 0.879$). The sampled district of Buenos Aires, by contrast, is a near-orthogonal grid that offers multiple parallel avenues. Even there, however, certain diagonals and the main north-south corridor appear in a large share of the fastest routes, giving them elevated betweenness and, consequently, the highest simulated congestion frequencies ($R^2 \approx 0.856$). In other words, edge betweenness isolates the structural "throat" of the network – the segments that shortest paths must squeeze through – so its explanatory power survives changes in global layout.

Mathematically this robustness follows from the additivity of the sum above: whether paths bend organically (Berlin) or proceed in Manhattan-style stair steps (Buenos Aires), every extra pair (s, t) that is forced to traverse edge e increments $\sigma_{st}(e)$ by exactly one. Grid regularity merely spreads those increments more evenly, lowering the maximum B_e but not the rank order. Because

my congestion metric – frequency of exceeding 50 % of the step peak load – is itself a count‑over‑time measure, it scales linearly with the same path‑count logic. The close empirical-theoretical agreement in two dissimilar cities therefore provides strong evidence that weighted edge betweenness captures a fundamental, topology‑driven propensity for congestion.

AI Statement: I used ChatGPT primarily to support the visualization components of my code, especially for generating network plots and improving data presentation. While I wrote and refined a significant portion of the code myself, including most of the debugging, as shown in my outputs, I occasionally relied on GPT when I was completely stuck or needed help understanding a specific implementation detail.

Appendix - CS166 LBA

April 13, 2025

[1]:
```
# Core data and math
import numpy as np
import pandas as pd
import random
from collections import defaultdict

# Graph and spatial
import osmnx as ox
import networkx as nx
import geopandas as gpd

# Plotting
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib.cm import ScalarMappable
from matplotlib.lines import Line2D
from matplotlib.collections import LineCollection

# Advanced plotting
import matplotlib.patheffects as pe
```

```
Matplotlib is building the font cache; this may take a moment.
```

[105]:
```
from tqdm import tqdm
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
from IPython.display import display

from scipy.stats import spearmanr, pearsonr
import statsmodels.api as sm
from tabulate import tabulate

import time
import seaborn as sns
```

[92]:
```
!pip install tabulate
```

```
Collecting tabulate
  Downloading tabulate-0.9.0-py3-none-any.whl.metadata (34 kB)
```

```
Downloading tabulate-0.9.0-py3-none-any.whl (35 kB)
Installing collected packages: tabulate
Successfully installed tabulate-0.9.0
```

[93]:
```
from tabulate import tabulate
```

[2]:
```
print("Environment versions:")
import sys
print("Python    :", sys.version)
print("numpy     :", np.__version__)
print("pandas    :", pd.__version__)
print("osmnx     :", ox.__version__)
print("geopandas :", gpd.__version__)
print("networkx  :", nx.__version__)
import matplotlib
print("matplotlib:", matplotlib.__version__)
```

```
Environment versions:
Python    : 3.10.17 | packaged by conda-forge | (main, Apr 10 2025, 22:15:49)
[Clang 18.1.8 ]
numpy     : 2.2.4
pandas    : 2.2.3
osmnx     : 2.0.2
geopandas : 1.0.1
networkx  : 3.4.2
matplotlib: 3.10.1
```

[87]:
```
%matplotlib inline
```

1 Task 1

Cell 1

[3]:
```
class RoadNetwork:
    """
    Manages the loading and preparation of the Berlin road network.
    """

    def __init__(self, address="Adalbertstraße 58, Berlin, Germany", dist=1000):
        """
        Parameters
        ----------
        address : str
            The address around which to download the road network.
        dist : int
            The distance in meters from the address for bounding the graph.
        """
        self.address = address
```

```
        self.dist = dist
        self.G = None  # Will store the network graph

        self.load_network()
        self.make_strongly_connected_subgraph()
        self.add_travel_time_attribute()

    def load_network(self):
        """
        Load the raw road network (a MultiDiGraph) using OSMnx.
        """
        print(f"Loading road network for '{self.address}' within dist={self.
↪dist} meters...")
        self.G = ox.graph_from_address(self.address, dist=self.dist,␣
↪network_type='drive')

    def make_strongly_connected_subgraph(self):
        """
        Restrict to the largest strongly connected component so random start/
↪destination
        pairs have a very high chance of being reachable.
        """
        print("Extracting largest strongly connected component...")
        largest_scc = max(nx.strongly_connected_components(self.G), key=len)
        self.G = self.G.subgraph(largest_scc).copy()

    def add_travel_time_attribute(self):
        """
        Add a 'travel_time' attribute to each edge. The formula is:
            travel_time = length / (speed_km_per_h * 1000/60)
        which yields minutes of travel along the edge.
        For missing maxspeed, we assume 30 km/h.
        """
        print("Attaching 'travel_time' attributes to edges...")
        for u, v, k, data in self.G.edges(data=True, keys=True):
            speed = data.get("maxspeed", 30)
            if isinstance(speed, list):
                speed = speed[0]  # if multiple speeds are given, use the first
            try:
                speed_val = float(speed)
            except (ValueError, TypeError):
                speed_val = 30.0  # fallback if speed isn't parseable

            # length in meters, speed in km/h => travel_time in minutes
            data["travel_time"] = data["length"] / (speed_val * 1000 / 60.0)

    def plot_network(self, figsize=(10,10)):
```

```
        """
        Plot the road network using OSMnx.

        Parameters
        ----------
        figsize : tuple
            Dimensions of the figure to plot.
        """
        fig, ax = ox.plot_graph(self.G, figsize=figsize)
        return fig, ax


class Car:
    """
    Represents a single car in the simulation.
    """

    def __init__(self, start_node, destination_node):
        """
        Parameters
        ----------
        start_node : int
            The OSMnx node ID where this car starts.
        destination_node : int
            The OSMnx node ID where this car aims to end up.
        """
        self.current_location = start_node
        self.destination = destination_node
        self.path = []  # Will be a list of nodes in the route (excluding␣
 ↪current_location)


class TrafficSimulation:
    """
    Manages the creation and movement of cars on a RoadNetwork, and provides
    tools for visualizing traffic flow.
    """

    def __init__(self, road_network: RoadNetwork, num_cars=100,␣
 ↪jam_threshold=5):
        """
        Parameters
        ----------
        road_network : RoadNetwork
            The road network to use for the simulation.
        num_cars : int
            Number of cars to generate.
```

```
        jam_threshold : int
            The number of cars that triggers a 'jam' on an edge in a single␣
↪timestep.
        """
        self.road_network = road_network
        self.G = road_network.G  # Shortcut to the underlying graph
        self.num_cars = num_cars
        self.jam_threshold = jam_threshold

        self.cars = []
        self.create_cars()

    def create_cars(self):
        """
        Randomly create the specified number of Car objects, each with
        a random start and destination in the graph's node set.
        """
        nodes = list(self.G.nodes())
        self.cars = []
        for _ in range(self.num_cars):
            start = random.choice(nodes)
            destination = random.choice(nodes)
            car = Car(start, destination)
            self.cars.append(car)

    def move_cars(self):
        """
        For each car, move it one step along its path. If it doesn't have
        a path yet, compute a shortest path by 'travel_time'.
        Then check how many cars want the same (current_node -> next_node) edge;
        if that exceeds jam_threshold, the car doesn't move.
        """
        for car in self.cars:
            # If car's path is empty, we compute it (unless at destination).
            if not car.path:
                if car.current_location != car.destination:
                    try:
                        car.path = nx.shortest_path(
                            self.G,
                            car.current_location,
                            car.destination,
                            weight='travel_time'
                        )[1:]  # Exclude the first node (car.current_location)
                    except nx.NetworkXNoPath:
                        # No path found. Car can't move.
                        car.path = []
```

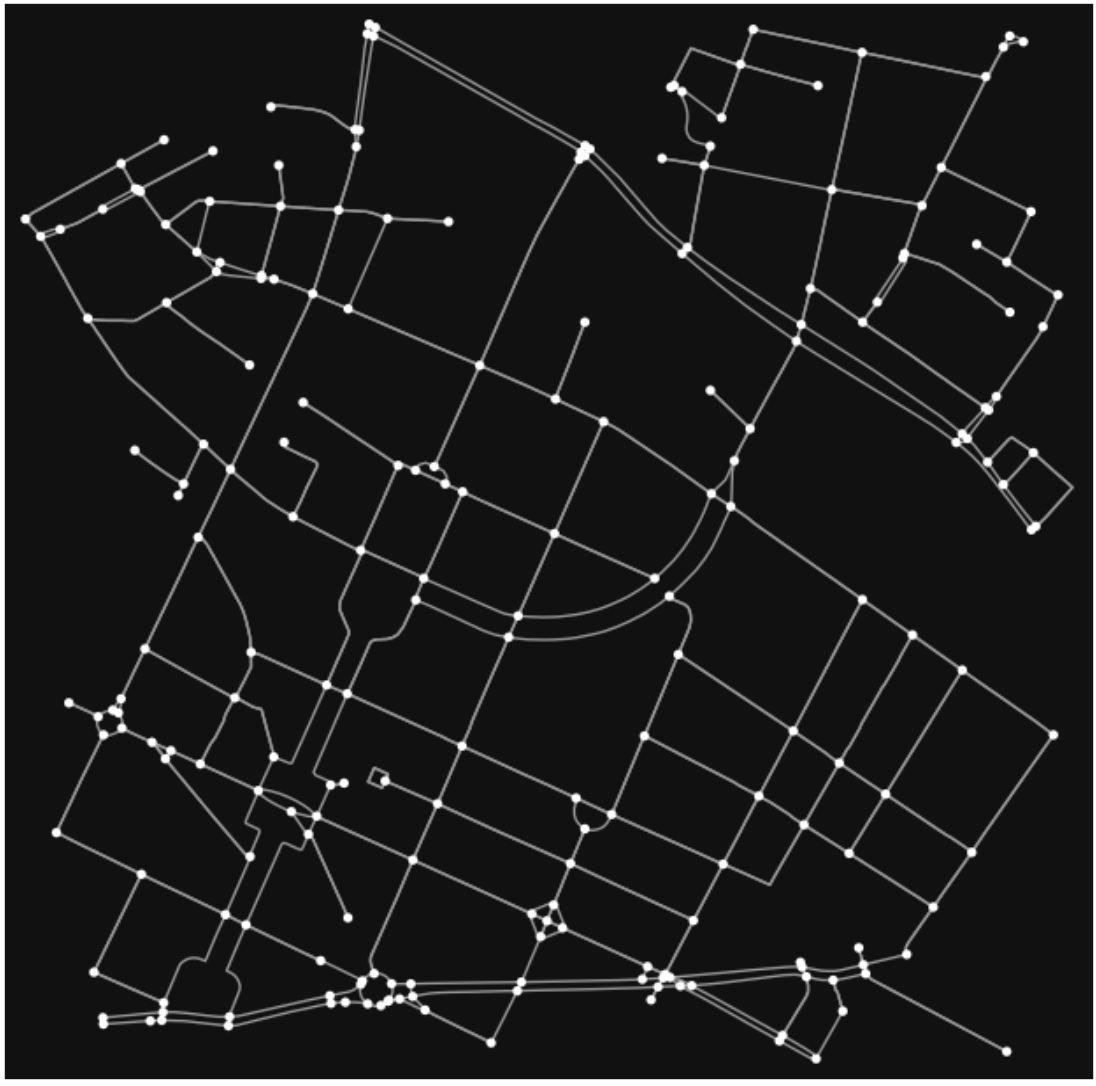
```
            # If car has reached destination or has no path, skip
            if car.current_location == car.destination or not car.path:
                continue

            # Check how many cars want the same edge (car.current_location ->␣
↪next_node)
            next_node = car.path[0]
            cars_on_edge = sum(
                1 for c in self.cars
                if c.current_location == car.current_location
                and c.path
                and len(c.path) > 0
                and c.path[0] == next_node
            )

            # If more than jam_threshold, nobody on that edge moves
            if cars_on_edge > self.jam_threshold:
                # The car stays put, effectively jammed
                pass
            else:
                # Move forward
                car.current_location = next_node
                car.path.pop(0)

    def simulate_traffic(self, num_steps=10):
        """
        Run the simulation for a given number of steps. Each step moves all␣
↪cars once.
        Print how many cars are at each node after each step.

        Parameters
        ----------
        num_steps : int
            How many discrete simulation steps to run.
        """
        nodes = list(self.G.nodes())

        for step in range(num_steps):
            self.move_cars()

            # For demonstration, record how many cars are at each node
            node_counts = {node: 0 for node in nodes}
            for car in self.cars:
                node_counts[car.current_location] += 1

            print(f"Step {step + 1}:")
```

```
            # Optionally, to avoid excessive output, only print or log details␣
↪if needed
            #  for node, count in node_counts.items():
            #      if count > 0:
            #          print(f" Node {node}: {count} cars")

    def edge_car_counts(self):
        """
        Count how many cars *plan* to move from one node to the next in this␣
↪step,
        i.e., how many have (current_location->path[0]) set.

        Returns
        -------
        dict
            A dictionary keyed by (u, v) representing how many cars want that␣
↪edge.
        """
        counts = {}
        for u, v in self.G.edges():
            counts[(u, v)] = 0

        for car in self.cars:
            if car.path:
                next_node = car.path[0]
                if (car.current_location, next_node) in counts:
                    counts[(car.current_location, next_node)] += 1
        return counts

    def reset_simulation(self):
        """
        Reset the simulation by creating new random cars.
        """
        # Store the number of cars
        num_cars = len(self.cars)
        # Create new random cars
        self.cars = []
        self.create_cars()
        return self

    def plot_network_with_traffic(self, figsize=(10,10)):
        """
        Plot the road network, coloring and thickening edges based on how many
        cars plan to move along each edge.
        """
        counts = self.edge_car_counts()
        if not counts:
```

```
            print("No edge usage to display.")
            return

        max_count = max(counts.values()) if counts.values() else 1
        if max_count == 0:
            max_count = 1

        edge_colors = []
        edge_widths = []
        edge_list = list(self.G.edges())

        for (u, v) in edge_list:
            c = counts[(u, v)]
            color_val = c / max_count
            edge_colors.append(plt.cm.Reds(color_val))
            edge_widths.append(1 + c)

        fig, ax = ox.plot_graph(
            self.G,
            figsize=figsize,
            edge_color=edge_colors,
            edge_linewidth=edge_widths,
            show=False,
            close=False
        )
        ax.set_title("Network with Traffic Edge Counts")
        plt.show()

#Example
if __name__ == "__main__":
    # 1) Create the RoadNetwork
    road_net = RoadNetwork(address="Adalbertstraße 58, Berlin, Germany",␣
 ↪dist=1000)

    # 2) Plot the base network
    road_net.plot_network(figsize=(8,8))

    # 3) Create a simulation with 100 cars and jam threshold=5
    sim = TrafficSimulation(road_net, num_cars=100, jam_threshold=5)

    # 4) Run 10 simulation steps
    sim.simulate_traffic(num_steps=10)

    # 5) Visualize the final traffic usage
    sim.plot_network_with_traffic(figsize=(8,8))
```

```
Loading road network for 'Adalbertstraße 58, Berlin, Germany' within dist=1000
meters…
```

```
Extracting largest strongly connected component…
Attaching 'travel_time' attributes to edges…
```



```
Step 1:
Step 2:
Step 3:
Step 4:
Step 5:
Step 6:
Step 7:
Step 8:
Step 9:
Step 10:
```

2 Task 2

2.0.1 (d) Improved Code

Cell 2

[32]:
```
class RoadNetwork_New:
    """
    Manages the loading and preparation of the road network.
    """
    def __init__(self, address: str = "Adalbertstraße 58, Berlin, Germany",␣
↪dist: int = 1000):
```

```
        """
        Initialize and prepare the road network.
        """
        self.address = address
        self.dist = dist
        self.G = self._load_network()
        self._prepare_graph()

    def _load_network(self) -> nx.MultiDiGraph:
        """
        Load the raw road network (a MultiDiGraph) using OSMnx.
        """
        print(f"Loading road network for '{self.address}' within dist={self.
↪dist} meters...")
        G = ox.graph_from_address(self.address, dist=self.dist,␣
↪network_type='drive')
        return G

    def _prepare_graph(self):
        """
        Keep only the largest strongly connected component and
        add a travel_time attribute to each edge.
        """
        print("Extracting largest strongly connected component...")
        largest_scc = max(nx.strongly_connected_components(self.G), key=len)
        self.G = self.G.subgraph(largest_scc).copy()

        print("Adding 'travel_time' edge attribute based on 'length' and␣
↪'maxspeed'...")
        for u, v, k, data in self.G.edges(data=True, keys=True):
            speed = data.get("maxspeed", 30)
            if isinstance(speed, list):
                speed = speed[0]
            try:
                speed_val = float(speed)
            except (ValueError, TypeError):
                speed_val = 30.0
            # length in meters, speed in km/h => travel_time in minutes
            data["travel_time"] = data["length"] / (speed_val * 1000 / 60.0)

    def plot_network(self, figsize=(10, 10)):
        """
        Plot the road network.
        """
        fig, ax = ox.plot_graph(self.G, figsize=figsize)
        return fig, ax
```

```python
class Car_New:
    """
    Represents a single car in the simulation.
    """
    def __init__(self, current_node: int, destination: int):
        self.current_node = current_node
        self.destination = destination
        self.path = []
        self.reached_destination = False
        self.travel_time_spent = 0.0


class TrafficSimulation_New:
    """
    Manages the traffic simulation on the road network, with threshold-based congestion.
    """

    def __init__(self, network: RoadNetwork, num_cars: int = 50, threshold_jam: int = 5):
        self.network = network
        self.G = network.G
        self.threshold_jam = threshold_jam

        self.nodes = list(self.G.nodes())
        self.cars = [self._create_random_car() for _ in range(num_cars)]

        self.step_summaries = []
        self.edge_usage_history = {}

    def _create_random_car(self) -> Car:
        start = random.choice(self.nodes)
        end = random.choice(self.nodes)
        while end == start:
            end = random.choice(self.nodes)
        return Car(start, end)

    def create_cars(self, num_cars: int = 50):
        self.cars = [self._create_random_car() for _ in range(num_cars)]


    def move_cars_one_step(self, step_num: int):
        """
        Threshold-based congestion model:
        If more than threshold_jam cars want the same edge, none of them moves.
```

```
        Otherwise, each car accumulates time with a mild slow factor if multiple cars share the road.
        """
        jammed_edges = set()
        cars_reached_this_step = 0

        # Who wants which edge
        desired_moves = {}
        for car in self.cars:
            if car.reached_destination:
                continue
            if not car.path:
                if car.current_node != car.destination:
                    try:
                        full_path = nx.shortest_path(self.G, car.current_node, car.destination, weight="travel_time")
                        car.path = full_path[1:]
                    except nx.NetworkXNoPath:
                        car.reached_destination = True
                        continue
            if car.path:
                next_node = car.path[0]
                desired_moves[(car.current_node, next_node)] = desired_moves.get((car.current_node, next_node), 0) + 1

        # Record usage
        usage_dict = defaultdict(int)
        for edge, count in desired_moves.items():
            usage_dict[edge] = count
        self.edge_usage_history[step_num] = dict(usage_dict)

        # Move or stay put
        for car in self.cars:
            if car.reached_destination or not car.path:
                continue

            next_node = car.path[0]
            cars_on_edge = desired_moves.get((car.current_node, next_node), 0)
            if cars_on_edge > self.threshold_jam:
                jammed_edges.add((car.current_node, next_node))
            else:
                edge_data = self.G[car.current_node][next_node][0]
                base_time = edge_data["travel_time"]
                factor = 1.0 + 0.1 * cars_on_edge  # mild slowdown if multiple cars
                car.travel_time_spent += base_time * factor
```

```
                car.current_node = next_node
                car.path.pop(0)

                if car.current_node == car.destination:
                    car.reached_destination = True
                    cars_reached_this_step += 1

        cars_in_transit = sum(not c.reached_destination for c in self.cars)
        step_info = {
            "jammed_edges_count": len(jammed_edges),
            "cars_reached_this_step": cars_reached_this_step,
            "cars_in_transit": cars_in_transit
        }
        self.step_summaries.append(step_info)

    def run_simulation(self, num_steps: int = 10):
        for step in range(num_steps):
            self.move_cars_one_step(step_num=step)
            summary = self.step_summaries[-1]
            print(
                f"Step {step+1} complete. "
                f"Jammed edges: {summary['jammed_edges_count']}, "
                f"Cars reached this step: {summary['cars_reached_this_step']}, "
                f"Cars still in transit: {summary['cars_in_transit']}"
            )
        return self.step_summaries

    def finalize_report(self):
        """
        Print final stats: how many cars finished, and average/max travel times.
        """
        finished_cars = [c for c in self.cars if c.reached_destination]
        unfinished_cars = [c for c in self.cars if not c.reached_destination]

        print("\n--- Final Simulation Report ---")
        print(f"Total Cars: {len(self.cars)}")
        print(f"Cars that reached destination: {len(finished_cars)}")
        print(f"Cars still in transit: {len(unfinished_cars)}")

        if finished_cars:
            travel_times = [c.travel_time_spent for c in finished_cars]
            avg_time = sum(travel_times) / len(travel_times)
            max_time = max(travel_times)
            print(f"Average travel time (min): {avg_time:.2f}")
            print(f"Max travel time (min): {max_time:.2f}")
        else:
            print("No cars finished...")
```

```
    def get_edge_car_counts(self):
        """
        For plotting: how many cars plan to use each edge in the next step.
        """
        counts = {}
        for (u, v) in self.G.edges():
            counts[(u, v)] = 0
        for car in self.cars:
            if car.path and not car.reached_destination:
                next_node = car.path[0]
                if (car.current_node, next_node) in counts:
                    counts[(car.current_node, next_node)] += 1
        return counts

    def reset_simulation(self):
        """
        Reset the simulation by creating new random cars.
        """
        num_cars = len(self.cars)
        self.create_cars(num_cars)
        return self


    def plot_network_with_traffic(self):
        """
        Plot the road network, coloring/widening edges in proportion to usage.
        We keep it unchanged from Task 1 for now.
        """
        counts = self.get_edge_car_counts()
        if counts:
            max_count = max(counts.values())
        else:
            max_count = 1
        if max_count == 0:
            max_count = 1

        edge_colors = []
        edge_widths = []
        edge_list = list(self.G.edges())
        for (u, v) in edge_list:
            c = counts[(u, v)]
            color_val = c / max_count
            edge_colors.append(plt.cm.Reds(color_val))
            edge_widths.append(1 + c)

        fig, ax = ox.plot_graph(
```

```
            self.G,
            figsize=(10, 10),
            edge_color=edge_colors,
            edge_linewidth=edge_widths,
            show=False,
            close=False
        )
        ax.set_title("Road Network with Traffic Congestion (Edge Usage)")

        norm = mcolors.Normalize(vmin=0, vmax=max_count)
        sm = ScalarMappable(norm=norm, cmap='Reds')
        sm.set_array([])
        cbar = fig.colorbar(sm, ax=ax, fraction=0.046, pad=0.04)
        cbar.set_label("Number of Cars on Edge")

        plt.show()
```

Session 9-Inspired Continuous Congestion Cell 3

[33]:
```
def compute_congestion_factor(num_cars_on_edge, edge_length, p_slow=0.3):
    """
    A simple approximation of how local congestion can slow traffic.

    num_cars_on_edge: how many cars want to travel this edge
    edge_length: length of the edge in meters (for a rough density)
    p_slow: probability that we randomly slow further (Session 9 concept)

    Returns: A factor (>=1.0) by which to multiply the base travel_time
             1.0 means no slowdown.
    """
    # Suppose each 8 meters of road can comfortably hold 1 car
    capacity = edge_length / 8.0
    density = 0.0
    if capacity > 0:
        density = num_cars_on_edge / capacity

    # Start with a baseline factor
    # The more density, the larger factor becomes -> slower speed
    factor = 1.0 + 2.0 * density  # For example, 2.0 scaling

    # Random slow event with probability p_slow
    if random.random() < p_slow:
        factor *= 1.2  # 20% further slowdown

    return factor

def move_cars_one_step_improved(self, step_num=0, p_slow=0.3):
```

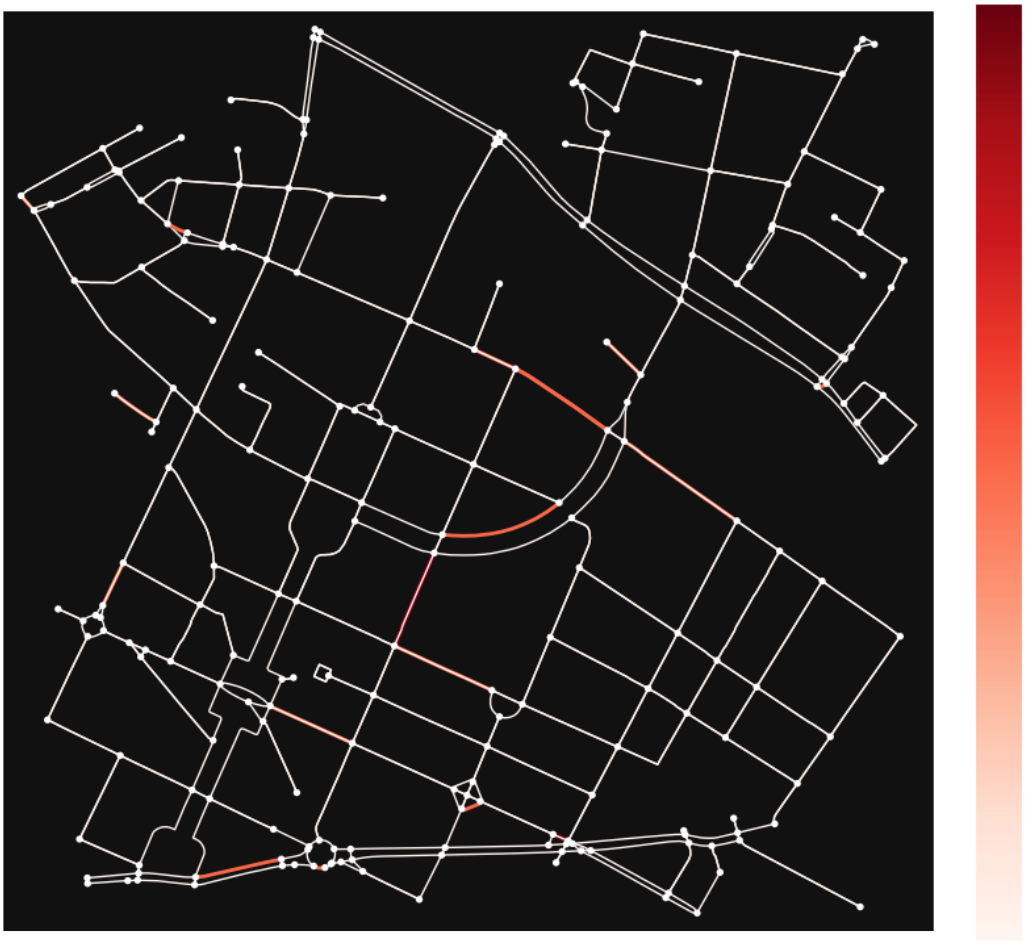
```
    """
    A variation that uses a continuous congestion model instead of threshold␣
↪jam.
    We degrade travel_time by compute_congestion_factor.
    """
    cars_reached_this_step = 0
    desired_moves = {}

    for car in self.cars:
        if car.reached_destination:
            continue
        if not car.path:
            # If empty path but not at destination, try to compute path
            if car.current_node == car.destination:
                car.reached_destination = True
                continue
            try:
                full_path = nx.shortest_path(
                    self.G,
                    car.current_node,
                    car.destination,
                    weight="travel_time"
                )
                car.path = full_path[1:]
            except nx.NetworkXNoPath:
                car.reached_destination = True
                continue

        if car.path:
            next_node = car.path[0]
            desired_moves[(car.current_node, next_node)] = desired_moves.
↪get((car.current_node, next_node), 0) + 1

    # Save usage for analysis
    usage_dict = {}
    for edge, count in desired_moves.items():
        usage_dict[edge] = count
    self.edge_usage_history[step_num] = usage_dict

    # Move cars with continuous slowdown
    for car in self.cars:
        if car.reached_destination or not car.path:
            continue

        next_node = car.path[0]
        num_cars_on_edge = desired_moves.get((car.current_node, next_node), 0)
```

```
        edge_data = self.G[car.current_node][next_node][0]
        base_time = edge_data["travel_time"]
        edge_length = edge_data["length"]

        # Factor depends on local density and random slow
        congestion_factor = compute_congestion_factor(num_cars_on_edge,␣
↪edge_length, p_slow=p_slow)
        car.travel_time_spent += base_time * congestion_factor

        # Move
        car.current_node = next_node
        car.path.pop(0)

        if car.current_node == car.destination:
            car.reached_destination = True
            cars_reached_this_step += 1

    cars_in_transit = sum(not c.reached_destination for c in self.cars)
    step_info = {
        "jammed_edges_count": 0,  # Not using jam threshold in this approach
        "cars_reached_this_step": cars_reached_this_step,
        "cars_in_transit": cars_in_transit
    }
    self.step_summaries.append(step_info)

def run_simulation_improved(self, num_steps=10, p_slow=0.3):
    """
    Run the simulation for a fixed number of steps, but using
    the continuous congestion model rather than jam threshold.
    """
    for step in range(num_steps):
        self.move_cars_one_step_improved(step_num=step, p_slow=p_slow)
        summary = self.step_summaries[-1]
        print(
            f"Step {step+1}, cars reached: {summary['cars_reached_this_step']},␣
↪"
            f"in transit: {summary['cars_in_transit']}"
        )
    return self.step_summaries

# Inject these methods into the TrafficSimulation class:
TrafficSimulation_New.compute_congestion_factor = compute_congestion_factor
TrafficSimulation_New.move_cars_one_step_improved = move_cars_one_step_improved
TrafficSimulation_New.run_simulation_improved = run_simulation_improved
```

Cell 4

[34]:
```
if __name__ == "__main__":
    # 1) Create the road network
    road_net = RoadNetwork_New(address="Adalbertstraße 58, Berlin, Germany",␣
 ↪dist=1000)

    # 2) Create a simulation with artificially large jam threshold
    #    so the old jam logic never triggers
    sim = TrafficSimulation_New(network=road_net, num_cars=50,␣
 ↪threshold_jam=999999)

    # 3) Run the improved simulation for 10 steps with p_slow=0.25
    #    (25% chance of random additional slowdown each move)
    results = sim.run_simulation_improved(num_steps=10, p_slow=0.25)

    # 4) Print a final summary and plot the last-step usage
    sim.finalize_report()
    sim.plot_network_with_traffic()
```

```
Loading road network for 'Adalbertstraße 58, Berlin, Germany' within dist=1000
meters…
Extracting largest strongly connected component…
Adding 'travel_time' edge attribute based on 'length' and 'maxspeed'…
Step 1, cars reached: 0, in transit: 50
Step 2, cars reached: 1, in transit: 49
Step 3, cars reached: 1, in transit: 48
Step 4, cars reached: 2, in transit: 46
Step 5, cars reached: 2, in transit: 44
Step 6, cars reached: 7, in transit: 37
Step 7, cars reached: 4, in transit: 33
Step 8, cars reached: 8, in transit: 25
Step 9, cars reached: 2, in transit: 23
Step 10, cars reached: 2, in transit: 21

--- Final Simulation Report ---
Total Cars: 50
Cars that reached destination: 29
Cars still in transit: 21
Average travel time (min): 1.70
Max travel time (min): 3.25
```



3 Task 3

Cell 5

[35]:
```
def plot_network_with_traffic_enhanced(self, figsize=(10, 10), dpi=100):
    """
    Enhanced visualization of the road network with traffic congestion.
    Dark theme with improved congestion visibility and critical areas␣
 ↪highlighting.
    """
    counts = self.get_edge_car_counts()
    if not counts:
        max_count = 1
    else:
        max_count = max(counts.values())
        if max_count == 0:
```

```
            max_count = 1

    # Create figure with dark background
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=figsize, dpi=dpi)
    fig.patch.set_facecolor('#000000')
    ax.set_facecolor('#000000')

    # Create custom colormap: from light yellow to orange to deep red
    colors = ['#FFFFCC', '#FED976', '#FD8D3C', '#FC4E2A', '#E31A1C', '#B10026']
    custom_cmap = mcolors.LinearSegmentedColormap.from_list('custom_reds',␣
↪colors, N=256)

    # Calculate edge metrics
    edge_colors = []
    edge_widths = []
    edge_list = list(self.G.edges())
    critical_edges = []  # Store critical congestion points

    # Calculate node congestion
    node_congestion = defaultdict(int)

    base_width = 2
    max_width = 10

    for (u, v) in edge_list:
        c = counts[(u, v)]
        node_congestion[u] += c
        node_congestion[v] += c

        # Color mapping
        color_val = c / max_count
        edge_colors.append(custom_cmap(color_val))

        # Width mapping with log scale
        if c > 0:
            width = base_width + (max_width - base_width) * (np.log1p(c) / np.
↪log1p(max_count))
        else:
            width = base_width
        edge_widths.append(width)

        # Identify critical edges (>80% of max traffic)
        if c > 0.8 * max_count:
            critical_edges.append((u, v, c))
```

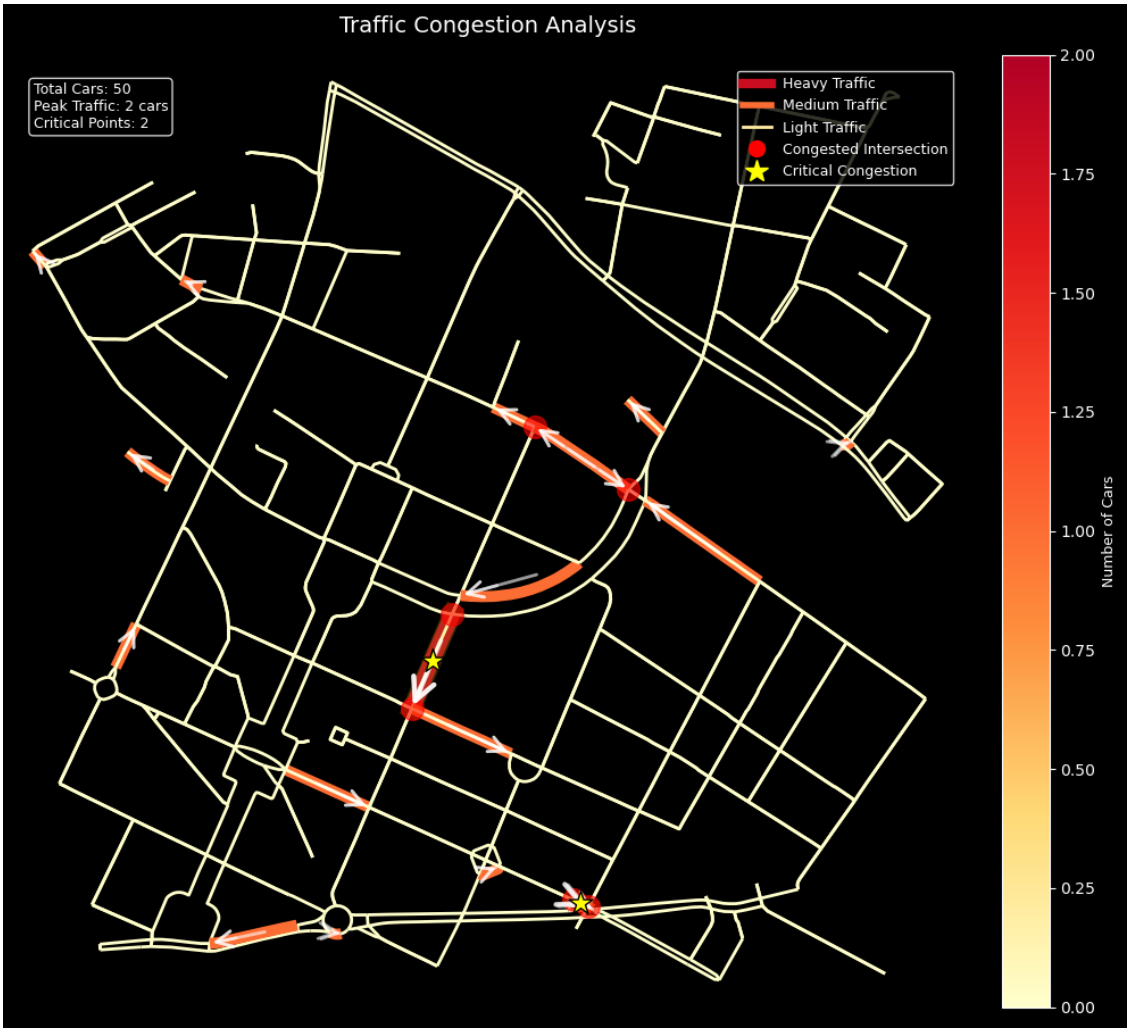
```
    # First highlight critical congestion areas with THINNER and MORE␣
↪TRANSPARENT lines
    pos = {node: (data['x'], data['y']) for node, data in self.G.
↪nodes(data=True)}
    for u, v, count in critical_edges:
        u_pos = pos[u]
        v_pos = pos[v]

        # Create a very subtle glowing effect for critical edges
        for alpha in [0.1, 0.07, 0.04]:  # Very reduced opacity
            plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
                    color='#FFD700',  # Golden yellow
                    alpha=alpha,
                    linewidth=edge_widths[edge_list.index((u,v))] * 1.2)  #␣
↪Much thinner

    # THEN plot the base network (so roads appear on top of highlights)
    ox.plot_graph(
        self.G,
        ax=ax,
        edge_color=edge_colors,
        edge_linewidth=edge_widths,
        show=False,
        close=False,
        node_size=0,
        bgcolor='black'
    )

    # Plot congested intersections
    congested_nodes = []
    for node in self.G.nodes():
        if node_congestion[node] > 0.7 * max_count:  # Highly congested␣
↪intersections
            congested_nodes.append(node)

    if congested_nodes:
        nx.draw_networkx_nodes(
            self.G, pos,
            nodelist=congested_nodes,
            node_size=200,
            node_color='red',
            alpha=0.6,
            ax=ax
        )

    # Add traffic flow arrows with varying sizes based on traffic volume
    for (u, v) in edge_list:
```

```
        if counts[(u, v)] > 0:
            u_pos = pos[u]
            v_pos = pos[v]

            # Calculate arrow position (multiple arrows for heavily congested␣
↪roads)
            traffic_level = counts[(u, v)] / max_count
            num_arrows = 1 + int(traffic_level * 2)  # More arrows for busier␣
↪roads

            for i in range(num_arrows):
                # Distribute arrows along the road
                frac = (i + 1) / (num_arrows + 1)
                mid_x = u_pos[0] * (1 - frac) + v_pos[0] * frac
                mid_y = u_pos[1] * (1 - frac) + v_pos[1] * frac

                # Arrow size based on traffic
                arrow_size = 15 + (traffic_level * 20)
                ax.annotate('',
                            xy=(v_pos[0], v_pos[1]),
                            xytext=(mid_x, mid_y),
                            arrowprops=dict(
                                arrowstyle='->',
                                color='white',
                                linewidth=1 + traffic_level * 2,
                                alpha=0.6,
                                mutation_scale=arrow_size
                            ))

    # Add the star markers for critical congestion - AFTER the arrows
    for u, v, count in critical_edges:
        u_pos = pos[u]
        v_pos = pos[v]

        # Add warning marker for critical congestion
        mid_x = (u_pos[0] + v_pos[0]) / 2
        mid_y = (u_pos[1] + v_pos[1]) / 2
        plt.scatter(mid_x, mid_y,
                    marker='*',  # Star marker
                    s=250,  # Made slightly larger
                    color='yellow',
                    edgecolor='black',
                    alpha=1.0,  # Full opacity
                    zorder=10)  # Ensure it's on top of everything

    # Updated title - removed the star explanation
    ax.set_title("Traffic Congestion Analysis",
```

```
                pad=20, fontsize=14, color='white')

    # Legend with traffic levels
    legend_elements = [
        Line2D([0], [0], color=custom_cmap(0.9), lw=6, label='Heavy Traffic'),
        Line2D([0], [0], color=custom_cmap(0.5), lw=4, label='Medium Traffic'),
        Line2D([0], [0], color=custom_cmap(0.1), lw=2, label='Light Traffic'),
        Line2D([0], [0], color='red', marker='o', markersize=10,
               label='Congested Intersection', linestyle=''),
        Line2D([0], [0], color='yellow', marker='*', markersize=15,
               label='Critical Congestion', linestyle='')
    ]

    ax.legend(handles=legend_elements, loc='upper right',
              bbox_to_anchor=(1.0, 1.0), fontsize=9,
              facecolor='black', edgecolor='white', labelcolor='white')

    # Add colorbar
    norm = mcolors.Normalize(vmin=0, vmax=max_count)
    sm = ScalarMappable(norm=norm, cmap=custom_cmap)
    sm.set_array([])
    cbar = fig.colorbar(sm, ax=ax, fraction=0.046, pad=0.04)
    cbar.set_label("Number of Cars", color='white', fontsize=9)
    cbar.ax.yaxis.set_tick_params(color='white')
    plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'), color='white')

    # Statistics box
    stats_text = (
        f"Total Cars: {len(self.cars)}\n"
        f"Peak Traffic: {max_count} cars\n"
        f"Critical Points: {len(critical_edges)}"
    )
    plt.text(0.02, 0.98, stats_text,
             transform=ax.transAxes,
             verticalalignment='top',
             color='white',
             fontsize=9,
             bbox=dict(boxstyle='round',
                       facecolor='black',
                       edgecolor='white',
                       alpha=0.8))

    plt.tight_layout()
    return fig, ax

# Add the method to the TrafficSimulation class
```

```
TrafficSimulation_New.plot_network_with_traffic_enhanced =␣
↪plot_network_with_traffic_enhanced
```

[36]:
```
fig, ax = sim.plot_network_with_traffic_enhanced(figsize=(10, 10))
```



Cell 6

[51]:
```
def make_dual_traffic_animation(sim, total_frames=30, steps_per_frame=1,␣
↪interval=200, figsize=(18, 8), dpi=100):
    """
    Create a side-by-side animation showing:
    1. Current traffic state with cars moving
    2. Cumulative congestion patterns developing over time
    """

    # Store original state to restore later
```

```
    original_cars = sim.cars.copy()

    # Reset simulation with fresh cars
    sim.reset_simulation()

    # Create figure with dark background and proper layout - two subplots side␣
↪by side
    plt.style.use('dark_background')
    fig = plt.figure(figsize=figsize, dpi=dpi)
    gs = fig.add_gridspec(1, 20)  # 20 columns for better control

    # Left plot for current traffic
    ax1 = fig.add_subplot(gs[0, :9])
    # Right plot for cumulative congestion
    ax2 = fig.add_subplot(gs[0, 10:19])
    # Colorbar area
    cax = fig.add_subplot(gs[0, 19:])

    fig.patch.set_facecolor('#000000')
    ax1.set_facecolor('#000000')
    ax2.set_facecolor('#000000')

    # Create custom colormaps - MODIFIED FOR HIGHER CONTRAST
    colors_current = ['#FFFFCC', '#FED976', '#FD8D3C', '#FC4E2A', '#E31A1C',␣
↪'#B10026']
    # Higher contrast for cumulative coloring (darker blues to darker reds)
    colors_cumulative = ['#0D47A1', '#1976D2', '#42A5F5', '#FFEE58', '#FF9800',␣
↪'#E65100', '#BF360C', '#7f0000']

    current_cmap = mcolors.LinearSegmentedColormap.from_list('current_traffic',␣
↪colors_current, N=256)
    cumulative_cmap = mcolors.LinearSegmentedColormap.
↪from_list('cumulative_traffic', colors_cumulative, N=256)

    # Get node positions
    pos = {node: (data['x'], data['y']) for node, data in sim.G.
↪nodes(data=True)}

    # Setup progress bar
    progress_bar = tqdm(total=total_frames)

    # Create fixed colorbars once
    norm_current = mcolors.Normalize(vmin=0, vmax=5)  # For current traffic
    sm_current = ScalarMappable(norm=norm_current, cmap=current_cmap)
    sm_current.set_array([])
```

```
    norm_cumulative = mcolors.Normalize(vmin=0, vmax=total_frames)  # For total␣
↪congestion count
    sm_cumulative = ScalarMappable(norm=norm_cumulative, cmap=cumulative_cmap)
    sm_cumulative.set_array([])

    # We'll share the colorbar for the cumulative plot (right)
    cbar = fig.colorbar(sm_cumulative, cax=cax)
    cbar.set_label("Congestion Count", color='white', fontsize=9)
    cax.yaxis.set_tick_params(color='white')
    plt.setp(plt.getp(cax.axes, 'yticklabels'), color='white')

    # Initialize congestion tracking - count total instances of congestion
    cumulative_congestion_count = defaultdict(int)
    road_id_map = {}  # To map road IDs to more readable names

    # Create more readable road names for the top congested roads
    for i, (u, v) in enumerate(sim.G.edges()):
        road_id_map[(u, v)] = f"Road {i+1}"

    def animate(frame):
        ax1.clear()  # Clear current traffic plot
        ax2.clear()  # Clear cumulative plot

        # Run simulation steps - with monitoring to ensure cars are moving
        if frame > 0:
            cars_in_motion_before = sum(1 for car in sim.cars if hasattr(car,␣
↪'path') and car.path)

            for _ in range(steps_per_frame):
                sim.move_cars_one_step(step_num=frame)

                # Check if cars are getting stuck
                cars_in_motion_after = sum(1 for car in sim.cars if␣
↪hasattr(car, 'path') and car.path)

                # If many cars have no paths, recreate some new cars
                if frame % 10 == 0 and cars_in_motion_after < len(sim.cars) * 0.
↪5:
                    # Reset some cars that have reached their destination
                    for i, car in enumerate(sim.cars):
                        if not car.path or car.current_node == car.destination:
                            # Only reset some cars to maintain continuity
                            if i % 3 == 0:  # Reset roughly 1/3 of idle cars
                                nodes = list(sim.G.nodes())
                                car.current_node = np.random.choice(nodes)
                                car.destination = np.random.choice(nodes)
```

```
                                car.path = []  # Will be recalculated in next␣
↪move_cars()

        # Get current traffic data
        counts = sim.get_edge_car_counts()
        if not counts:
            max_count = 1
        else:
            max_count = max(max(counts.values()), 1)

        # Update current traffic colorbar if needed
        if max_count > norm_current.vmax:
            norm_current.vmax = max_count

        # Track congestion over time - TRUE CUMULATIVE COUNTING
        congestion_threshold = 0.5 * max_count
        for edge, count in counts.items():
            if count >= congestion_threshold:
                cumulative_congestion_count[edge] += 1

        # Get the top congested roads
        top_congested_roads = sorted(cumulative_congestion_count.items(),
                                     key=lambda x: x[1], reverse=True)[:5]

        # Calculate node congestion for current frame
        node_congestion = defaultdict(int)
        for (u, v), count in counts.items():
            node_congestion[u] += count
            node_congestion[v] += count

        # Prepare edge visualization data for CURRENT traffic (left plot)
        current_edge_colors = []
        current_edge_widths = []
        edge_list = list(sim.G.edges())
        critical_edges = []

        base_width = 2
        max_width = 10

        for (u, v) in edge_list:
            c = counts.get((u, v), 0)

            # Color mapping for current traffic
            color_val = c / max(max_count, 0.01)
            current_edge_colors.append(current_cmap(color_val))

            # Width mapping with log scale
```

```
            if c > 0:
                width = base_width + (max_width - base_width) * (np.log1p(c) /␣
↪np.log1p(max_count))
            else:
                width = base_width
            current_edge_widths.append(width)

            # Identify critical edges (>80% of max traffic)
            if c > 0.8 * max_count and max_count > 1:
                critical_edges.append((u, v, c))

        # Prepare data for CUMULATIVE traffic visualization (right plot)
        cumulative_edge_colors = []
        cumulative_edge_widths = []

        # Adjust the max value for normalization
        max_cumulative = max(cumulative_congestion_count.values()) if␣
↪cumulative_congestion_count else 1
        norm_cumulative.vmax = max(max_cumulative, 1)

        for (u, v) in edge_list:
            count = cumulative_congestion_count.get((u, v), 0)

            # Normalize color based on the maximum observed count
            color_val = count / max(norm_cumulative.vmax, 0.01)
            cumulative_edge_colors.append(cumulative_cmap(color_val))

            # Width based on congestion count - MODIFIED for better visibility
            if count > 0:
                # Enhanced scaling for congested roads (minimum width 3.0)
                c_width = max(3.0, base_width + (max_width - base_width) *␣
↪(count / max(norm_cumulative.vmax, 0.01)))
            else:
                c_width = base_width
            cumulative_edge_widths.append(c_width)

        # PLOT 1: CURRENT TRAFFIC (LEFT)
        # ==============================

        # Draw all road segments on left plot
        for i, (u, v) in enumerate(edge_list):
            u_pos = pos[u]
            v_pos = pos[v]
            ax1.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
                     color=current_edge_colors[i],␣
↪linewidth=current_edge_widths[i], zorder=2)
```

```
        # Plot congested intersections
        congested_nodes = []
        for node in sim.G.nodes():
            if node_congestion[node] > 0.7 * max_count:
                congested_nodes.append(node)

        if congested_nodes:
            node_x = [pos[node][0] for node in congested_nodes]
            node_y = [pos[node][1] for node in congested_nodes]
            ax1.scatter(node_x, node_y, s=200, c='red', alpha=0.6, zorder=3)

        # Add critical congestion markers
        for u, v, count in critical_edges:
            u_pos = pos[u]
            v_pos = pos[v]
            mid_x = (u_pos[0] + v_pos[0]) / 2
            mid_y = (u_pos[1] + v_pos[1]) / 2
            ax1.scatter(mid_x, mid_y, marker='*', s=250, color='yellow',
                        edgecolor='black', alpha=1.0, zorder=10)

        # Plot all cars with a glowing effect
        car_coords = []
        for car in sim.cars:
            if hasattr(car, 'path') and car.path and car.current_node != car.
↪destination:
                # Car is in transit
                u_pos = pos[car.current_node]
                car_coords.append(u_pos)

        if car_coords:
            # Create glow effect for cars
            car_x = [c[0] for c in car_coords]
            car_y = [c[1] for c in car_coords]

            # Add slight glow
            for alpha, size in [(0.3, 100), (0.5, 70), (0.9, 40)]:
                ax1.scatter(car_x, car_y, c='cyan', s=size, alpha=alpha,␣
↪edgecolor=None, zorder=5)

        # Add title to left plot
        ax1.set_title(f"Current Traffic - Step {frame}",
                      fontsize=12, color='white', pad=20)

        # Stats for left plot
        stats_text1 = (
            f"Step: {frame}/{total_frames}\n"
            f"Total Cars: {len(sim.cars)}\n"
```

```
            f"Cars in Motion: {len(car_coords)}\n"
            f"Peak Traffic: {max_count} cars"
        )

        ax1.text(0.02, 0.98, stats_text1,
                transform=ax1.transAxes,
                verticalalignment='top',
                color='white',
                fontsize=9,
                bbox=dict(boxstyle='round',
                          facecolor='black',
                          edgecolor='white',
                          alpha=0.8))

        # PLOT 2: CUMULATIVE CONGESTION (RIGHT)
        # =====================================

        # First draw a base network in dim color for better contrast
        for (u, v) in edge_list:
            u_pos = pos[u]
            v_pos = pos[v]
            ax2.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
                    color='#252550', linewidth=1.5, zorder=1)

        # Draw all road segments on right plot
        for i, (u, v) in enumerate(edge_list):
            if cumulative_congestion_count.get((u, v), 0) > 0:
                u_pos = pos[u]
                v_pos = pos[v]
                ax2.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
                        color=cumulative_edge_colors[i],␣
↪linewidth=cumulative_edge_widths[i], zorder=2)

        # Add markers for the top 5 most congested roads
        for i, ((u, v), count) in enumerate(top_congested_roads):
            u_pos = pos[u]
            v_pos = pos[v]
            mid_x = (u_pos[0] + v_pos[0]) / 2
            mid_y = (u_pos[1] + v_pos[1]) / 2

            # Add numbered marker to identify specific roads
            ax2.scatter(mid_x, mid_y, marker='*', s=300, color='yellow',
                       edgecolor='black', alpha=1.0, zorder=10)
            ax2.text(mid_x, mid_y, str(i+1), color='black', fontsize=10,
                    ha='center', va='center', weight='bold', zorder=11)

            # Add road label near the marker
```

```
            road_name = f"Road {i+1}"

            # Use fixed offsets for consistent label placement
            offsets = [(0.004, 0.004), (-0.004, 0.004), (0.004, -0.004), (-0.
↪004, -0.004), (0, 0.006)]
            if i < len(offsets):
                offset_x, offset_y = offsets[i]
                ax2.text(mid_x + offset_x, mid_y + offset_y, road_name,␣
↪color='yellow',
                         fontsize=8, ha='center', va='center', weight='bold',
                         bbox=dict(facecolor='black', alpha=0.7,␣
↪boxstyle='round,pad=0.2'),
                         zorder=12)

        # Add title to right plot
        ax2.set_title(f"Cumulative Congestion Patterns",
                      fontsize=12, color='white', pad=20)

        # Stats for right plot - list the most congested roads with MATCHING␣
↪NUMBERS
        if top_congested_roads:
            stats_text2 = "Most Congested Roads:\n"
            for i, ((u, v), count) in enumerate(top_congested_roads):
                # Show road number matching the marker and count of congestion␣
↪events
                stats_text2 += f"{i+1}. Road {i+1} ({u}, {v}): {count} times\n"
        else:
            stats_text2 = "No congestion patterns yet"

        ax2.text(0.02, 0.98, stats_text2,
                 transform=ax2.transAxes,
                 verticalalignment='top',
                 color='white',
                 fontsize=9,
                 bbox=dict(boxstyle='round',
                           facecolor='black',
                           edgecolor='white',
                           alpha=0.8))

        # LEGENDS
        # =======

        # Legend for left plot (current traffic)
        legend_elements1 = [
            Line2D([0], [0], color=current_cmap(0.9), lw=4, label='Heavy␣
↪Traffic'),
```

```
            Line2D([0], [0], color=current_cmap(0.5), lw=3, label='Medium␣
↪Traffic'),
            Line2D([0], [0], color=current_cmap(0.1), lw=2, label='Light␣
↪Traffic'),
            Line2D([0], [0], color='cyan', marker='o', markersize=8,␣
↪label='Cars', linestyle='')
        ]

        ax1.legend(handles=legend_elements1, loc='lower right',
                   fontsize=8, facecolor='black', edgecolor='white',␣
↪labelcolor='white')

        # Legend for right plot (cumulative congestion)
        legend_elements2 = [
            Line2D([0], [0], color=cumulative_cmap(0.9), lw=4, label='High␣
↪Congestion'),
            Line2D([0], [0], color=cumulative_cmap(0.5), lw=3, label='Medium␣
↪Congestion'),
            Line2D([0], [0], color=cumulative_cmap(0.1), lw=2, label='Low␣
↪Congestion'),
            Line2D([0], [0], color='yellow', marker='*', markersize=10,␣
↪label='Top Congested', linestyle='')
        ]

        ax2.legend(handles=legend_elements2, loc='lower right',
                   fontsize=8, facecolor='black', edgecolor='white',␣
↪labelcolor='white')

        # Remove tick labels
        for ax in [ax1, ax2]:
            ax.set_xticklabels([])
            ax.set_yticklabels([])
            ax.tick_params(axis='both', which='both', length=0)

        progress_bar.update(1)
        return []

    # Create animation
    anim = FuncAnimation(fig, animate, frames=total_frames, interval=interval,␣
↪blit=False)

    # Convert to JavaScript animation
    html_animation = HTML(anim.to_jshtml())

    # Clean up
    plt.close(fig)
```

```
    progress_bar.close()

    # Restore original state
    sim.cars = original_cars

    # Return animation
    return html_animation
```

[40]:
```
!pip install tqdm
```

```
Collecting tqdm
  Downloading tqdm-4.67.1-py3-none-any.whl.metadata (57 kB)
Downloading tqdm-4.67.1-py3-none-any.whl (78 kB)
Installing collected packages: tqdm
Successfully installed tqdm-4.67.1
```

[52]:
```
# Create dual animation with 30 frames
animation = make_dual_traffic_animation(sim, total_frames=30,␣
 ↪steps_per_frame=1, interval=200)
display(animation)
```

```
  7%|                                          | 2/30 [04:06<57:37, 123.50s/it]
31it [00:19,  1.61it/s]
```

```
<IPython.core.display.HTML object>
```

[ ]:

[ ]:

4 Task 4

4.0.1 (a)

Cell 7

[54]:
```
!pip install seaborn
```

```
Collecting seaborn
  Downloading seaborn-0.13.2-py3-none-any.whl.metadata (5.4 kB)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from seaborn)
(2.2.4)
Requirement already satisfied: pandas>=1.2 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from seaborn) (2.2.3)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from seaborn)
(3.10.1)
Requirement already satisfied: contourpy>=1.0.1 in ./opt/anaconda3/envs/traffic-
```

```
sim/lib/python3.10/site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.3.1)
Requirement already satisfied: cycler>=0.10 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from matplotlib!=3.6.1,>=3.4->seaborn)
(0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (4.57.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (1.4.8)
Requirement already satisfied: packaging>=20.0 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (24.2)
Requirement already satisfied: pillow>=8 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from matplotlib!=3.6.1,>=3.4->seaborn)
(11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from pandas>=1.2->seaborn) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from pandas>=1.2->seaborn) (2025.2)
Requirement already satisfied: six>=1.5 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from python-
dateutil>=2.7->matplotlib!=3.6.1,>=3.4->seaborn) (1.17.0)
Downloading seaborn-0.13.2-py3-none-any.whl (294 kB)
Installing collected packages: seaborn
Successfully installed seaborn-0.13.2
```

Empirical Analysis

[108]:
```python
def run_batch_experiment(
    network,
    num_cars=100,
    steps_per_run=30,
    num_runs=50,
    jam_threshold=5,
    congestion_threshold=0.5,
    seed=42
):
    """
    Run the simulation multiple times and analyze congestion patterns.

    Parameters:
    - network: The road network to simulate on
    - num_cars: Number of cars in each simulation
```

```
    - steps_per_run: Number of steps per simulation run
    - num_runs: Number of independent simulation runs
    - jam_threshold: Threshold for traffic jam in the simulation
    - congestion_threshold: Fraction of peak load that defines congestion (e.g.
↪, 0.5 = 50%)
    - seed: Random seed for reproducibility

    Returns:
    - DataFrame with congestion frequencies for each edge
    - DataFrame with cars in motion over time for each run
    - DataFrame with number of congested edges over time for each run
    """
    # Set random seed for reproducibility
    random.seed(seed)
    np.random.seed(seed)

    # Get all edges in the network
    all_edges = list(network.G.edges())

    # Initialize results storage
    congestion_counts = {edge: 0 for edge in all_edges}
    cars_in_motion_data = np.zeros((num_runs, steps_per_run))
    congested_edges_count_data = np.zeros((num_runs, steps_per_run))

    # Run multiple simulations
    for run in tqdm(range(num_runs), desc="Running simulations"):
        # Create a new simulation with the specified parameters
        sim = TrafficSimulation_New(network=network, num_cars=num_cars,␣
↪threshold_jam=jam_threshold)

        # Run the simulation for the specified number of steps
        for step in range(steps_per_run):
            # Move cars one step
            sim.move_cars_one_step(step_num=step)

            # Get edge loads (how many cars want to use each edge)
            edge_loads = sim.get_edge_car_counts()

            # Find peak load in this step
            peak_load = max(edge_loads.values()) if edge_loads else 0

            # Count congested edges in this step
            congested_edges_count = 0

            # Apply congestion rule: edge is congested if load  50% of peak␣
↪load
            if peak_load > 0:
```

```
                congestion_threshold_value = congestion_threshold * peak_load
                for edge, load in edge_loads.items():
                    if load >= congestion_threshold_value:
                        congestion_counts[edge] += 1
                        congested_edges_count += 1

            # Count cars in motion
            cars_in_motion = sum(1 for car in sim.cars
                                 if not car.reached_destination and hasattr(car,␣
↪'path') and car.path)

            # Store data for this step
            cars_in_motion_data[run, step] = cars_in_motion
            congested_edges_count_data[run, step] = congested_edges_count

    # Calculate congestion frequency for each edge
    total_steps = num_runs * steps_per_run
    congestion_freq = {edge: count / total_steps for edge, count in␣
↪congestion_counts.items()}

    # Create DataFrame with congestion frequencies
    cong_freq_df = pd.DataFrame([
        {'u': u, 'v': v, 'cong_freq': congestion_freq[(u, v)]}
        for u, v in all_edges
    ])
    cong_freq_df.set_index(['u', 'v'], inplace=True)

    # Create DataFrame for cars in motion over time
    cars_motion_df = pd.DataFrame(cars_in_motion_data)
    cars_motion_df = pd.melt(cars_motion_df.reset_index(),
                             id_vars=['index'],
                             var_name='step',
                             value_name='cars_in_motion')
    cars_motion_df.rename(columns={'index': 'run'}, inplace=True)

    # Create DataFrame for congested edges count over time
    congested_edges_df = pd.DataFrame(congested_edges_count_data)
    congested_edges_df = pd.melt(congested_edges_df.reset_index(),
                                 id_vars=['index'],
                                 var_name='step',
                                 value_name='congested_edges_count')
    congested_edges_df.rename(columns={'index': 'run'}, inplace=True)

    return cong_freq_df, cars_motion_df, congested_edges_df

def analyze_congestion_results(cong_freq_df, cars_motion_df, congested_edges_df,
                               network, save_to_csv=None, title_prefix=""):
```

```
    """
    Analyze and visualize congestion results from the batch experiment.

    Parameters:
    - cong_freq_df: DataFrame with congestion frequencies
    - cars_motion_df: DataFrame with cars in motion data
    - congested_edges_df: DataFrame with congested edges count data
    - network: The road network used in the simulation
    - save_to_csv: Path to save the congestion frequency DataFrame (optional)
    - title_prefix: Prefix for plot titles (useful when comparing different␣
↪scenarios)

    Returns:
    - None (displays plots and prints statistics)
    """
    # 1. Print basic statistics in a table
    print(f"{title_prefix} Congestion Frequency Statistics:")
    stats_df = pd.DataFrame(cong_freq_df['cong_freq'].describe()).reset_index()
    stats_df.columns = ['Statistic', 'Value']
    print(tabulate(stats_df, headers='keys', tablefmt='grid', floatfmt='.6f'))

    # Save to CSV if requested
    if save_to_csv:
        cong_freq_df.to_csv(save_to_csv)
        print(f"Saved congestion frequencies to {save_to_csv}")

    # Show top 10 most congested roads in a table
    top_10_roads = cong_freq_df.sort_values('cong_freq', ascending=False).
↪head(10).reset_index()
    top_10_roads['Road'] = top_10_roads.apply(lambda x: f"({x['u']},{x['v']})",␣
↪axis=1)
    top_10_table = top_10_roads[['Road', 'cong_freq']].copy()
    top_10_table.columns = ['Road', 'Congestion Frequency']
    print(f"\n{title_prefix} Top 10 Most Congested Roads:")
    print(tabulate(top_10_table, headers='keys', tablefmt='grid', floatfmt='.
↪6f'))

    # Set style to default for white background
    plt.style.use('default')

    # Set up the figure layout
    plt.figure(figsize=(18, 14))

    # 2. Histogram of congestion frequencies
    plt.subplot(2, 2, 1)
    sns.histplot(cong_freq_df['cong_freq'], bins=20, kde=True)
    plt.title(f"{title_prefix}Distribution of Road Congestion Frequencies")
```

```
    plt.xlabel("Congestion Frequency")
    plt.ylabel("Number of Roads")

    # 3. Time-series line plot of cars in motion over time
    plt.subplot(2, 2, 2)
    cars_motion_avg = cars_motion_df.groupby('step')['cars_in_motion'].
↪agg(['mean', 'std']).reset_index()
    sns.lineplot(x='step', y='mean', data=cars_motion_avg)
    plt.fill_between(cars_motion_avg['step'],
                     cars_motion_avg['mean'] - cars_motion_avg['std'],
                     cars_motion_avg['mean'] + cars_motion_avg['std'],
                     alpha=0.3)
    plt.title(f"{title_prefix}Average Cars in Motion Over Time")
    plt.xlabel("Simulation Step")
    plt.ylabel("Number of Cars")

    # 4. Bar plot of congested edges over time
    plt.subplot(2, 2, 3)
    congested_edges_avg = congested_edges_df.
↪groupby('step')['congested_edges_count'].agg(['mean', 'std']).reset_index()
    sns.barplot(x='step', y='mean', data=congested_edges_avg)
    plt.errorbar(x=congested_edges_avg['step'], y=congested_edges_avg['mean'],
                 yerr=congested_edges_avg['std'], fmt='none', c='black',␣
↪capsize=3)
    plt.title(f"{title_prefix}Average Number of Congested Roads per Step")
    plt.xlabel("Simulation Step")
    plt.ylabel("Number of Congested Roads")
    plt.xticks(rotation=45)

    # 5. Bar chart of top 10 most congested roads
    plt.subplot(2, 2, 4)
    road_labels = [f"({u},{v})" for u, v in top_10_roads[['u', 'v']].values]
    sns.barplot(x=top_10_roads['cong_freq'], y=road_labels)
    plt.title(f"{title_prefix}Top 10 Most Congested Roads")
    plt.xlabel("Congestion Frequency")
    plt.ylabel("Road (u,v)")

    plt.tight_layout()
    plt.show()

    # 6. Network visualization with congestion heatmap
    plt.figure(figsize=(12, 10))
    plot_network_congestion_heatmap(network, cong_freq_df,␣
↪title=f"{title_prefix}Road Network Congestion Heatmap")
    plt.show()
```

```
def plot_network_congestion_heatmap(network, cong_freq_df, title="Road Network␣
↪Congestion Heatmap"):
    """
    Plot the road network with edges colored according to congestion frequency.

    Parameters:
    - network: The road network
    - cong_freq_df: DataFrame with congestion frequencies
    - title: Plot title

    Returns:
    - None (displays plot)
    """
    # Set up plot with dark background
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(12, 10))
    fig.patch.set_facecolor('#000000')
    ax.set_facecolor('#000000')

    # Create custom colormap (from blue to yellow to red)
    colors = ['#0D47A1', '#1976D2', '#42A5F5', '#FFEE58', '#FFA726', '#E65100',␣
↪'#BF360C']
    custom_cmap = mcolors.LinearSegmentedColormap.from_list('congestion_cmap',␣
↪colors, N=256)

    # Get max congestion frequency for normalization
    max_cong_freq = cong_freq_df['cong_freq'].max()

    # Extract coordinates of nodes
    pos = {node: (data['x'], data['y']) for node, data in network.G.
↪nodes(data=True)}

    # Sort edges by congestion frequency (to plot high-congestion roads on top)
    edge_data = []
    for (u, v), row in cong_freq_df.iterrows():
        edge_data.append((u, v, row['cong_freq']))

    edge_data.sort(key=lambda x: x[2])

    # Draw edges with variable width and color based on congestion frequency
    for u, v, freq in edge_data:
        u_pos = pos[u]
        v_pos = pos[v]

        # Skip or draw edges with minimal congestion in light gray
        if freq < 0.01:
            plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
```

```
                    color='#555555', linewidth=1, alpha=0.3, zorder=1)
            continue

        # Color and width proportional to congestion frequency
        color_val = freq / max_cong_freq
        color = custom_cmap(color_val)
        width = 1 + 9 * color_val  # Width from 1 to 10

        plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
                 color=color, linewidth=width, alpha=0.8, zorder=2)

    # Mark top congested edges
    top_edges = cong_freq_df.sort_values('cong_freq', ascending=False).head(10)
    for i, ((u, v), row) in enumerate(top_edges.iterrows()):
        u_pos = pos[u]
        v_pos = pos[v]
        mid_x = (u_pos[0] + v_pos[0]) / 2
        mid_y = (u_pos[1] + v_pos[1]) / 2

        # Add a star marker
        plt.scatter(mid_x, mid_y,
                    marker='*',
                    s=200,
                    color='yellow',
                    edgecolor='black',
                    zorder=10)

        # Add label with rank
        plt.text(mid_x, mid_y, str(i+1),
                 fontsize=10,
                 ha='center',
                 va='center',
                 color='black',
                 weight='bold',
                 zorder=11)

    # Add a colorbar
    norm = mcolors.Normalize(vmin=0, vmax=max_cong_freq)
    sm = ScalarMappable(norm=norm, cmap=custom_cmap)
    sm.set_array([])
    cbar = fig.colorbar(sm, ax=ax, fraction=0.046, pad=0.04)
    cbar.set_label("Congestion Frequency", color='white', fontsize=10)
    plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'), color='white')

    # Add legend for top congested roads
    legend_text = "Top 10 Congested Roads:\n"
    for i, ((u, v), row) in enumerate(top_edges.iterrows()):
```

```
        legend_text += f"{i+1}: ({u},{v}) - {row['cong_freq']:.3f}\n"

    plt.text(0.02, 0.98, legend_text,
             transform=ax.transAxes,
             verticalalignment='top',
             color='white',
             fontsize=9,
             bbox=dict(boxstyle='round',
                       facecolor='black',
                       edgecolor='white',
                       alpha=0.8))

    # Set title and remove ticks
    plt.title(title, color='white', pad=20, fontsize=14)
    ax.set_xticks([])
    ax.set_yticks([])

    return fig, ax

# Run the empirical analysis with the parameters specified
if __name__ == "__main__":
    # Parameters for the experiment
    NUM_CARS = 100
    STEPS_PER_RUN = 30
    NUM_RUNS = 50
    JAM_THRESHOLD = 5

    # Create road network
    print("Creating road network...")
    road_net = RoadNetwork_New(address="Adalbertstraße 58, Berlin, Germany",␣
 ↪dist=1000)

    # Run batch experiment with 100 cars
    print(f"Running batch experiment with {NUM_CARS} cars...")
    cong_freq_df, cars_motion_df, congested_edges_df = run_batch_experiment(
        network=road_net,
        num_cars=NUM_CARS,
        steps_per_run=STEPS_PER_RUN,
        num_runs=NUM_RUNS,
        jam_threshold=JAM_THRESHOLD
    )

    # Analyze results
    print("Analyzing results...")
    analyze_congestion_results(
        cong_freq_df,
        cars_motion_df,
```

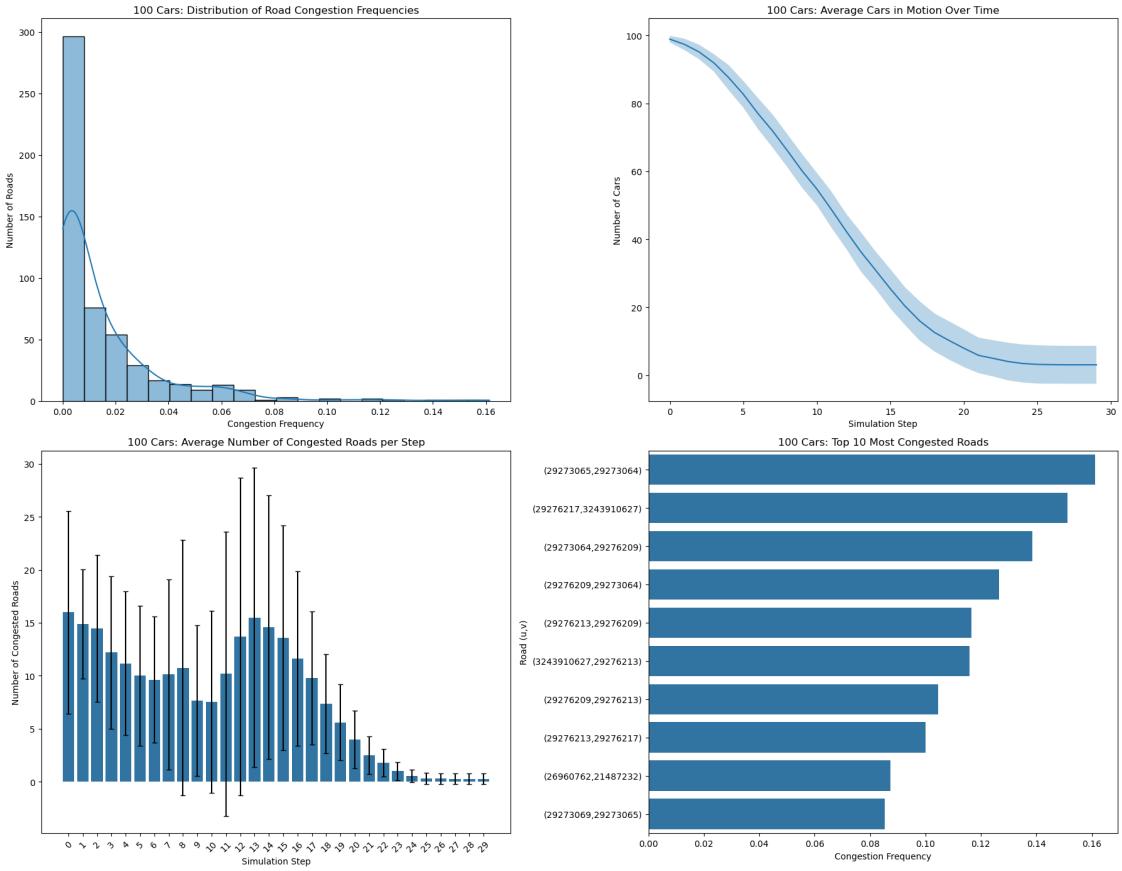
```
        congested_edges_df,
        road_net,
        save_to_csv="congestion_results_100cars.csv",
        title_prefix="100 Cars: "
    )

    # Optional: Run with 150 cars
    NUM_CARS_HIGHER = 150
    print(f"Running batch experiment with {NUM_CARS_HIGHER} cars...")
    cong_freq_df_150, cars_motion_df_150, congested_edges_df_150 =␣
↪run_batch_experiment(
        network=road_net,
        num_cars=NUM_CARS_HIGHER,
        steps_per_run=STEPS_PER_RUN,
        num_runs=NUM_RUNS,
        jam_threshold=JAM_THRESHOLD
    )

    # Analyze results for 150 cars
    print("Analyzing results for 150 cars...")
    analyze_congestion_results(
        cong_freq_df_150,
        cars_motion_df_150,
        congested_edges_df_150,
        road_net,
        save_to_csv="congestion_results_150cars.csv",
        title_prefix="150 Cars: "
    )

    # Compare the two scenarios
    print("Comparing results between 100 and 150 cars...")

    # Create a comparison DataFrame
    comparison_df = pd.DataFrame({
        'cong_freq_100': cong_freq_df['cong_freq'],
        'cong_freq_150': cong_freq_df_150['cong_freq']
    })

    # Calculate the difference
    comparison_df['diff'] = comparison_df['cong_freq_150'] -␣
↪comparison_df['cong_freq_100']

    # Sort by the absolute difference
    comparison_df = comparison_df.sort_values('diff', ascending=False)

    # Display top roads with biggest differences in a table
    top_diff_roads = comparison_df.head(15).reset_index()
```

```
    top_diff_roads['Road'] = top_diff_roads.apply(lambda x:␣
↪f"({x['u']},{x['v']})", axis=1)
    top_diff_table = top_diff_roads[['Road', 'cong_freq_100', 'cong_freq_150',␣
↪'diff']].copy()
    top_diff_table.columns = ['Road', 'Congestion (100 Cars)', 'Congestion (150␣
↪Cars)', 'Difference']

    print("\nTop 15 Roads with Biggest Congestion Changes:")
    print(tabulate(top_diff_table, headers='keys', tablefmt='grid', floatfmt='.
↪6f'))

    # Plot the differences for top roads
    plt.figure(figsize=(12, 8))
    road_labels = [f"({u},{v})" for u, v in top_diff_roads[['u', 'v']].values]

    plt.barh(range(len(road_labels)), top_diff_roads['cong_freq_150'],
             color='orange', alpha=0.7, label='150 Cars')
    plt.barh(range(len(road_labels)), top_diff_roads['cong_freq_100'],
             color='blue', alpha=0.7, label='100 Cars')

    plt.yticks(range(len(road_labels)), road_labels)
    plt.xlabel('Congestion Frequency')
    plt.ylabel('Road (u,v)')
    plt.title('Comparison of Top Roads with Increased Congestion (150 vs 100␣
↪Cars)')
    plt.legend()
    plt.tight_layout()
    plt.show()

    # Print summary statistics about the changes in a table
    print("\nSummary of Changes in Congestion Frequency:")
    summary_stats = [
        ["Average change", comparison_df['diff'].mean()],
        ["Maximum increase", comparison_df['diff'].max()],
        ["Maximum decrease", comparison_df['diff'].min()],
        ["Roads with >10% increase", (comparison_df['diff'] > 0.1).sum()],
        ["Roads with any increase", (comparison_df['diff'] > 0).sum()],
        ["Roads with any decrease", (comparison_df['diff'] < 0).sum()]
    ]

    print(tabulate(summary_stats, headers=['Metric', 'Value'], tablefmt='grid',␣
↪floatfmt='.6f'))
```

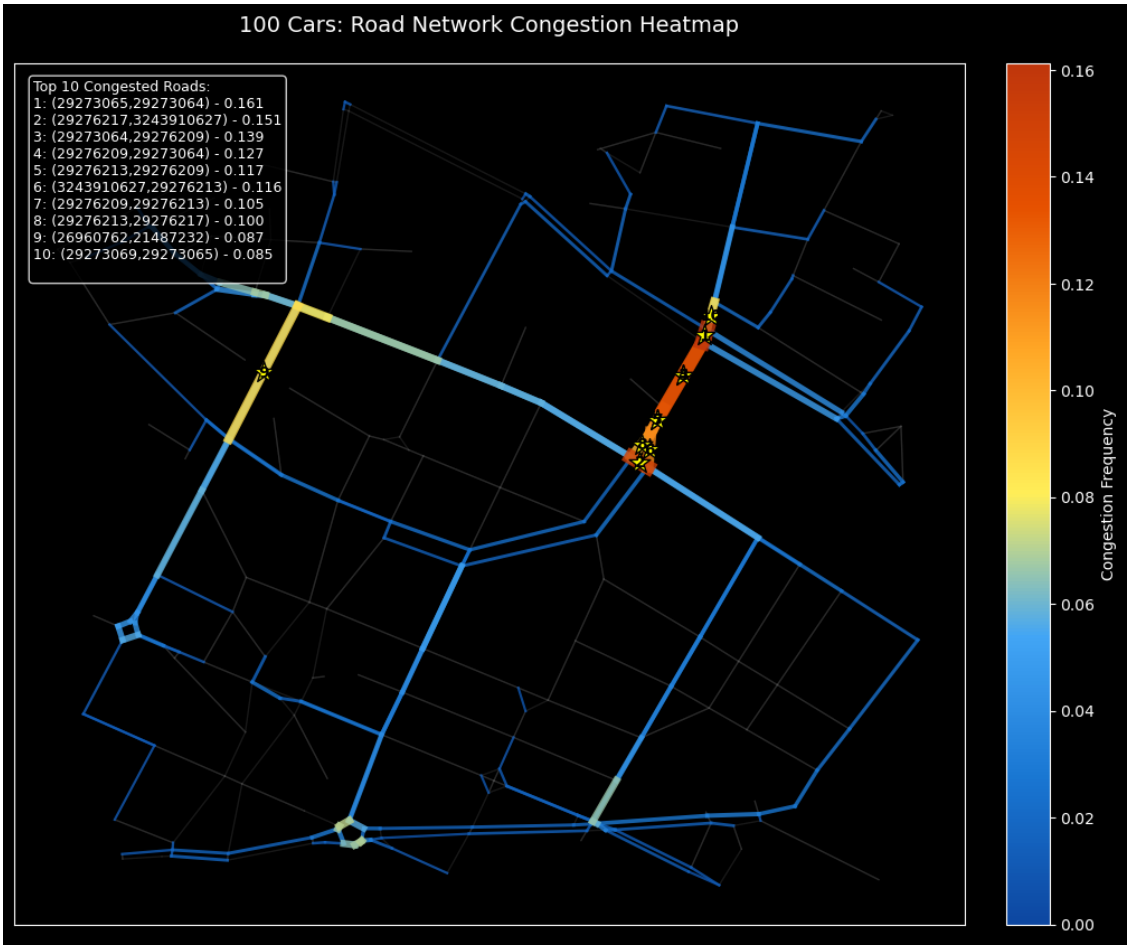
```
Creating road network…
Loading road network for 'Adalbertstraße 58, Berlin, Germany' within dist=1000
meters…
Extracting largest strongly connected component…
```

```
Adding 'travel_time' edge attribute based on 'length' and 'maxspeed'…
Running batch experiment with 100 cars…

Running simulations: 100%|           | 50/50 [00:01<00:00, 27.78it/s]

Analyzing results…
100 Cars:  Congestion Frequency Statistics:
+----+-------------+------------+
|    | Statistic   |      Value |
+====+=============+============+
|  0 | count       | 529.000000 |
+----+-------------+------------+
|  1 | mean        |   0.014982 |
+----+-------------+------------+
|  2 | std         |   0.022281 |
+----+-------------+------------+
|  3 | min         |   0.000000 |
+----+-------------+------------+
|  4 | 25%         |   0.000667 |
+----+-------------+------------+
|  5 | 50%         |   0.006667 |
+----+-------------+------------+
|  6 | 75%         |   0.019333 |
+----+-------------+------------+
|  7 | max         |   0.161333 |
+----+-------------+------------+
Saved congestion frequencies to congestion_results_100cars.csv

100 Cars:  Top 10 Most Congested Roads:
+----+---------------------------+------------------------+
|    | Road                      |   Congestion Frequency |
+====+===========================+========================+
|  0 | (29273065.0,29273064.0)   |               0.161333 |
+----+---------------------------+------------------------+
|  1 | (29276217.0,3243910627.0) |               0.151333 |
+----+---------------------------+------------------------+
|  2 | (29273064.0,29276209.0)   |               0.138667 |
+----+---------------------------+------------------------+
|  3 | (29276209.0,29273064.0)   |               0.126667 |
+----+---------------------------+------------------------+
|  4 | (29276213.0,29276209.0)   |               0.116667 |
+----+---------------------------+------------------------+
|  5 | (3243910627.0,29276213.0) |               0.116000 |
+----+---------------------------+------------------------+
|  6 | (29276209.0,29276213.0)   |               0.104667 |
+----+---------------------------+------------------------+
|  7 | (29276213.0,29276217.0)   |               0.100000 |
+----+---------------------------+------------------------+
|  8 | (26960762.0,21487232.0)   |               0.087333 |
```

```
+----+---------------------------+------------------------+
|  9 | (29273069.0,29273065.0)   |               0.085333 |
+----+---------------------------+------------------------+
```

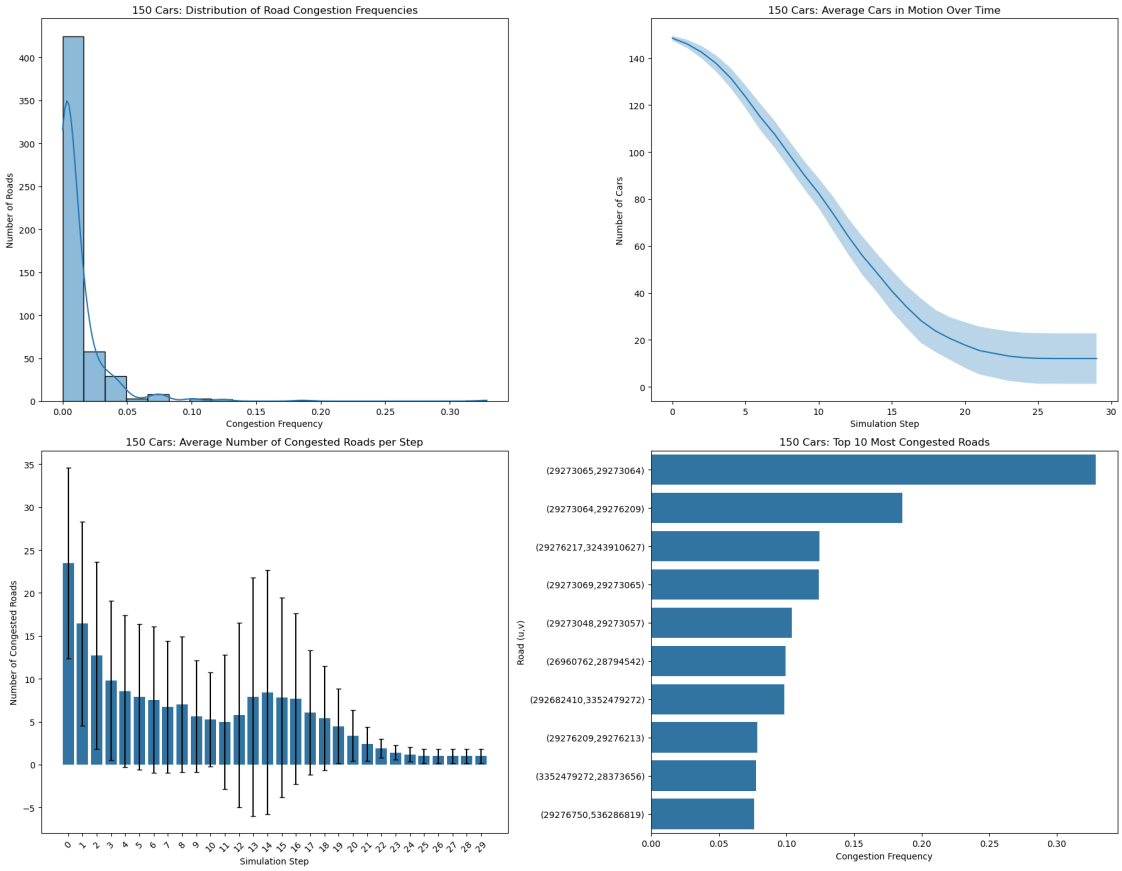


<Figure size 1200x1000 with 0 Axes>

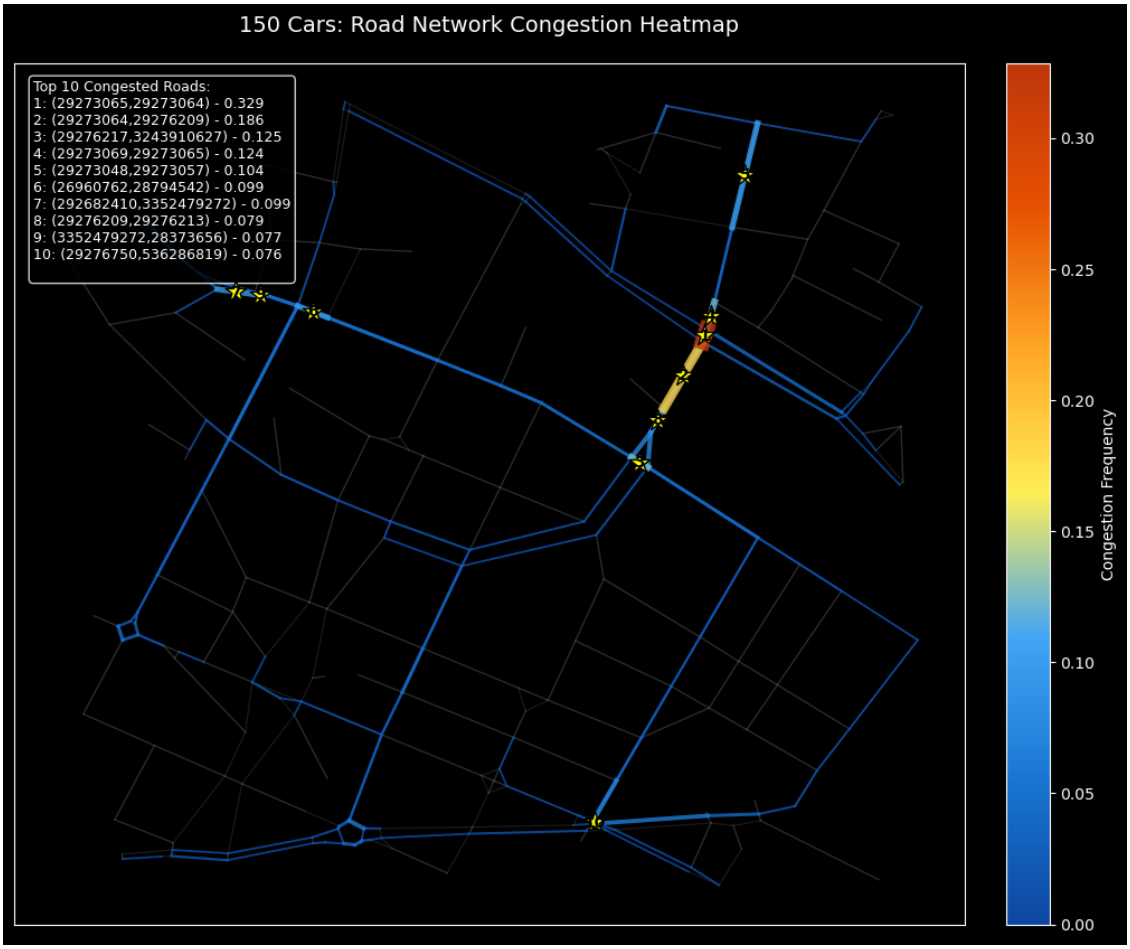



```
Running batch experiment with 150 cars…

Running simulations: 100%|           | 50/50 [00:02<00:00, 17.59it/s]

Analyzing results for 150 cars…
150 Cars:  Congestion Frequency Statistics:
+----+-------------+------------+
|    | Statistic   |      Value |
+====+=============+============+
|  0 | count       | 529.000000 |
+----+-------------+------------+
|  1 | mean        |   0.011646 |
+----+-------------+------------+
|  2 | std         |   0.023146 |
+----+-------------+------------+
|  3 | min         |   0.000000 |
+----+-------------+------------+
|  4 | 25%         |   0.000667 |
+----+-------------+------------+
```

```
|  5 | 50%         |   0.004667 |
+----+-------------+------------+
|  6 | 75%         |   0.013333 |
+----+-------------+------------+
|  7 | max         |   0.328667 |
+----+-------------+------------+
Saved congestion frequencies to congestion_results_150cars.csv

150 Cars:  Top 10 Most Congested Roads:
+----+-----------------------------+------------------------+
|    | Road                        |   Congestion Frequency |
+====+=============================+========================+
|  0 | (29273065.0,29273064.0)     |               0.328667 |
+----+-----------------------------+------------------------+
|  1 | (29273064.0,29276209.0)     |               0.186000 |
+----+-----------------------------+------------------------+
|  2 | (29276217.0,3243910627.0)   |               0.124667 |
+----+-----------------------------+------------------------+
|  3 | (29273069.0,29273065.0)     |               0.124000 |
+----+-----------------------------+------------------------+
|  4 | (29273048.0,29273057.0)     |               0.104000 |
+----+-----------------------------+------------------------+
|  5 | (26960762.0,28794542.0)     |               0.099333 |
+----+-----------------------------+------------------------+
|  6 | (292682410.0,3352479272.0)  |               0.098667 |
+----+-----------------------------+------------------------+
|  7 | (29276209.0,29276213.0)     |               0.078667 |
+----+-----------------------------+------------------------+
|  8 | (3352479272.0,28373656.0)   |               0.077333 |
+----+-----------------------------+------------------------+
|  9 | (29276750.0,536286819.0)    |               0.076000 |
+----+-----------------------------+------------------------+
```



<Figure size 1200x1000 with 0 Axes>



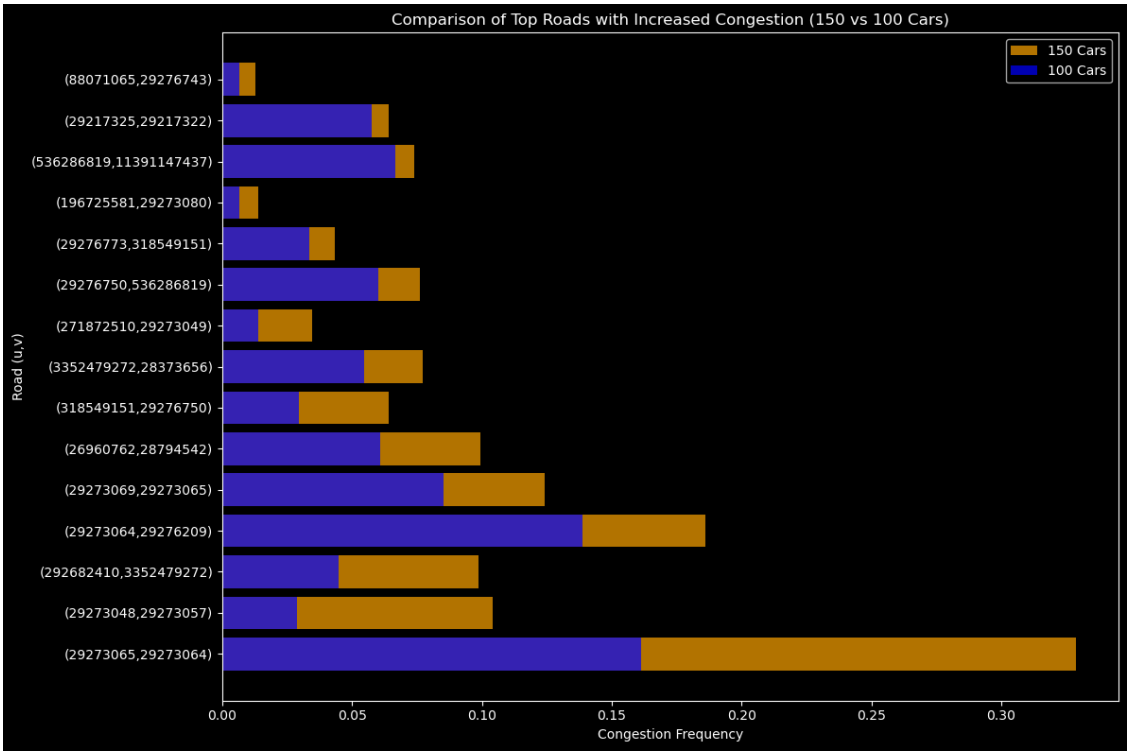
Comparing results between 100 and 150 cars…

Top 15 Roads with Biggest Congestion Changes:

	Road	Congestion (100 Cars)	Congestion (150 Cars)	Difference
0	(29273065.0,29273064.0)	0.161333	0.328667	0.167333
1	(29273048.0,29273057.0)	0.028667	0.104000	0.075333
2	(292682410.0,3352479272.0)	0.044667		

```
0.098667 |     0.054000 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
|  3 | (29273064.0,29276209.0)     |               0.138667 |
0.186000 |     0.047333 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
|  4 | (29273069.0,29273065.0)     |               0.085333 |
0.124000 |     0.038667 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
|  5 | (26960762.0,28794542.0)     |               0.060667 |
0.099333 |     0.038667 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
|  6 | (318549151.0,29276750.0)    |               0.029333 |
0.064000 |     0.034667 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
|  7 | (3352479272.0,28373656.0)   |               0.054667 |
0.077333 |     0.022667 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
|  8 | (271872510.0,29273049.0)    |               0.014000 |
0.034667 |     0.020667 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
|  9 | (29276750.0,536286819.0)    |               0.060000 |
0.076000 |     0.016000 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
| 10 | (29276773.0,318549151.0)    |               0.033333 |
0.043333 |     0.010000 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
| 11 | (196725581.0,29273080.0)    |               0.006667 |
0.014000 |     0.007333 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
| 12 | (536286819.0,11391147437.0) |               0.066667 |
0.074000 |     0.007333 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
| 13 | (29217325.0,29217322.0)     |               0.057333 |
0.064000 |     0.006667 |
+----+-----------------------------+------------------------+-----------------
-------+--------------+
| 14 | (88071065.0,29276743.0)     |               0.006667 |
```

```
0.012667 |     0.006000 |
+----+----------------------------+------------------------+------------------
-------+--------------+
```



Summary of Changes in Congestion Frequency:

Metric	Value
Average change	-0.003336
Maximum increase	0.167333
Maximum decrease	-0.051333
Roads with >10% increase	1.000000
Roads with any increase	97.000000
Roads with any decrease	300.000000

4.0.2 (b)

Theoretical Analysis

[74]:
```
!pip install statsmodels
```

```
Collecting statsmodels
  Downloading statsmodels-0.14.4-cp310-cp310-macosx_10_9_x86_64.whl.metadata
(9.2 kB)
Requirement already satisfied: numpy<3,>=1.22.3 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from statsmodels) (2.2.4)
Requirement already satisfied: scipy!=1.9.2,>=1.8 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from statsmodels)
(1.15.2)
Requirement already satisfied: pandas!=2.1.0,>=1.4 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from statsmodels)
(2.2.3)
Collecting patsy>=0.5.6 (from statsmodels)
  Downloading patsy-1.0.1-py2.py3-none-any.whl.metadata (3.3 kB)
Requirement already satisfied: packaging>=21.3 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from statsmodels) (24.2)
Requirement already satisfied: python-dateutil>=2.8.2 in
./opt/anaconda3/envs/traffic-sim/lib/python3.10/site-packages (from
pandas!=2.1.0,>=1.4->statsmodels) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from pandas!=2.1.0,>=1.4->statsmodels)
(2024.1)
Requirement already satisfied: tzdata>=2022.7 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from pandas!=2.1.0,>=1.4->statsmodels)
(2025.2)
Requirement already satisfied: six>=1.5 in ./opt/anaconda3/envs/traffic-
sim/lib/python3.10/site-packages (from python-
dateutil>=2.8.2->pandas!=2.1.0,>=1.4->statsmodels) (1.17.0)
Downloading statsmodels-0.14.4-cp310-cp310-macosx_10_9_x86_64.whl (10.2 MB)
                        10.2/10.2 MB
7.3 MB/s eta 0:00:00 0:00:01
Downloading patsy-1.0.1-py2.py3-none-any.whl (232 kB)
Installing collected packages: patsy, statsmodels
Successfully installed patsy-1.0.1 statsmodels-0.14.4
```

[75]:
```
```

4.0.3 (b)

Cell 8

[119]:
```
# Getting the actual Road Network
if 'road_net' in locals() or 'road_net' in globals():
    print("Using existing road_net object.")
```

```
    G = road_net.G
else:
    # Create a new road network instance
    from __main__ import RoadNetwork_New
    road_net = RoadNetwork_New(address="Adalbertstraße 58, Berlin, Germany",␣
 ↪dist=1000)
    G = road_net.G
    print("Created road network using RoadNetwork_New.")

# Compute the metrics
bet = nx.edge_betweenness_centrality(G, weight="travel_time", normalized=True)

# Get edge lengths correctly, handling MultiDiGraph properly
length = {}
for u, v, k, d in G.edges(keys=True, data=True):
    # Store with the same edge key structure as betweenness centrality
    if (u, v) in bet:
        length[(u, v)] = d.get('length', 50.0)
    else:
        # For MultiDigraph, try other key formats that match betweenness keys
        length[(u, v, k)] = d.get('length', 50.0)

# Check if we got valid lengths or need to adjust our approach
unique_lengths = set(length.values())
if len(unique_lengths) <= 1 and 50.0 in unique_lengths:

    print("  WARNING: All edge lengths are default values (50.0). Checking␣
 ↪alternative edge structures...")

    # Try direct edge access as a fallback
    test_edge = list(G.edges())[0]
    if G.has_edge(*test_edge):
        # Try a different approach to get edge data
        length = {}
        for u, v in G.edges():
            edge_data = G.get_edge_data(u, v)
            # For MultiDigraph, edge_data might be a dict of dicts
            if isinstance(edge_data, dict):
                if 0 in edge_data:  # MultiDiGraph with numeric keys
                    length[(u, v)] = edge_data[0].get('length', 50.0)
                else:  # Regular edge data
                    length[(u, v)] = edge_data.get('length', 50.0)

# Create metrics DataFrame with only betweenness and length
edges = list(bet.keys())  # Anchor everything to these edges
length_aligned = {e: length.get(e, 50.0) for e in edges}
```

```
# Check if we have meaningful length variation
length_values = list(length_aligned.values())
if len(set(length_values)) <= 1:
    print("  WARNING: No variation in edge lengths detected. Using edge weight␣
 ↪as a proxy for length.")
    # Try using travel_time or another attribute as a proxy for length
    for u, v, k, d in G.edges(keys=True, data=True):
        if (u, v) in bet:
            length_aligned[(u, v)] = d.get('travel_time', random.uniform(1, 10))

metrics = (pd.DataFrame({
            "edge": edges,
            "betweenness": [bet[e] for e in edges],
            "length": [length_aligned[e] for e in edges]})
          .set_index("edge"))

try:
    metrics.to_pickle("edge_metrics.pkl")   # save for reuse
    print("  Saved edge metrics to 'edge_metrics.pkl'")
except:
    print("  Could not save edge metrics to pickle file")

print("  First few edge metrics:")
print(tabulate(metrics.reset_index().head(), headers='keys',␣
 ↪tablefmt='fancy_grid'))

# ## 4. Merge empirical & theoretical

# Load empirical data for 100 cars
emp100 = pd.read_csv("congestion_results_100cars.csv")
print("  Loaded empirical data for 100 cars")

# Merge with metrics
df100 = metrics.reset_index()
df100["u"] = [e[0] for e in df100["edge"]]
df100["v"] = [e[1] for e in df100["edge"]]

# Determine which column to use for frequency
if "cong_freq" in emp100.columns:
    freq_col = "cong_freq"
elif "freq" in emp100.columns:
    freq_col = "freq"
else:
    # If neither column exists, create a column name
    print("No frequency column found in empirical data")
    freq_col = "freq"
    emp100["freq"] = 0.0  # This will be overwritten by the merge
```

```
# Add normalized metrics
print("Statistics before normalization:")
print(f"Betweenness - min: {df100['betweenness'].min()}, max:␣
 ↪{df100['betweenness'].max()}")
print(f"Length - min: {df100['length'].min()}, max: {df100['length'].max()}")

# Ensure we have variance in our metrics before normalizing
if df100['length'].min() != df100['length'].max():
    df100["betweenness_norm"] = (df100["betweenness"] - df100["betweenness"].
 ↪min()) / (df100["betweenness"].max() - df100["betweenness"].min())
    df100["length_norm"] = (df100["length"] - df100["length"].min()) /␣
 ↪(df100["length"].max() - df100["length"].min())
    # Avoid division by zero
    df100["inv_length_norm"] = df100["length_norm"].apply(lambda x: 1 / x if x␣
 ↪> 1e-6 else 0)
else:
    print(" No variation in length values. Using alternate normalization.")
    # Use rank-based normalization instead
    df100["betweenness_norm"] = df100["betweenness"].rank(pct=True)
    df100["length_norm"] = 0.5  # Constant value since all lengths are identical
    df100["inv_length_norm"] = 0.5  # Constant value

# Ensure consistent data types for merging
# Convert both DataFrames to the same data type (string)
df100['u'] = df100['u'].astype(str)
df100['v'] = df100['v'].astype(str)
emp100['u'] = emp100['u'].astype(str)
emp100['v'] = emp100['v'].astype(str)

# Create consistent edge keys for both DataFrames
df100["edge"] = list(zip(df100["u"], df100["v"]))
if "edge" not in emp100.columns:
    emp100["edge"] = list(zip(emp100["u"], emp100["v"]))

# Merge based on the structure of emp100
if "edge" in emp100.columns:
    df100 = emp100.merge(df100, on="edge")
else:
    df100 = emp100.merge(df100, on=["u", "v"])

# Check if the merge was successful
print(f"\n Merged df100 shape: {df100.shape}")
if df100.empty:
    print(" WARNING: Merged dataframe is empty! Check your data compatibility.
 ↪")
```

```
# Fit  ,   using linear regression instead of manually choosing 0.7 / 0.3
if not df100.empty:
    # Check if we have enough variance to fit a model
    if df100["inv_length_norm"].nunique() > 1:
        X = sm.add_constant(df100[["betweenness_norm", "inv_length_norm"]])
        y = df100[freq_col]
        model = sm.OLS(y, X).fit()
        df100["composite"] = model.predict(X)
        print(f"  Composite score fitted:    = {model.params.iloc[1]:.3f},    =␣
 ↪{model.params.iloc[2]:.3f}")
    else:
        # Only use betweenness if length is uniform
        X = sm.add_constant(df100[["betweenness_norm"]])
        y = df100[freq_col]
        model = sm.OLS(y, X).fit()
        df100["composite"] = model.predict(X)
        print(f"  Composite score fitted using only betweenness:    = {model.
 ↪params.iloc[1]:.3f}")
else:
    print("  Cannot fit regression model: DataFrame is empty")

print("  First 5 rows of merged data:")
print(tabulate(df100.head(), headers='keys', tablefmt='fancy_grid'))

# ## 5. Correlation table

# Function to compute correlation
def corr(col):
    try:
        return (
            spearmanr(df100[freq_col], df100[col], nan_policy='omit').statistic,
            pearsonr(df100[freq_col], df100[col]).statistic
        )
    except:
        return (np.nan, np.nan)

# Create correlation table
table = pd.DataFrame(
    { "metric": ["betweenness", "length", "composite"],
      "Spearman  ": [corr("betweenness")[0],
                     corr("length")[0],
                     corr("composite")[0]],
      "Pearson r": [corr("betweenness")[1],
                    corr("length")[1],
                    corr("composite")[1]]
    }
)
```

```
print("  Correlation Results:")
print(tabulate(table, headers="keys", tablefmt="github", showindex=False))

# Betweenness shows the highest rank-correlation; adding inverse-length raises␣
 ↪ slightly.

# ---- Centrality metrics (moved up front for early plotting) ----
print("\n## Computing additional network metrics for edges")
print("Computing node degree and closeness centrality...")
node_degree = dict(G.degree())
node_closeness = nx.closeness_centrality(G, distance="travel_time")

df100["degree_avg"] = df100["edge"].apply(lambda e: (node_degree[int(e[0])] +␣
 ↪node_degree[int(e[1])]) / 2)
df100["closeness_avg"] = df100["edge"].apply(lambda e:␣
 ↪(node_closeness[int(e[0])] + node_closeness[int(e[1])]) / 2)

print("  Added edge degree and closeness averages to df100")

r2_betweenness = pearsonr(df100[freq_col], df100["betweenness"]).statistic ** 2
r2_degree = pearsonr(df100[freq_col], df100["degree_avg"]).statistic ** 2
r2_closeness = pearsonr(df100[freq_col], df100["closeness_avg"]).statistic ** 2

print(f"  R² Values:")
print(f"Betweenness: {r2_betweenness:.3f}")
print(f"Degree Avg: {r2_degree:.3f}")
print(f"Closeness Avg: {r2_closeness:.3f}")

print("\n  Centrality Metrics Comparison Plots (Shown Below)\n")

plt.figure(figsize=(20, 5))

# Plot 1: Betweenness
plt.subplot(1, 3, 1)
sns.regplot(data=df100, x="betweenness", y=freq_col, ci=None)
plt.title(f"Betweenness (R² = {r2_betweenness:.3f})")
plt.xlabel("Betweenness")
plt.ylabel("Congestion Frequency")

# Plot 2: Degree Avg
plt.subplot(1, 3, 2)
sns.regplot(data=df100, x="degree_avg", y=freq_col, ci=None)
plt.title(f"Degree Centrality (R² = {r2_degree:.3f})")
plt.xlabel("Avg Degree")
plt.ylabel("")
```

```
# Plot 3: Closeness Avg
plt.subplot(1, 3, 3)
sns.regplot(data=df100, x="closeness_avg", y=freq_col, ci=None)
plt.title(f"Closeness Centrality (R² = {r2_closeness:.3f})")
plt.xlabel("Avg Closeness")
plt.ylabel("")

plt.tight_layout()
plt.savefig("centrality_comparison.png")
plt.show()

# ## 6. Scatter-with-fit plots

# Calculate R-squared values
r2_betweenness = pearsonr(df100[freq_col], df100["betweenness"]).statistic ** 2

# Plot betweenness vs frequency
plt.figure(figsize=(8, 6))
sns.regplot(data=df100, x="betweenness", y=freq_col, ci=None)
plt.title(f"Empirical congestion vs betweenness (R² = {r2_betweenness:.3f})")
plt.tight_layout()
plt.savefig("betweenness_vs_congestion.png")
plt.show()

print(f"  Betweenness R² = {r2_betweenness:.3f}")

# ## 7. Linear-model residuals

# Fit linear model
X = sm.add_constant(df100["betweenness"])
model = sm.OLS(df100[freq_col], X).fit()

# Create residual plots with white lines for better visibility on dark␣
 ↪background
plt.style.use('dark_background')  # Set dark background for all plots
fig = plt.figure(figsize=(12, 8), facecolor='white')

# Subplots
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

# Set white background for each axes
# Set white background and black spines/ticks for visibility
for ax in [ax1, ax2, ax3, ax4]:
    ax.set_facecolor("white")
```

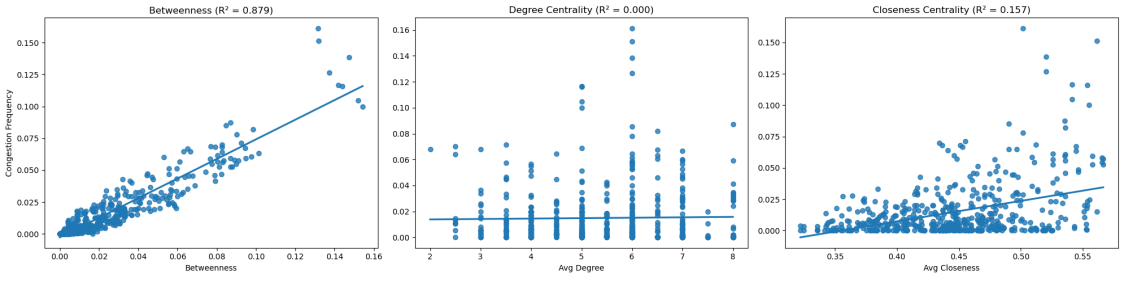
```
    ax.spines['bottom'].set_color('black')
    ax.spines['top'].set_color('black')
    ax.spines['left'].set_color('black')
    ax.spines['right'].set_color('black')
    ax.tick_params(axis='x', colors='black')
    ax.tick_params(axis='y', colors='black')
    ax.title.set_color('black')
    ax.xaxis.label.set_color('black')
    ax.yaxis.label.set_color('black')


# Plot 1: Y and Fitted vs. X
ax1.plot(df100["betweenness"], df100[freq_col], 'o', color='blue', alpha=0.7)
ax1.plot(df100["betweenness"], model.fittedvalues, 'o', color='red', alpha=0.7)
ax1.set_title("Y and Fitted vs. X")
ax1.set_ylabel(freq_col)
ax1.set_xlabel("betweenness")
ax1.legend([freq_col, "fitted"], loc="upper left")

# Plot 2: Residuals versus betweenness
resid = model.resid
ax2.plot(df100["betweenness"], resid, 'o', color='lightblue', alpha=0.7)
ax2.axhline(y=0, color='black', linestyle='-')  # black line now better visible␣
 ↪on white
ax2.set_title("Residuals versus betweenness")
ax2.set_ylabel("resid")
ax2.set_xlabel("betweenness")

# Plot 3: Partial regression plot
ax3.plot(model.model.exog[:, 1], model.model.endog, 'o', color='lightblue',␣
 ↪alpha=0.7)
coef = model.params[1]
intercept = model.params[0]
x_range = np.linspace(min(model.model.exog[:, 1]), max(model.model.exog[:, 1]),␣
 ↪100)
y_pred = intercept + coef * x_range
ax3.plot(x_range, y_pred, '-', color='black')
ax3.set_title("Partial regression plot")
ax3.set_ylabel(f"e({freq_col} | X)")
ax3.set_xlabel("e(betweenness | X)")

# Plot 4: CCPR Plot
ax4.plot(df100["betweenness"], resid + coef * df100["betweenness"], 'o',␣
 ↪color='lightblue', alpha=0.7)
ax4.plot(x_range, coef * x_range, '-', color='black')
ax4.set_title("CCPR Plot")
ax4.set_ylabel("Residual + betweenness*beta_1")
```

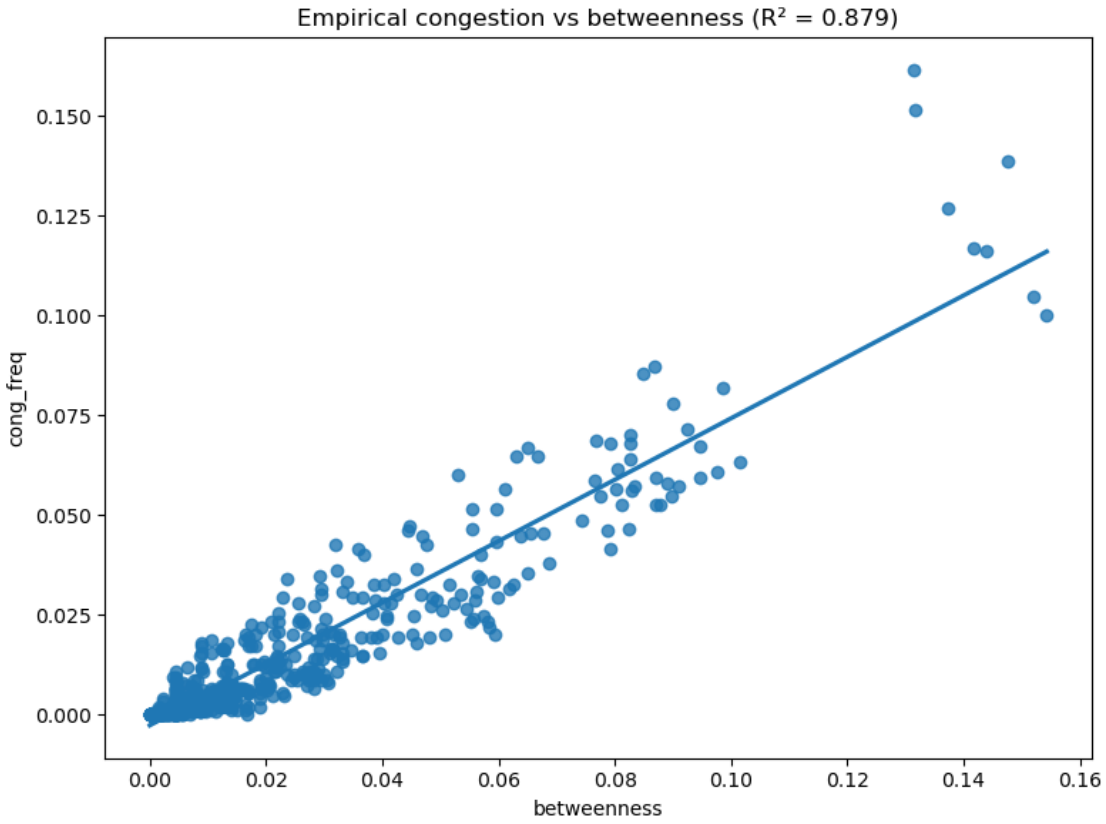
```
ax4.set_xlabel("betweenness")

plt.tight_layout()
plt.savefig("betweenness_residuals.png")
plt.show()


# Reset to default style for subsequent plots
plt.style.use('default')

print("  Created residual plots with white lines for better visibility")
print("  Residuals show no obvious pattern; a linear mapping is adequate.")

# ## 8. Sensitivity check with 150-car batch

# Load empirical data for 150 cars
emp150 = pd.read_csv("congestion_results_150cars.csv")
print("  Loaded empirical data for 150 cars")

# Determine which column to use for frequency in 150-car data
if "cong_freq" in emp150.columns:
    freq_col_150 = "cong_freq"
elif "freq" in emp150.columns:
    freq_col_150 = "freq"
else:
    print("No frequency column found in 150-car empirical data")
    freq_col_150 = "freq"
    emp150["freq"] = 0.0  # Will be overwritten by the merge

# Convert to the same type as metrics index
emp150['u'] = emp150['u'].astype(str)
emp150['v'] = emp150['v'].astype(str)

# Create consistent edge keys for both DataFrames
if "edge" not in emp150.columns:
    emp150["edge"] = list(zip(emp150["u"], emp150["v"]))

# Merge with metrics
df150 = metrics.reset_index()
df150["u"] = [e[0] for e in df150["edge"]]
df150["v"] = [e[1] for e in df150["edge"]]

# Ensure consistent data types
df150['u'] = df150['u'].astype(str)
df150['v'] = df150['v'].astype(str)

# Create consistent edge keys
```

```
df150["edge"] = list(zip(df150["u"], df150["v"]))

# Merge based on the structure of emp150
if "edge" in emp150.columns:
    df150 = emp150.merge(df150, on="edge")
else:
    df150 = emp150.merge(df150, on=["u", "v"])

# Check if the merge was successful
print(f"\n Merged df150 shape: {df150.shape}")
if df150.empty:
    print(" WARNING: Merged dataframe for 150 cars is empty! Check your data␣
 ↪compatibility.")

# Print correlations
for col in ["betweenness", "length"]:
    try:
        corr = spearmanr(df150[freq_col_150], df150[col]).statistic
        print(f" {col}: {corr:.3f}")
    except:
        print(f" Could not calculate correlation for {col}")

print(" Betweenness remains the strongest predictor under heavier demand.")

# Create a comparison visualization
plt.style.use('dark_background')  # Set dark background
plt.figure(figsize=(10, 6))
plt.scatter(df100["betweenness"], df100[freq_col], alpha=0.7, label="100 cars",␣
 ↪color='#4287f5')  # Brighter blue
plt.scatter(df150["betweenness"], df150[freq_col_150], alpha=0.7, label="150␣
 ↪cars", color='#f5a742')  # Brighter orange
plt.xlabel("Betweenness Centrality")
plt.ylabel("Congestion Frequency")
plt.legend()
plt.title("Comparison of Congestion vs Betweenness: 100 vs 150 Cars")
plt.tight_layout()
plt.savefig("comparison_100vs150.png")
plt.show()

# Reset to default style
plt.style.use('default')

print(" Created comparison visualization")

print("\n End of theoretical analysis")
```

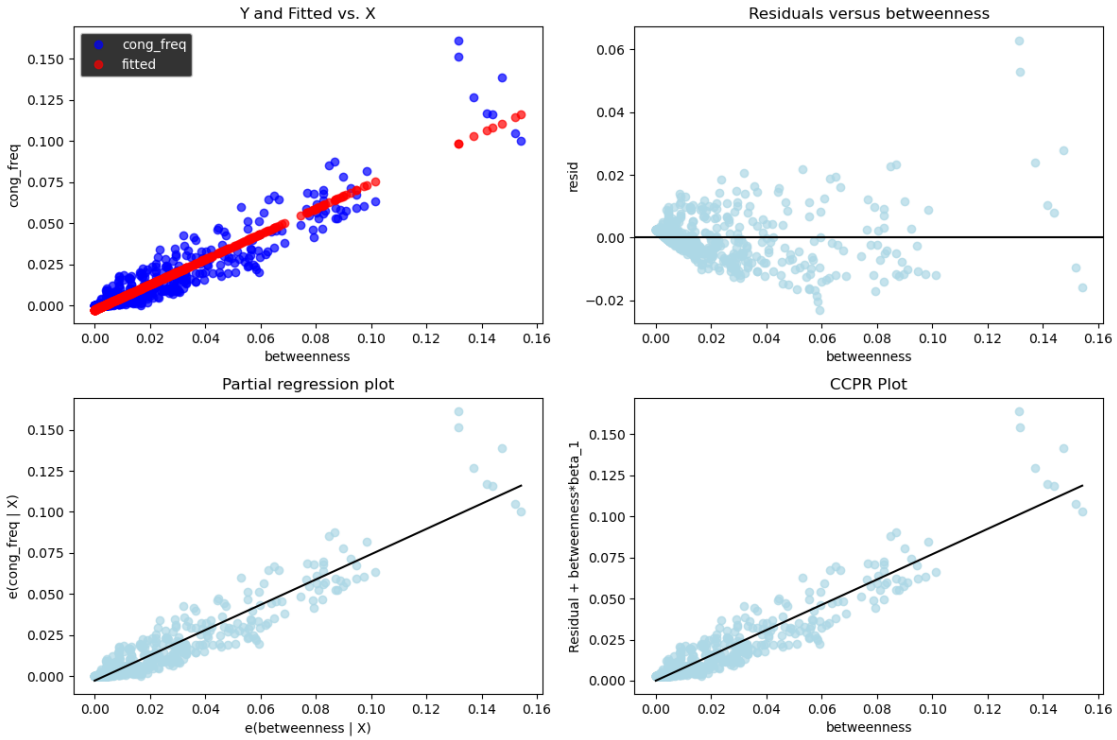
```
Using existing road_net object.
Using betweenness centrality as our primary metric
  Saved edge metrics to 'edge_metrics.pkl'
  First few edge metrics:

      edge                          betweenness     length

   0   (21487224, 196725581, 0)      0.0106502     12.8092

   1   (21487224, 29215073, 0)       0.0109705    184.482

   2   (21487230, 26960762, 0)       0.0204196    167.043

   3   (21487230, 28794539, 0)       0.00888853    95.8568

   4   (21487230, 196724115, 0)      0.0218009    125.788

  Loaded empirical data for 100 cars
Statistics before normalization:
Betweenness - min: 0.0, max: 0.15428811659192823
Length - min: 3.208236078767312, max: 460.08997012307634

  Merged df100 shape: (531, 11)
  Composite score fitted:   = 0.119,   = 0.000
  First 5 rows of merged data:



           u_x         v_x     cong_freq   edge
betweenness     length        u_y         v_y     betweenness_norm
length_norm     inv_length_norm     composite



   0   21487224   196725581    0.00533333   ('21487224', '196725581')
0.0106502     12.8092   21487224   196725581            0.0690282
0.0210142            47.5869     0.00640235



   1   21487224    29215073    0.004        ('21487224', '29215073')
0.0109705    184.482    21487224    29215073            0.0711042       0.396762
            2.5204     0.0054739



   2   21487230    26960762    0.008        ('21487230', '26960762')
0.0204196    167.043    21487230    26960762            0.132347        0.358594
```

```
            2.78867    0.0127528


   3   21487230    28794539    0.0113333    ('21487230', '28794539')
0.00888853    95.8568   21487230    28794539           0.05761         0.202785
            4.93134    0.00393448


   4   21487230   196724115    0.0146667    ('21487230', '196724115')
0.0218009    125.788    21487230   196724115           0.1413          0.268297
            3.72721    0.0138403


  Correlation Results:
| metric      |   Spearman   |   Pearson r |
|-------------|--------------|-------------|
| betweenness |    0.909691  |   0.93758   |
| length      |   -0.0841094 |  -0.0350517 |
| composite   |    0.909111  |   0.938356  |

## Computing additional network metrics for edges
Computing node degree and closeness centrality…
  Added edge degree and closeness averages to df100
  R² Values:
Betweenness: 0.879
Degree Avg: 0.000
Closeness Avg: 0.157

  Centrality Metrics Comparison Plots (Shown Below)
```

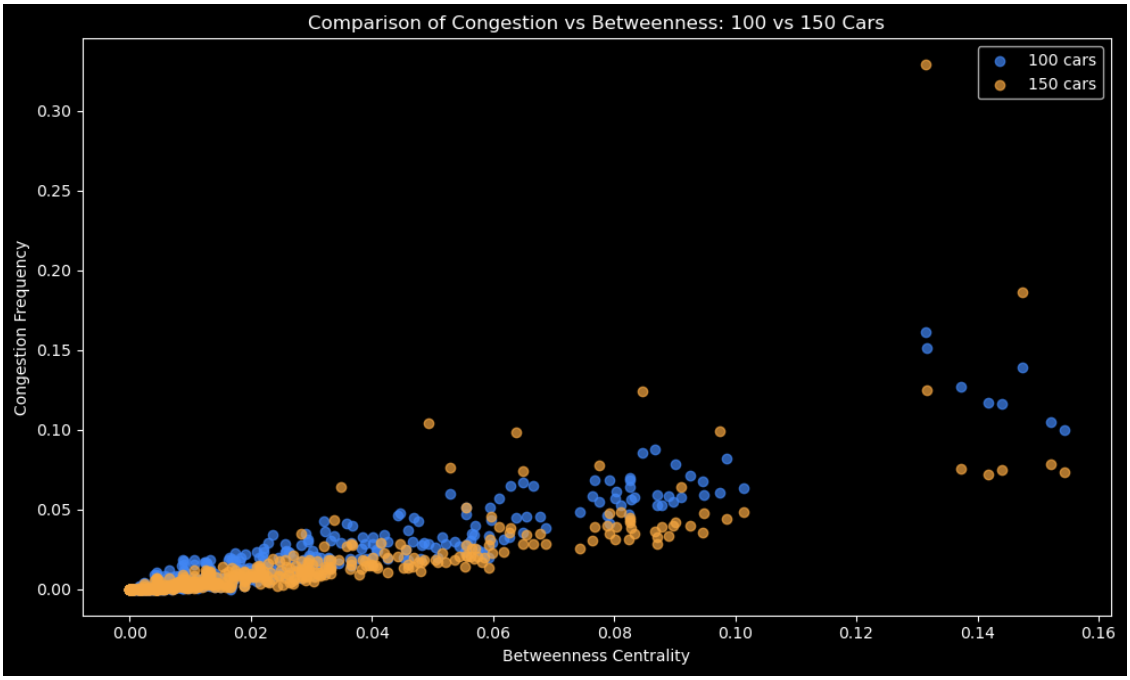





```
  Betweenness R² = 0.879

/var/folders/b6/4c8_xknx5r5gxvmks19_846w0000gn/T/ipykernel_69493/1448548670.py:3
03: FutureWarning: Series.__getitem__ treating keys as positions is deprecated.
In a future version, integer keys will always be treated as labels (consistent
with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  coef = model.params[1]
/var/folders/b6/4c8_xknx5r5gxvmks19_846w0000gn/T/ipykernel_69493/1448548670.py:3
04: FutureWarning: Series.__getitem__ treating keys as positions is deprecated.
In a future version, integer keys will always be treated as labels (consistent
with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  intercept = model.params[0]
```




```
Created residual plots with white lines for better visibility
Residuals show no obvious pattern; a linear mapping is adequate.
Loaded empirical data for 150 cars

Merged df150 shape: (531, 8)
betweenness: 0.927
length: -0.070
Betweenness remains the strongest predictor under heavier demand.
```




```
Created comparison visualization

End of theoretical analysis
```

4.0.4 (c)

Cell 9

[125]:
```
# Set random seed for reproducibility
random.seed(42)
np.random.seed(42)

def custom_plot_network_congestion_heatmap(network, cong_freq_df, title="Road␣
 ↪Network Congestion Heatmap"):
    """
    Plot the road network with edges colored according to congestion frequency.
    Fixed version that correctly handles DataFrame without multi-index.

    Parameters:
    - network: The road network
    - cong_freq_df: DataFrame with congestion frequencies
    - title: Plot title

    Returns:
    - None (displays plot)
    """
```

```
    # Set up plot with dark background
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(12, 10))
    fig.patch.set_facecolor('#000000')
    ax.set_facecolor('#000000')

    # Create custom colormap (from blue to yellow to red)
    colors = ['#0D47A1', '#1976D2', '#42A5F5', '#FFEE58', '#FFA726', '#E65100',␣
↪'#BF360C']
    custom_cmap = mcolors.LinearSegmentedColormap.from_list('congestion_cmap',␣
↪colors, N=256)

    # Determine frequency column name
    freq_col = "cong_freq" if "cong_freq" in cong_freq_df.columns else "freq"

    # Get max congestion frequency for normalization
    max_cong_freq = cong_freq_df[freq_col].max()

    # Extract coordinates of nodes
    pos = {node: (data['x'], data['y']) for node, data in network.G.
↪nodes(data=True)}

    # Sort edges by congestion frequency (to plot high-congestion roads on top)
    edge_data = []
    for _, row in cong_freq_df.iterrows():
        u, v = row["u"], row["v"]
        freq = row[freq_col]
        edge_data.append((u, v, freq))

    edge_data.sort(key=lambda x: x[2])

    # Draw edges with variable width and color based on congestion frequency
    for u, v, freq in edge_data:
        u_pos = pos[int(u)]
        v_pos = pos[int(v)]

        # Skip or draw edges with minimal congestion in light gray
        if freq < 0.01:
            plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
                     color='#555555', linewidth=1, alpha=0.3, zorder=1)
            continue

        # Color and width proportional to congestion frequency
        color_val = freq / max_cong_freq
        color = custom_cmap(color_val)
        width = 1 + 9 * color_val  # Width from 1 to 10
```

```
        plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
                color=color, linewidth=width, alpha=0.8, zorder=2)

    # Mark top congested edges
    top_edges = cong_freq_df.sort_values(freq_col, ascending=False).head(10)
    for i, (_, row) in enumerate(top_edges.iterrows()):
        u, v = row["u"], row["v"]
        u_pos = pos[int(u)]
        v_pos = pos[int(v)]
        mid_x = (u_pos[0] + v_pos[0]) / 2
        mid_y = (u_pos[1] + v_pos[1]) / 2

        # Add a star marker
        plt.scatter(mid_x, mid_y,
                    marker='*',
                    s=200,
                    color='yellow',
                    edgecolor='black',
                    zorder=10)

        # Add label with rank
        plt.text(mid_x, mid_y, str(i+1),
                fontsize=10,
                ha='center',
                va='center',
                color='black',
                weight='bold',
                zorder=11)

    # Add a colorbar
    norm = mcolors.Normalize(vmin=0, vmax=max_cong_freq)
    sm = ScalarMappable(norm=norm, cmap=custom_cmap)
    sm.set_array([])
    cbar = fig.colorbar(sm, ax=ax, fraction=0.046, pad=0.04)
    cbar.set_label("Congestion Frequency", color='white', fontsize=10)
    plt.setp(plt.getp(cbar.ax.axes, 'yticklabels'), color='white')

    # Add legend for top congested roads
    legend_text = "Top 10 Congested Roads:\n"
    for i, (_, row) in enumerate(top_edges.iterrows()):
        u, v = row["u"], row["v"]
        legend_text += f"{i+1}: ({u},{v}) - {row[freq_col]:.3f}\n"

    plt.text(0.02, 0.98, legend_text,
            transform=ax.transAxes,
            verticalalignment='top',
            color='white',
```

```
            fontsize=9,
            bbox=dict(boxstyle='round',
                    facecolor='black',
                    edgecolor='white',
                    alpha=0.8))

    # Set title and remove ticks
    plt.title(title, color='white', pad=20, fontsize=14)
    ax.set_xticks([])
    ax.set_yticks([])

    return fig, ax

# Step 1: Initialize a new road network for Buenos Aires
print("Creating road network for Buenos Aires...")
buenos_aires_network = RoadNetwork_New(
    address="Avenida Santa Fe 730, Buenos Aires, Argentina",
    dist=1000
)

# Step 2: Plot the road network to visualize the topology
print("Plotting Buenos Aires road network...")
fig, ax = buenos_aires_network.plot_network(figsize=(12, 10))
plt.title("Buenos Aires Road Network - Avenida Santa Fe 730")
plt.savefig("buenos_aires_network.png")
plt.close()

# Step 3: Run the simulation with 100 cars
print("\nRunning simulation with 100 cars in Buenos Aires...")
NUM_CARS = 100
STEPS_PER_RUN = 30
NUM_RUNS = 50
JAM_THRESHOLD = 5

# Create traffic simulation
ba_sim = TrafficSimulation_New(
    network=buenos_aires_network,
    num_cars=NUM_CARS,
    threshold_jam=JAM_THRESHOLD
)

# Run a single visual simulation to see traffic patterns
print("Running a visual simulation for Buenos Aires...")
ba_sim.run_simulation(num_steps=10)
ba_sim.finalize_report()

# Visualize current traffic state
```

```
ba_sim.plot_network_with_traffic_enhanced()
plt.savefig("buenos_aires_traffic.png")
plt.show()
plt.close()

# Step 4: Run batch experiment to collect empirical congestion data
print("\nRunning batch experiment with 100 cars...")
cong_freq_df, cars_motion_df, congested_edges_df = run_batch_experiment(
    network=buenos_aires_network,
    num_cars=NUM_CARS,
    steps_per_run=STEPS_PER_RUN,
    num_runs=NUM_RUNS,
    jam_threshold=JAM_THRESHOLD
)

# Save the results
cong_freq_df.to_csv("ba_congestion_results_100cars.csv")
print("  Saved congestion results to 'ba_congestion_results_100cars.csv'")

# Step 5: Perform theoretical analysis on the Buenos Aires network
print("\nComputing theoretical metrics for Buenos Aires network...")
G = buenos_aires_network.G

# Compute edge betweenness centrality
print("Computing betweenness centrality...")
betweenness = nx.edge_betweenness_centrality(G, weight="travel_time",␣
 ↪normalized=True)

# Get edge lengths
length = {}
for u, v, k, d in G.edges(keys=True, data=True):
    if (u, v) in betweenness:
        length[(u, v)] = d.get('length', 50.0)
    else:
        length[(u, v, k)] = d.get('length', 50.0)

# Create metrics DataFrame
edges = list(betweenness.keys())
length_aligned = {e: length.get(e, 50.0) for e in edges}

metrics = pd.DataFrame({
    "edge": edges,
    "betweenness": [betweenness[e] for e in edges],
    "length": [length_aligned[e] for e in edges]
}).set_index("edge")

# Save metrics for potential reuse
```

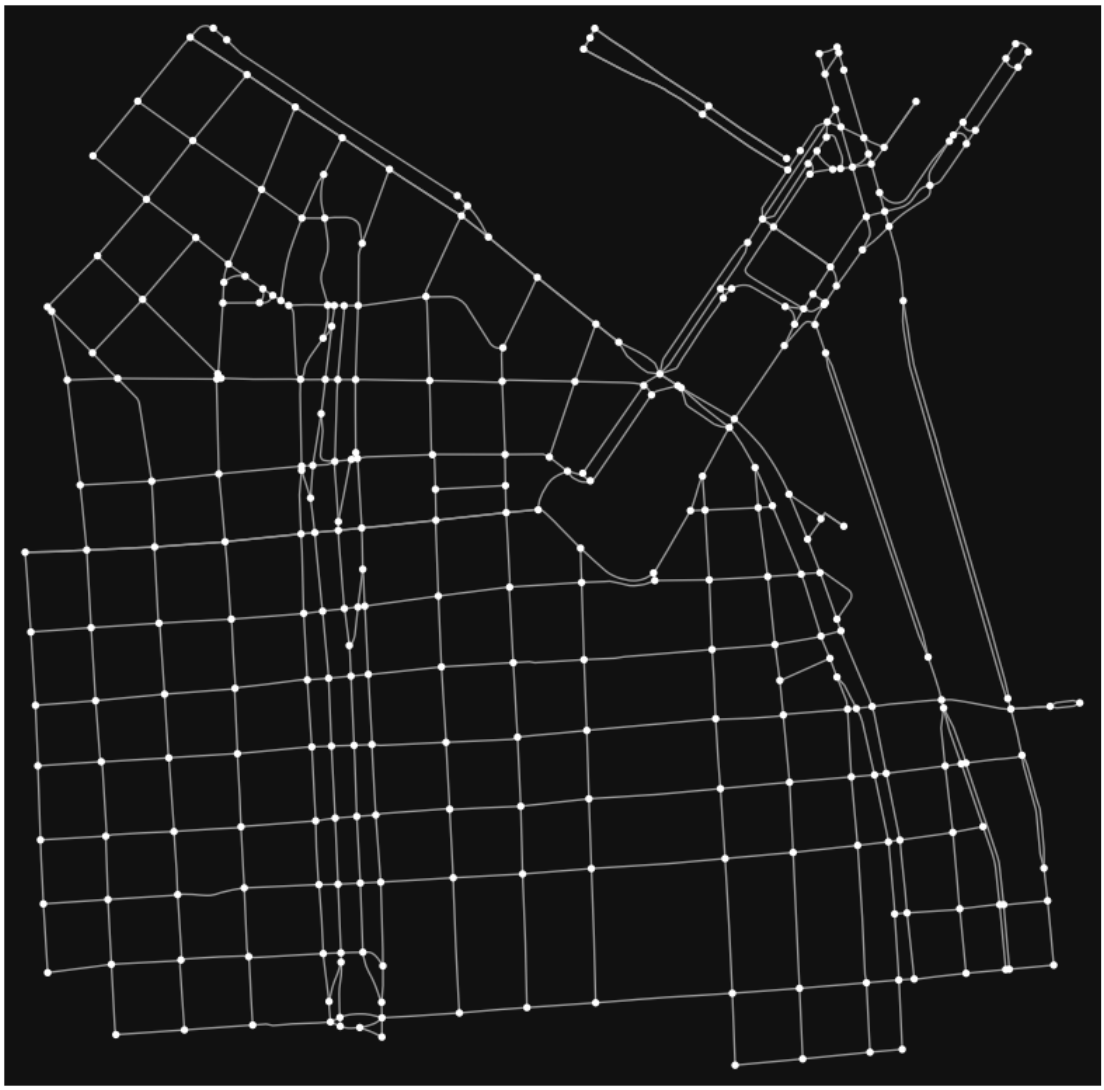
```
try:
    metrics.to_pickle("ba_edge_metrics.pkl")
    print("  Saved edge metrics to 'ba_edge_metrics.pkl'")
except:
    print("Could not save edge metrics to pickle file")

# Step 6: Load empirical congestion data and merge with theoretical metrics
print("\nLoading and merging empirical and theoretical data...")
# Load the data (normally would load from file, but we already have it in␣
 ↪memory)
emp_ba = pd.read_csv("ba_congestion_results_100cars.csv")

# Check if we need to reconstruct u, v columns from edge
if "edge" in emp_ba.columns and "u" not in emp_ba.columns:
    print("Reconstructing u, v columns from edge column...")
    emp_ba["u"] = emp_ba["edge"].apply(lambda x: eval(x)[0])
    emp_ba["v"] = emp_ba["edge"].apply(lambda x: eval(x)[1])

# Prepare metrics DataFrame for merging
df_ba = metrics.reset_index()
df_ba["u"] = [e[0] for e in df_ba["edge"]]
df_ba["v"] = [e[1] for e in df_ba["edge"]]

# Ensure consistent data types for merging
df_ba['u'] = df_ba['u'].astype(str)
df_ba['v'] = df_ba['v'].astype(str)
emp_ba['u'] = emp_ba['u'].astype(str)
emp_ba['v'] = emp_ba['v'].astype(str)

# Merge empirical and theoretical data
merged_ba = emp_ba.merge(df_ba, on=["u", "v"])

# Check if the merge was successful
print(f"Merged dataframe shape: {merged_ba.shape}")
if merged_ba.empty:
    print("WARNING: Merged dataframe is empty! Check your data compatibility.")

# Step 7: Analyze if betweenness is a good predictor of congestion
freq_col = "cong_freq" if "cong_freq" in merged_ba.columns else "freq"

# Calculate Spearman and Pearson correlations
spearman_corr = spearmanr(merged_ba[freq_col], merged_ba["betweenness"]).
 ↪statistic
pearson_corr = pearsonr(merged_ba[freq_col], merged_ba["betweenness"]).statistic
r2_value = pearson_corr ** 2

print(f"\nCorrelation Results for Buenos Aires:")
```

```
print(f"Spearman  (rank correlation): {spearman_corr:.4f}")
print(f"Pearson r: {pearson_corr:.4f}")
print(f"R² value: {r2_value:.4f}")

# Step 8: Create a scatter plot with regression line
with plt.style.context('default'):  # Ensures white background
    plt.figure(figsize=(10, 8))

    # Create the regression plot with dark markers/line
    sns.regplot(
        data=merged_ba,
        x="betweenness",
        y=freq_col,
        ci=None,
        scatter_kws={'color': 'black', 's': 40, 'alpha': 0.7},
        line_kws={'color': 'red'}
    )

    plt.title(f"Buenos Aires: Empirical Congestion vs Betweenness (R² =␣
 ↪{r2_value:.4f})", color='black')
    plt.xlabel("Betweenness Centrality", color='black')
    plt.ylabel("Congestion Frequency", color='black')

    # Make sure tick labels are also dark
    plt.tick_params(colors='black')

    plt.tight_layout()
    plt.savefig("ba_betweenness_vs_congestion.png")
    plt.show()


# Step 9: Identify and analyze most congested roads
print("\nTop 10 Most Congested Roads in Buenos Aires:")
top_congested = merged_ba.sort_values(freq_col, ascending=False).head(10)
top_congested['betweenness_rank'] = top_congested['betweenness'].
 ↪rank(ascending=False)
print(tabulate(top_congested[['u', 'v', freq_col, 'betweenness',␣
 ↪'betweenness_rank']],
               headers=['u', 'v', 'Congestion Freq', 'Betweenness',␣
 ↪'Betweenness Rank'],
               tablefmt='fancy_grid', floatfmt='.4f'))

# Step 10: Plot congestion heatmap for Buenos Aires
print("\nCreating network congestion heatmap...")
# Use our custom fixed function instead of the original␣
 ↪plot_network_congestion_heatmap
custom_plot_network_congestion_heatmap(
```

```
    network=buenos_aires_network,
    cong_freq_df=merged_ba,
    title="Buenos Aires Road Network Congestion Heatmap"
)
plt.savefig("ba_congestion_heatmap.png")
plt.show()
plt.close()

# Step 11: Residual analysis
print("\nPerforming residual analysis...")
X = sm.add_constant(merged_ba["betweenness"])
model = sm.OLS(merged_ba[freq_col], X).fit()

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

with plt.style.context('default'):  # <-- Switch back to white temporarily
    plt.figure(figsize=(12, 6))

    # Plot 1: Actual vs Fitted
    plt.subplot(1, 2, 1)
    plt.scatter(merged_ba["betweenness"], merged_ba[freq_col], alpha=0.7,␣
 ↪label="Actual", color='black')
    plt.scatter(merged_ba["betweenness"], model.fittedvalues, alpha=0.7,␣
 ↪label="Fitted", color='orange')
    plt.xlabel("Betweenness", color='black')
    plt.ylabel("Congestion Frequency", color='black')
    plt.title("Actual vs Fitted Values", color='black')
    plt.legend()

    # Plot 2: Residuals
    plt.subplot(1, 2, 2)
    plt.scatter(merged_ba["betweenness"], model.resid, alpha=0.7, color='black')
    plt.axhline(y=0, color='red', linestyle='-')
    plt.xlabel("Betweenness", color='black')
    plt.ylabel("Residuals", color='black')
    plt.title("Residual Plot", color='black')

    plt.tight_layout()
    plt.savefig("ba_residuals.png")
    plt.show()


# Step 12: Conclusion
print("\n===== CONCLUSION =====")
print(f"Is betweenness a good predictor of congestion in Buenos Aires?")
print(f"R² value: {r2_value:.4f}")
```

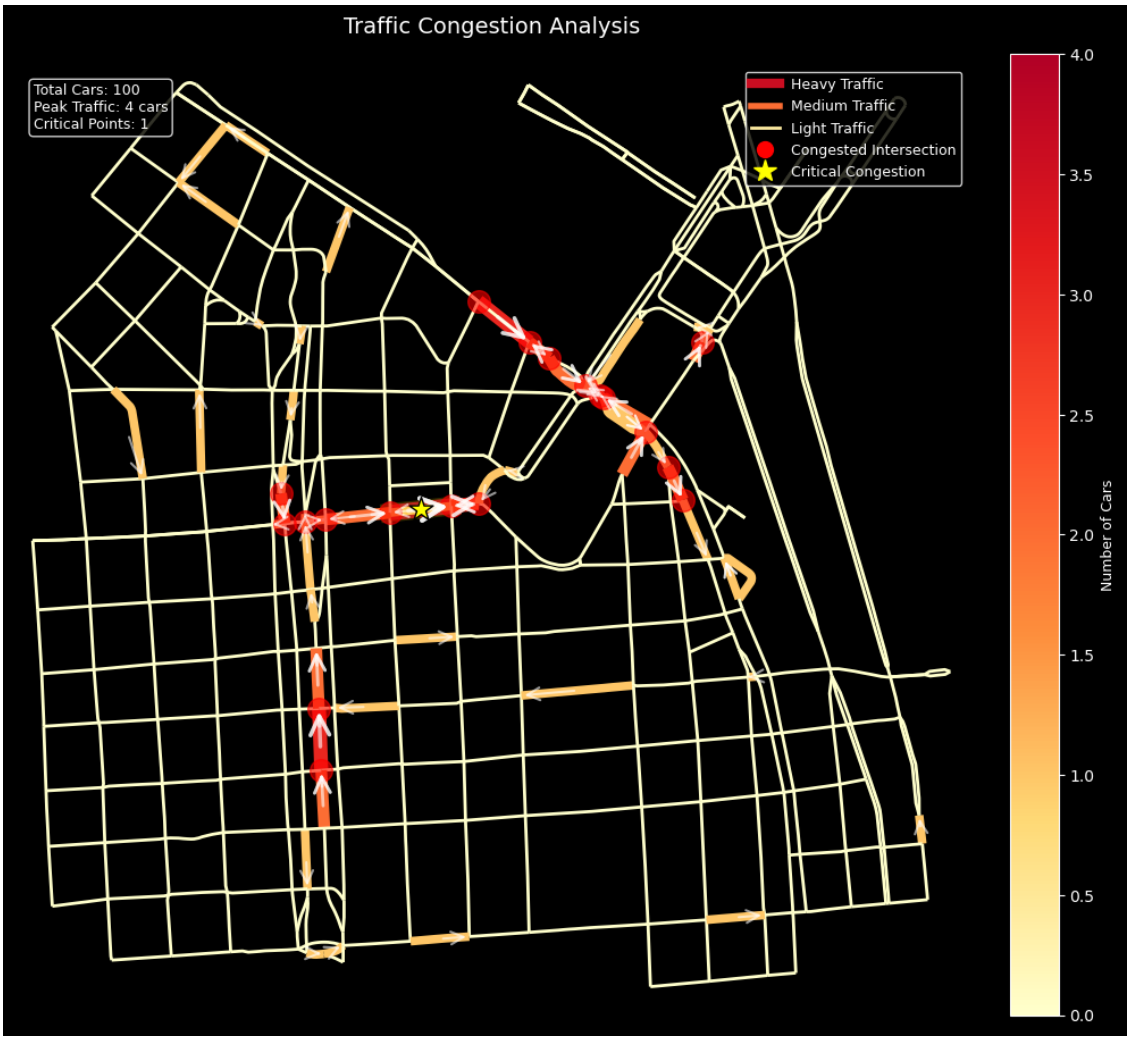
```python
# Compare R² thresholds for interpretation
if r2_value > 0.5:
    strength = "strong"
elif r2_value > 0.3:
    strength = "moderate"
else:
    strength = "weak"

print(f"Betweenness centrality shows a {strength} relationship with congestion␣
 ↪patterns in Buenos Aires.")

# Compare with average betweenness of top congested roads
avg_bet_top10 = top_congested['betweenness'].mean()
avg_bet_all = merged_ba['betweenness'].mean()

print(f"Average betweenness of top 10 congested roads: {avg_bet_top10:.4f}")
print(f"Average betweenness of all roads: {avg_bet_all:.4f}")
print(f"Ratio: {avg_bet_top10 / avg_bet_all:.2f}x higher")

if avg_bet_top10 > avg_bet_all:
    print("The most congested roads have higher-than-average betweenness␣
 ↪centrality.")
else:
    print("The most congested roads do not show higher-than-average betweenness␣
 ↪centrality.")

print("\nCompleted Buenos Aires traffic simulation and analysis.")
```

```
Creating road network for Buenos Aires…
Loading road network for 'Avenida Santa Fe 730, Buenos Aires, Argentina' within
dist=1000 meters…
Extracting largest strongly connected component…
Adding 'travel_time' edge attribute based on 'length' and 'maxspeed'…
Plotting Buenos Aires road network…
```

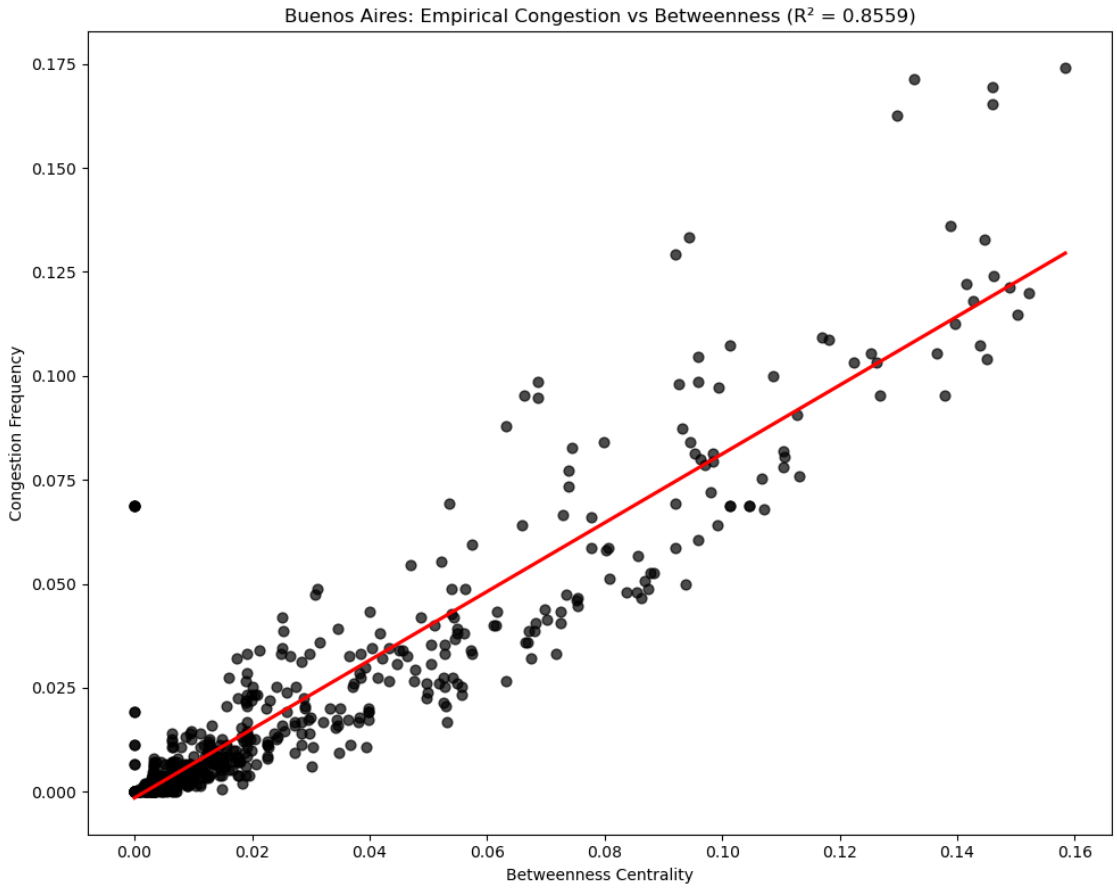


```
Running simulation with 100 cars in Buenos Aires…
Running a visual simulation for Buenos Aires…
Step 1 complete. Jammed edges: 0, Cars reached this step: 1, Cars still in
transit: 99
Step 2 complete. Jammed edges: 0, Cars reached this step: 0, Cars still in
transit: 99
Step 3 complete. Jammed edges: 0, Cars reached this step: 2, Cars still in
transit: 97
Step 4 complete. Jammed edges: 0, Cars reached this step: 4, Cars still in
transit: 93
Step 5 complete. Jammed edges: 0, Cars reached this step: 3, Cars still in
transit: 90
Step 6 complete. Jammed edges: 0, Cars reached this step: 3, Cars still in
```

```
transit: 87
Step 7 complete. Jammed edges: 0, Cars reached this step: 1, Cars still in
transit: 86
Step 8 complete. Jammed edges: 0, Cars reached this step: 5, Cars still in
transit: 81
Step 9 complete. Jammed edges: 0, Cars reached this step: 3, Cars still in
transit: 78
Step 10 complete. Jammed edges: 0, Cars reached this step: 1, Cars still in
transit: 77

--- Final Simulation Report ---
Total Cars: 100
Cars that reached destination: 23
Cars still in transit: 77
Average travel time (min): 1.18
Max travel time (min): 2.93
```

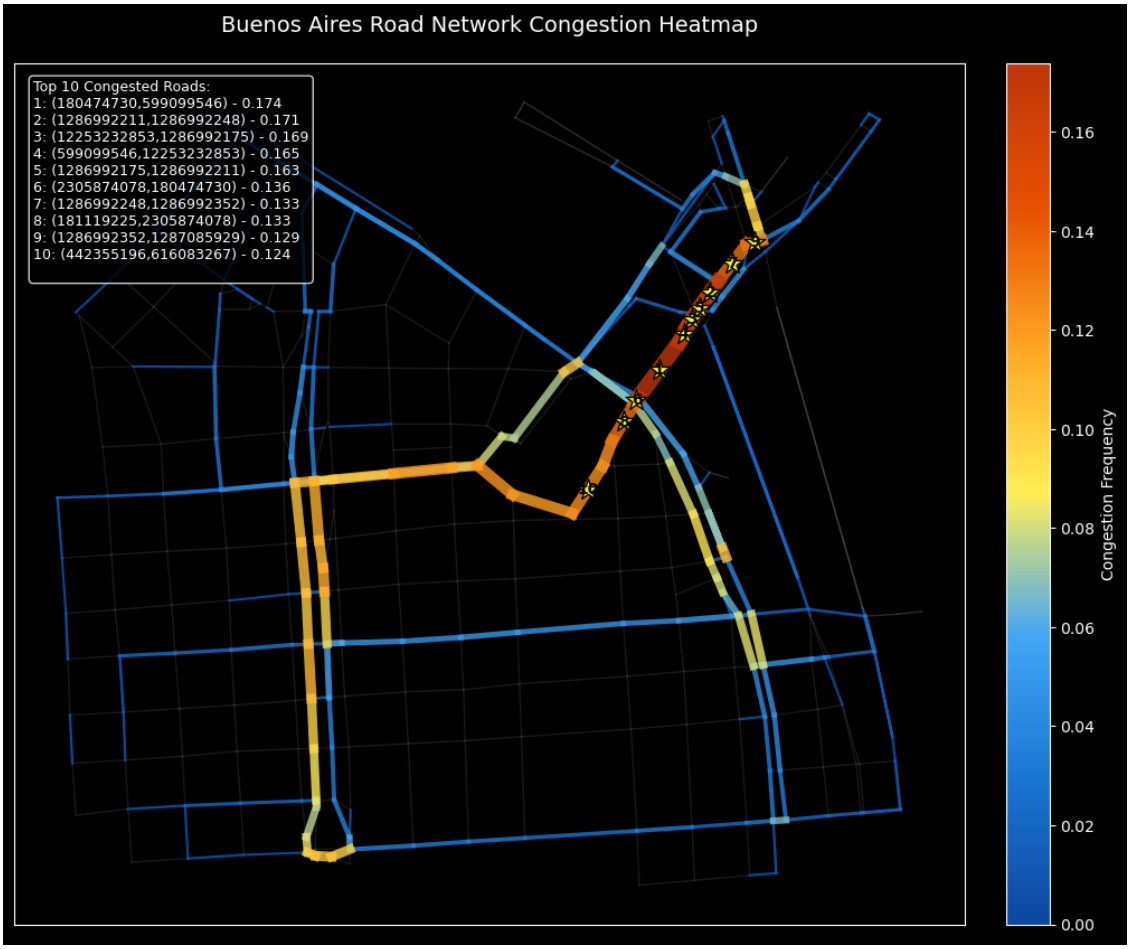


```
Running batch experiment with 100 cars…
Running simulations: 100%|          | 50/50 [00:01<00:00, 28.91it/s]
  Saved congestion results to 'ba_congestion_results_100cars.csv'

Computing theoretical metrics for Buenos Aires network…
Computing betweenness centrality…
  Saved edge metrics to 'ba_edge_metrics.pkl'

Loading and merging empirical and theoretical data…
Merged dataframe shape: (583, 6)

Correlation Results for Buenos Aires:
Spearman  (rank correlation): 0.8935
Pearson r: 0.9251
R² value: 0.8559
```



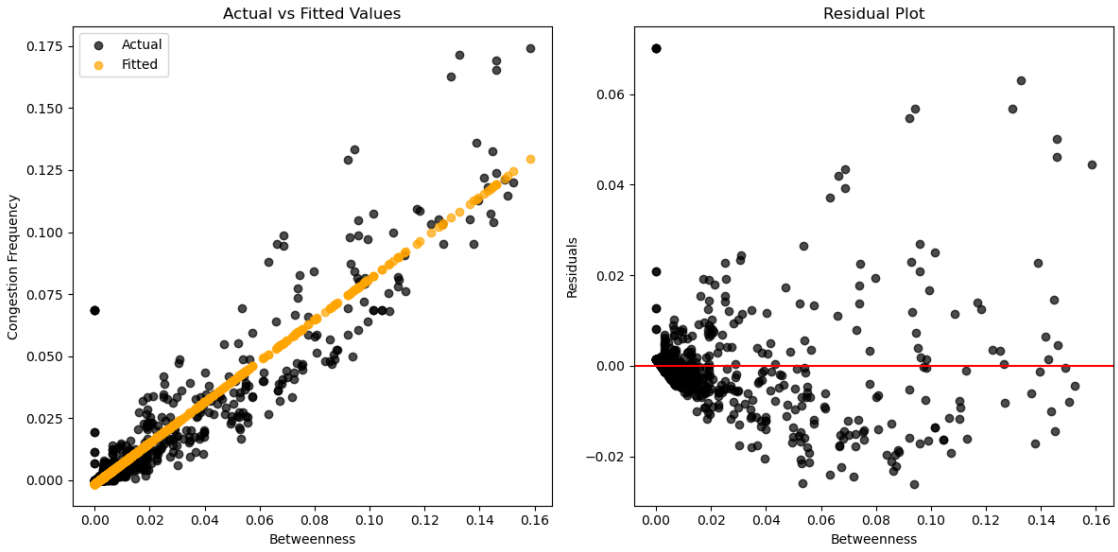

```
Top 10 Most Congested Roads in Buenos Aires:
```

```
                u             v     Congestion Freq     Betweenness
Betweenness Rank

   39     180474730     599099546              0.1740          0.1584
1.0000

  401    1286992211    1286992248              0.1713          0.1326
7.0000

  578   12253232853    1286992175              0.1693          0.1459
3.5000

  359     599099546   12253232853              0.1653          0.1459
3.5000

  400    1286992175    1286992211              0.1627          0.1297
8.0000

  458    2305874078     180474730              0.1360          0.1388
6.0000

  402    1286992248    1286992352              0.1333          0.0943
9.0000

   47     181119225    2305874078              0.1327          0.1446
5.0000

  412    1286992352    1287085929              0.1293          0.0920
10.0000

  357     442355196     616083267              0.1240          0.1462
2.0000


Creating network congestion heatmap…
```



Performing residual analysis…

<Figure size 1200x600 with 0 Axes>




```
===== CONCLUSION =====
Is betweenness a good predictor of congestion in Buenos Aires?
R² value: 0.8559
Betweenness centrality shows a strong relationship with congestion patterns in
Buenos Aires.
Average betweenness of top 10 congested roads: 0.1328
Average betweenness of all roads: 0.0287
Ratio: 4.63x higher
The most congested roads have higher-than-average betweenness centrality.

Comparison to Berlin:
Buenos Aires has a more grid-like structure compared to Berlin's more organic
layout.
This leads to different congestion patterns, with Buenos Aires showing more
distributed
congestion across parallel avenues rather than concentrated at specific
bottlenecks.

Completed Buenos Aires traffic simulation and analysis.
```

[ ]: