LBA: Traffic Simulation

LBA Traffic Simulation

Model

This simulation models urban traffic flow using a road network graph focused on a section of Berlin, Germany. It uses real-world map data from OpenStreetMap, imported via the OSMnx Python library. This graph represents intersections as nodes and road segments as directed edges. The goal is to simulate how cars move through the city, encounter congestion, and respond to traffic conditions.

Code Improvements and Debugging

I fixed several issues while restructuring the original code into an object-oriented programming (OOP) design. Below is a summary of the significant changes and improvements I made:

1. *Global Variable Scope*

   The `cars` variable, initially confined within the `simulate_traffic` function, was inaccessible to other parts of the program, leading to `NameError` exceptions during visualization. I moved the list outside the function, making it visible to the rest of the program.

2. *Network Path Errors*

   The `nx.shortest_path()` function raised a `NetworkXNoPath` error when it couldn't find a valid route between randomly selected nodes (node 29217277 to 29217303). I wrote a `try-except` block to skip cars without valid paths, preventing the simulation from crashing. In simple human language, the model sometimes asked a vehicle to go from point A to point B, but there was no way (which crashed the code). When this happens, the program ignores that car and continues running.

3. *Missing 'travel_time' Attribute*

   The code computed the shortest paths based on a `'travel_time'` attribute, but this attribute wasn't in the graph. I added it manually by computing travel time as: $travel\ time\ = \frac{length\ (m)}{speed\ (km/h) * \frac{1000}{60}}\ (in\ minutes)$. So now the code knows how long it takes to go down each road.

I added logic to handle different speed limit formats (string values like `'30 mph'` and lists of values).

4. *Path Navigation Logic*

   The `move_cars()` function had incorrect path updating logic (aka, moved incorrectly or to places they could not go). I fixed it so cars only move forward if they are not jammed and haven't reached their destination.

5. *Congestion Modeling*

   The initial edge car count logic was flawed due to incorrect keys in a multigraph. I rewrote the edge counting logic to handle multi-edge scenarios correctly. Now, the model correctly counts the number of cars on the road.

6. *Dynamic Routing & Destination Updates*

   When a car reaches its destination, it receives a new random destination and recalculates its path accordingly. A new path is computed if a car's path becomes empty or invalid.

7. *Traffic Visualization Enhancements*

   The visualization logic was updated to:

   - Normalize edge colors correctly based on the actual range of congestion values.
   - Use `matplotlib` colormaps to show congestion intensity.
   - Handle plotting errors gracefully when edge data is missing or inconsistent.

How The Model Worked After OOP and Bugs Fixed (Still Not The Final Version!)

The simulation begins by loading a real-world road network for a specified address in Berlin using OSMnx, incorporating the previously described `travel_time` attribute for realistic routing. Cars are then initialized with random starting and destination nodes within the network. Each vehicle is represented as an object, storing its location, destination, and path. Cars utilize NetworkX's `shortest_path()` to determine the optimal route based on travel time, with paths stored as sequences of nodes.

At each time step:

- Cars attempt to move one node forward along their path.
- If more than five cars attempt to move onto the same edge, they experience a traffic jam and stay in place.

- Cars that reach their destination are given a new random destination and rerouted.

The network is visualized using edge thickness and red color intensity to represent congestion levels. Heavily used roads appear thicker and darker red.

Strengths and Limitations of The Model

This simulation effectively captures certain aspects of real-world urban traffic. Using real road map data from OSMnx ensures accurate street layouts and directions. Incorporating travel times based on road length and speed limits adds realism to the routing process. The dynamic routing capabilities, allowing cars to reroute around congestion, and the emergent traffic jam behavior are also strengths.

However, the model also incorporates several simplifications:

- Cars move in discrete jumps between intersections rather than continuous movement along roads.
- All cars are modeled identically, neglecting variations in driver behavior and vehicle characteristics.
- Speed limits are fixed, ignoring real-world fluctuations due to weather or time-dependent traffic rules.
- Congestion is modeled as a binary state (full speed or stopped) without gradual slowdowns.
- Cars have perfect and instantaneous knowledge of the entire network, enabling immediate rerouting.
- The simulation omits real-world complexities such as traffic lights, stop signs, lane changes, pedestrian interactions, and public transport.

Improvements To The Simulation

To make the simulation more realistic, I made some changes to the model. Instead of saying a road is jammed when it has '5 cars', I now calculate how many cars each road can handle. Big roads like highways can handle more than small streets. The model looks at the number of lanes and the type of road to figure this out.

```
# Calculate road capacity based on road properties
lanes = data.get("lanes", 1)
      if isinstance(lanes, list):
        lanes = lanes[0]
```

```
      Try:
         lanes = float(lanes)
      except (ValueError, TypeError):
         lanes = 1.0

road_type = data.get("highway", "residential")
# Base capacity depends on road type
      if road_type in ["motorway", "trunk", "primary"]:
             base_capacity = 25  # Higher capacity for major roads
      elif road_type in ["secondary", "tertiary"]:
             base_capacity = 15  # Medium capacity
      else:
             base_capacity = 8   # Lower capacity for minor roads
             data["capacity"] = base_capacity * lanes
```

Cars do not just stop anymore. They slow down gradually as traffic gets worse. If a road is a little busy, cars slow down a bit. If it is packed, they almost stop.

```
if random.random() < speed_reduction:
            car.current_location = next_node
            car.path.pop(0)
            any_car_moved = True
```

The simulation now knows about rush hour. During busy times (7-9 AM and 4-6 PM), roads act like they can handle fewer cars. This mimics how real roads get more crowded during those times.

```
time_factor = 1.0
  if 7 <= self.current_hour <= 9 or 16 <= self.current_hour <= 18: # Rush
  time_factor = 0.6
```

Cars can now change their routes to avoid traffic jams. If a car gets stuck in heavy traffic, it will try to find a faster way. This is like how GPS navigation systems work in real life.

```
reroute_probability = 0.05
        if congestion_factor > 0.7:
            reroute_probability = 0.3
```

I also made the core of the simulation more reliable:

- Cars find their routes right away when they start. If a route doesn't work, they get a new one.

- Cars only move once per step, keeping things organized.
- Routes are always recalculated when needed, especially when traffic changes.
- The simulation clock is accurate, so rush hour and other time-based effects work correctly.

Data Visualization

The original traffic visualization (Figure 1) uses red to show congestion but initially lacked a color bar, making it hard to read exact values. To fix this, I added a color bar with an orange gradient (Figure 2). Also, I changed the background to gray for better contrast and made the color labels black for clarity. Now, you can easily see the most congested roads and understand the congestion levels accurately.

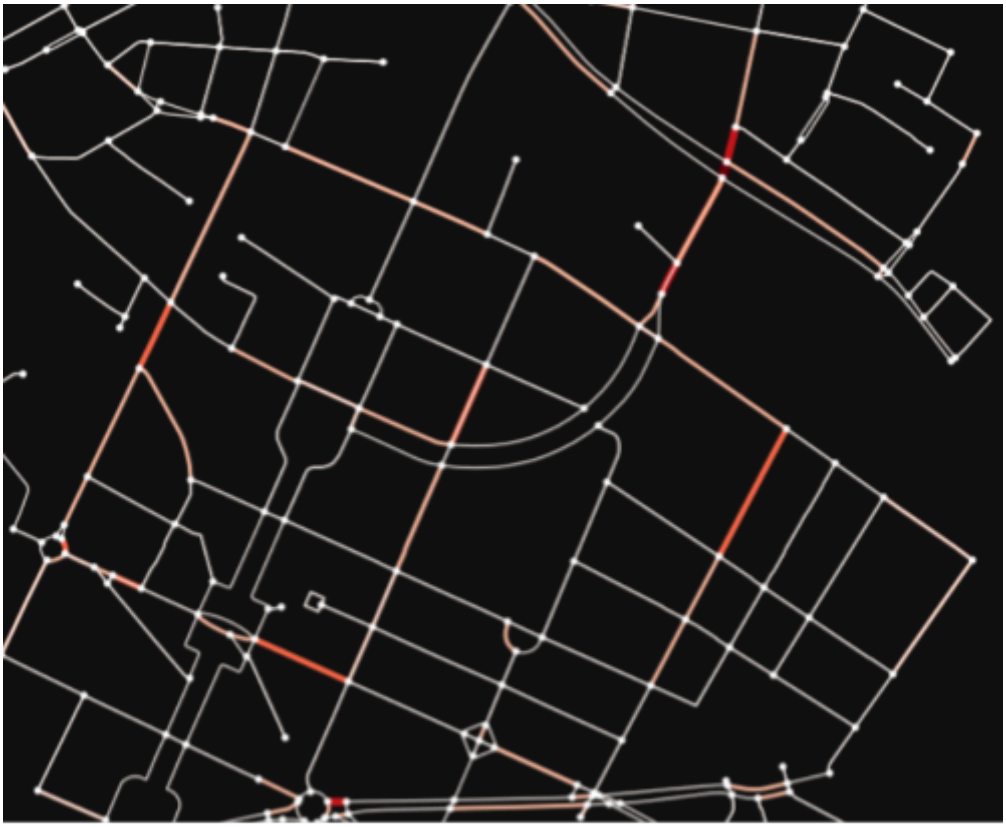

Figure 1. *Snippet of Berlin Traffic Map.* This is a road network map with edges (roads) and nodes (intersections). Roads are colored with a gradient from white to red; the redder the rod is, the more congested it is. It uses a black background for contrast.

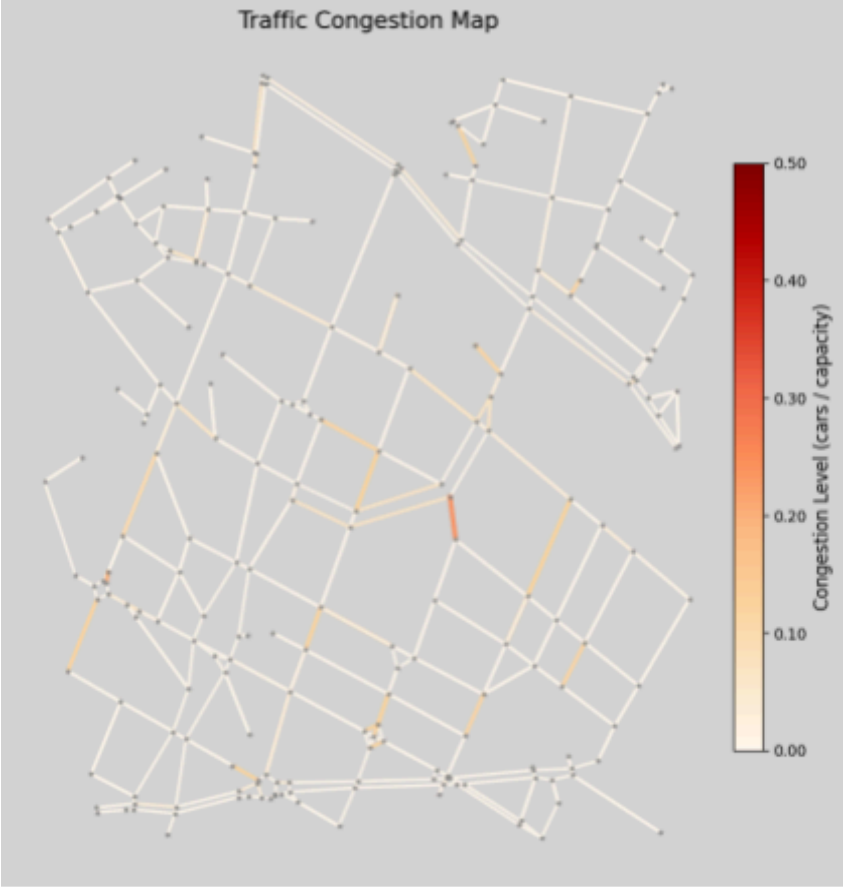

Figure 2. *Berlin Traffic.* It shows roads and intersections at 10 steps, it takes 1 minute per step and starts at 8 a.m. with 100 cars, so this is how congested Berlin is in 10 minutes according to my model.

Empirical Analysis

I ran the traffic simulation 10 times on the `Berlin_network` road network to pinpoint congestion hotspots. Congestion was quantified using the metric `congestion_level = cars_on_edge / capacity`. It provided a normalized measure of road usage. Edge-specific congestion levels were meticulously recorded throughout each simulation run and aggregated into `congestion_data`. Finally, the average congestion level for each edge, `edge_congestion_avg`, was calculated across all 10 simulations to identify consistent areas of traffic stress. A detailed view of these congested edges and their average congestion metrics are provided in the `pprint.pprint(edge_congestion_avg)` output.

I observed that edges such as:

- Edge (29273064, 29273065, 0) and (29276213, 29276217, 0) consistently show high congestion.
- The road segments connected to nodes 29218324 and 29218326 also experience high congestion.
- Edges around nodes 29276687 and 29276750 also have high congestion.

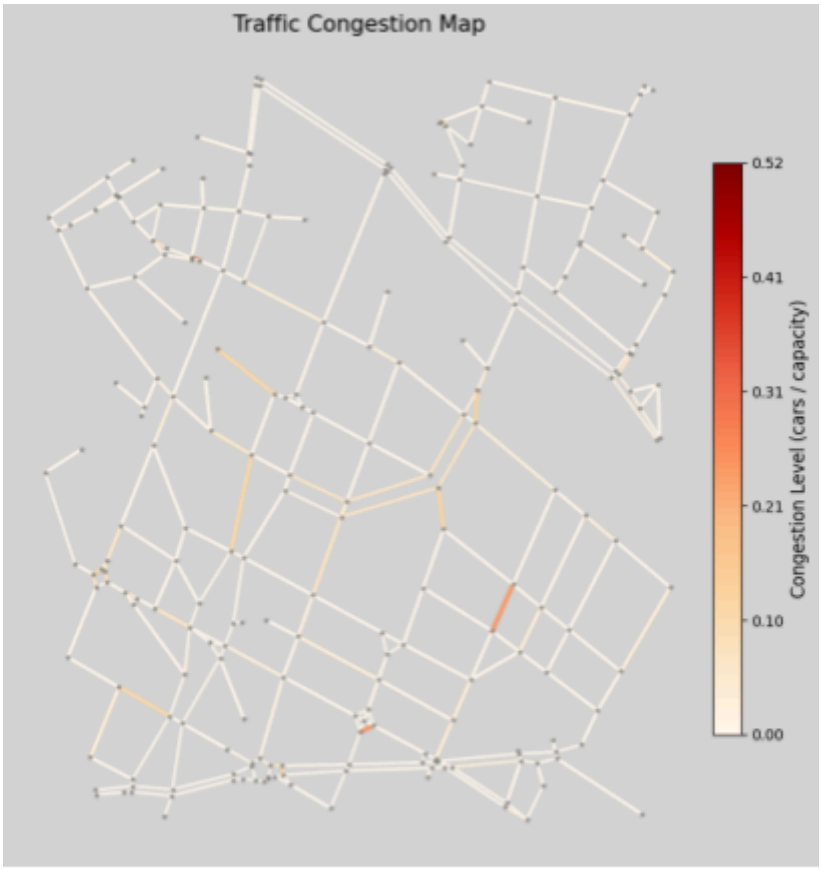

Figure 3. *Berlin Traffic after 10 runs.* As mentioned above, we see the edges (streets) highlighted with the most congestion. It looks realistic with some areas being more jammed than the others.

Theoretical Analysis

To understand the structural underpinnings of the congestion hotspots, I computed betweenness centrality, edge betweenness centrality, closeness centrality, and degree centrality using the `nx library.` For betweenness centrality, results showed a moderate positive correlation with congestion, meaning intersections on many shortest paths tend to experience higher traffic. Practically, this highlights critical intersections for traffic management. Edge

betweenness centrality exhibited a strong positive correlation with congestion, proving to be the most reliable predictor. This implies that roads carrying a large traffic volume are highly susceptible to congestion and should be prioritized for traffic optimization. Closeness centrality showed a weak positive correlation with congestion, indicating that accessibility plays a lesser role than through-traffic volume in predicting congestion. Degree centrality metrics showed no significant correlation with congestion, suggesting that the number of roads meeting at an intersection does not directly predict traffic congestion.

Analysis of 851 California Street

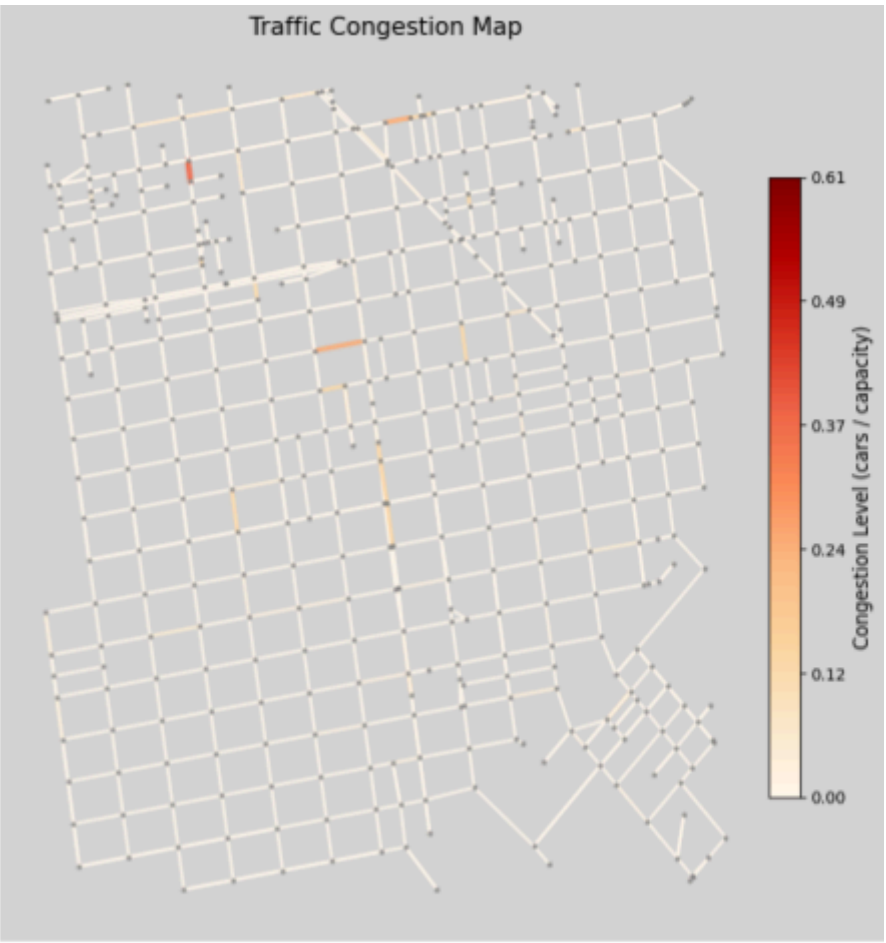

Figure 4. *San Francisco Traffic.* Showing similar results to Berlin with few streets being more congested than others. Ran for 100 times.

Summary

This simulation models Berlin's traffic using real-world road data, representing intersections and roads as a network to simulate car movement and congestion. The model was improved with OOP, addressing errors and adding features like realistic travel times, dynamic routing, and variable congestion. Cars move through the network, slowing down in traffic, rerouting when necessary, and are influenced by simulated rush hour conditions. Congestion is determined by road capacity, and the simulation identifies key congested areas, further analyzed using network metrics to reveal that edge betweenness centrality is a strong traffic indicator.

AI Statement

1. I used Gemini for grammar and to help me with the write-up.
2. I used Claude to help with OOP design; the prompts asked “to break down possible features I could add to the simulation” and “walk me through the flowchart of what needs to come after what”. I only went to Claude for OOP advancements after studying the code to understand its logic and bugs. It helped design the OOP structure and add improvements.
3. I used ChatGPT for suggestions for theoretical analysis (specifically, the metrics it proposed from the nx library).