**DESIGN AND SIMULATION , RESULTS**

**Create baseline network**

**Step 1: Validate Individual Files**

Create route for each vehicle type and combine them into one

Before combining, ensure each .trips.xml and .rou.xml works as expected:

* Run each individually in SUMO to confirm there are no issues:

sumo-gui -n osm.net.xml -r osm.bus.rou.xml

sumo-gui -n osm.net.xml -r osm.passenger.rou.xml

sumo-gui -n osm.net.xml -r osm.truck.rou.xml

sumo-gui -n osm.net.xml -r osm.motorcycle.rou.xml

**Step 2: Combine Route Files**

Use SUMO's duarouter tool to merge the routes into a single .rou.xml file:

duarouter -n osm.net.xml -r osm.bus.rou.xml,osm.passenger.rou.xml,osm.truck.rou.xml,osm.motorcycle.rou.xml -o combined\_routes.rou.xml

This will create combined\_routes.rou.xml, which contains routes for all vehicle types

**Step 3: Adjust Vehicle Proportions**

Adjust traffic proportions (., 70% passenger cars, 10% buses, 10% trucks, 10% motorcycles):

Using automated Pythons script called adjustTrips.py which creates:

* osm.passenger\_adjusted.trips.xml
* osm.bus\_adjusted.trips.xml
* osm.truck\_adjusted.trips.xml
* osm.motorcycle\_adjusted.trips.xml

**Step 4: Merge Trips into a Single File**

Merge trips into a single .trips.xml file:

**Using SUMO Tools:**

use duarouter to generate routes from multiple trip files:

duarouter -n osm.net.xml -t osm.bus\_adjusted.trips.xml,osm.passenger\_adjusted.trips.xml,osm.truck\_adjusted.trips.xml,osm.motorcycle\_adjusted.trips.xml -o combined\_trips\_routes.rou.xml

This combines the trips and generates a route file combined\_routes.rou.xml. Or another option is to use the merge\_trips.py code but this will give you cobined\_trips.xml but not necessary since you will have to create routes files again from the combined trips file.

**Step 5: Run the Simulation**

Use the combined files to run the full simulation:

sumo-gui -n osm.net.xml -r combined\_routes.rou.xml

STREAMLINING TO CREATE BASELINE

Codes reports.py does the followings:

**Simulation Management**:

* Inserts vehicles until the limit is reached.
* Randomly removes vehicles at a higher rate after the limit is reached to ease congestion.

**Dynamic Removal**:

* If vehicles exceed 2000, uses a higher removal rate.
* Below 2000, the removal rate is reduced to stabilize traffic.

**Data Collection**:

* Tracks vehicle counts and average speed at each simulation step.

**Visualization**:

* Plots:
  + **Vehicle Count**: Shows the rise to the peak and subsequent reduction.
  + **Average Speed**: Shows the change in ed over time, indicating how traffic eases after the peak.

**Report Generation**:

* Saves key metrics and trends to a text file.

DYNAMIC GRAPH THEORY APPLICATION:

**Dynamic vehicles reroute to avert traffic and congestion**

**1. Extract Start and End Coordinates**

The first task is to extract the **start** and **end coordinates** of the chosen vehicles:

* Use these coordinates to find routes between the origin and destination.
* Identify any congested areas along the default routes.

**2. Identify Congested Areas**

Analyzing the network to identify potential bottlenecks:

* **Use SUMO metrics**:
  + **Vehicle density**: Find edges with the highest vehicle count.
  + **Speed drops**: Identify edges where the average speed is significantly lower than the speed limit.
* **Dynamic graph analysis**:
  + Represent the network as a graph where:
    - Nodes are intersections.
    - Edges are roads with weights based on travel time or congestion level.

**3. Apply Dynamic Routing**

For each vehicle: (chosen 5-10 vehicles passenger v)

1. **Default Route**:
   * Compute the shortest path based on current conditions.
   * Avoid roads with high congestion weights.
2. **Dynamic Rerouting**:
   * If traffic density exceeds a threshold, dynamically reroute the vehicle to a less congested path.
   * Include potential passenger pickup/drop-off points along the route to optimize ride-sharing. (we will add delay of like 5-10 seconds for pickup and drop off)
   * Use NetworkX to model the road network and update weights dynamically.
3. **Algorithm**:
   * Use Dijkstra’s or A\* algorithm with dynamically updated edge weights:
     + Weight = travel time + penalty for congestion.
   * Update edge weights periodically based on real-time data.

**4. Re-Simulate the 5 Vehicles**

1. Create a **new route file** (dynamic\_routes.rou.xml) for the selected vehicles using the dynamically computed routes.
2. Run the simulation again with these updated routes.

**5. Compare Baseline and Dynamic Results**

After running the new simulation, we will compare the results to the baseline:

1. **Metrics to Compare**:
   * Total travel time.
   * Average speed.
   * Distance traveled.
   * Number of stops/delays.
   * Fuel consumption or emissions (optional).
2. **Visualization**:
   * We Plot graphs showing the differences in travel time, speed, and congestion levels before and after dynamic routing.

**6. Generate the Report**

Produce a detailed report for analysis:

* **Summary**:
  + Baseline vs. improved metrics.
* **Graphs**:
  + Travel time per vehicle (before vs. after).
  + Speed trends (before vs. after).
  + Congestion levels on key routes.
* **Conclusion**:
  + Highlight the benefits of applying dynamic graph theory.
  + Discuss limitations or areas for further optimization.

1. TRAFFIC LIGHT USING HEURISTIC ALGORITHM (optional)

The tlscycleadaptation.py script uses a **heuristic algorithm** to dynamically adapt the traffic signal cycle lengths of a traffic light system based on real-time traffic conditions.

**Purpose**

The script aims to improve traffic flow and reduce congestion at intersections controlled by traffic lights by adjusting signal timing dynamically. This adjustment helps:

1. **Reduce Waiting Times**: Optimizing the duration of green and red lights based on traffic volume and vehicle flow.
2. **Minimize Congestion**: By ensuring that the traffic flow is smooth and reducing bottlenecks.
3. **Enhance Overall Efficiency**: Adapting the cycle lengths to the current state of traffic improves throughput and minimizes delays.

**Heuristic Algorithm – optional to optimize on TLC**

A heuristic algorithm is used, meaning it applies practical methods and rules to find near-optimal solutions without guaranteeing absolute optimality (as exact methods would be computationally expensive in real-time).

1. **Traffic Observation**:
   * Script to monitors **incoming and outgoing traffic flows** at each junction or traffic signal.
   * to considers metrics such as:
     + Vehicle queues.
     + Vehicle arrival rates.
     + Intersection delays.
2. **Signal Timing Adjustment**:
   * Based on observed traffic patterns, the script will adjusts the **cycle length** (total duration of a signal phase) and **phase durations** (duration of green, yellow, and red lights for each direction).
   * Adjustments are based on heuristic rules:
     + If a lane has a high vehicle queue or inflow, increase the green light duration for that direction.
     + Reduce the cycle length during periods of low traffic to improve responsiveness.
3. **Continuous Feedback**:
   * The script runs iteratively, using real-time data at each step to refine the signal timing.
   * This ensures that the adjustments remain relevant as traffic conditions evolve.

**Dynamic Adaptation**

* **Peak Hours**: During heavy traffic, the algorithm extends green phases for heavily loaded directions to clear vehicle queues more efficiently.
* **Off-Peak Hours**: During light traffic, it reduces cycle lengths to minimize unnecessary waiting times at red lights.

**Advantages**

1. **Real-Time Optimization**: The heuristic algorithm provides quick adjustments based on current traffic conditions.
2. **Scalability**: It can be applied to various network sizes, from small intersections to city-wide networks.
3. **Computational Efficiency**: Since it avoids exhaustive computation, it is faster and suitable for real-time applications.