A logo of a globe

Description automatically generated

הפקולטה להנדסה

המעבדה לעיבוד אותות

Multi-agent reinforcement learning of traffic routing in networks of autonomous vehicles

אלון רייכר

ליעד גם

פרויקט שנה ד' לקראת תואר ראשון בהנדסה

מנחה: ליאון רביב

מנחה אקדמי: פרופ' אמיר לשם

אוקטובר 2023

תוכן עניינים

[Abstract 1](#_Toc146033165)

[Thanks 3](#_Toc146033166)

[Presenting the Problem 5](#_Toc146033167)

[Theoretical Background 6](#_Toc146033168)

[Alternative Solutions 10](#_Toc146033169)

[Chosen Solution 11](#_Toc146033170)

[Results 14](#_Toc146033171)

[Shortest path vs Q-learning 20](#_Toc146033172)

[UML and chosen code 26](#_Toc146033176)

[Conclusions 29](#_Toc146033177)

[Ideas To Expand Upon 31](#_Toc146033178)

[Bibliography 33](#_Toc146033179)

[Appendix 35](#_Toc146033180)

[Appendix A 35](#_Toc146033181)

[Appendix B 35](#_Toc146033182)

[Appendix C 35](#_Toc146033183)

[Appendix D 37](#_Toc146033184)

# Abstract

In the modern era of the 21st century, with the always-increasing number of vehicles on the roads, getting from one point to another takes more time. That, in turn, leads to higher energy consumption, pollution, and traffic jams [[1]](#_Bibliography). One possible solution to the problem is using AI-driven cars (or- agents) to navigate more efficiently [[2]](#_Bibliography).

In this project, we created a simulation for implementing, testing, analyzing, and comparing different algorithms to solve the problem. We also provide an easy-to-use, realistic environment for testing different routing algorithms, extracting agents’ data, and being able to replay previous runs of the simulation.

The simulator we created is part of a larger project based on a research suggestion [[3]](#_Bibliography), which aims to help navigate autonomous cars in urban areas. This is the simulator, which the agents run on, and is used to collect data about times, routes, the learning process of the agents, and other statistics. Our runs and testing were mainly done using the Tel Aviv map provided by Open Street Map (OSM) with the osmnx python library.

The traffic is modeled by referring to the speed of each road and not to individual cars on the road since we created the simulation according to the philosophy of a macroscopic traffic flow model [[4]](#_Bibliography).

The project includes embedding different routing methods and collecting data about them. It focuses on the shortest path routing algorithm and compares it to a Q-learning method.

We conclude that using the shortest path routing algorithm, with the current implementation of the simulation and the q-learning algorithm, gives the best results.

# 

# Thanks

We would like to thank Li-on Raviv and Amir Leshem for guiding us throughout the year, helping us construct and organize the simulator, and the agents’ learning process, and consulting us on implementing various ideas.

We would also like to thank The Engineering faculty in Bar Ilan for providing the relevant courses for the project.

# Presenting the Problem

In major urban areas, managing traffic efficiently is a major challenge. Our project introduces a car traffic simulator that explores three distinct routing strategies: random, shortest path, and Q-learning. This simulator serves as a tool to allow us to analyze and compare how these strategies influence traffic flow, congestion patterns, and responses to road obstacles in a controlled environment.

We've designed three different approaches for the car's navigation algorithms. First, there's the "random" approach, where cars make decisions without a specific plan, much like taking turns without any knowledge about the path to the destination. Then there's the "shortest path" approach, where cars always follow the shortest route, like using a GPS without having any knowledge about traffic. Lastly, we have the "Q-learning" approach, where cars learn from past experiences to figure out the most efficient routes, even when facing unexpected situations.

However, it's not just about smooth driving. Our virtual city throws in challenges like traffic jams and road closures, just like what happens in real life. This is where our simulator becomes valuable. We can use it to see which approach handles these challenges the best.

Our role in this project is similar to researchers studying traffic behavior. By observing and comparing how each approach performs, we can determine which strategy is most effective under different circumstances.

# Theoretical Background

Traffic simulation plays a pivotal role in understanding and managing the complex dynamics of vehicular traffic within urban and transportation systems. By employing computational models, traffic simulation allows researchers, urban planners, and policymakers to analyze traffic patterns, assess the impact of infrastructure changes, and formulate effective traffic management strategies.

Dijkstra's Algorithm for Shortest Path:

Dijkstra's algorithm is a well-known method used to find the shortest path between two points in a graph. In the context of our car traffic simulator, Dijkstra's algorithm plays a significant role in the "shortest path" routing strategy. This strategy focuses on guiding vehicles along the most direct route from their starting point to their destination, aiming to minimize the distance traveled.

How Dijkstra's Algorithm Works:

**Initialization**: The algorithm starts by assigning a tentative distance value to each node in the graph, with the initial node's distance set to zero and all other nodes' distances set to infinity.

**Exploration**: The algorithm examines neighboring nodes, updating their tentative distances if a shorter path is discovered.

**Selection**: The node with the smallest tentative distance is chosen as the current node, and its neighbors are explored.

**Termination**: The algorithm selects nodes and explores their neighbors until the destination node is reached or all nodes have been visited.

**Backtracking**: Once the destination node is reached, the algorithm backtracks from the destination node to the starting node, tracing the shortest path.

In the context of our simulator, Dijkstra's algorithm provides a systematic and deterministic approach to route vehicles through the road network. This strategy ensures that vehicles follow the path that minimizes the distance traveled, which can be particularly beneficial for optimizing travel times and reducing congestion under normal traffic conditions.

Reinforcement learning

Reinforcement learning is a machine learning paradigm that involves training agents to make sequential decisions in an environment to maximize a cumulative reward. It draws inspiration from behavioral psychology, where individuals learn through trial and error by interacting with their surroundings.

In reinforcement learning, an "agent" interacts with an "environment" to learn how to take actions that lead to the most favorable outcomes over time. The agent doesn't require explicit instructions; instead, it learns by receiving feedback from the environment in the form of rewards or penalties based on its actions. The goal of reinforcement learning is to discover the optimal strategy, known as a policy, that guides the agent's decisions to maximize its long-term rewards.

The Learning Process in Reinforcement Learning:

1. The agent starts with little to no knowledge of the environment and how its actions affect outcomes.
2. It interacts with the environment, taking actions and receiving rewards based on those actions.
3. The agent uses the received rewards to update its knowledge and refine its strategy (policy) to make better decisions in the future.
4. Over time, through repeated interactions and learning, the agent improves its policy to achieve higher cumulative rewards.

More specifically, we chose Q learning which is a specific reinforcement learning algorithm.

Q-learning for Adaptive Routing:

Q-learning is a reinforcement learning technique used to make decisions in dynamic and uncertain environments. In the "Q-learning" routing strategy of our car traffic simulator, vehicles learn and adapt their routes based on past experiences and traffic conditions. This approach enables vehicles to make informed decisions, even in the face of changing circumstances and obstacles.

How Q-learning Work:

State space: The state space refers to the set of all possible states that the environment can be in. A state can be defined as a specific configuration or situation that the agent can perceive or be in within the environment at any given time. It encapsulates all the relevant information that the agent needs to make decisions.

Action Space: The action space represents the set of all possible actions that an agent can take in each state. These actions are the choices available to the agent at any point in the environment, allowing it to transition from one state to another.

Q-Values: Each state-action pair is associated with a Q-value, which represents the expected cumulative reward for taking that action from that state.

Exploration and Exploitation: Vehicles use a balance of exploration (trying new actions) and exploitation (choosing known high-reward actions) to learn optimal strategies.

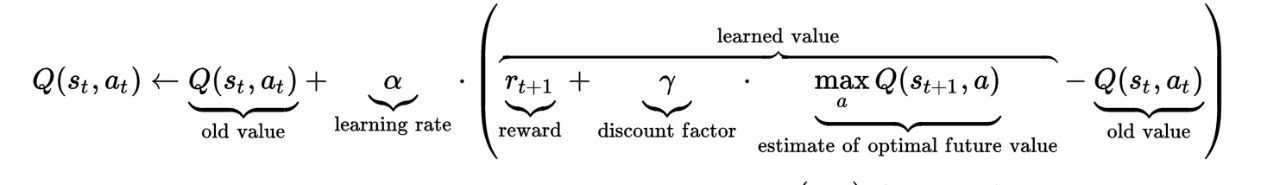
Learning rate: The learning rate determines to what extent newly acquired information overrides old information. It's a value between 0 and 1 that adjusts the weight given to new experiences compared to previously learned Q-values. A higher learning rate means the agent relies more on the most recent information, while a lower learning rate makes the agent consider past experiences more.

Discount factor: This factor determines the importance of future rewards in the agent's learning process. It's also a value between 0 and 1. A smaller discount factor means the agent prefers immediate rewards, while a higher discount factor encourages the agent to consider long-term rewards.

Epsilon greedy: This parameter controls how the agent explores the environment versus exploiting its current knowledge. During training, the agent needs to explore different actions to learn the environment better. The epsilon-greedy strategy allows the agent to choose a random action with probability epsilon and the action with the highest Q-value with probability

Updating Q-values: Q-values are updated using a learning rate and discount factor, incorporating rewards obtained from actions and future Q-values.

This is the Q learning update rule [[5]](#_Bibliography):



Learning and Adaptation: Over time, vehicles learn which actions lead to higher rewards in different states, allowing them to adapt their decisions to maximize their cumulative rewards.

Episodes in Q-learning: An "episode" refers to a complete run or cycle of interaction between an agent and its environment, starting from an initial state, taking actions, and ultimately reaching a terminal state. Each episode consists of a sequence of steps or actions that the agent performs until a specific condition is met, such as reaching a goal state, reaching a dead end or a maximum time limit.

Steps per episode: the number of actions or time-steps an agent takes within a single episode before the episode concludes. This parameter defines the duration or length of each learning episode. It's crucial to determine how much exploration or interaction an agent performs within the environment before an episode ends.

Q-learning can be effective for several reasons:

1. Adaptability to Dynamic Environments: Navigation often takes place in complex and dynamic environments, where conditions can change rapidly. Q-learning allows an agent to adapt its navigation strategy based on the feedback it receives from the environment. This adaptability is crucial for handling real-world scenarios such as traffic congestion, road closures, and unexpected obstacles.
2. Trial and Error Learning: Q-learning employs a trial-and-error approach, where the agent explores different actions and learns from the outcomes. This process allows the agent to discover which actions lead to better outcomes and adjust its behavior accordingly. In navigation, the agent can learn to avoid routes that consistently result in traffic jams or delays, leading to more efficient routing decisions over time.
3. Exploration and Exploitation: Q-learning balances the exploration of new paths and the exploitation of learned knowledge. This balance is vital for navigation, as it enables the agent to discover better routes while also leveraging its existing knowledge of effective paths. For example, the agent might explore new routes during periods of low traffic and exploit well-known routes during peak hours.
4. Long-Term Planning: Q-learning focuses on maximizing long-term cumulative rewards. In navigation, this means that the agent considers not only the immediate consequences of its actions but also the potential benefits or drawbacks of its choices over the entire route. This long-term perspective allows Q-learning to guide the agent towards routes that lead to the best overall outcomes.

Model-View-Controller

MVC (Model-View-Controller) is a concept use in software projects to create clear separation between the different parts of the project.

MVC separates the project into 3 different parts, the model, the view, and the controller.

Model: The Model encapsulates the application's data, business logic, and rules. It's the heart of the application where data is managed and processed. The Model is independent of the user interface (View) and doesn't concern itself with how the data is displayed or how users interact with it.

View: The View is responsible for the presentation layer. It's what users see and interact with. It presents the data from the Model in a visually understandable way. It can be graphical, textual, or any form relevant to the application.

Controller: The Controller acts as an intermediary between the Model and the View. It receives user inputs from the View, processes them, and decides how to interact with the Model. After processing, it updates the View with any changes in the data obtained from the Model. Contains application-specific logic that determines how user interactions affect the Model and View.

Interaction Flow:

* User interaction- the user interactes with the program's UI.
* Controller prosseccing- the view tranfer the interctations to the controller.
* Controller Operation: The Controller receives input, interacts with the Model to retrieve or modify data, and applies application logic.
* Model Update: The Model processes requests, updates data, and sends the result back to the Controller.
* View Update: The Controller updates the View with the modified data received from the Model.
* User Feedback: The updated View reflects changes, allowing the user to see the outcome of their interaction.

Benefits

* Separation of Concerns: Each component has a distinct role, making it easier to manage and modify specific functionalities.
* Reusability & Testability: Components can be reused across different parts of the application or tested independently for reliability.
* Scalability: Allows for parallel development by separating UI design, data handling, and application logic, making it easier to scale up the project.

# Alternative Solutions

For our navigation problem in a congested city, we had 2 alternative solutions:

1. Random navigation:

The first algorithm we implemented is a simple one, when a car needs to move to the next road the car simply chooses a random road and moves to the next road. This navigation model imitates a person who doesn’t know the city and just guesses where his destination is.

This is not a good algorithm because a most of the times the car doesn’t make it to the destination.

1. Shortest path navigation:

The second approach we tried was shortest-path navigation. This approach uses the Dijkstra algorithm for shortest path finding in a weighted graph.

Our graph's weights are the road's distances from one another, and they can be only positive, Dijkstra algorithm works on a non-negative weighted graph in contrast to Belman Ford, but Dijkstra is faster, that’s why we chose Dijkstra's algorithm.

This approach is faster and more reliable than random routing. On one hand, shortest path finds a path from the source to the destination if one exists. On the other hand, the algorithm doesn’t consider the dynamic nature of road traffic.

# Chosen Solution

Based on the weakness of the alternative solutions we chose the solution of Reinforcement Learning to be the navigation algorithm of the cars in the simulator.

The reinforcement learning algorithm we chose is Q-learning.

The Q-learning strategy in the context of navigation, empowers vehicles to learn from past experiences, such as traffic congestion and road closures. By adjusting their routes based on the learned Q-values, vehicles can dynamically navigate through the city, optimizing their paths to avoid road blockages and minimize travel times. This adaptive approach to routing is particularly useful for addressing challenges like congested streets and unexpected road closures, contributing to a more efficient and responsive traffic management strategy.

**Our Q-Learning algorithm works like this:**

1. Q-table Initialization

At the outset, the agent possesses no information about the quality of roads or paths in the environment. Thus, it begins with a blank Q-table. This Q-table represents a mapping of states (nodes in the map) to possible actions from each state. Initially, all Q-values in the table are set to zero.

1. Action Selection

Exploration: The agent decides whether to explore or exploit its current knowledge.

Exploration involves choosing a random action to gather new information about the environment.

Exploitation involves selecting the action that is believed to yield the highest reward based on past experiences stored in the Q-table.

1. Receiving a Reward

After selecting an action and traversing a road, the agent receives feedback in the form of a reward.

This reward is determined by several factors:

Proximity to the destination: Closer proximity to the goal yields a higher reward.

maximum speed: A higher road maximum speed might yield a higher reward.

1. Updating the Q-table

Based on the received reward, the agent updates the Q-value associated with the taken action at the current state.

The Q-value update is calculated using the Q-learning update rule, where the new Q-value is a blend of the previous value and the reward received, reflecting the agent's learning from the experience.

1. Iterative Learning Process

The agent repeats this process iteratively, exploring and learning from rewards over multiple episodes or iterations.

Through trial and error, the agent refines its knowledge and policy based on the observed rewards.

1. Learning from Mistakes

If certain actions from a particular state led to poor outcomes (low rewards), the agent learns from these mistakes and adjusts its strategy. It gradually avoids actions that result in unfavorable outcomes.

1. Achieving the Goal

As the agent accumulates experience and updates Q-values, it gradually learns the best sequence of actions (optimal path) from each node to reach the destination node.

Over time, the Q-values converge towards the optimal values, guiding the agent towards better decision-making and more efficient routes to the destination.

**Our implementation of Q-Learning:**

State - contains the current node ID that the agent is currently on (maybe additionally the last speed update time).

Action – action will be chosen from a list of the possible destination node IDs. at probability epsilon (by default 0.2 in the simulator) pick a random action, else pick the best action according to the q table.

Learning rate (Alpha) - we need to choose a value that is not too small (the q table won't update) and not too big ( this can lead to overfitting). we chose a default value of 0.1 for our agent.

The discount factor (Gamma) - determines the importance of future rewards. A factor of 0 will make the agent only consider current rewards while a factor approaching 1 will make it strive for a long-term high reward. We chose a discount factor of 0.9 by default to find a good balance between immediate and future rewards.

Reward function - to indicate the agent if the agent chooses a good or bad action, we initialize the reward function as such:

* 1000 if the agent arrived at the destination.
* -1000 if the agent arrived at a dead end.
* For every other road the reward was calculated as such:
  + -1 for every road the agent passed
  + -1 if the agent got further away from the destination
  + A small penalty is based on the road's maximum speed, as the road's max speed rise, the penalty lowers.
  + Return the sum of the reward and update the q table accordingly.

# Results

We ran some tests in our simulations to see how well the Q-learning routing algorithm works. First, we tried navigating from a random starting point to a destination. This helps us compare the Q-learning method's performance against just picking random paths.

For the Q Learning training, we set the following parameters:

Number of training episodes to 2000, the maximum number of steps in each episode to 100, the learning rate to 0.05, discount factor to 0.9, and epsilon to 0.2. The parameter's values were decided by trial and error, we tested and adjusted them until got them to a sweet spot where the agent's training wouldn't take too long but the agent will learn the optimal path.

For the simulation we use the map of Tel Aviv, the starting point is from "Rokach Boulevard" to "Dizengof Center".

After the agent's training finished, we got two diagrams that describe the course of the training:

1. תמונה שמכילה צילום מסך, עלילה, טקסט, קו

   התיאור נוצר באופן אוטומטיBar chart of travel time in each episode and an indicator if the agent made it to the destination:

We can see that at first, the agent mostly hadn’t reached the destination but as training continued the agent got to the destination and on a more optimized path. We can see that by the number of seconds being lower as the episode numbers grow.

1. תמונה שמכילה טקסט, קו, עלילה, צילום מסך

   התיאור נוצר באופן אוטומטיMean rewards over several episodes:

the x-axis is the mean rewards over the last 50 episodes, and we can see that the reward of the agent is growing as we train it more and more.

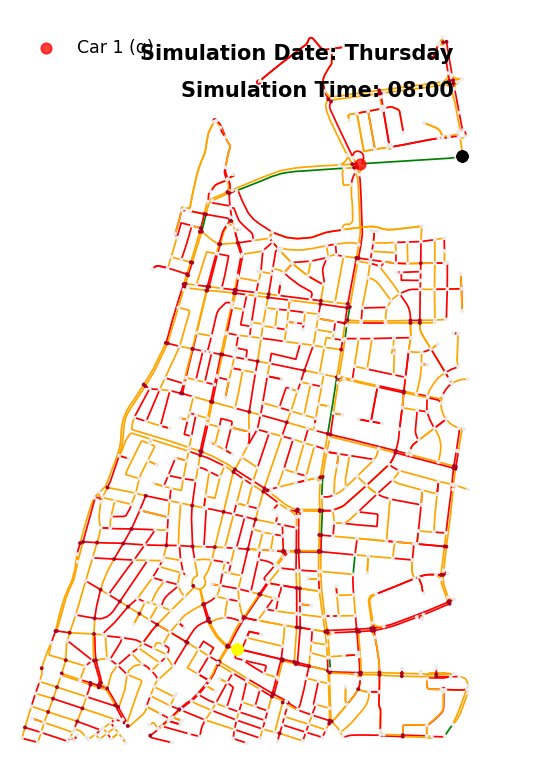
Now let’s look at the result, the agent's route:

תמונה שמכילה תרשים, תוכנית, שרטוט, מפה

התיאור נוצר באופן אוטומטי

The black mark indicates the starting point of the agent, and the yellow mark indicates the ending point.

Additionally, we have an animation of the agent's drive in real-time with the current time road's speeds whereas: red – heavy traffic, yellow- - medium traffic, and green–low traffic:

תמונה שמכילה טקסט, תרשים, מפה

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט, תרשים, מפה

התיאור נוצר באופן אוטומטיתמונה שמכילה טקסט, תרשים, מפה

התיאור נוצר באופן אוטומטי

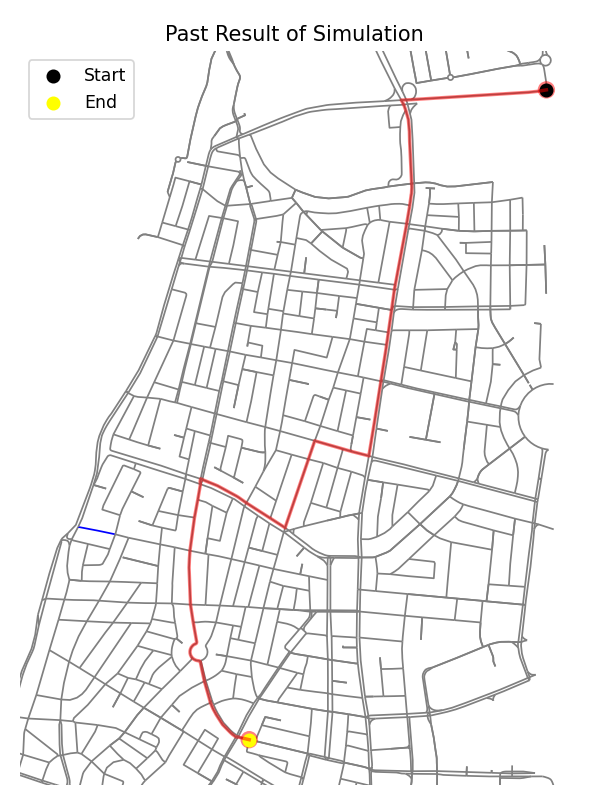
We can see the red dot, representing the q-learning agent, navigates from the starting point to the destination. Additionally, we have the day and the time of the simulation. The agent completed this drive in 19 minutes.

תמונה שמכילה צילום מסך, עלילה, צבעוני, קו

התיאור נוצר באופן אוטומטיNow, to test the q-learning algorithm let’s make a scenario, let’s say that on the following Sunday morning, the agent wanted to make the same route but this time a road in the original route that the agent took was blocked this morning, let’s see how well the agent adapt:

תמונה שמכילה קו, עלילה, צילום מסך, תרשים

התיאור נוצר באופן אוטומטי



We can see that there is a road colored blue in the map, this indicates that the road was blocked. Now we can see that the agent knew the road was blocked and chose a different route this time to avoid the closed road. This route took the agent 22 minutes to complete, so this route wasn’t chosen in the first simulation.

# Shortest path vs Q-learning

To compare the two algorithms to each other, we ran 100 simulations with 5 agents in each. (full code in appendix D)  
The method was as such:   
First, we chose 5 random pairs of source and destination (for each agent) \*, and 5 random starting time, whereas all the agents start their course on the same day, but at a random hour (from 0 to 22) and a random minute.

Then, we simulated with agents that utilized the shortest path algorithm.

Next, we simulated with the same parameters and agents but changed their routing algorithm to be q-learning (without using previous q-tables).

This is how each of the 100 iterations went.

\*The lists of possible source and destination nodes were chosen manually from the TLV map of the project and were manually put into a list in the Getters.py file in the utilities folder.

The nodes are divided into 4 groups, as seen in the next picture: top left (green), top right (blue), bottom left (red), and bottom right (purple). Top left nodes were paired with bottom right nodes, and top right nodes were paired with bottom left nodes.

A map of a city

Description automatically generated

The results we got are as such: **(sp = shortest path, q = q-learning**

# 

A graph of a running graph

Description automatically generated with medium confidenceA graph of a graph

Description automatically generated with medium confidence

# A screenshot of a computer program Description automatically generated

As we can see, the q-learning algorithm took less time to run than the shortest path algorithm (without any observable relation), giving no better results than the former.

That said, the sp time / q time ratio histogram shows a more comprehensive picture: while the q-learning navigation itself is not faster than the shortest path navigation, it gives very close results.

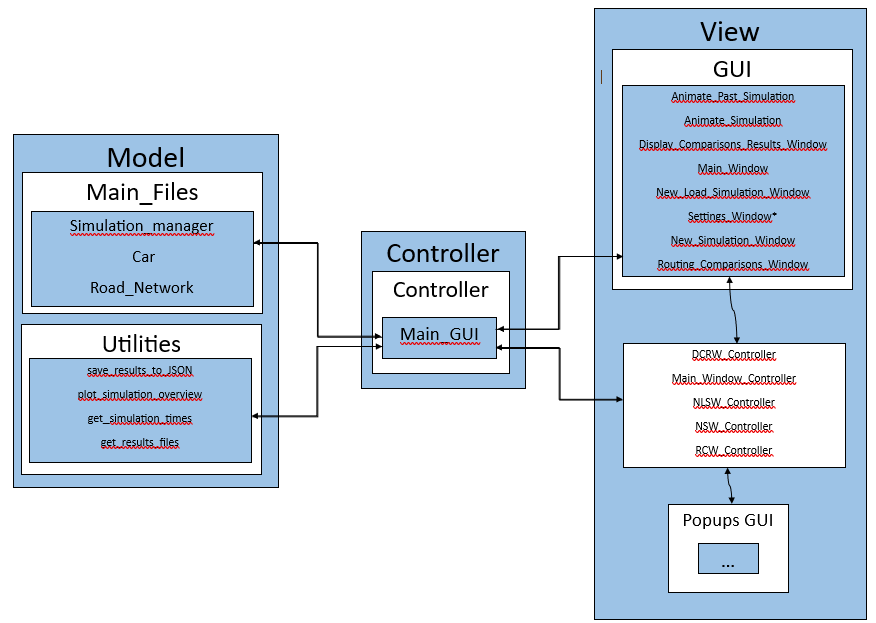
The q-learning also seems to run faster than the shortest path algorithm (once the q-tables are made), usually around 6-8 times faster, as seen in the sp run time/q run time histogram. In only 4 out of 100 simulations, the shortest path algorithm navigation ran faster than the q-learning navigation (for the same setting).

# UML and snippets of the code

The project was created in the MVC architecture (Model, View, Controller). It is also made of a few thousand code lines of code, thus we will present only the UML of the model (where the main logic happens), the MVC architecture of our project and chosen snippets of code from it. Documentation is available in the Simulator HLD appendix.

A diagram of a computer

Description automatically generated**The UML (also in the HLD appendix)**



The project was written in MVC architecture which means we have Model, Controller and View.

The Model has the classes required to run the simulator, this classes handle all the calculations and the logic behind the project. There are some additional classes in the Utillities package that consists of some general functions to help organize the code and make it easier to read.

The view consists of all the classes required to make the simulator work visually, the classes oversee the different GUI windows and the showing the simulation visually.

The controller consists of one main class that is called Main\_GUI, this class act as a middle man between the Model and the View. Every time the user click on a window of the program, then the view send the updated info to the controller which in return send it to the model and so on.

Shortest path algorithm code: in the appendix section (C).

Explanation for the shortest path algorithm code:  
The distances matrix is constructed using the shortest path algorithm of the network library (which by default uses the Dijkstra algorithm). The matrix’s line index is the current node ID, the column index is the destination node ID, and the value in each entry is the node ID of the next node in the shortest path. The agent gets its next node from the matrix, and if the next road is blocked, the agent tries to go through the next best option from its current node.

# Conclusions

We observed that with the current parameters of the Q learning algorithm, the best result we get converges to the shortest path algorithm, which is a bit unexpected.

The conclusion we got is that- the function we used for Q learning might not be the optimal one for getting the fastest route, under the current implementation of the simulation.

That said, the Q-learning is not far off the shortest path algorithm’s results, which could mean that minor tweaks to the function’s reward calculation or learning parameters could yield results that are on par or better than the shortest path algorithm.

# Ideas To Expand Upon

We suggest the following ideas for possible changes or to expand upon in the future:

* Adding other algorithms and methods to navigate, such as policy-based learning.
* Tweaking the settings for the Q learning may yield better results- changing the reward function, number of episodes, steps per episode, etc.
* Changing the traffic model (to a microscopic or a mesoscopic model) may yield better results, with the cost of performance.
* Changing the way that synthetic data is made to represent better the real world- for example, considering that cars that drive through a road will get to the adjacent road and affect its speed.
* Adding an algorithm for finding the fastest route for each trip and comparing it to the routes found by other algorithms could help with analyzing the results and give a better way to estimate other solutions.

# Bibliography

[1] Shabbar Ali, M., Adnan, M., Noman, S. M., & Baqueri, S. F. (2014, September 6). *Estimation of Traffic Congestion Cost-A Case Study of a Major Arterial in Karachi.* Retrieved from ScienceDirect: <https://www.sciencedirect.com/science/article/pii/S1877705814010078>

[2] Bischoffa, J., & Maciejewski, M. (2016, May 12). *Simulation of City-wide Replacement of Private Cars with Autonomous Taxis in Berlin.* Retrieved from ScienceDirect: <https://www.sciencedirect.com/science/article/pii/S1877050916301442>

[3] Leshem, A., & Sung, Y. (n.d.). Deep multi-agent reinforcement learning of Markov games for distributed traffic management in smart cities.

[4] *Macroscopic traffic flow model*. (n.d.). Retrieved from Wikipedia: <https://en.wikipedia.org/wiki/Macroscopic_traffic_flow_model>

[5] *Q-learning*. (n.d.). Retrieved from Wikipedia: <https://en.wikipedia.org/wiki/Qlearning>

# Appendix

## Appendix A

"Software Requirements Specification" – more specific software descriptions of the program (separate file).

## Appendix B

"Simulator HLD" – provides information about configurable parameters, used data, and user documentation of all the classes in the project (separate file).

## Appendix C

Shortest path algorithm code:

Outer layer in Route.py

def decide\_first\_road(self):  
 if self.src\_node == self.dst\_node:  
 return None  
 first\_road = self.road\_network.get\_next\_road\_shortest\_path(self.src\_node, self.dst\_node)  
 self.current\_node = first\_road.destination\_node.id  
 self.path.append(self.current\_node)  
 return first\_road  
  
  
def get\_next\_road(self):  
 if self.current\_node == self.dst\_node:  
 return None  
 next\_road = self.road\_network.get\_next\_road\_shortest\_path(self.current\_node, self.dst\_node)  
 self.current\_node = next\_road.destination\_node.id  
 self.path.append(self.current\_node)  
 return next\_road  
  
def get\_alt\_road(self):  
 self.path.pop()  
 self.current\_node = self.path[-1]  
 print("get\_alt\_road")  
 adjacency\_list = self.road\_network.node\_connectivity\_dict[self.current\_node] # list of all the adjacent nodes ids  
 for next\_node in adjacency\_list:  
 road = self.road\_network.get\_road\_from\_src\_dst(self.current\_node, next\_node) # this is "road"  
 if not road.is\_blocked:  
 # if the road is not blocked, we check if the any road on the shortest path is blocked  
 # if not, we return the road  
 if self.road\_network.get\_shortest\_path(next\_node, self.dst\_node) is not None:  
 self.current\_node = next\_node  
 return road  
  
 return None

The construction of the distance matrix in the Road\_Network.py class:

def add\_shortest\_path\_to\_matrix(self, src\_id: int, dst\_id: int):  
 *"""  
 Add the shortest path between two nodes to the distances matrix.  
  
 Args:  
 src (int): Source node ID.  
 dest (int): Destination node ID.  
  
 Returns:  
 None  
 """* path = nx.shortest\_path(self.nx\_graph, src\_id, dst\_id, weight='length')  
 path\_length = nx.shortest\_path\_length(self.nx\_graph, src\_id, dst\_id, weight='length')  
  
 #updating the distances matrix  
 previous\_edge\_length = 0  
 for i,node in enumerate(path[:-1]):  
 self.next\_node\_matrix[node][path[-1]] = path[i + 1]#adds the relevant next node to the distances matrix  
 self.distances\_matrix[node][path[-1]] = path\_length - previous\_edge\_length  
 previous\_edge\_length += self.get\_road\_from\_src\_dst(node,path[i + 1]).length  
 return

## Appendix D

Code in Statistice\_for\_project\_book.py – the script file used to generate the simulation and gather running times and simulation times:

import datetime  
import json  
import random  
import time  
  
from Main\_Files import Car  
import Simulation\_manager  
import GUI.Animate\_Simulation as AS  
from Utilities.Getters import get\_random\_src\_dst  
from Utilities.Results import save\_results\_to\_JSON, read\_results\_from\_JSON, car\_times\_bar\_chart, \  
 print\_simulation\_results, plot\_past\_result, get\_simulation\_times  
from Utilities import Getters  
  
START\_TIME1 = datetime.datetime(year=2023, month=7, day=2, hour=0, minute=0, second=0)  
START\_TIME2 = datetime.datetime(year=2023, month=6, day=2, hour=0, minute=0, second=0)  
START\_TIME3 = datetime.datetime(year=2023, month=6, day=2, hour=0, minute=0, second=0)  
START\_TIME4 = datetime.datetime(year=2023, month=6, day=2, hour=0, minute=0, second=0)  
START\_TIME5 = datetime.datetime(year=2023, month=6, day=2, hour=0, minute=0, second=0)  
  
ADD\_TO\_DAY = [0, 1, 2, 3, 4, 5, 6] # to cover every day of the week  
  
ADD\_TO\_HOUR = [i for i in range(0, 22)] # to cover every hour of the day from 0:00 to 22:00  
ADD\_TO\_MINUTE = [i for i in range(0, 60)] # to cover every minute of the hour  
  
# Constants for time intervals  
WEEK = 604800  
DAY = 86400  
HOUR = 3600  
MINUTE = 60  
  
# Simulation parameters  
NUMBER\_OF\_SIMULATIONS = 1  
TRAFFIC\_LIGHTS = False  
TRAFFIC\_WHITE\_NOISE = False  
Rain\_intensity = 0 # 0-3 (0 = no rain, 1 = light rain, 2 = moderate rain, 3 = heavy rain)  
  
# Q-Learning parameters  
USE\_ALREADY\_GENERATED\_Q\_TABLE = False  
NUM\_EPISODES = 2000  
  
# Animation parameters  
ANIMATE\_SIMULATION = False  
REPEAT = False  
SIMULATION\_SPEED = 10 # X30 faster than one second interval  
  
PLOT\_RESULTS = False  
  
NUM\_OF\_RUNS = 100  
PLACE\_NAME = 'TLV'  
NUM\_OF\_CARS = 5  
  
ALGORITHMS = ["q", "sp"]  
SP\_IND = 1  
Q\_IND = 0  
  
# nodes from list in place i will be one endpoint of the route  
# nodes from list in place 3-i will be the other endpoint of the route  
NODES = [Getters.bottom\_left\_nodes, Getters.bottom\_right\_nodes, Getters.top\_left\_nodes, Getters.top\_right\_nodes]  
  
def checkif\_path\_is\_exist(src,dst,RN):  
 try:  
 path = RN.get\_shortest\_path(src, dst)  
 return True  
 except:  
 return False  
  
  
def choose\_random\_src\_dst():  
 src\_list = random.randint(0, len(NODES) - 1)  
 dst\_list = len(NODES) - 1 - src\_list  
 src = random.choice(NODES[src\_list])  
 dst = random.choice(NODES[dst\_list])  
 return src, dst  
  
  
def create\_time\_delta(days):  
 hour = random.choice(ADD\_TO\_HOUR)  
 minute = random.choice(ADD\_TO\_MINUTE)  
 time\_delta = datetime.timedelta(days=days, hours=hour, minutes=minute)  
 return time\_delta  
  
  
  
def generate\_cars(existing\_settings, algorithm\_ind, RN):  
 # cars are in the same road network, same day, different starting times, different src and dst  
 cars = []  
 day = random.choice(ADD\_TO\_DAY)  
 for i in range(NUM\_OF\_CARS):  
  
 time\_delta = create\_time\_delta(day)  
 src, dst = choose\_random\_src\_dst()  
 while not checkif\_path\_is\_exist(src,dst,RN):  
 src, dst = choose\_random\_src\_dst()  
 while (src, dst, time\_delta, algorithm\_ind) in existing\_settings: # make sure there are no duplicates  
 time\_delta = create\_time\_delta(day)  
 existing\_settings.append((src, dst, time\_delta))  
 cars.append(Car.Car(i, src, dst, START\_TIME1 + time\_delta, RN, route\_algorithm=ALGORITHMS[algorithm\_ind],  
 use\_existing\_q\_table=USE\_ALREADY\_GENERATED\_Q\_TABLE))  
 return cars  
  
def print\_algorithms\_success\_rate(SM\_results):  
 total\_q = NUM\_OF\_CARS\*NUM\_OF\_RUNS  
 total\_sp = NUM\_OF\_CARS\*NUM\_OF\_RUNS  
 q\_success = 0  
 sp\_success = 0  
 for i in range(NUM\_OF\_RUNS):  
 for j in range(NUM\_OF\_CARS):  
 if SM\_results[i][j][ALGORITHMS[Q\_IND]][Getters.Reached\_destination]:  
 q\_success += 1  
 if SM\_results[i][j][ALGORITHMS[SP\_IND]][Getters.Reached\_destination]:  
 sp\_success += 1  
 print("q success rate: ", q\_success/total\_q)  
 print("sp success rate: ", sp\_success/total\_sp)  
  
  
def change\_route\_algorithm(cars, algorithm\_ind):  
 for car in cars:  
 car.set\_new\_routing\_algorithm(ALGORITHMS[algorithm\_ind])  
  
def get\_cars\_times(SM):  
 times = {} # format: key = simulation index, value = list of times of cars in the simulation and algorithm  
 for i, result in enumerate(SM.simulation\_results):  
 times[i] = []  
 for array\_index, result\_dict in result.items():  
 if array\_index != Simulation\_manager.Simulation\_number:  
 times[i].append(result\_dict[Getters.Time\_taken])  
  
def organize\_simulation\_times(times):  
 organized\_times = {} # key = simulation index, value = list of times of cars in the simulation and algorithm  
 # exery even index is shortest path, every odd index is q learning  
 for i in range (0, int(len(times)/NUM\_OF\_CARS)):  
 organized\_times[i] = []  
 for j in range(0,NUM\_OF\_CARS):  
 organized\_times[i].append(times[i\*NUM\_OF\_CARS+j])  
 return organized\_times  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 out\_times\_data = [] # format: key= run number and algorithm, value = time for full simulation  
 SM = Simulation\_manager.Simulation\_manager(PLACE\_NAME, 7 \* DAY, TRAFFIC\_LIGHTS, Rain\_intensity, TRAFFIC\_WHITE\_NOISE,  
 PLOT\_RESULTS, START\_TIME1)  
 RN = SM.road\_network  
 used\_settings = []  
 run\_time\_data = {} # format: key= run number value = time for full simulation in seconds and algorithm  
  
 # main loop  
 for i in range(NUM\_OF\_RUNS):  
 print("\*\*\*\*\*run number: ", i,"\*\*\*\*\*")  
 run\_time\_data[i] = {}  
 cur\_cars = generate\_cars(used\_settings, SP\_IND, RN)  
 start\_sp = time.time()  
 SM.run\_full\_simulation(cur\_cars, NUMBER\_OF\_SIMULATIONS, num\_episodes=NUM\_EPISODES, max\_steps\_per\_episode=100)  
 end\_sp = time.time()  
 run\_time\_data[i][ALGORITHMS[SP\_IND]] = end\_sp - start\_sp  
 change\_route\_algorithm(cur\_cars, Q\_IND)  
 start\_q = time.time()  
 cur\_learning\_time = SM.run\_full\_simulation(cur\_cars, NUMBER\_OF\_SIMULATIONS, num\_episodes=NUM\_EPISODES, max\_steps\_per\_episode=100)  
 end\_q = time.time()  
 run\_time\_data[i][ALGORITHMS[Q\_IND]] = end\_q - start\_q - cur\_learning\_time  
 run\_time\_data[i]["learning\_time"] = cur\_learning\_time  
 # SM.simulation\_results = read\_results\_from\_JSON(SM.graph\_name)  
 # times = get\_simulation\_times(SM)  
  
 json\_name = save\_results\_to\_JSON(SM.graph\_name, SM.simulation\_results)  
 print(organize\_simulation\_times(get\_simulation\_times(SM)))  
 print(run\_time\_data)  
 json.dump(run\_time\_data, open("run\_time\_data.json", 'w')  
 , indent=4)  
 json.dump(organize\_simulation\_times(get\_simulation\_times(SM)), open("times\_data.json", 'w'), indent=4)  
 # print\_algorithms\_success\_rate(SM.simulation\_results)  
 pass

Visualize\_statistics\_for\_project\_book.py – the script file used to analyze the simulation results, visually and mathematically.

import statistics  
  
import matplotlib.pyplot as plt  
import json  
import Utilities.Getters as Getters  
import osmnx as ox  
from Main\_Files import Road\_Network  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with open('run\_time\_data.json', 'r') as infile:  
 new\_simulation\_results = json.load(infile)  
 sp\_run\_times = []  
 q\_run\_times = []  
 q\_learning\_times = []  
 for run in new\_simulation\_results:  
 sp\_run\_times.append(new\_simulation\_results[run]["sp"])  
 q\_run\_times.append(new\_simulation\_results[run]["q"])  
 q\_learning\_times.append(new\_simulation\_results[run]["learning\_time"])  
  
 # avarage run times and standard deviation  
 sp\_avg = sum(sp\_run\_times) / len(sp\_run\_times)  
 q\_avg = sum(q\_run\_times) / len(q\_run\_times)  
 q\_learning\_times\_avg = sum(q\_learning\_times) / len(q\_learning\_times)  
 sp\_std = statistics.stdev(sp\_run\_times)  
 q\_std = statistics.stdev(q\_run\_times)  
 q\_learning\_times\_std = statistics.stdev(q\_learning\_times)  
 print("sp run times in seconds is %.3f" % sp\_avg, "+-%.3f" % sp\_std, " standard deviation")  
 print("q run times in seconds is %.3f" % q\_avg, "+-%.3f" % q\_std, " standard deviation")  
 print("q learning times in seconds is %.3f" % q\_learning\_times\_avg, "+-%.3f" % q\_learning\_times\_std, " standard deviation")  
  
 # plot run times with sp as x axis and q as y axis  
 plt.plot(sp\_run\_times, q\_run\_times, 'o')  
 plt.xlabel("sp run times in seconds")  
 plt.ylabel("q run times in seconds")  
 plt.show()  
  
 # car's driving times: sp vs q learning  
 with open('times\_data.json', 'r') as infile:  
 times\_data = json.load(infile)  
 sp\_times = []  
 q\_times = []  
 sp\_times\_by\_sim = []  
 q\_times\_by\_sim = []  
 # organize times by algorithm. every even index is sp, every odd index is q.  
 # every index is a list of times of cars in the simulation, currently when writing this code  
 # and comment-the simulation is with 5 cars.  
 for run in times\_data:  
 if int(run) % 2 == 0:  
 sp\_times\_by\_sim.append(times\_data[run])  
 for time in times\_data[run]:  
 sp\_times.append(time)  
 continue  
 q\_times\_by\_sim.append(times\_data[run])  
 for time in times\_data[run]:  
 q\_times.append(time)  
  
 # average times and standard deviation  
 sp\_avg\_drive\_time = sum(sp\_times) / len(sp\_times)  
 q\_avg\_drive\_time = sum(q\_times) / len(q\_times)  
 sp\_std\_drive\_time = statistics.stdev(sp\_times)  
 q\_std\_drive\_time = statistics.stdev(q\_times)  
 print("sp drive times in seconds is %.3f" % sp\_avg\_drive\_time, "+- %.3f" % sp\_std\_drive\_time)  
 print("q drive times in seconds is %.3f" % q\_avg\_drive\_time, "+-%.3f" % q\_std\_drive\_time)  
 print("\*" \* 50)  
  
 # calculate how many times Q learning was faster than SP  
 q\_faster\_count = 0  
 q\_equal\_count = 0  
 sp\_fast\_count = 0  
 for i in range(len(sp\_times)):  
 if sp\_times[i] > q\_times[i]:  
 q\_faster\_count += 1  
 elif sp\_times[i] == q\_times[i]:  
 q\_equal\_count += 1  
 else:  
 sp\_fast\_count += 1  
 print("q navigation was faster than sp navigation", q\_faster\_count, " times")  
 print("q navigation was equal to sp navigation", q\_equal\_count, " times")  
 print("sp navigation was faster than q navigation", sp\_fast\_count, " times")  
 print("\*" \* 50)  
  
 # scatter q times vs sp times  
 x\_values = []  
 for i in range(len(sp\_times)):  
 x\_values.append(i)  
 plt.plot(x\_values, sp\_times, 'o', label="sp")  
 plt.plot(x\_values, q\_times, 'o', label="q")  
 plt.xlabel("agent time index in the times array")  
 plt.ylabel("time in seconds")  
 plt.title("sp vs q learning times")  
 plt.legend()  
 plt.show()  
  
 plt.cla()  
 plt.clf()  
 plt.close()  
  
 q\_to\_sp\_time\_ratio = []  
 for i in range(len(sp\_times)):  
 q\_to\_sp\_time\_ratio.append(sp\_times[i] / q\_times[i])  
  
 plt.hist(q\_to\_sp\_time\_ratio, bins=20, edgecolor='black', linewidth=1.2)  
 plt.xlabel("sp time / q time")  
 plt.ylabel("number of cars")  
 plt.title("sp time / q time histogram")  
 plt.show()  
  
 plt.cla()  
 plt.clf()  
 plt.close()  
  
 sp\_run\_faster\_count = 0  
 q\_run\_faster\_count = 0  
 same\_time\_count = 0  
  
 sp\_to\_q\_run\_time\_ratio = []  
 for i in range(len(sp\_run\_times)):  
 sp\_to\_q\_run\_time\_ratio.append(sp\_run\_times[i] / q\_run\_times[i])  
 if sp\_run\_times[i] > q\_run\_times[i]:  
 q\_run\_faster\_count += 1  
 elif sp\_run\_times[i] < q\_run\_times[i]:  
 sp\_run\_faster\_count += 1  
 else:  
 same\_time\_count += 1  
 plt.hist(sp\_to\_q\_run\_time\_ratio, bins=20, edgecolor='black', linewidth=1.2)  
 plt.xlabel("sp run time / q run time")  
 plt.ylabel("number of runs")  
 plt.title("sp run time / q run time histogram")  
 plt.show()  
  
 plt.cla()  
 plt.clf()  
 plt.close()  
  
 print("sp ran faster than q run", sp\_run\_faster\_count, " times")  
 print("q ran faster than sp run", q\_run\_faster\_count, " times")  
 print("sp ran equal to q run", same\_time\_count, " times")  
  
 # scatter the nodes used in the simulation  
 RN = Road\_Network.Road\_Network("TLV")  
 fig, ax = ox.plot\_graph(RN.graph, bgcolor='white', node\_color='black', show=False, close=False)  
 plt.title("Nodes used in the simulation")  
 # plt.show()  
  
 all\_nodes = ["blue"] + Getters.top\_right\_nodes + ["purple"] + Getters.bottom\_right\_nodes + [  
 "red"] + Getters.bottom\_left\_nodes + ["green"] + Getters.top\_left\_nodes  
 cur\_color = "blue"  
 for node in all\_nodes:  
 if type(node) == str:  
 cur\_color = node  
 continue  
 x, y = RN.get\_xy\_from\_node\_id(node)  
 ax.scatter(x, y, color=cur\_color, s=50)  
 # plt.legend()  
 plt.show()