Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Reinforcement Learning for Autonomous Intersection Management

Jack Lowe, BAI

A Research Project

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Arts in Engineering (Computer and Electronic)

Supervisor: Melanie Bouroche

April 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Jack Lowe

April 21, 2022

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Jack Lowe

April 21, 2022

Reinforcement Learning for Autonomous Intersection Management

Jack Lowe, Master of Arts in Engineering

University of Dublin, Trinity College, 2022

Supervisor: Melanie Bouroche

This project consists of the investigation of data needs for efficient Q-Learning for distributed Autonomous Intersection Management (AIM). Specifically, this entails the design and implementation of a lightweight Q-Learning framework that uses Connected Autonomous Vehicles (CAVs) as agents to decrease the delay in an unsignalised intersection. The literature review concludes that while Q-Learning is and has been an efficient tool in the design of scheduling policies for traffic light control, its potential for distributed and unsignaled AIM remains globally unresolved, specifically due to the neglect of current implementations to consider the materialistic need for Q-functions to be unencumbered by excessive data if they are to be stored locally inside CAVs. By minimising both agents and observations in our Q-Learning model, this project seeks to validate the hypothesis that minimal-data Q-functions are viable solutions to AIM scheduling policies considering the complexity of modern AIM modelling.

The results show that while the framework is capable of performing Q-learning, the fine line between an excessive number of states and sufficient agent information was not met. Various choices of observations and actions are nevertheless explored and the consequences they have on performance are analysed.

Acknowledgments

I would like thank my thesis supervisor Professor Mélanie Bouroche for her guidance throughout the process. Her expertise was vital to the project.

Jack Lowe

University of Dublin, Trinity College
April 2022

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Background

With the increasing traffic congestion that has come with contemporary urbanisation, alternative systems of traffic control have been widely sought and explored, namely through the development of Connected Autonomous Vehicles (CAVs). Considering the potential safety and time benefits of fully cooperative CAVs as opposed to Human driven Vehicles (HVs), Autonomous Intersection Management (AIM) has been at the forefront of traffic research since the pivotal work of Dresden and Stone in 2008 [4]. The continuous technological innovation in this time-frame has lead to significant scaling of AIM modelling complexity with every passing year. The advancements of simulation frameworks have enabled new considerations such as pedestrianisation[19], vehicle acceleration [15] and vehicle prioritisation [5]. The development of communication protocols and superior telecommunication hardware have also favoured the use of distributed systems over centralised intersections, thus adding to the the complexity of decision making policies.

Scheduling policies in particularly have been crucial to the development of AIM as academic standards evolved from fairness policies such as First Come First Serve (FCFS) to optimisation methods such as linear programming and finally to more heuristic approaches, specifically through the implementation of Artificially intelligent (AI) techniques.

As AI revolutionises algorithmic algebra, rule-based scheduling policies become increasingly difficult to apply to such complex models while heuristic or self-learning methods take the forefront. Reinforcement-learning (RL), and specifically Q-Learning has shown much success in the programming of traffic lights. It offers the advantage of not needing any labelled data while respecting adjustable reward systems. As we shall outline below, this difference is pivotal to the adaptability of Reinforcement Learning models for different intersections and use cases.

1.2 Motivation

A unique characteristic of Q-Learning is the significant data storage cost of the Q-function (post-training). In a centralised system (i.e. traffic light agents), heavy data storage is not a concern given the unique locality and space of storage. On the other hand, an unsignalised intersection requires CAVs to store scheduling functions locally. Despite the proven success of RL in the realm of AIM, its application to distributed and unsignalised AIM models demands the consideration of the trade-off between the minimisation of data costs and the efficiency of a Q-function for AIM. While very few attempts to apply RL to such a system have been made (possibly for this very reason), those that have do not consider the impact of this trade-off on real world applications.

1.3 Aims

My hypothesis is that the minimisation of data storage in Q-functions could be implemented for distributed AIM without significant concessions in the overall reward (i.e. Average Waiting Time of vehicles).

To investigate this, a simulation framework applying distributed Q-Learning to an unsignalised intersection using CAVs as agents will be be created. The SUMO traffic simulation package lends itself to this purpose. Such a framework will allow for the implementation of distributed Q-Learning with an emphasis on minimising the number of states inside the Q-table. In order to assist with the learning process, Imitation Learning (IL) will be used as a starting point for learning with the Longest Queue First (LQF) as expert and RL for fine tuning. I will analyse the performance of the learning and the resulting Q-function with: (i) an application of Q-learning that makes no attempt to minimise data costs and (ii) standard intersection baselines including *First Come First Served* (FCFS) and centralised RL applied to traffic signals.

1.4 Contribution

The real-world implications of this investigation would be the material viability of Q-Learning as an option for distributed AIM. Data storage costs of resulting Q-functions would be sufficient to envisage local storage in CAVs. With the standardisation of traffic infrastructures, a CAV would store Q-functions for every type of intersection. With the standardisation of traffic infrastructures, it is therefore reasonable to assume that trained Q-functions would suffice to implement near optimal scheduling policies for specific rewards in any AIM context.

1.5 Report Plan

The report will be divided into the following sections:

The Literature Review is designed to complete two objectives. The first is to establish the state of the art for the AIM domain as a whole, noting its evolution and evaluating the impact of both developments and gaps in research. The second is to review and analyse current literature on RL applications to AIM from a critical standpoint in order to assess the benefit of my own contribution with an appropriate contextual framing.

The Design Section outlines the overarching design decisions made to verify my hypothesis. Concretely, this will include the RL algorithms, action/reward design and network choices (intersection parameters and vehicle generation).

The Implementation Section describes the structure of framework code, the limitations of its functionality and the temporal structure of every simulation time step and learning process.

The Evaluation Section considers the learning outcomes of the implementation. This includes proof of concept (demonstrating when learning occurs without analysing its performance), the final performance of my implementation in terms of chosen metrics (training time, learning curve, Q-function size and final average vehicle waiting time) and the evaluation of this performance with chosen baselines (FCFS, traffic light RL and full data distributed RL).

The conclusion consists of a self-review of the extent of veracity of my hypothesis shown through the implementation and the potential measures that could be taken to further develop or exploit this hypothesis.

Chapter 2

Literature Review

2.1 Background

2.2 Aims of AIM systems

In their "seminal" [20] contribution to AIM, Dresner and Stone describe the properties desired in any traffic framework[4]:

- **Autonomy** The autonomy of individual CAVs is crucial to the robustness to failure and the minimisation of computational power.
- **Deadlock/Starvation Avoidance** The eventual crossing of every vehicle in an intersection must be assured, even at the cost of optimal efficiency of other vehicles.
- **Incremental Deployability** The system should allow for human driven vehicles to exist independently.
- **Safety** The system should be designed to ensure that vehicles never collide, even in the case of system, CAV or communication failure.
- **Efficiency** Delay caused by the intersections on all CAVs should be minimised.

Although partly outdated, this list of *desiderata* has been a significant consideration in the selection of Spatial frameworks, scheduling policies, adaptation methods to human drivers and methods of AI implementation. The criteria listed are critical in the evaluation of AIM systems.

2.3 Spatial Frameworks

Although most reservation systems are built from the original tile/grid-based spatial framework, developed by Dresner and Stone [5], it is important to take into consideration that other approaches may be better for specific circumstances. Namely, Zhong redefines four frameworks: intersection-based, tile-base, conflict-point-based and vehicle-based reservation systems [21].

- The Intersection-based reservation system imitates a structure similar to that of stop sign intersections: one car is allowed in the intersection at a time[21]. Despite a lack of efficiency, this model fulfills the deployability desiderata as it can be easily implemented with mixed traffic. It is also useful for a heuristic evaluation of other dimensions of AIM such as scheduling policies.

- The tile-based reservation system discretizes space into a grid, allowing for the computational needs to be adjusted by granularity. It also inherently sets out a geometric mapping for vehicles, simplifying communication protocols and the identification of conflict detection points [5].

- The Conflict Point based reservation system consists of a continuous representation of lanes within the intersection space. Routes are predefined and offer little robustness to route adaptation. On the other hand,

- The Vehicle Based (VB) reservation system seeks to allow CAVs to be guided throughout an intersection regardless of lane boundaries. Although this system is described as "near-optimal" [11], it is inherently computationally expensive given its non-linearity and its research is relatively undeveloped.

2.4 Distributed and Centralised Intersection Management

CAVs have the ability to communicate with each other (Vehicle-to-Vehicle, V2V), road infrastructures (Vehicle-to-Infrastructure, V2I), and even non-motorized road users including pedestrians(Vehicle-to-X,V2X) (Wu et al 2019).

In the context of AIM, this technology allows for the specification of either a distributed or centralized approach to information sharing and decision making.

The former consists of information sharing directly between CAVs so that CAVs can cooperatively establish a scheduling policy while the latter consists of the intervention of

a third party with full information of the intersection to externally coordinate the traffic control. These two approaches are illustrated below:

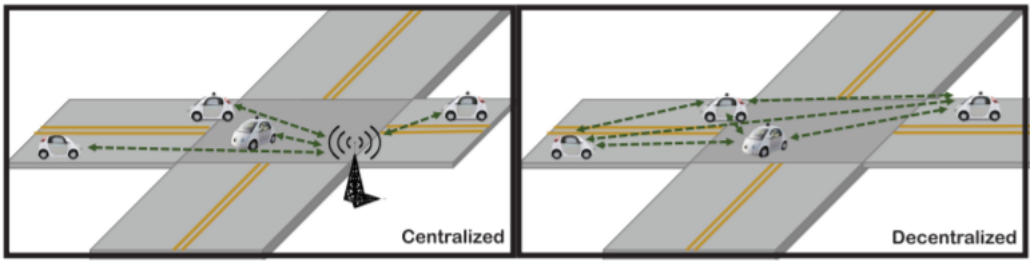


Figure 2.1: Centralized approach to AIM (V2I) and distributed approach (V2V). [8]

2.4.1 Distributed

Distributed AIM approaches are the most commonplace in contemporary approaches to AIN due to the following advantages:

- **Scalability**
  Distributed systems are able to function regardless of any external infrastructure. This means that CAVs may function efficiently in the countryside as efficiently as in urban areas.

- **Cost efficiency**
  CAVs are more autonomous in distributed systems and no long need the support of an external station. This saves the construction and maintenance of control stations.

- **Robustness**
  A dangerous risk for centralized systems is the failure of the controlling station. With the added autonomy of distributed systems, CAVs are able to detect danger and function correctly even if one CAV malfunctions.

2.4.2 Centralized

Although centralized AIM was the norm in initial approaches, it has become outmoded due to the scalability issues that came with V2X technology. When the potential for thousands of agents became possible, the concern of managing increasing numbers of agents became dubious. Nevertheless, the ideas related to this approach have potential to influence the future of the technologies.

Centralized AIM is based on two strategies:

- **Query based centralized AIM** consists of the querying of CAVs to the intersection manager (IM) with a trajectory and and velocity. After calculating the positioning of the car and verifying that there are no given collision points. If the course is rejected then the CAV will decelerate and query again after a timeout has elapsed. This system thus works in a quasi-random selection of prioritisation as CAVs must depend on the fortune of their time of querying. In addition to the poor fairness of the system, the time delay cannot be optimised due to the random parameter and there is a high risk of cars coming to a complete stop during busy intersections.

- **Assignment based centralized AIM** concerns the request and assignment of CAV courses directly from the IM to CAVs when approaching an intersection. This approach therefore gives more control to the IM to optimize its scheduling policy. On the other other hand, the processing time of the IM is long with assignment based approaches.

2.5 Scheduling Policies

Scheduling policies are defined by the fundamental prioritisation belief that drives them: fairness, optimisation and heuristic beliefs[8].

Dresner and Stone set the standard of **fairness-based** scheduling policies with the First Come First Serve (FCFS) policy [4]. As the name suggests, this policy consists of a queuing format where the first vehicle to arrive at an intersection has priority over any other.
Fairness based policies have a tendency to imitate human behaviour in a way that does not fully exploit the potential efficiency developed through AI.

Optimisation policies are rule based policies that aim to reduce the overall delay of all vehicles arriving at the intersection. They generally consist of the modelling of vehicle arrival times and the algebraic resolution of delay minimisation from these models [21]. The issue of modelling traffic on the other hand is difficult to resolve as traffic varies at an hourly, daily, monthly and yearly cycle. The adaptation of the model to given traffic circumstances is essential to best verify the efficiency criteria of a traffic system.

Heuristic based policies compromise between fairness and optimisation by seeking a satisfactory result given external information. This can be the congestion of specific lanes, vehicle prioritisation or game-theory based individual intentions [2]. The Longest Queue First (LQF) policy considers the former. Despite being less common than FCFS, this policy has shown to be significantly more efficient [20].

2.6 Vehicle prioritisation

Vehicle Prioritisation is closely linked to the adopted scheduling policy but takes on another dimension: the differentiation of vehicles. Dresden et al. first considers prioritization of emergency vehicles through the The FCFS-Emerg Policy [4]. This policy is an adaptation of the FCFS policy whereby all the vehicles in the lane of the emergency vehicle are granted priority access.

Lam et al. tackle the concept of Emergency vehicles through an auction based scheduling policy [10]. This heuristic policy consists of every CAV calculating an individually optimized crossing plan for themselves. CAVs would then contest for the most efficient overall crossing plan. In the context of emergency vehicles, the crossing plan of the emergency vehicle would have priority over all others regardless of overall optimization.

Makarem and Gillet [15] consider prioritization beyond of emergency vehicles by taking into account the inertia of vehicles. This is a more realistic application of optimization policies that, for example, would give priority to a heavy lorry, given the increased deceleration/acceleration cost of time and fuel. The results of Gillet and Makarem show that their model is slightly less time efficient than most optimization scheduling policies but show a significant increase in energy consumption and traffic smoothness.

2.7 Mixed Traffic implementations

The *Incremental Deployability* consideration requires the accessibility of Autonomous Intersections to Human Driven Vehicles (HDVs) in a *Mixed traffic Scenario*. Several solutions are considered that seek the implementation of optimisation techniques through HDV and CAV cooperation.

Lane following, similar to platooning, consists of the batching of HDVs behind a CAV. The idea is often related to motor way driving and cruise control given its suitability to open roads however it has been shown to also be effective in AIM under certain circumstances. Peng et al. [16] apply this model a figure of 8 style traffic network demonstrating a clear increase of efficiency over time through learning. Platooning, however, requires a minimum number of CAVs to function efficiently and a complex system to coordinate CAVs and HDVs. Moreover, the variability of HDV behaviour renders the risk much more significant.

A more realistic approach to mixed traffic AIM to the Shared-Phase-Dedicated-Lane (SPDL) based intersection control [14] . Ma et al. designed a traffic light based system in which CAVs use dedicated lanes giving them the ability to go through red lights given the information of possible safe passage. Although the experiments were applied to classic 4 lane intersections (including 1 CAV dedicated), the authors claim that there is much

potential for adaptability to different numbers of lanes and traffic scenarios.

2.8 Pedestrians in AIM

The problem of pedestrians crossing at an autonomous intersection without traffic lights has been scarely considered until Wu et al.'s proposed solution in January 2022[19]. This paper invents the *automated pedestrian shuttle* (APS) to transport pedestrians through designated shuttles. The solution is implemented in a two lane intersection where four pedestrian nodes are described at each corner of the intersected, as represented in Figure 1 below.

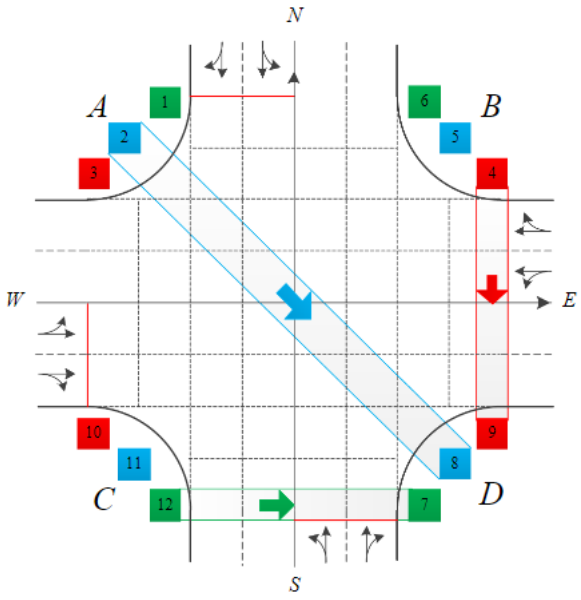


Figure 2.2: Screenshot of sumo Simulation *Cross Network* [19]

The solution proposes a conflict detection scheme founded on the grid based reservation system that involves shuttles and pedestrians simultaneously. It seeks to optimise the delay for both pedestrians and CAVs however does not consider the quantity of passengers within a shuttle. Although this is problematic for efficiency, it is a convenient solution to the deadlock/ starvation goal of the desiderata as the shuttle will always be considered as long as there is a single passenger. This trade-off is also beneficial to the deployability goal as it conserves the current policy of giving priority to pedestrians.

2.9 Reinforcement Learning

As seen in the above sections, the modelling of intersections is becoming increasingly complex as they include modern considerations such as pedestrianisation, vehicle categorisation (in terms of acceleration abilities and socially defined priorities) and mixed traffic implementations. A contemporary solution to this question of over-complexity is the definition of scheduling policies through Reinforcement Learning (RL).

2.9.1 Markov Decision Processes

RL can be applied to Markov Decision Processes (MDPs) to define near-optimal algorithms for specific reward functions [**RL theory**]. A MDP is a model consisting of:

Agents are actors who's behaviour modifies the environment they're in. Specifically, these actors attempt to use their behaviour to change the environment in a specified way.

States refer to the observed state of the environment from the point of view of an agent. The information contained in a state is all the information an agent has on its environment before implementing an action.

Actions consist of the possible actions that an agent can take in order to change its environment purposefully.

Rewards refer to the feedback that an agent receives on the change that its actions have had on the environment. Essentially, this is informing the agent whether its action in the given state was positive or negative and the extent to either.

The classic example used to describe MDPs is its application to human behaviour: When a human (**Agent**) is hungry (**state**), it will therefore eat (**action**) and receive gratification (**reward**).

2.9.2 RL algorithms

RL is an intermediate between supervised and unsupervised learning. As the knowledge it receives is limited to the reward (a number), it must itself **explore** the environment to seek out action paths that return the greatest return. On the other hand, an agent must weary of the volatility of rewards as the environment changes. Indeed, the reward for the same state/action pair may vary from outside factors. It is therefore important to strive toward maximal reward, **exploiting** its knowledge of the system, thus checking

that the state/action pair is still the best possible solution. This compromise is managed in different ways for different algorithms:

Brute force is the algorithm consisting of testing every action from a state and finally selecting that with the highest reward. This algorithm can be very computationally demanding and can lead to local minima if the reward function is volatile.

ϵ-Greedy consists of the direct application of the exploration/exploitation trade-off: with probability ϵ, the algorithm will choose a random action and with probability $1 - \epsilon$, the algorithm will choose the best action. Thereby, it will keep exploring while maintaining the optimisation of reward at an extent set by ϵ.

While the above algorithms describe the ways to select an action path, Q-learning describes how to process and store rewards for state/action pairs in a way that is most likely to reflect both the history and the potential of state/action rewards in order to accurately demonstrate its overall value, or more specifically, its Q-value.

Q-Learning

In Q-learning, Q-values are stored and updated in a Q-table. This table consists of the states on one side and the Q-values for every item in the action-space on the other. This, as well as the training process is shown in the Figure 2.3 below:

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 2.3: Q table [20]

As shown in 2.3, the final table or Q-function is particularly heavy in data storage given the large number of states in many common use cases.

The updating of Q-values is achieved by the equation below:

$$Q_{new}(s_t, a_t) = Q_{old}(s_{t-1}, a_{t-1}) + \alpha * (r_t + \gamma * maxQ(s_{t+1}) - Q(s_t, a_t))$$

Here, the variables are defined as:

$Q_{new}(s_t, a_t)$ refers to the new update of the current Q-value being processed.

$Q_{old}(s_{t-1}, a_{t-1})$ refers to the last version of the current Q-value being processed.

$maxQ(s_{t+1})$ refers to the maximum Q-value out of all the actions possible from the current state. This shows that the Q-value includes not only rewards but the expected potential rewards as well.

α **learning rate** This parameter is a multiplier that defines the extent to which a new action/reward input can change the Q-value of a specific state. If the learning state is high then the Q-value may reach it's *final value* more quickly however it would be more volatile and susceptible to local maxima.

Γ **discount rate** Also a multiplier, gamma defines the impact of estimated *future potential* achieved through the action and the state that resulted from this action. In terms of Q-learning, this future potential refers to the maximum reward stored for the new state.

2.10 Implementations of DRL in AIM

The significant advancements of Deep Learning since 2012 have revolutionised modelling in almost all domains of engineering. Artificial Intelligent techniques such as Deep-RL also show much potential to solve complex modelling issues such as those discussed in traffic modelling.

Although few applications of DRL have been implemented in unsignalised AIM, Deep-RL has shown to be very effective with signalized intersections [6]. Huo et al. apply imitiation learning (IL) for pre-training and DRL for fine tuning their model. The framework is applied to networks between 1 to 9 intersections and varying traffic flow rates (including unbalanced flows). The quantitative analysis is thorough, including measures for queue length, average vehicle waiting time and average vehicle fuel consumption for the experimental model, a predefined optimization scheduling policy and two fixed-time policies.

2.11 Overview

Throughout this section, we have subtly examined the factors showing the significant gap in research regarding the implementation of Q-Learning for distributed AIM. Indeed, the domain preference for distributed systems over centralised systems suggests a progressive rejection of classic traffic light control and its replacement with modern AI based scheduling policies as CAV technology becomes increasingly commonplace in society. Reinforcement learning, a new and promising branch of AIM that has scarcely been implemented for unsignalised traffic intersections is a powerful tool for creating models too complex for algebraic resolution. With the added complexity of a distributed system added to the modern considerations of pedestrianisation, spatial frameworks, vehicle prioritisation and mixed traffic implementations, the viability of distributed RL applications to unsignalised traffic intersection must be resolved.

2.12 Closely-Related Projects

2.12.1 Success of RL in Traffic light control

In the work achieved by Liu et al. [12], the intersection model is reduced to a Markov Decision Process in order to implement reinforcement learning. Here, the **agents** are traffic lights, the **reward** is the average waiting time and the actions are the setting of the next light configuration, as shown in Figure 2.4 below:

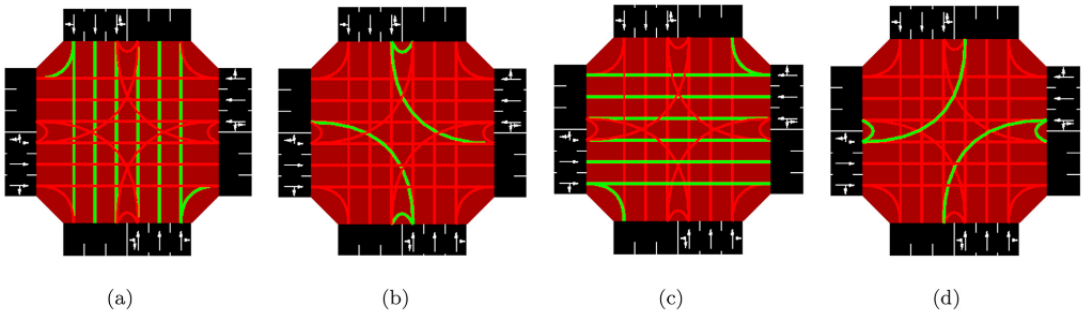


Figure 2.4: Action Space for traffic light control: light configurations [12]

Indeed, the training of the model was shown to be very effective with the final performance shown in the Figure 2.5 below:

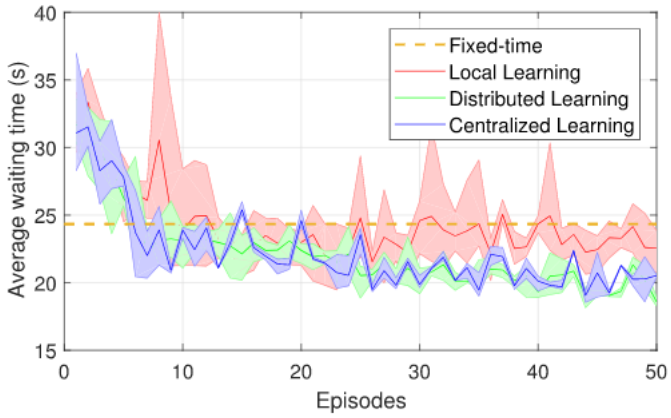


Figure 2.5: Training performance of different RL applications with *fixed time* traffic lights as a baseline[12]

Through this figure we can observe the success of RL implemetations in traffic signal control specifically for centralised learning. Specifically we see three types of RL used:

Local learning refers to sets of traffic lights that only have access to the traffic observations of their junctions

Distributed Learning refers to the sharing of information between independent agents each learning from this information

Centralised Learning refers to all sets of traffic lights being *slaves* to a *master* agent intersection manager

Note: In this paper, the term *Distributed Learning* is used to refer to shared learning between different sets of traffic lights. In the context of this report, we will use the term *Distributed* to refer to models where CAVs are learning.

2.12.2 RL applications unsignalised intersections

Wu et al [20] describe the Decentralized Coordination Learning of Autonomous Intersection Management (DCL-AIM). This approach consists of a reinforced learning application to AIM in the same tile-based intersection type as described by Dresner and Stone.
They use the tiled framework to implement a tile reservation protocol. When the reservation protocol returns a potential conflict, a flag is raised:

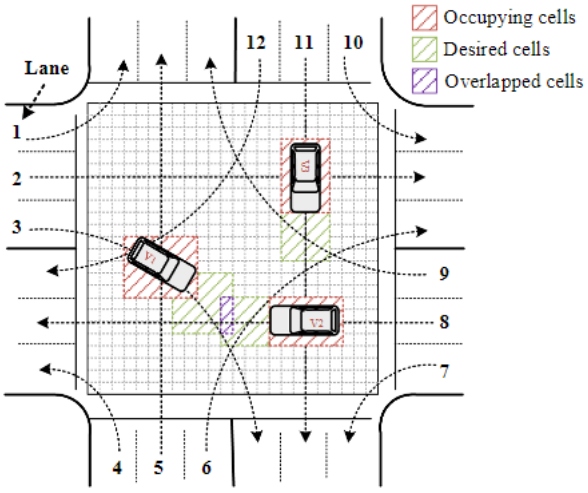


Figure 2.6: Intersection diagram in the framework created by Wu et al. [20]

When the flag is raised, the MDP model is considered where the components are:

agents are CAVs coming into conflict

observations are *current position, speed, moving intention and the queue length of its current lane.*

actions are m acceleration rates

reward is set as the negative of the vehicle delay for the given CAV

The paper as a whole included a thorough critical analysis of its results including learning effectiveness and final performance however fails to consider real world obstacles such as data storage costs.

Moreover, while the results were successful in terms of learning performance, the final performance still remains close to the Longest-Queue-First baseline, as we can see in Figure 2.7 below:

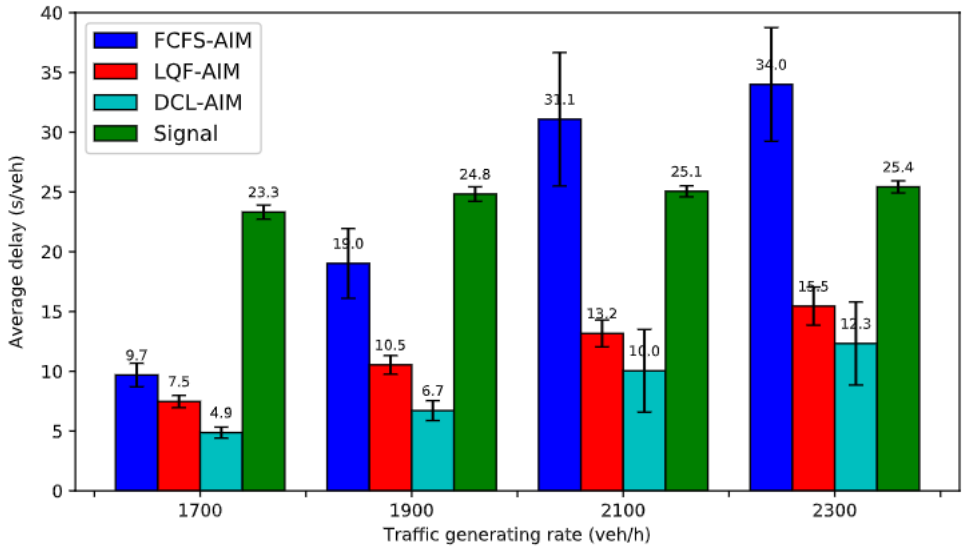


Figure 2.7: Final performance of Wu et al.'s Distributed RL application to AIM [20]

We can observe that in this application, the authors use a reward function benefiting a specific CAV. However, the final performance metrics consider the intersection as a whole. Here, the authors show how individualistic actions can benefit the environment as a whole. However it does not consider whether communal actions can benefit better.

2.13 Summary

While Reinforcement Learning has been scarcely applied to unsignalised intersections using individual CAVs as agents, such applications have consisted of specific conflict resolution frameworks between two CAVs at a time. The case of an agent inside a MPD environment similar to that of traffic light control applications, with a communal reward benefiting the junction as a whole rather than specific CAVs has not been considered. On the other hand, the storage of Q-functions inside CAVs is a significant real world consideration for distributed AIM unlike that of traffic light control. The minimisation of data storage in a Q-function would therefore be instrumental to the real world viability of such a design.

Chapter 3

Design

Over the course of this chapter, we will explore the design choices that went into the implementation of my distributed Q-learning application to AIM. The objectives of these choices are to:

1. Reflect real life traffic scenarios
2. Efficiently apply Q-Learning with minimal data in the resulting Q-function (thereby validating my hypothesis)
3. Put my implementation in context with existing AIM Q-Learning applications.

The design choices that will be covered concern the creation of an appropriate intersection model, the RL algorithms used (including rule based boot-strapping) and the Q-Learning model that minimises the resulting Q-function.

3.1 Traffic intersection model

Assessing which date is collected and processed in traffic simulations is pivotal to the efficacy of the learning process. To this end, the configuration specifications of a network have a significant part to play. They will define not only the foundation from which a RL model can be built but also the data that stems from it. Case in point: the length of the lanes leading to a junction changes the degree of relevance of any average taken of the simulation as a whole when interested only in the junction itself. In this section, we will cover the design choices of the intersection model and discuss the consequences these may have.

3.1.1 Intersection type

In order to consider the baseline application of Q-Learning with full data usage ([20]), we must adhere to its network parameters. Specifically, this application uses a 3 lane intersection as shown in the figure below:

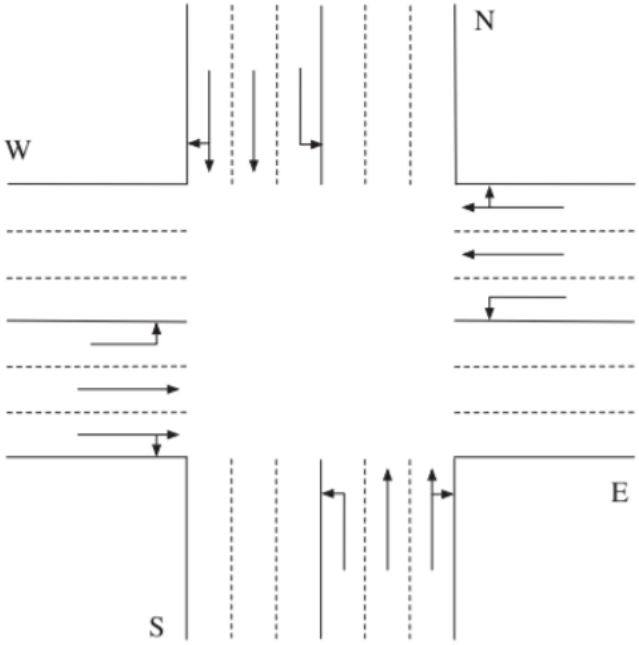


Figure 3.1: Diagram of a 3 lane intersection

This intersection is indeed an archetype of traffic infrastructure and therefore lends itself well to the imitation of real life where traffic intersections are increasingly standardised. It also offers three types of routes: straight, right turn and left turn. The characteristics of these routes are unique between each other as their traffic flows are impacted by different factors.

The particular factor that comes into play most significantly in the design of the Q-Learning model is that of the *foes*. The foes of a particular route are defined as all other routes that could potentially come into conflict with it. Namely:

The foes of the straight route are both straight routes emanating from the lanes perpendicular to it and left-turn routes emanating from the opposite side of the junction and from the right of the concerned CAV. It has a total of four foes.

The foes of the left-turn route are straight and left-turn routes emanating from all other directions. With a total of six foes, this route is the most likely to cause congestion.

Right hand turns are for all intents and purposes independent of outside traffic. Although the right hand turn can be considered superfluous to RL given its independence, it can serve as a valuable baseline to show learning by reciprocity: if left-turn and straight routes become more efficient while right-turns remain the same the learning has been successful.

3.1.2 Vehicle Generation

The way that vehicles are generated throughout simulations has a significant impact on the learning outcomes of RL:

- Vehicle generation with a larger **standard deviation** will inevitably benefit the learning process as the RL model will be exposed to more variability in traffic circumstances. On the other hand, this might reduce the speed of learning for specific flow rates as vehicles observe the same observed states more infrequently. Moreover, higher variability hinders the observability of learning over time as observed metrics are less dependant on performance and more dependant on circumstance. Case in point: the average waiting time of vehicles will be higher with a bigger influx of vehicles regardless of whether the scheduling policy better knows how to deal with it.

- Conversely, learning stemming from vehicle generation with a low standard deviation will be limited to a certain type of traffic flow (that of the predefined flow rate). While this is less beneficial to traffic management as a whole, it demonstrates more acutely the process of learning for specific traffic scenarios.

The choice of standard deviation in our vehicle distribution therefore relates fundamentally to the objectives set out for this project: proving the **viability** of Q-Learning with our given model. In this context, a lower variability factor is more appropriate and can be appropriately adjusted after proof of concept.
In order to verify goal 1, we will now consider distributions that traditionally represent traffic flow.

Poisson Distribution The most commonly used probability distribution to emulate traffic flows is the Poisson distribution. This distribution assumes the following are true [9]:

- The arrival of a vehicle does not affect the probability that a second arrival might occur (ie: vehicle generations are independent of each other).

- The arrival rate of vehicles is theoretically independent of outside factors however practically may change over time.
- Only one vehicle can arrive in the same lane at the same time.

These assumptions are verified for most traffic scenarios, thus validating the appropriateness of the model.

Binomial Distribution Given that the Poisson distribution is the limit of a binomial distribution where trials are considered continuous, the binomial distribution of inserted vehicles approximates a Poisson Distribution for low probabilities [17]. This is particularly valid when considering the discretization of simulation steps as a Bernoulli process.

3.2 Choice of RL algorithms

Beyond the analysis of successes and failures of AI algorithms in AIM considered in the literature review (specifically, section 2.10), the choice of Q-learning was made among various other RL variations. In this section, we will see why Q-learning is particularly suitable to the application and consider the specifications of its implementation.

3.2.1 Overview of system requirements and RL algorithms

As established in the literature review, RL lends itself well to AIM where the environment and action space can be stated as a Markov Decision Process (MDP) but the complexity of the environment renders its concrete algebraic modelling near impossible to define.
The use of a simulator such as SUMO refines the criteria of the RL solution to implement. Simulators with discrete time steps offer speed and variability with regards to the circumstances of the MDP. Moreover, we are able to define a realistic environment with a large number of states and changeability in the environment. This signifies that the standard deviation of rewards for a same state will be relatively large. Algorithms such as brute force or Monte Carlo algorithms which have high computational costs are therefore likely to waste resources on local minima. Q-learning and the application of the temporal difference algorithm to define Q-values, however, takes the unreliability of data into account. By choosing to perform analysis of a single action at a time, it enables longer training and the encounter of more simulated scenarios.

3.2.2 Q-learning variables

The Q-learning variables defined in section 2.9.2 play a significant role in the refining of the Q-learning model. In order to refine this model to optimise learning by trial-and-error,

these variables are arguments in the calling of our system.
Recalling the variable definitions from the Q-value equation:

$$Q_{new}(s_t, a_t) = Q_{old}(s_{t-1}, a_{t-1}) + \alpha * (r_t + \gamma * maxQ(s_{t+1}) - Q(s_t, a_t))$$

α **learning rate** This parameter is a multiplier that defines the extent to which a new action/reward input can change the Q-value of a specific state. If the learning state is high then the Q-value may reach it's *final value* more quickly however it would be more volatile and susceptible to local maxima.

Γ **discount rate** Also a multiplier, gamma defines the impact of estimated *future potential* achieved through the action and the state that resulted from this action. In terms of Q-learning, this future potential refers to the maximum reward stored for the new state.

Recalling the variable definitions from the $\epsilon - greedy$ algorithm:

Epsilon ϵ This variable defines the likelihood of choosing a random action of the action space (exploration) or of choosing the action with the biggest chance of success.

Decay rate While larger values of epsilon are favourable at the start of a simulation to explore all options, it is also advantageous to set a low epsilon after all actions have been considered to verify the accurateness of the maximum Q-value action. For this we define a decay rate as a multiplier to ϵ : with every learning cycle, the value of ϵ gets smaller. From a macroscopic point of view, this represents a transition from a state of exploration to a state of exploitation of the system.

ϵ **Minimum** While exploitation is favourable in the later stages of Q-learning, total exploitation goes against the principle of Q-learning: actions other than that of the maximal Q-value will no longer get the opportunity to update their Q-values and compete with the maximal reward action. To ensure that learning is still happening, we set a minimal ϵ so that some exploration is still possible.

By calling the arguments every time we train the model, the variables can be comfortably tuned depending on learning performance.

3.2.3 Rule based imitation learning

Imitation Learning, as defined by Huo et al. [6] is the learning and updating of Q-values through the imitation of an *expert*. While this definition strays subtly from the usual definition of learning as the imitation of human behaviour [7], it puts forward an interesting

idea: the implementation of a *rule based* scheduling policy to start the learning process from an advanced position. Indeed, Huo's implementation consists of this: imitation learning followed by fine tuning through Deep RL. Algorithmically, this implementation is relatively uncomplicated, with the replacement of the $\epsilon - Greedy$ algorithm by a rule based algorithm until a certain time (or simulation step) T. At this point, the Q-function will have learnt the benefits of the rule and be able to refine the model where the rule is sub-optimal.
The decision of which rule to is therefore crucial to the final performance of the Q-function. As it is a starting point for learning, the rule need not be particularly computationally costly. In the works of Wu et al. [20], both FCFS and Longest queue first are used as baselines, shown in Figure 3.2 below.

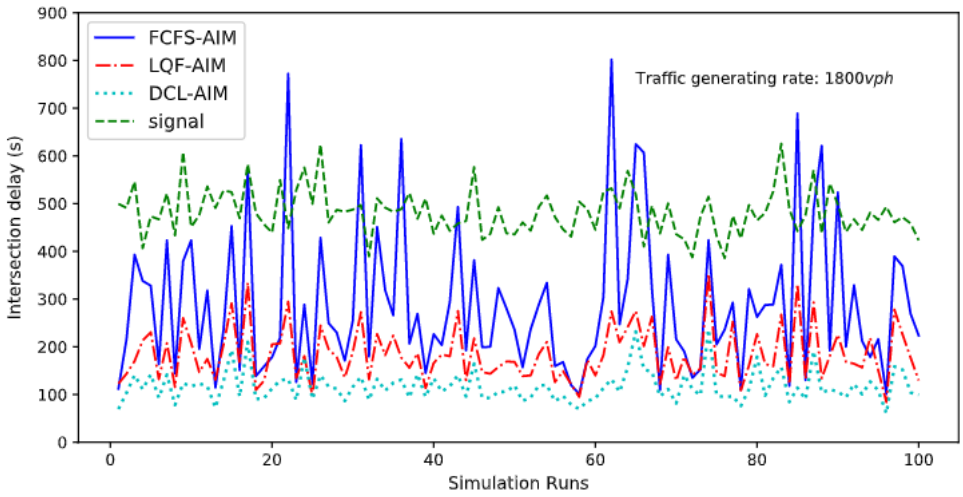


Fig. 10. Fluctuation of intersection delay of 100 simulation runs.

Figure 3.2: Graph of the Q-function results of Hu et al. with LQF, FCFS and traffic light scheduling policies as baselines

As shown in Wu's results, FCFS performs very well, almost matching their own implementation of RL. This rule is therefore chosen as the basis learning of our model.

3.3 Q-Learning Model

3.3.1 Definition

As defined in section 2.9.2, Q-Learning models are defined by agents, action spaces, observation spaces, the reward function and action/reward time intervals. In the context of distributed AIM, each of these actors have their own subtleties as described below:

The Agents : In the context of unsignalised AIM, agents are CAVs themselves. In the learning process, they both receive actions and learn from these actions. In real life applications, CAVs use the q-function to establish the best course of action.

The Action Space : Whereas a centralised model might be able to explicitly decide which CAVs have priority over which and grant them access, a distributed model must define the priority of the intersection in a more implicit manner where specific agents take action rather than every CAV as a collective.

The Observation Space : This factor defines the contextual information that is relevant to the decision making process. It is highly dependant on (i) what the agents actions are applied to and (ii) the type of actions themselves.

The Reward Function : This function defines the extent of success of the applied actions and applies it to the generated Q-values. For efficient learning, a clear and significant correlation must exist between the action space and the reward.

Action/Reward Interval: The time interval between applying an action and assessing the impact that this action has had on its environment is crucial to the learning process. While the effect of an action may achieve its maximum at different moments in time depending on the environment state and the action in question, a rule must be made to seek out this maximum for all cases.

3.3.2 Definition of factors and model minimisation

While the closest rival to this project (DCL-AIM [20]) takes every CAV as agent and applies controlled simulations to learn how to deal with conflict, a top down approach is taken in this system, similar to those implemented to traffic light control in section 2.12.1. In this case, the general state of the intersection is considered rather than any specific conflict resolutions.

Agents When considering the state of a junction, the behaviours of every CAV in incoming lanes do not carry the same significance. Rather, the CAVs at the front of a lane have the largest impact on both the next state of the junction and of the CAVs behind them in the lane. As seen in section 2.7, the RL lane-following solution put forward by Peng et al. [16] offers an elegant method to minimise model complexity while retaining significant control over all actors of the intersection. Reducing the number of agents to one per lane also has the advantage of simplifying the observation space as the environment is more specific. Case in point, the positioning of the CAVs with regards others in the lane is no longer a factor to be considered: it is always at the front. **We will therefore define agents as the leaders of every lane incoming to the junction.**

Actions The definition of actions is likely the most controversial and changeable factor in such a system. Unlike in RL application to traffic lights, actions in a distributed system have the challenge of having to rival other environment factors in terms of impact on the junction. In other terms, the consequences of the actions applied to one or several leading CAVs must be sufficiently noticeable so as not to be *drowned out* by changes in environment stemming from traffic flow not affected by the actions. Moreover, the size of the action space has a proportional relation to the speed of learning: the large the number of possible actions implies a longer learning rate. The action space must therefore be minimised as is required for an appropriate learning rate. The chosen action space has evolved over various trial-and-error implementations. Initially the action space was the setting of acceleration of a given agent to different rates (i.e., 3 acceleration rates are "decelerate - remain the same - accelerate"). Thereby, with relevant information on the state of the junction and incoming traffic, a CAV would learn to adjust its speed accordingly. However, and even with varying numbers of acceleration rates (3, 5 and 7), this action space proved ineffective to learning. While this solution had theoretical promise, its failure to learn is attributed to an insufficient impact on the junction. The final chosen action space is more similar to that of the conflict resolution system implemented by Wu et al.[20]. It consists of the election of an agent among its foes. The elected agent accelerates

or has some form of priority advantage against its foes depending on the implementation restraints. CAVs on a *straight route* and those on a *left turn* route each have four *foes* so the action space will therefore be an integer between 0 and 4, thus selecting the CAV to accelerate.

Action/Reward Time interval The time interval between action and reward has a significant impact on agent learning. It must seek maximisation of effect on the junction. While former attempts to apply distributed Q-Learning set this interval to a constant parameter Δ seconds or simulation steps, a more effective attribution of the interval is acquired by a rule based trigger. Namely, the maximum effect of the action can often be found as the acted-upon agent vacates the junction. The computation of reward is therefore best triggered upon this event. The application of this trigger in context of a Q-Learning cycle is shown in figure 3.3 below:

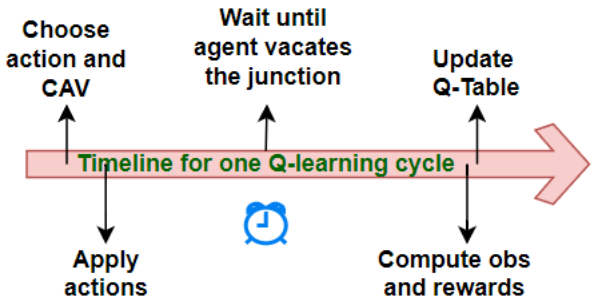


Figure 3.3: Timing Diagram of a Q-Learning step

Observations The observations that influence the decisions of an agent must be those most relevant to the desired reward. Namely, if the goal is to resolve conflict with minimal overall congestion within the junction then the relevant information of incoming traffic is:

- The vicinity of incoming lane leaders to the junction
- The speed of incoming lane leaders
- The vehicle density of incoming lanes

Note that the same *leader following* approach is taken for observations: the speed and position of every CAV is not always significant. The capture of observations at specific instances define the model *states*. This signifies that fewer observations would lead to fewer states. This contributes on one hand to the minimisation of data storage, and on the other can have a positive effect on the learning process: data gathered on incoming traffic

that does not significantly effect the trajectory of a vehicle is redundant and strains the learning process. Indeed, x states that are differentiated only by superfluous information retain only $\frac{1}{x}$ of the information gathered. The learning process will therefore be x times slower. A further step toward this goal is the consideration of only the information from conflicting lanes (or foes). Any other information can be deemed superfluous.

Reward Although the primary goal of the reward function is to define the metric that is to be optimised through the system, its definition must also take into account the effectiveness of the learning that stems from it. Indeed, the optimisation of one metric in the short term might have greater repercussions for another metric in the long term. For example, the minimisation of fuel consumption involves penalising vehicle breaking. A system that aspires to reduce the congestion of a junction would also seek to conserve vehicle momentum by penalising vehicle breaking. Thereby, the desired outcome may be very similar while learning characteristics may differ. For this reason, and with the overall objective of minimising the average waiting time of vehicles, the following reward systems can be implemented and contrasted:

Accumulated Waiting Time This refers to the total delay (sum of the waiting times of all approaching vehicles) changed in relation to the previous time-step. With the waiting time defined as *"the time in which the vehicle speed was below or equal 0.1m/s"* [18], this reward function does not learn efficiently unless there is sufficient congestion in the junction. More significantly, however, it is not proportional to the number of vehicle in the simulation. Its implementation is therefore limited to cases where the flow rate of vehicles is constant. As seen in section 3.1.2, this might be sufficient for the proof of concept of my hypothesis, where the standard deviation of vehicle generation can be low.

Number of vehicles under a specified speed threshold This reward offers a solution to the latter problem where the user can define an appropriate cutoff depending on the congestion of the junction. Again, this solution is only appropriate for constant vehicle flow.

Average speed within the intersection Although this reward is normalised with regards to the number of vehicles (and therefore applicable to varying flow rates and congestion), it strays further from the desired goal due to the lack of consideration of elements such as momentum. For example, it would not be ideal for the same Q-value to be rewarded to an environment where one vehicle is at a standstill and one is at maximum speed and an environment where both vehicles are at medium speed.

Average fuel consumption As described in the example above, this solution penalises breaking, which in turn favours the minimisation of waiting time. On the other hand, fuel consumption occurs most during acceleration which often continues after vehicles leave the junction and may continue after this reward has been computed. The tailoring of the action/reward interval therefore risks significant complexity.

3.3.3 Agent taxonomy and learning implications

Shared learning is the sharing of Q-tables between different agents of the same type. This process involves exploiting and updating Q-tables. The fundamental requirement for this process to be appropriate is that the states and actions be considered in the same way. In other words, the observations of two different agents are for all intents and purposes considered to have the same significance.

As establish in section 3.1.1, every CAV will have its own set of foes depending on its route and starting point. The observations established in 3.3.2 are specific to the lane of an agent and its foes. While left-turn routes are well suited to *shared learning* between them (given that they all consider the same set of foes), the structure of the states must present the observations in a coherent manner to be interpretable in the same way for all agents. Namely, the ordering of the observations must consider the context of the CAV as shown in the example below:

```
(a)
State = [Obs_'lane_x', Obs_'lane_y', Obs_lane_z']
(b)
State=["Obs_from_lane_to_my_right",...
...,"Obs_from_lane_across","Obs_from_lane_to_my_left"]
```

Listing 3.1: Examples of (a) false and (b) correct structuring of states for successful shared learning

Similarly, straight routes have specific foes that contribute to the retrieval of observations. By the same contextual structuring of observations, all straight routes can apply shared learning between each other.

Q-Learning can therefore be applied to each route as a collective of agents. The CAVs (agents) receive observations on their particular foes, structure these observations as a *state* that is structurally uniform for all agents on a particular route and update the Q-table accordingly.

3.4 Summary

3.4.1 Overview of the design

To summarise all aspects of the distributed Q-Learning AIM framework design:

The network design consists of a three-lane four-way intersections. Three possible route types are therefore possible: right-turn, straight and left turn. Vehicle generation will be constant as long as the goal is to verify viability of the system. In any other context, vehicle flow should be set to a Poisson Distribution.

Q learning is to be implemented firstly with LQF learning followed by ϵ-Greedy learning. Input arguments for the trainig stage should be: Q-value variables (α and γ), ϵ-greedy variables (initial ϵ, decay rate and final ϵ) as well as the transition time T from imitation learning to Q Learning.

The learning model has leading CAVs as **agents**, a *foe election* as **action**, the vacating of the junction of the acted upon CAV as **the trigger for learning**, position of and speed of leading vehicles and density of incoming foe lanes as **states**, a selection of choice **rewards** and **shared-learning** for all agents on the same route type.

3.4.2 Pseudo code

```
Initialise environment
Initialise Q-agent types
    Declare Q-tables, action space, observation space
While Vehicles generated:
    Update the current agents
    Choose agent at random
    state = Get observations (agent)
    if time<T:
        Apply action: LQF
    else:
        Apply action: epsilon-greedy
    Wait til agent leaves the intersection
    new_state = Get observations (agent)
    Compute reward
    Compute Q-value (state, action, new_state, reward)
    Update Q-table
```

Listing 3.2: Pseudo code describing the RL algorithm applied to distributed AIM

Note: Given that an action changes the state of the environment, the choice of CAV to act upon should be taken at random. This avoids the bias that comes with cyclical CAV selection.

```
Foes = [Acting agent, lane 1, ..., lane n]
Dense[] = Get_density(foes)
LQ = max(Dense).index
Accelerate (LQ)
```

Listing 3.3: Pseudo code describing the LQF action selection

```
Foes = [Acting agent, lane 1, ..., lane n]
Get epsilon, epsilon_min
if np.rand() < self.epsilon:
    elected = Foes.random
else:
    elected = max(q_table[state])
Accelerate (elected)
if epsilon > epsilon_min:
    epsilon = epsilon*decay
```

Listing 3.4: Pseudo code describing the ϵ-greedy action selection

```
For veh in simulated_vehicles:
    if on_junction(veh):
        accumulated_speed = getSpeed(veh)
        junction_count ++
speed_reward = accumulated_speed/junction_count
```

Listing 3.5: Pseudo code describing the computation of reward = "average speed within the junction"

Chapter 4

Implementation

4.1 Tools for implementation: Sumo environment and network configuration

4.1.1 Overview of Sumo

Sumo, an acronym for Simulation of Urban MObility, is a traffic simulation package provided by Eclipse and designed for the implementation of complex traffic modelling and simulation. The package takes into account the many considerations of modern AIM modelling established in the Literature review, allowing users to emulate traffic scenarios with different vehicles classes such as cars, buses, lorries, cyclists, pedestrians and many more. SUMO also supplies many tools for the automation of modelling. A more elaborate example of this is the network importation tool that can be used to automatically generate networks from road-maps. The Graphical User Interface (GUI) is also well designed and snapshots from the GUI will be used for illustrations in this section.

4.1.2 Traffic Control Interface and Sumolib

The Traffic Control Interface (TraCI) uses a TCP based client/server architecture to provide access to sumo [13]. This tool thus allows its users to run simulations and retrieve or set variables mid-simulation from a Ubuntu terminal. It is through this tool that the design can be implemented, specifically to:

1. Run the simulation for **training**
2. Retrieve the desired **observations**
3. Apply the desired **actions**

4. Retrieve information pertinent to the **reward**

Note: The above design tasks are therefore constrained by the commands available in TraCI and the addition python package *Sumolib*. This may impact the specifications of these design tasks.

4.1.3 Network specification

Sumo provides an elaborate framework for designing networks, supplying tools for its automation. The configuration of the network generations consists of the following XML file:

- **mynodes.nod.xml** describes the nodes and its characteristics (the junction and the extremities of each of the four roads leading to it)
- **myedges.edg.xml** describes the road connections between nodes specifying the number of lanes.
- **mycross.net.xml** is automatically generated using Sumo's *netconvert* file. It includes specifications of the entire network design either from the node and edge files or from assumed norms.
- **myvehs.rou.xml** specifies and configurates the traffic generation throughout the simulation
- **mycross.sumocfg** points to the xml files and specifies simulation length and automated output files.
- **gui-setting.cfg** configurates the gui (ie zoom, placement, etc...) This file is optional, as is gui use.

Nodes and edges

As discussed in the design (section 3.1), the length of the edges have an impact on data retrieved from the system as a whole. After a process of trial and error, it was found that a road length of 60m is sufficient to allow for queuing but not so much that it disproportionately skews any measured averages: the junction remains a fundamental consideration at any point in the road.
The node and edge XML files can be seen in appendix .1.1.

Vehicle generation

The mycross.rou.xml file, shown in appendix .1.2 contains the definition of routes and the populating of these routes with *traffic flows*. The definition of these traffic flows is achieved using either arguments:

vehsPerHour which defines a constant traffic flow. In this case it is important to set every route to a slightly different flow rate in order to avoid identical traffic cycles

probability which sets the vehicle generation to a binomial distribution with x probability of generating a vehicle every second. As discussed in the Design section 3.1.2, the binomial model approximates a Poisson distribution for low probabilities. In this case, this is set around 0.07. Note: the Poisson distribution is available in the next release of SUMO.

As can be seen in the rou.xml file (appendix .1.2), the binomial distribution has been commented out but is still available for after proof of concept.

4.1.4 Resulting network

The design of the network can be seen in the GUI capture of Figure 4.1 below:

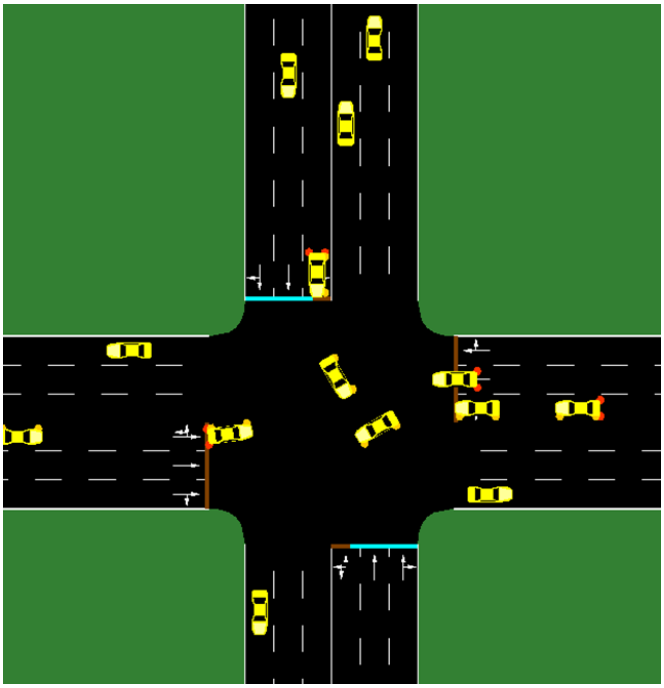


Figure 4.1: Capture of the mycross network in the SUMO GUI mid-simulation with zoom = 300

4.2 Solution overview

This section presents a brief overview of the solution implemented through python classes and a python script.

4.2.1 Classes and coordination

The framework is comprised of 3 classes, coordinated by a "Main.py" script that computes the overall algorithm. These classes are:

SumoEnvironment Modelled from classic multi agent *Gym control environment* [3], SumoEnvironment's main functionalities are to run the simulation, retrieve and save simulation information for performance analysis and to communicate commands between *Main.py* and Lane instances.

Lane The *Lane* class represents instances of every incoming lane affected by our Q-Learning environment. For the sake of agent continuity, these are essentially the delegates to our model's agents. The main functionalities are therefore to retrieve observations from the point of view of the agents and apply actions to relevant CAVs.

QL_agent This class contains the functions and static variables required for Q-learning. It feeds actions and retrieves states and rewards to and from the main algorithm (Main.py).

The coordination, communication channels and main functionalities of each of these classes are summarised in the UML diagram of Figure 4.2 below:

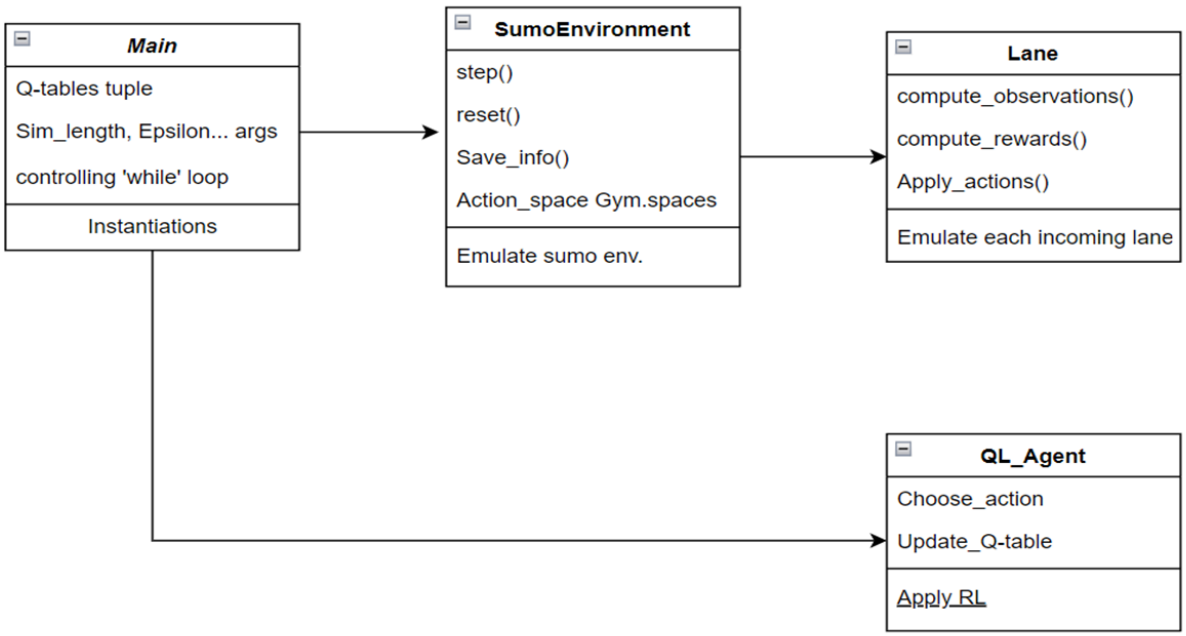


Figure 4.2: UML diagram of overarching class structures

4.2.2 High level implementation of Pseudo code in *Main.py*

The user inputs to Main.py are those described in the design summary (section 3.4.1) with the addition of *–gui* and *–s* arguments. The former, as its name would suggest, allows users to observe the simulation through training while the former defines the time length of the simulation.

The pseudo-code describing the high level model algorithm concluding the Design Section (section 3.4.2) is manifested almost identically in the Main.py script, as shown in appendix .2. The most significant differences are the use of lanes as proxies to agents and the updating of epsilon at the highest level of code. The reason for this is the variability of QL agents: epsilon cannot be updated locally for every QL_Agent instance or it will decay on average at a rate of $1/N_{agents}$ times slower than intended, where N_{agents} refers to the number of agents.

4.3 Reinforcement Learning framework

The Reinforcement Learning framework used for this project is inspired by the *SUMO-rl* framework developed by Alegre, Terry and Kwiatkowski for RL application to traffic light control[1]. Specifically, this work inspired the configuration of class coordination, the sumo-environment class and the agent class.

4.3.1 Overview of SUMO-RL

SUMO-RL provides a framework to apply RL techniques for traffic light control on any signalised network. The RL algorithms applied in experiments include Q-Learning, SARSA and Deep Q-Networks. It itself uses and is based off other known RL packages such as Gym, PettingZoo, stable-Baselines3 and RLib.

This framework implements RL for the following MDP model:

- Traffic lights as **agents**
- Green phase configurations as **action**
- Current traffic light phase, current length of the phase, lane density and lane queue as **observations**
- And *Total Waiting Time* as **reward**

4.3.2 Agent class and *shared learning*

The QLAgent class is designed to host the algorithms and the variables related to the Q-Learning process. Unlike in SUMO-rl, this class is not designed to be completely self contained. To implement *shared learning* between agents of the same route-type (refer to Design section 3.3.3), the Q_table is passed as an argument to and from its functions. Concretely, the class contains two functions:

act() This function applies the epsilon-greedy algorithm to choose an action from the action space. The action is then returned.

learn This function consists of the updating of the route Q_table with the computed Q-value.

To see the precise implementation of these fundamental components of Q-learning, refer to Append .3.

Given the sharing of Q_tables between agents, coordination between each is performed through nested Q-tables. As shown in the Figure 4.3 below, the "Q_table" is the key pointing to the value *route*. The route itself is a key pointing to the states which they themselves point to their associated Q-values.

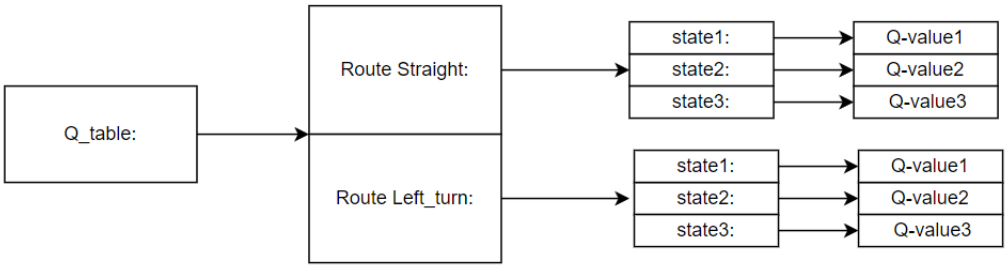


Figure 4.3: Diagram of nested dicts representing shared Q_tables

Through this use of nested dictionaries, the framework becomes scalable to any number of agent types as it becomes possible to iterate through them at any level.

4.4 Implementation of MDP

In this section, the different elements of the Markov Decision Process are specified. Namely: observations, actions and rewards.

4.4.1 Observations: simplification, shared-learning and Lane class Properties

Every lane is instantiated with the list of its inherent *foes*. This being the lanes from which incoming CAVs risk causing conflict. The ordering of this list also has tremendous significance: it is in the order of foe positioning relative to that lane. Ie: starting lanes starting from the right and ending on the left of the lane in question. This will enable the contextual structuring of observations necessary for shared learning, as described in section 3.3.3.
The application of this is seen in the function to get the leading vehicle speed and position (Appendix 11). Indeed, the *foes* list is iterated in contextual order, thus defining leader speed and position in that same order. Hence, the observations and states will maintain this structure defined by contextual position rather than arbitrary name.

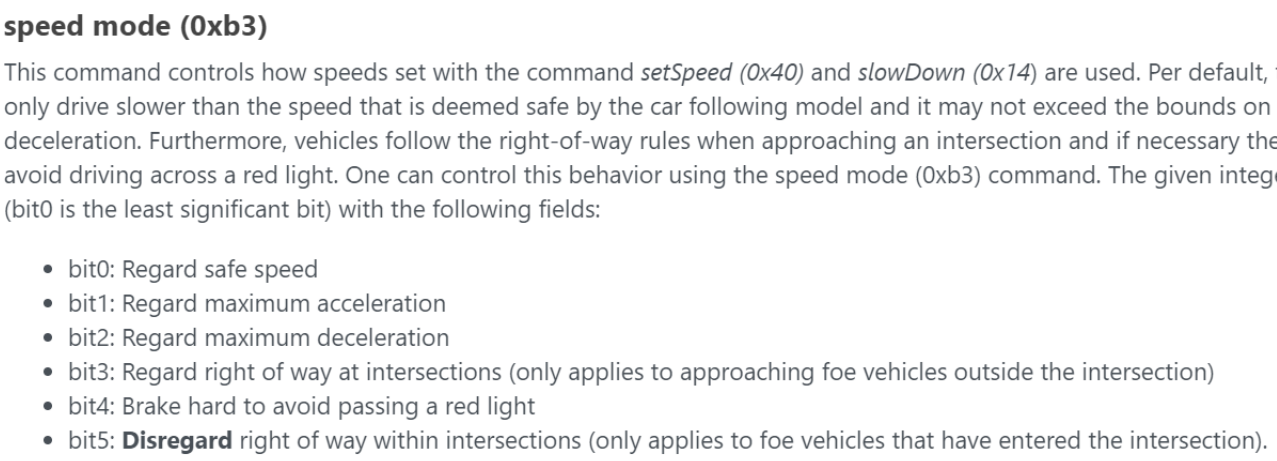
The simplification of data is also achieved in this function: Rather than storing the position of vehicles inside the lane, leaders are ordered as a ranking based from their vicinity to the junction. The result is therefore a list of integers between 0 N_{foes} rather than floats.

Similarly, we can see in the compute_observation() (appendix 11) function that vehicle speeds are reduced to 4 speed brackets, thus reducing the variability of observation. A further difference from the design specifications is the dropping of the *Densities* observation.

This is due to the excessive state variability that came with its representation, even as an integer.

4.4.2 Actions: Applied to LQF and epsilon-greedy

The final action chosen in the design of the framework is that of foe election and acceleration. As described in the Design section (section 3.3.2), this action consists of the selection of one CAV amid its foes to give an advantage to in the potential conflict. The implementation of this action can be seen in appendix .4.2. Here, we see the specific advantages granted: The raising of its maximum allow speed from 14 to 25 and the setting of it's "speed mode". Speed mode refers to a register of checks as defined by Sumo in Figure 4.4 below:



speed mode (0xb3)

This command controls how speeds set with the command *setSpeed (0x40)* and *slowDown (0x14)* are used. Per default, the veh only drive slower than the speed that is deemed safe by the car following model and it may not exceed the bounds on acceler deceleration. Furthermore, vehicles follow the right-of-way rules when approaching an intersection and if necessary they brak avoid driving across a red light. One can control this behavior using the speed mode (0xb3) command. The given integer is a b (bit0 is the least significant bit) with the following fields:

- bit0: Regard safe speed
- bit1: Regard maximum acceleration
- bit2: Regard maximum deceleration
- bit3: Regard right of way at intersections (only applies to approaching foe vehicles outside the intersection)
- bit4: Brake hard to avoid passing a red light
- bit5: **Disregard** right of way within intersections (only applies to foe vehicles that have entered the intersection).

Figure 4.4: Capture of the Sumo description of the *setSpeedMode* function [13]

In this capture, we can observe the checks that are usually considered by SUMO vehicles. The advantage given is the waiving of all of these checks by setting the register to [1 0 0 0 0 0] = 32.

4.4.3 Rewards

Various reward functions have been implemented and tested in throughout this project. As discussed in the Design section, the concern with respect to rewards is that of volatility. For accurate learning, reward should be specific and stable. As described in the Design section, several reward functions should be experimented with. These rewards, shown in appendix .4.3 are:

get_ave_speed_in_junction() This function has been applied using the sumo getDistance function to establish the vehicles in and around the junction. getDistance() returns

the distance vehicles have travelled since entering the simulation. The correct bracket (here [100-150]m) is therefore a good indication of its proximity to the junction. It there are no vehicles in or around the junction then the reward is set to -999 which is later filtered off.

waiting_time_reward() As specified in the system designs, this reward focuses on the minimisation of vehicles at standstill. Although this reward neglects the consideration of optimising the delay of moving vehicles, it has been observed through simulation that the breaking of left-turn vehicles to standstill when awaiting their foes is the greatest obstacle to junction efficiency. Theoretically, this reward would therefore address this obstacle.

get_num_slow_vehs() This function acts as intended in the design where the cut off limit for vehicles considered *slow* is under 11m/s. This reward therefore aims to optimise the passing of vehicles at a high speed. The corollary of this is that standstill vehicles are less emphasized than the previous reward function.

4.5 Summary

The framework was successfully implemented in 3 python classes and a command script through the use of tools and libraries such as sumo, netedit and traci. This implementation followed the intended design with detail. The most significant differences are:

- The use of lanes as *proxy agents* to the CAVs themselves
- The use of *SpeedMode* and an elevated acceleration maximum as the action to an elected CAV. This was defined in the design as any advantage in the claiming of junction priority.
- The dropping of the *densities* observation and the categorisation of vicinity and speed into a ranking and bracket categories respectively.
- The implementation of three reward functions: *Average junction speed*, *Total waiting time* and *Number of vehicles under a specific thresh-hold*.

Chapter 5

Evaluation

The overall goals of the evaluation chapter is to define if and to what extent, the implemented framework has solved the aims set out in the introduction 1.3. As a reminder, the primary goal of this project is to validate or disprove the hypothesis that the minimisation of data storage in Q-functions could be implemented for distributed AIM without significant concessions in the overall performance. To assess the success of this ambition, the chapter will consist of 4 sections:

- How to evaluate: Metrics and baselines
- What to evaluate: Experiments performed
- Results
- Analysis

5.1 How to evaluate: Metrics and baselines

The primary goal of this project, as described above, has two implications:

- The framework must successfully demonstrate learning
- The resulting Q-function must have perform rates comparable to the rival distributed RL application [20] and standard baselines in the AIM field.

5.1.1 Metrics

In order to assess the performance of either learning or Q-functions, we must decide what metrics these performance rates are based on. Fortunately, a set of metrics have

already been described as functions in our framework: the reward functions. These reward functions are:

- Average speed in and around the junction
- Total waiting time of vehicles
- Number of vehicles slower than x in the junction

5.1.2 Demonstrating Learning

The effects of learning are seen through the improvement of performance metrics over the training period. The overall slopes of performance curves are a good indicator of learning rate. For this indicator, the decay of ϵ is a significant consideration in the interpretation of the slope: it must also be apparent.

Another indicator of learning is the time taken to achieve a specific performance. While this indicator is more specific and does less to show the learning performance as a whole, it offers a quantitative result comparable to other implementations. It is also more adaptable to real life applications where we might train until reaching a specific performance level.

The baselines that will be used to compare the learning performance of our design will be: the rival distributed RL application [20] and the Q-Learning framework for Traffic Light Control. Although the latter cannot be compared in terms of final performance due to the physical constraints of traffic lights, the learning process can be examined.

5.1.3 Evaluating Q-functions

The performance of Q-functions essentially uses the same performance metrics as those of learning performance but considers only the final instances of training. These can be compared to any other scheduling policies for unsignalised AIM. For this application we will use as baselines: the rival distributed RL AIM application, FCFS and LQF. The two latter scheduling policies are standard scheduling policies used as baselines throughout the AIM field [21].

5.2 What to evaluate: Experiments performed

A significant characteristic of Reinforcement Learning is the notorious design challenge of selecting actions and rewards that learn well together. It is for this reason that our design comprises several possible actions and several rewards.

Experiment	Action	Reward	-s	-t	-ϵ	-d	α
1	Election	Junc. Speed	100,000	30,000	0.8	0.999	0.1
2	Election	$N_{slowvehs}$	100,000	30,000	0.8	0.999	0.1
3	Acceleration: 3 rates	Junc. Speed	100,000	30,000	0.8	0.999	0.1
4	Acceleration: 3 rates	$N_{slowvehs}$	100,000	30,000	0.8	0.999	0.1
5	Acceleration: 5 rates	Junc. Speed	100,000	30,000	0.8	0.999	0.1
6	Acceleration: 5 rates	$N_{slowvehs}$	100,000	30,000	0.8	0.999	0.1

Table 5.1: Table summarising the simulations performed

5.3 Results

5.3.1 Baselines

Learning performance

Although the performance metrics themselves are not comparable to those of our experiment given that the Traffic Control RL framework is designed for a 2-lane intersection rather than a 3-lane intersection, as discusses in section 5.1.2, the evolution of these metrics can be used to evaluate our own. This is shown in figure 5.1 below:

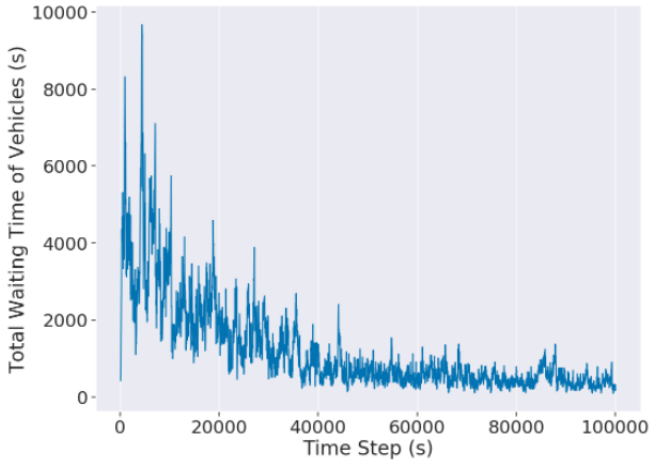


Figure 5.1: Plot of total waiting time of simulated vehicles over a training period of 100,000s

We can indeed see that the total waiting time has a clear decrease throughout the training time. As discussed in the Literature Review (section 2.10), this evolution can be expected to be exponential.

The convergence the rival paper can be seen in appendix .5. In this, we can observe a much more immediate convergence and stabilisation. The convergence rate of the model is indeed specific to the use case.

5.3.2 Experiment performances

Learning

The plotting of performance metrics over the training period of every experiment can be found in the appendix .7. An example of these results is shown in Figure 5.2 below:

Figure 5.2: Plot of performance metrics over a training period of 100,000s for Experiment 1

The results of experiment 1 are representative of the results for every experiment. We can observe that neither the average speed of vehicles in the junction nor the number of vehicles show any particular evolution over the training period. This is symptomatic of unsuccessful learning.

Q-function performance

Given that learning did not occur through the designed framework, the final Q-function essentially implements Longest-Queue-First with random variations. The final Q-table is in fact not a product product of Q-learning and therefore not a Q-function. Its performance is therefore superfluous to the already established performance rates of LQF.

Nevertheless, the final size of the Q-table is indicative of the potential size of a Q-function. For each experiment, these are shown in the table below:

Experiment	Left-turn	Straight	-s
1	5005	4498	100,000
1	7252	6521	200,000
1	10108	9133	400,000
2	4349	4926	100,000
3	5287	4464	100,000
4	5591	5352	100,000
5	5392	4221	100,000

Table 5.2: Table summarising the length of generated Q-tables for every experiment

In the table above we can observe that no significant relationship can be deduced between the Q-function model described and the exploration of the environment. On the other hand, the increasing size of the Q-table for increased training periods is an indication of vast unexplored territory in the MDP environment, as will be discussed below.

5.4 Analysis

5.4.1 Investigation

In order to understand the reasons that learning did not occur, we must look at samples of the generated Q_tables themselves. A sample of the Left-turn Q-table of experiment 1 for a 40,000s training period is shown in Figure 5.3 below:

```
(3.0, 0.0, 2.0, 1.0, 4.0, 3.0, 0.0, 2.0, 3.0, 0.0) : [0, 0, 0, 0, 2.5670638594393944]
(0.0, 0.0, 4.0, 0.0, 3.0, 0.0, 0.0, 1.0, 0.0, 2.0) : [9.359725776410508, 0, 0, 0, 1.2156840287543595]
(0.0, 1.0, 4.0, 1.0, 3.0, 0.0, 3.0, 0.0, 3.0, 2.0) : [0, 0, 0, 0, 0]
(3.0, 4.0, 0.0, 1.0, 2.0, 3.0, 0.0, 3.0, 2.0, 3.0) : [0.5838969566765994, 0, 0, 0, 0]
(3.0, 4.0, 2.0, 1.0, 0.0, 2.0, 3.0, 3.0, 2.0, 0.0) : [0.6735994655519968, 0, 0, 0, 1.9917527798906618]
(4.0, 1.0, 0.0, 3.0, 2.0, 1.0, 3.0, 0.0, 0.0, 3.0) : [0.5839235280800773, 0, 0, 0, 0]
(0.0, 2.0, 1.0, 3.0, 4.0, 0.0, 3.0, 3.0, 0.0, 1.0) : [0, 0, 0, 0, 0]
(3.0, 1.0, 0.0, 2.0, 4.0, 3.0, 2.0, 2.0, 2.0, 2.0) : [0, 0, 0, 0, 0]
(2.0, 4.0, 1.0, 0.0, 3.0, 2.0, 2.0, 3.0, 0.0, 2.0) : [0, 2.3935425567827275, 0, 0, 0]
(3.0, 0.0, 2.0, 1.0, 4.0, 2.0, 0.0, 2.0, 3.0, 1.0) : [6.195259567932499, 0, 0, 0, 0]
(1.0, 4.0, 0.0, 2.0, 3.0, 2.0, 2.0, 0.0, 1.0, 1.0) : [0, 1.0797941166957092, 0, 0, 0]
(2.0, 1.0, 0.0, 3.0, 4.0, 3.0, 3.0, 3.0, 2.0, 3.0) : [1.033593628980033, 0, 0, 0, 0]
(0.0, 2.0, 4.0, 1.0, 3.0, 0.0, 2.0, 2.0, 2.0, 2.0) : [0, 0, 0, 0, 3.8219565815953613]
(3.0, 2.0, 4.0, 0.0, 1.0, 2.0, 3.0, 1.0, 0.0, 2.0) : [1.6335902911709712, 0, 0, 0, 0]
```

Figure 5.3: Sample of the Straight Q-table of experiment 1: States are shown on the left and action Q-values on the right

From Figure 5.3 we can observe the states in the form [leader position rank, leader speed group] and Q-values in the form [Q_{CAV0}, Q_{CAV1}, Q_{CAV2}, Q_{CAV3}, Q_{CAV4}].

This figure reveals two particular characteristics of the Q-table:

- For many states, specific actions have not been tested. This is manifested
- The distribution of actions is biased toward specific action values: the same action is taken a disproportionate amount.

The former of these characteristics is symptomatic of an excessive number of both states and actions. This is further shown in table 5.2 of section 5.3.2. Here, we see that the number of states within the Q-table has not stabilised by the end of the training period.

Although the solution of extending the training period to 400,000s has been considered and implemented (Results shown in appendix), this solution still fails to yield clear signs of learning, as shown in appendix .7.2.

Indeed, the total number of possible states are :
$N_{pos-permutations} = n!/(n-r)! = N_{foes}!/(N_{foes} - N_{positions})$
$N_{speed-permutations} = N_{foes}^{N_{speeds}}$
Where $N_{foes} = 5, N_{positions} and N_{speeds} = 4$, we have:
$N_{states} = N_{pos-permutations} * N_{speed-permutations}$
$N_{states} = 120 * 625$ $N_{states} = 75000$

The maximum number of states are therefore 75000. In any of the specified training times, it is therefore inconceivable to train a Q-table with so many states: adjustments must be made.

5.4.2 Adjustment to framework states: One hot encoding

Given the excessive number of states in the current generation of Q-tables, another strategy was adopted: that of one-hot encoding of the observations. This consists of the identifying of the closest and the fastest leading vehicles by assigning them to "1" and assigning all other vehicles to "0". It's implementation is seen in Figure 5.4 below:

```
max_pos = max(veh_pos)
ind = veh_pos.index(max_pos)
veh_pos2 = []
for _ in range(len(self.foes)):
    veh_pos2.append(0)
veh_pos2[ind] = 1

#sort_veh = sorted(veh_pos)
#veh_pos = [sort_veh.index(veh) for veh in veh_pos]

return veh_pos2, get_speed
```

Figure 5.4: Code snippet of 1-hot-encoding variation $get_observations function$

The reduction of states in this way is therefore very significant, going from 75000 states to a mere 160, as shown in table 5.3 below:

Experiment	Left-turn	Straight	-s
7	160	160	100,000
7	160	160	200,000

Table 5.3: Table summarising the length of generated Q-tables for experiment 7

Although the number of states are restricted, this is achieved at a significant cost in the agent knowledge of the environment.

The results stemming from this adjustment and applied to the same conditions as Experiment 1 are shown in appendix .7.3. We can observe in this plotting that learning is still unsuccessful.

5.4.3 Proof of concept

Despite the unsuccessful learning process stemming from the thin line between an excessive number of states in the Q-table and an insufficient amount of data for agents to make an informed decision, we can still demonstrate the functionality of the framework in the generation of accurate Q-tables. To this end, data must be extracted from the table demonstrating positive reward for the correct action given a state. This is achieved by retrieving the maximum rewards in Q-tables and verifying that they correspond to the appropriate action for the given state. The execution of this task for Experiment 7 is shown in Figure 5.5 below:

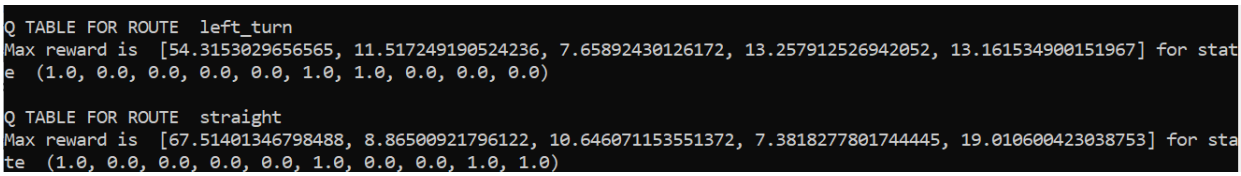


Figure 5.5: Capture of maximum rewards and associated states for experiment 7

We can see in Figure 5.5 that an extremely high reward is attributed to the appropriate action. In this case, the one-hot encoding indicates that lane 1 is both closest to the intersection and at a high speed for Q-table *left turn*. It therefore follows that a very high Q-value is allocated to the election of lane 1. The same case is considered in the *straight* Q-table for the same lane.

Chapter 6

Conclusions & Future Work

6.1 Conclusions

Although this project yielded results that were unable to concretely validate or disprove the hypothesis I set out to investigate, a deep analysis into the needs and characteristics of the application of distributed Q-learning to unsignalised AIM was achieved. Specifically, I saw the extent to which the minimisation of possible permutations of observations is fundamental to the learning process: a small increase in the observation space can lead to a significant increase in the number of states of a Q-table, thus slowing the learning process to an exponential degree. Moreover, over the course of this project I was able to design and implement a functional framework for the implementation of distributed Q-Learning to unsignalised AIM. This framework includes easily modifiable observation, state and reward functions in order to further explore the fine line between the minimisation of Q-table entries and insufficient agent knowledge over its environment.

6.2 Next steps

The most immediate step in the development of this project is the further exploration of observation/action combinations to yield well performing learning results. Concretely, this could be achieved by the variations of the one-hot encoding technique such as indicating the x CAVs closest to the junction.

Long term, this framework could lend itself well to other implementations of reinforcement learning such as Deep-Q networks.

Bibliography

[1] Lucas N. Alegre. *SUMO-RL*. https://github.com/LucasAlegre/sumo-rl. 2019.

[2] Lei Chen and Cristofer Englund. "Cooperative Intersection Management: A Survey". In: *IEEE Transactions on Intelligent Transportation Systems* 17.2 (2016), pp. 570–586. DOI: 10.1109/TITS.2015.2471812.

[3] *Classic Control environments*. 2022. URL: https://gym.openai.com/envs/#classic_control.

[4] Kurt Dresner and Peter Stone. "A Multiagent Approach to Autonomous Intersection Management". In: *Journal of Artificial Intelligence Research*. 33 (2008), pp. 591–657.

[5] Kurt Dresner and Peter Stone. "Learning policy selection for autonomous intersection management". In: (Jan. 2004), pp. 34–39.

[6] Yusen Huo, Qinghua Tao, and Jianming Hu. "Cooperative Control for Multi-Intersection Traffic Signal Based on Deep Reinforcement Learning and Imitation Learning". In: *IEEE Access* 8 (2020), pp. 199573–199585. DOI: 10.1109/ACCESS.2020.3034419.

[7] Ahmed Hussein et al. "Imitation Learning: A Survey of Learning Methods". In: *ACM Comput. Surv.* 50.2 (2017). ISSN: 0360-0300. DOI: 10.1145/3054912. URL: https://doi.org/10.1145/3054912.

[8] Mohammad Khayatian et al. "A Survey on Intersection Management of Connected Autonomous Vehicles". In: *ACM Trans. Cyber-Phys. Syst.* 4.4 (Aug. 2020). ISSN: 2378-962X. DOI: 10.1145/3407903. URL: https://doi-org.elib.tcd.ie/10.1145/3407903.

[9] William Koehrsen. "The Poisson Distribution and Poisson Process Explained". In: *Towards Data Science* (2019). URL: https://towardsdatascience.com/the-poisson-distribution-and-poisson-process-explained-4e2cb17d459.

[10] Stanley Lam and Jayantha Katupitiya. "Cooperative Intersection Negotiation for Multiple Autonomous Platoons". In: vol. 8. June 2013, pp. 48–53. ISBN: 9783902823366. DOI: 10.3182/20130626-3-AU-2035.00073.

[11] Bai Li et al. “Near-Optimal Online Motion Planning of Connected and Automated Vehicles at a Signal-Free and Lane-Free Intersection”. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018, pp. 1432–1437. DOI: 10.1109/IVS.2018.8500528.

[12] Bo Liu and Zhengtao Ding. “A distributed deep reinforcement learning method for traffic light control”. In: *Neurocomputing* (2021). ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.11.106. URL: https://www.sciencedirect.com/science/article/pii/S092523122101818X.

[13] Pablo Alvarez Lopez et al. “Microscopic Traffic Simulation using SUMO”. In: *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL: https://elib.dlr.de/124092/.

[14] Wanjing Ma, Jinjue Li, and Chunhui Yu. “Shared-phase-dedicated-lane based intersection control with mixed traffic of human-driven vehicles and connected and automated vehicles”. In: *Transportation Research Part C: Emerging Technologies* 135 (2022), p. 103509. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2021.103509. URL: https://www.sciencedirect.com/science/article/pii/S0968090X21004940.

[15] Laleh Makarem and Denis Gillet. “Fluent coordination of autonomous vehicles at intersections”. In: *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2012, pp. 2557–2562. DOI: 10.1109/ICSMC.2012.6378130.

[16] Bile Peng et al. “Connected autonomous vehicles for improving mixed traffic efficiency in unsignalized intersections with deep reinforcement learning”. In: *Communications in Transportation Research* 1 (2021), p. 100017. ISSN: 2772-4247. DOI: https://doi.org/10.1016/j.commtr.2021.100017. URL: https://www.sciencedirect.com/science/article/pii/S2772424721000172.

[17] sumo. *Random number generation (RNG)*. 2022. URL: https://sumo.dlr.de/docs/Simulation/Randomness.html.

[18] sumo. *TripInfo: Waiting$_t$ime*. 2022. URL: https://sumo.dlr.de/docs/Simulation/Output/TripInfo.html.

[19] Wei Wu et al. “Autonomous intersection management with pedestrians crossing”. In: *Transportation Research Part C: Emerging Technologies* 135 (2022), p. 103521. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2021.103521. URL: https://www.sciencedirect.com/science/article/pii/S0968090X21005039.

[20] Yuanyuan Wu, Haipeng Chen, and Feng Zhu. "DCL-AIM: Decentralized coordination learning of autonomous intersection management for connected and automated vehicles". In: *Transportation Research Part C: Emerging Technologies* 103 (2019), pp. 246–260. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2019.04.012. URL: https://www.sciencedirect.com/science/article/pii/S0968090X18316279.

[21] M. Nejad Z. Zhong and E. E. Lee. "Autonomous and Semi autonomous Intersection Management: A Survey". In: *IEEE Intelligent Transportation Systems Magazine* 13.2 (2021), pp. 53–70.

Appendix

.1 Network configuration

.1.1 Network configuration

```
<nodes>
    <node id="n" x="0.0" y="75.0" type="priority"/>
    <node id="s" x="0.0" y="-75.0" type="priority"/>
    <node id="e" x="75.0" y="0.0" type="priority"/>
    <node id="w" x="-75.0" y="0.0" type="priority"/>
    <node id="t" x="0.0" y="0.0" type="unregulated"/>
</nodes>
```

Figure 1: Snippet of the definition of nodes in mynodes.edg.xml

```
<edges>
    <edge from="n" id="n_t" to="t" numLanes="3"/>
    <edge from="t" id="t_n" to="n" numLanes="3"/>

    <edge from="w" id="w_t" to="t" numLanes="3"/>
    <edge from="t" id="t_w" to="w" numLanes="3"/>

    <edge from="t" id="t_s" to="s" numLanes="3"/>
    <edge from="s" id="s_t" to="t" numLanes="3"/>

    <edge from="t" id="t_e" to="e" numLanes="3"/>
    <edge from="e" id="e_t" to="t" numLanes="3"/>
</edges>
```

Figure 2: Snippet of the definition of edges in myedges.edg.xml

.1.2 Vehicle generation

```
<route id="route_ns" edges="n_t t_s" maxSpeed="10"/>
<route id="route_nw" edges="n_t t_w" maxSpeed="10"/>
<route id="route_ne" edges="n_t t_e" maxSpeed="10"/>
<route id="route_we" edges="w_t t_e" maxSpeed="10"/>
<route id="route_wn" edges="w_t t_n" maxSpeed="10"/>
<route id="route_ws" edges="w_t t_s" maxSpeed="10"/>
<route id="route_ew" edges="e_t t_w" maxSpeed="10"/>
<route id="route_en" edges="e_t t_n" maxSpeed="10"/>
<route id="route_es" edges="e_t t_s" maxSpeed="10"/>
<route id="route_sn" edges="s_t t_n" maxSpeed="10"/>
<route id="route_se" edges="s_t t_e" maxSpeed="10"/>
<route id="route_sw" edges="s_t t_w" maxSpeed="10"/>
```

Figure 3: Snippet of the definition of routes in mycross.rou.xml

```
<!-- <flow id="flow_ns" route="route_ns" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" depart
<flow id="flow_nw" route="route_nw" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_ne" route="route_ne" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_sw" route="route_sw" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_sn" route="route_sn" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_se" route="route_se" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_en" route="route_en" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_ew" route="route_ew" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_es" route="route_es" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_wn" route="route_wn" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_we" route="route_we" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
<flow id="flow_ws" route="route_ws" begin="0" end="100000" probability="0.07" departSpeed="max" departPos="base" departLane=
```

Figure 4: Snippet of binomial distribution vehicle generation in mycross.rou.xml (commented out)

```
<flow id="flow_ns" route="route_ns" begin="0" end="100000" vehsPerHour="250" departSpeed="max" departPos="base"
<flow id="flow_nw" route="route_nw" begin="0" end="100000" vehsPerHour="255" departSpeed="max" departPos="base"
<flow id="flow_ne" route="route_ne" begin="0" end="100000" vehsPerHour="260" departSpeed="max" departPos="base"
<flow id="flow_sw" route="route_sw" begin="0" end="100000" vehsPerHour="245" departSpeed="max" departPos="base"
<flow id="flow_sn" route="route_sn" begin="0" end="100000" vehsPerHour="240" departSpeed="max" departPos="base"
<flow id="flow_se" route="route_se" begin="0" end="100000" vehsPerHour="235" departSpeed="max" departPos="base"
<flow id="flow_en" route="route_en" begin="0" end="100000" vehsPerHour="230" departSpeed="max" departPos="base"
<flow id="flow_ew" route="route_ew" begin="0" end="100000" vehsPerHour="225" departSpeed="max" departPos="base"
<flow id="flow_es" route="route_es" begin="0" end="100000" vehsPerHour="265" departSpeed="max" departPos="base"
<flow id="flow_wn" route="route_wn" begin="0" end="100000" vehsPerHour="270" departSpeed="max" departPos="base"
<flow id="flow_we" route="route_we" begin="0" end="100000" vehsPerHour="275" departSpeed="max" departPos="base"
<flow id="flow_ws" route="route_ws" begin="0" end="100000" vehsPerHour="250" departSpeed="max" departPos="base"
```

Figure 5: Snippet of constant vehicle generation in mycross.rou.xml

.2 Main.py

```
while env.sim_step < env.sim_max_time:
    #Randomly select route and lane as the next agent.
    route = random.choice(list(ql_agents.keys()))
    ln = random.choice(list(ql_agents[route].keys()))
    #Check that there is a CAV in the lane
    if list(env.sumo.lane.getLastStepVehicleIDs(ln))!= None:
        curr_state = env._compute_observations(ln)
        if env.sim_step < args.transition:
            actions = env.lqf(ln)
        else :
            actions = ql_agents[route][ln].act(q_table[route], curr_state, env.epsilon)
        env.epsilon = max(env.epsilon*args.decay, args.min_epsilon)
        new_state, reward = env.step(action=actions, lane = ln)
        q_table[route] = ql_agents[route][ln].learn(old_state = tuple(curr_state),
        next_state=tuple(new_state), reward=reward, q_table=q_table[route])
```

Figure 6: Snippet of the designed algorithm coded in main.py

.3 QLAgent class

```
def act(self, q_table, state):
    state = tuple(state)
    if state not in q_table:
        q_table[state] = [0 for _ in range(self.action_space.n)]
    if np.random.rand() < self.epsilon:
        action = int(self.action_space.sample())
    else:
        action = np.argmax(q_table[state])

    self.action = action
    self.epsilon = max(self.epsilon*self.decay, self.min_epsilon)
    return self.action
```

Figure 7: Snippet of Act() function performing epsilon-greedy in the QLAgent class

```
def learn(self, old_state, next_state, reward, q_table):
    if next_state not in q_table:
        q_table[next_state] = [0 for _ in range(self.action_space.n)]
    s = tuple(old_state)
    s1 = next_state
    a = self.action
    q_table[s][a] = q_table[s][a] + self.alpha*(reward + self.gamma*max(q_table[s1]) - q
    self.acc_reward += reward
    return q_table
```

Figure 8: Snippet of learn() function updating the Q-table in the QLAgent class

.4 MDP

.4.1 getObservations()

```
def compute_observation(self):
    density = self.get_lanes_density()
    veh_pos, get_speed = self._get_veh_list()
    get_speedr = [round(num, 1) for num in get_speed]
    densityr = [round(num, 2) for num in density]
    observation = np.array(densityr +veh_pos +get_speedr, dtype=np.float16)
    return observation
```

Figure 9: Snippet of compute_observations() function in the Lanes class

```
def _get_veh_list(self):
    veh_pos = []
    get_speed = []
    for lane in self.foes:
        veh=list(self.sumo.lane.getLastStepVehicleIDs(lane))
        if veh:
            car = veh[-1]
            veh_pos += [self.sumo.vehicle.getLanePosition(car)]
            get_speed += [self.sumo.vehicle.getSpeed(car)]
        else:
            veh_pos += [-1]
            get_speed += [-1]

    sort_veh = sorted(veh_pos)
    veh_pos = [sort_veh.index(veh) for veh in veh_pos]

    return veh_pos, get_speed
```

Figure 10: Snippet of the function for fetching leading vehicle position and speed in the Lanes class

1 hot encoding of observations

```
def _get_veh_list(self):
    veh_pos = []
    get_speed = []
    for lane in self.foes:
        veh=list(self.sumo.lane.getLastStepVehicleIDs(lane))
        if veh:
            car = veh[-1]
            veh_pos += [self.sumo.vehicle.getLanePosition(car)]
            get_speed += [self.sumo.vehicle.getSpeed(car)]
        else:
            veh_pos += [-1]
            get_speed += [-1]

    sort_veh = sorted(veh_pos)
    veh_pos = [sort_veh.index(veh) for veh in veh_pos]

    return veh_pos, get_speed
```

Figure 11: Snippet of the function for fetching leading vehicle position and speed in the Lanes class

.4.2 Actions

```
def apply_accelerate_elected(self, elected):
    veh=list(self.sumo.lane.getLastStepVehicleIDs(self.foes[elected]))
    if veh:
        car = veh[-1]
        sped = self.sumo.vehicle.getSpeed(car)
        print(sped)
        self.sumo.vehicle.setMaxSpeed(car, 25)
        self.sumo.vehicle.setSpeedMode(car, 32)
        return car
    else: return None
```

Figure 12: Snippet of the foe election and accelerate action function in the Lanes class

```
def apply_accelerate(self, acceleration):
    veh=list(self.sumo.lane.getLastStepVehicleIDs(self.ln_id))
    if veh:
        car = veh[-1]
        if acceleration == 0:

            sped = self.sumo.vehicle.getSpeed(car)
            if sped>4:
                sped2 = sped*0.6
                sped = self.sumo.vehicle.setSpeed(car, sped2)
        elif acceleration == 1: ···
        elif acceleration == 3: ···
        elif acceleration == 4:
            sped = self.sumo.vehicle.getSpeed(car)
            if sped>3:
                sped2 = sped*1.4
                sped = self.sumo.vehicle.setSpeed(car, sped2)
            else:
                sped2 = 12
                sped = self.sumo.vehicle.setSpeed(car, sped2)
        return car
```

Figure 13: Snippet of the acceleration action function in the Lanes class

.4.3 Rewards

```
def get_ave_speed_in_junction(self):
    acc_intersec_speed = 0
    intersec_veh = 0
    for veh in self.sumo.vehicle.getIDList():
        dist = self.sumo.vehicle.getDistance(veh)
        if  dist > 100 and  dist < 150:
            acc_intersec_speed += self.sumo.vehicle.getSpeed(veh)
            intersec_veh += 1
    if intersec_veh > 0:
        ave_intersec_speed = acc_intersec_speed/intersec_veh
        return ave_intersec_speed
    else: return -999
```

Figure 14: Snippet of the Speed_within_junction reward function in the Lanes class

```
def _waiting_time_reward(self):
    ts_wait = sum(self.get_waiting_time_per_lane()) / 100.0
    reward = self.last_measure - ts_wait
    self.last_measure = ts_wait
    return reward

def get_waiting_time_per_lane(self):
    wait_time_per_lane = []
    for lane in self.lanes:
        veh_list = self.sumo.lane.getLastStepVehicleIDs(lane)
        wait_time = 0.0
        for veh in veh_list:
            veh_lane = self.sumo.vehicle.getLaneID(veh)
            acc = self.sumo.vehicle.getAccumulatedWaitingTime(veh)
            if veh not in self.env.vehicles:
                self.env.vehicles[veh] = {veh_lane: acc}
            else:
                self.env.vehicles[veh][veh_lane] = acc - sum([self.env.vehicles[veh][la
            wait_time += self.env.vehicles[veh][veh_lane]
        wait_time_per_lane.append(wait_time)
    return wait_time_per_lane
```

Figure 15: Snippet of the two functions defining the Waiting_time reward in the Lanes class

```
def _waiting_time_reward(self):
    ts_wait = sum(self.get_waiting_time_per_lane()) / 100.0
    reward = self.last_measure - ts_wait
    self.last_measure = ts_wait
    return reward

def get_waiting_time_per_lane(self):
    wait_time_per_lane = []
    for lane in self.lanes:
        veh_list = self.sumo.lane.getLastStepVehicleIDs(lane)
        wait_time = 0.0
        for veh in veh_list:
            veh_lane = self.sumo.vehicle.getLaneID(veh)
            acc = self.sumo.vehicle.getAccumulatedWaitingTime(veh)
            if veh not in self.env.vehicles:
                self.env.vehicles[veh] = {veh_lane: acc}
            else:
                self.env.vehicles[veh][veh_lane] = acc - sum([self.env.vehicles[veh][la
            wait_time += self.env.vehicles[veh][veh_lane]
        wait_time_per_lane.append(wait_time)
    return wait_time_per_lane
```

Figure 16: Snippet of the get_num_slow_vehs reward function in the Lanes class

.5 Baseline

.6 Results

.6.1 Training period: 100,000s

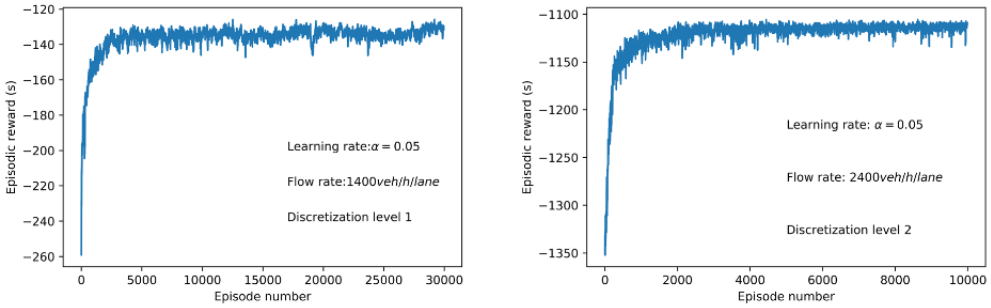


Fig. 7. An example of convergence of DCL-AIM algorithm in two discretization level experiments.

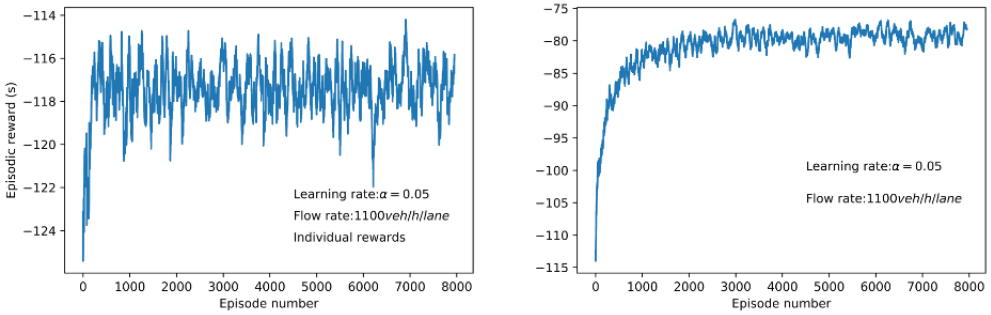


Figure 17: Convergence of individual rewards (left), coordination learning (right) for the DCL-AIM rival algorithm [20]

.7 Results

.7.1 Training period: 100,000s

Figure 18: Plot of performance metrics over a training period of 100,000s for Experiment 1

.7.2 Training period: 400,000s

Figure 19: Plot of performance metrics over a training period of 400,000s for Experiment 1

.7.3 1-hot-encoding for observations

Figure 20: Plot of performance metrics over a training period of 100,000s for Experiment 7