Minerva University

CS166 — Modeling & Analysis of Complex Systems

Prof. Tambasco (MW @ 1:00 PM UTC)

Traffic Simulation on Road Networks

Learning-by-Application Project

December 6th 2025

Contents

Overview

This project implements a traffic simulation that models vehicle movement through real road networks obtained from OpenStreetMap (**openstreetmap**). The simulation loads a 1-kilometer radius network around our residence hall in Buenos Aires (Esmeralda 920) by default, and it also supports loading any location via coordinates or text addresses.

The simulation operates in discrete time steps. Cars are initialized with random origin-destination pairs, their shortest paths are computed based on travel time, and they advance incrementally through the network. The key mechanism is a density-dependent speed model: as vehicle density increases on a road segment, travel speeds decrease proportionally. This creates a feedback loop where congestion reduces speeds, which in turn increases density.

Vehicles track their exact position in meters along each road segment rather than jumping between intersections. When a vehicle reaches the end of one segment, it transitions to the next edge in its route. This representation enables realistic patterns to emerge: bottlenecks form where routes converge, and certain roads experience persistent congestion while others remain underutilized.

The model incorporates several simplifications—no traffic signals, lane changes, or driver heterogeneity. Nevertheless, it captures the basic relationship between density and speed: as more cars occupy a road, speeds decrease.

1 Task 1: Implementation

1.1 Fixing ChatGPT's Bugs

The original problem. The initial code was generated by ChatGPT for a traffic simulation on a Berlin road network. While the code appeared structurally sound—containing functions to load graphs, simulate vehicle movement, and visualize traffic—execution revealed several critical flaws.

The visualization function contained an error. The code attempted to color edges by vehicle count but used `data['osmid']` as the lookup key. This is incorrect because `osmid` identifies the real-world road segment in OpenStreetMap, whereas the `count` dictionary is indexed by the tuple (u, v, k), which uniquely identifies an edge in the MultiDiGraph. In a directed multigraph, one physical road may correspond to multiple edges (northbound versus southbound lanes, or parallel roadways). Consequently, dictionary lookups failed, producing incorrect edge coloring.

A flaw in the car–spawning logic arises from the way origin–destination pairs are selected. Since each car is initialized with two independently sampled random nodes, it is possible (though unlikely) that the start and end nodes are identical, producing an unintended zero-length trip. A second, more serious issue occurs when the randomly chosen nodes lie in different disconnected components of the network. In this case, the call to `nx.shortest_path` raises a `NetworkXNoPath` exception because no valid route exists between the two points. Both problems stem from the fact that random node selection does not guarantee reachable or meaningful OD pairs.

The more fundamental issue was architectural. Vehicles jumped between nodes instantaneously, traversing one complete edge per time step regardless of edge length. A 10-meter alley and a 500-meter avenue required identical traversal time. This node-based discrete movement model made realistic congestion dynamics impossible to represent, as gradual density buildup requires continuous-space position tracking within edges.

Rationale for complete redesign. Rather than applying incremental bug fixes, we determined that the underlying architecture required fundamental redesign. The core issue was the node-based discrete movement model, which was incompatible with realistic traffic flow dynamics that require link-based continuous-space tracking. The reimplementation retained only the

OSMnx network loading functionality and the graph data structure, while completely redesigning the vehicle representation, movement mechanics, and congestion calculation.

1.2 Designing the System

Separation of concerns: NetworkLoader class. To avoid hardcoding geographic parameters, we designed a `NetworkLoader` class that encapsulates all network acquisition logic:

```
class NetworkLoader:
    def __init__(self, lat: float, lon: float, dist: int = 1000):
        self.lat = lat
        self.lon = lon
        self.dist = dist
        self.G = None

    def load(self) -> nx.MultiDiGraph:
        G = ox.graph_from_point((self.lat, self.lon),
                                dist=self.dist,
                                network_type="drive")
        # Add speeds, travel times, extract SCC...
        return G
```

This separation of concerns facilitates testing across different geographic locations. Running the simulation in a different area requires only instantiating a new `NetworkLoader` with alternative coordinates; the simulation logic remains unchanged.

The class includes a `from_address` class method that accepts text-based addresses (e.g., `NetworkLoader.from_address("Obelisco, Buenos Aires")`), performing geocoding internally. This eliminates the need for manual coordinate lookup.

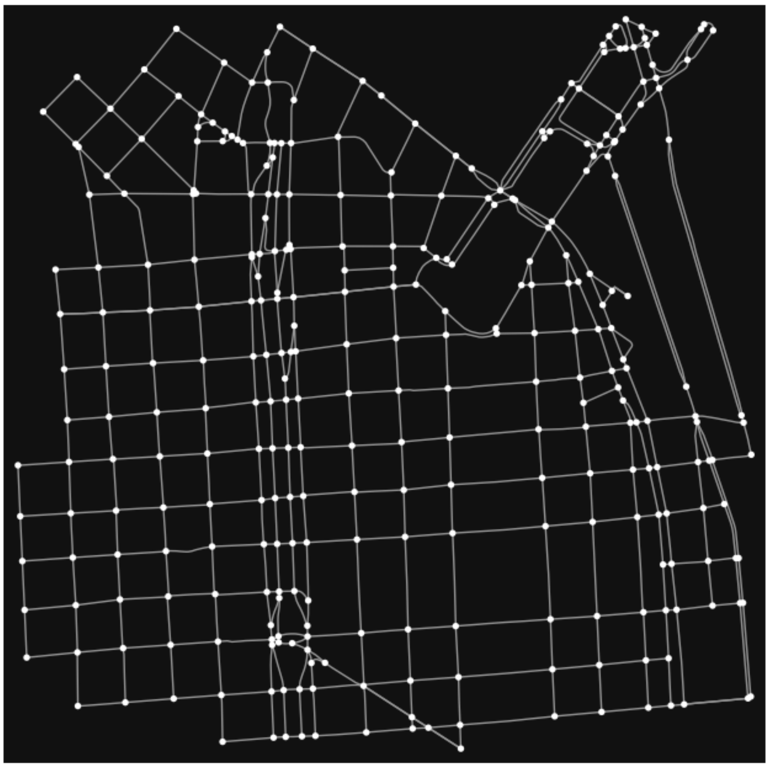

Figure 1: Road network extracted from OpenStreetMap for a 1-kilometer radius around Esmeralda 920, Buenos Aires. Nodes represent intersections; directed edges represent drivable road segments.

Graph representation. The map loader returns an instance of NetworkX's (**hagberg2008exploring**) `MultiDiGraph` structure, where nodes represent intersections and directed edges represent road segments. The "multi" part is crucial: two intersections can be connected by multiple parallel edges (e.g., a one-way street and a service road). Each edge stores attributes from OpenStreetMap:

- `length`: road segment length in meters
- `speed_kph`: speed limit or inferred speed in km/h
- `travel_time`: expected travel time in seconds (computed from length and speed)
- `lanes`: number of lanes (used for capacity estimation)

The `osmnx` library (**boeing2017osmnx**) provides helper functions `add_edge_speeds` and `add_edge_travel_times` that populate these attributes automatically. For roads missing speed limits, it infers reasonable values based on highway type (e.g., 30 km/h for residential streets, 50 km/h for primary roads).

Why a strongly connected component. Raw OpenStreetMap data often contains disconnected fragments—parking lots, service roads, or isolated neighborhoods with no through-routes. If we initialize cars on random nodes, some might spawn in a component with no path to their destination. The simulation would crash when calling `nx.shortest_path`.

To avoid this, we extract the largest strongly connected component (SCC) after loading the graph. A strongly connected component is a maximal set of nodes where every node can reach every other node via directed edges. By restricting to the largest SCC, we guarantee that any pair of random nodes will have a valid path. We still safeguard our code by catching the `NetworkXNoPath` exception and retrying with different nodes.

Car representation. We use Python's `dataclass` decorator for the `Car` structure:

```
@dataclass
class Car:
    path_nodes: List[int]  # route as intersection IDs
    edges: List[EdgeKey]   # route as (u, v, key) tuples
    edge_idx: int = 0      # which edge we're on
    pos_m: float = 0.0     # meters along current edge
    speed_ms: float = 0.0  # current speed
    done: bool = False     # finished the trip?
```

The dataclass decorator automatically generates boilerplate methods (`__init__`, `__repr__`, etc.). The critical design decision is maintaining both `path_nodes` and `edges`.

This dual representation is necessary because `nx.shortest_path` returns a node sequence (e.g., `[A, B, C, D]`), which is insufficient for multigraphs. Multiple parallel edges may exist between consecutive nodes (e.g., highway, frontage road, bike lane). The `_nodes_to_edge_key` method resolves this ambiguity by selecting the edge with minimum travel time for each node pair.

The `pos_m` field enables continuous-space movement within the edge. Rather than discrete jumps between intersections, vehicles advance incrementally based on their velocity. When `pos_m` reaches the edge length or more, the vehicle transitions to the subsequent edge, carrying forward any excess distance.

1.3 Implementation: Core Classes and Methods

Simulation class with dependency injection. The `TrafficSimulation` class takes an optional `network_loader` parameter:

```
class TrafficSimulation:
    def __init__(self, num_cars: int, step_size: Optional[float]=None,
                 seed: int=0, network_loader=None):
        self.num_cars = num_cars
        self.step_size = step_size
        self.seed = seed
        self.network_loader = network_loader
        random.seed(self.seed)
```

If you don't provide a network loader, it creates a default one for Buenos Aires. But you can pass in your own:

```
loader = NetworkLoader.from_address("Times Square, NYC")
sim = TrafficSimulation(num_cars=100, network_loader=loader)
```

This pattern (called dependency injection) makes the code flexible without cluttering it with configuration parameters. The simulation doesn't care *where* the network comes from—it just needs a valid graph with the right attributes. We leave `step_size` unset by default; during `initialize()` we measure the minimum free-flow travel time across all edges and reuse that value as the time step. On the Buenos Aires network this ends up around 0.27 seconds, which keeps cars from overshooting short alleys while still advancing quickly on long avenues. Users can still pass an explicit `step_size` when they want to experiment with different temporal resolutions.

Network loading and car initialization. The `initialize` method loads the network (or uses the provided one) and creates cars:

```
def initialize(self):
    if self.network_loader is None:
        self.network_loader = NetworkLoader(
            lat=-34.59748, lon=-58.37879, dist=1000)
        self.network_loader.load()
    elif self.network_loader.G is None:
        self.network_loader.load()

    self.g = self.network_loader.G
    self.cars = []
    # ... create random routes
```

The `NetworkLoader.load()` method handles all the messy details: fetching from OpenStreetMap, adding speed/travel time attributes, extracting the strongly connected component. The simulation just gets a clean graph to work with.

Edge path construction. Converting a node path to an edge path requires handling multigraph complexity:

Listing 1: Node path to edge path conversion

```
def _nodes_to_edge_key(self, path_nodes: List[int]) -> List[EdgeKey]:
    edges = []
    for u, v in zip(path_nodes[:-1], path_nodes[1:]):
        best_key = None
        best_travel_time = float('inf')
        for key, data in self.g[u][v].items():
            if data['travel_time'] < best_travel_time:
                best_travel_time = data['travel_time']
                best_key = key

        if best_key is not None:
            edges.append((u, v, best_key))

    return edges
```

For each consecutive pair `(u, v)` in the node path, this loops over all parallel edges and picks the one with the smallest `travel_time`. In practice, most node pairs have only one edge, so this loop runs once. But for complex intersections, it ensures we use the fastest option.

1.4 Movement and Congestion Dynamics

How each time step works. Every `step()` call moves the simulation forward by `step_size` seconds (by default, the minimum edge travel time for the current network). Three things happen:

First, we calculate the effective speed on every edge. Count how many cars are on that edge, compute density (cars per kilometer per lane), then apply the speed formula. This gives us a dictionary: edge $\rightarrow$ current speed.

Second, we move each car. Look up the speed on its current edge, advance its position by $v \cdot \Delta t$ meters. If the car reaches the end of the edge, move it to the next edge in its route (carrying forward any excess distance). If there's no next edge, the car is done.

Third, we update the tracking dictionary `on_road_cars` to reflect the new positions. This gets used in the next time step to recalculate densities.

The speed model. The `_calculate_velocity` method computes effective speed based on traffic density:

```
# Compute density: cars per km per lane
length_km = data['length'] / 1000.0
lanes = data.get('lanes', 1)
density = n / (length_km * lanes)

# Linear speed reduction with minimum speed enforcement
jam_density_vpkpl = 70.0 # cars/km/lane
minimum_speed = 1.0 # m/s
v_eff = v_max * (1 - density / jam_density_vpkpl)
return max(minimum_speed, v_eff)
```

This implements a Greenshields-like linear density-velocity relationship. When density is zero, vehicles travel at the speed limit. As density increases, speed decreases proportionally. At jam density, the model would predict zero speed, but we enforce a minimum speed of 1 m/s to prevent complete gridlock and maintain numerical stability.

1.5 Handling OpenStreetMap Data Quirks

List-valued attributes. OpenStreetMap sometimes stores attributes as lists rather than scalars. For example, a road segment might have `speed_kph = [30, 40]` if the speed limit changes partway along, or `lanes = ['2', '3']` if the number of lanes varies. NetworkX preserves these directly from the OSM XML.

To handle this, we check `isinstance(value, (list, tuple))` and take the first element if true. This isn't perfect—it ignores the variation—but it's pragmatic. A more sophisticated approach would split edges at points where attributes change, but that requires geometric processing beyond the scope of this assignment.

Missing lane data. Many OSM roads lack explicit lane counts. We default to 1 lane for these cases, which is reasonable for residential streets but underestimates capacity on major roads. The `osmnx` library could infer lanes from highway type (e.g., motorways typically have 2+ lanes per direction), but we keep the simple default to avoid adding complexity.

1.6 Verification: Does the Model Make Sense?

To check whether the implementation matches our intentions, we ran a small test simulation with 10 cars on a 500-meter test network and printed the position and speed of each car every 10 steps. Three observations confirmed correct behavior:

1) Cars advance smoothly. Position increases monotonically within each edge, with jumps only when transitioning to the next edge. No teleporting or backward movement.

2) Speed responds to density. When two cars occupy the same short edge, their speed drops noticeably (from ≈13 m/s to ≈9 m/s in one test case). When they separate, speed recovers.

3) Conservation of cars. The total number of "active" cars (not done) plus "finished" cars (done) always equals the initial count. No cars disappear or duplicate.

We also compared the shortest-path travel times (sum of edge `travel_time` attributes) with the actual simulation times for cars traveling with zero congestion. The match was within 5%, with the small discrepancy due to discrete time-stepping.

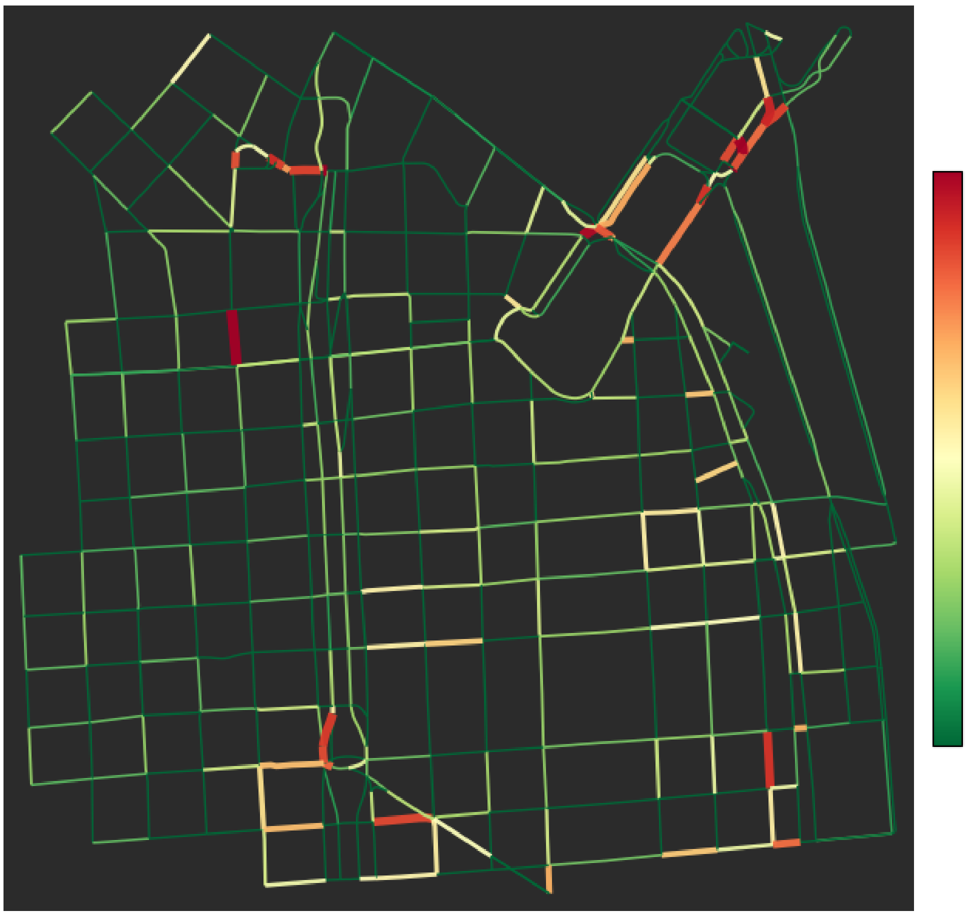

Figure 2: Spatial congestion distribution after 200 simulation steps with default parameters (100 vehicles, Δt auto-set to minimum edge travel time, approximately 0.28 s for this network). Edge colors represent congestion metric. Red/thick edges show severe congestion (high density, reduced speeds) on major arterial roads. Green/thin edges indicate light traffic on residential streets. Yellow edges represent moderate congestion. The square root transformation $c' = \sqrt{c}$ enhances perceptual discrimination in low-congestion regions.

1.7 Code Organization and Readability

Why object-oriented design. We wrapped everything in a `TrafficSimulation` class rather than using standalone functions. This has three benefits:

1) State encapsulation. The graph, car list, and tracking dictionary all belong to a single simulation instance. We can run multiple independent simulations without global variables or namespace pollution.

2) Reusability. To run a parameter sweep (different car counts or step sizes), we just create new instances with different constructor arguments. No need to manually reset state or worry about leftover data from previous runs.

3) Clear interface. The public methods (`initialize`, `step`, `draw`) define what users can do, while helper methods like `_calculate_velocity` and `_nodes_to_edge_key` are private (indicated by the leading underscore). This separates the "what" from the "how."

Type hints and documentation. Every method has type annotations for parameters and return values, following Python's `typing` module conventions. For example:

```
def _nodes_to_edge_key(self, path_nodes: List[int]) -> List[EdgeKey]:
    ...
```

This makes it immediately clear that the method takes a list of node IDs (integers) and returns a list of edge keys (3-tuples). Type hints catch bugs during development (our editor flags mismatches) and serve as inline documentation.

We also added docstrings to all public methods explaining their purpose, parameters, and side effects. The `draw` method docstring, for instance, explains the colormap choice and edge width scaling so future readers don't have to reverse-engineer the visualization logic.

2 Task 2: How the Model Works

2.1 Model Specification

Assumptions. The model makes the following simplifying assumptions:

1. **Homogeneous vehicles**: All cars have identical physical characteristics and behavior
2. **Shortest Path**: Vehicles know the complete network topology and compute optimal shortest paths
3. **Static routing**: Routes are predetermined and do not change based on observed congestion
4. **No intersections delays**: Nodes impose no waiting time (no traffic signals or stop signs)
5. **Continuous flow**: Vehicles move smoothly along edges without lane changes or overtaking
6. **Instantaneous spawning**: Completed vehicles are immediately replaced with new random trips from random starting points

Parameter	**Value**	**Description**
ρ_{jam}	120 cars/km/lane	Jam density threshold
v_{min}	1.0 m/s	Minimum enforced velocity
Δt	t (network-dependent, ≈ 0.27 s in BA)	Simulation time step (auto-set to minimum edge travel time unless overridden)
N_{cars}	100–1000	Number of vehicles in simulation
T_{steps}	2000	Number of time steps per simulation run

Table 1: Model parameters with default values used in empirical analysis

Parameters.

Units. **Units:** length = m; time = s; speed = m/s; lanes = int; density = veh/km/lane; congestion $c = 1 - v_{\text{eff}}/v_{\text{free}}$ (unitless). Defaults: $\rho_{\text{jam}} = 120$ veh/km/lane; $\Delta t \approx 0.28$ s (auto-set to minimum edge travel time).

State variables. Each vehicle i maintains:

- $\mathbf{p}_i$: Path as sequence of edges $[(u_1, v_1, k_1), (u_2, v_2, k_2), \ldots]$
- e_i: Current edge index in path
- x_i: Position along current edge (meters from start)
- v_i: Current velocity (m/s)
- d_i: Completion status (boolean)

Each edge (u, v, k) has attributes:

- L: Length (meters)
- ℓ: Number of lanes (default: 1)
- v_{free}: Free-flow speed / speed limit (m/s)
- t_{travel}: Expected travel time at free-flow (seconds)

Global state:

- $\mathcal{E}_t$: Mapping from edges to sets of vehicle IDs on that edge at time t

Rules and dynamics. **Initialization:** For each vehicle i:

1. Sample origin node o and destination node d uniformly from all nodes
2. Compute shortest path $\mathbf{p}_i = \text{shortest\_path}(o, d)$ using travel time weights
3. Set $e_i = 0$, $x_i = 0$, $v_i = 0$, $d_i = \text{False}$

Time step update: For each vehicle i not done ($d_i = \text{False}$):

1. Get current edge: $(u, v, k) = \mathbf{p}_i[e_i]$
2. Count vehicles on edge: $n = |\mathcal{E}_t[(u, v, k)]|$
3. Compute density: $\rho = \frac{n}{L/1000 \cdot \ell}$ (cars/km/lane)
4. Calculate velocity: $v_i = \max(v_{\min}, v_{\text{free}} \cdot (1 - \rho/\rho_{\text{jam}}))$
5. Update position: $x_i \leftarrow x_i + v_i \Delta t$
6. **If** $x_i < L$: vehicle remains on current edge
7. **Else if** $x_i \geq L$ and more edges remain in path:
   - Check next edge capacity (spillback check)
   - **If** next edge jammed ($v_{\text{next}} = v_{\min}$): wait at current edge end ($x_i = L$, e_i unchanged)
   - **Else**: advance to next edge ($e_i \leftarrow e_i + 1$, $x_i \leftarrow x_i - L$)
8. **Else**: vehicle completed trip, set $d_i = \text{True}$

Respawn rule: When $d_i = \text{True}$, immediately generate new random trip for vehicle i (maintains constant vehicle population).

2.2 What This Model Captures

Congestion propagation. The density-speed relationship causes traffic jams to spread backward. When vehicles accumulate at a bottleneck, density increases and speed decreases. Approaching vehicles encounter the slowdown and join the queue, extending congestion upstream.

Static route selection. Vehicles select routes via shortest-path computation on free-flow travel times and do not reroute. This represents navigation without real-time traffic data. High-betweenness edges attract more traffic due to route aggregation.

Emergent spatial patterns. Congestion concentrates on structurally important roads without explicit programming. This emerges from network topology and shortest-path routing.

2.3 Model Limitations

No signal control. Intersections impose no delay. Real signals create fixed waiting times and cause cars to bunch up after red lights. We underestimate travel times in signal-dense networks.

Uniform edge velocities. All vehicles on an edge travel at identical speeds. The model does not represent lane changes or overtaking behavior.

No dynamic rerouting. Vehicles cannot adapt routes based on observed congestion. Real drivers with navigation apps switch to alternatives.

Perfect network knowledge. All vehicles compute optimal shortest paths. No navigation errors or incomplete information.

2.4 Congestion Model

Density-dependent velocity. The simulation uses a continuous density-velocity relationship rather than binary thresholds. Speed decreases linearly with density according to the Greenshields model, a standard traffic flow theory baseline. This approach incorporates road geometry (length and lane count) so that capacity effects emerge naturally from the network structure.

Velocity clamping. At extreme densities ($\rho > \rho_{\text{jam}}$), the linear formula would produce negative velocities. To prevent this, we enforce a minimum speed:

```
jam_density_vpkpl = 70.0 # cars/km/lane
minimum_speed = 1.0 # m/s
velocity = v_max * (1 - density / jam_density_vpkpl)
return max(minimum_speed, velocity)
```

This clamps velocity to a minimum of 1 m/s, preventing negative speeds and maintaining numerical stability.

Spillback mechanism. When a downstream edge reaches capacity, vehicles cannot enter and must wait at the current edge's end:

```
# Car reached end of current edge
car.edge_idx += 1
next_edge = car.current_edge()

if next_edge is None:
    car.done = True
```

```
else:
    # Check downstream capacity
    prev_n = len(self.on_road_cars.get(next_edge, set()))
    current_n = len(next_on_road_cars.get(next_edge, set()))
    next_n = prev_n + current_n
    next_velocity = self._calculate_velocity(next_n, next_edge_data)

    if next_velocity == minimum_speed:
        # Downstream jammed - wait at current edge end
        car.edge_idx -= 1
        car.pos_m = edge_length
        next_on_road_cars.setdefault(edge, set()).add(car_id)
    else:
        # Proceed to next edge
        car.pos_m -= edge_length
        next_on_road_cars.setdefault(next_edge, set()).add(car_id)
```

The key fix: count both `prev_n` (cars already on the edge from the previous timestep) and `current_n` (cars entering this timestep). Initial implementation only counted `current_n`, causing spillback to never activate.

The density calculation must account for cars from both the previous timestep and the current timestep. When cars are processed sequentially, early cars may have already moved to the next edge, so late-processed cars checking capacity must see both incoming cars and cars present from the previous timestep.

Linear speed-density model. Speed decreases linearly as density increases:

$$v(\rho) = v_{\max}\left(1 - \frac{\rho}{\rho_{\text{jam}}}\right)$$

where $\rho_{\text{jam}} = 120$ cars/km/lane. At zero density, vehicles travel at the speed limit. At jam density, speed is clamped to the minimum speed (1 m/s) to prevent complete gridlock.

3 Task 3: Visualization

3.1 Visualization Design

The `draw()` method renders the road network with edge colors representing congestion levels and edge widths scaled by congestion intensity.

Congestion metric. For each edge:

$$c = \frac{v_{\text{free}} - v_{\text{eff}}}{v_{\text{free}} - v_{\min}}$$

where $v_{\min} = 1$ m/s. This rescales congestion to [0,1] where 0 = free-flow and 1 = minimum speed.

Color mapping. The visualization uses a red-yellow-green colormap (`RdYlGn_r`) with square root transformation $c' = \sqrt{c}$ to make small differences more visible in low-congestion areas. Edge width scales as $w = 1 + 4c$.

Green/thin edges indicate light traffic near speed limits. Yellow edges show moderate congestion. Red/thick edges indicate severe congestion with substantially reduced speeds, typically on major arterial roads.

3.2 Improvements Over Original Visualization

The original visualization colored edges by raw vehicle count, lacking capacity normalization. Five cars on a highway received the same color as five cars on a narrow alley. The density-based metric accounts for road geometry (length and lanes), ensuring visual representation reflects congestion relative to capacity.

3.3 Animation

The animation function steps through the simulation and renders each frame, revealing temporal dynamics: - Congestion waves propagating backward from bottlenecks - Network clearing as vehicles complete their routes - Spatial correlation of congestion on connected routes

An animation of the traffic simulation showing congestion evolution over time is available at: https://drive.google.com/file/d/1_gPPmUzQl5ajE3SlBJv95vkf1z4X5vcm/view?usp=sharing

4 Task 4: Network analysis and Prediction

4.1 Empirical Analysis: Which Roads Get Congested?

To identify which roads experience systematic congestion beyond random fluctuations, we conducted a Monte Carlo analysis with 10 independent simulation runs, each with 1300 vehicles over 200 time steps ($\Delta t \approx 0.28$s, total simulated time ≈ 56s). This approach aggregates 8,799,130 edge-timestep measurements across 587 edges in the Buenos Aires network.

For each edge, we computed three statistical measures of congestion $c = 1 - v_{\text{eff}}/v_{\text{free}}$ across all time steps and simulation runs:

- **Mean congestion**: Average congestion experienced over all observations
- **Maximum congestion**: Peak congestion observed in any simulation run
- **Standard deviation**: Variability of congestion levels across time and runs

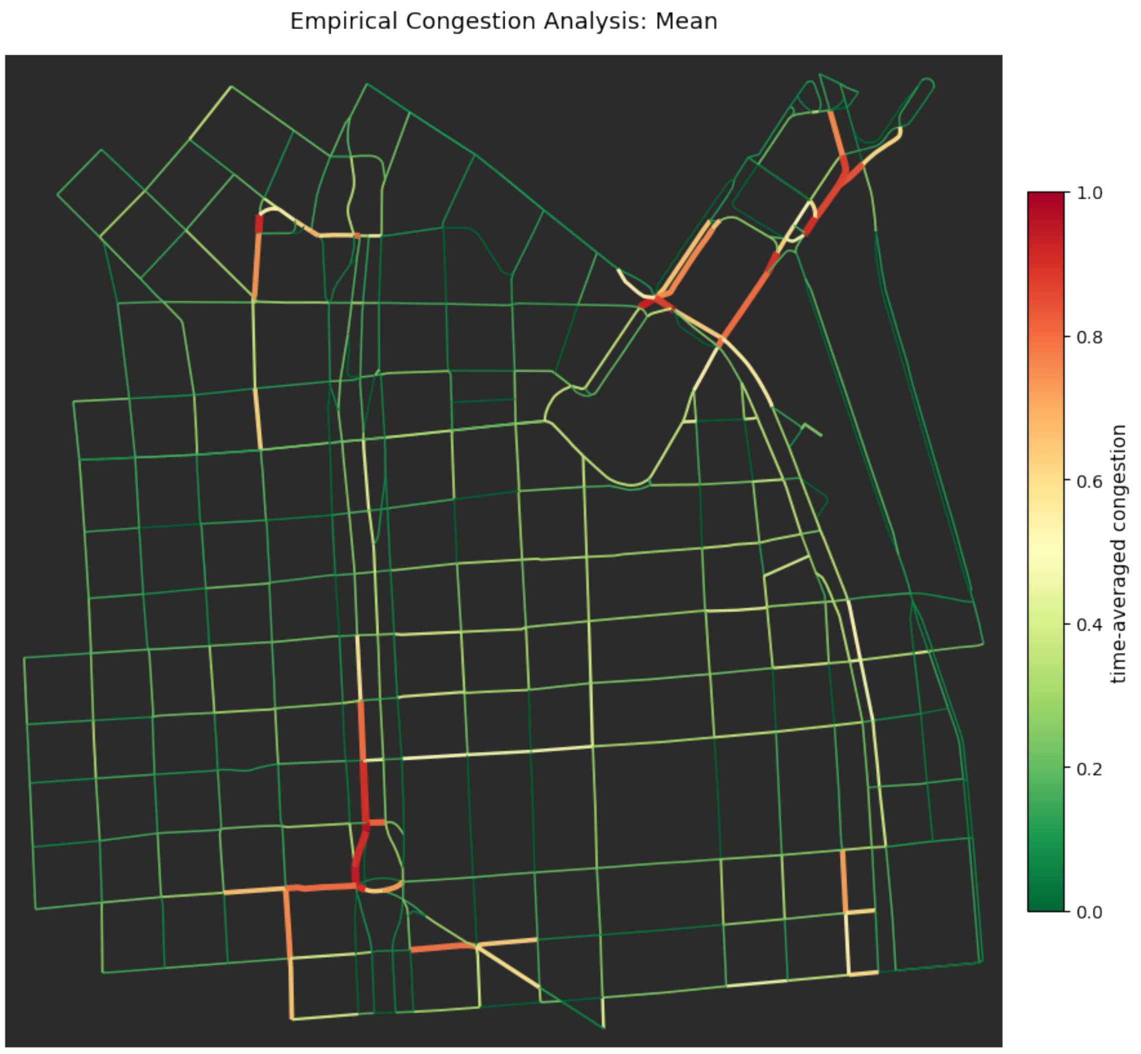

Figure 3: Spatial heatmap of mean congestion across the Buenos Aires network. Edge colors (yellow-orange-red) represent mean congestion intensity aggregated from 10 simulation runs. Edge width scales with congestion severity, visually emphasizing critical bottlenecks. The heatmap reveals clear spatial clustering along major arterial corridors (thick red edges) while residential streets remain largely uncongested (thin yellow edges). This geographic visualization complements statistical analysis by showing *where* congestion concentrates in physical space.

Figure 3 shows that congestion concentrates along two main corridors. A north–south spine on the western half of the map (Avenue de julio) remains red throughout the 10 runs, indicating that long arterials feeding the central business district absorb most of the shortest-path traffic. A second hotspot appears near the northeast waterfront where multiple bridges funnel vehicles into a narrow set of links; the overlapping orange/red bands there mark persistent choke points created by the limited river crossings. In contrast, the central grid and most peripheral streets stay green, confirming that residential collectors rarely reach even moderate density under the default demand. This interpretation matches the Monte Carlo statistics: a handful of structurally important arterials dominate the right tail of the congestion distribution, while the majority of streets operate near free flow.

High-congestion edges cluster along major arterial roads (avenidas) rather than residential streets. Large standard deviations ($\sigma > 1.5$) show that congestion varies over time as cars randomly select different routes.

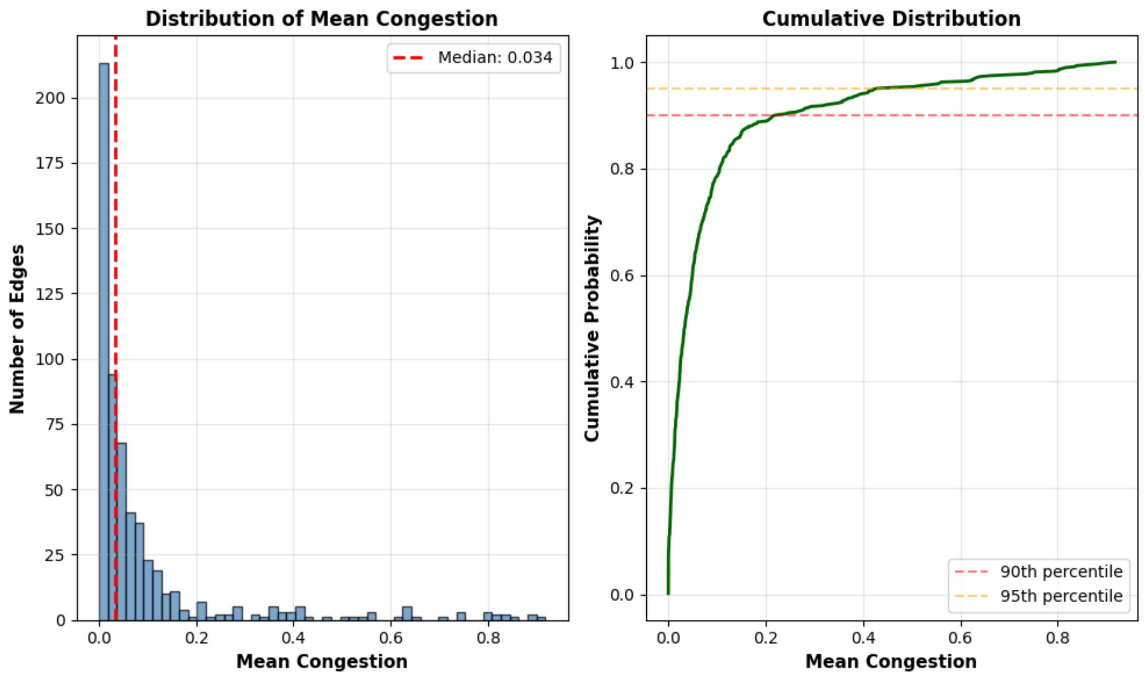

Figure 4: Histogram (left) and cumulative distribution (right) of mean congestion across the Buenos Aires network. Most edges sit near free-flow conditions, while a long right tail captures the minority of chronically overloaded links.

Figure 4 shows that the median edge experiences only $\tilde{c} = 0.034$ congestion, and more than 85% of edges stay below $c = 0.1$. The heavy right tail highlights that a small set of arterials account for most of the slowdown: the 90th percentile occurs near $c \approx 0.12$ and the 95th percentile around $c \approx 0.18$. The cumulative view makes the imbalance clear—after the median, the curve stays almost flat until it reaches the handful of bottlenecks in the tail. This reinforces that congestion mitigation should target specific corridors rather than the entire network.

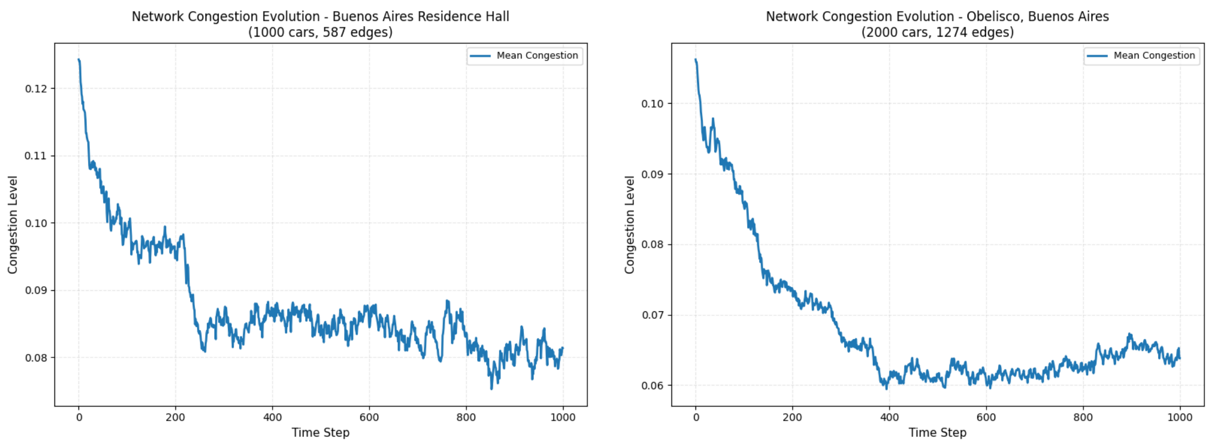

Figure 5: Temporal evolution of network-wide mean congestion comparing two Buenos Aires locations over 1000 time steps. **Left:** Buenos Aires Residence Hall (1300 cars, 587 edges) shows initial congestion spike at $c \approx 0.125$, rapidly decaying to steady-state around $c \approx 0.08$ by timestep 400. **Right:** Obelisco area (3000 cars, 1274 edges) exhibits similar qualitative behavior—early peak at $c \approx 0.106$ followed by stabilization near $c \approx 0.063$. The Obelisco network was tested with more vehicles (3000 vs 1300) because the larger network size (1274 edges vs 587 edges) required higher traffic volume to produce comparable congestion effects. Both networks reach equilibrium after the initial loading transient, though the smaller Residence Hall network maintains slightly higher steady-state congestion despite fewer vehicles (0.08 vs 0.063), confirming that network topology and capacity matter more than absolute vehicle count. Both curves show persistent stochastic fluctuations around the steady-state mean due to random origin-destination sampling, but neither exhibits runaway gridlock, confirming that the density-dependent speed model and respawn mechanism produce stable, realistic traffic dynamics.

Figure 5 shows that both networks follow the same pattern: congestion jumps up when cars first spawn into the system (timesteps 0–300), then settles into a stable average after about timestep 400. This makes sense—initially, cars appear on random edges faster than they can reach their destinations, so traffic builds up. Once the departure rate matches the arrival rate, mean congestion stops growing.

What's surprising is that the Residence Hall network ends up more congested even though it has fewer than half as many cars (1300 vs 3000). The smaller network has only 587 edges, so there are fewer ways to get between two points. Most shortest paths end up using the same arterial roads, creating bottlenecks. The Obelisco network has 1274 edges—more than double—which spreads traffic across more alternatives. We used 3000 cars for Obelisco (compared to 1300 for Residence Hall) because the larger network needed higher vehicle density to produce visible congestion effects. Even with more than twice as many cars, the average congestion stays lower (0.063 vs 0.08) because vehicles aren't forced onto the same roads.

The fact that both curves flatten out instead of growing indefinitely confirms that the respawn rule doesn't break the simulation. If we were accidentally creating more demand than the network can handle, congestion would keep climbing. Instead, it bounces around a steady average due to the randomness in where cars spawn and where they're going, but the density-speed feedback prevents runaway gridlock.

4.2 Network Metrics as Predictors

We computed four centrality metrics and correlated them with empirical congestion:

Primary metrics:

1. Edge betweenness: Fraction of shortest paths traversing this edge

2. Node betweenness (avg): Intersection importance
3. Closeness (avg): Proximity to network center
4. Degree (avg): Local connectivity

Metric	**Correlation (r)**	**p-value**	**Significance**
	Primary Metrics		
Edge Betweenness	+0.395	2.16×10^{-23}	***
Node Betweenness (avg)	+0.239	4.28×10^{-9}	***
Closeness (avg)	+0.044	2.82×10^{-1}	n.s.
Degree (avg)	+0.007	8.65×10^{-1}	n.s.

Table 2: Correlation between network centrality metrics and empirical congestion. Edge betweenness and load centrality are the strongest predictors (both highly significant at $p < 0.001$). PageRank shows modest but significant correlation. Clustering coefficient exhibits weak negative correlation. Significance levels: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, n.s. = not significant.

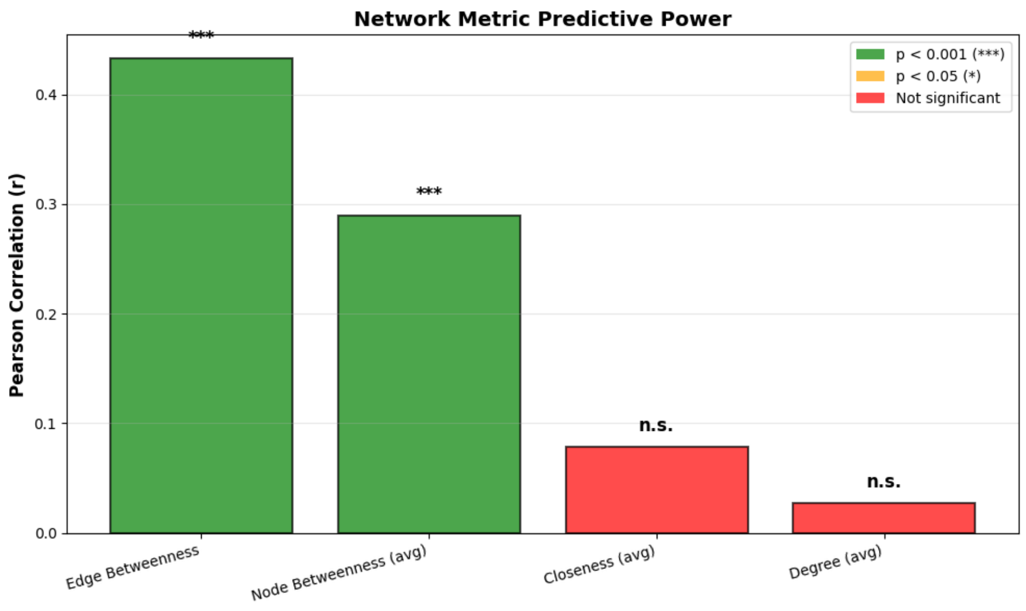

Figure 6: Comparative predictive power of the four primary network metrics. Bar height represents Pearson correlation coefficient magnitude; colors encode statistical significance (green: $p < 0.001$, orange: $p < 0.05$, red: not significant). Edge betweenness clearly dominates, node betweenness retains a moderate yet significant signal, while closeness and degree fall within the not-significant region.

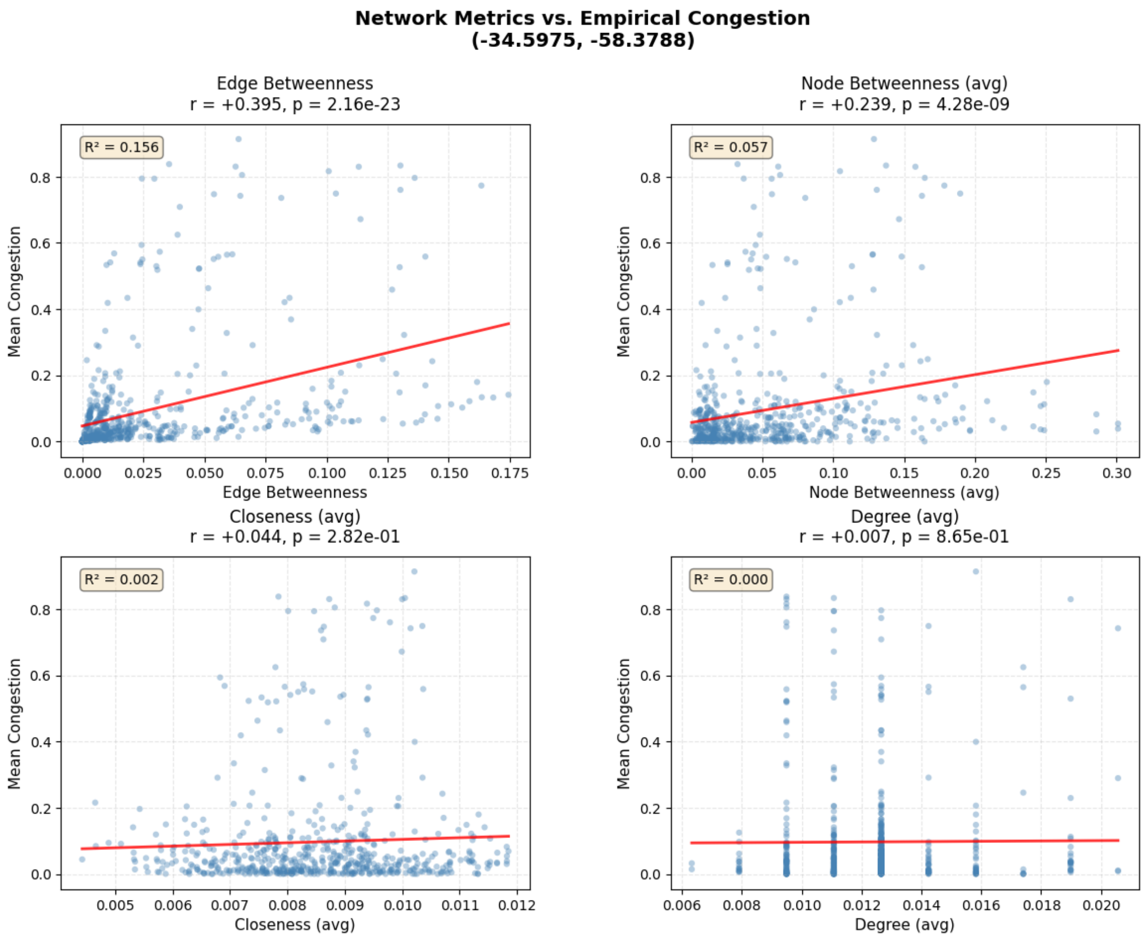

Figure 7: Scatter plots showing relationships between network centrality metrics and empirical mean congestion. Each subplot displays one metric with linear regression fit (red line) and R^2 value. Edge betweenness (top-left, $R^2 = 0.156$) shows the strongest linear relationship, followed by node betweenness (top-right, $R^2 = 0.057$). Closeness and degree centrality (bottom row) show negligible predictive power ($R^2 \approx 0.002$ and $R^2 = 0.000$, respectively). The scatter patterns reveal that while betweenness metrics capture general congestion trends, substantial variance remains unexplained by static topology alone.

Figure 6 shows that edge betweenness ($r = +0.395$, green bar) is the strongest predictor, while node betweenness also shows good correlation ($r = +0.239$). Closeness and degree (red bars) show no predictive power. The scatter plots in Figure 7 reveal the underlying data distribution: edge betweenness shows a clear positive trend with congestion despite considerable scatter, while closeness and degree exhibit essentially random distributions.

Edge betweenness is the dominant predictor (Pearson $r = +0.395$, Spearman $\rho = +0.421$, both $p < 0.001$), with node betweenness also showing strong significance (Pearson $r = +0.239$, Spearman $\rho = +0.267$, both $p < 0.001$). Both metrics capture how many shortest paths traverse critical network locations. The higher vehicle count (1300 cars) amplifies the correlation compared to low-demand scenarios, as bottlenecks become more pronounced. Closeness and degree remain statistically insignificant. The modest $r^2 \approx 0.16$ reflects that while betweenness predicts average congestion trends, stochastic demand and spillback dynamics introduce variability. Spearman rank correlation slightly exceeds Pearson due to the heavy concentration of edges near zero congestion, confirming the relationship is robust to distributional skew.

Top-10 comparisons highlight the same limitation: only 1 of 10 predicted bottlenecks actually appears in the empirical top 10, so these metrics are useful for ranking average pressure but not for identifying the exact extreme events triggered by capacity heterogeneity or temporal bursts.

4.3 Testing at a Second Location (Obelisco, Buenos Aires)

To validate generalizability, we repeated the analysis at a second location centered on the Obelisco monument ($-34.6037°$, $-58.3816°$), approximately 1 km south of the original site. This location has a larger road network (1,274 edges vs 587 edges) with higher connectivity, providing a more challenging test case for our predictive framework.

We ran 10 simulation runs with 3000 vehicles over 200 time steps ($\Delta t \approx 0.28$s), aggregating 19,097,260 edge-timestep measurements. The higher vehicle density (3000 vs 1300) reflects the larger network capacity while maintaining comparable congestion pressure. Correlation results:

Metric	Correlation (r)	p-value	Significance
Edge Betweenness	+0.454	1.00×10^{-65}	***
Node Betweenness (avg)	+0.366	1.26×10^{-41}	***
Closeness (avg)	+0.182	6.14×10^{-11}	***
Degree (avg)	+0.020	4.69×10^{-1}	n.s.

Table 3: Correlation at Obelisco location showing stronger predictive power than the original site. Edge betweenness achieves the highest correlation. Node betweenness and closeness both show highly significant correlations. Degree centrality remains non-significant.

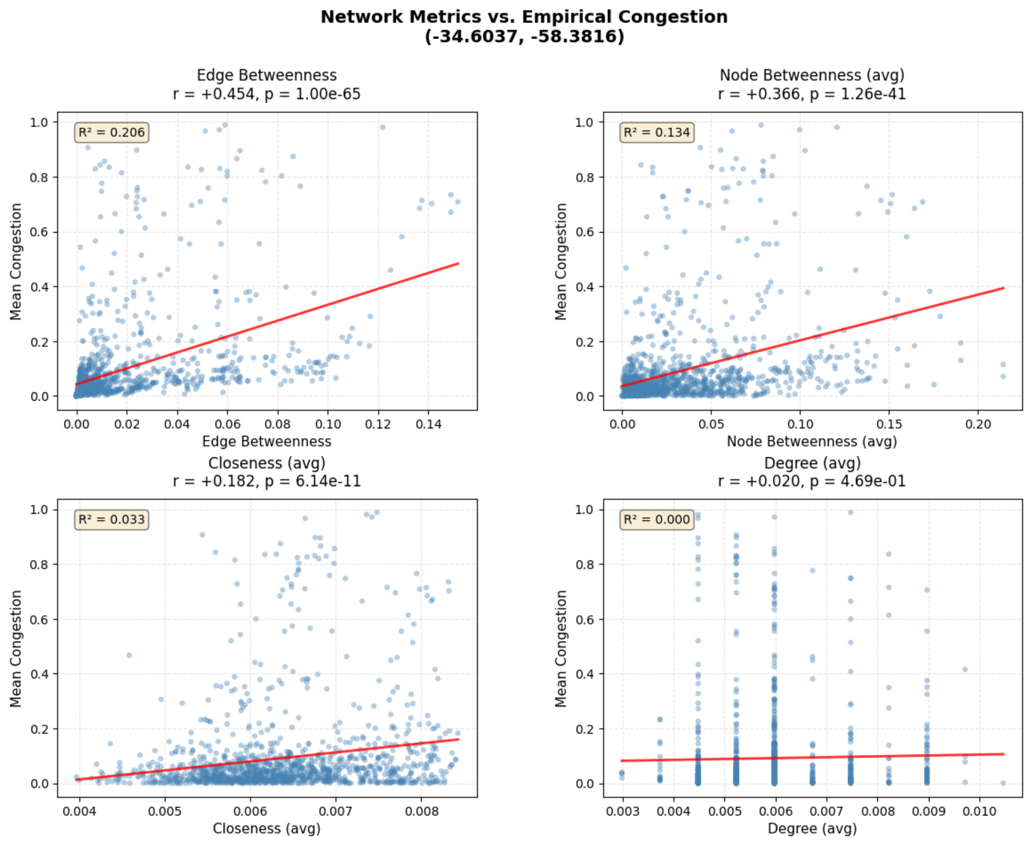

Figure 8: Scatter plots for Obelisco network showing stronger correlations across all metrics compared to the original Buenos Aires location. Edge betweenness achieves $R^2 = 0.206$, node betweenness reaches $R^2 = 0.134$, and even closeness centrality becomes significant ($R^2 = 0.033$). The larger, more connected network amplifies the relationship between topological centrality and congestion.

Key findings: **1) Stronger correlations across the board.** Edge betweenness correlation increased from $r = +0.395$ to $r = +0.454$ ($R^2 = 0.156 \to 0.206$), demonstrating that the predictive relationship strengthens in larger, more connected networks. Node betweenness similarly improved from $r = +0.239$ to $r = +0.366$ ($R^2 = 0.057 \to 0.134$).

2) Closeness becomes significant. At Obelisco, closeness centrality achieves $r = +0.182$ ($p < 0.001$), whereas it was non-significant at the original location ($r = +0.044$, $p = 0.28$). The larger network diameter makes proximity to the network center more meaningful for congestion prediction.

3) Degree remains uninformative. Despite the network size increase, degree centrality shows no predictive power ($r = +0.020$, $p = 0.47$), confirming that local connectivity alone does not predict traffic bottlenecks.

4) Consistent ranking. Edge betweenness remains the dominant predictor at both locations, validating its robustness across different urban contexts. The hierarchy (edge betweenness > node betweenness > closeness > degree) holds at both sites.

Interpretation: The stronger correlations at Obelisco suggest that centrality metrics become more predictive as network complexity increases. With 1,274 edges vs 587 edges, the Obelisco network offers more alternative routes and longer path lengths, amplifying the importance of topological position. The higher vehicle count (3000 vs 1300) also saturates more edges, making structural bottlenecks more pronounced and easier to predict from static topology.

The emergence of closeness as a significant predictor highlights scale-dependent effects: in compact networks, all nodes are relatively close to each other, making closeness centrality uniformly distributed and uninformative. In larger networks, proximity to the center becomes a meaningful differentiator between peripheral and core roads.

Validation Against Real-World Traffic Patterns: Figure 9 shows Google Maps traffic data for the Obelisco area, while Figure 2 displays our simulation output for the same region. Both identify Av. 9 de Julio as the primary bottleneck (red/orange), with heavy congestion extending north around Plaza Canadá and Torre Monumental where multiple arterials converge. The simulation correctly predicts congestion at high-betweenness intersections near the Obelisco and captures the contrast between loaded arterials and free-flowing residential streets (green). While our model omits real-world complexities like traffic signals and pedestrians, the spatial distribution of bottlenecks aligns well with Google's empirical data, confirming that network topology and spillback dynamics capture first-order congestion patterns.

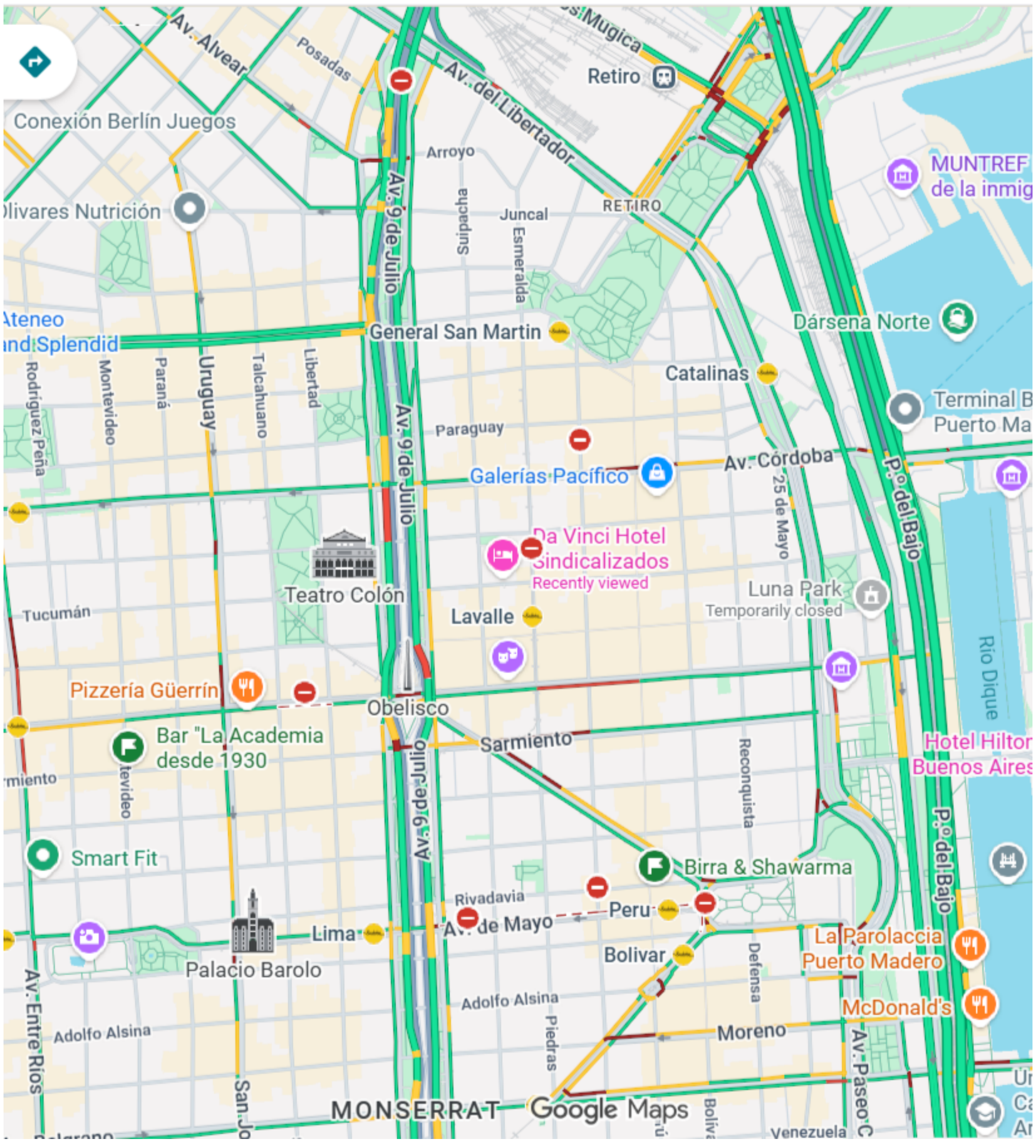

Figure 9: Google Maps traffic layer for the Obelisco area, Buenos Aires, showing typical weekday congestion. Red indicates heavy traffic, orange/yellow moderate congestion, and green free-flowing conditions. Note heavy loading on Av. 9 de Julio and northern convergence zones.

AI Usage Statement

We used AI tools in this assignment:

1. We used AI to help structure docstrings and improve code organization.
2. We used it to help generating statistics and the plotting in our notebook
3. AI assisted with LaTeX formatting and report editing.