Multi Agents in A city traffic Simulation High Level Design

ALon ReicHer, Liad gam

‏2023

תוכן

[Preface 7](#_Toc143274867)

[Notes for current version 7](#_Toc143274868)

[Simulation Concept – Model Description 7](#_Toc143274869)

[Environment 7](#_Toc143274870)

[Agent 8](#_Toc143274871)

[GUI 8](#_Toc143274872)

[Simulation Parameters 9](#_Toc143274873)

[Imported Data 9](#_Toc143274874)

[Fixed parameters 9](#_Toc143274875)

[Configurable parameters 9](#_Toc143274876)

[Exported Data 9](#_Toc143274877)

[Software Architecture 10](#_Toc143274878)

[Class Diagram (Unified Modeling Language) 10](#_Toc143274879)

[Data Structures 11](#_Toc143274880)

[Package: Main\_Files 11](#_Toc143274881)

[Class: Route (Abstract Base Class) 11](#_Toc143274882)

[Method: decide\_first\_road () 11](#_Toc143274883)

[Method: get\_next\_road () 11](#_Toc143274884)

[Method: get\_alt\_road () 11](#_Toc143274885)

[Class: Random\_route (Route) 12](#_Toc143274886)

[Method: \_\_init\_\_(self, src\_node, dst\_node, road\_network) 12](#_Toc143274887)

[Method: decide\_first\_road (self) 12](#_Toc143274888)

[Method: get\_next\_road (self) 12](#_Toc143274889)

[Method: get\_alt\_road (self) 12](#_Toc143274890)

[Class: Q\_Learning\_Route (Route) 13](#_Toc143274891)

[Method: \_\_init\_\_(self, src\_node, dst\_node, road\_network, start\_time) 13](#_Toc143274892)

[Method: get\_tables\_directory (self, tables\_directory) 13](#_Toc143274893)

[Method: load\_q\_table(self, src, dst, save\_path) 14](#_Toc143274894)

[Method: find\_best\_available\_road (self, next\_node): 14](#_Toc143274895)

[Method: decide\_first\_road (self) 14](#_Toc143274896)

[Method: get\_next\_road (self) 15](#_Toc143274897)

[Method: get\_alt\_road(self) 15](#_Toc143274898)

[Class: Shortest\_path\_route (Route) 16](#_Toc143274899)

[Method: \_\_init\_\_(self, src\_node, dst\_node, road\_network) 16](#_Toc143274900)

[Method: decide\_first\_road (self) 16](#_Toc143274901)

[Method: get\_next\_road (self) 16](#_Toc143274902)

[Method: get\_alt\_road (self) 16](#_Toc143274903)

[Class: Car 18](#_Toc143274904)

[\_\_init\_\_(self, id, source\_node, destination\_node, starting\_time, road\_network, route\_algorithm='random', num\_episodes = 2000, use\_existing\_q\_table = True): 18](#_Toc143274905)

[decide\_route\_algorithm(self, route\_algorithm, source\_node, destination\_node): 19](#_Toc143274906)

[start\_car(self): 19](#_Toc143274907)

[move\_next\_road(self, time): 19](#_Toc143274908)

[check\_if\_finished(self): 20](#_Toc143274909)

[force\_finish(self): 20](#_Toc143274910)

[update\_time\_until\_next\_road(self, road): 20](#_Toc143274911)

[update\_travel\_time(self, time): 21](#_Toc143274912)

[get\_routing\_algorithm(self): 21](#_Toc143274913)

[get\_time\_until\_next\_road(self): 21](#_Toc143274914)

[get\_xy\_source(self): 22](#_Toc143274915)

[get\_xy\_destination(self): 22](#_Toc143274916)

[get\_xy\_current(self): 22](#_Toc143274917)

[Class: Road 23](#_Toc143274918)

[\_\_init\_\_(self, id, source\_node, destination\_node, length, max\_speed, activate\_traffic\_lights, rain\_intensity = 0): 23](#_Toc143274919)

[calculate\_time(self): 23](#_Toc143274920)

[update\_speed(self, current\_time, traffic\_white\_noise): 24](#_Toc143274921)

[update\_road\_speed\_dict(self, new\_speed): 24](#_Toc143274922)

[update\_eta\_dict(self): 25](#_Toc143274923)

[calculate\_eta(self, speed): 25](#_Toc143274924)

[get\_eta(self, time): 25](#_Toc143274925)

[block (self): 26](#_Toc143274926)

[unblock (self): 26](#_Toc143274927)

[Class: Node 27](#_Toc143274928)

[\_\_init\_\_(self, id, osm\_id, x, y, traffic\_lights, street\_count): 27](#_Toc143274929)

[Class: Road\_Network 28](#_Toc143274930)

[\_\_init\_\_(self, graph\_name, activate\_traffic\_lights = False, rain\_intensity = 0, traffic\_white\_noise = True): 28](#_Toc143274931)

[set\_nodes\_array(self): 28](#_Toc143274932)

[set\_roads\_array(self, activate\_traffic\_lights, rain\_intensity): 29](#_Toc143274933)

[create\_grpah(self): 29](#_Toc143274934)

[set\_adjacency\_roads(self): 29](#_Toc143274935)

[plan\_road\_blockage (self , road\_id, start\_time, end\_time): 30](#_Toc143274936)

[block\_road(self, road\_id): 30](#_Toc143274937)

[unblock\_road(self, road\_id): 30](#_Toc143274938)

[set\_roads\_speeds\_from\_dict(self, roads\_speeds, current\_time): 31](#_Toc143274939)

[update\_roads\_speeds(self, current\_time: str): 31](#_Toc143274940)

[add\_shortest\_path\_to\_matrix (self, src\_id, dst\_id): 32](#_Toc143274941)

[get\_next\_road\_from\_matrix (self, src\_id, dst\_id): 32](#_Toc143274942)

[get\_next\_road\_shortest\_path (self, src\_id, dst\_id): 32](#_Toc143274943)

[get\_xy\_from\_node\_id(self, node\_id: int): 33](#_Toc143274944)

[get\_xy\_from\_osm\_id(self, osm\_id: int): 33](#_Toc143274945)

[get\_node\_from\_osm\_id(self, osm\_id: int): 34](#_Toc143274946)

[get\_road\_from\_src\_dst(self, src\_id, dst\_id): 34](#_Toc143274947)

[get\_shortest\_path(self, src\_id, dst\_id): 35](#_Toc143274948)

[check\_if\_path\_is\_blocked(self, path): 35](#_Toc143274949)

[Class: Car\_Manger 36](#_Toc143274950)

[\_\_init\_\_(self): 36](#_Toc143274951)

[add\_update\_to\_dictionary(self, time, car\_id, x, y, node\_id): 36](#_Toc143274952)

[add\_car(self, car, time): 37](#_Toc143274953)

[sort\_cars\_in\_simulation(self): 37](#_Toc143274954)

[calc\_nearest\_update\_time(self, time: datetime.datetime): 37](#_Toc143274955)

[find\_earliest\_waiting\_car(self): 38](#_Toc143274956)

[update\_cars(self, timeStamp: int, current\_datetime: datetime.datetime): 38](#_Toc143274957)

[is\_car\_stuck(self, car): 38](#_Toc143274958)

[is\_car\_finished(self, car): 39](#_Toc143274959)

[clear(self): 39](#_Toc143274960)

[get\_all\_cars\_ids(self): 40](#_Toc143274961)

[Class: Simulation\_Manager 41](#_Toc143274962)

[\_\_init\_\_(self, graph\_name, time\_limit, activate\_traffic\_lights=False, rain\_intensity = 0, traffic\_white\_noise = True, start\_time=datetime.datetime(year=2023, month=6, day=29, hour=8, minute=0, second=0)): 41](#_Toc143274963)

[run\_full\_simulation(self, cars, number\_of\_simulations=1): 41](#_Toc143274964)

[set\_up\_simulation(self, cars: list): 42](#_Toc143274965)

[start\_simulation(self): 43](#_Toc143274966)

[end\_simulation(self, simulation\_number): 43](#_Toc143274967)

[update\_simulation\_clocks(self, time: int): 43](#_Toc143274968)

[update\_simulation\_roads\_speed\_dict(self): 44](#_Toc143274969)

[update\_simulation\_roads\_current\_speeds(self): 44](#_Toc143274970)

[Update\_block\_roads(self): 44](#_Toc143274971)

[read\_road\_speeds(self, datetime\_obj: datetime.datetime): 45](#_Toc143274972)

[update\_road\_blockage (self, road\_id, start\_time = None, end\_time = None): 45](#_Toc143274973)

[write\_simulation\_results(self, copy\_cars: list, i: int): 46](#_Toc143274974)

[get\_simulation\_routes(self, cars: list, simulation\_number: int): 46](#_Toc143274975)

[Package: Q\_Learning\_Classes 48](#_Toc143274976)

[Class: Q\_Agent 48](#_Toc143274977)

[\_\_init\_\_(self, src, dst, start\_time, road\_network) 48](#_Toc143274978)

[initialize\_q\_table(self) 49](#_Toc143274979)

[choose\_action(self, epsilon) 49](#_Toc143274980)

[get\_next\_road(self) 49](#_Toc143274981)

[update\_q\_table (self, reward, learning\_rate, discount\_factor) 49](#_Toc143274982)

[add\_time\_to\_simulation\_time(self) 50](#_Toc143274983)

[Update\_state(self) 50](#_Toc143274984)

[Update\_path(self) 50](#_Toc143274985)

[Move\_to\_next\_road(self, max\_steps\_per\_episode) 51](#_Toc143274986)

[step(self, epsilon) 51](#_Toc143274987)

[calculate\_reward\_basic(self, blocked\_roads) 51](#_Toc143274988)

[calculate\_reward(self, blocked\_roads) 52](#_Toc143274989)

[reset(self) 52](#_Toc143274990)

[Class: Q\_Learning 53](#_Toc143274991)

[\_\_init\_\_(self, road\_network, learning\_rate=0.1, discount\_factor=0.9, epsilon=0.2) 53](#_Toc143274992)

[save\_q\_table(self, agent , save\_path) 54](#_Toc143274993)

[train (self, start\_time: datetime, epsilon\_decay\_rate=0.9995, is\_plot\_results=True) 54](#_Toc143274994)

[check\_all\_agents\_done(self) 55](#_Toc143274995)

[Class: Animate\_Simulation 55](#_Toc143274996)

[\_\_init\_\_(self, animation\_speed = 1, repeat = True) 55](#_Toc143274997)

[plotting\_custom\_route(self, SM, custom\_routes: list, cars: list) 56](#_Toc143274998)

[animate\_route (self, SM, ax, fig, scatter\_list, chosen\_cars\_ids) 56](#_Toc143274999)

[Package: Utilities 58](#_Toc143275000)

[Class: Getters 58](#_Toc143275001)

[get\_graph (graph\_name): 58](#_Toc143275002)

[get\_q\_tables\_directory() 58](#_Toc143275003)

[get\_simulation\_speeds\_file\_path (graph, graph\_name): 59](#_Toc143275004)

[time\_delta\_to\_seconds (time): 59](#_Toc143275005)

[node\_route\_to\_osm\_route (route): 59](#_Toc143275006)

[get\_random\_src\_dst (RN): 60](#_Toc143275007)

[get\_key\_from\_value(dictionary, value): 60](#_Toc143275008)

[Class: Speeds 60](#_Toc143275009)

[add\_max\_speed\_to\_graph (graph): 61](#_Toc143275010)

[Generate\_speed (highway, max\_speed, day, current\_time): 61](#_Toc143275011)

[Generate\_day\_data (graph, graph\_name): 61](#_Toc143275012)

[color\_edges\_by\_speed(SM, current\_time, blocked\_roads): 62](#_Toc143275013)

[Class: Results 62](#_Toc143275014)

[save\_results\_to\_JSON (graph\_name, results): 62](#_Toc143275015)

[read\_results\_from\_JSON (graph\_name): 63](#_Toc143275016)

[print\_simulation\_results(SM) 63](#_Toc143275017)

[plot\_past\_result(past\_result\_json\_name) 63](#_Toc143275018)

[car\_times\_bar\_chart (SM, car\_number) 64](#_Toc143275019)

[get\_simulation\_times (SM) 64](#_Toc143275020)

[car\_times\_bar\_chart\_Q\_agent(src, dst, all\_training\_paths\_nodes, all\_training\_times, ax=None) 64](#_Toc143275021)

[plot\_rewards(src, dst, var:list, ax=None) 65](#_Toc143275022)

[plot\_results(src, dst, all\_training\_paths\_nodes, all\_training\_times, mean\_rewards) 65](#_Toc143275023)

[Timing Diagram 67](#_Toc143275024)

[Bibliography 68](#_Toc143275025)

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Description** |
|  | 18.8.23 |  |  |
| 0.1 |  |  | Creation |

## Preface

### Notes for current version

# Simulation Concept – Model Description

## Environment

The environment is made up of one-way roads, each road starts at an intersection and ends at another intersection. Each road will include its length, the current speed on it, the roads you can pass to from it. As soon as an agent reaches the road, the agent will calculate with the help of the road data the time it will take him to complete the trip on this road.

## Agent

An agent is a car driving on the roads on the way to a predetermined destination. The agent will be able to use different ways to determine the mode of movement. It can be randomly oriented, according to the shortest distance from the target, or a reinforcement-based learning algorithm (Q-learning).

An agent's state is defined by the current time (time and day of the week), current node and destination node. An agent will take information from the relevant table for the day of the week from a table updated for that day and the time slice of the previous week, according to its current location, the current time slice, and the target vertex.

For example: a car in edge no.1 that wants to reach edge no.3, at 10:00 on Sunday, will check last week's table - on Sunday at 10:00, when the current curve is 1 and the destination is curve 3, which next curve (continuation of curve 1) will give the maximum return.

In the simulation we will have a Q table from each source vertex to each destination vertex on the simulation graph, the return for each arc is negative, and is defined in direct proportion to the time it takes to cross it.

An agent will also contribute to filling the distance table: when an agent uses the shortest path algorithm - if the data for his trip is not in the distance table, he will update the distance table according to his origin and destination. Otherwise, the relevant data for the agent already exists in the table, and he will take it from there.

● In the event that an agent uses the shortest path algorithm, and encounters a block in an arc he wanted to enter - he will choose an arc from among the other optional arcs, and will continue to act according to the same principle.

● For all algorithms - if an agent cannot move forward due to blocking of all arcs, it will wait/stop completely

## GUI

At the end of the simulation, the map of the area will be printed with two main features:

1. Tracking the travel path of specific agents on the map.

2. Color the edges colors based on road's speeds.

# Simulation Parameters

### Imported Data

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Calculated value** |
| G | Graph of the roads, as imported with the OSM API. | --- |
| Speed Json | Data (in a JSON format) about cars speed on the roads in the imported OSM graph. |  |
|  |  |  |

### Fixed parameters

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Value** |
| INTERVAL\_TIME | The time intervals of the speed data. | 10 (minutes) |
|  |  |  |

### Configurable parameters

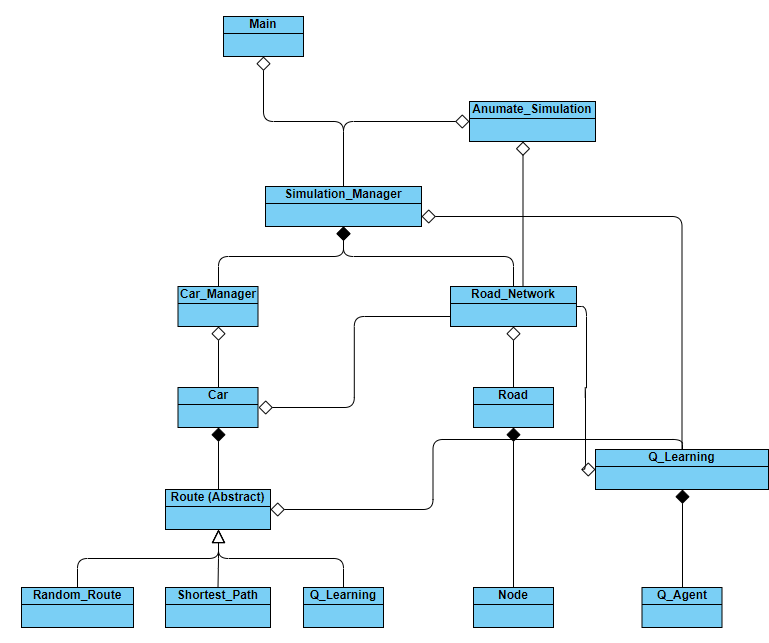
|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Valid values** |
| location | The simulated location. A city/ part of a city. | City names |
| routing\_algorithm | The algorithm which is used by agents to determine the next road/edge of their route. | "random"/"shortest path"/"q learning" |
| Traffic\_lights | Determine if to enable traffic signals in the simulation | True/ False |
| TRAFFIC\_WHITE\_NOISE | Determine if to enable gaussian white noise to speeds in the simulation\_speed json file | True/ False |
| Rain\_intensity | Determine the level of rain in the simulation: 0- no rain, 3- heavy rain | 0/1/2/3 |

### Exported Data

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Explanation** | **Calculated value** |
| Simulation results | Saves a Json file of the simulation results for every car including: has it reached destination, start time, end time, day of the week, routing algorithm, drive time, nodes route, roads route and drive time in each road. | JSON file |
| Q\_table | Calculated Q table from source to destination | List of lists that represents the q table.  pkl file |
| Graph speed | The last week's speeds for every road in an interval of 10 minutes. | JSON file |

# Software Architecture

## Model Class Diagram (Unified Modeling Language):

this is a class diagram that describes the relationships between the different classes in our project:

## Data Structures

# Package: Main\_Files

# Class: Route (Abstract Base Class)

This abstract base class defines the interface and common methods for different routing strategies.

### Method: decide\_first\_road ()

**Functionality:**

* This method is abstract and should be implemented by subclasses.
* It is responsible for deciding the first road that a car should take.
* Returns the selected first road.

### Method: get\_next\_road ()

**Functionality:**

* This method is abstract and should be implemented by subclasses.
* It determines the next road that the car should take based on the current road and other parameters.
* Returns the selected next road.

### Method: get\_alt\_road ()

**Functionality:**

* This method is abstract and should be implemented by subclasses.
* It determines an alternative road when the chosen road is blocked or unavailable.
* Returns the selected alternative road, or None if no other road is available.

# Class: Random\_route (Route)

This class represents a random routing strategy for car simulation.

### Method: \_\_init\_\_(self, src\_node, dst\_node, road\_network)

**Input:**

* src\_node (int): The source node ID.
* dst\_node (int): The destination node ID.
* road\_network (Road\_Network): The road network object.

### Method: decide\_first\_road (self)

**Functionality:**

* Selects the first road randomly from roads connected to the source node.

**Output:**

* (Road): the selected first road.

### Method: get\_next\_road (self)

**Functionality:**

* Selects the next road randomly from the list of adjacent roads.
* Ensures that the selected road is not blocked and has adjacent roads.

**Output:**

* (Road): the selected road.

### Method: get\_alt\_road (self)

**Input and Functionality:**

Try to find an alternative road to the current road if it's blocked.

**Output:**

* (Road): the alternative road, or None.

# Class: Q\_Learning\_Route (Route)

This class represents a Q-learning based routing strategy for car simulation.

### Method: \_\_init\_\_(self, src\_node, dst\_node, road\_network, start\_time)

Input:

* src\_node (int): The source node ID.
* dst\_node (int): The destination node ID.
* road\_network (Road\_Network): The road network object.
* start\_time (datetime.datetime): The starting time of the simulation.

Functionality:

* Initializes the Q-learning route with source, destination, and other parameters.
* Creates an instance of the QLearning class to perform Q-learning.
* Loads or trains the Q-table based on previous experience or initializes a new one.
* Defines the agent's path using Q-learning.

Output:

* None

### Method: get\_tables\_directory (self, tables\_directory)

Input:

* tables\_directory (str): The name of the directory.

Functionality:

* Get the path to the Q-value tables directory.

Output:

* None

### Method: load\_q\_table(self, src, dst, save\_path)

Input:

* src (int): The source node index.
* dst (int): The destination node index.
* save\_path (str): The path to load the Q-value table.

Functionality:

* Load a Q-value table from a file.
* If the file is not found, the Q-value table will not be loaded.

Output:

* (bool): true if loaded, false if not.

### Method: find\_best\_available\_road (self, next\_node):

Input:

* next\_node (int): the id of the next node to check.

Functionality:

* Finds the best road that start from next\_node according to the q table.

Output:

* (Road or None): if finds road that isn’t blocked then return a Road, else return None.

### Method: decide\_first\_road (self)

Functionality:

* Chooses the first road using Q-learning based on the source node.

Output:

* (Road): the selected first road.

### Method: get\_next\_road (self)

Functionality:

* This method uses Q-learning to select the next road based on the current road's destination node.

Output:

* (Road): the selected road.

### Method: get\_alt\_road(self)

Functionality:

* Try to find an alternative road to the current road if it's blocked, based on the q table of the agent.

Output:

* (Road): the alternative road, or None.

# Class: Shortest\_path\_route (Route)

This class represents a shortest path-based routing strategy for car simulation.

### Method: \_\_init\_\_(self, src\_node, dst\_node, road\_network)

Input:

* src\_node (int): The source node ID.
* dst\_node (int): The destination node ID.
* road\_network (Road\_Network): The road network object.

### Method: decide\_first\_road (self)

Functionality:

* Determines the first road using a shortest path algorithm.

Output:

* (Road): the selected first road.

### Method: get\_next\_road (self)

Functionality:

* Determines the next road using a shortest path algorithm.

Output:

* (Road): the selected road.

### Method: get\_alt\_road (self)

Functionality:

* This method calculates an alternative road based on minimum distance from adjacent roads.

Output:

* (Road): the alternative road, or None.

# Class: Car

This class represents a car within a road network simulation.

### \_\_init\_\_(self, id, source\_node, destination\_node, starting\_time, road\_network, route\_algorithm='random', num\_episodes = 2000, use\_existing\_q\_table = True):

Input:

* id (int): The unique identifier for the car.
* source\_node (int): The source node id which the car starts its journey.
* destination\_node (int): The destination node id where the car aims to arrive.
* starting\_time (datetime): The time when the car's journey begins.
* road\_network (Road\_Network): The road network within which the car operates.
* route\_algorithm (str: optional): The algorithm the car will use to decide its route (default: 'random').
* num\_episodes (int): \*only matters if the car uses Q Learning\*. The number of episodes the agent will do in its training.
* use\_existing\_q\_table (bool): \*only matters if the car uses Q Learning\*. Determines if the Q Learning agent will use past experience of the same path, if exists.

Functionality:

* Initializes a new instance of the Car class with the provided attributes.
* Sets up various attributes to manage the car's state, route, and movement within the simulation.
* Sets up attributes for the q learning route.

Output:

• None

### decide\_route\_algorithm(self, route\_algorithm, source\_node, destination\_node):

Input:

* route\_algorithm (str): The string representing the route algorithm.
* source\_node (int): Source node ID.
* destination\_node (int): Destination node ID.

Functionality:

* Decides the route algorithm based on the provided string.
* Returns the selected route algorithm instance.

Output:

* (Route): The selected route algorithm.

### start\_car(self):

Functionality:

* Moves the car to the first road based on the starting node and the car's route object. Updates the car's time until the next road.

Output:

* (Road): The first road the car will travel to.

### move\_next\_road(self, time):

Input:

* Time (float): The time interval.

Functionality:

* Moves the car to the next road based on the route's next node.
* Updates the car's time until the next road.
* Returns the next road the car will travel to, or "blocked" if the road is blocked.

Output:

* (Road or str): The next road the car will travel to, or "blocked" if the road is blocked.

### check\_if\_finished(self):

Functionality:

* Checks if the car has reached its destination.

Output:

* (bool): True if the car has reached its destination, False otherwise.

### force\_finish(self):

Functionality:

* Used on we force the car to finish
* Appends the last road to the past\_roads list.

Output:

* None

### update\_time\_until\_next\_road(self, road):

Input:

* road (Road): the road that the car is driving in.

Functionality:

* Updates the time until the car finishes the current road and reaches the next road.

Output:

* None

### update\_travel\_time(self, time):

Input:

* time (float): The time interval.

Functionality:

* Updates the total travel time and the time until the next road.

Output:

* (datetime.timedelta): the car's total travel time

### get\_routing\_algorithm(self):

Functionality:

* Gets the routing algorithm the car uses.

Output:

* str: The routing algorithm name.

### get\_time\_until\_next\_road(self):

Functionality:

* Gets the time until the car reaches the next road.
* If car is blocked then return 600 seconds.

Output:

* (int): The time in seconds.

### get\_xy\_source(self):

Functionality:

* Gets the x, y coordinates of the source node.

Output:

* tuple: (x, y) coordinates.

### get\_xy\_destination(self):

Functionality:

* Gets the x, y coordinates of the destination node.

Output:

* tuple: (x, y) coordinates.

### get\_xy\_current(self):

Functionality:

* Gets the x, y coordinates of the current node.

Output:

* tuple: (x, y) coordinates.

# Class: Road

Represents a road segment in a road network.

### \_\_init\_\_(self, id, source\_node, destination\_node, length, max\_speed, activate\_traffic\_lights, rain\_intensity = 0):

Input:

* id (int): The unique identifier for the road.
* source\_node (Node): The source node from which the road originates (class Node).
* destination\_node (Node): The destination node where the road ends (class Node).
* Length (int): The length of the road in meters.
* max\_speed (int): The maximum speed allowed on the road in km/h.
* activate\_traffic\_lights (bool): A boolean indicating whether traffic lights are active on the road.
* rain\_intensity (int): optinal, represents the intensity of the rain where 0 = no rain, 1 = little bit of rain, 2 = moderate rain, 3 = heavy rain.

Functionality:

* Initializes a new instance of the Road class with the provided attributes.
* Sets up various attributes to represent the road's characteristics and traffic conditions.

Output:

* None

### calculate\_time(self):

Functionality:

* Calculates the estimated time it takes to travel the road, based on self.current\_speed.
* If activate\_traffic\_lights is true then, adds a random time based on the road's destination node number of intersecting roads. Supposed to simulate real life traffic signals time.

Output:

* (float): The estimated time in seconds.

### update\_speed(self, current\_time, traffic\_white\_noise):

Input:

* current\_time (datetime): The current time.
* traffic\_white\_noise (bool): adds a little white noise to the known speed of last week to get a little bit of diversity.

Functionality:

* Updates the current speed of the road based on the provided time, traffic\_white\_noise and rain\_intensity.
* Recalculates the estimated time of arrival (ETA).

Output:

* (float): The updated estimated time of arrival (ETA) in seconds.

### update\_road\_speed\_dict(self, new\_speed):

Input:

* new\_speed (dict): A dictionary containing times and corresponding road speeds.

Functionality:

* Updates the road speed dictionary and recalculates ETA values.

Output:

* None

### update\_eta\_dict(self):

Functionality:

* Updates the estimated time of arrival (ETA) dictionary based on last week's road speeds.

Output:

* None

### calculate\_eta(self, speed):

Input:

* Speed (int): The speed of travel on the road in km/h.

Functionality:

* Calculates the estimated time of arrival (ETA) based on the provided speed.
* This function used in update\_eta\_dict.

Output:

* float: The estimated time of arrival (ETA) in seconds.

### get\_eta(self, time):

Input:

* time (str): The time for which ETA is requested as a string. Represented by hours and minutes, i.e: "08:00".

Functionality:

* Gets the estimated time of arrival (ETA) for the specified time.

Output:

* float: The estimated time of arrival (ETA) in seconds.

### block (self):

Functionality:

* Turning on self.is\_blocked flag.
* Making the road's eta infinite.

Output:

* None

### unblock (self):

Functionality:

* Turning off self.is\_blocked flag.
* Recalculates the road's eta

Output:

* (float, int): the new eta and the new current speed.

# Class: Node

This is an helper class, stores the attributes of every node in the graph, and making it easily accessable.

### \_\_init\_\_(self, id, osm\_id, x, y, traffic\_lights, street\_count):

Input:

* id: The unique identifier for the node.
* osm\_id: The OpenStreetMap identifier for the node.
* x: The x-coordinate of the node's location.
* y: The y-coordinate of the node's location.
* traffic\_lights: A boolean indicating whether the node has traffic lights.
* street\_count: The number of streets connected to the node.

Functionality:

* Initializes a new instance of the Node class with the provided attributes.
* Sets up various attributes to represent the node's characteristics and location.

Output:

* None

# Class: Road\_Network

This class represents a road network, containing the graph of the simulation, all the roads in the graph, and related data.

### \_\_init\_\_(self, graph\_name, activate\_traffic\_lights = False, rain\_intensity = 0, traffic\_white\_noise = True):

Input:

* graph\_name (str): The name of the graph (if exist) or alternatively the name of the desired city.
* activate\_traffic\_lights (bool, optional): Whether to activate traffic lights for roads (default: False).
* rain\_intensity (int, optional): Indicatting the intensity of the rain where 0 = no rain, 3 = heavy rain. (default: 0).
* traffic\_white\_noise (bool, optional): Whether to activate a white noise to last week's speeds (default: True).

Functionality:

* Initializes a new instance of the Road\_Network class using the provided graph data and optional parameters.
* Sets up various attributes to manage the road network, nodes, roads, and adjacency information.

Output:

* None

### set\_nodes\_array(self):

Functionality:

* Initialize and populate the nodes\_array attribute with Node objects.

Output:

* None

### set\_roads\_array(self, activate\_traffic\_lights, rain\_intensity):

Input:

* activate\_traffic\_lights (bool): Whether to activate traffic lights for roads.
* rain\_intensity (int): rain intensity (0=no rain, 3= heavy rain).

Functionality:

* Initialize and populate the roads\_array attribute with Road objects based on the provided graph data and traffic light activation.

Output:

* None

### create\_grpah(self):

Functionality:

* Create a new nx.MultiDiGraph graph from the roads\_array attributes and stores it in self.nx\_grpah.

Output:

* None

### set\_adjacency\_roads(self):

Functionality:

* Set the adjacent\_roads attribute for each road in the roads\_array.

Output:

* None

### plan\_road\_blockage (self , road\_id, start\_time, end\_time):

Input:

* Road\_id (int): the road id attribute of the road we want to block.
* Start\_time (datetime): the time we block the road.
* End\_time (datetime): the time we unblock the road.

Functionality:

* Plan a road blockage for a specific road.

Output:

* None

### block\_road(self, road\_id):

Input:

* road\_id (int): ID of the road to be blocked.

Functionality:

* Block a specific road by marking it as blocked.
* Activates block function in Road class.
* Change attributes in self.nx\_graph to represent the road being closed.

Output:

* None

### unblock\_road(self, road\_id):

Input:

* road\_id (int): ID of the road to be unblocked.

Functionality:

* Unblock a specific road by marking it as unblocked.
* Activates unblock function in Road class.
* Change attributes in self.nx\_graph to represent the road being open again.

Output:

* None

### set\_roads\_speeds\_from\_dict(self, roads\_speeds, current\_time):

Input:

* roads\_speeds (dict): Dictionary of road\_id: speed for different times of the day.
* current\_time (str): Current time in the simulation.

Functionality:

* Update road speeds based on the provided speeds dictionary and current time.

Output:

* None

### update\_roads\_speeds(self, current\_time: str):

Input:

* current\_time (str): Current time in the simulation.

Functionality:

* Update road speeds based on the current time.

Output:

* None

### add\_shortest\_path\_to\_matrix (self, src\_id, dst\_id):

Input:

* src (int): Source node ID.
* dest (int): Destination node ID.

Functionality:

* Calculates the shortest path between two nodes, updates distances matrix, and adds information to the matrix about the path.

Output:

* None

### get\_next\_road\_from\_matrix (self, src\_id, dst\_id):

Uses:

* get\_road\_from\_src\_dst

Input:

* src (int): Source node ID.
* dest (int): Destination node ID.

Functionality:

* Get the next road on the shortest path from the matrix.

Output:

* (Road): Next road on the shortest path.

### get\_next\_road\_shortest\_path (self, src\_id, dst\_id):

Uses:

* get\_next\_road\_from\_matrix
* add\_shortest\_path\_to\_matrix

Input:

* src (int): Source node ID.
* dest (int): Destination node ID.

Functionality:

* Get the next road on the shortest path, calculating if needed.

Output:

* (Road): Next road on the shortest path.

### get\_xy\_from\_node\_id(self, node\_id: int):

Input:

* node\_id (int): Node ID.

Functionality:

* Get x, y coordinates of a node based on its node ID.

Output:

* tuple: x, y coordinates.

### get\_xy\_from\_osm\_id(self, osm\_id: int):

Input:

* osm\_id (int): OSM ID of the node.

Functionality:

* Get x, y coordinates of a node based on its OSM ID.

Output:

* tuple: x, y coordinates.

### get\_node\_from\_osm\_id(self, osm\_id: int):

Input:

* osm\_id (int): OSM ID of the node.

Functionality:

* Get a node object based on its OSM ID.

Output:

* Node object: Node corresponding to the OSM ID.

### get\_road\_from\_src\_dst(self, src\_id, dst\_id):

Input:

* src\_id (int): Source node ID.
* dst\_id (int): Destination node ID.

Functionality:

* Get a road object based on source and destination node IDs.

Output:

* Road object: Road between the source and destination nodes.

### get\_shortest\_path(self, src\_id, dst\_id):

Uses:

* check\_if\_path\_is\_blocked.

Input:

* src\_id (int): Source node ID.
* dst\_id (int): Destination node ID.

Functionality:

* Get the shortest path between two node IDs.

Output:

* list: List of node IDs representing the shortest path.

### check\_if\_path\_is\_blocked(self, path):

Input:

* path (list): List of node IDs representing a path.

Functionality:

* Check if a given path is blocked by any blocked roads.

Output:

* bool: True if the path is blocked, False otherwise.

# Class: Car\_Manger

This class manages cars within a road network simulation. It adds, removes, and updates cars during the simulation.

### \_\_init\_\_(self):

Input:

• None

Functionality:

* Initializes a new instance of the CarManager class.
* Sets up various attributes to manage cars in different stages of the simulation.

Output:

* None

### add\_update\_to\_dictionary(self, time, car\_id, x, y, node\_id):

Input:

* time (datetime.datetime): Time of the update.
* car\_id (int): ID of the car.
* x (float): X-coordinate of the car's position.
* y (float): Y-coordinate of the car's position.
* node\_id (int): ID of the current node.

Functionality:

* Add car update information to the dictionary.

Output:

* None

### add\_car(self, car, time):

Input:

* car (Car): Car object to be added.
* time (datetime.datetime): Current time in the simulation.

Functionality:

* Add a car to the simulation based on its starting time.

Output:

* None

### sort\_cars\_in\_simulation(self):

Functionality:

* Sort cars in the simulation based on the time until the next road.
* Update the nearest update time.

Output:

* bool: False if no cars are in simulation, True otherwise.

### calc\_nearest\_update\_time(self, time: datetime.datetime):

Input:

* time (datetime.datetime): Current time in the simulation.

Functionality:

* Calculate the time of the nearest update based on waiting and simulation cars.

Output:

* float: The time of the nearest update.

### find\_earliest\_waiting\_car(self):

Functionality:

* Find the earliest time that a car is waiting to enter the simulation.

Output:

* datetime.datetime or None: The earliest starting time of a waiting car, or None if no cars are waiting.

### update\_cars(self, timeStamp: int, current\_datetime: datetime.datetime):

Input:

* timeStamp (int): The simulation time step.
* current\_datetime (datetime.datetime): Current time in the simulation.

Functionality:

* The most important function in Car manager.
* Update all the cars in the simulation based on the current time step.
* Adds waiting cars to the simulation if their starting time has come.

Output:

* None

### is\_car\_stuck(self, car):

Input:

* car (Car): The car to check.

Functionality:

* Check if a car is stuck in the simulation.

Output:

* bool: True if the car is stuck, False otherwise.

### is\_car\_finished(self, car):

Input:

* car (Car): The car to check.

Functionality:

* Check if a car has finished its journey in the simulation.

Output:

* bool: True if the car has finished, False otherwise.

### clear(self):

Input:

* None

Functionality:

* Clear all simulation data and reset the CarManager.
* Used when NUMBER\_OF\_SIMULATIONS is bigger than one.

Output:

* None

### get\_all\_cars\_ids(self):

Input:

* None

Functionality:

* Get the IDs of all cars currently in the simulation and those blocked.

Output:

* list[int]: List of car IDs.

# Class: Simulation\_Manager

Manages the simulation by creating and updating the road network, car manager, running the simulation and saving results.

### \_\_init\_\_(self, graph\_name, time\_limit, activate\_traffic\_lights=False, rain\_intensity = 0, traffic\_white\_noise = True, start\_time=datetime.datetime(year=2023, month=6, day=29, hour=8, minute=0, second=0)):

Input:

* graph\_name (str): The name of the existing graph or a city of choice.
* time\_limit (int): The maximum time the simulation will run in seconds.
* activate\_traffic\_lights (bool, optional): Indicates whether traffic lights are activated. Default is False.
* rain\_intensity (int, optinal): indicates about the rain intensity where 0 is no rain and 3 is heavy rain. Default is 0 (no rain).
* traffic\_white\_noise (bool, optional): if true then, the road's speeds in the simulation will be a little bit different from the speeds in the simulation speeds JSON file. Default is true.
* start\_time (datetime.datetime, optional): The starting datetime of the simulation. Default is June 29, 2023, 08:00:00.

Functionality:

* Initializes a new instance of the Simulation\_manager class.
* Sets up various attributes to manage the simulation, road network, and car manager.

Output:

* None

### run\_full\_simulation(self, cars, number\_of\_simulations=1):

Input:

* cars (list): List of Car objects for the simulation.
* number\_of\_simulations (int, optional): Number of simulations to run. Default is 1.

Functionality:

* Run the full simulation process including setup, execution, and result printing.

Output:

* None

**start\_q\_learning\_simulation(self, copy\_cars, num\_episodes, max\_steps\_per\_episode)**

Input:

* copy\_cars (list): List of Car objects for the simulation.
* num\_episodes (int): will determine the number of episodes to train the cars.
* max\_steps\_per\_episode (int): will determine the maximum number of steps per episode.

Functionality:

* Start the Q-Learning simulation by training the cars that use Q-Learning.

Output:

* None

### set\_up\_simulation(self, cars: list):

Input:

* cars (list): List of Car objects to be added to the simulation.

Functionality:

* Reset the simulation clocks.
* Set up the simulation by adding cars to the car manager.

Output:

* None

### start\_simulation(self):

Input:

* None

Functionality:

* Start the simulation and run it until all cars finish or time limit is reached.

Output:

* None

### end\_simulation(self, simulation\_number):

Input:

* simulation\_number (int): The index of the current simulation.

Functionality:

* Print the results of the simulation.

Output:

* None

### update\_simulation\_clocks(self, time: int):

Input:

* time (int): The time interval to update the simulation clock in seconds.

Functionality:

* Update the simulation clock and road speeds.

Output:

* None

### update\_simulation\_roads\_speed\_dict(self):

Functionality:

* Check if a day is passed in the simulation and if true then update the road speeds accordingly.

Output:

* None

### update\_simulation\_roads\_current\_speeds(self):

Functionality:

* Check if 10 minutes passed in the simulation and update the road speeds accordingly.

Output:

* None

### Update\_block\_roads(self):

Uses:

* Road\_network.blocked\_roads\_dict.
* Road\_network.block\_road(road\_id)
* Road\_network.unblock\_road(road\_id)

Functionality:

* Update all the blocked and soon to be blocked roads in the simulation, according to the current time.

Output:

* None

### read\_road\_speeds(self, datetime\_obj: datetime.datetime):

Uses:

* Getters. get\_simulation\_speeds\_file\_path(graph, graph\_name).
* Road\_network.set\_roads\_speeds\_from\_dict(data,time).

Input:

* datetime\_obj (datetime.datetime): The datetime for which road speeds are to be read.

Functionality:

* Read road speeds from a JSON file and update the roads using road network.

Output:

* None

### update\_road\_blockage (self, road\_id, start\_time = None, end\_time = None):

Uses:

* road\_network.plan\_road\_blockage(road\_id, start\_time, end\_time)

Input:

* road\_id (int): The id of the road to be blocked.
* start\_time (datetime.datetime): The datetime the road is blocked.
* end\_time (datetime.datetime): The datetime the road is unblocked.

Functionality:

* Read road speeds from a JSON file and update the roads using road network.

Output:

* None

### write\_simulation\_results(self, copy\_cars: list, i: int):

Input:

* copy\_cars (list): List of Car objects used in the simulation.
* i (int): Index of the current simulation.

Functionality:

* Write simulation results to the simulation results list.

Output:

* None

### get\_simulation\_routes(self, cars: list, simulation\_number: int):

Input:

* cars (list): List of Car objects.
* simulation\_number (int): Index of the simulation.

Functionality:

* Retrieve routes passed by cars in a simulation.

Output:

* list: List of routes passed by cars in the simulation.

**get\_fixed\_node\_id(self, osm\_id: int)**

Input:

* osm\_id (int): the original OSM node id.

Functionality:

* Get the node id of a node.

Output:

* (Int): the new node id.

# Package: Q\_Learning\_Classes

# Class: Q\_Agent

This is a class created for the q learning training, it represents an agent on the graph and will store important attributes for the q learning class, as well as update functions in order to train the agent.

Class attributes:

* src (int): src node id
* dst (int): dst node id
* start\_time (datetime): the start time of the simulation of the agent.
* Simulation\_time (datetime): The current simulation time.
* Road\_network (road\_network): the road network that the agent will be training on.
* node\_list (list): list of lists representing the nodes connectivity in the road network.
* q\_table (list): A list of lists representing the Q-table.
* current\_road (Road): The current road the agent is on.
* next\_road (Road): The next road the agent will travel on.
* num\_of\_steps (int): The number of steps the agent has taken.
* current\_state (int): The current node id.
* next\_state (int): The next node id that the agent will go to.
* action (int): The chosen action index.
* self.path\_nodes (list): The list of nodes in the path.
* path\_roads (list): The list of roads in the path.
* blocked (bool): A flag indicating whether the agent is blocked.
* finished (bool): A flag indicating whether the agent has reached its destination.

### \_\_init\_\_(self, src, dst, start\_time, road\_network)

Input:

* src (int): src node id
* dst (int): dst node id
* start\_time (datetime): the start time of the simulation of the agent.
* Road\_network (road\_network): the road network that the agent will be training on.

### initialize\_q\_table(self)

Functionality:

* Initializes the Q values table with zeros based on the number of nodes and their connectivity.

Output:

* None

### choose\_action(self, epsilon)

input:

* epsilon (float): The epsilon value.

Functionality:

* Choose an action based on the current state using an epsilon-greedy policy.

Output:

* int: The chosen action index.

### get\_next\_road(self)

Functionality:

* Retrieves the next road to travel based on the chosen action.

Output:

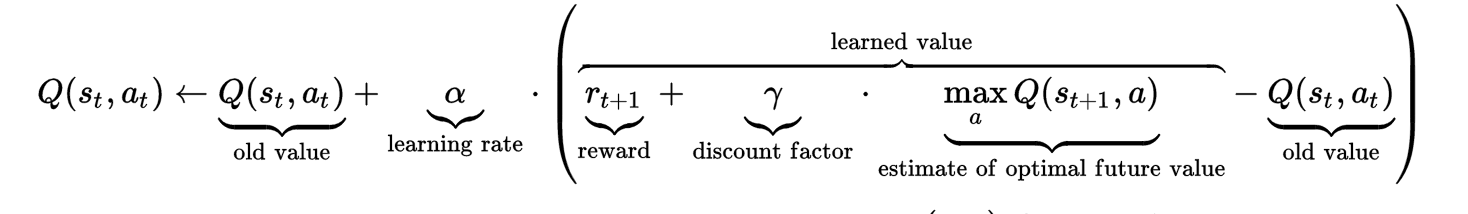
* (Road): representing the next road to travel.

### update\_q\_table (self, reward, learning\_rate, discount\_factor)

Inputs:

* reward (float): The reward received for the current action.
* learning\_rate (float): The learning rate.
* discount\_factor (float): The discount factor.

Functionality:

* Updates the Qvalue table based on the Qlearning update rule that is shown below.

Output:

* None

### add\_time\_to\_simulation\_time(self)

Functionality:

* Adds the time of the next road to the simulation time.

Output:

* None

### Update\_state(self)

Functionality:

* Update the current state to the next state.

Output:

* None

### Update\_path(self)

Functionality:

* Update the paths with the next road and the next state (aka next node).

Output:

* None

### Move\_to\_next\_road(self, max\_steps\_per\_episode)

Input:

* max\_steps\_per\_episode (int): The maximum number of steps per episode.

Functionality:

* Move the agent to the next road.

Output:

* None

### step(self, epsilon)

Input:

* epsilon (float): The epsilon value.

Functionality:

* Perform one step in the simulation.
* Uses the epsilon-greedy policy to choose an action, and then updates the state, action, next\_state.
* Adds the time to the simulation time.
* Updates the path with the next road.

Output:

* (int,Road, int): The chosen action, the next road, the next state.

### calculate\_reward\_basic(self, blocked\_roads)

Inputs:

* blocked\_roads (dict): A dictionary of blocked roads and their blockage times.

Functionality:

* Calculates the basic reward for a given action, considering blockages and reaching the destination.

Output:

* (float): The calculated reward value

### calculate\_reward(self, blocked\_roads)

Inputs:

* blocked\_roads (dict): A dictionary of blocked roads and their blockage times.

Functionality:

* Calculates the reward for a given action based on various factors.
* 1000 if the agent reached the destination.
* -1000 if the agent got blocked
* -3 if the agent got further away from the destination in his current step.
* -1 if the agent got closer to the destination.

Output:

* (float): The calculated reward value.

### reset(self)

Functionality:

* Reset the agent's parameters.
* Used at the end of each episode.

Output:

* None

# Class: Q\_Learning

A Q-learning algorithm implementation for route optimization in a road network.

Class attributes:

* road\_network (Road\_Network): The road network for which the Q-learning algorithm is applied.
* node\_list (list): A list of lists representing the nodes connectivity in the road network.
* simulation\_time (datetime): The current simulation time.
* learning\_rate (float): The learning rate for updating Q-values.
* discount\_factor (float): The discount factor for future rewards in Q-learning.
* epsilon (float): The exploration-exploitation trade-off factor.
* q\_table (dict): A dictionary representing the Q-table.
* rewards (list): A list of rewards received in each episode.
* blocked (bool): A flag indicating whether the agent is blocked.
* finished (bool): A flag indicating whether the agent has reached its destination.

### \_\_init\_\_(self, road\_network, learning\_rate=0.1, discount\_factor=0.9, epsilon=0.2)

Input:

* road\_network (Road\_Network): The road network for which the Q-learning algorithm is applied.
* learning\_rate (float): The learning rate for updating Q-values.
* discount\_factor (float): The discount factor for future rewards in Q-learning.
* epsilon (float): The exploration-exploitation trade-off factor.

Functionality:

* Initializes the Q\_Learning class with the given parameters and sets up attributes.

Output:

* None

### save\_q\_table(self, agent , save\_path)

Inputs:

* agent (Q\_Agent): The agent for which the Q-value table is saved.
* save\_path (str): The path to save the Qvalue table.

Functionality:

* Saves the Qvalue table to a file.
* The file name will include the graph name, src, dst, and the blocked roads, for example: "q\_table\_tel aviv\_1\_200\_blocked\_roads\_5\_20", that means that the graph name is "tel aviv", the source is 1, destination is 200, and the blocked roads are 5 and 20.

Output:

* None

### train (self, start\_time: datetime, epsilon\_decay\_rate=0.9995, is\_plot\_results=True)

Inputs:

* start\_time (datetime): The start time of the simulation.
* epsilon\_decay\_rate (float): The rate of epsilon decay. That means that evey episode the epsilon value is multiplied by this parameter.
* is\_plot\_results (bool): Whether to plot the results.

Functionality:

* Trains all the Q learning agents of the simulation for a source destination pair using Q learning algorithm.
* Plots the results if is\_plot\_results is True.
* Saves the Q tables.

Output:

* None

### check\_all\_agents\_done(self)

Functionality:

* Check if all agents are blocked or reached their destination at this episode.

Output:

* (bool): True if all agents are blocked or reached their destination, False otherwise.

# Class: Animate\_Simulation

Class attributes:

* animation (animation): This variable will store the animation when exists.
* animation\_speed(int): The speed in which the simulation will run.
* repeat (bool): Indicates if the animation will show once or infinite times.
* last\_speed\_update\_time (datetime): indicates the last time the color of the edges updated.
* Edge\_colors (list): will store all the colors of the edges for the animation.
* Node\_colors (list): will store all the colors for the nodes based if they have traffic signals or not.
* origins (list): A list of the origins of all the cars we add to the simulation.
* destinations (list): A list of the destinations of all the cars we add to the simulation.

### \_\_init\_\_(self, animation\_speed = 1, repeat = True)

Input:

* animation\_speed(int): The speed in which the simulation will run.
* repeat (bool): Indicates if the animation will show once or infinite times.

Functionality:

* Initializes the Animate\_Simulation class with the given parameters and sets up attributes.

Output:

* None

### plotting\_custom\_route(self, SM, custom\_routes: list, cars: list)

Input:

* SM (simulation\_manager): The simulation that we want to animate.
* Custom\_routes (list): the routes of the cars.
* Cars (list): all the cars that were in the simulation.

Functionality:

* Initializing the initial edges colors, node colors, sources, destinations and the cars for the animation.
* Calls self.animate\_route which will activate the animation.

Output:

* None

### animate\_route (self, SM, ax, fig, scatter\_list, chosen\_cars\_ids)

Input:

* SM (simulation\_manager): The simulation that we want to animate.
* Ax (matplotlib.axes.\_axes.AxesSubplot): Matplotlib AxesSubplot object for the animation.
* Fig (matplotlib.figure.Figure): Matplotlib Figure object for the animation.
* Scatter\_list (list): List of scatter plot objects representing car positions on the map.
* Chosen\_car\_ids (list): List of IDs of the cars in the simulation.

Functionality:

* Animates the movement of cars along their routes on the map. Provides interactive controls to pause and resume animation.
* Pressing space bar will either stop or continue the animation based on the current situation.
* Pressing escape will close the simulation.

Output:

* None

# Package: Utilities

This package contains a few classes, and every class is responsible for different type of general functions that were used in the project. These functions are general so they can be used in every other class without problem.

# Class: Getters

### get\_graph (graph\_name):

Input:

* graph\_name (str): the name of the graph file, without the extension or alternatively the name of the city.

Functionality:

* loads the graph from the "Graphs" package if exists.
* If doesn’t exist, load the graph from OSM, adds relevant data and saves the graph. Dist is set to 1500, that means the radius of the map from the center of the city is 1500 meters, this was done to prevent the map being too large (too many edges and nodes).

Output:

* (nx.MultiDiGraph, str): the new graph and the path to where the graph is saved.

### get\_q\_tables\_directory()

Functionality:

* get the directory of the q tables

Output:

* (str): the path to where the q tables are saved.

### get\_simulation\_speeds\_file\_path (graph, graph\_name):

Input:

* Graph (nx.MultiDiGraph): the graph
* graph\_name (str): the name of the graph file (without the extension) or the city.

Functionality:

* Load a dictionary of speeds for each road in the graph if exists.
* If doesn’t exist, generate one and saves it.

Output:

* ( str): the path to where the speed file is saved.

### time\_delta\_to\_seconds (time):

Input:

* Time (datetime.timedelta)

Functionality:

* transform a time delta to seconds.

Output:

* ( int): the number of seconds.

### node\_route\_to\_osm\_route (route):

Input:

* route (list): List of node IDs representing a route.

Functionality:

* Transform a route from node IDs to OSM IDs.

Output:

* list: List of OSM IDs representing the transformed route.

### get\_random\_src\_dst (RN):

Input:

* RN (road\_network): the road network that we will get the src and dest from.

Functionality:

* Get a random source and destination from the road network that have a path between them.

Output:

* (int, int): src id and dst id.

### get\_key\_from\_value(dictionary, value):

Input:

* dictionary (dict): The dictionary to search in.
* value: The value to search for.

Functionality:

* Retrieve the corresponding key from a dictionary given a value.

Output:

* key: The key corresponding to the given value.

# Class: Speeds

We define a few dictionaries in order to generate speeds for roads in the graph:

Max\_speed\_mapping – defines the max speed for every road type.

Mean\_mapping – defines the mean speed for every road type under no traffic, low traffic, medium traffic, and high traffic.

Std\_mapping defines the standard deviation of the speed for every road type under no traffic, low traffic, medium traffic and high traffic.

### add\_max\_speed\_to\_graph (graph):

Input:

* graph (multiDiGraph): The graph to add max\_speeds into.

Functionality:

* Add the max speed of each road to the graph if doesn’t exist.

Output:

* (multiDiGraph): the updated graph.

### Generate\_speed (highway, max\_speed, day, current\_time):

Input:

* Highway (str): the road's type
* Max\_speed (int): road's max speed.
* Day (int): representation of the day of the week in digits ( monday = 0, .. , Sunday = 7).
* Current\_time (datetime)

Functionality:

* Generate a speed for a road based on the day and time.

Output:

* (int): the calculated speed.

### Generate\_day\_data (graph, graph\_name):

Input:

* graph (multiDiGraph): The graph to add max\_speeds into.
* graph\_name (str): the name of the graph file (without the extension) or the city.

Functionality:

* Generates a dictionary of speeds for each road in the graph based on the day of the week and the time of day.
* Generates a json file with the Graphs.

Output:

* None

### color\_edges\_by\_speed(SM, current\_time, blocked\_roads):

Input:

* SM (Simulation\_Manager): the simulation that we want to color edge by.
* current\_time (Current\_time): the time we want to color the edges.
* Blocked\_roads (dict): dict that have all the blocked roads in the simulation in this format: {road\_id: [start\_time, end\_time]}.

Functionality:

* Colors the edges of the graph according to the speed of the road at the current time.
* If road is blocked then it will be black, no traffic is green, medium traffic is yellow and hevy traffic is red.

Output:

* (list): will have the colors for every edge.

# Class: Results

### save\_results\_to\_JSON (graph\_name, results):

Input:

* graph\_name (str): name of the graph
* result (dict): All the simulation results.

Functionality:

* Saves all the results in a JSON file according to the graph's name.

Output:

* None

### read\_results\_from\_JSON (graph\_name):

Input:

* graph\_name (str): name of the graph

Functionality:

* Read simulation results from a JSON file.

Output:

* dict: dict that contains all the simulation results.

### print\_simulation\_results(SM)

Input:

* SM (Simulation\_Manager): An object representing the simulation manager, containing simulation results.

Functionality:

* The function iterates through the simulation results stored in the simulation\_results attribute of the SM object and prints it to the console.

Output:

* None

### plot\_past\_result(past\_result\_json\_name)

Input:

* past\_result\_json\_name (str): Name of the JSON file containing past simulation results.

Functionality:

* Loads and plots past simulation results from a JSON file, including custom routes, starting points, and destination points of cars.

Output:

* None

### car\_times\_bar\_chart (SM, car\_number)

Input:

* SM (Simulation\_Manager): The object that runs the simulation.
* Car\_number (int): The car id for which the chart will be plotted.

Functionality:

* his function plots a bar chart of the times of a specific car in the simulation.

Output:

* None

### get\_simulation\_times (SM)

Input:

* SM (Simulation\_Manager): The object that runs the simulation.

Functionality:

* This function returns the times of all cars in the simulation in a list.

Output:

* (list): all the times driven of the cars in the simulation

### car\_times\_bar\_chart\_Q\_agent(src, dst, all\_training\_paths\_nodes, all\_training\_times, ax=None)

Input:

* src (int): Source node ID.
* dst (int): Destination node ID.
* all\_training\_paths\_nodes (list): List of paths (lists of node IDs) taken by the Q-learning agent during training.
* all\_training\_times (list): List of time taken for each training episode.
* ax (matplotlib.axes.\_axes.AxesSubplot, optional): Matplotlib AxesSubplot object to plot on (default is None).

Functionality:

* Plots a bar chart of the times taken by the Q-learning agent for different training episodes. Bars are colored based on whether the agent reached its destination.

Output:

* None

### plot\_rewards(src, dst, var:list, ax=None)

Input:

* src (int): Source node ID.
* dst (int): Destination node ID.
* var (list): List of mean rewards over training episodes.
* ax (matplotlib.axes.\_axes.AxesSubplot, optional): Matplotlib AxesSubplot object to plot on (default is None).

Functionality:

* Plots the mean rewards over training episodes for the Q-learning agent.

Output:

* None.

### plot\_results(src, dst, all\_training\_paths\_nodes, all\_training\_times, mean\_rewards)

Input:

* src (int): Source node ID.
* dst (int): Destination node ID.
* all\_training\_paths\_nodes (list): List of paths (lists of node IDs) taken by the Q-learning agent during training.
* all\_training\_times (list): List of time taken for each training episode.
* mean\_rewards (list): List of mean rewards over training episodes.

Functionality:

* Plots the Q-learning agent's training results, including the bar chart of times taken and the mean rewards plot, in a two-subplot layout.

Output:

* None

## Timing Diagram

* 1. Run the simulator using Main/ Gui.
  2. A Simulator Manager will be created and it will initialize Road Network and Car Manager objects.
  3. Cars will be created and will be saved in a list.
  4. Simulator\_Manager.run\_full\_simulation will be called, the function gets the list of cars and will run a simulation using its road network and car manager.
  5. Results will be saved in json file name: "simulation\_results\_{graph name}" where {graph\_name} is the name of the graph or the city name depends on what was given to simulator manager.
  6. The car's routes will be shown
  7. The car's routes animation will play.

## Bibliography

|  |  |
| --- | --- |
| [1] | IEEESA, *IEEE 802.11n2009—Amendment 5: Enhancements for Higher Throughput,* 2009. |
| [2] | IEEESA, *IEEE 802.112007,* 2007. |