NA*viDrive: A CV-based Enhanced Intelligent Navigation Simulator

Introduction to Intelligent Vehicles, Final Project

李梁玉軒 B10902112

Abstract

This study presents the **implementation** and **research** of an advanced navigation simulator that integrates computer vision with simple path-finding algorithms and incorporates traffic flow theory. The simulator is designed to accurately extract edge lengths and detect congestion levels on a given map. By leveraging real-time data, the system dynamically evaluates and identifies the optimal route from a specified starting point to the desired destination. The approach combines recording processing techniques for map analysis with robust algorithms for path optimization, providing a comprehensive solution for efficient and effective navigation in varying environments. The results demonstrate the system's capability to adapt to real-world conditions and make informed routing decisions, highlighting its potential applications in autonomous navigation and urban planning.

Disclaimer

This report does not contain any double assignment or any completed work prior to this semester.

Contents

1 Introduction

In Lecture 08 - Sensing and Perception, we've explored a wide range of sensors, sparking our curiosity about their potential integration into intelligent vehicle systems. Traffic congestion is a common phenomenon in real life that must be considered in route navigation systems. Combining these insights, we have decided to design and implement a system that incorporates traffic cameras with path-finding algorithms for vehicle route planning.

2 Related work

In recent years, significant progress has been made in the fields of computer vision-based real-time vehicle tracking and traffic flow theory. However, these two domains have largely been explored independently, with limited efforts to integrate them for comprehensive traffic management solutions.
Computer Vision-Based Real-Time Vehicle Tracking. Numerous studies have focused on the development of real-time vehicle tracking systems using computer vision techniques. Using traffic cameras as sensors, Li et al.(2014)[6] proposed a method utilizing openCV to detect and track vehicles on lanes with high accuracy. It draws a set of rectangular regions of interest (ROIs) in each lane as the virtual detector in image space, and determines the presence of vehicles by monitoring changes in area of virtual detector.
Path-finding Algorithms. Daniel et al.(2021)[7] provided a systematic review of the basic concept of A* algorithm, as well as other advanced related path-finding algorithms that have been widely used or published these day. Integrating these kind of algorithms with a decent heuristic function can help us find the shortest path efficiently.

3 Method

We develop a real-time route navigation system for intelligent vehicles. This system takes several configurations and recordings from traffic cameras, along with an undirected graph representing the road map, as inputs. It calculates the estimated time to reach the destination, clearly visualized using the Python package Matplotlib. Specifically, the program plots the entire graph as a real-time updating map, highlighting the node our vehicle just passed with a distinct color. Additionally, it marks the goal node with another different color for clarity.

3.1 Framework and Architecture

The simulator is built on Python with PyTorch and OpenCV framework, as well as the famous state-of-the-art object detection model - You Only Look Once(YOLO), providing a solid performance on vehicle detection. We select the most up-to-date public version - YOLOv8.2, as our default model in implementation. Finally, we enable the matplotlib library to display the map and the vehicle's current location, along with an updating countdown that signals the estimated time of arrival. For visualization purpose, we do not turn off the OpenCV's object detection window - the simulator still show the real-time recordings of each traffic camera, with the rectangle on detected objects, but it can be turned off for better computer performance explicitly.

3.2 Implementation

Input Format. The simulator first takes an undirected graph as input, representing the map we will simulate on. It then prompts for a list of node names, typically landmarks or spots. After entering the names, it continues by asking for the distance between each pair of nodes, in meters. Next, you should associate each edge with a traffic camera recording. The configuration of each camera is also necessary. Considering the fact that traffic cameras are usually fixed, this information is stored in the file ”conf.dat,” which the program automatically loads to enhance efficiency. Once the configurations are loaded, the program prompts the user to specify a starting node and a destination node. Finally it starts simulating.

Simulation with Models. The simulation stage is closely integrated with OpenCV and various YOLO models from the series. We've extensively tested several YOLO open-source models using the PyTorch framework, observing their vehicle detection capabilities with various images. Among these models, we found that YOLOv8s is the most robust model considering the accuracy and time cost on vehicles, especially when provided with a predefined list of detection classes. Most importantly, it is easy to install and load while not taking too much time to inference. The further result of model testing will be included in the experiment section.

Recordings and Live Traffic Cameras. Recall that in our simulation model, each edge of the map utilizes a video recording as its input. In a real-world scenario, data from live traffic cameras would be continuously transmitted to a central computing facility, where sophisticated algorithms would perform object detection and calculate optimal paths in real-time. However, for the purposes of this simulation, we will employ pre-recorded videos as inputs. These recordings will be treated as live streams, providing a realistic approximation of actual traffic conditions. This approach serves as a provisional implementation, allowing us to develop and test the system in a controlled environment before integrating real-time data in future iterations.

Route-planning Algorithms. In path-finding, traditional algorithms like Dijkstra's Algorithm and A* Algorithm are effective when edge costs are constant and do not vary over time. However, for real-time route-replanning, it is necessary to replan the path whenever approaching an intersection. This ensures that the path from the starting node S to the destination node D is optimized to minimize the cost, taking into account dynamic factors such as traffic congestion. We adapt the A* Algorithm by introducing a self-defined "apparent distance," which has the following definition:

Definition 1. *The apparent distance is the adjusted distance between two points on a road network, accounting for the effects of traffic congestion. It reflects the increased travel time due to congestion, making it a crucial factor in dynamic path-finding algorithms.*

The apparent distance is calculated within the `weight_function` to account for the effects of traffic congestion on travel distance. The calculation proceeds as follows:

1. **Retrieve Parameters:**
   - `max_vehicle_length`: Maximum length of a vehicle, set to 5 meters.
   - `speed_limit`: Speed limit for the road segment, obtained from `conf.dat` using the video attribute of the edge.
   - `lanes`: Number of lanes on the road segment, also retrieved from `conf.dat`.
   - `cost_now`: Current congestion cost, calculated as the number of vehicles on the road divided by the captured lane length from `conf.dat`.
2. **Calculate Maximum Density:**
   - k_m: Maximum density of vehicles per unit length, calculated as the number of lanes divided by the maximum vehicle length.
3. **Determine Free-Flow Speed:**
   - v_f: Free-flow speed, set to the speed limit of the road segment.
4. **Calculate Real Speed:**
   - `v`: Real speed of vehicles on the road segment, determined by the function `real_speed` which takes into account the free-flow speed (v_f), current congestion cost (`cost_now`), and maximum density (k_m). We use the default formula from Underwood, R. T.[3] since it best meets the congestion case in our traffic recordings.
5. **Compute Apparent Distance:**

- Adjust the actual distance with the ratio of the free-flow speed to the real speed (v_f/v).

The formula for apparent distance is then given by:

$$\text{apparent\_distance} = \text{distance} \times \frac{v_f}{v}$$

where:

- distance is the actual distance of the edge.
- v_f is the free-flow speed (speed limit).
- v is the real speed of vehicles, which decreases as congestion increases.

By incorporating the congestion-adjusted speed, the apparent distance effectively represents the increased travel time due to traffic, this way, areas with higher congestion will have a longer "apparent distance" compared to their original distance. The A* algorithm's property is defined as $f(n) = g(n) + h(n)$, where:

- $g(n)$ is the "apparent distance" from the start node S to node n, factoring in congestion.
- $h(n)$ is the heuristic estimate of the distance from node n to the goal node D.

In our modification, $h(n)$ is the real (straight-line) distance to D, which serves as an underestimate of the true cost, considering congestion.
An algorithm is admissible if the heuristic $h(n)$ never overestimates the true cost to reach the goal node D.

Theorem 1. *The modified A* algorithm is admissible.*

Proof. Let $h^*(n)$ be the true cost from node n to the goal node D, considering the actual congestion. For admissibility, we need to show that for all nodes n,

$$h(n) \leq h^*(n)$$

Given $h(n)$ is the real (straight-line) distance, which is always less than or equal to the true cost $h^*(n)$ that includes congestion. Thus, the heuristic $h(n)$ is admissible. □

An algorithm is consistent (or monotonic) if, for every node n and every successor n' of n generated by any action a,

$$h(n) \leq c(n, n') + h(n')$$

where $c(n, n')$ is the step cost from n to n'.

Theorem 2. *The modified A* algorithm is consistent.*

Proof. Let n' be a successor of n. The consistency condition requires:

$$h(n) \leq c(n, n') + h(n')$$

Given:

- $h(n)$ is the real distance from n to D.
- $h(n')$ is the real distance from n' to D.
- $c(n, n')$ is the cost to move from n to n', factoring in congestion.

By the triangle inequality in Euclidean space, where $d(n, n')$ is the straight-line distance from n to n':

$$h(n) \leq d(n, n') + h(n')$$

Since $c(n, n')$ includes the effect of congestion and is at least $d(n, n')$,

$$h(n) \leq d(n, n') + h(n') \leq c(n, n') + h(n')$$

the heuristic $h(n)$ satisfies the consistency condition. □

By proving the admissibility and consistency of the modified A* algorithm, we ensure that the each call of the algorithm remains optimal with multiple recalculations in dynamic environments where edge costs change over time due to traffic congestion. The ”apparent distance” approach effectively converts congestion factors into additional distance, maintaining the reliability and efficiency of real-time route-replanning.

Timing Analysis of the Algorithm. The efficiency of the A* algorithm with the modified ”apparent distance” heuristic is crucial for real-time route-replanning. Timing analysis involves understanding the time complexity and factors that influence the execution time. The time complexity of the A* algorithm is primarily determined by the number of nodes expanded during the search process. This depends on several factors:

- **Heuristic Function:** The accuracy of the heuristic $h(n)$ greatly influences the efficiency. A more accurate heuristic reduces the number of nodes expanded. Our demo program use Dijkstra's Algorithm to get the real distance in heuristic function, which is not quite a good practice but, we will later discuss why we are prone to this.
- **Edge Costs:** Dynamic edge costs due to traffic congestion can increase the complexity, as the algorithm needs to frequently re-evaluate paths. But in real world, the processing center works as a server which gets data consistently from each sensors - worked as client. This means the updating of the data can be done in parallel, saving lots of time.
- **Graph Properties:** The size of the graph (number of nodes $|V|$ and edges $|E|$) directly impacts the time complexity. For a graph with $|V|$ nodes and $|E|$ edges, the worst-case time complexity is $O(|E| + |V| \log |V|)$ when using a priority queue for the open set.

To achieve real-time performance, several optimizations can be implemented:

- **Efficient Data Structures:** Utilizing efficient priority queue implementations (e.g., binary heaps, Fibonacci heaps) for managing the open set can significantly reduce the computational overhead. In our case, we usually store each information needed in computation in a dictionary, which uses hash techniques to achieve $O(1)$ retrieval time.
- **Incremental Updates:** Instead of recalculating paths from scratch at every intersection, incremental pathfinding techniques can be employed to update paths efficiently based on changes in edge costs. But since the data of congestion of each node will be updated in real-time, there can't be much effort to take.

Through theoretical analysis, we ensure that the modified A* algorithm with ”apparent distance” runs in $O(|D||T|(|E| + |V| \log |V|))$, where $|D|$ is the number of nodes the vehicle would passed in the end of simulation (it may turn back in some case, where some nodes would be counted twice or more) and $|T|$ is the time we retrieve the current distance to the destination in heuristic function. In our demo program, it takes as long as Dijkstra's Algorithm since it recalculate it using dijkstra path function at each call of heuristic function. This slow down the performance, but the map isn't fixed in each of our execution and for simplicity, we choose to do this online. However, in real-time scenarios, we can first retrieve a range of map from cloud and probably store it in a hash table. This offline method will significantly save our time.

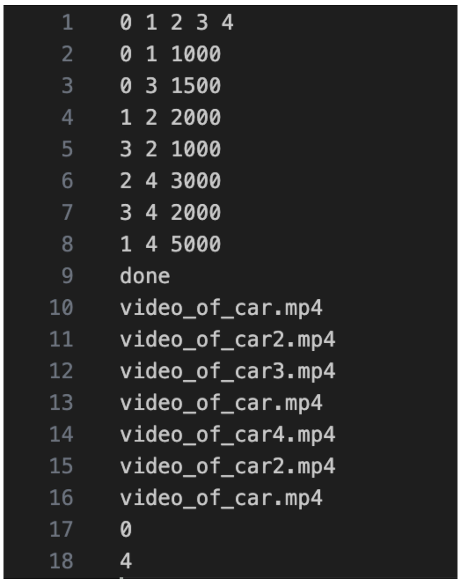
3.3 Interface and Demonstration

Here is a simple demonstration of the interface:

1. **Input and Configuration Format**:
    - Figure 1(a) shows the input format for our program. The first line consists of K names, each represented the name of each node as we previously described. After this there are N lines until the string ”*done*”, indicates the distance between each two nodes in the

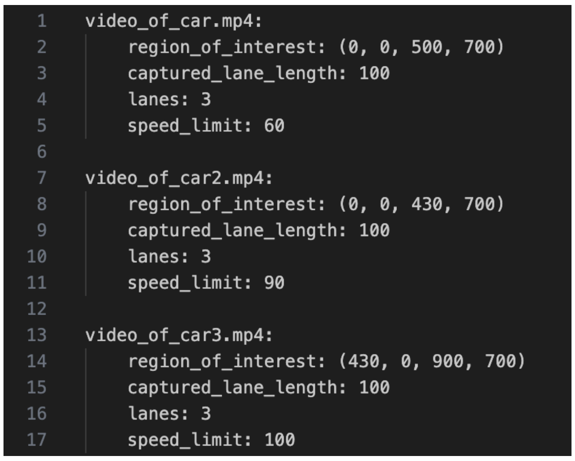
format ”*nodename nodename distance*(m)” if there is an edge. If an edge between two node doesn’t exist, simply don’t enter it. Then the program will ask for an assignment for recordings used by the congestion level detector on each edge. Of course, there are N of them. Finally, the last two line represents the starting nodename and the destination nodename.

- The configuration file of the cameras is shown in Figure 1(b). For each video name followed by a colon, there are four variables, each preceded by four spaces. First is the region of interest (ROIs), which should be set to the range of the road in the camera view with coordinates (x_1, y_1, x_2, y_2). The capture lane length (m) is the maximum detectable length for the camera, which will be related to the camera’s resolution and the object detection model’s performance. The next variable is lanes, which should be set to the number of lanes on the road. Lastly, speed limit represents the speed limit of the road. These four constants should be observed or looked up by a human before the system is enabled.



```
0 1 2 3 4
0 1 1000
0 3 1500
1 2 2000
3 2 1000
2 4 3000
3 4 2000
1 4 5000
done
video_of_car.mp4
video_of_car2.mp4
video_of_car3.mp4
video_of_car.mp4
video_of_car4.mp4
video_of_car2.mp4
video_of_car.mp4
0
4
```

(a) Input format



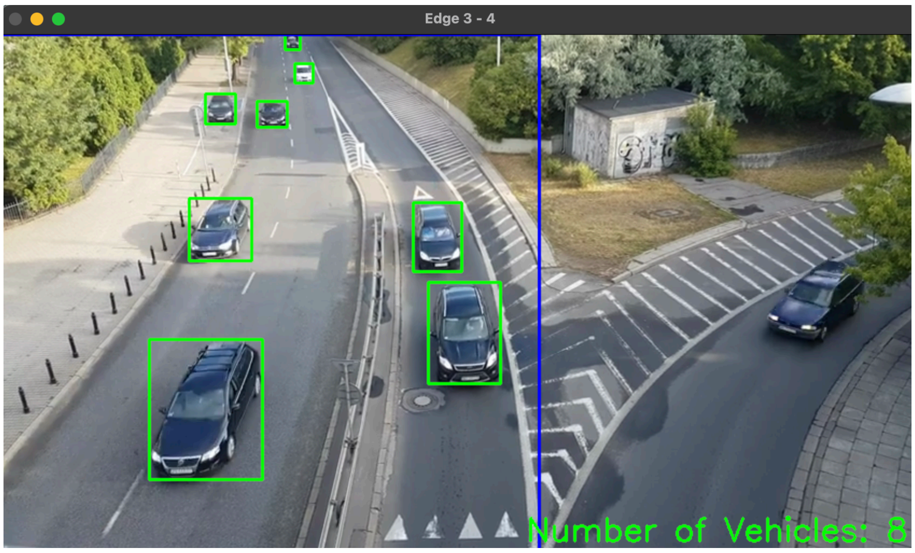
```
video_of_car.mp4:
    region_of_interest: (0, 0, 500, 700)
    captured_lane_length: 100
    lanes: 3
    speed_limit: 60

video_of_car2.mp4:
    region_of_interest: (0, 0, 430, 700)
    captured_lane_length: 100
    lanes: 3
    speed_limit: 90

video_of_car3.mp4:
    region_of_interest: (430, 0, 900, 700)
    captured_lane_length: 100
    lanes: 3
    speed_limit: 100
```

(b) Configuration file format

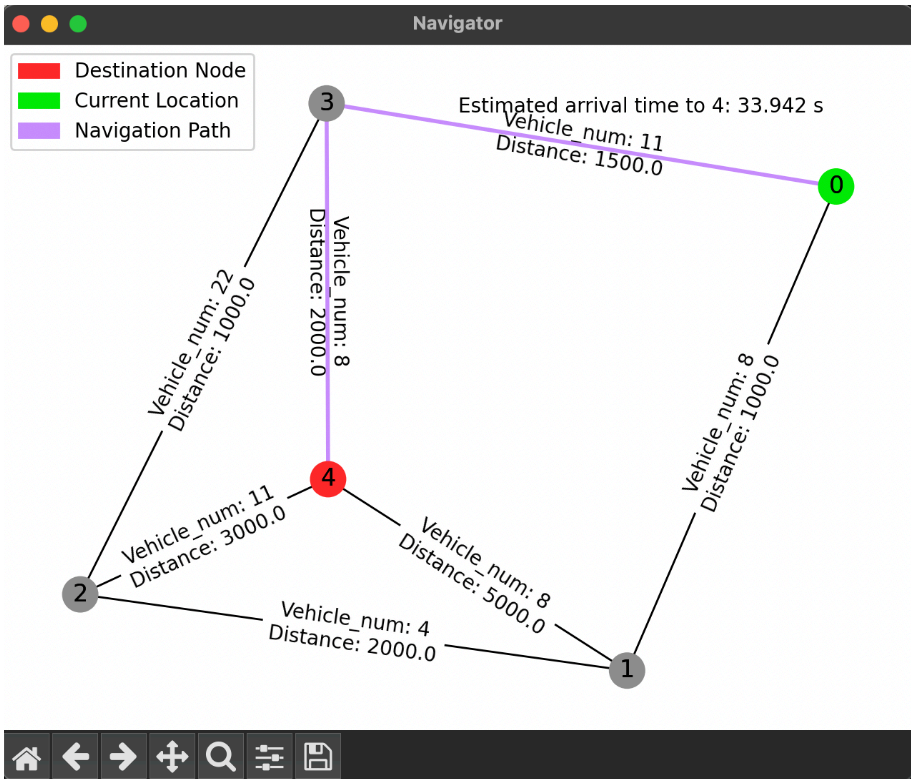
Figure 1: Input and configuration file formats

2. **Vehicle Detection Window**:
    - After getting the needed data, for each edge, a detecting window built by OpenCV automatically pops up like figure 2(a).
    - The above shows the name of the edges, and on the video part, the region specifies by our previously defined ROIs will be marked with a blue rectangle. By contrast, the detected vehicles will be marked with a green rectangle. The program counts the number of vehicles and show it to the bottom right corner.

3. **Navigator Window**:
    - The navigator window is built by matplotlib, a useful Python plotting library.
    - Figure 2(b) shows an overview of the navigator window. There is a graph indicating the map we considered from the input. The program marked the vehicle’s current node with green and destination node with red. Other nodes are simply marked with the default gray color.
    - For each edge, the distance and vehicle number is displayed, while vehicle number will be update in real-time. We currently use plt.pause() method to achieve it, not bothering to find a better or more efficient way.



(a) The detection window



(b) The navigator window

Figure 2: Detection and navigator windows

- On the upper right corner, an arrival time is estimated by the formula of single stage traffic flow theory models which will be described later, assuming we are driving at the maximum available ”real speed”. However, in real world scenarios, it can be simply estimated by the speed of the car itself with higher accuracy. This is the restriction we have to face while building a simulator.

4. **Console**:
   - The console logs the time the vehicle passing each node for observing or debugging purpose.

5. **Special System Requirements and Notices**:
   - Our program is designed to run on macOS, which take advantages of its metal performance shader. If the device does not support it, it by default use CPU for computation.
   - We use threading for a pseudo-parallel processing, mimicking the real world case that each data are processed in parallel.
   - Networkx is also imported for a more convenient algorithm design and construction.

Our system provides a comprehensive interface for users to input data, configure settings, visualize vehicle detection, and navigate through a simulated traffic network. It incorporates both automatic detection windows generated using OpenCV and navigator windows created with matplotlib, offering a real-time overview of vehicle positions and traffic conditions. With these features and considerations, our system aims to provide users with a robust simulator for navigating and analyzing traffic flow in various scenarios.

4 Experiment and Discussion

Below, we introduce our experiment involving the variation in the size of the You Only Look Once (YOLO) v8 model and different types of classic models from traffic flow theory.

4.1 YOLO models

One of the most important parts of computer vision is object detection, a technique that uses neural networks to localize and classify objects in images. We decided to use the You Only Look Once (YOLO) series models to help us deal with detecting vehicles in the camera. There are multiple different sizes of YOLO models available on different versions, with a trade-off between speed and accuracy. Models with larger sizes have slower computation speeds than smaller models, but their accuracy tends to be higher than that of smaller models. Table 1 shows part of our result on various open-source models over the kaggle public dataset provided by Sandeep[8] - where we compare YOLOv5[9] and YOLOv8[10], along with the newest YOLOv9[11], as they are related famous and published by the same company with similar structures. The model over 50MB will be considered too large, which has more cost on both CPU and GPU, as we need to strike a balance between real-time detection and accuracy, we should not consider them due to our computer performance. However, the smallest models in YOLOv9 series slightly exceed 50MB(but quite near), so we still consider them in our experiment. We first conduct the performance on accuracy and time consumption of 5 vehicle classes, just like we considered in our project. They are bicycles, cars, motorcycles, buses and trucks respectively.
From the resulting table, we found that YOLOv9 series models has the best average accuracy. However, the significant slow inference time marked the impossibility for it to be integrated in our real-time system, which detect the vehicle from frame to frame. Striking a balance between accuracy and inference time, we strongly recommend using the YOLOv8n or YOLOv8s models in our program, as they have relatively reasonable average performance.

Model	Accuracy (%)	Accuracy by Type (%)				
	Geometric Mean	Bicycles	Cars	Motorcycles	Buses	Trucks
YOLOv5n	39.53	37.70	41.84	48.06	80.91	49.55
YOLOv5s	63.09	48.36	54.28	67.13	86.89	65.30
YOLOv5m	74.85	60.66	65.49	82.18	92.02	78.18
YOLOv8n	68.89	54.10	57.15	72.90	92.31	74.60
YOLOv8s	75.17	59.84	65.49	83.69	93.45	78.35
YOLOv8m	80.96	71.31	69.88	89.03	94.33	83.15
YOLOv9c	81.24	68.85	71.10	89.34	96.01	84.26
YOLOv9c-seg	82.21	73.77	70.24	90.13	95.67	84.08

Table 1: Vehicle Detection Accuracy Comparison

Model	Time by Type (sec)				
	Bicycles	Cars	Motorcycles	Buses	Trucks
YOLOv5n	5	52	34	15	20
YOLOv5s	8	75	49	23	35
YOLOv5m	14	130	91	44	69
YOLOv8n	9	87	63	25	40
YOLOv8s	17	163	111	52	64
YOLOv8m	31	289	207	99	145
YOLOv9c	55	532	365	156	232
YOLOv9c-seg	65	621	393	164	274

Table 2: Vehicle Detection Inference Time

4.2 Traffic-Flow Theory

We evaluated five classic traffic flow models to determine the most suitable formula for our traffic recordings. However, assessing their suitability is challenging due to the absence of labeled speed data in our video dataset. Consequently, we must estimate vehicle speeds. The following results are based on our observations and hypotheses, and should be carefully examined before being integrated into different traffic systems.

4.2.1 Greenshields (1935)[1]: $v = v_f \left(1 - \frac{k}{k_j}\right)$

The formula $v = v_f \left(1 - \frac{k}{k_j}\right)$ is a key feature of traffic flow theory, where v represents the velocity of traffic flow, v_f is the free-flow velocity, k is the traffic density, and k_j is the jam density. It illustrates how traffic velocity decreases as traffic density approaches jam density, indicating the relationship between traffic flow and congestion.

4.2.2 Greenberg (1959)[2]: $v = v_m \ln\left(\frac{k_j}{k}\right)$

Greenberg's model describes a logarithmic relationship between traffic velocity and density, where v is the velocity, v_m is the maximum velocity, k is the traffic density, and k_j is the jam density. This model highlights the impact of increasing density on speed, particularly under conditions of higher traffic density, and suggests a different pattern compared to linear models.

4.2.3 Underwood (1961)[3]: $v = v_f \exp\left(-\frac{k}{k_m}\right)$

Underwood explored the exponential relationship between traffic speed and density, where v represents the traffic velocity, v_f is the free-flow velocity, k is the current congestion density, and k_m is the maximum density, when the traffic flow comes to a complete stop. This model indicates its applicability to lower type roadways but not expressway-type facilities. It introduces a generalized speed-volume diagram with three zones—normal flow, unstable flow, and forced flow—each defined in terms of probabilities. These zones offer insights into traffic flow theory and help reconcile existing models by depicting the transition from free-flow conditions to congested states.

4.2.4 Northwestern University (1965)[4]: $v = v_f \exp\left(-\frac{1}{2} \cdot \frac{k}{k_m}\right)$

The Northwestern University model is a modification of the Underwood model, incorporating a factor of $\frac{1}{2}$ into the exponential term. Here, v represents velocity, v_f is the free-flow velocity, k is the traffic density, and k_m is a constant related to the maximum density. This model provides an alternative perspective on the exponential relationship between speed and density, emphasizing moderate congestion levels.

4.2.5 Pipes (1967)[5]: $v = v_f \left(1 - \left(\frac{k}{k_j}\right)^n\right)$

Pipes' model generalizes the Greenshields model by introducing an exponent n. In this model, v is the velocity, v_f is the free-flow velocity, k is the traffic density, k_j is the jam density, and n is a parameter that can be adjusted to fit different traffic conditions. This model allows for greater flexibility in describing how velocity decreases with increasing density, accommodating various traffic scenarios.
We try out different kinds of models and want to find out which one is more appropriate for our simulation. For simplicity, we assume that the vehicles on each lane has identical behavior - they keep the same speed and share same congestion level. We notice that each formula performs quite the same when the traffic density is relatively low. Hence, we conduct our experiment on a severely serious traffic jam, which illustrated by Figure 3. In this case, most of our object

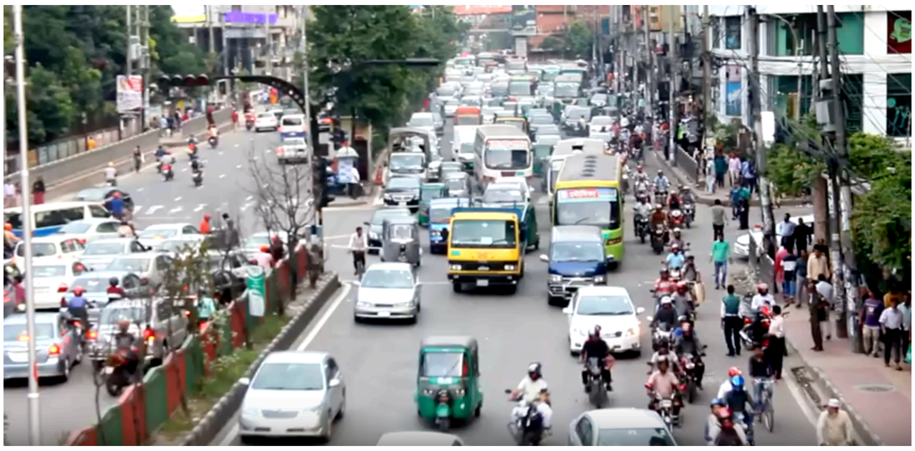


Figure 3: The horrible traffic jam

detection models failed to detect vehicles beyond the intersection in the figure due to severe traffic congestion. As a result, the captured lane length has been reduced to a significantly smaller value (set to 40 meters in this case). Upon applying the formula we collected earlier, we found that Greenshields' and Underwood's traffic flow model is particularly susceptible to congestion levels, undergoing significant changes. Other models such as Greenberg's and Pipes' are not as prone to such sensitivity. Figure 4 presents a comparison of these models in the situation assumed in Figure 3.

$$\begin{aligned}
v_f &= 50 && \text{(free flow speed in km/h)} \\
k_j &= 500 && \text{(jam density in vehicles per km)} \\
v_m &= 50 && \text{(maximum speed in Greenberg's model, for comparison)} \\
k_m &= 500 && \text{(maximum density in Underwood and Northwestern models)} \\
n &= 2 && \text{(exponent in Pipes' model)}
\end{aligned}$$

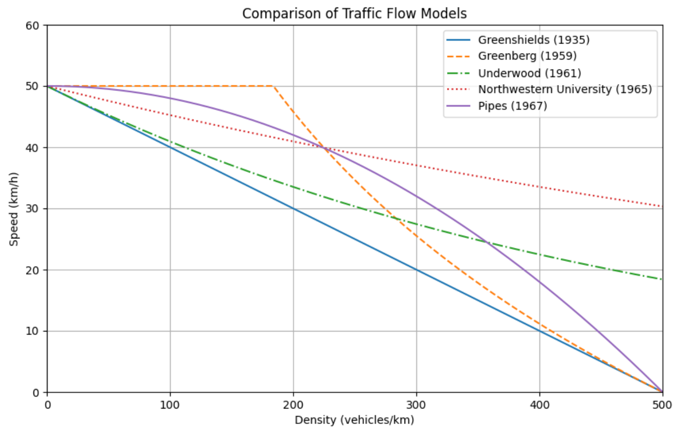


Figure 4: Comparison of models

The linear Greenshields model is not selected as the default model in our program due to its limitations in accurately representing traffic flow dynamics under congested conditions. While the Greenshields model offers simplicity and ease of use in modeling traffic flow, it assumes a linear relationship between traffic density and traffic speed. However, in real-world scenarios, traffic flow dynamics are often nonlinear, especially under heavy congestion. Underwood's model, on the other hand, is chosen as the default model because it accounts for nonlinearities in traffic flow dynamics, particularly in congested situations. This makes it more suitable for accurately capturing and predicting traffic behavior under varying traffic densities, which is crucial for our program's objectives.

5 Conclusion

In conclusion, this study proposed a novel approach to navigation simulation that leverages computer vision, path-finding algorithms, and traffic flow theory to create an adaptable and effective routing system. By integrating real-time data analysis with dynamic route optimization, the simulator demonstrates its ability to navigate through complex environments while considering factors such as edge lengths and congestion levels. The findings indicate promising applications in autonomous navigation systems and urban planning, where informed routing decisions are essential for optimizing traffic flow and enhancing overall efficiency. Moving forward, there are still lots of factors that our project hasn't concern. Further research and development in this area could lead to advancements in navigation technology, contributing to safer and more efficient transportation systems in urban areas.

6 Source Code

For the source code of our simulator, please refer to: https://github.com/nukkk97/iiv_final_project

References

[1] Bruce D. Greenshields, J. Rowland Bibbins, W.s. Channing, and Harvey H. Miller. A study of traffic capacity. 1935.

[2] Harold Greenberg. An analysis of traffic flow. *Operations Research*, 7(1):79–85, 1959.

[3] R. T. Underwood. Speed, volume, and density relationships. Technical report, Pennsylvania State University, Bureau of Highway Traffic, University Park, PA, United States, 16802, 1961.

[4] J. L. Drake, Joseph L. Schofer, and A. D. May. A statistical analysis of speed-density hypotheses. *Highway Research Record 154*, pages 53–87, 1966.

[5] Louis A. Pipes. An Operational Analysis of Traffic Dynamics. *Journal of Applied Physics*, 24(3):274–281, 03 1953.

[6] Da Li, Bodong Liang, and Weigang Zhang. Real-time moving vehicle detection, tracking, and counting system implemented with opencv. In *2014 International Conference on Information Science and Technology (ICIST)*. IEEE, 2014.

[7] Marchel Budi Kusuma Novita Hanafiah Eric Gunawan Daniel Foead, Alifio Ghifari. A systematic literature review of a* pathfinding. In *Procedia Computer Science*, pages 507–514. ELSEVIER, 2021.

[8] Sandeep. Vehicle data set. `https://www.kaggle.com/datasets/iamsandeepprasad/vehicle-data-set`, n.d.

[9] Glenn Jocher. Ultralytics yolov5, 2020.

[10] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics yolov8, 2023.

[11] Chien-Yao Wang and Hong-Yuan Mark Liao. YOLOv9: Learning what you want to learn using programmable gradient information. 2024.