UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Cooperative Collision Avoidance for Autonomous Vehicles

Cale van der Westhuizen
20705786

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at Stellenbosch University.

Supervisor: Dr JAA Engelbrecht

November 2020

Acknowledgements

I would like to express my very great appreciation to the following people for supporting me during the writing of this report:

- A special gratitude to my final year project supervisor, Dr JAA Engelbrecht, for contributing stimulating suggestions that kept my project on the optimal path both literally and figuratively.
- To the Electrical Engineering Department of Stellenbosch, for constructing an impeccable course, that helped develop my critical thinking skills and for providing a collaborative environment that made learning enjoyable.
- My parents, for affording me the privilege of an excellent education, and supporting me throughout my studies.
- To my grandma, for always encouraging me to give my very best in everything that I do, with her undying wish to see me graduate cum laude.

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.

3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Student number: 20705786	*Signature:* 
Initials and surname: CA VAN DER WESTHUIZEN	*Date:* 01/11/2020

Abstract

Cooperative Collision Avoidance is a multi-agent path-finding problem where each agent should find a collision-free course to their respective destination, having full knowledge of other agents' intentions and whereabouts. This report presents a deterministic and a machine learning approach to solving this problem. Both algorithms are decoupled methods, that decompose the problem into a sequence of single-agent path-finding problems. Hierarchical A* makes use of a heuristic to search through space-time, in the continuous state-space, to find a route to the destination without collisions. Reinforcement Learning allows each agent to perform different actions in an environment, and through trial and error, learns which actions will result in a non-colliding route. The results show that A* is efficient and consistent at finding optimal paths, due to the adaptability of the algorithm. Reinforcement Learning was also capable of finding successful paths, but required extensive computing time and substantial complexity to achieve similar results to that of A*.

Opsomming

Samewerkende botsingsvermyding is 'n multi-agent padbeplanning probleem waar elke agent 'n botsingsvrye roete na hul onderskeie bestemming moet vind, met kennis van die posisies en beplande roetes van die ander agente. Hierdie verslag beskryf beide 'n deterministiese en 'n masjienleer oplossing vir hierdie probleem. Beide algoritmes is ontkoppelde metodes wat die probleem in 'n reeks van enkel-agent-soekprobleme ontbind. Hiërargies A* maak gebruik van 'n heuristiek om deur ruimte-tyd te soek, in die kontinue toestandsruimte, om 'n roete na die bestemming te vind sonder botsings. Versterkingsleer stel elke agent in staat om verskillende aksies in 'n omgewing uit te voer, en deur middel van probeer en tref te leer watter aksies na 'n botsingvrye roete sal lei. Die resultate toon dat A* doeltreffend en konsekwent is om optimale paaie te vind, as gevolg van die aanpasbaarheid van die algoritme. Versterkingsleer is ook in staat om suksesvolle paaie te vind, maar behels aansienlik meer kompleksiteit en benodig meer verwerkingstyd om soortgelyke resultate as dié van A* te behaal.

Contents

List of Figures

List of Tables

Nomenclature

Variables and functions

$\mathbf{V}$	Velocity vector
$\mathbf{A}$	Position vector
$\mathbf{U}$	Unit vector in direction of current trajectory
k	Discrete time index
ΔT	Length of time for a discrete time-step
n	Node
$P(X)$	Probability of event X occurring.
ϵ	Probability of choosing to explore (Epsilon-Greedy)
Π	Policy for a Reinforcement Learning Agent
t	Time index
S	Current State
S'	Next State
a	Action in current state
a'	Action in next state
γ	Discounting Coefficient for Future rewards
$G(t)$	Return - The sum of all future rewards
$V(S)$	Value Function - Measure of how good it is to be in state S
$R(t)$	Reward received at time t
E	Expectation
N	Total number of iterations - Amount of training episodes
Q^*	Optimal Q-value
ct	Count - Number of times a certain state-action pair has been accessed
α	Learning rate - Weighting of learned Q-values to old Q-values

Acronyms and abbreviations

RL	Reinforcement Learning
DNN	Deep Neural Network
RH	Receding Horizon
Co-DDPG	Cooperative Deep Deterministic Policy Gradient

Chapter 1

Introduction

1.1. Background

Autonomous driving has shown rapid advancements in recent years, with an increased effort from numerous industrial companies to incorporate this technology into their vehicles [1]. Major developments in artificial intelligence and sensor technologies have allowed such a rise to take place. This technology has the ability to improve people's lives, by reducing human-error related accidents, easing traffic congestion and enhancing the mobility of people [2]. By the release of this report, Tesla had just released full self-driving capability, allowing complete autonomy of all vehicles, limiting the need for any human intervention [3]. The production of Tesla's autonomous fleet alone, is projected to be around 20 million vehicles within the next 5 years [4]. Currently, autonomous vehicles are entirely independent, and make use of computer vision or LiDAR scanners to perceive their environment and make decisions on how to act in the real world [5]. However, with an increasing number of intelligent self-driving vehicles on the road, a vehicle's ability to perceive the world can be further improved by allowing it to cooperatively plan with other autonomous vehicles. Each vehicle can broadcast their current trajectory and intent, allowing others to incorporate this information into their plan and adapting it to either avoid collisions or to find more optimal routes to their destination.

Path-finding is a crucial component to achieving the above-mentioned capabilities. As such, extensive research has been conducted into the various methods of approaching this problem. There is a natural tendency to almost immediately lean towards machine learning as a desired approach due to the numerous applications and impressive performance it has proven to produce. Specifically, developments in Reinforcement Learning (RL) have demonstrated a human-like methodology to problem-solving, as it has the capacity to plan for the future and to learn from past interactions with its environment. These intelligent agents have the ability to predict stock prices, solve Atari games, or even beat a world champion at the game of chess, just to name a few, with no starting knowledge of the situation and simply learning through trial and error.

This being said, it is important to realize that machine learning is not always the most appropriate solution to all problems. There are certain cases, where the use of machine learning is completely unnecessary and can lead to numerous difficulties in implementation. The stochastic nature of machine learning can lead to undesirable outcomes, as the system has no contextual awareness of the situation. In such cases, deterministic algorithms may be more desirable as it is easier to embed physical constraints into decision-making [6]. When applying machine learning techniques to deterministic systems, the algorithm will succeed, but the algorithm will not know when it is violating any physical laws which may produce strange behavior [6].

This paper introduces both a deterministic and a machine learning approach to dealing with the task of cooperative collision avoidance. It presents a fair comparison between two different approaches to the same problem and attempts to investigate the validity of the above claims.

1.2. Problem Statement

The central objective is to allow multiple vehicles to predict imminent collisions and then cooperatively re-plan their paths to avoid the collisions, while minimising both their control effort and their deviation from their original paths, and while obeying their own vehicle motion constraints. A secondary objective of this report is to both qualitatively and quantitatively evaluate the performance of deterministic and machine learning algorithms, to investigate the perception that machine learning exhibits superior performance to conventional problem-solving algorithms in almost all use cases. Two fundamentally different approaches will thus be developed to critically examine the benefits and drawbacks of these techniques.

1.3. Objectives

- A deterministic, and machine learning algorithm, must be developed to achieve multi-agent path planning and ultimately avoid collisions.
- Both algorithms must be implemented in simulation to function within the same environment under different traffic scenarios.
- Construct various real-world scenarios that will highlight the main features of each algorithm as well as enabling a fair comparison between them.
- Both algorithms should clearly display the ability to successfully avoid all collisions.

- The different methods should attempt to minimize each vehicle's control effort and deviation from its original path as far as possible.
- Algorithms should be verified using Monte Carlo simulations and should investigate the effects of an increased number of vehicles on the performance.
- The performance of both algorithms should be critically discussed, after which the various advantages and disadvantages of each will be identified.
- Conclusions should be drawn on the relevance of each method, and provide clarity as to which situations either would be applicable or preferred.

1.4. Summary of work

Research was conducted to first discover the current strategies for collision avoidance. Of the various strategies, it was decided to further investigate Reinforcement Learning and A*, due to the promising results that had been presented. The A* algorithm and Reinforcement Learning algorithm were then successfully developed and implemented in simulation. A thorough comparison between the two approaches was performed after testing under identical scenarios. Both approaches demonstrated fundamentally different results which highlighted the capabilities of each.

The A* algorithm was consistent, fast, and always gave the optimal route. It was much easier to implement and could be constructed in the continuous domain, which allowed for speed-up and slow-down capabilities.

It was found that in highly constrained problems, where there is certain expected behaviour, RL does not perform as well. However, it was argued that RL has the ability to perform equally to the deterministic algorithm, but requires an enormous extra effort to achieve this through complicated reward management. RL required significantly more time to find a solution, but this was due to the features and circumstances incorporated during training.

It was concluded that both approaches are feasible solutions to cooperative collision avoidance. Due to the deterministic nature of the problem, the conclusion was drawn that the A* algorithm is preferred, however the RL solution demonstrated impressive results that exposed its potential, and encouraged the need for further development.

1.5. Scope

This report focuses exclusively on cooperative path-finding where all vehicles have full knowledge of each other's intended plans. It is therefore assumed that all vehicles broadcast their current position, velocity and intent to one another. Collision avoidance will be performed using horizontal manoeuvres only and should be scalable to multiple autonomous vehicles. The vehicle model neglects any attempt to incorporate any control system behaviour or disturbance rejection along with any other uncertainties involved. The report focuses solely on evaluating the performance of the algorithms under ideal conditions, and as such any added complexity in this regard is deemed unnecessary. Vehicles are assumed independent and focuses on a decoupled approach to path finding, refraining from centralized methods. Furthermore, each vehicle will be modelled as a point mass with a corresponding velocity whilst allowing for acceleration and motion that is strictly linear. The section on Reinforcement Learning focuses primarily on standard Q-learning, and does not extend into the realm of Deep Reinforcement Learning.

1.6. Report Layout

- Chapter 1 introduces contextual information towards the formulation of the problem. The objectives for the report are presented, followed by a short summary of the execution of these objectives. The chapter closes by defining the scope and providing details on the limitations of the work presented in this report.
- Chapter 2 provides a literature review of the current approaches used to tackle cooperative collision avoidance. This section also provides an outline of the necessary information that was acquired to solve the problem, along with additional contextual information that the reader may require.
- Chapter 3 further describes the experimental setup of the problem. The various performance metrics that will be used to evaluate both methods will be introduced.
- Chapters 4 and 5 present the detailed design, implementation, verification and evaluation of the A* and Reinforcement Learning algorithms respectively.
- Chapter 6 provides a comparison between the different approaches. The objectives and results will be further discussed after which a comparison between the two approaches will be made. The report will conclude by highlighting the relevance of each method as well as providing recommendations for future work.

Chapter 2

Literature Review

Cooperative collision avoidance is a broad topic, that has been explored using numerous different methods. Various methods such as optimization-based [7] , rule-based [8] , machine learning and more recently deep learning [9] methods have been proposed. A concise breakdown of a few of these methods will be provided in the following section.

2.1. Centralized vs Decentralized

Amongst the various methods there is one main feature that distinguishes approaches from one another. Regardless of the algorithm, approaches can be broken down into either centralized or decentralized [10]. Centralized methods take into account all vehicles at once, and seek a combined optimal solution. In essence, the conjunction of all vehicles forms a single agent that can perform combinations of different actions to achieve a desired outcome. This however requires the need for a central hub to process the decisions which has shown to increase computation time [11]. In contrast, decentralized methods decompose the collision avoidance problem into independent problems for each vehicle [12]. Depending on the current states of other vehicles, each vehicle would then greedily search for a route to its destination. The decentralized approach is much better suited to real-time applications [13]. Both cases were subject to further investigation.

2.2. Decentralized methods

Typical examples of decentralized methods include a rule-based approach, suggested by H. Izadi [8], which computes an analytical set of boundaries with respect to all vehicles such that when a vehicle approaches this boundary, predefined collision avoidance manoeuvres are performed. Although this technique greatly reduces computational complexity, trajectories are likely sub-optimal when collision avoidance action is taken. P. Chotiprayanakul proposes creating a three-dimensional force field for collision avoidance in complex dynamic environments [14]. When vehicles approach the force field of another, a repulsive force is generated to divert it away from the collision. A. Yang employs a unique strategy of translating the collision avoidance problem into a formation-based

problem [7]. Oncoming traffic is grouped into clusters, whereby intra-group members would then vary their formation to avoid collisions with another cluster.

Amongst the various decoupled approaches, this report is more focused specifically towards the application of A* to cooperative collision avoidance. Therefore, some of the existing methods that incorporate this algorithm are further discussed:

2.2.1. A* Algorithms

Standard A* is a method of single-agent path finding that can find a route from the start to an end node with the shortest possible distance, and while avoiding all obstacles in the path. It achieves this by creating a search tree of possible routes to the goal which terminates when the goal is reached. Examples of how this method has been extended to multi-agent path-finding is demonstrated below:

Hierarchical A* Algorithm

Hierarchical A* is a systematic approach designed to deal with multiple decoupled agents [15]. This strategy makes use of a space-time grid and a reservation table to keep track of all vehicles' planned paths. Once a path is selected, the corresponding cells are reserved so that other vehicles know to avoid these cells while planning. As shown in Figure 2.1, while an agent moves through each point in space towards its goal, the corresponding blocks in time have been shaded in to represent a reservation. These reservations can be interpreted as static obstacles, and as such standard A* can now be applied to the system. D. Silver has stated that a naive application of this algorithm can be very inefficient as it is entirely dependent on the order of the chosen hierarchy [15]. Any solution found may thus be locally optimal for a chosen hierarchy, but not a globally optimal solution for collision avoidance.

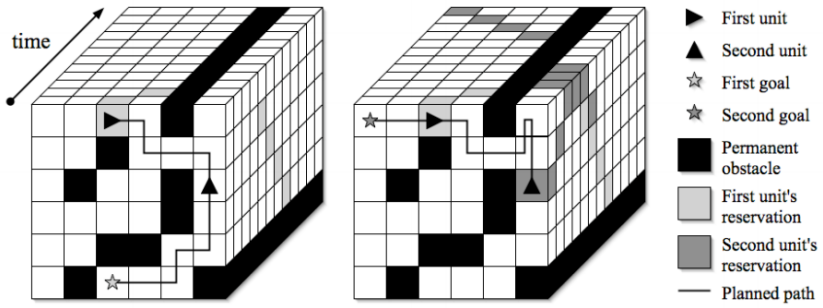


Figure 2.1: Figure showing an example of two agents navigating to their destinations whilst avoiding collisions using the space-time reservation table. [15] The image on the left demonstrates the first agent finding a path to the goal, without regard for the second agent, and recording its path in the space-time reservation table. The right image shows the second agent planning while taking into account the path taken by the first agent.

Unfortunately, as the block size is fixed, this approach only allows for discrete steps to be taken in the four cardinal directions, and also cannot be extended to comply with vehicles that can adapt their speed.

Windowed Hierarchical Cooperative A* Algorithm

A windowed approach attempts to rectify the previous issue regarding the sensitivity to agent ordering [12]. The A* search path is broken down into much smaller, fixed length windows. This allows the algorithm to dynamically vary the agent order, enabling all vehicles to have a higher hierarchy for a short period of time. Each vehicle will then begin searching for a partial path to its destination. The search window will begin to shift at regular intervals. Only agents within the search window will be considered for possible collisions and otherwise ignored. Therefore, this approach will find the lowest cost sequence rather efficiently, but the solution will only result in the optimal partial route to the destination. Using a limited search horizon allows for real-time applications, as the processing time can now be distributed across all agents. Results demonstrated that consistently successful routes were found, with shorter paths and decreased computation times [12].

2.3. Centralized Methods

B. Alrifaee demonstrated Mixed integer linear planning as a possible centralized approach, which gave solutions very close to the optimal solution [11]. It was however concluded that the performance of the system is significantly affected by computation time. W. Li attempted to improve on computing time with a Receding Horizon (RH) control scheme which proposed dynamically determining trajectories by executing planning over a shorter action horizon, but yielded similar results that would not allow for real-time applications [13]. Existing reinforcement learning techniques will be elaborated on, as this is a central topic in this report.

2.3.1. Reinforcement Learning

Reinforcement learning is a machine learning technique that when applied to cooperative collision avoidance, allows an agent to learn from trial and error which actions resulted in collisions, then enables those decisions to be adapted so that collisions will not appear in the future. Applications of standard RL are generally limited to low-dimensional domains [16]. However, the use of approximation functions, like neural-nets, combined with reinforcement learning, has allowed for exceptional performance within high-dimensional, complex situations. As such, it is well suited to centralized approaches that require a

high-dimensional space. One of the cutting-edge examples of this application is shown below:

Deep Reinforcement Learning

Reinforcement learning involves calculating a policy which maps all possible states to desired actions, which are stored in a finite database. This state-action mapping, which is generally characterised by a lookup table, is now replaced with a deep neural network as shown in Figure 2.2. The use of approximations allows many more possible state-action pairs, making the implementation of RL viable. Y. Yuan proposes using Co-DDPG, which allows previous techniques to be adapted for environments with a continuous action-space [17]. While the system is learning, the algorithm enforces varying levels of strict penalties when a collision occurs or to any situations that may result in a potential collision, which allows for an extremely robust learning environment. Y. Yuan observed an exponential decrease in the number of recorded collisions, which outperforms many of the existing techniques [17]. This technique demonstrated great scalability to multiple vehicles and was even applicable to real-time applications.

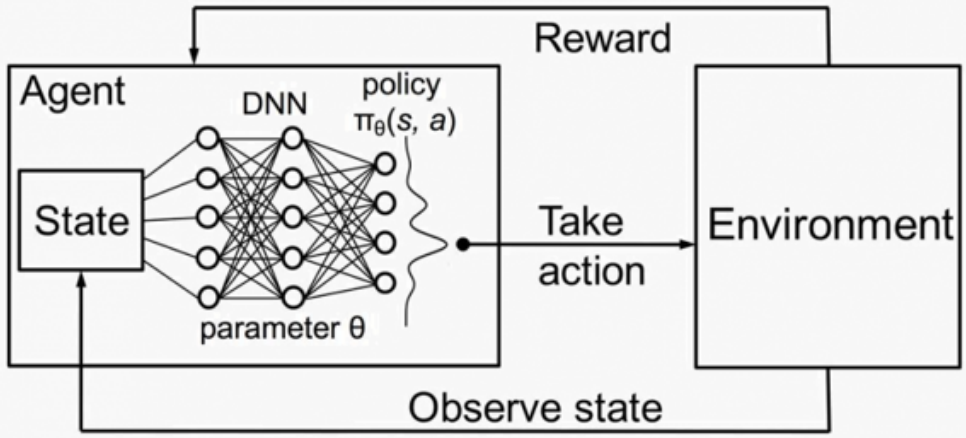


Figure 2.2: Figure showing the architecture of deep reinforcement learning. [18]

Chapter 3

System Modelling and Conceptualization

This section serves as a preliminary walk-through of the experimental setup. It will elaborate on the vehicle model, as well as the environmental setup, which enables further evaluation of the two algorithms. Furthermore, various metrics that standardize the performance of each algorithm will be presented.

3.1. Vehicle Model

A relatively simple vehicle model is constructed that will be used during simulation to verify the developed algorithms. This decision was made to defer all complexity towards the algorithms, rather than allowing for any uncertainty in the model to obscure the observed results. Sufficient detail is still included in the model to ensure that the experiment demonstrates the required outcomes.

As required by the problem statement, the actions of each vehicle should be constrained to horizontal manoeuvres whilst adhering to a minimum forward speed. Vehicles are assumed to have a control system that allows for perfect tracking of reference trajectories, as well as having access to precise location coordinates. The full state vector, $\mathbf{X}$, for a vehicle is shown in Equation 3.1, where A and V represent the respective position and velocity of the vehicle.

$$\mathbf{X} = [A_x, A_y, V_x, V_y] \tag{3.1}$$

The trajectory of a vehicle is represented by a unit vector, $\mathbf{U}$, in the direction of travel. The velocity vector, $\mathbf{V}$, is constructed by scaling the trajectory by the speed of each vehicle. The relationship is shown in Equation 3.2 below.

$$\mathbf{U} = \frac{\mathbf{V}}{||\mathbf{V}||} \quad \text{and} \quad ||\mathbf{V}|| = \sqrt{V_x^2 + V_y^2} \tag{3.2}$$

The movement of each vehicle is characterized by the equations of motion shown in 3.3. The motion is described by discrete time-steps of fixed length, ΔT. The next position is

determined from the current position added to the product of the velocity and the length of each time-step.

$$\begin{aligned} A_x(k+1) &= A_x(k) + V_x(k)\Delta T \\ A_y(k+1) &= A_y(k) + V_y(k)\Delta T \end{aligned} \tag{3.3}$$

The model will allow for speeding up and slowing down within limits. Here is a list of allowable actions for each vehicle: veer left, veer right, continue straight, speed-up and slow-down. To veer in each direction implies a change in trajectory of 45 degrees to each side. Continuing straight and speed changing actions makes no course corrections, but will scale the velocity vectors of each vehicle accordingly. Each vehicle is currently modelled as a point mass. However, to determine whether vehicles have collided, the vehicle model also needs to incorporate an exclusion zone. An overlap in the exclusion zones of any vehicles allows for the detection of a collision. An illustration of the complete vehicle model is depicted in Figure 3.1.

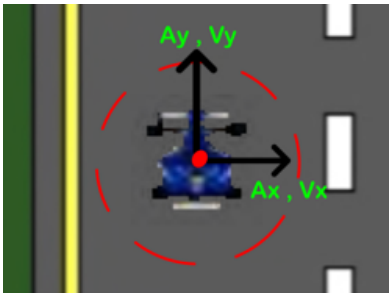


Figure 3.1: Figure showing the exclusion zone of a vehicle

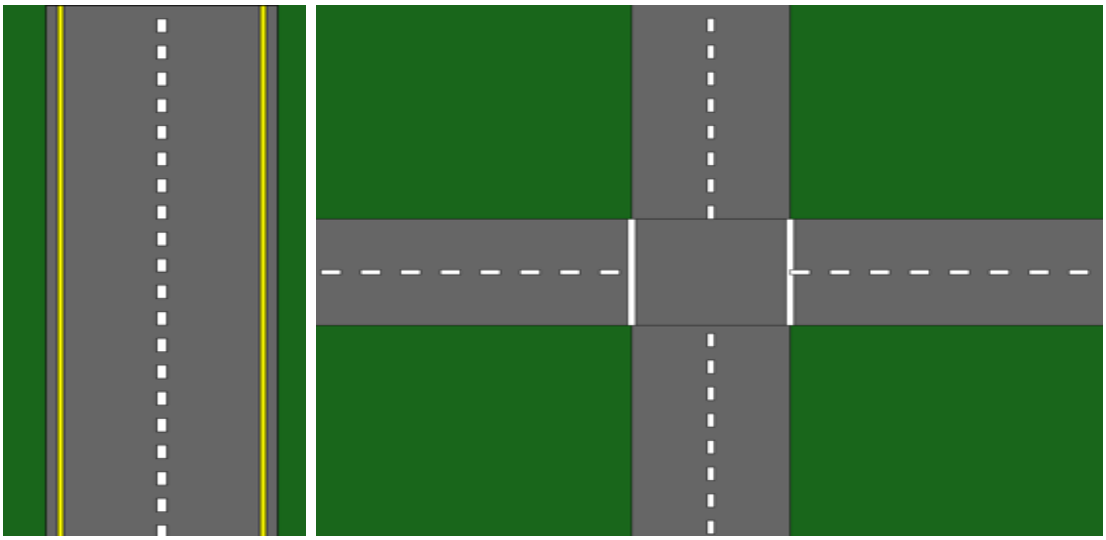
3.2. System Conceptualization

3.2.1. Environment

The environment was constructed based on the typical use-cases in which collisions may occur in practice. Namely, a two-way, single lane, main road and a cross-road with a priority highway. All scenarios will make use of either of the environments in Figures 3.2a and 3.2b during experimentation. No vehicles are permitted to leave the main road and should refrain from movement off of the marked terrain.

3.2.2. Performance Metrics

Performance criteria need to be set up to allow for a fair comparison between both approaches. The goal of any given vehicle is to make its way from the starting to the end location whilst avoiding collisions. The problem statement requires that all vehicles



(a) Two-way Highway **(b)** Cross-road Scenario

minimize their control effort, as well as their deviation from their original paths. The relevant metrics required to achieve this will be further discussed. Moreover, performance criteria that measure the computing time and consistency of the algorithms is also appended, as they are deemed necessary for a comprehensive comparison. The specific purpose of these metrics is to further gauge the quality of decisions made by an algorithm. This report places a particular focus on gaining insight into the manner in which deterministic vs machine learning algorithms achieve solutions, opposed to merely measuring the success rate of each.

Control effort

A relevant metric that would allow the control effort to be measured, would be recording a tally of the combined number of actions taken by each vehicle for the entire path duration. An action in this sense, is defined to be any change in the current trajectory as well as a change in speed to the vehicle.

Ideal Path Deviation

To measure the deviation from the current path, the total number of time-steps that all vehicles spend off of the ideal path will be recorded. To further clarify how this will be achieved, it is first necessary to define what is meant by the ideal path.

The ideal path is calculated as a discrete number of way-points, directly between the start and the end goal. It forms a straight line, and represents the trajectory that a vehicle would take, if there were no other vehicles in the space, without the need to avoid collisions. Figure 3.3 demonstrates the ideal path for a single vehicle. As shown, the length of a time-step in this case is equivalent to the length of the white marking lines in

the environment, which can be used as a general marker to track the number of time-steps a vehicle spends off the ideal path. Going forward, the end location for any given vehicle will be depicted by a circular dot with the same colour as the corresponding vehicle as shown below.

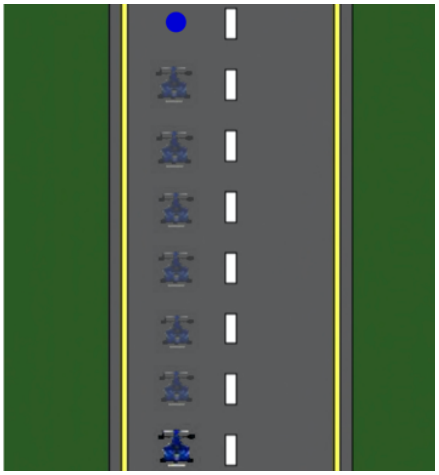


Figure 3.3: Figure showing the ideal path of a vehicle

Computing Time

Whether or not a certain algorithm is capable of real-time performance is of particular concern in the topic of collision avoidance. If path-finding is required, there should be sufficient overhead to ensure collisions do not occur during planning. Furthermore, particular algorithms demonstrate a significant increase in computation time when scaling to multiple vehicles. A performance measure of the scalability of algorithms is therefore also necessary. The computing time will thus be a measurement taken from the time at which the path-finding algorithm is initiated, up until the time that suitable routes for all vehicles have been calculated.

Monte Carlo Simulations

All the above metrics will be used to analyse the performance within a particular scenario. However, it is also imperative that an algorithm can consistently perform well in a number of different scenarios. This calls for Monte Carlo simulations that will perform numerous iterations over scenarios where the start and end locations will be randomized. Additional checks are performed to ensure vehicles are not initialized within the same exclusion zone which would yield an invalid test. Each simulation will therefore calculate a distribution for each of the above-mentioned metrics. Moreover, a distribution will be calculated for an increasing number of vehicles incorporated in the scenario, as another measure of the scalability of the algorithm.

Chapter 4

A* Algorithm

The A* algorithm is a deterministic method that will be implemented to solve the task of cooperative collision avoidance. The A* algorithm is well suited for solving very complex problems in an optimal way. It is also well suited for constrained problems where vehicle behavior is more controllable and predictable as the algorithm has a sense of direction [12]. In contrast to RL, there is no need for any stochastic behavior, allowing the algorithm to radically reduce the time needed to find the shortest path to the goal [16]. A* is different from conventional deterministic algorithms as it has the ability to "think", with the capability of finding very efficient routes to the goal without exploring any unnecessary options [15]. The algorithm is used extensively in many real-time games due to its reliability and efficiency.

The standard version of the A* algorithm is fairly simple to implement for a single agent trying to reach its goal. Yet, the adaptation of the algorithm to perform the task of cooperative collision avoidance is rather involved, due to the nature of time complexity. The algorithm will be tested in multiple real-world scenarios in which the viability of the algorithm can be verified. The performance in each scenario will then be evaluated according to the metrics discussed in the system conceptualization section. Various advantages and disadvantages will be discussed, after which general conclusions will be made towards the feasibility of this algorithm to performing the task of cooperative collision avoidance.

4.1. Algorithm Design

4.1.1. Single Agent Path-finding

The standard implementation of A* is commonly constructed in grid form, with each discrete block corresponding to a node [15]. This makes the identification and labeling process of different nodes easier to implement as there are a fixed number of discrete nodes that can be accessed. However, the same concept of A* is just as applicable to the continuous space, where a node is represented by a coordinate in 2D space using index (x,y) as a key. To distinguish between different nodes, an inclusion radius will be used

that can be adjusted to get the desired resolution. The continuous representation is more powerful as it gives the agent a wider range of movement allowing for the detection of more optimal paths. The goal is to use A* to find the best route of nodes to traverse from the starting node to the target node in as few steps as possible.

The manner in which the algorithm decides which nodes should be considered as having the best chance of reaching the goal is determined by the so-called F-value. The value represents the total cost of the node, n, and is calculated by Equation 4.1.

$$F(n) = G(n) + H(n) \tag{4.1}$$

The G-value is the total cost of moving to the current node from the starting node. This value is calculated as the total length of path travelled. The H-value is a heuristic value which estimates the remaining distance left to travel from the current node to the target node. There are various methods of estimating this value and it is imperative that the calculation be an underestimation of the remaining path for the algorithm to function correctly [15]. It is only an estimation as the actual path may be longer if it is necessary for the agent to take a longer route to avoid collisions. The Euclidean distance between the current and target node will be used to calculate the H-value. By choosing nodes with the lowest F-value at each time-step, the cost function is in effect minimized. The means by which a series of decisions is conducted by the A* algorithm is best illustrated through the Example in Figure 4.1. This graphic assumes only 3 allowable actions namely: straight, veer-Right and veer-Left with the respective nodes for each possible action created.

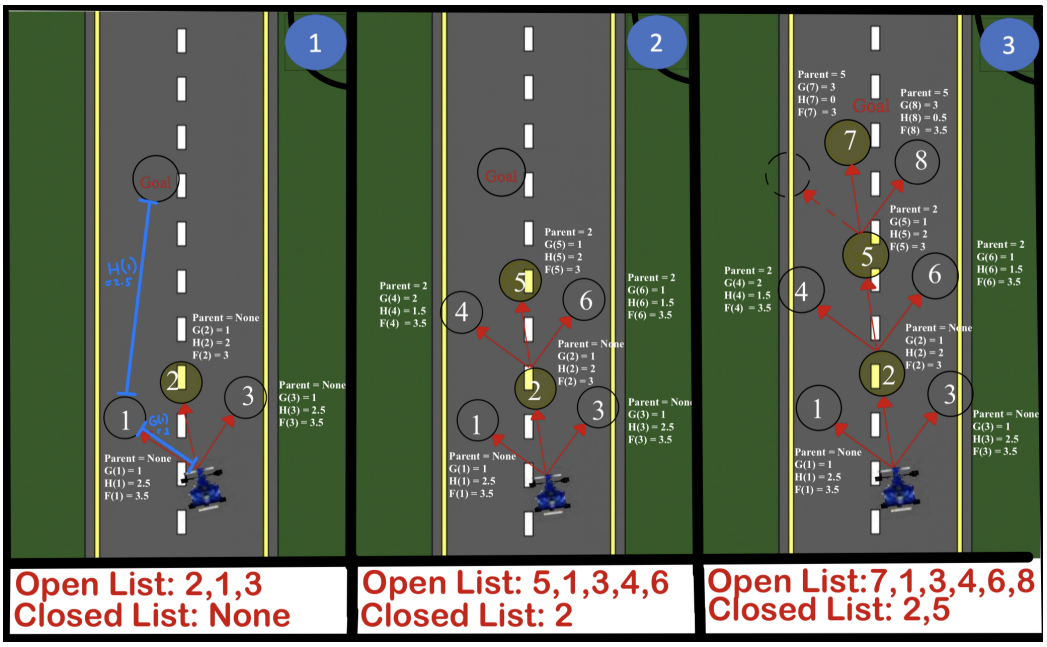


Figure 4.1: Figure showing the standard A* algorithm performing path planning for a single agent over 3 time-steps. Nodes that are to be added to the closed-list are highlighted in yellow. The inclusion radius for a node is demonstrated by the black circles. Note that the size of nodes has been over-exaggerated for illustration purposes.

The example shows that only nodes with the lowest F-value are further expanded. After a node is expanded, it is then added to the closed-list, which serves as a record to ensure that nodes are not evaluated more than once. The open-list is where all of the currently available nodes are stored, which is organized by rank of their F-value and is used to determine which node will be expanded in the next iteration. For the time being, the fact that certain nodes may be impassable due to other objects has been neglected. However, it can be seen in time-step 3 that a node has not been added as the inclusion radius falls outside the allowable range of motion.

The actual implementation of A*, that will be discussed in the report, consists of 5 allowable actions with the added ability to slow down or speed up at any point. Speed changing is accomplished by altering the length between nodes in the current direction of travel for succeeding iterations. Hence, the setup for an individual expanded node is more accurately represented in Figure 4.2.

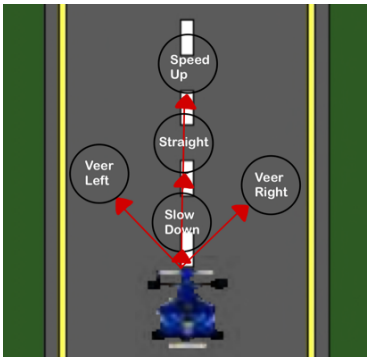


Figure 4.2: Figure showing the available actions for A*

Consecutive actions are therefore difficult to illustrate, but the concept is identical to the previous illustration in Figure 4.1. However, at this point the question may arise whether the algorithm would not then always prioritize actions to speed up the vehicle as this would allow the goal to be reached in the least amount of time and the fewest steps possible. This is in fact the case, and further adaptation of the algorithm is necessary to prevent such a situation. With reference to the metrics for performance evaluation, it is desirable to have the vehicle stay on the ideal path for as long as is possible. The illustration below demonstrates the new method that will be used to ensure that the vehicle prioritizes the ideal path whilst also preventing excessive speeding.

In Figure 4.3, an ideal path is first generated as a set of stepping stones directly between the start and target node. Together they form a set of way-points that represent where the vehicle would like to be at every time-step if there were no obstacles in the way and is depicted in the figure by the opaque vehicles. The ideal path way-points are generated according to the vehicle's initial velocity at the start of a scenario, which determines the distance between each of the points. Therefore, with no obstacles in the path, the vehicle should be able to follow these points exactly. However, when an obstacle is encountered,

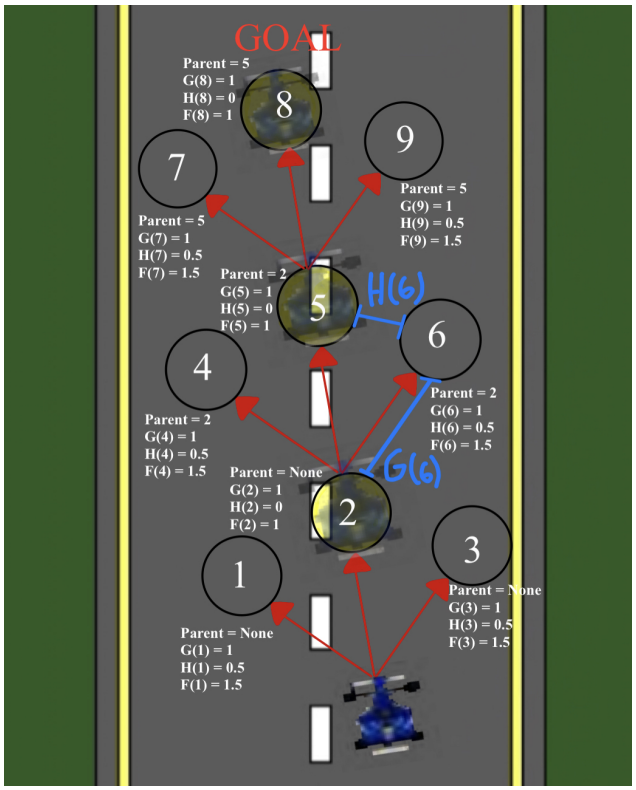


Figure 4.3: Figure showing the updated A* algorithm. The ideal path way-points are represented by the lightly shaded vehicles at each time-step.

the vehicle is required to either slow down or divert its path. After navigating past the obstacle, the vehicle would then choose to speed up after falling behind the ideal way-point for that time-step. Figure 4.3 also demonstrates how the cost function has been adapted to achieve this result. The G-value is instead measured as the total distance from where the vehicle should have been at the previous time-step according to the ideal path. Likewise, the H-value is now a heuristic that estimates the remaining distance to travel to reach the next way-point on the ideal path. The total cost remains the same as calculated in Equation 4.1. The node with the lowest F-value now represents the point that has the best chance of taking the vehicle back to the ideal position, opposed to the previous strategy of merely moving closer to the final position. This also perfectly suits the need for the vehicle to stay on the ideal path whilst also solving the issue of excessive speeding. The full cost of a node is thus calculated by Equation 4.2.

$$\begin{aligned} F(n) &= G(n) + H(n) \\ F(n) &= \sqrt{(X_{pos} - idealX_{pos}[t-1])^2 + (Y_{pos} - idealY_{pos}[t-1])^2} \\ &\quad + \sqrt{(X_{pos} - idealX_{pos}[t])^2 + (Y_{pos} - idealY_{pos}[t])^2} \end{aligned} \tag{4.2}$$

As a disclaimer, one could argue that this implementation of the algorithm is technically not A*. This is due to the fact that A* is defined explicitly by using the total path cost, with the cost-to-come and the cost-to-go both equally weighted. As the priority queue is no longer ordered by the total cost with only a local measurement of the path utilized, the history of deviation back to the start node is in effect neglected. Nevertheless, this "Sliding A*" method has proven to be extremely effective at avoiding collisions as well as sticking to the ideal path, which will be further demonstrated in the results section. A reasonable explanation for this is that the order of expansion of nodes follows a similar pattern to the regular approach.

4.1.2. Cooperative Collision Avoidance

As previously mentioned, confronting the task of cooperative collision avoidance raises further complications due to the added dimension of time. Each vehicle requires complete knowledge of the planned paths for each of the other vehicles, so that it can determine whether an action at a certain time-step is allowable. It is also desirable to allow vehicles to be at the same point in space at different points in time, which cannot be accomplished by simply replicating the standard A* algorithm for each vehicle. It is therefore necessary to extend the A* algorithm to 3 dimensions by constructing a new space-time map [15]. The position of each node will now be accessed by the index (t, x, y) in 3-dimensional space. To distinguish between different nodes, the same inclusion radius will be used. However, it should be noted that only nodes at the same time-step can possibly be considered equal. For example, moving one step forward in the x-direction will result in the movement from Cell (t, x, y) to Cell $(t + 1, x + 1, y)$. This procedure will be called space-time A*, with the goal to now reach the destination node at any time with as few steps as possible. Note, there is no need to adjust the cost function to include duration as this is inherent with the current goal.

A simplified version of the entire collision avoidance process is demonstrated by Figure 4.4 below. To accomplish path-finding, a hierarchical decoupled approach will be used. In other words, starting with the vehicle of highest priority, each vehicle will individually perform the space-time A* algorithm. After finding the optimal path to reach the destination, the spatial position of the vehicle at each point in time is entered into a reservation table. This reservation table will be used by succeeding vehicles to determine whether or not the creation of a node would result in a collision.

With reference to Figure 4.4, the blue vehicle is assumed to have the highest priority, and as such it computes the optimal path with the space-time implementation of A*

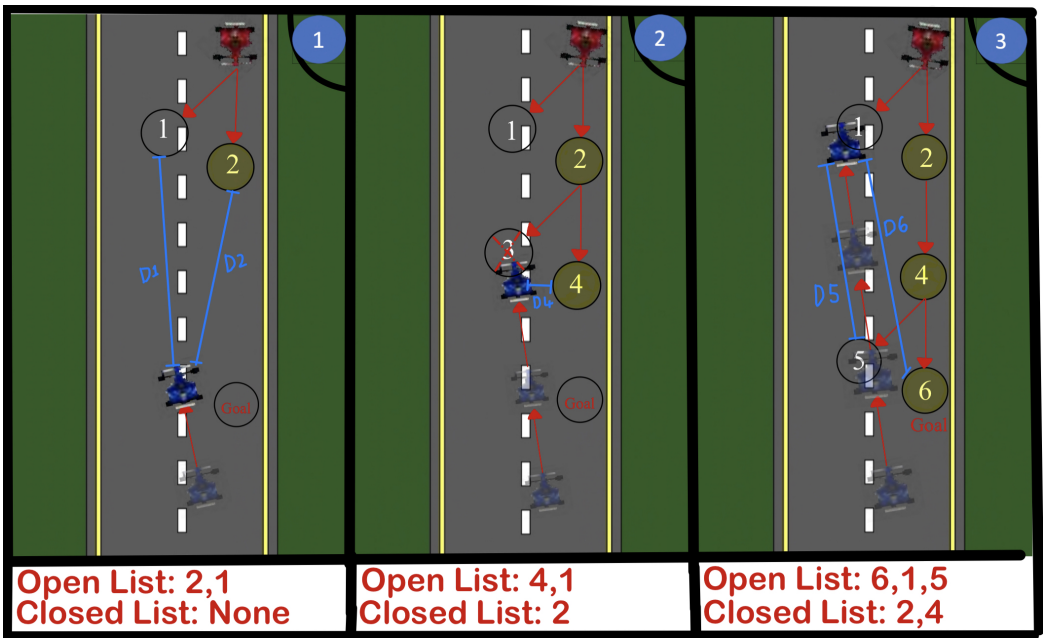


Figure 4.4: Figure showing the space-time implantation of A*. The blue vehicle in bold represents the position in the reservation table for the priority vehicle at a specific time-step. Distances D1-D6 represent the relevant collision distances.

with no regard for any other vehicles. The calculated positions are then entered into the reservation table, with the bold vehicle at each time-step representing the relevant position in the table. The red vehicle will assume the vehicle of lower priority and, as such, should plan a route in accordance to the path taken by the blue vehicle. Depending on the time index of the current node under consideration, distances to all priority vehicles at the same time index are first computed. If this distance is less than a certain threshold, then the current node will not be created. This is clearly demonstrated in time-step 2, in Figure 4.4, where node 3 is not created as it would result in a collision. Furthermore, one should note that the construction of node 5 in time-step 3 is completely valid, as although it lies on the path taken by the priority vehicle, it occurs at a different point in time. In this manner, the optimal path for the red vehicle is calculated. As such, in a scenario with multiple vehicles, the optimal paths for all succeeding vehicles would follow the exact same approach.

4.2. Algorithm Implementation

Pseudo-code for the decoupled space-time version of the A* algorithm, to accomplish the task of cooperative collision avoidance, is shown in Algorithm 1 from Figure 4.5. The full process involves a hierarchical method where the desired hierarchy of the cars is specified by the user depending on the scenario at hand. The specific details of the algorithm will be thoroughly discussed before proceeding to the algorithm demonstration in various real-world scenarios.

Starting with the vehicle of highest priority, the algorithm will create start and end nodes at given positions using the node class. The ideal path for this vehicle will then be constructed as a set of way-points between the start and end nodes. Thus, the matrix idealPath contains the ideal (x,y) coordinates at each point in time with the distance between points determined from the given initial velocity. The start node will then be added to the open-list and thereafter the main loop will commence.

The main loop will continue indefinitely until either the destination is reached, or no possible route can be found, corresponding to an empty open-list. For each iteration of this loop, the current node will be determined based on the value in open-list with the smallest F-value. This current node will then be added to the closed-list, after which it will be further expanded by generating child nodes. However, it first identifies whether the position of the current node under evaluation is equivalent to that of the end node. If this is the case, the destination has been reached, and the path can simply be found by backtracking from this node through each of the parent nodes until the start node is reached. Otherwise, child node positions will be generated from the current node position depending on the different actions that the vehicle can perform.

For each action, a node at the relevant position is then created. This new node has to undergo a series of tests to first ensure that this is a valid possible position in the action space. First it is determined whether the action would result in the vehicle moving off of the allowable terrain. If the vehicle under consideration is of a lower hierarchy, it is necessary to consult the reservation table to determine whether other vehicles might be at this point in space-time and as such would result in a collision. Finally, a check is made to verify whether the node under question has previously been evaluated or created. In any of these cases, this node would be discarded, and the remaining child nodes will be consulted.

If the node is valid, the F-value is determined from Equation 4.2 that was previously derived. Thereafter, the node will be added to the open-list, forming a valid possible action. By repeating this process and choosing nodes with the smallest F-values at each iteration, the optimal path can be found as these nodes are identified as having a large chance of reaching the goal.

By continually iterating, the end-node will eventually be reached where the above-mentioned backtracking process will be used to calculate the path. The path for this vehicle will then be appended to the reservation table. Hence, all succeeding vehicles will take this planned path into account when deciding whether or not a certain node is valid.

Algorithm 1: A* PathPlanning Algorithm

```
//Create a class to represent a Node;
[Class Node] F , G , H , parent, position(t,x,y);
;
Initialise reservationTable;
for car = 1...numCars do
    Initialise startNode , endNode;
    Initialise path;
    Calculate idealPath(startNode,endNode);
    Initialise openList, closedList;
    Add startNode to openList;
    while openList is not Empty do
        //Get the currentNode;
        Arrange openList by F-value; //Smallest to biggest;
        currentNode = openList[0]; //Smallest F-value;
        Remove currentNode from openList;
        Add currentNode to closedList;
        //Found the goal;
        if currentNode == endNode then
            //Backtrack to get the path;
            while currentNode != startNode do
                Add currentNode to path;
                currentNode = currentNode–parent;
            end
            Return path;
        end
        //Generate childNodes (Veer Left, Straight, VeerRight, Slow Down , Speed Up);
        Create childNodes from currentNode;
        for each child in childNodes do
            if child position is off-road then
                Skip to next child;
            end
            //Check for a collision;
            if child position is in reservationTable then
                Skip to next child;
            end
            //Check if node has been created or visited;
            if child is already in openList/closedList then
                Skip to next child;
            end
            child–G = euclidean distance between child–position and idealPath[t-1];
            child–H = euclidean distance between child–position and idealPath[t];
            child–F = child–G + child–H;
            Add child to openList;
        end
    end
    Update reservationTable(path);
end
```

2

Figure 4.5

4.3. Algorithm Verification and Results

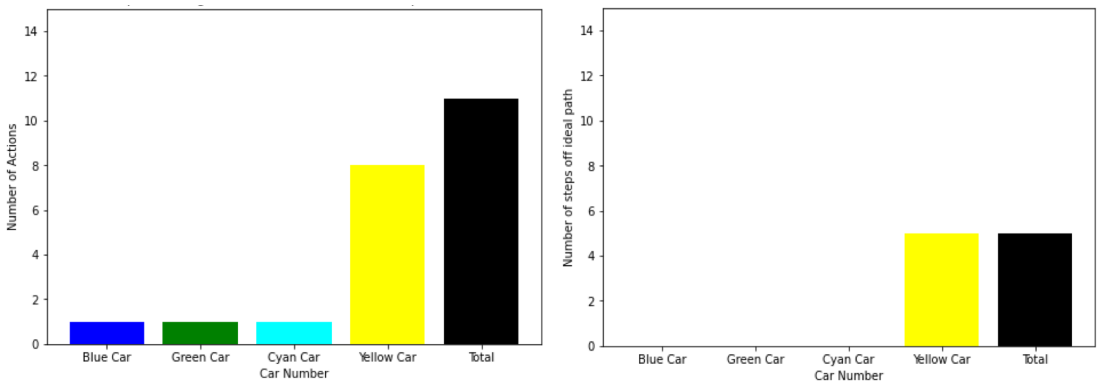
The implementation of the cooperative collision avoidance algorithm will be tested in four real-world scenarios. The first two scenarios are used to demonstrate the full capability of the A* algorithm. All vehicles have a complete range of actions including the ability to speed up and slow down as previously discussed and are tested in scenarios where such actions are critical to finding the optimal path. The performance will be evaluated according to the metrics discussed in the system conceptualization section. In the next chapter, it is found that the implementation of the reinforcement learning algorithm is only possible through the use of discrete blocks. Thus, the RL algorithm does not include

speed-up and slow-down capabilities. To ensure a fair comparison, the next two scenarios have been set up to limit the A* algorithm by removing these actions. Each scenario is split up into four time-steps with the trails of each vehicle up until the current time-step included for illustration. The trails for a vehicle in the first two scenarios, with speed changing capability, is represented by a dotted line where the time between each point is kept constant. Thus, closely spaced dots imply that the vehicle is busy slowing down and dots with large spaces between represent speeding up. A solid line for the trail implies that the vehicle has kept a constant velocity throughout. Adjacent to each scenario is the complimentary space-time map showing the optimal paths for each of the vehicles. With the time dimension added on the z-axis it is now clearly visible how the reservation table of nodes is constructed, and it demonstrates the creation of nodes at the same position in space at different points in time.

Two-way overtake Scenario

The figure below demonstrates a single highway overtaking scenario. The hierarchy of the vehicles in this scenario from highest to lowest priority is: blue, green, cyan, yellow. The first three vehicles are all initialized with the same constant speed. However, the yellow vehicle, of the lowest priority, is initialized with a significantly higher speed. This implies that the ideal path, created by the yellow vehicle, requires it to reach its destination before the blue vehicle. The scenario demonstrates that the algorithm is able to judge that the yellow vehicle has sufficient speed as well as enough space between the green and cyan vehicles to enable a successful overtaking manoeuvre. Time-step 2 in Figure 4.7 shows that initially the vehicle has to slow down, visible by the closely spaced dots, and wait for the cyan vehicle to pass before executing the overtake manoeuvre. The vehicle then speeds up accordingly to catch up to the ideal path way-point as shown if time-step 3. The space-time map verifies that the yellow vehicle is able to reach its destination prior to the blue vehicle as well as return to the ideal path as soon as possible.

After the entire procedure, it is noted that only 11 total actions are required to avoid



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 4.6: Overtaking scenario: Action cost and path deviation cost per vehicle.

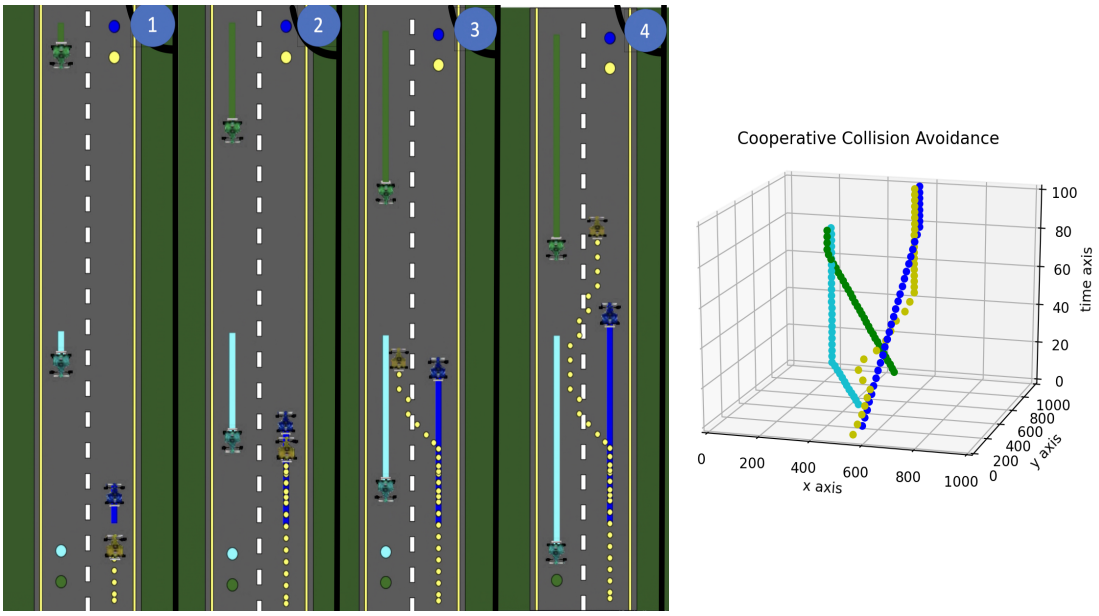


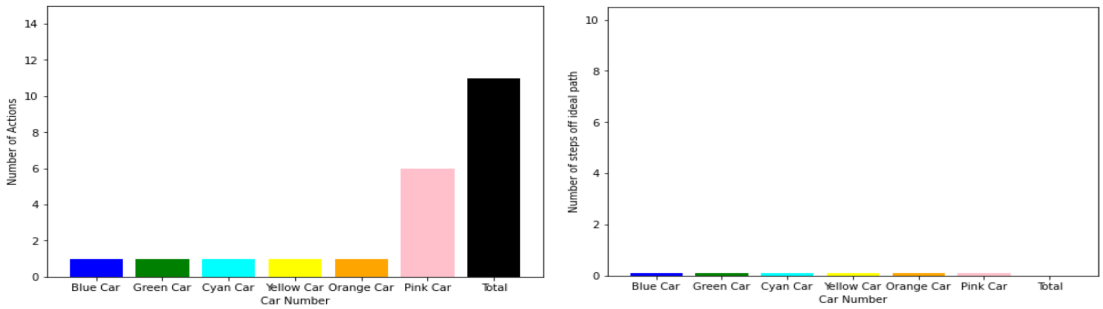
Figure 4.7: Figure showing an overtaking scenario involving 4 vehicles. Note the number of actions recorded includes actions taken to slow down and speed-up.

the collision with the majority performed by a single vehicle. The algorithm mimics the decision process that a human would take in a similar scenario of which the importance herein will become apparent in the next chapter. Only 5 steps are spent off of the ideal path which is an impressive result. The time taken to compute the planned paths for all vehicles is 8.1 seconds. Additional tests were also performed on the same scenario which showed some interesting results. By either decreasing the gap between the green and cyan vehicles, or slightly decreasing the initial speed of the yellow vehicle, the algorithm would then determine that an overtaking manoeuvre is not possible and the yellow vehicle would remain slowed down behind the blue vehicle until both oncoming vehicles had passed.

Crossroad-Yield Scenario

The second scenario is that of a crossroad with a priority highway and is illustrated in Figure 4.9 below. This requires that other vehicles on the secondary road yield to the oncoming traffic. The hierarchy of the vehicles in this scenario from highest to lowest priority are: blue, green, cyan, yellow, orange, pink. Apart from the pink vehicle, all vehicles are able to reach their destinations without needing to change their trajectory. Initially, the pink vehicle is required to slow down at the intersection and wait for the first two oncoming vehicles to pass as shown in time-step two. Time-step 3 demonstrates the vehicle then speeding up again, which is visible by the increased space between the dotted trail. At this point, it then has to slow down once again until the traffic in the other lane has also passed, before proceeding further as shown in time-step 4.

The space-time map for this scenario illustrates very good performance where all vehicles keep to the ideal path throughout. Collisions are avoided by primarily using



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 4.8: Crossroad-Yield scenario: Action cost and path deviation cost per vehicle

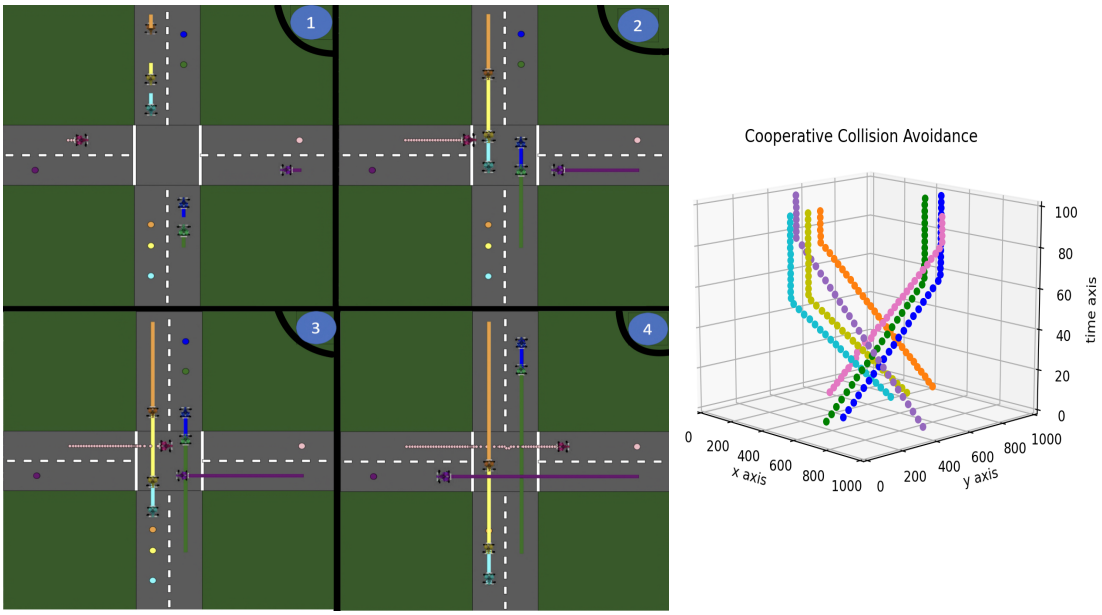


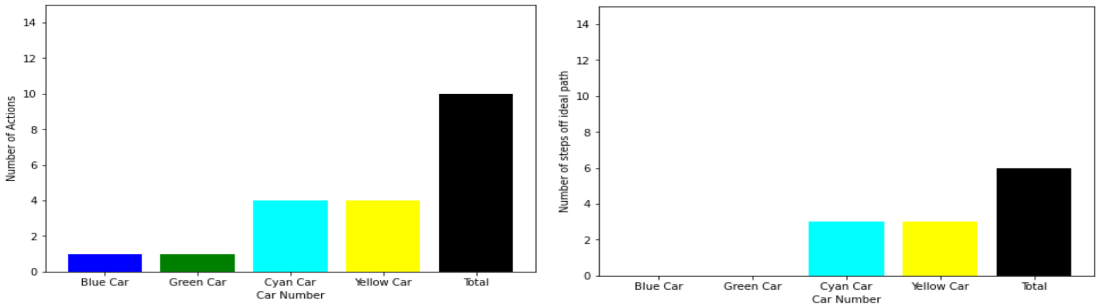
Figure 4.9: Figure showing a Crossroad-Yield Scenario involving 7 vehicles. Note the number of actions recorded includes actions taken to slow down and speed-up.

speed changes with a combined effort of only 10 actions required to achieve this. The point is raised that although this procedure performs very well in all categories as well as successfully avoiding any collisions, the result may not be entirely desirable. If it is preferred for the vehicle to not wait in the middle of an intersection, as could be expected, then further adaptation is necessary to try and limit the creation of these nodes. This could be achieved by either further increasing the collision radius, or incorporating additional rules that prohibit this behavior. The time taken to compute the planned paths for all vehicles is 5.2 seconds with the reduced time attributed to all vehicles remaining on the ideal path, limiting the size of the node tree expansion.

Head-On Collision Scenario

Figure 4.11 below demonstrates a head-on collision scenario. This scenario is identical to that in the RL chapter for a suitable comparison. The speed changing ability has been removed from all vehicles. Both the cyan and yellow vehicles with low priority perform

efficient manoeuvres to weave around the oncoming traffic and immediately return to the ideal path, as shown in time-steps two and three. The result is a total of only 6 steps off of the ideal path with 10 combined actions needed to avoid collisions. The time taken to compute the planned paths for all vehicles is now only 4.1 seconds, as a reduced action-set implies that much fewer nodes are created and thus the size of the node tree is also reduced.



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 4.10: Head-On scenario: Action cost and path deviation cost per vehicle

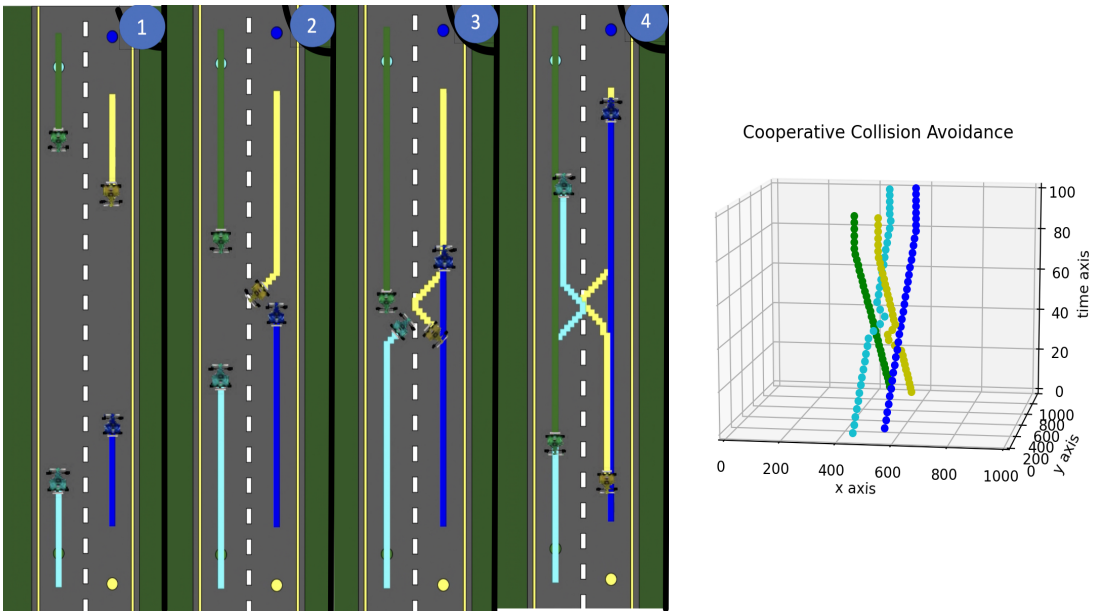
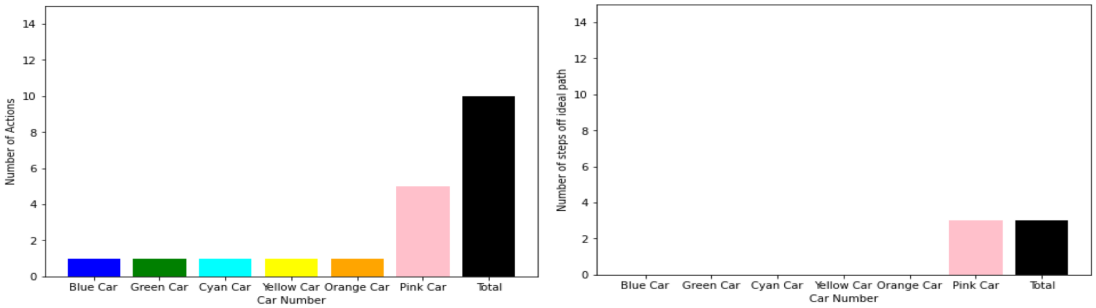


Figure 4.11: Figure showing a Head-On Collision Scenario involving 4 vehicles. Note the scenario does not allow for speed changing capabilities

Crossroad Scenario

The final scenario, illustrated in Figure 4.13 below, is again that of a priority crossroad. However, in this case the speed capabilities have been removed, which requires the vehicles to weave rather than yield at the crossing. The space-time map for the pink vehicle highlights the weaving manoeuvre performed in time-steps two and three, which requires 6 actions to perform. The number of actions required is similar to that of the previous

crossroad scenario with speed changing functionality, although in this case the actions are to the detriment of adhering to the ideal path. The time taken to compute the planned paths for all vehicles is 4.3 seconds, for the same reasons discussed in the previous scenario.



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 4.12: Crossroads scenario without speed changes: Action cost and path deviation cost per vehicle

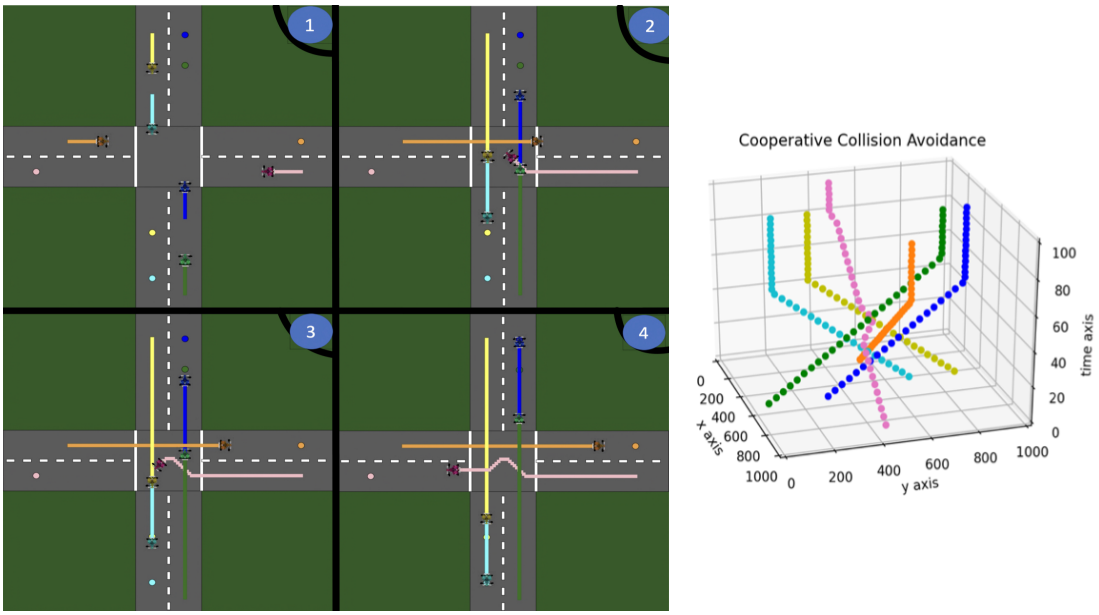
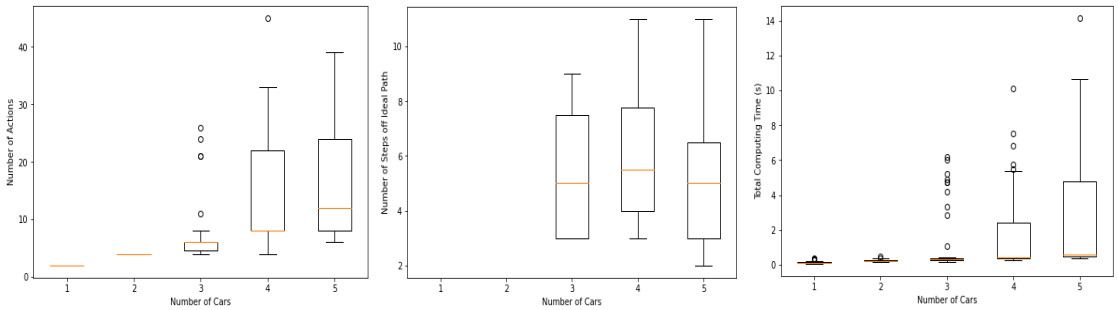


Figure 4.13: Figure showing a Crossroad Scenario involving 6 vehicles. Note the scenario does not allow for speed changing capabilities

Monte Carlo Simulations

In addition to the above-mentioned scenarios, Monte Carlo simulations were performed for the two-way scenario in Figure 3.2a. For the Monte Carlo simulations, the manoeuvres were limited to lane changes, and speed changes were not allowed, to enable a fair comparison with the Reinforcement Learning results in the next chapter. The results represent a distribution for the combined performance of all cars. The Monte Carlo simulation is performed per number of vehicles in the scenario, as this impacts the performance

significantly. For a given number of vehicles, the corresponding distribution is shown, representing 100 simulations where each vehicle's start and end positions are randomized.



(a) Action cost **(b)** Path deviation cost **(c)** Computing Time

Figure 4.14: Figure showing a distribution of the performance measures after 100 simulations for an increasing number of vehicles

The results confirm consistently fast computing times, with a mean time of less than a second for any number of vehicles. However, it also demonstrates that the computing time can be significantly higher in some situations with a high complexity. The action cost increases steadily with an increase in the number of vehicles. It is noticeable that the control effort and path deviation have been kept well within an acceptable range as the distribution is positively skewed. The number of actions required for more than 4 vehicles to avoid collisions becomes much more dispersed, noticeable by the interquartile range increasing suddenly from 2 actions to 12 actions in Figure 4.14a. As shown in Figure 4.14, the distribution confirms that the two-way scenario demonstrated the mean performance and that the metrics reported in the specific scenario provided are not circumstantial.

4.4. Advantages and Disadvantages

The A* algorithm is an efficient algorithm that assesses which is the best direction to take at each stage of searching rather than exhaustively looking through all possible paths [15]. It has been shown that the algorithm can determine optimal paths quickly and thus is applicable to real-time situations. However, it was noted that increasing the number of allowable actions had a significant impact on the duration of the search. This is due to the large branching factor created with many more nodes required to evaluate in the search tree. The heuristic is what gives the A* algorithm power, as it can find the optimal path much more quickly by having a sense of directionality [15]. If this algorithm were to be practically implemented, it would require specific optimized hardware that would enable real-time decision making. One significant advantage of this algorithm is that it can be adapted to generate next states that can fall in a continuous state-space domain. In other words, the A* algorithm can be manipulated to plan with differential constraints, so that the next state is not limited to a finite set of quantized values or discrete blocks.

Furthermore, this also allows for speed-changing capabilities. However, a continuous state-space increases the computing time, but this can be easily controlled by adjusting the inclusion radius of nodes to get the desired resolution and accuracy required.

The ability to alter the cost function to favor certain actions is a major advantage. This allowed the algorithm to prioritize adhering to the ideal path which resulted in suitable performance. More human-like behavior was also achieved as a result of this. For example, consider a driver being stuck behind a truck on a single lane highway. After a long period of waiting for all traffic to clear, the driver would then overtake and naturally speed up afterwards to make up on any lost time. This exact behavior was replicated in the results section and proves the relevance of the algorithm. In all the scenarios that were evaluated, collisions were able to be avoided with minimal control effort and the combined number of actions performed were always kept to as little as was necessary.

All the above-mentioned advantages support the reason why A* has been the preferred algorithm for most path planning problems that require efficient performance. It should be noted, however, that the speed of the execution is entirely dependent on the accuracy of the heuristic [12]. In this problem setup, all vehicles had complete knowledge on exactly where the goal was located as well as of all the other vehicles' future states, which enabled the heuristics in the cost function to be extremely accurate. This may serve as a disadvantage to problems where this is not the case.

4.5. Summary of A*

This section commenced by exploring the concepts involved in a standard A* algorithm. The algorithm design portion described the evolution of the algorithm to comply with speed-changing capabilities. The algorithm was then further improved to function in the space-time domain for compatibility with multiple vehicles. The decoupled hierarchical approach was discussed to solve the task of cooperative collision avoidance. The inner workings of the algorithm were explained and tested in numerous real-world scenarios. These scenarios demonstrated the full functionality of the algorithm whilst also providing a fair comparison scenario for the RL section. It was found that the space-time algorithm performed extremely well in all the given scenarios attributed to its ability to minimize the deviation from the ideal path, along with the total control effort near perfectly. In addition to this, the algorithm was efficient and could be incorporated in real-world applications. However, it was also noted that this was entirely dependent on the accuracy of the heuristics [12], as well as the number of allowable actions, as this would significantly increase the size of the search tree along with the computing time. The space-time A* algorithm was thus concluded to be a feasible solution for cooperative collision avoidance.

Chapter 5

Reinforcement Learning

Reinforcement learning is a subsection of machine learning, particularly useful for so-called "fuzzy" problems, where there is no proper way to perform a specific task although there are clearly specified rules which the model must follow [19]. Training in reinforcement learning happens in a completely different way to traditional machine learning techniques. In contrast to techniques such as supervised learning, RL does not require a set of targets for given inputs, but rather generates data from interacting with its environment [9]. Where supervised learning instantly knows whether a decision was right or wrong from its target, RL is different in the sense that it is dynamic, meaning that the agent only knows that its decisions were correct if it eventually reaches the goal. Reinforcement learning is often known for being able to find strange methods of achieving its goal that are not intuitive to humans [19]. This fact is exactly what makes RL so powerful, in that it is not subject to human biases that may be sub-optimal in many situations.

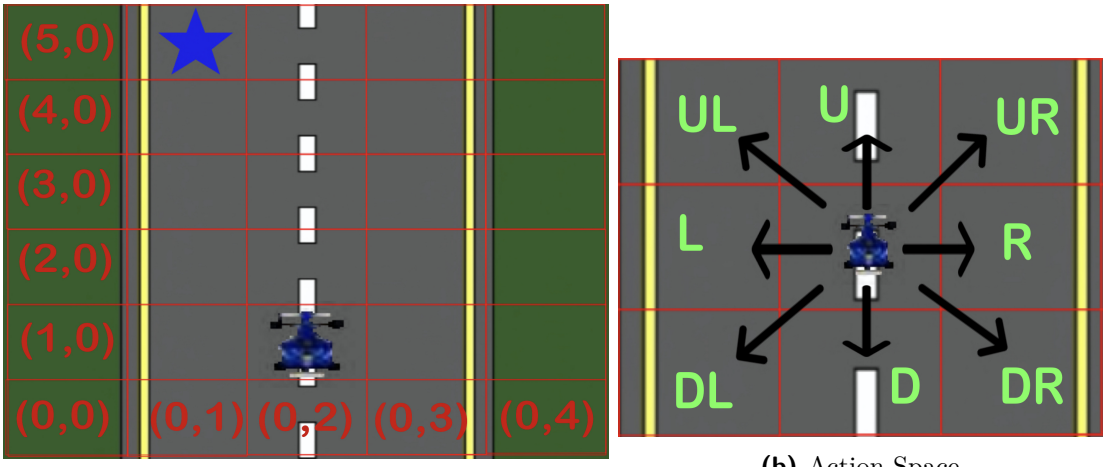
The primary goal is to use reinforcement learning to create an intelligent agent that is capable of performing suitable actions within an environment in order to maximize a given reward. The agent is able to achieve this by following what is referred to as the optimal policy, which is learnt and developed after performing many iterations of a specific episode. These concepts will first be discussed in further detail, before proceeding to the development of the algorithm. This process is best explained with reference to a simple example, of a single agent in an environment which will later be adapted to perform the task of cooperative collision avoidance.

The algorithm will then be tested on the same two scenarios as in the previous section for a fair comparison of the respective performance. The related performance metrics will be further discussed, after which the various advantages and disadvantages of the algorithm examined.

5.1. Algorithm Design

5.1.1. Markov Decision Process

Figure 5.1a below depicts a simple 6x6 grid that will form the environment in this example. Each cell in the grid is referred to as the state S, denoted by the agent's position in 2D space. The set of all possible states in known as the state-space, and in this example is fully visible [20]. As an important side note, states need not be fully visible and may comprise anything from velocities and inter-object distances, to pixels on a screen [21]. The car in this example is known as the agent, and it can take on any of a number of allowable actions, depending on which state it is in. The action space is illustrated in Figure 5.1b and consists of 8 possible actions: 'UP', 'Down', 'Left', 'Right', 'Up Left', 'Up Right', 'Down Left' and 'Down Right'. For a car in state (0,0), the set of allowable actions reduces to: 'Up', 'Right' and 'Up Right'.



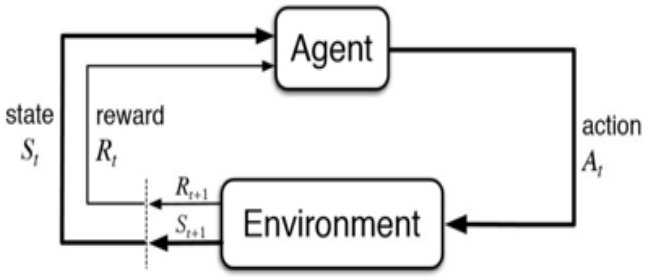
(a) Figure showing an example of the RL grid setup with allowable states

(b) Action-Space

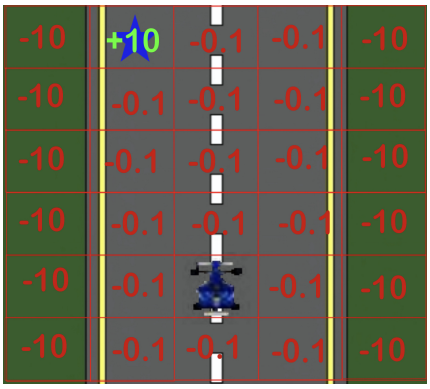
Figure 5.1

With reference to Figure 5.2b, with every action that the car performs, a corresponding reward is received. These rewards are generated from the environment and in this example, the reward is specific to the state in which the car lands. The full process illustrated in Figure 5.2a is what is known as the Markov Decision Process. Recall that the goal of an agent is to maximize the sum of all future rewards. An important concept to realize is that by simply maximizing the reward, the agent will come up with whatever plan is necessary automatically. These rewards therefore need to be carefully designed in order to get the desired behavior from the agent. Figure 5.2b shows the reward setup for this particular example. A reward of -0.1 is given for each step that the car takes, which encourages the car to not continuously wonder around in random directions. By doing this the car would continue accumulating negative reward and therefore is rather encouraged to move towards the goal and thus finish faster. We would like to encourage behavior that results

in the car reaching its goal, and thus a reward of +10 is given if the car reaches the goal state. To prevent the car from driving off the road, a reward of -10 is assigned to the edges of the road which would discourage the car from making any manoeuvres where this may happen. It should be noted that excessive reward management may result in human biases being introduced, where it would rather be more desirable to allow the agent enough flexibility to discover new approaches that may be more optimal.



(a) Markov Decision Process [22]



(b) Figure showing an example of the RL rewards setup

Figure 5.2

Added complexity arises with the interdependence of states. In general, the next state may depend on an entire sequence of previous states. The first order Markov assumption is shown in Equation 5.1. S_t represents the state at a specific time instance and thus $P(S_t|S_{t-1})$ represents the probability of transitioning into the current state given the previous state. Simply stated, Equation 5.1 says that the assumption will be made that the current state is only dependent on the previous state. In this example the state sequence is deterministic, in the sense that for any given state-action pair, there is only one possible next state, which is not always the case in more general problems.

$$P(S_t|S_{t-1}, S_{t-2}, ..., S_1) = P(S_t|S_{t-1}) \tag{5.1}$$

A policy is the equivalent of the agent's brain and is a function that maps the state that the agent is in into a corresponding action and is often referenced by the symbol Π [20]. The objective is to hopefully create a policy that leads the agent to its goal in the most efficient way possible [20]. An example of an optimal policy is shown in Figure 5.3, and represents the best possible actions that the car should take in every state that would lead it to its goal. This process of mapping a state to an action is achieved using a lookup table. However, it should be noted that this function could also be replaced by any function approximator such as a neural network, which is particularly useful when the state space is extremely large and will be further elaborated on later in the chapter [21].

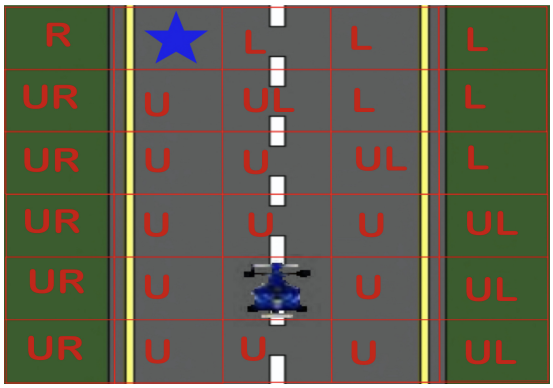


Figure 5.3: Figure showing an example of the Optimal Policy

The car will then explore the environment according to a given policy until it reaches a terminal state. For this case, the terminal state is when the car reaches the goal state (5,1) or when the car exceeds 10 time-steps. The time taken from the car's initial state to reach its terminal state is called an episode.

The Markov Decision Process is particularly useful in any system that is controlled in which the states are fully observable [21]. Now that the basic terminology has been clarified, we will proceed to the derivation of the Bellman equation, which forms the backbone of the entire RL algorithm.

5.1.2. Bellman Optimality Equation

The Bellman equation is the most important equation in Reinforcement Learning and is central to a number of different RL techniques [20]. To derive the Bellman equation, we will begin with the so-called value function, which is a fundamental building block of RL. Simply stated, the entirety of RL boils down to merely different algorithms used to solve for the value function [21]. The Bellman equation forms one of these methods and is what will be used in this example to compute the value function.

Recall that the goal is to create an intelligent agent that will act in such a way as to maximize the sum of all future rewards. The sum of all future rewards is denoted by the Return, $G(t)$, as shown in Equation 5.2, where $R(t)$ represents the reward at a specific time-step.

$$G(t) = R(t+1) + R(t+2) + ... = \sum_{\tau=0}^{\infty} R(t+\tau+1) \tag{5.2}$$

However, rewards that are further in the future should not carry the same weight as more immediate rewards. A discounting coefficient, γ, is added to decrease the weight of future rewards and lies in the range [0,1] with values closer to zero favouring the immediate reward more heavily. This is a hyper-parameter that needs to be chosen through experimentation with typical values of $(0.9 - 0.97)$. Without this discounting factor the sum of future

rewards is potentially infinite.

$$\begin{aligned} G(t) &= R(t+1) + \gamma R(t+2) + \gamma^2 R(t+3) + ... \\ &= \sum_{\tau=0}^{\infty} \gamma^\tau R(t+\tau+1) \end{aligned} \tag{5.3}$$

But how does the car know what the sum of future rewards will be? The answer is directly dependent on the policy that the car is following as well as the state that the car is in. If the car is following the optimal policy demonstrated previously in Figure 5.3 , then the sum of future returns from the car's current position and assuming $\gamma = 0.9$ will be calculated as follows:

$$\begin{aligned} G(t) &= \sum_{\tau=0}^{\infty} \gamma^\tau R(t+\tau+1) \\ &= 0.9^0(-0.1) + 0.9^1(-0.1) + 0.9^2(-0.1) + 0.9^3(10) \\ &= 7.02 \end{aligned} \tag{5.4}$$

It is clearly noticeable that sub-optimal policies would result in many more steps being taken or even negative rewards for policies that lead the car off the road and therefore have a significantly less return.

However, the return is not often calculated deterministically in this way. In order to learn, it is not enough to consider the return under a current policy from a current state, as we would also like to know what would happen if another action is taken that is not dictated by the policy. For any given state, there is a certain probability for any possible action to be taken and thus the corresponding sequence of states and associated return may be different. Therefore, the more pertinent calculation is rather the average value of the return. This realization forms the value-function, $V(s)$, which is described as the expected value of the return, given a certain state s, and it is this function shown below that we seek to maximize.

$$V(s) = E[G(t)|S_t = s] \tag{5.5}$$

The value function is an indication to the agent of how good it is to be in a certain state [20]. For this all-important function to be of any use, it is first necessary to further simplify the value-function into a different form. From Equation 5.6 it is first realized that the return can be written as a recursive function in terms of the return at the next time instance. The expected return at the next state, s', is recognized to be the value-function at this state and thus the computation of the value function can also be written as a

recursive function [21].

$$\begin{aligned} V(s) &= E[G(t)|S_t = s] \\ &= E[R(t+1) + \gamma G(t+1)|S_t = s] \\ &= E[R(t+1)|S_t = s] + \gamma E[G(t+1)|S_t = s] \\ &= E[R(t+1)|S_t = s] + \gamma E[E[G(t+1)|S_{t+1} = s']|S_t = s] \\ &= E[R(t+1) + \gamma V(s')|S_t = s] \end{aligned} \tag{5.6}$$

Calculating the value function at the current state is only determined by the value at the next state. Reflecting further on this result raises some important arguments. There is no need to traverse an infinite number of possible future trajectories or search an entire probability tree, as is necessary with the A* algorithm. In order to get the expected value of the return, it is only necessary to look one step ahead to the value of the next state, and thus the car can plan without having to look far ahead in the future. This is a very important concept necessary for Q-learning which will be further discussed in the next section.

Going forward, it will prove to be more useful to know the value of a certain state-action pair opposed to just the value for a specific state [20]. A new Q-function is developed which has the added feature of also being conditioned on the action at a specific time-step, A_t, and is shown in Equation 5.7.

$$Q(s, a) = E[G(t)|S_t = s, A_t = a] \tag{5.7}$$

The expected reward is broken further down per action, a, and is useful to the agent as it can now answer questions such as: Given a state s, what is the best action to take? Each different policy has a corresponding value function. The optimal policy is the policy with the largest corresponding value-function for all states in the state space, which will also be the optimal value-function. The relationship between the optimal value-function V* and the optimal state-action function Q* is governed by Equation 5.8 below.

$$V^*(s) = \max_a Q^*(s, a) \tag{5.8}$$

Combining Equations 5.6 and 5.8 results in a fundamental property of the optimal state-value function that must satisfy the below equation. Equation 5.9 is the Bellman optimality equation which can be understood intuitively as the breakdown of the current state-action value into the immediate expected reward from the next state and the discounted expected rewards from all future states. This equation proves to be instrumental in finding the optimal policy and will be evaluated using a technique known as Q-Learning [20].

$$Q^*(s, a) = E[R(t+1) + \gamma \max_{a'} Q^*(s', a')] \tag{5.9}$$

5.1.3. Q-Learning

As aforementioned, the goal of Q-learning is to find the optimal policy by first finding the associated optimal Q-values for each state-action pair [21]. The process of continually updating the Q-values is known as value iteration, which is an iterative algorithm that will keep updating these values until the function converges to the optimal function, Q*. Simply stated, we will keep applying the Bellman equation again and again until the Q-values stop changing. The derivation for the Q-learning algorithm can be found in Appendix C with the result repeated in Equation 5.10 below.

$$Q^{new}(s,a) = (1-\alpha)Q^{old}(s,a) + \alpha(R(t+1) + \gamma \max_{a'} Q^{old}(s',a')) \tag{5.10}$$

The new Q-value is the weighted sum of the old Q-value and the learned value. Each value is weighted by the learning rate, α , which is a measure of how quickly the agent will accept new learned Q-values opposed to previous values [21]. The learning rate is also a hyper-parameter that should be chosen through experimentation. Research has shown that a decaying learning rate significantly improves performance and can increase the speed at which the function converges [21]. It would be desirable for each state-action pair to have its own learning rate that will decrease with the number of times the specific state-action has been updated. We will assign $\alpha(s,a) = \frac{\alpha_0}{count(s,a)}$, and thus the agent will learn more for new data and less for state-actions that have already been seen.

To keep track of the results of each state-action pair, the values are stored in a Q-table. This table is initialized to all zero values which will converge to the optimal Q-values over many iterations. To further illustrate this learning process, we will continue from the previous Example in Figure 5.1a and perform two iterations to the given incomplete Q-table extracted from a simulation shown below. Assume a discounting factor of $\gamma = 0.9$ and initial learning rate $\alpha_0 = 0.8$, with the car in its starting state of (1,2) and currently taking actions following the optimal policy, shown in Figure 5.3, corresponding to the largest Q-values in Table 5.1.

Table 5.1: Extract from Q-Table showing all the state-action pairs for a simulation of Example 5.1a. Updates to the table after two iterations are shown in Equation 5.11

ct represents the number of times a certain state-action pair has been updated.

State — Action	Up	Down	Left	Right	Up Left	Up Right	Down Left	Down Right
(1,2)	2.5 $ct = 10$	−1.1 $ct = 5$	1.2 $ct = 8$	−3.6 $ct = 5$	2.1 $ct = 7$	−4.1 $ct = 3$	−2.2 $ct = 2$	−7.3 $ct = 1$
(2,2)	6.7 $ct = 13$	−1.0 $ct = 2$	1.9 $ct = 4$	−2.6 $ct = 5$	5.2 $ct = 2$	−2.1 $ct = 2$	−4.2 $ct = 7$	−6.3 $ct = 1$
(3,2)	5.5 $ct = 8$	−1.1 $ct = 5$	2.2 $ct = 6$	−3.6 $ct = 0$	8.2 $ct = 15$	−3.5 $ct = 4$	−2.9 $ct = 3$	−7.9 $ct = 4$

$$
\begin{aligned}
Q^{new}(s,a) &= (1-\alpha)Q^{old}(s,a) + \alpha(R_{t+1} + \gamma \max_{a'} Q^{old}(s',a')) \\
Q^{new}([1,2],Up) &= (1-\frac{0.8}{10})(2.5) + \frac{0.8}{10}(-0.1 + 0.9 \max_{a'}[6.7,-1,1.9,-2.6,5.2,-2.1,-4.2,-6.3]) \\
&= 2.78 \\
Q^{new}([2,2],Up) &= (1-\frac{0.8}{13})(6.7) + \frac{0.8}{13}(-0.1 + 0.9 \max_{a'}[5.5,-1.1,2.2,-3.6,8.2,-3.5,-2.9,-7.9]) \\
&= 6.74
\end{aligned}
\tag{5.11}
$$

The method shows how the Q-table is updated after every time-step. As we naively follow the optimal policy in this example, it is expected for the updated Q-values to increase with each iteration. However, it is not desirable to naively choose an action in each step according to the highest Q-value. Likewise, it is also undesirable to continually perform random actions as the agent would act sub-optimally most of the time and result in long episodes. Q-learning is an off-policy method, meaning that the action that is actually taken does not have to be the best possible action according to the Q-table [20]. Random actions may also be taken which allows for adequate exploration and has been shown to still result in the convergence of Q* [20].

We would like to strike a suitable balance between exploiting the current highest Q-value whilst also exploring whether other possible actions may lead to better outcomes. To solve what is known as the explore-exploit dilemma, the strategy of epsilon greedy will be applied.

$$
Action - at - time(t) \begin{cases} \max Q_t(a); & P > \epsilon \\ any - action(a) & P < \epsilon \end{cases} \tag{5.12}
$$

A random number P [0,1] , will be generated and depending on the chosen value of epsilon, will either exploit the maximum Q-values for $P > \epsilon$ or choose to explore a random action for $P < \epsilon$. Epsilon is a hyper-parameter and is often chosen as a fixed value depending on the nature of the problem. However, research has suggested that superior performance can be achieved by using a decaying value of epsilon [21]. The concept is raised that initially, the agent will have a very poor understanding of its environment and as the agent begins to gain more experience, should begin relying less and less on random actions and start exploiting the optimal Q-values more often. A decaying epsilon is thus chosen as shown below in relation to the current number of iterations, n, that have been performed.

$$
\epsilon = \frac{0.5}{1 + 0.001n} \tag{5.13}
$$

5.1.4. Cooperative collision avoidance

Up until this point the main focus has been placed on a single agent being able to reach its goal. To expand this idea further to multiple agents reaching their respective goals

without collisions, further adaptation of existing concepts is necessary. Very little research has been conducted on the implementation of reinforcement learning in cooperative collision avoidance systems and that which has been conducted requires the application of further approximation techniques. For example, a method suggested by Q. Wang involves constructing an augmented state vector where the state is a combination of the states of the entire system, consisting of each vehicles position [23]. The goal state would then be the state in which all cars had reached their respective goals and only at this state will the system be rewarded. This is a centralized approach in which the agent consists of multiple vehicles that would cooperatively perform actions to reach the goal state.

However, with this approach the problem arises that the number of possible state-action pairs becomes enormous and will require much more iterations to converge. Assuming only two vehicles (4 states) in the previous 6x6 grid example, there will now be $8^2 = 64$ possible action combinations, with $6^4 = 1296$ possible states. From the equation below, the number of possible state-action pairs is thus $N = (64)(1296) = 82944$ which is significantly larger than the $N = 288$ required for a single agent.

$$Q - Table - size = (Number - Of - Actions)(Number - Of - States) \tag{5.14}$$

This value increases exponentially with the number of states and becomes infeasible for more than two vehicles. Thus, Q. Wang suggests using the method of least squares as an approximation method to estimate the Q-value given a specific state-action pair [23].

Hence, a new decoupled method, similar to the implementation of A* is proposed. The vehicles will be decomposed into single agents with a hierarchical priority order. However, vehicles with a lower hierarchy will need to be aware of the calculated policy of the previous vehicles, so that the expected vehicle locations can be taken into account. The decoupled approach reduces the number of state-action pairs to the equivalent of a single agent, which is then repeated per vehicle. The computation complexity is thus linear and does not increase quadratically as was the case with centralized reinforcement learning.

Each agent will need to be discouraged from moving into states where vehicles with a higher priority are located. However, as the location of the higher priority vehicles changes with each time-step, the process is not as simple as merely allocating a negative reward as was previously performed. The invention of a new adaptive reward changing function is necessary to constantly update the rewards table for an agent at every time-step. The process is best illustrated through an example. An additional vehicle is added to the previous example and assigned a policy as shown in Figure 5.4. Assuming that the pink vehicle has the highest priority, the policy for this vehicle would then be calculated in a standard way. For the blue vehicle, the reward table over the first two time-steps is illustrated in Figure 5.5.

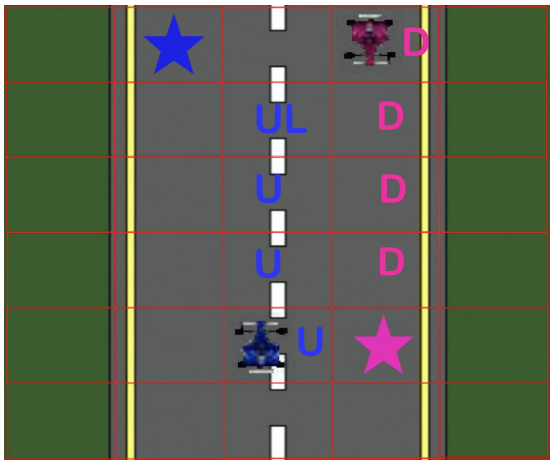


Figure 5.4: RL Decoupled Hierarchical Cooperative Collision Avoidance Example

Allocating a reward of -10 at the location of priority vehicles will adjust the policy of secondary vehicles in such a way that it will now take actions that would avoid collisions.

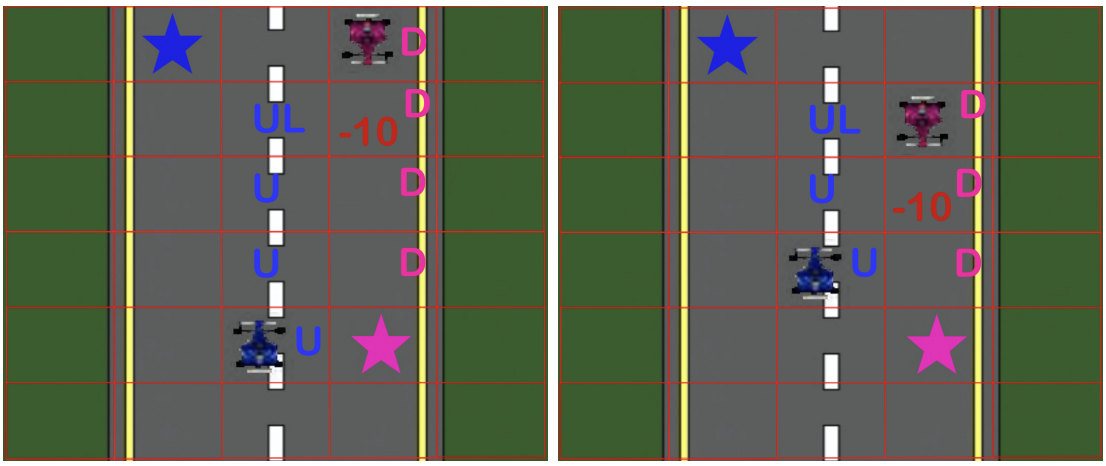


Figure 5.5: RL Cooperative Collision Avoidance Rewards Setup Over Two Time-steps

5.2. Algorithm Implementation

Pseudo-code for the procedure of the decoupled approach to cooperative collision avoidance through reinforcement learning is shown in Algorithm 1 from Figure 5.6. The full process involves a hierarchical method where the desired hierarchy of the cars is specified by the user depending on the scenario at hand. The algorithm is first analyzed here in greater detail before proceeding to the demonstration of the algorithm in various scenarios.

Beginning with the car of the highest hierarchy, the algorithm will start by running a number of iterations for each vehicle, whilst continually leveraging on the learnt knowledge throughout each step. The number of iterations required is entirely dependent on the complexity of the scenario in which a suitable number of iterations should be

allowed so that the Q-values have sufficient time to converge. The Q-values converge when the agent becomes more certain that the learned policy is indeed the optimal policy. One useful strategy to test for convergence of these values would be to keep a track of the old Q-value and after each step record the change from the new Q-value. For each vehicle, the number of iterations could then be plotted against the largest change, after which the number of iterations could be adjusted until the related delta values start to remain constant.

For each iteration, the car will then commence with a single episode. It begins exploring its environment according to the previously mentioned epsilon-greedy strategy, in Equation 5.12, from its initial position. The car will continue to explore until it reaches a terminal state or the maximum number of steps has been performed.

During each episode, the car will repeatedly perform the process of Q-learning. According to the current time-step, the table of rewards is first updated according to the positions of each of the previous cars at every state. An action is then performed after which the corresponding next-state S' and reward are received from the environment. The maximum Q-value from the next state S' as well as the estimated best action a' in that state are also computed. The Q-value for the state-action pair of the current state S is then updated according to the bellman optimality equation that was previously derived. Note that the learning rate $\alpha(s, a)$ has first been adjusted according to the amount of times that the specific state-action pair has already been performed. The agent then moves into the next state and continues this process until termination.

After the optimal Q-table has been calculated for a specific vehicle, the corresponding optimal policy can now be easily obtained. For each state in the state space, the policy for a specific state is simply determined to be the largest Q-value for that state. The argument for the largest Q-value is the related best action and will be recorded in the current policy table.

This policy will then be added to a list of all the cars' policies. This will be used to calculate the state of each vehicle in a specific time-step, which is imperative to constructing the rewards table for the proceeding vehicle of lower hierarchy.

5.3. Algorithm Verification and Results

The cooperative collision avoidance algorithm is tested in two relevant real-world scenarios where acceptable performance can be verified. The same two scenarios that were used in the section on A* will be inspected so that a direct comparison between the two algorithms

Algorithm 1: Reinforcment Learning PathPlanning Algorithm

```
for car = 1...numCars do
    Initialise Q(s,a) table = 0;
    Initialise ALPHA,GAMMA;
    Initialise allPolicies;
    for n = 1...numIterations do
        s = start-state;
        a = argmax(Q[s']);
        timestep = 0;
        while state!=goalState AND timestep<20 do
            a = epsilon-greedy(a , epsilon=0.5/(1+0.0001n));
            update-rewards-table(allCarStates(timestep));
            s' , reward = do-action(a);
            a' = argmax(Q[s']);
            Qs'max = max(Q[s']);
            count[s][a] = count[s][a] +1;
            ALPHA[s][a] = ALPHA/count[s][a];
            Q[s][a] = (1- ALPHA[s][a])(Q[s][a]) + (ALPHA[s][a])(reward +
              GAMMA(Qs'max));
            s=s';
            a=a';
            timestep = timestep+1;
        end
        Initialize policy;
        for s = start-state...allStates do
            best-action = argmax(Q[s]);
            policy[s] = best-action;
        end
        Return policy;
    end
    Append policy to allPolicies;
    allCarStates = calculate-car-states(allPolicies);
end
```

Figure 5.6

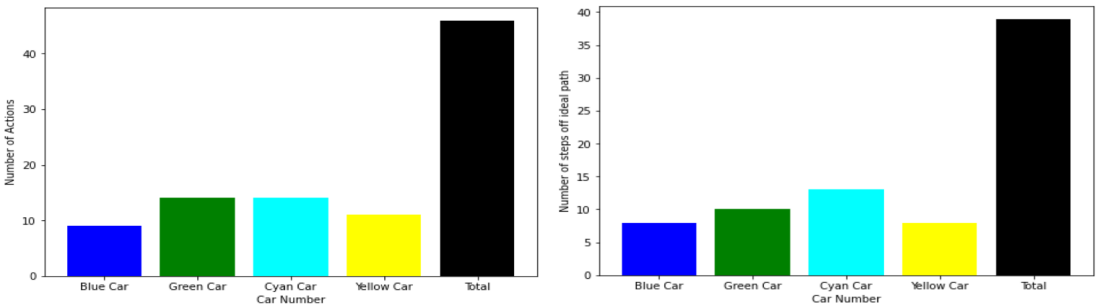
is possible. The scenario is set up according to the same principles as was developed in the design section and expanded to be compatible on a 20x20 grid. The ideal path in each case will be assumed to be a straight line from the start to the goal point. Each of the provided images contain snapshots of the vehicles for 6 different time-steps so that the relative motion of the vehicles can be illustrated.

5.3.1. Head-On Collision Scenario

The first scenario will be that of multiple head-on collisions with 4 vehicles. The hierarchy of the vehicles in this scenario from highest to lowest priority is: blue, green, cyan, yellow. The program is set to run for 20000 iterations per vehicle. During the iterative learning process the policy at 500 iterations as well as 10000 iterations are recorded so that the

learning process can be demonstrated. The respective hyper-parameters for the tested scenario are: $\alpha = 0.3$ and $\gamma = 0.9$.

500 iterations



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 5.7: Head-On scenario 500 iterations: Action cost and path deviation cost

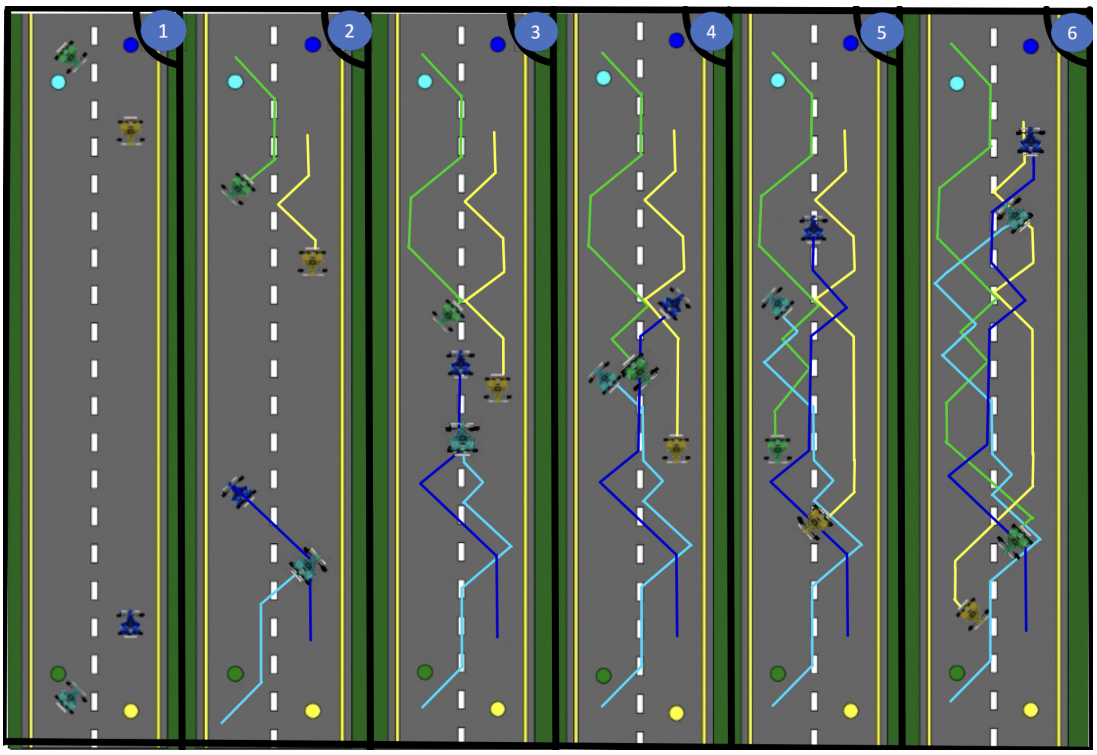
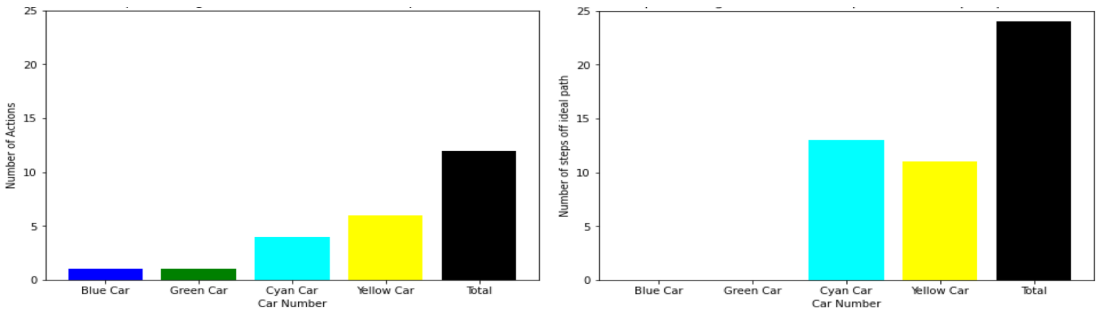


Figure 5.8: Figure showing a Head-On Collision Scenario involving 4 vehicles after 500 iterations.

After 500 iterations, as shown in Figure 5.8, it is clear that the Q-values have not yet converged, resulting in the erratic actions performed by all vehicles. Although the agent has been able to learn which actions to take to avoid collisions, it has not yet optimized each of the vehicle trajectories. There are a total of 46 actions taken between all the vehicles from the start positions to the goal. A total of 39 steps were spent off of the

ideal path. The time taken to compute the planned paths for 500 iterations is 1.9 sec per vehicle.

10000 iterations



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 5.9: Head-On scenario 10000 iterations: Action cost and path deviation cost

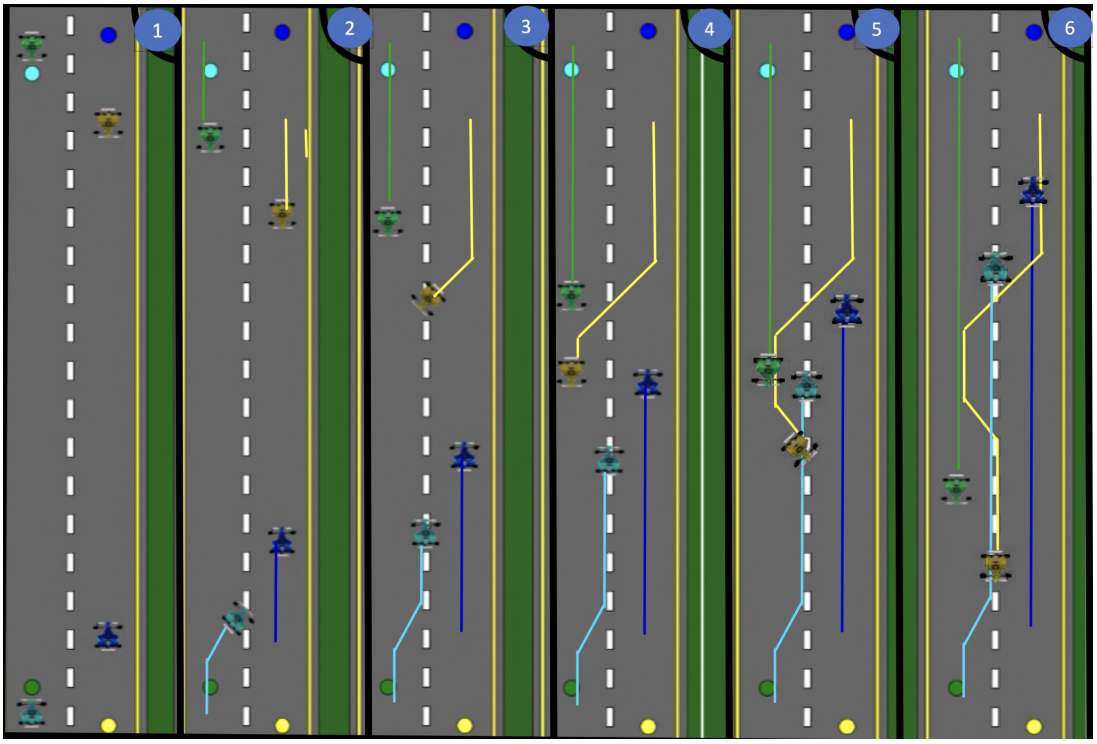
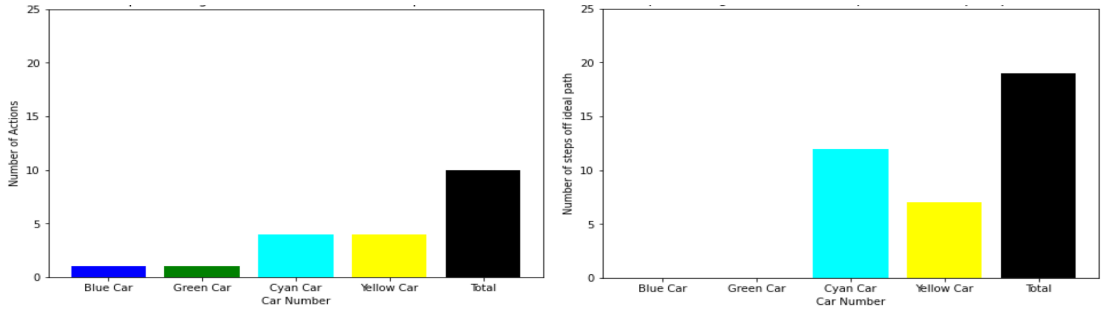


Figure 5.10: Figure showing a Head-On Collision Scenario involving 4 vehicles after 10000 iterations.

After 10000 iterations, as shown in Figure 5.10, the Q-values seem to have converged as a total of only 12 actions have now been taken. Although this seems to be a suitable solution due to the combined control effort being relatively low, it should be emphasised that the number of steps off the ideal path is not desirable. The total number of steps off the ideal path amounts to 24 steps. As shown in time-step 2, the Cyan car makes a lane change long in advance of when the action is actually necessary. Similarly, time-step 5

shows how after the collision is avoided, the Cyan and Yellow cars fail to return back to their original lane until they are within close proximity of the goal. The time taken to compute the planned paths for 10000 iterations is 20 sec per vehicle.

20000 iterations



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 5.11: Head-On scenario 20000 iterations: Action cost and path deviation cost

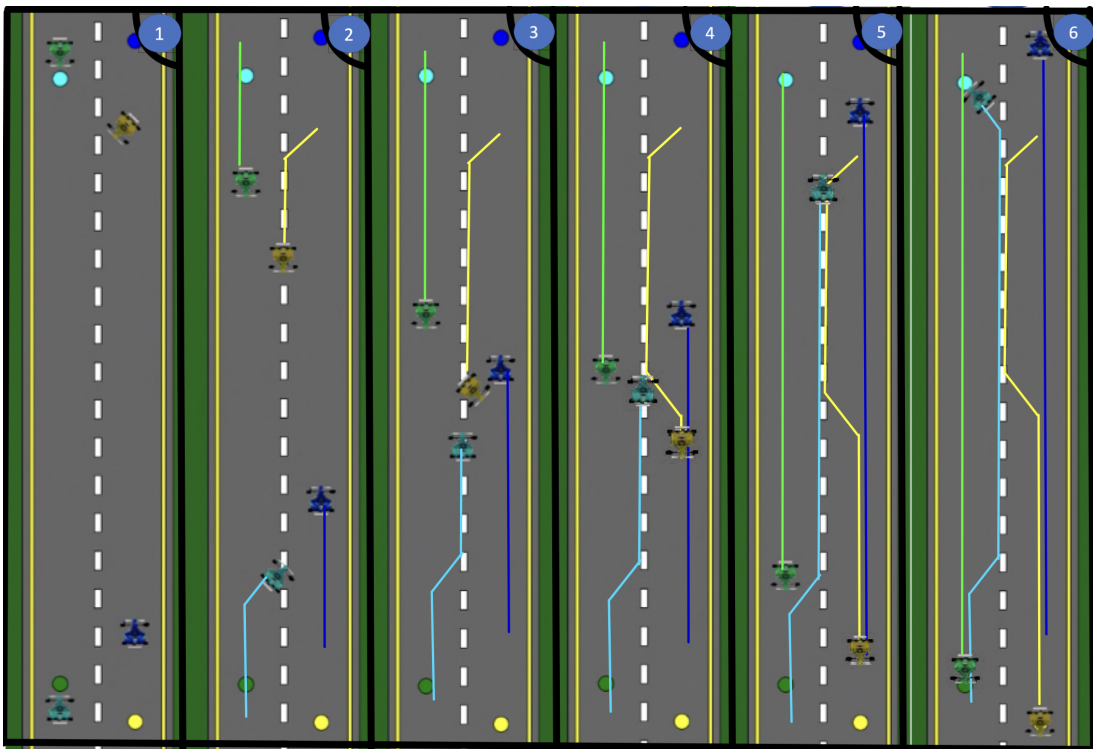


Figure 5.12: Figure showing a Head-On Collision Scenario involving 4 vehicles after 20000 iterations.

After 20000 iterations, the overall performance has been further improved. Only 10 actions are now necessary for successful collision avoidance. To achieve this, the agent has learned to perform a rather risky but also rather impressive manoeuvre. In time-step 3, the Yellow vehicle performs an intricate tactic of weaving between the small space between the oncoming two vehicles. It can be argued whether this manoeuvre is desirable, but

nevertheless it demonstrates an important feature of RL, which is that of being able to find inventive optimal solutions that are unconstrained by human biases by simply trying to maximize the reward [21]. The time spent off of the ideal path has improved to 19 steps but is still not ideal as time-step 2 exhibits the same behavior of taking actions earlier than necessary. The time taken to compute the planned paths for 20000 iterations is 40 sec per vehicle

As previously mentioned, a useful strategy to test for convergence of Q-values is to keep a track of the old Q-value and after each step record the change from the new Q-value. For the vehicle of the lowest priority, the number of iterations is then plotted against the largest delta Q-value from a single episode. Figure 5.13 below shows that after 20000 iterations the Q-table has now completely converged, and there would thus be no benefit in simulating for more iterations.

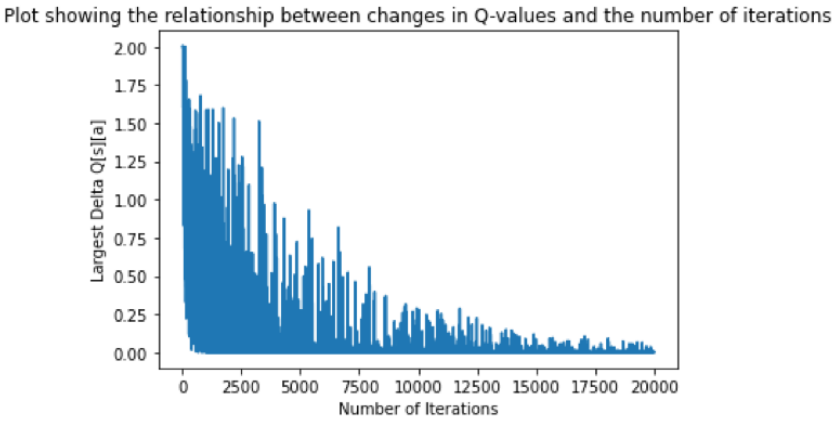


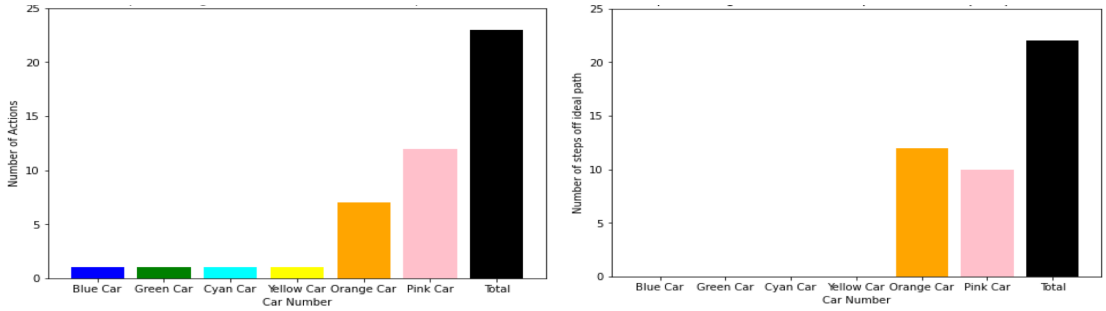
Figure 5.13

5.3.2. Crossroad Scenario

The second scenario will be that of a crossroad with a priority road and 6 vehicles. The scenario seeks to prove the viability of the algorithm for more complex use cases. The hierarchy of the vehicles in this scenario from highest to lowest priority are: blue, green, cyan, yellow, orange, pink. The program is set to run for 30000 iterations per vehicle. The respective hyper-parameters are equivalent to the previous scenario. The reward setup is as such to prevent the cars from driving off-road.

15000 iterations

After 15000 iterations, the two non-priority vehicles have found an alternative route around the flow of traffic to reach their destinations safely. The manoeuvres from these vehicles are by no means ideal as although the car is on the road, they are far removed from their ideal paths for 22 steps and have veered significantly onto the priority road as shown in



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 5.14: Crossroad scenario 15000 iterations: Action cost and path deviation cost

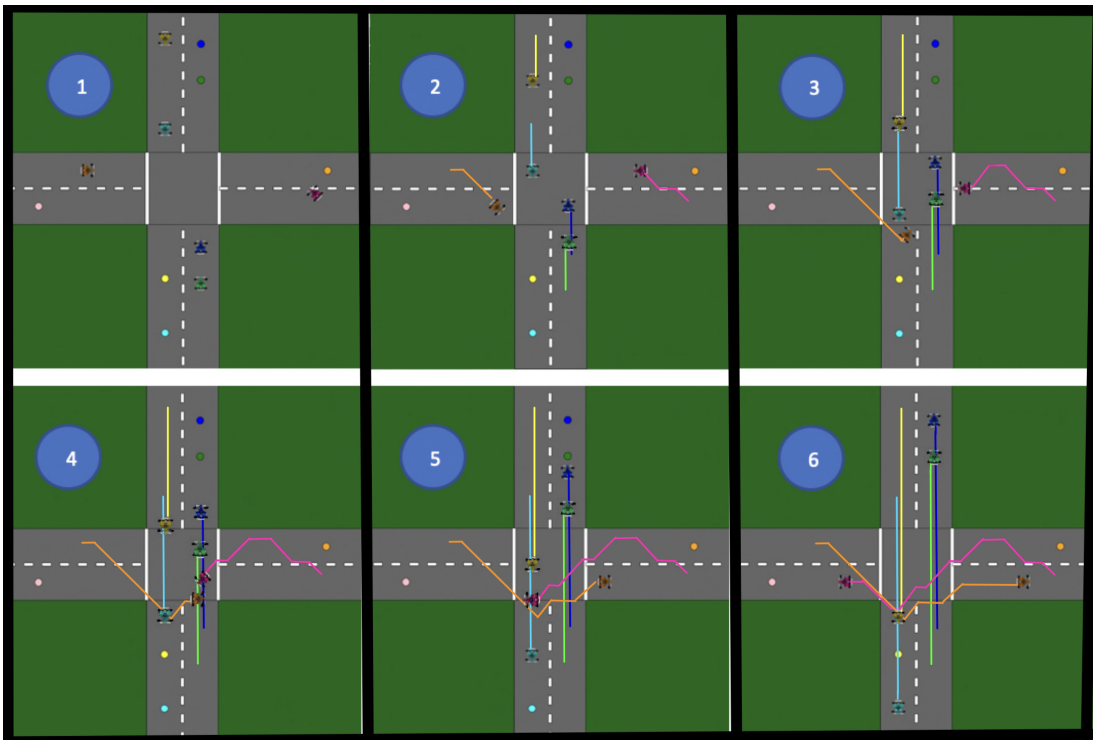
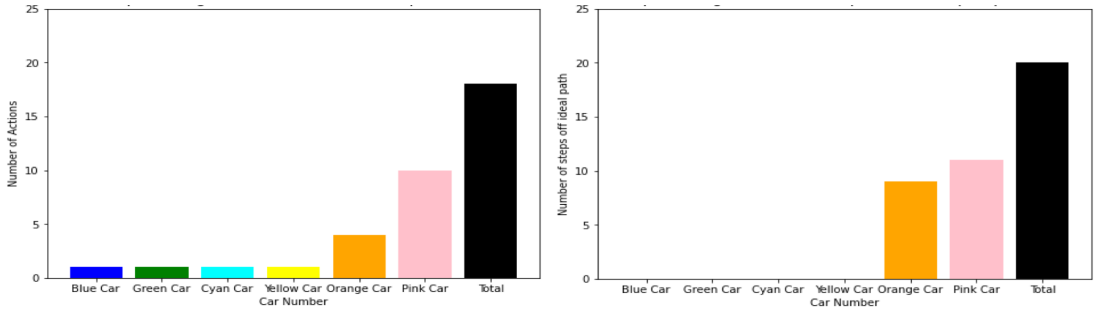


Figure 5.15: Figure showing a Crossroad Scenario involving 6 vehicles after 15000 iterations.

time-steps 3 and 4. The time taken to compute the planned paths for 15000 iterations is 30 sec per vehicle.

30000 iterations

However, after 30000 iterations, the two non-priority vehicles have found a more suitable path to weave between traffic on the priority highway with only 18 actions necessary. This scenario proves the capability of the algorithm to handle more complex situations with many vehicles. To get to this result, it is noted the considerable increase in the number of iterations necessary. Figure 5.18 below shows a much more gradual convergence in Q-values with around 3 times the number of iterations necessary than the first scenario to achieve acceptable performance.



(a) Number of Actions taken **(b)** Number of steps off the ideal path

Figure 5.16: Crossroad scenario 30000 iterations: Action cost and path deviation cost

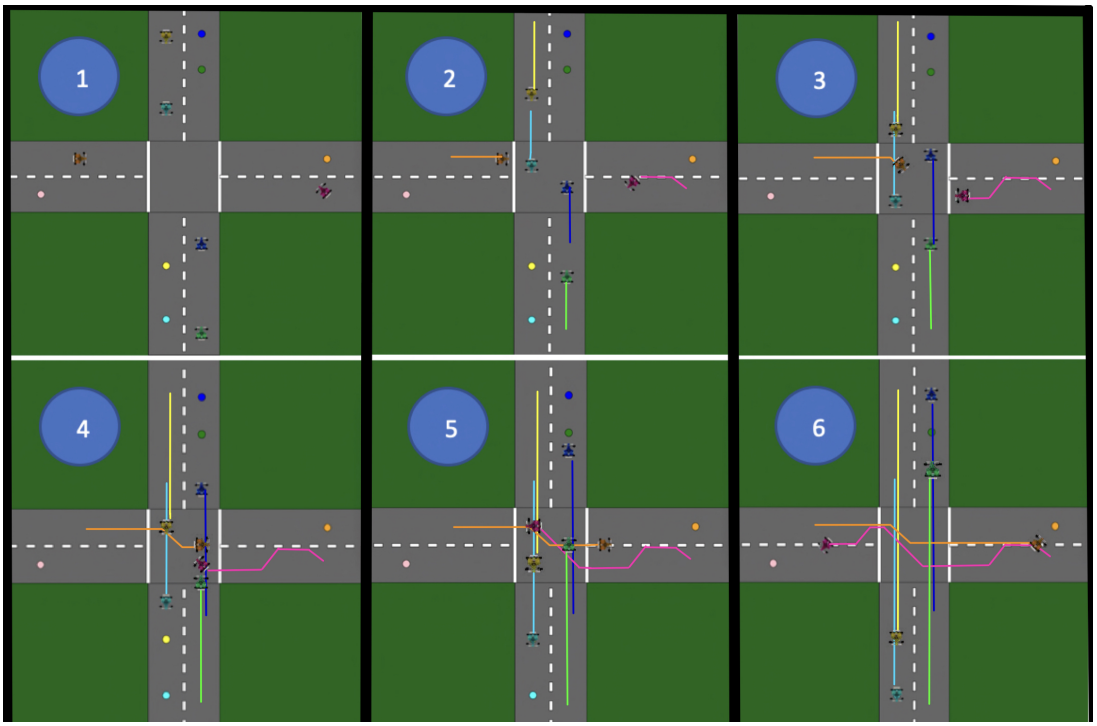


Figure 5.17: Figure showing a Crossroad Scenario involving 6 vehicles after 30000 iterations.

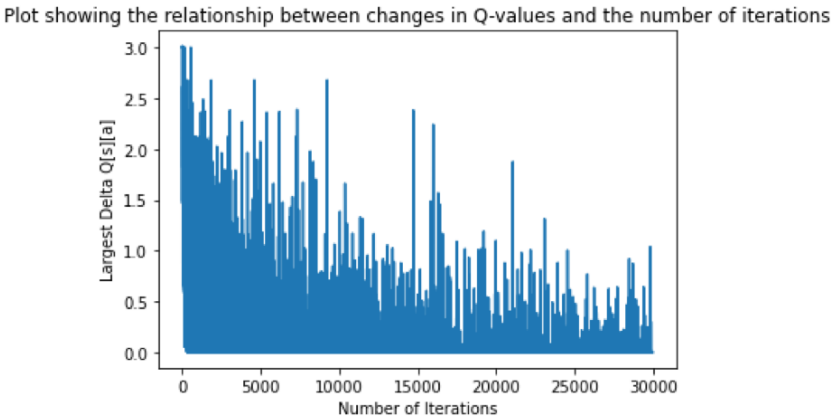
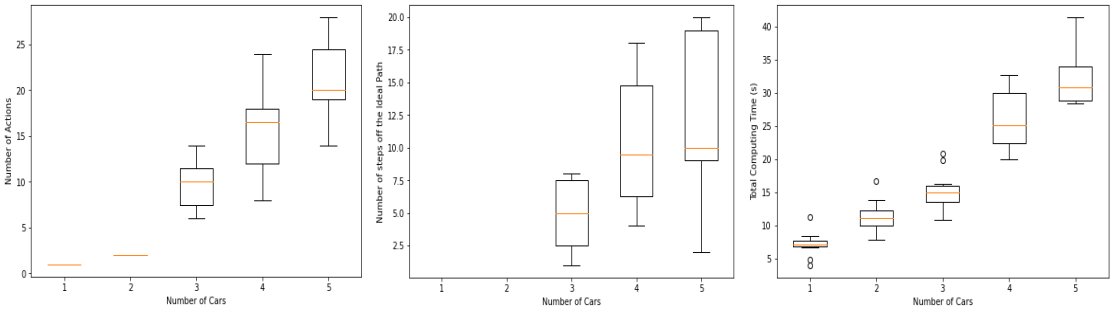


Figure 5.18

5.3.3. Monte Carlo Simulations

As was done in the section on A*, Monte Carlo simulations will be performed on the single lane highway scenario. Again, the distribution for a given number of vehicles is shown, representing 100 simulations where each vehicle's start and end positions are randomized. With RL, it is further necessary to perform the same number of Monte Carlo simulations for a varying number of training iterations. As was shown in the demonstrated scenarios, the performance is greatly impacted based on the number of training iterations allowed. The Monte Carlo simulations for 500 iterations as well as 2000 iterations can be found in Appendix D, which highlights clear improvements in the number of actions taken and marginal improvements to the ideal path deviation. There is a noticeable increase in the computing time required per vehicle. The final simulation after 10000 iterations is shown in Figure 5.19.

10000 iterations



(a) Action cost **(b)** Path deviation cost **(c)** Computing Time

Figure 5.19: Figure showing a distribution of the performance measures for 10000 training iterations and after 100 simulations for an increasing number of vehicles

After 10000 iterations there is clear evidence of convergence. As expected for two or less vehicles, in different lanes, there is no deviation from the ideal path or any erratic actions. There is much better consistency in the number of actions performed, as the interquartile range for all actions in Figure 5.19 has just about halved from that after 2000 iterations in Figure D.2 in Appendix D. For 4 vehicles there is on average 16 actions necessary in order to avoid collisions, which is a respectable result. This is still much higher than the 10 actions required by A* for the same scenario, but it can be argued that the function has not yet completely converged, as it was noticed in the previous scenario that there were still improvements when 20000 iterations were reached. Further increasing the number of iterations is however impractical to perform 100 Monte Carlo simulations on, as for 5 vehicles with 10000 iterations, an average of 2 minutes computation time per scenario is necessary which correlates to multiple hours of simulations. Figure 5.19b clearly demonstrates no particular trend in minimizing the deviation from the ideal path, with the

number of iterations having very little effect in this area. The data for this metric is very dispersed, and the trend indicates a drastic increase with the number of vehicles involved.

5.4. Advantages and Disadvantages

The results for the illustrative scenarios highlight some of the major advantages and disadvantages of Reinforcement Learning. One known advantage of RL that was observed is the innovative ability of the algorithm to come up with entirely new solutions that may not have been considered by humans [9]. It is able to find interesting optimal paths to the goal without the need for any direction or guidance from the user. However, this capability also brings about a few drawbacks. Although these methods may be optimal in terms of maximizing the reward, they may not be completely desirable [9]. Each scenario has illustrated that at times, dangerous and erratic manoeuvres may be performed to avoid collisions, opposed to the more obvious path. The agent is incapable of judging the circumstances surrounding a specific action and only knows whether such an action resulted in overall success.

RL is therefore definitely not suited for situations where certain behaviour is expected, and cases where many constraints need to be adhered to. In trying to incorporate various constraints into the model, the focus then shifts to a task of highly detailed reward management in order to get the desired behaviour out of the vehicle. Deciding on the relative weighting of rewards is not a simple task to perform. This is clearly not the purpose of RL, and it should be able to figure out different optimal solutions to solving problems.

One further drawback is that the number of steps off of the ideal path is too large. For the agent, the time in which some decisions are made has no influence on the expected future rewards. There is therefore no incentive for the agent to stay on the ideal path. This leads to cars driving on the incorrect side of the road for unnecessarily long periods of time. One could attempt to rectify this by adding slightly more negative rewards for leaving the ideal path, but this would once again require complicated reward management to implement.

Another advantage of RL is that after training it does not have to traverse a tree of all possibilities to find the goal. For this reason, RL has the potential to find solutions relatively quickly, but this solution will not necessarily be the optimal solution. However, finding an optimal solution has proved to require many iterations, especially for complex solutions, which can make the algorithm less suitable for certain situations.

Improvements could however be made if there was an augmented system in which vehicles could fully cooperate to finding optimal solutions as was proposed by Q. Wang [23].

Though, it was also shown that this solution would rely on function approximators to estimate the optimal action from a specific state. The decoupled solution was proposed to solve these shortcomings but was shown to also encompass various drawbacks.

The decoupled solution does not scale very well for massive problems, as we observed that the number of iterations needed to find an optimal solution increases significantly with the number of vehicles. This is due to the increased complexity of a rapidly changing rewards grid that requires enough time to converge.

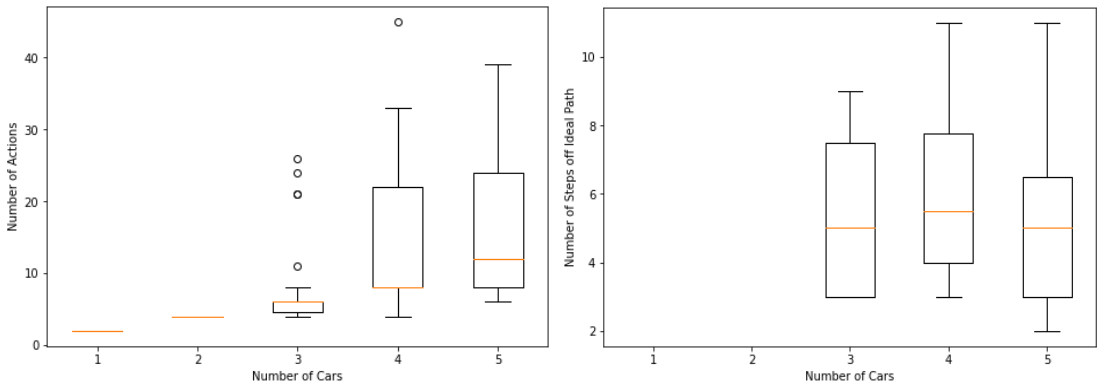
Reflecting upon the time taken to discover a solution, one would realize that the ability to incorporate this algorithm into real-time applications may be problematic. It must however be realized that this solution made no attempt to optimize the computing time for this specific scenario and was implemented on hardware that is not best suited for the task. It also raises the concept of when and how training should occur. This is by no means a solution that should require training in real-time when a collision is imminent. For this to be applicable, further improvement is necessary to change the state-space and reward setup to encompass inter-object distances opposed to using a coordinate. This would allow the agent to then be more adaptable such that it is capable of adjusting to new environments without the need for retraining in each scenario.

5.5. Summary of RL

Various reinforcement learning concepts were explored and then further adapted to perform the task of cooperative collision avoidance. The decoupled hierarchical approach was studied in two real-world scenarios of varying complexity. The performance of the algorithm was measured according to the combined control effort of the vehicles as well as the deviation from the ideal path. Performance in both metrics was found to improve to acceptable levels with an increasing number of iterations. It was noted that there was still plenty room for improvement with regards to the deviation from the ideal path as all improvements found in this metric were merely a byproduct and not something that the algorithm intended to optimize. It was also found that significantly more iterations were necessary to achieve acceptable performance in the more complex scenario and thus lead to concerns towards the scalability of the algorithm. Nonetheless, the algorithm showed rather promising results and encompasses ample room for further development. Reinforcement learning was thus concluded to be a feasible solution for cooperative collision avoidance although it may not be ideal for such situations where certain behavior is expected.

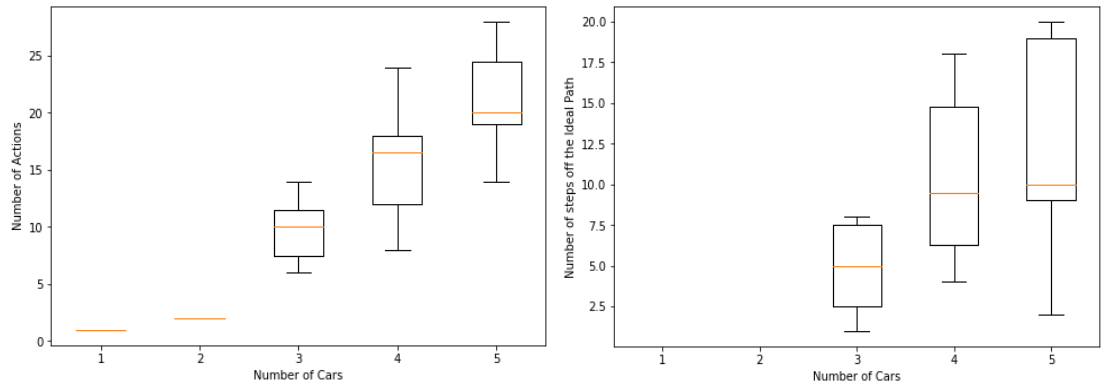
5.6. Comparison between the results of A* and RL

Monte Carlo simulations were performed to directly compare the average performance between the A* algorithm with that of RL, after 10000 training iterations, in the same environment for an increasing number of vehicles. The results have been replicated in Figures 5.20 and 5.21. The distribution of actions for A* is positively skewed, requiring on average around half the number of actions than RL, although it was argued that the RL function had not yet completely converged. A comparison of the ideal path distribution, shows a clear effort from A* to consistently limit the deviation from the ideal path with an interquartile range of less than 4. The distribution for RL shows no clear trend to optimize this metric, with a large interquartile range.



(a) Number of Actions Taken Distribution **(b)** Steps off ideal path Distribution

Figure 5.20: A* - Figure showing a distribution of the performance measures after 100 simulations for an increasing number of vehicles



(a) Number of Actions Taken Distribution **(b)** Steps off ideal path Distribution

Figure 5.21: RL - Figure showing a distribution of the performance measures for 10000 training iterations and after 100 simulations for an increasing number of vehicles

Chapter 6

Summary and Conclusion

Cooperative collision avoidance has become a focus point in the modern era of autonomous vehicles. Various methods were discussed to illustrate the current state of a few solutions that have been proposed. This report immersed into the inner workings of two of the most intriguing approaches. Both A* and reinforcement learning presented two completely different methods, in which the appropriateness of each solution was difficult to judge by merely glancing at the performance of each. These two algorithms were then designed, implemented and tested in various scenarios. By exploring the manner in which both of these algorithms made decisions, further insight was gained into a deterministic vs machine learning approach to the same problem. Both solutions demonstrated the capability of successfully avoiding collisions, but when implemented in the same traffic scenarios, both presented a completely different strategy to collision avoidance.

Monte Carlo simulations were performed to compare the average performance of A* with that of RL, after 10000 iterations. The A* algorithm was efficient at finding optimal paths. It was also adaptable, which meant that the cost function could be easily manipulated to get the desired behaviour, and that vehicle movement could be implemented without having to quantize the state-space. This functionality allowed vehicles to adhere well to the ideal path whilst consistently being able to perform the optimal actions to avoid collisions. RL was also found to be a feasible solution for collision avoidance. It was able to successfully determine collision-free routes towards the goal. However, the time required to train all the agents required numerous iterations and significant computing time, deeming it incapable of real-time applications. The algorithm was able to optimize the control effort required, but failed to prioritize adhering to the ideal path. There was however an unfairness involved in that the RL algorithm had no incentive to stay on the ideal path, as was not the case with A*. The lack in adaptability of the RL algorithm severely complicated any attempts to improve this behaviour, with increased complexity in the reward management involved. RL also demonstrated certain instances of unpredictable behaviour due to the great difficulty involved in constraining certain actions, and as such it was reasoned to be unsuited for situations in which selected behaviour is expected.

Due to the deterministic nature in which this system was modelled and simulated, where precise locations and trajectories were available, implied that conditions were always favourable to A*. The performance of A* is known to be entirely dependent on the accuracy of the heuristics [15], which in this case was extremely precise. For a given hierarchy, A* would always calculate the optimal solution. On the other hand, RL would only tend towards the optimal solution with an increasing number of iterations. This system was therefore not playing to the strengths of RL, as it would be most beneficial in situations that are difficult to model and incorporates an element of uncertainty [19]. This is where the ability to explore ones environment and test out certain actions becomes prominent.

More generally stated, when presented with a particular problem, one should not be under the impression that machine learning could always exhibit the best performance. The nature of the problem should first be thoroughly inspected. Questions such as: Would I be able to solve this problem deterministically? Or, can I fully capture the required functionality in a model? should first be asked. Simply put, machine learning techniques should only be used for problems that cannot be solved effectively in other ways. This report has provided a clear cut demonstration of such a case.

Future work should attempt to expand on the RL approach that was implemented. This algorithm showed plenty room for improvement, with regards to adherence to the ideal path, if more suitable reward management techniques were employed. Deep learning techniques could be incorporated in the current algorithm which would allow for the action-space to be expanded to include speed changing capabilities. Instead of training agents based on their current position or state, research could be done on rather using inter-object distances, which enables training to occur once rather than repeatedly for varying scenarios. Both solutions implemented a decoupled approach, which implies that any solution that is found may only be optimal for the given hierarchy, and not optimal for the entire scenario. Further research into centralized methods could allow for efficient combinations of actions that could greatly outperform some of the best attempts made by sequential planning.

“Machine learning and artificial intelligence will continue to revolutionize industry and will only become more prevalent in the coming years. Whilst I recommend you utilize machine learning and AI to their fullest extent, I also recommend that you remember the limitations of the tools you use — after all, nothing is perfect.” - Matthew Stewart [6]

Bibliography

[1] C.-Y. Chan, "Advancements, prospects, and impacts of automated driving systems," *International journal of transportation science and technology*, vol. 6, no. 3, pp. 208–216, 2017.

[2] Y. Liu, X. Wang, L. Li, S. Cheng, and Z. Chen, "A novel lane change decision-making model of autonomous vehicle based on support vector machine," *IEEE Access*, vol. 7, pp. 26 543–26 550, 2019.

[3] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti *et al.*, "Compute solution for tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020.

[4] F. K. Yılmaz, "Valuation of tesla, inc."

[5] J. Yu and L. Petnga, "Space-based collision avoidance framework for autonomous vehicles," *Procedia Computer Science*, vol. 140, pp. 37 – 45, 2018, cyber Physical Systems and Deep Learning Chicago, Illinois November 5-7, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091831963X

[6] M. Stewart, "The limitations of machine learning —towards data science," https://towardsdatascience.com/the-limitations-of-machine-learning-a00e0c3040c6, July 2019.

[7] A. Yang, W. Naeem, M. Fei, and X. Tu, "A cooperative formation-based collision avoidance approach for a group of autonomous vehicles: Formation-based collision avoidance approach," *International Journal of Adaptive Control and Signal Processing*, vol. 31, 05 2015.

[8] H. A. Izadi, B. W. Gordon, and Y. Zhang, *Rule-Based Cooperative Collision Avoidance Using Decentralized Model Predictive Control*. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2011-1610

[9] K. Budek, "What is reinforcement learning? the complete guide - deepsense.ai," https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/, July 2018.

[10] G. Hoffmann and C. Tomlin, "Decentralized cooperative collision avoidance for acceleration constrained vehicles," 01 2009, pp. 4357 – 4363.

[11] B. Alrifaee, M. Ghanbarpour, and D. Abel, "Centralized non-convex model predictive control for cooperative collision avoidance of networked vehicles," 10 2014.

[12] D. Silver, "Cooperative pathfinding." 01 2005, pp. 117–122. [Online]. Available: https://www.davidsilver.uk/wp-content/uploads/2020/03/coop-path-AIIDE.pdf

[13] W. Li and C. G. Cassandras, "Centralized and distributed cooperative receding horizon control of autonomous vehicle missions," *Mathematical and Computer Modelling*, vol. 43, no. 9, pp. 1208 – 1228, 2006, optimization and Control for Military Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0895717705005467

[14] P. Chotiprayanakul, D. K. Liu, D. Wang, and G. Dissanayake, "A 3-dimensional force field method for robot collision avoidance in complex environments," in *Automation and Robotics in Construction-Proceedings of the 24th International Symposium on Automation and Robotics in Construction*, 2007. [Online]. Available: https://opus.lib.uts.edu.au/bitstream/10453/7180/1/2006009439.pdf

[15] D. Silver, "The problem with a* cooperative pathfinding." 03 2005. [Online]. Available: https://www.davidsilver.uk/wp-content/uploads/2020/03/coop-path-AIWisdom.pdf

[16] J. Sun and Y. Zhang, "A reinforcement learning-based decentralized method of avoiding multi-uav collision in 3-d airspace," in *Proceedings of the 2019 3rd International Conference on Computer Science and Artificial Intelligence*, 2019, pp. 77–82.

[17] Y. Yuan, R. Tasik, S. S. Adhatarao, Y. Yuan, Z. Liu, and X. Fu, "Race: reinforced cooperative autonomous vehicle collision avoidance," *IEEE Transactions on Vehicular Technology*, 2020.

[18] V. V. PV, "Deep reinforcement learning: Value functions, dqn, actor-critic method, back-propagation through stochastic functions — by vishnu vijayan pv — medium," https://medium.com/@vishnuvijayanpv/deep-reinforcement-learning-value-functions-dqn-actor-critic-method-backpropagation/through-83a277d8c38d, August 2020.

[19] M. Kirschte, "What to expect from reinforcement learning? — towards data science," https://towardsdatascience.com/what-to-expect-from-reinforcement-learning-a22e8c16f40c, March 2019.

[20] "Reinforcement learning series intro - syllabus overview - deeplizard," https://deeplizard.com/learn/video/nyjbcRQ-uQ8.

[21] "Artificial intelligence: Reinforcement learning in python — udemy," https://www.udemy.com/course/artificial-intelligence-reinforcement-learning-in-python/.

[22] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Trans. Audio, Speech, Language Process.*, vol. 20, no. 1, pp. 30–42, 2012.

[23] Q. Wang and C. Phillips, "Cooperative collision avoidance for multi-vehicle systems using reinforcement learning," in *2013 18th International Conference on Methods Models in Automation Robotics (MMAR)*, 2013, pp. 98–102.

Appendix A

Project Planning Schedule

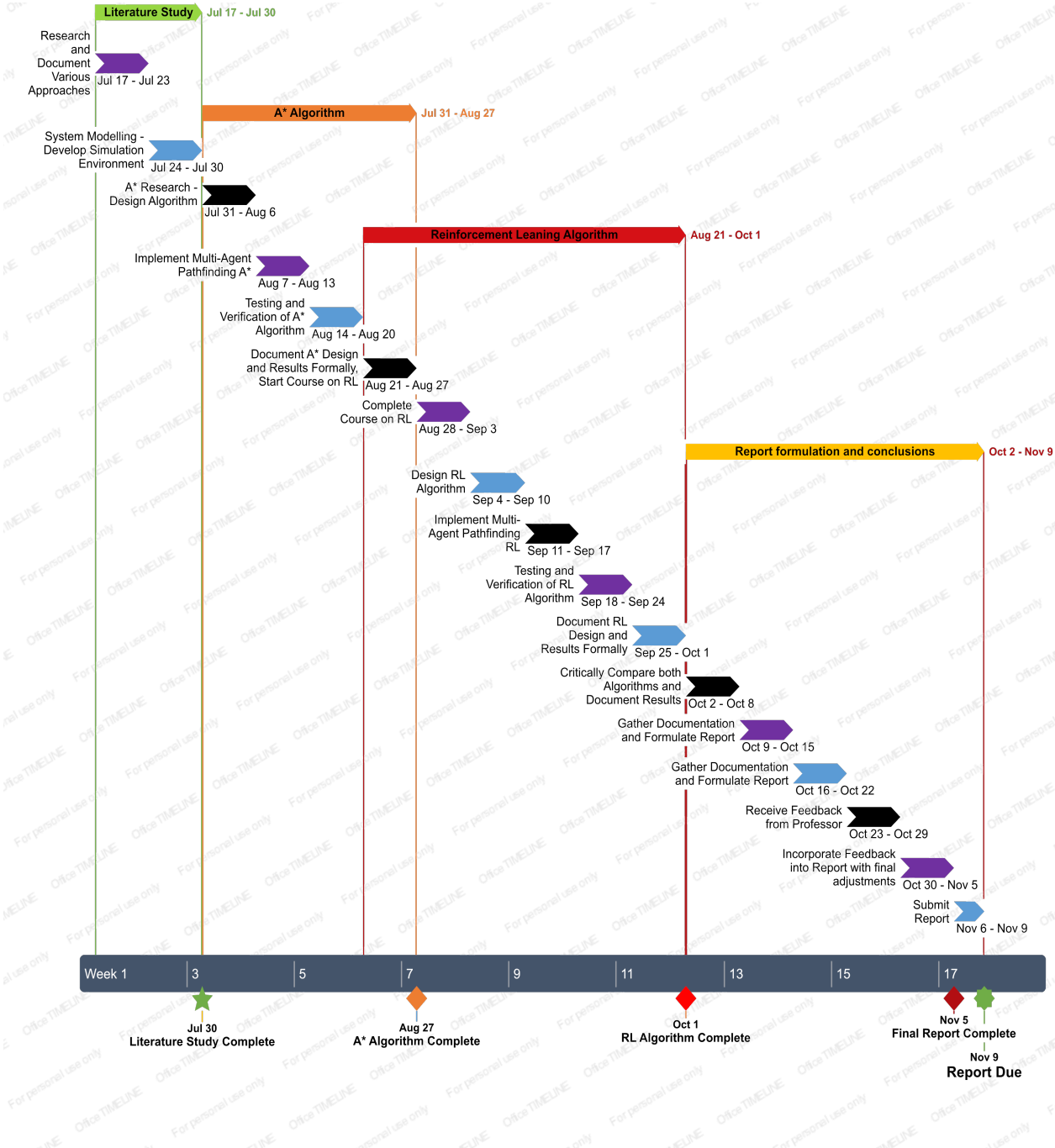


Figure A.1: Figure showing a Gant chart of the planned project schedule

Appendix B

Outcomes Compliance

Table B.1: ECSA outcomes.

Outcome	Description of Outcome in Report	Chapter(s)
ELO 1. Problem Solving	This project required implementing a simulation that performed collision avoidance for autonomous vehicles. The different methods that are currently used were identified and analyzed before formulating an approach to the problem. Current A* methods only allowed for discrete movement and thus it was necessary to innovate a new solution which used the same principles but expanded the functionality of the vehicle such that it could perform speed changes and move in the continuous state-space. Furthermore RL presented the greatest challenge as it required fully implementing an entire working solution from the first principles using equations that were found during research. New equations had to be derived and implemented in code as an entirely new approach was created that had not been done before. Thereafter, to critically analyse the performance, innovative real-world test cases and Monte Carlo simulations were performed so that the benefits and drawbacks of each approach could be discussed.	Chapter 4.1, 4.3, 4.4 Chapter 5.1, 5.2, 5.3
ELO 2. Application of scientific and engineering knowledge	Applied mathematics was used to construct the model for the vehicle used in simulations. Computer science knowledge was used to design suitable algorithms and test scenarios. Machine learning knowledge was used in the section on RL, where techniques such as hyper-parameter tuning and model training were extensively utilized. Furthermore, concepts from control systems, regarding optimization, were applied to the path-planning algorithms.	Chapter 3.1 Chapter 4.1, 4.2, 4.3 Chapter 5.1, 5.2, 5.3
ELO 3. Engineering Design	An engineering design procedure was followed in order to construct both path-planning algorithms. The problem was researched thoroughly, after which possible solutions were brainstormed. The algorithms were implemented and tested to determine whether the requirements were adhered to. An iterative design and improvement procedure was followed until suitable performance in the test scenarios and Monte Carlo simulations were observed.	Chapter 3 Chapter 4.1, 4.2, 4.3 Chapter 5.1, 5.2, 5.6

Table B.2: ECSA outcomes.

Outcome	Description of Outcome in Report	Chapter(s)
ELO 4. Investigations, experiments and data analysis	A complete literature study was performed on existing methods to solve the problem. Additional background research was conducted by reading articles and research papers to fully understand the problem. Numerous experiments such as the real-world scenario tests and Monte Carlo simulations were performed. Each Monte Carlo simulation generated a distribution of costs. This data was critically analysed so that the algorithms could be evaluated, which lead to the formulation of conclusions.	Chapter 2 Chapter 4.3 Chapter 5.3
ELO 5. Engineering methods, skills and tools, including Information Technology	Python was the main engineering tool that was used to implement the algorithms and perform simulations. Proficient knowledge on all of the computer packages and libraries for this software was demonstrated through simulation development and data generation techniques. Competency in Latex was also shown through the writing of this report.	Chapter 4.2, 4.3 Chapter 5.2, 5.3
ELO 6. Professional and technical communication	A professional report was written, in Latex, which communicated the full scope of all the work that was performed. Video presentations also demonstrate competent verbal communication and the ability to communicate complex terminology in a simple manner that anyone can understand.	Throughout the report
ELO 8. Individual, team and multidisciplinary working	Individual work was performed throughout the project, from the learning of new concepts, to the design phase and implementation of algorithms. The report and supporting documentation was individually written.	Throughout the report
ELO 9. Independent Learning Ability	Numerous concepts were learnt throughout the project by reading over 30 articles and research papers on the topic. An online course on RL was completed, and the concepts which were learnt from this course were directly applied during the implementation of the project. Python programming libraries were researched for use in path-planning algorithms.	Chapter 2 Chapter 4.2 Chapter 5

Appendix C

Q-Learning Algorithm Derivation

The derivation of the Q-Learning algorithm begins with Bellman Optimality Equation, shown in C.1 that was developed in Chapter 5.

$$Q^*(s,a) = E[R(t+1) + \gamma \max_{a'} Q^*(s',a')] \tag{C.1}$$

The expectation shown in Bellman Optimality Equation can be approximated by the sample mean, $\overline{Q}_N$, as shown in Equation C.2. The sample mean will approach the expected value after many training iterations.

$$Q(s,a) = E[G(t)|S_t = S, A_t = a] \approx \overline{Q}_N(s,a) = \frac{1}{N}\sum_{i=1}^{N} Q_i \tag{C.2}$$

This sample mean can also be evaluated in terms of the previous sample mean as shown below which is also desirable for storage and processing time limitations.

$$\begin{aligned}
&\overline{Q}_N(s,a) = \frac{1}{N}(\sum_{i=1}^{N-1} Q_i + Q_N) \\
&\overline{Q}_N(s,a) = \frac{1}{N}((N-1)\overline{Q}_{N-1} + Q_N) \quad ; \quad \left\{ \overline{Q}_{N-1} = \frac{1}{N-1}\sum_{i=1}^{N-1} Q_i \quad \therefore \quad \sum_{i=1}^{N-1} Q_i = (N-1)\overline{Q}_{N-1} \right\} \\
&\overline{Q}_N(s,a) = \overline{Q}_{N-1} + \frac{1}{N}(Q_N - \overline{Q}_{N-1}) \\
&\overline{Q}_N(s,a) = (1 - \frac{1}{N})\overline{Q}_{N-1} + \frac{1}{N}(Q_N) \\
&\overline{Q}_N(s,a) = (1 - \frac{1}{N})Q^{old} + \frac{1}{N}(Q^{learned})
\end{aligned} \tag{C.3}$$

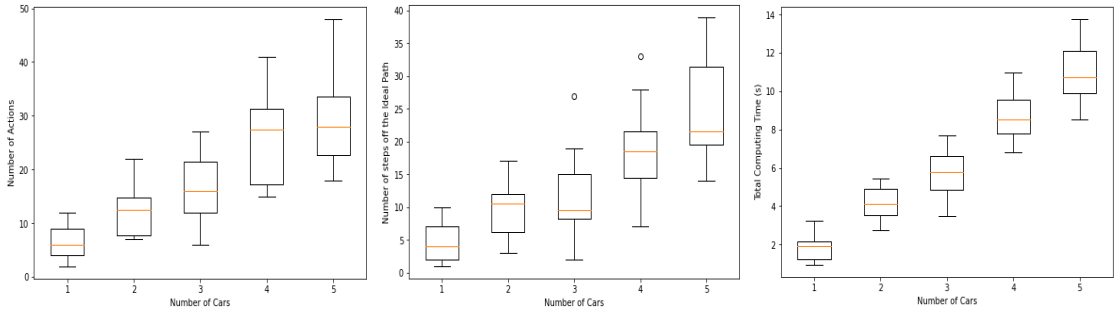
From this result, we notice that the new Q-value can be calculated as the weighted sum of the old Q-value and the learned Q-value. The weighting is assigned the symbol α and is referred to as the learning rate. The learning rate describes how quickly the agent will adopt newer learned values to previously calculated values. Combining the result from Equation C.3 with that of Equation C.1, results in the Q-learning Algorithm C.4.

$$Q^{new}(s,a) = (1-\alpha)Q^{old}(s,a) + \alpha(R_{t+1} + \gamma \max_{a'} Q^{old}(s',a')) \tag{C.4}$$

Appendix D

Monte Carlo Simulations for RL

500 iterations

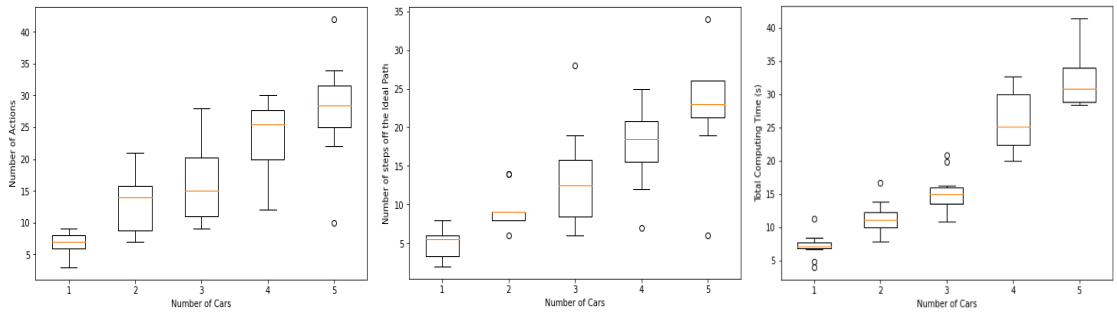


(a) Action cost **(b)** Path deviation cost **(c)** Computing Time

Figure D.1: Figure showing a distribution of the performance measures for 500 training iterations and after 100 simulations for an increasing number of vehicles

After 500 iterations it is once again clear that not enough training time was allowed for convergence. Even with only a single vehicle on the road, there is 8 unnecessary actions performed to reach the goal. The mean computation time is roughly 2 seconds per vehicle, which corresponds to that which was found in the demonstrated scenario.

2000 iterations



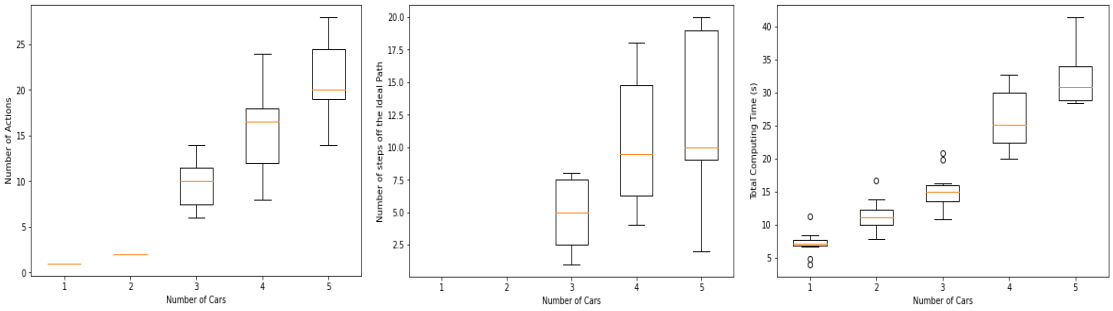
(a) Action cost **(b)** Path deviation cost **(c)** Computing Time

Figure D.2: Figure showing a distribution of the performance measures for 2000 training iterations and after 100 simulations for an increasing number of vehicles

After 2000 iterations, the function has still not converged. There is however a noticeable

improvement in the number of actions taken, although the adherence to the ideal path remains mostly the same. Computation time has now increased to around 8 sec per vehicle.

10000 iterations



(a) Action cost **(b)** Path deviation cost **(c)** Computing Time

Figure D.3: Figure showing a distribution of the performance measures for 10000 training iterations and after 100 simulations for an increasing number of vehicles

After 10000 iterations, there is clear evidence of convergence. As expected for two or less vehicles, in different lanes, there is no deviation from the ideal path or any erratic action. There is much better consistency in the number of actions performed, as the interquartile range for all actions in Figure D.3 has just about halved. For 4 vehicles there is on average 16 actions necessary in order to avoid collisions, which is a respectable result.