L-Università ta' Malta
Faculty of Science
Department of Statistics & Operations Research

TRAFFIC CONTROL OPTIMIZATION USING MARKOV DECISION PROCESSES

Leonard Farrugia

August 2025

Supervisors: Dr David Suda, Ms Monique Sciortino, Dr Kenneth Scerri

A dissertation presented to the Faculty of Science in part fulfilment of the requirements for the degree of M.Sc. in Statistics at the University of Malta

Dedicated to My Late Father

ABSTRACT

Leonard Farrugia, M.Sc.

Department of Statistics & OR, August 2025

University of Malta

Traditional traffic light systems often operate inefficiently, especially in high-intensity traffic situations, due to their reliance on a fixed cycle (FC). These traditional traffic light systems follow predetermined cycles, switching between red, yellow and green signals at fixed intervals, without accommodating real time traffic conditions. As a result of inefficient time allocation, traffic jams often build up. Throughout the years, researchers approached this problem using different techniques, many of which using the Markov Decision Process (MDP) framework. The MDP is a framework for analysing the existence and structure of good policies, which are self-thought rules behind choosing an action and for devising procedures for finding such policies.

The aim of this dissertation is to identify policies that help manage traffic efficiently by considering various factors contributing to traffic congestion. This is achieved by analysing the evolution of the state of the intersection and by applying MDP based policies to have dynamic control of the traffic light system. The results in this study illustrate that an MDP-driven cycle significantly outperforms a traditional FC, particularly during high-intensity traffic conditions, when it comes to managing traffic efficiently. This is demonstrated in this dissertation by virtue of the MDP-model that has been developed and formulated originally by the author.

ACKNOWLEDGEMENTS

My deepest gratitude goes to my tutors, Dr David Suda, Ms Monique Sciortino and Dr Kenneth Scerri, for their patience and invaluable guidance throughout this dissertation. I would also like to take this opportunity to thank my tutors for their profound understanding and empathy, particularly during challenging personal times throughout my studies. Their continuous support and personal time invested are unwavering and greatly appreciated.

On a personal level, I am eternally grateful to my family, who have supported me throughout my studies. I am also greatly indebted to my girlfriend Jessica, whose support and love helped me during the entire dissertation.

Contents

List of Figures

List of Tables

List of Symbols

$\lambda_\circ$	Car arrival rate at location $\circ \in \{S, G, M, K\}$
c^a	Number of arriving cars
t	Time index
μ	Car departure rate
S_t	State at time t, which consists of $\{I_t, C_t\}$
s_t	State realization at time t, where $s_t \in S_t$
I_t	State of intersection at time t
C_t	State of the cars at time t
$k_{max}, l_{max}, m_{max}, n_{max}$	Maximum queue length categories for each road
$A_{s_t,t}$	The action taken at time t at state s_t
a_t	Action realization at time t, where $a_t \in A_{s_t,t}$
k, l, m, n	Queue length realization for each road at time t
k', l', m', n'	Queue length realization for each road at time $t+1$
$\tilde{k}, \tilde{l}, \tilde{m}, \tilde{n}$	Midpoint of queue length category for each road
$\bar{k}, \bar{l}, \bar{m}, \bar{n}$	The current capacity percentage of each road
w_1, w_2, w_3, w_4	Constant ranked weights for reward prioritization
$f_*(\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n})$	Returns the probability of ending in state s_{t+1}, having taken an action in state s_t
$R(s_{t+1}\|s_t, a_t)$	Reward of transitioning to state s_{t+1} after being in state s_t and taking action a_t
γ	The discount factor

Chapter 1

Introduction

This dissertation will be introduced by firstly providing a brief overview of the topic under study, followed by a detailed lost of the terminology that will be used throughout the dissertation. Subsequently, the author provides a literature review together with a generic introduction to MDPs. Lastly, the structure of the dissertation is laid out.

1.1 A Brief Overview

Every day, people make several decisions; decisions which bear immediate and long-term consequences. Thus, decisions must not be made independently; today's decision will have an impact on tomorrow's decision, and tomorrow's decision on the day after. By not accounting for the connection between present and future decisions and outcomes, a good overall performance may not be attained. The decisions humans and computers make usually produce two kinds of effects: (a) they cost or save time, money, or other resources, or they bring revenues, as well as (b) influence the future by conditioning the dynamics. To address and optimise these immediate and long-term consequences, Markov Decision Processes (MDPs) provide a framework for studying the structure and existence of good policies; these being self-thought rules behind choosing an action, and for devising procedures for finding such policies.

MDPs, apart from their advantageous theoretical aspects, are considered by

many researchers as pivotal for real-time problems, due to their specific characteristics; mainly the ability of sequential decision-making and providing optimal policies. Moreover, various engineering and business applications use MDP models. Consequently, the analysis of MDPs gives rise to interesting mathematical and computational challenges. Numerous experts consider Shapley (1953) as the first study on MDPs, since the relationship between MDPs and stochastic games assimilates the relationship between a typical game and an optimisation problem: a stochastic game with a single player is considered an MDP. Additionally, the book by Howard (1960) introduced policy iteration algorithms, which initiated the systematic study of MDPs. Some of the most common applications of MDPs are inventory control (Althaqafi 2024), production planning, cancer treatment (Bayer et al., 2021), and operations management. In this study, the MDP framework will be applied to a traffic management related problem.

In the present era, managing traffic efficiently has become more challenging. Due to the gradual development of road traffic, traffic management is still an evolving problem. Researchers in computer science, as well as in engineering, have approached the problem in various ways throughout the years. In urban areas, traffic is normally controlled by *traffic light systems*, which are sets of red, amber and green lights placed where roads meet that control traffic by indicating when vehicles need to stop and when they can go. Unfortunately, certain traffic light systems operate in a predetermined manner, that is, lights are green for a deterministic period of time, and red for a deterministic period of time while other traffic lights in the system take their turn. Consequently, a traffic jam may build up due to inefficient allocation of traffic light times.

This study focuses on the optimisation of the decision strategies applied to control a traffic light system at an *intersection*, which is an area where two or more roads meet, designated for vehicles to turn to different directions to reach their respective destination. The main goal of this dissertation is to find policies that help manage traffic efficiently by taking into account different measures which represent factors contributing to traffic congestion. This is achieved by analysing and learning the evolution of the state of the intersection, which is the primary focus of this study.

When solving a real-life problem, setting a formalisation is an important initial

step. The most appropriate approach for the problem under study would be that of using MDPs, which will serve as a mathematical framework to formalise the system in this study. In order to solve an MDP, one can opt for a method such as dynamic programming, or methods that learn from experience, such as those which fall under Reinforcement Learning (RL). In this study, a policy is chosen after developing multiple MDP profiles (different times of day with an optimal policy for each time period), depending on the proximity of the simulation to the MDP profiles. The latter are different traffic situations that one may encounter in the test intersection being studied throughout the day.

Section 1.2 presents an extensive list of important terminology related to the application under study, together with their respective definitions, which will be mentioned over the course of this study. This section retains its relevance since such traffic related terms may not be common to the average individual.

1.2 Terminology

This list contains the explanations of all traffic related terms used in this dissertation.

- *Waiting time* - The time (in seconds) from when a vehicle reaches the traffic light system (passes the first detector) and joins the queue (if any) to the time at which it traverses the road's stop line (passes the second detector), inclusive of traversal time (the time it takes for the vehicle to travel through that road without a queue).

- *Delay* - The additional time (in seconds) a driver experiences when traversing in traffic compared to traversing an empty road. Hence, the time from when a vehicle reaches the traffic light system (passes the first detector) and joins the queue (if any) to the time at which it traverses the road's stop line (passes the second detector), not including traversal time.

- *Detectors* - Instruments which receive traffic detection information and convert the information into an impulse that can be analysed.

- *Queue Length* - The number of cars waiting to exit the traffic light system

which are found between the first (entry) and the second detectors (exit) in a particular road.

- *Cycle* - One complete rotation through all of the indications in the traffic light system.
- *Cycle Length* - The time (in seconds) it takes for one complete rotation to happen.
- *Fixed Cycle* - A cycle which has the same cycle length throughout the whole process.
- *Green-Time* - The time during which a given traffic direction receives a green indication, and vehicles are permitted to enter the intersection.
- *Traffic Light Controller* - Someone or something whose job is to alternate service between conflicting traffic movements. This requires assigning more or less green-time to each traffic light in the intersection.
- *Cyclic control* - A form of signal control where traffic movements are served in a repeating, fixed sequence of phases (sequence in which different movements get green) within a signal cycle of a certain length.
- *Acyclic control* - A form of signal control where there is no fixed cycle length and no predetermined phase sequence. Instead, phases are activated dynamically in response to real-time traffic demand.
- *Traffic Congestion* - A traffic condition which is characterized by slower speeds, delays, and increased vehicular queues.
- *Simulation* - An imitative representation of a real world system.
- *Road Network* - A set of intersecting roads and their connections.
- *Input Flow* - The flow of data (cars) that enters a system.
- *Speed Limit* - The maximum or minimum speed allowed.
- *Signal Phase* - A specific interval of time during which a particular set of traffic movements is given the right-of-way (green signal) at an intersection. During this time, vehicles permitted by that phase can move safely, while all conflicting movements receive a red signal.

- *Rush Hour* - An hour in a day when traffic is at its heaviest.
- *Saturation* - The point at which detectors cannot accurately detect and process vehicles due to the maximum measurable road capacity.
- *Throughput* - The number of vehicles exiting the system in a time period.

By considering previous literature, in Section 1.3, one can better comprehend how the traffic light problem was tackled by other researchers.

1.3 History of Traffic Light Control

1.3.1 An Overview

Traffic lights were introduced by the railroad signal engineer J. P. Knight, more than 20 years before the motor car was invented. Their primary use was to control the flow of horse carriages and pedestrians to make road traffic safer. These were being installed at places where traffic from different directions crossed the same road segment, often called intersections or crossings. By giving right of way to traffic in some direction(s), traffic approaching from other directions must wait before proceeding.

After the motor car was invented at the beginning of the twentieth century, traffic signal development and use grew rapidly. At the onset of traffic control systems, in the 1910s, traffic lights consisted of signals displaying 'Stop' and 'Proceed', while being manually operated. At the end of the decade, automated signals were introduced using red (Stop) and green (Proceed) lights, eliminating the need for manual intervention in switching between the two. Some years later, the amber (yellow) light was introduced to the traffic signals in order to warn drivers about the change from green to red or vice versa.

In the early 1950s, traffic light control was being studied from a different perspective. Since then, countless *delay* models and control policies have been developed. Minimising delay, which is accumulated due to waiting at intersections, for car drivers, was the main concern. Intelligent control of traffic lights can keep to

a minimum the overall *waiting time* of cars. In practice, traffic engineers aim to set a good control plan with the help of simulation software, delay formulas and experience. Furthermore, there are other objectives that the controller may have for this problem.

In this study, the aim is to identify policies that help manage traffic efficiently by considering various factors contributing to traffic congestion. To bring this aim to fruition, information is gathered using *detectors* placed at the ends of each road heading towards the intersection. Hence, *queue length* is assumed to be the amount of cars waiting between the first (placed at a certain stretch of road away from the stopping line) and second (placed near the stopping line) detector of the road. Queue length can then be used to extract the respective waiting time of cars.

Traffic light optimization is a complex problem. In the case of single junctions, finding an optimal solution can be challenging. However, in the case of multiple junctions, the problem becomes even more complex. The reason behind this complexity, can be attributed to the state of one traffic light influencing the flow of traffic towards other traffic lights. The constant change in traffic flow creates further complications, especially since traffic accumulation is dependent upon multiple factors of time, such as the time of the day, week, and year. Roadworks and accidents further affect complexity and performance. With the evolution of computer science, artificial intelligence methods namely, fuzzy logic, neural networks, evolutionary algorithms and RL, have been successively applied to traffic light control problems. This section considers past studies related to traffic light control, with importance being given to those utilising MDPs.

Section 1.3.2 provides a literature review of the approach relevant to this study, while Sections 1.3.3 and 1.3.4 review the literature of RL briefly as well as other approaches used to deal with traffic control problems.

1.3.2 Literature review on the Markov Decision Process Approach

The MDP approach was studied by Steingröver et al. (2005) who present a model-based RL approach in order to solve a traffic congestion problem. They take into consideration the system of an isolated junction while representing it as an MDP. Moreover, they include the amount of traffic in neighbouring junctions in the set of states. The approach taken by Steingröver et al. (2005), is expected to optimise the whole network rather than a local network composed of an isolated junction. In addition, the transition probabilities are estimated using a standard maximum likelihood modelling method. Similarly, Haijema & van der Wal (2008) present a novel approach for the dynamic control of a signalized intersection, where the problem of when to switch the lights is modelled as an MDP in discrete time. The aim of their study is to minimize vehicle delay at intersections. The authors start from a near-optimal *fixed cycle* policy and, consequently, a one-step policy improvement, which involves evaluating the current policy and then making a single improvement step to update the policy, is proposed. This gives a near-optimal policy for the problem of their study. Their approach results in less delay, shorter queues, and is shown to be robust for various traffic volumes.

Similarly, Minoarivelo (2009) models a traffic intersection as an MDP, while using the value iteration algorithm to solve the optimisation problem. The author also performs a simulation to perceive the proposed policy. A slightly different approach is that of Brechtel et al. (2011), who present a method for high standard decision making in traffic environments. By making use of probability theory, decisions are automatically derived from the knowledge of human behaviour in cars over time. The latter study constructs a mathematical framework to acquire states from difficult continuous temporal models, which are encoded as Dynamic Bayesian Networks. As a consequence, discrete MDP states are represented by random variables.

Another paper using MDPs to solve traffic related problems is that of Xu et al. (2016). The authors propose a Markov state transition model for an intersection and formulate a traffic signal control problem as an MDP. With the aim of mitigating the computational burden, a sensitivity-based policy iteration algorithm is used to

find an optimal policy. Furthermore, the model can also be extended in order to be applied to a traffic network, rather than an isolated intersection.

1.3.3 Reinforcement Learning Approach

RL, falling under the class of decision-making methods, is part of the machine learning framework. In RL, the decision maker, often referred to as an agent, learns how to act through trial and error by environment interaction. Based on the observations made, the agent chooses certain actions while getting feedback through rewards (or penalties) and changes actions over time in order to get the best possible reward.

One of the first attempts to solve the traffic light control problem using RL, was made by Thorpe and Anderson (1996). They define the state of the environment by the number of vehicles and their respective position in the north, south, east and west roads approaching the intersection. Moreover, the actions consist of enabling either the vehicles from the north-south lanes to pass, or those from the east-west lanes to pass. The desired state is when the number of waiting cars is 0. The authors use a neural network to predict the waiting time of cars near a junction, while controlling the junction using the SARSA (State-Action-Reward, State-Action) algorithm. The latter is referred to as an on-policy RL algorithm, since it updates the Q-values by making use of the actions performed following the current policy. Thorpe and Anderson trained a single *traffic light controller*. The controller was then tested by instantiating it on a grid of 4×4 traffic lights. Even though their study consists of a large state space, with learning time and variance possibly being substantial, the simulations show that the model provides a near optimal performance.

A slightly different approach was taken by Abdulhai et al. (2003). The authors demonstrated that the use of RL, more specifically Q-learning, is advantageous when solving the traffic light control problem. Q-learning is known as an off-policy method, because it updates the Q-values using the best possible action irrespective of the policy being followed. In the said study, the states consist of the duration of each phase and the queue lengths of the roads around the intersection. Furthermore, the action is either "change to the next phase" or "extend the current phase" of the traffic light. When used to control isolated traffic lights, this approach showed a

satisfying performance in Abdulhai et al. (2003).

During the subsequent year, Wiering et al. (2004) used a multi-agent RL to control a system of junctions. Whilst a model-based RL is used in the mentioned study, the system is modelled in its microscopic representation, therefore, considering the behaviour of each vehicle. The authors counted the incidence of every possible transition, and the sum of rewards received, corresponding with each action taken. Note that the actions are whether to choose green or not at each traffic node, while the rewards/costs are 1 if the car stays at the same place and 0 if it moves. The state was inclusive of the road where the car is, the direction of the car, the position of the car in the queue, and its destination. The aim in Wiering et al. (2004) was to minimise the total waiting time of all cars at every intersection, at each time step.

In Abdoos et al. (2011), a relatively large traffic network was modelled as a multi-agent system, while using multi-agent RL techniques. Specifically, Q-learning, where in order to estimate states the average queue length in upcoming links is used. By representing the action space parametrically, the method becomes more flexible and hence can be used for various types of intersections. The author's simulation results showed that the Q-learning approach proposed was better than a fixed time method, even when considering different traffic demands.

Further recent studies, such as Liang et al. (2019) and Han et al. (2022), propose a deep RL model to control the traffic lights. Liang et al. (2019) collect traffic data and split the intersection into smaller grids; hence, they quantify complex traffic scenarios as states. Moreover, the actions correspond to the duration changes of a traffic light, which are modelled as a high-dimensional MDP. The reward is taken to be the cumulative waiting time difference between two cycles. In order to solve the model, the authors applied a convolutional neural network to map states to rewards.

Furthermore, Han et al. (2022) consider the waiting time of pedestrians. They put forward a pedestrian-considered deep RL traffic signal control approach. A technique known as the "Discrete Traffic State Encoding" is applied and enhanced to define the more exhaustive states and rewards. During the learning process of the neural network, the technique referred to as "multi-process operation method" is chosen and several environments are executed to ameliorate the model's learning

efficiency. With the help of a simulation software, the simulations were carried out on real intersection scenarios. When compared to dueling deep Q-networks, the waiting time using the proposed approach decreased by 58.76% and the amount of people waiting also decreased by 51.54%.

1.3.4 Other Approaches

There are several other approaches that can be used in relation to the problem in question. Fuzzy logic is one of such approaches and is a form of multi-valued logic, that originates from fuzzy set theory. This can handle approximate logic instead of precise logic. Tan et al. (1995) were among the first to use fuzzy logic to control a junction. The fuzzy logic controller was built to identify the amount of time the traffic light should remain at a particular phase prior to changing to the next phase. Despite the phase order being pre-determined, the controller may skip a phase, if deemed necessary. The decision is then taken based on the number of vehicle arrivals and number of vehicles waiting, which are represented as fuzzy variables: many, medium and none.

Building on the previous work, Lee et al. (1995) considered the control of multiple intersections using fuzzy logic. The controllers in Lee et al. (1995) receive information from the preceding and adjacent junctions to coordinate the frequency of green light phases of a specific junction. Another application is that of Chiu (1992), where fuzzy logic was used to adjust various parameters, such as the degree of saturation at each intersection and the cycle time.

Jafari et al. (2021) present a new stable TS (Takagi–Sugeno) fuzzy controller for the traffic light problem. The vehicle's average waiting time at an isolated intersection and the queue length are both represented in the state space. To have a light control based on the queue length, the authors designed a fuzzy intelligent controller. Consequently, the stability of the environment was proven using Lyapunov's theorem. The queue length and the number of arrived and departed vehicles from each lane are considered as the input variables.

Artificial neural networks, which were mentioned when delving into past literature, are another approach that can be used for modelling, learning and controlling

traffic lights. In artificial intelligence, a neural network denotes a simplified model inspired by the neural processing within the human brain. Additionally, this approach can be used to complement other methods, such as enhancing the precision of a fuzzy algorithm control procedure.

In Patel and Ranganathan (2001), the traffic light problem was modelled using artificial neural networks. The author's aim was to predict the traffic light's parameters for the subsequent time period and to calculate its duration using this approach. The data collected from the detectors around the junction served as input to the artificial neural network, while the timing for the green and red lights served as the output.

Moreover, some studies combine fuzzy logic and neural networks, in order to model traffic light control. An example would be the work of Henry et al. (1998), who propose a neuro-fuzzy approach that controls the intersection at each second. For each signal, the neuro-fuzzy control selects whether to switch on or off. Although using artificial neural networks has shown to be efficient, such networks lack flexibility in terms of adaptability. Thus, they are not easily generalised, which makes them complex to modify for real-world situations.

Evolutionary algorithms are another flexible approach that can solve non-linear programming problems. Normally, the application of evolutionary algorithms in traffic control is that of managing the time period of traffic light phases. For instance, in Foy et al. (1992), the authors used a genetic algorithm to control four junctions. In Teo et al. (2010), the genetic algorithm is introduced for the optimization of traffic flow control. In addition, the genetic algorithm's main characteristics are that it adjusts its own parameters, such as mutation and crossover rates, and selection methods. This improves the algorithm and allows it to find the optimal solution over successive generations. As input to the genetic algorithm, the authors place the current queue length, whereas the optimized green time for the intersection will be the output. This approach is further improved by the introduction of incoming traffic flow amid red time for each phase.

A different evolutionary algorithm for the traffic signal timing optimisation application was used in Hirulkar et al. (2013). In their study, Hirulkar et al. (2013)

utilized the Particle Swarm Optimization (PSO) algorithm for the traffic signal timing optimization problem. They propose a new PSO algorithm that minimizes the average delay and average number of stops for neighbouring junctions. Moreover, they make use of an actuated traffic control plan to compute the green time for each phase in a cycle. Another approach is that of Jintamuttha et al. (2016), who use the Bat Algorithm (BA). They propose a finite-interval model that finds the effective green time for each of the four phases in each cycle. Their objective is that of finding a solution using the BA to minimize spoilage time, which is the duration for which traffic signals remain active without improving traffic flow, at the intersection.

Out of all the approaches discussed, the choice of using MDPs for such a problem was taken since this probabilistic framework ensures that decisions are not only reactive but also optimised over time. Section 1.4 gives an overview of the chosen approach.

1.4 A General Introduction to MDPs

This section provides a generic outline of certain basic yet important concepts behind MDPs, which are essential to understand the subsequent chapters. Note that some terms are also relevant to the RL framework since MDPs can be a component of such a framework. First and foremost, an MDP or as it is otherwise known, a 'stochastic dynamic programming problem' or a 'discrete-time stochastic control problem', has as its basic object a discrete-time stochastic system whose transition process (moving from one state to another) can be controlled gradually. Each control policy determines the stochastic process and values of the objective function related to this process, with its main goal being of selecting a favourable control policy.

It is also crucial to grasp the concept of the *agent-environment* relationship. An *agent*, or as otherwise known, the decision maker, is the individual or program, in the case of a computerized system, interacting with the *environment* through *actions*. Actions are decisions that have an associated *reward* that is also typically dependent on the current *state* of the environment. Therefore, the environment, real-world or simulated, is what the decision maker will interact with. Figure 1.1 illustrates how the agent interacts with the environment.

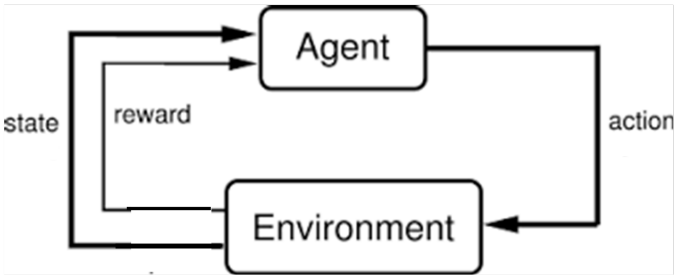

Figure 1.1: The Agent-Environment Relationship (Adapted from 'Reinforcement Learning: An Introduction' (Sutton & Barto, 1999))

Additionally, a state is the position of the decision maker at a specific time-step within the environment, whereas moving from one state to another is referred to as a *transition*. Consequently, a *transition probability* is the probability that the decision maker will move from one state to another. It is important to note that the MDP framework has a flexible mathematical structure when it comes to modelling sequential decision-making problems. For instance, time steps can be fixed time intervals or arbitrary, while states can be room descriptions or sensor readings. Thus, the latter attributes, together with other characteristics, can take different forms.

An MDP models the interaction between an agent and an environment by means of states, actions, transition probabilities, and rewards. This abstraction has proved sufficient since it provides a structured way to model both the environment dynamics and the agent's objectives. Even though real-world problems may involve incomplete information and conflicting aims, this framework is still ideal for analysing and solving sequential decision problems.

In Chapter 2, the author describes in detail the approach chosen in this study, namely that of MDPs. This study focuses on minimising traffic performance indicators, such as vehicle waiting time, at a traffic light intersection. The approach taken is that of achieving dynamic control of a traffic system by modelling as an MDP the traffic light switching. Lastly, Section 1.5, presents the structure of the dissertation to help the reader in navigating through this study.

1.5 Structure of the Dissertation

This dissertation is structured into six different chapters. This present chapter, being the introductory chapter, provided a brief overview of the traffic intersection problem and the approach that will be subsequently employed in an attempt to solve it. Following a brief introduction to the problem, the application terminology is presented, with each term thoroughly defined. Thereafter, the relevant literature encountered during research, which mainly concerns the MDP-related applications, is discussed.

Chapter 2 introduces MDPs and the essential preliminary definitions in the subsequent sections. These preliminary definitions lay the foundation to commence the mathematical formulation of MDPs. This chapter then shifts to the MDP framework and discusses in detail the components of the said framework in their respective sections. These components lay the path to decide the type of objective function to be utilised in the model.

Chapter 3 discusses the fundamental concepts in dynamic programming and reinforcement learning. The value functions, the Bellmann equation, the optimal value functions and the Bellmann optimality equation, are all discussed in this chapter. The policy iteration algorithm, which will eventually be used to find an optimal policy for the problem under study, is then described in detail. Finally, the chapter discusses the convergence of the said algorithm.

Chapter 4 then addresses the problem infrastructure and the model of the application. The various control schemes applied to the problem under study are all considered individually. Moreover, this chapter introduces the test intersection and the manner in which it is modelled through the Aimsun programme. Subsequently, the problem infrastructure is further delved into, and the several factors contributing to it are presented. Lastly, this chapter provides the MDP formulation for the problem under study by defining states, actions, transition probabilities and the reward function.

Chapter 5 is entirely dedicated to computational results obtained using the techniques discussed in the previous two chapters. A description of the performance met-

rics used to measure the efficiency of the traffic intersection is provided, together with an explanation of their significance. Following that, the results of the sensitivity analysis conducted are presented. The core of the chapter, being the comparison between a fixed cycle and an MDP driven cycle, is then thoroughly analysed.

In the concluding chapter of this study, *Chapter 6*, a brief summary of the findings is provided. This chapter also includes the limitations encountered throughout the study and suggestions for further improvements that can be pursued in future research.

Chapter 2

Markov Decision Processes

In this chapter, certain preliminaries which essentially form the foundation of MDPs will be introduced and progressively build toward their formal framework in Section 2.1. In Section 2.2, the general notation of MDPs is then discussed and defined. Furthermore, the specification of state and action spaces, transition probabilities, returns and expected rewards, together with policies are presented. Finally, the possible objective functions are then considered in Section 2.3, namely, the expected total discounted reward and the expected average reward. Altogether, these sections help the reader to go from foundational principles involving MDPs to advanced optimisation criteria, hence, providing a theoretical and practical view for sequential decision problems.

2.1 Preliminaries

In Sections 2.1.1 to 2.1.4, certain essential preliminaries will be introduced, starting off with the notion of time steps, followed by an introduction to the Markov property, then ending with the definition of a Markov state space and characterisation of Markov processes. Such concepts constitute the formal mathematical foundation on which MDPs are defined.

2.1.1 Time Steps

Time steps are points of time where a state is perceived and a decision is made (an action is taken). Let T denote the set of all time steps, which can either be a discrete set or a continuous set. In addition, in the discrete case, the set T is either finite or countably infinite, hence $T = \{1, 2, \dots, N\}$, $N < \infty$, or $T = \{1, 2, \dots\}$. For the continuous case, the set T can either be a bounded interval $T = [0, N]$, $N < \infty$, or an unbounded interval $T = [0, \infty)$.

When considering discrete time problems, time is segregated into periods, where a period is described as the time between one time step and another. Therefore, the beginning of a period corresponds to a time $t \in T$. In essence, when the elements of T are discrete and T is a finite set, the problem will be referred to as a *discrete-time finite-horizon* problem, while when the elements of T are discrete and T is an infinite set it is considered as a *discrete-time infinite-horizon* problem. On the other hand, if the elements of T are continuous and T is a bounded set, it is known as a *continuous-time finite-horizon* problem. Otherwise, if the elements of T are continuous and T is an unbounded set it is known as a *continuous-time infinite-horizon* problem. Section 2.1.4 shows the different classes of Markov chains and Markov processes which differ based on how the time steps are defined in the process.

Before delving into MDPs, Sections 2.1.2 - 2.1.4 introduce the Markov property and Markov processes which are the foundation of MDPs.

2.1.2 The Markov Property

The word "Markov" is used since the transition probability and reward functions (by which the reward is generated) depend only on the past through the current state of the environment and the action selected by the decision maker in that state. What separates a *Markov process* from an MDP is the actions (decisions), where a Markov process does not incorporate decision making. Consequently, actions will be disregarded up until they are reintroduced in Section 2.2.

A stochastic process $\{S_t\}$, where $t \in \mathbb{N}$ denotes a discrete time index and $S_t \in \mathcal{S}_t$

denotes the state at time t in state space $\mathcal{S}_t$, is Markov if and only if:

$$\mathbb{P}[S_{t+1} \in \mathcal{S}_{t+1} | S_t = s_t] = \mathbb{P}[S_{t+1} \in \mathcal{S}_{t+1} | S_1 = s_1, \ldots, S_t = s_t], \quad \text{for } t \in \mathbb{N}, \tag{2.1}$$

meaning that the future is independent of the past given the present. If (2.1) holds, for state S_t and successor state S_{t+1}, the state transition probabilities can be denoted by $p_t(s_{t+1}|s_t)$. If continuous time is considered, equation (2.1) becomes:

$$\mathbb{P}[S_{t_{k+1}} \in \mathcal{S}_{t_{k+1}} | S_{t_k} \in \mathcal{S}_{t_k}] = \mathbb{P}[S_{t_{k+1}} \in \mathcal{S}_{t_{k+1}} | S_{t_1} = s_{t_1}, \ldots, S_{t_k} = s_{t_k}], \tag{2.2}$$

where $0 < t_1 < t_2 < \cdots < t_k < t_{k+1}$, $k \in \mathbb{N}$.

When an environment has this property, its one-step dynamics help predict the subsequent state. Producing a sequence of random outcomes at time steps, which are indexed by t, is referred to as a process. Such process-generated random outcomes may be key metrics of interest, such as valuations of financial derivatives or the value of investor portfolios. To better understand the change of such random outcomes of a process, it is deemed useful to make the internal representation as the focal point of the process at each time t. This is the main reason for motivating the outcomes produced by the process, which is referred to as the state of the process at time t. Before discussing this in detail, a Markov State Space is defined in Section 2.1.3, and the different forms a state space may have are also introduced.

2.1.3 A Markov State Space

In a discrete state space, the set of possible states is countable. In contrast, a continuous state space consists of an uncountably infinite number of states. For instance, a continuous state space is when considering a car cruising on the road, where the state space represents the continuous range of feasible positions and velocities of the car. On the other hand, an example of a discrete state space, is the case of a simple board game, where the state space corresponds to the finite number of positions of the pieces on the board. Each cell on the board corresponds to a unique state, while the board represents the space.

Additionally, the nature of the state space can vary depending on the specific application and system being modelled. Consequently, a Markov state space can be

discrete or continuous amongst other types, which are not that common, namely a mixture of both discrete and continuous, structured and partially observable state spaces. For partially observable state spaces, the decision maker may not always have complete access to information on the underlying state; thus, in these cases the decision maker receives partial information through observations. Structured state spaces are used in scenarios where states have a specific organisational structure, composed of different dimensions. Moreover, multi-dimensional state spaces are useful when problems involve a huge amount of information or require a great extent of detail.

Apart from $\mathcal{S}_t$ being able to take different forms as seen above, the set T defined in Section 2.1.1 can also take different forms. The different Markov Processes representing the latter variations will be observed in Section 2.1.4.

2.1.4 Markov Processes

Markov processes are typically classified by the type of state space and time parameter being considered. For example, a Markov chain is one type of Markov process where its state space is discrete. In general, a Markov process is a memoryless random process, i.e., a chain of random states $S_1, S_2, \ldots$ having the Markov property. A Markov process is a tuple $\langle(\mathcal{S}_t)_{t\in T}, (p_t)_{t\in T}\rangle$, where:

- T is the set of all time steps t;
- $\mathcal{S}_t$ is a set of possible states at time $t \in T$;
- $p_t(s_t, s_{t+1})$ are state transition probabilities for time $t \in T$, defined as:

$$p_t = \mathbb{P}[S_{t+1} = s_{t+1} | S_t = s_t] = P_{s_t, s_{t+1}}.$$

Markov processes can be classified into discrete time Markov processes and continuous time Markov processes. Moreover, a Markov process normally has a discrete state space or a continuous state space (apart from the other possibilities mentioned in Section 2.1.3). In general, Markov processes are classified in the following manner:

- A discrete time Markov Process with a discrete state space is referred to as *a discrete time Markov Chain*.
- A discrete time Markov Process with a continuous state space is referred to as *a discrete time Markov Process*.
- A continuous time Markov Process with a discrete state space is referred to as *a continuous time Markov Chain*.
- A continuous time Markov Process with a continuous state space is referred to as *a continuous time Markov Process*.

It is important to note that when dealing with continuous time, the transition probabilities are derived differently (using a transition rate matrix) than when dealing with discrete time (using a transition matrix). Firstly, in a continuous time setting, the rate of transitioning from one state to the next is defined through a transition rate matrix. As a result, the transition probabilities are derived by using the matrix exponential of the transition rate matrix.

There are other types or extensions of Markov processes, each with unique characteristics and properties. These are: a Hidden Markov Model (HMM), a Semi-Markov Process, a Markov Jump Process and an MDP. The latter variant is discussed in great detail in Section 2.2. In conclusion, an MDP is an extension of a Markov process that includes an agent/decision maker that makes decisions that affect the progression of the environment over time while also incorporating rewards. This enables the author to model sequential decision making problems. Throughout this study, a discrete time Markov chain is suitable for the application considered.

2.2 MDP Framework

An MDP model mainly comprises of five elements: time steps, states, actions, transition probabilities, and rewards. These elements and other important characteristics of an MDP are discussed in further detail in Sections 2.2.1 - 2.2.5. It is important to note that the theory focuses on a finite horizon, discrete time MDP with discrete state and action spaces.

2.2.1 Definition of an MDP

An MDP is a 4-tuple $\langle (S_t)_{t\in T}, (A_{s_t,t})_{s_t\in S_t, t\in T}, (p_t(r_t, s_{t+1}|s_t, a_t))_{t\in T}, (R_t(s_{t+1}|s_t, a_t))_{t\in T}\rangle$, with each component being discussed in detail in Sections 2.2.2 and 2.2.3, where:

- $S_t \in \mathcal{S}_t$ is a random state at time t,
- $A_{s_t,t} \in \mathcal{A}_{s_t,t}$ is a random action in state s_t at time t,
- $p_t(r_t, s_{t+1}|s_t, a_t)$ is the probability at time t of transitioning from state s_t by taking action a_t and ending up in state s_{t+1} with reward r_t, which is defined as

  $$p_t(r_t, s_{t+1}|s_t, a_t) := \mathbb{P}[R_t = r_t, S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t]. \tag{2.3}$$

  This joint formulation is adopted for generality, allowing rewards to be stochastic and jointly generated with state transitions; the standard MDP formulation is recovered as a special case.

- $R_t(s_t, a_t, s_{t+1})$ is the reward function at time t of ending in s_{t+1} given that action a_t is taken when in state s_t, which is defined as

  $$R_t(s_t, a_t, s_{t+1}) = \mathbb{E}[R_t|S_t = s_t, A_t = a_t, S_{t+1} = s_{t+1}].$$

As can be observed, an MDP is an extension of a Markov process, inclusive of decision-making and control. Even though both models are based on the Markov property and state transitions, MDPs provide a framework for modelling situations where a decision maker's actions affect the state transitions and rewards. Markov processes focus entirely on the probabilistic evolution of an environment, however, MDPs extend this to include a decision-making process, allowing for optimal control and planning in uncertain environments. As a result, more properties that formulate and distinguish an MDP from a Markov process are explored from Section 2.2.2 onward.

2.2.2 State and Action Sets

Actions are the moves taken by the decision maker within the environment, while a state is a situation returned by the environment after each action taken by the decision maker. If S_t were to be time independent, then $S_t \equiv S$, $\forall t \in T$, where S would be the time independent state random variable. The decision maker, at time $t \in T$ and state $S_t = s_t$, chooses action $A_{s_t,t} \in \mathcal{A}_{s_t,t}$, where $\mathcal{A}_{s_t,t}$ is the set of the actions allowed in state s_t at time t, referred to as the action space in state s_t at time t.

The variations of a state space were already discussed in Section 2.1.3; hence, only the variations of an action space are discussed in detail in this section. An action space can be either discrete (finite or countably infinite) or continuous. Apart from discrete or continuous action spaces, there exist action spaces which are structured, hybrid or high-dimensional. Structured spaces are used in scenarios where actions have a specific organisational structure, composed of different dimensions. In some MDPs, both discrete and continuous action spaces may coexist; these scenarios are referred to as hybrid. Moreover, high-dimensional spaces are useful when problems involve a huge amount of information or require a great extent of detail.

In the case of a discrete action space, the decision maker takes decisions about which action to perform from a finite or countably infinite action set. For instance, in a simple grid navigation problem, the decision maker's action set may include actions similar to "go up", "go down", "go left", and "go right". Since there are four alternative actions in this case, the action set is referred to as discrete.

On the other hand, a continuous action space comprises real-valued actions within a certain range, thus allowing for a wide range of actions. These are normally encountered in tasks that require precise and smooth control, namely robotic manipulation and autonomous vehicle navigation amongst others. Solving problems with continuous action spaces can be more challenging than having a discrete action space because of the potentially infinite number of possible actions. This makes the action selection process more complex, such that it requires specialised algorithms to handle the continuous nature of the space. For example, in a robotic arm control problem, the continuous action space can correspond to the different angles the

robot can apply to move its arm easily.

When an action is taken by the decision maker at time t, the environment reacts, while time is ticking over to $t + 1$, by producing the next time step's state and generating a reward based on that transition. Therefore, transition probabilities and rewards, delved into in Section 2.2.3, are what follows after discussing states and actions.

2.2.3 Transition Probabilities and Rewards

Transition probabilities describe the dynamics of the environment. They play a similar role to the next state function in a problem solving search. The only difference is that every state is considered to be a possible outcome of taking a particular action in a given state. Hence, for each state $s_t \in \mathcal{S}_t$ and action $a_t \in \mathcal{A}_{s_t,t}$, the probability that the next state will be $s_{t+1} \in \mathcal{S}_{t+1}$ is determined. This can be viewed as being represented as a set of matrices, a matrix for each action indexed in both dimensions by states and a square matrix only if $\mathcal{S}_t$ and $\mathcal{S}_{t+1}$ have the same size. For example, consider a grid world of safe and dangerous states, with actions being to stay still or to move, the probabilities for each action differ. Hence, the problem would have two 2×2 transition probability matrices, one for each action, representing all the different possible state-action pairs.

Time is assumed to be discrete in order for transition probabilities and rewards theory to be introduced with respect to this study's application. For each time step $t \in T$, a finite-horizon MDP consists of a finite set of states $\mathcal{S}_t$, a finite set of actions $\mathcal{A}_{s_t,t}$, $\forall s_t \in \mathcal{S}_t$, transition probabilities $p_t(r_t, s_{t+1}|s_t, a_t)$, rewards R_t and a finite time horizon. At each time step, the decision maker interacts with the environment by choosing an action, transitioning to a new state, and receiving a reward. Additionally, it is assumed that reward R_t is received instantly at the beginning of time step t. As a consequence, the environment and agent produce a sequence that begins as follows:

$$S_1 = s_1, A_1 = a_1 \in \mathcal{A}_{s_1,1}, R_1 = r_1, S_2 = s_2, A_2 = a_2 \in \mathcal{A}_{s_2,2}, R_2 = r_2, \ldots.$$

Transitioning to a new state and receiving a certain reward from one time step to the next requires transition probabilities. This is defined in expression (2.3).

It is important to note that if R_t is independent of the future state S_{t+1}, then the R_t's in equation (2.3) can be discarded. In this work, rewards are defined on state-action-next state triples and are assigned after the transition is realised. However, a generalised definition was provided due to the theoretical nature of this chapter. The | in the expression is taken from conditional probability notation. It recalls that p_t specifies a probability distribution for each s_t and a_t chosen, which results in:

$$\sum_{s_{t+1}\in\mathcal{S}_{t+1}} \sum_{r_t\in\mathcal{R}_t} p_t(r_t, s_{t+1}|s_t, a_t) = 1, \quad \forall s_t \in \mathcal{S}_t, a_t \in \mathcal{A}_{s_t,t}. \tag{2.4}$$

Here, $\mathcal{R}_t \subseteq \mathbb{R}$ denotes the set of all possible reward values at time t. Any information about the environment can be computed using equation (2.3). For instance, the *state-transition probabilities*,

$$\bar{p}_t(s_{t+1}|s_t, a_t) := \mathbb{P}[S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t] = \sum_{r_t\in\mathcal{R}_t} p_t(r_t, s_{t+1}|s_t, a_t). \tag{2.5}$$

The accumulation of the above-discussed rewards, referred to as returns, will be delved into in Section 2.2.4.

2.2.4 Returns and Expected Rewards

One possible objective of an MDP is to find a policy that maximises the expected cumulative reward over the given time horizon. The accumulation of rewards R_t, for time steps t up to T, is denoted by the return:

$$G_t = R_t + R_{t+1} + R_{t+2} + \cdots + R_T, \tag{2.6}$$

where $t \in \{1, 2, \ldots, T\}$. As a result, the return is the sum of rewards received from time step t up until the final time step is attained. Due to the fact that the sum in equation (2.6) may not converge to a finite value, this formulation is not suited for infinite time horizon problems.

In the case of an infinite horizon MDP, equation (2.6) is altered accordingly. This is achieved by including a discount factor; thus, the decision maker will deal with a sum of discounted rewards instead. The discount factor determines the present value of future rewards, thus enabling the decision maker to model scenarios where

a future reward is not as attractive as a current reward of the same quantity. The discounted return is denoted by:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \cdots + \gamma^{T-t} R_T = \sum_{k=0}^{T-t} \gamma^k R_{t+k}, \tag{2.7}$$

where $t \in \{1, 2, \ldots, T\}$. The discount factor γ can take values in the range $0 < \gamma < 1$. The closer γ is to 0, the less importance the decision maker is giving to future rewards. Conversely, as γ approaches to 1, the more important future rewards are to the decision maker, hence the decision maker becomes more farsighted. If $\gamma = 1$, then the return could diverge. By ensuring $\gamma < 1$, the larger the exponent of γ becomes, the lower the effect of the future rewards will be. Consequently, equation (2.7) converges to a finite value.

In relation to rewards, equation (2.3) can be used to compute expected rewards for state-action pairs and for state-action next-state triples. The former, which depends on the current state and action, is computed as follows:

$$\begin{aligned} \tilde{r}_t(s_t, a_t) &= \mathbb{E}[R_t | S_t = s_t, A_t = a_t] \\ &= \sum_{r_t \in \mathcal{R}_t} r_t \sum_{s_{t+1} \in \mathcal{S}_{t+1}} \mathbb{P}\big(R_t = r_t,\, S_{t+1} = s_{t+1} \mid S_t = s_t, A_t = a_t\big) \\ &= \sum_{r_t \in \mathcal{R}_t} r_t \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1} | s_t, a_t), \end{aligned} \tag{2.8}$$

while the latter, which depends on the next state apart from the current and next state, is given as follows:

$$\begin{aligned} \tilde{\tilde{r}}_t(s_t, a_t, s_{t+1}) &= \mathbb{E}[R_t | S_t = s_t, A_t = a_t, S_{t+1} = s_{t+1}] \\ &= \frac{\sum_{r_t \in \mathcal{R}_t} r_t p_t(r_t, s_{t+1} | s_t, a_t)}{\bar{p}_t(s_{t+1} | s_t, a_t)}. \end{aligned} \tag{2.9}$$

The expectation is used in these equations due to the fact that transitions are uncertain, and that the decision maker needs a single deterministic numerical estimate of how good a state-action pair, or state-action next state triple, is. In Sections 2.3.1 to 2.3.2, the different formulations the objective function can take, depending on the time horizon and one's approach, are discussed. Based on what has been discussed in Sections 2.2.1 to 2.2.4, the main goal is to find a policy that is optimal with respect to the selected objective function. The definition of a policy will be provided in Section 2.2.5.

2.2.5 Policies

In an MDP, an assumption is that you could potentially go from any state to any other state in one step. Therefore, in order to be prepared for action selection, it is necessary to compute a policy, which is a sequence of mappings from states to actions. As shall be seen, no matter what state one happens to find himself in, a policy points out the action to take.

A policy defines the behaviour of the decision maker in making decisions at each time step $t \in T$, where it is assumed that time is discrete. It can be described as a sequence of mappings that specify which action to take in each state of the MDP at different time steps. Therefore, a policy, or strategy, π is a mapping from states to probability distributions over actions, where the probability of selecting action $a_t \in A_{s_t,t}$ at a particular state s_t at time t is defined as:

$$\pi(a_t|s_t) = \mathbb{P}[A_t = a_t|S_t = s_t]. \quad \forall t \in T, s_t \in \mathcal{S}_t, a_t \in \mathcal{A}_{s_t,t}$$

A policy is said to be stationary if the action probabilities do not depend explicitly on time, that is, $\pi(a_t|s_t) = \pi(a|s)$ for all $t \in T$, and non-stationary otherwise. The policy must satisfy the normalisation condition:

$$\sum_{a_t \in \mathcal{A}_{s_t,t}} \pi(a_t|s_t) = 1, \quad \forall s_t \in \mathcal{S}_t.$$

Moreover, the set of all policies will be denoted by Π.

A policy in which the action probability distribution for each state is concentrated on a single action is referred to as a *deterministic policy*. This satisfies the property that:

$$\pi(a_t|s_t) = \begin{cases} 1, & \text{if } a_t = a'_t, a'_t \in \mathcal{A}_{s_t,t} \\ 0. & \text{if } a_t \neq a'_t, \forall a_t \in \mathcal{A}_{s_t,t} \end{cases}$$

As per Puterman (1994), Proposition 1 states that an optimal deterministic policy exists under certain conditions, where a Markovian policy is a policy where the action at any given time t depends only on the current state of the system and not on the past states or actions. Although policies are defined in general as random mappings, the following result shows that, under the assumptions considered in this work, there always exists an optimal policy that is deterministic and Markovian.

Consequently, restricting attention to this class of policies does not result in any loss of optimality. In Proposition 1, v^* denotes the optimal value function, which will be defined in Chapter 3.

Proposition 1. *Assume $\mathcal{S}_t$ is finite or countable, and that $\mathcal{A}_{s_t,t}$ is finite for each $s_t \in \mathcal{S}_t$. Then, there exists a deterministic Markovian policy that is optimal, $\pi^* \in \Pi$.*

Proof. By definition of v^*, for each $s_t \in \mathcal{S}_t$, there exists $a'_t \in \mathcal{A}_{s_t,t}$, for which:

$$\tilde{r}_t(s_t, a'_t) + \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(s_{t+1}|s_t, a'_t) v^*(s_{t+1}) = \sup_{a_t \in \mathcal{A}_{s_t,t}} \left\{ \tilde{r}_t(s_t, a_t) + \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(s_{t+1}|s_t, a_t) v^*(s_{t+1}) \right\}.$$

Since the action set $\mathcal{A}_{s_t,t}$ is finite, the supremum is attained. Define a policy π^* by selecting action a'_t with probability one in state s_t. This policy is deterministic and Markovian, and achieves v^*. □

This result is important because it guarantees that the search for optimal policies can be restricted to deterministic Markovian policies without loss of optimality, which significantly simplifies both theoretical analysis and algorithmic implementation. In Section 2.3, the different formulations the objective function can take, depending on the time horizon and one's approach, are discussed.

2.3 Objective Functions of an MDP

As per the literature, the objective function may take one of two main forms, which are the expected total discounted reward and the expected average reward per transition. The former criterion can be considered with discount factor 1, which is a special case that results in the expected total reward. Due to the nature of the application at hand, rewards for a finite horizon problem are going to be considered in this section.

2.3.1 The Expected Total Discounted Reward

Consider a finite horizon problem where the aim is to maximise the expected total discounted reward (TDR) over a period of length T. Therefore, having the discount factor satisfying $0 < \gamma < 1$, the following needs to be maximised:

$$\rho_{\pi}^{TDR}(s_1) = \mathbb{E}_{\pi}\left[\sum_{t=1}^{T} \gamma^{t-1} R_t \middle| S_1 = s_1\right]. \tag{2.10}$$

In equation (2.10), π is the implemented policy, $s_1 \in \mathcal{S}_1$ is the initial state of the MDP, while the remaining components were defined in Section 2.2.4. Equation (2.10) defines the expected total discounted reward obtained by following policy π from initial state s_1 over a finite horizon T. The discount factor γ controls the relative importance of immediate versus future rewards, and the expectation is taken over all state–action trajectories induced by the policy and system dynamics. When the goal is defined as an expected total discounted reward, it follows naturally to accumulate as much reward as possible in the initial phases. This is due to the fact that the weights of rewards diminish gradually over time. One of the advantages of using this criterion for an infinite horizon setting, is that it avoids dilemmas such as infinite returns and sums, with the help of the discount factor.

When $\gamma = 1$, an *expected total reward* (TR) needs to be dealt with, rather than a discounted one. A decision maker may choose such a value for γ when they consider all rewards within the horizon to be equally important, and that there is no modelling need to prioritise immediate rewards over future ones. The expression to be maximised becomes:

$$\rho_{\pi}^{TR}(s_1) = \mathbb{E}_{\pi}\left[\sum_{t=1}^{T} R_t \middle| S_1 = s_1\right]. \tag{2.11}$$

The expectation in (2.11) is taken over all state–action trajectories generated by following policy π from the initial state s_1, with respect to the underlying transition dynamics of the system. Equivalently, it can be interpreted as the probability-weighted average of the total reward accumulated over the horizon T, and corresponds to the finite horizon value function evaluated at s_1.

2.3.2 The Expected Average Reward

The decision maker is not always interested in finding a policy that maximises the expected accumulation of rewards. When decisions are made frequently, the decision maker may prefer to compare policies based on their expected average reward instead. Thus, seeking policies that yield the highest expected payoff per step. As a consequence, the expected average reward criterion is exploited in queuing control theory. In such applications, the decision maker makes frequent decisions and evaluates system performance based on, for example, the average time a job or package remains in the system.

The expected average reward (AvR) starting from state $s_1 \in \mathcal{S}_1$ and following policy π is denoted by $\rho_\pi^{AvR}(s_1)$ and is defined as:

$$\rho_\pi^{AvR}(s_1) = \mathbb{E}_\pi \left[\frac{1}{T} \sum_{t=1}^{T} R_t \middle| S_1 = s_1 \right]. \tag{2.12}$$

Additionally, when the supremum exists, the goal is expressed as:

$$\rho^{AvR}(s_1) = \sup_\pi \rho_\pi^{AvR}(s_1). \tag{2.13}$$

Expression (2.13) defines the optimal average reward as the supremum over all admissible policies, while equation (2.15) shows that this supremum is in fact attained by a Markovian stationary policy, allowing the optimisation problem to be solved within a restricted policy class without loss of optimality. An important term pertaining to policy π in this case is the *bias*, which is the quantity that measures the total deviation of rewards from the asymptotic expected average reward. Blackwell (1962) introduced the bias-optimality notion, while also developing the framework for average reward MDPs. More specifically, he portrayed how the gain, which is the long-run expected reward per time step obtained when the policy is followed indefinitely, and bias terms are connected to the expected total discounted reward, discussed in Section 2.3.1, by using a Laurent series expansion. For initial state $s_1 \in \mathcal{S}_1$ and policy π, the bias value can be defined as:

$$b_\pi(s_1) = \lim_{T \to \infty} \mathbb{E}_\pi \left[\sum_{t=1}^{T} (R_t - \rho_\pi^{AvR}(s_1)) \middle| S_1 = s_1 \right]. \tag{2.14}$$

2.3.3 Optimal Policies and the Supremum Criterion

For any objective function (TDR, TR, AvR), a policy which achieves the supremum is known as a *gain optimal* policy π^*. This is defined as $\rho^{\circ}_{\pi^*}(s_1) \geq \rho^{\circ}_{\pi}(s_1)$ over all policies $\pi \in \Pi$ and states $s_1 \in \mathcal{S}_1$. Note that $\circ$ can be any of TDR, TR or AvR. As per Puterman's (1994) results, the optimal gain for any criterion mentioned, can be obtained by searching the space of Markovian stationary policies in the following manner:

$$\rho^{\circ}(s_1) = \max_{\pi:\text{Markovian stationary}} \rho^{\circ}_{\pi}(s_1), \qquad \forall s_1 \in \mathcal{S}_1. \tag{2.15}$$

All metrics discussed in Section 2.3 are essential for assessing policy quality under various conditions. Having determined these concepts, Chapter 3 will delve into the foundational theory and methods for evaluating and solving an MDP.

Chapter 3

Evaluating and Solving an MDP

This chapter introduces some important concepts and mathematical approaches which are necessary for solving MDPs. Value functions are primarily discussed, followed by the Bellman equation, which, as will be observed, is a recursive formulation that allows value functions to be computed by breaking them into simpler sub-problems. Subsequently, optimal value functions, which present the best possible expected rewards, and the Bellman optimality equation are discussed. Following this, the policy iteration algorithm, which is used in this study to find an optimal policy, is introduced together with its convergence properties and an illustrative example.

3.1 Value Functions

In an effort to capture the benefit attained from taking an action and reaching a specific state, *value functions* are defined. Value functions can adopt different forms. Essentially, they are functions which take states as an input, and then assess how good it is for the decision maker to be in a given state. This definition can be extended for value functions having state-action pairs as input, which calculate how effective a certain action is in a particular state. Moreover, value functions can change depending on the policy applied.

As a result, the *state-value function* for policy π, denoted v_π, gives the expected

return when beginning in state $s_t \in \mathcal{S}_t$ and following policy π. Formally, for an MDP, this is defined as:

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t|S_t = s_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t} \gamma^k R_{t+k} \middle| S_t = s_t\right], \ \forall s_t \in \mathcal{S}_t. \tag{3.1}$$

Note that, here G_t is taken as in expression (2.7), but it can also be taken as in expressions (2.11) and (2.12). If G_t were to take the form as in expressions (2.11) and (2.12), the process of obtaining the respective state-value function is similar to that in (3.1).

Another form a value function can take is that of an *action-value function*. An action-value function, under a policy π, takes an input state $s_t \in \mathcal{S}_t$ and action $a_t \in \mathcal{A}_{s_t,t}$, and is denoted by $q_\pi(s_t, a_t)$. It gives the expected return when taking action a_t in state s_t and following policy π thereafter:

$$\begin{aligned} q_\pi(s_t, a_t) &= \mathbb{E}_\pi[G_t|S_t = s_t, A_t = a_t] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{T-t} \gamma^k R_{t+k} \middle| S_t = s_t, A_t = a_t\right], \ \forall s_t \in \mathcal{S}_t, a_t \in \mathcal{A}_{s_t,t}. \end{aligned} \tag{3.2}$$

The mentioned functions can be estimated through the use of *Monte Carlo methods*. This involves averaging over numerous random samples of actual returns. In order to overcome the challenge that numerous states bring, that of storing all the averages for each state, the functions v_π and q_π may also be parametrised. Through parameter modification, the functions should match with the observed returns. Although this can be a viable method, it depends on the accuracy of the parametrised equations. Since the Bellman equations and dynamic programming will be used to compute the value or action functions exactly, there will be no need for approximations for the application, hence, this is not explored in further detail. This is analysed further in Sutton & Barto (2018), where the authors give a complete way on how to deal with such approximations.

Having introduced value functions, the Bellman equation is discussed in Section 3.2. Such an equation effectively captures the interplay between a state's value and that of its successor states.

3.2 The Bellman Equation

This section portrays how the Bellman equation paves the way to determine $v_\pi(s_t)$ and $q_\pi(s_t, a_t)$. This equation must be satisfied for any state $s_t \in \mathcal{S}_t$ and policy π, and outlines the relationship between the value of state $s_t \in \mathcal{S}_t$ and that of $s_{t+1} \in \mathcal{S}_{t+1}$, which can be any state reachable in the next time step. Thus, by starting with the definition of the value function, the derivation of the Bellman equation is achieved as follows:

$$\begin{aligned} v_\pi(s_t) &= \mathbb{E}_\pi[G_t|S_t = s_t] \\ &= \mathbb{E}_\pi[R_t|S_t = s_t] + \gamma\mathbb{E}_\pi[G_{t+1}|S_t = s_t], \end{aligned}$$

where

$$\begin{aligned} \mathbb{E}_\pi[R_t|S_t = s_t] &= \sum_{a_t \in \mathcal{A}_{s_t,t}} \mathbb{E}_\pi[R_t|S_t = s_t, A_t = a_t]\mathbb{P}[A_t = a_t|S_t = s_t] \\ &= \sum_{a_t \in \mathcal{A}_{s_t,t}} \tilde{r}_t(s_t, a_t)\pi_t(a_t|s_t) \\ &= \sum_{a_t \in \mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{r_t \in \mathcal{R}_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} r_t \; p_t(r_t, s_{t+1}|s_t, a_t), \end{aligned}$$

and

$$\begin{aligned}
\mathbb{E}_\pi[G_{t+1}|s_t] &= \sum_{g_{t+1}\in\mathcal{G}_{t+1}} g_{t+1}\ \mathbb{P}(G_{t+1}=g_{t+1}|s_t) \\
&= \sum_{g_{t+1}\in\mathcal{G}_{t+1}} \sum_{a_t\in\mathcal{A}_{s_t,t}} g_{t+1}\ \mathbb{P}(g_{t+1}|s_t,a_t)\mathbb{P}(a_t|s_t) \\
&= \sum_{a_t\in\mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{g_{t+1}\in\mathcal{G}_{t+1}} g_{t+1}\ \mathbb{P}(g_{t+1}|s_t,a_t) \\
&= \sum_{a_t\in\mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{g_{t+1}\in\mathcal{G}_{t+1}} \sum_{s_{t+1}\in\mathcal{S}_{t+1}} g_{t+1}\ \mathbb{P}(g_{t+1}|s_t,a_t,s_{t+1})\mathbb{P}(s_{t+1}|s_t,a_t) \\
&= \sum_{a_t\in\mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{s_{t+1}\in\mathcal{S}_{t+1}} \mathbb{P}(s_{t+1}|s_t,a_t) \sum_{g_{t+1}\in\mathcal{G}_{t+1}} g_{t+1}\ \mathbb{P}(g_{t+1}|s_t,a_t,s_{t+1}) \\
&= \sum_{a_t\in\mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{s_{t+1}\in\mathcal{S}_{t+1}} \mathbb{P}(s_{t+1}|s_t,a_t) \\
&\qquad \cdot \sum_{g_{t+1}\in\mathcal{G}_{t+1}} \sum_{r_t\in\mathcal{R}_t} g_{t+1}\ \mathbb{P}(g_{t+1}|s_t,a_t,s_{t+1},r_t)\mathbb{P}(r_t|s_t,a_t) \\
&= \sum_{a_t\in\mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{r_t\in\mathcal{R}_t} \sum_{s_{t+1}\in\mathcal{S}_{t+1}} \mathbb{P}(r_t|s_t,a_t,s_{t+1})\mathbb{P}(s_{t+1}|s_t,a_t) \\
&\qquad \cdot \sum_{g_{t+1}\in\mathcal{G}_{t+1}} g_{t+1}\ \mathbb{P}(g_{t+1}|s_t,a_t,s_{t+1},r_t) \\
&= \sum_{a_t\in\mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{r_t\in\mathcal{R}_t} \sum_{s_{t+1}\in\mathcal{S}_{t+1}} p_t(r_t,s_{t+1}|s_t,a_t) \\
&\qquad \cdot \sum_{g_{t+1}\in\mathcal{G}_{t+1}} g_{t+1}\ \underbrace{\mathbb{P}(g_{t+1}|s_t,a_t,s_{t+1},r_t))}_{\mathbb{E}_\pi[G_{t+1}|s_t,a_t,s_{t+1},r_t]}.
\end{aligned}$$

Here $\mathcal{G}_t$, under some of the summations, denotes the set of all possible return values at time t. Now,

$$\mathbb{E}_\pi[G_{t+1}|s_t,a_t,s_{t+1},r_t] = \mathbb{E}_\pi[R_{t+1}+\gamma G_{t+2}|s_t,a_t,s_{t+1},r_t].$$

Using the Markov property, hence the fact that $R_{t+1}+\gamma G_{t+2}$ depends only on S_{t+1} and A_{t+1}, the expression can be simplified as follows:

$$\begin{aligned}
\mathbb{E}_\pi[G_{t+1}|s_t,a_t,s_{t+1},r_t] &= \mathbb{E}_\pi[G_{t+1}|s_{t+1}] \\
&= v_\pi(s_{t+1}).
\end{aligned}$$

As a result:

$$\begin{aligned} v_\pi(s_t) &= \mathbb{E}_\pi[R_t|s_t] + \gamma \mathbb{E}_\pi[G_{t+1}|s_t] \\ &= \sum_{a_t \in \mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{r_t \in \mathcal{R}_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} r_t \; p_t(r_t, s_{t+1}|s_t, a_t) \\ &\qquad + \gamma \sum_{a_t \in \mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{r_t \in \mathcal{R}_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1}|s_t, a_t) v_\pi(s_{t+1}) \\ &= \sum_{a_t \in \mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) \sum_{r_t \in \mathcal{R}_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1}|s_t, a_t)[r_t + \gamma v_\pi(s_{t+1})]. \end{aligned} \tag{3.3}$$

Here, G_t can also be like in equations (2.11) and (2.12). Hence, rather than an expected total discounted reward, one can consider an expected total reward or an expected average reward. Additionally, due to the expression:

$$v_\pi(s_t) = \sum_{a_t \in \mathcal{A}_{s_t,t}} \pi_t(a_t|s_t) q_\pi(s_t, a_t),$$

$q_\pi(s_t, a_t)$ is derived from (3.3), which results in:

$$q_\pi(s_t, a_t) = \sum_{r_t \in \mathcal{R}_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1}|s_t, a_t) \left[r_t + \gamma \left(\sum_{a_{t+1} \in \mathcal{A}_{s_{t+1},t+1}} \pi_{t+1}(a_t|s_t) q_\pi(s_{t+1}) \right) \right]. \tag{3.4}$$

Expression (3.3) is referred to as the *Bellman equation*, which shows the relation between a state value, on the left hand side, and the value of the succeeding states, on the right hand side. Such an expression weighs each state by the probability of being in such a state. Moreover, (3.4) is also a Bellman equation, but one that shows the relation among an action value and the value of the succeeding actions. The state value function v_π and action value function q_π, are the only solutions to their respective Bellman equation, as will be illustrated in Section 3.5.

3.3 Optimal Value Functions

In the context of MDPs, the main concern is to establish a policy π^* that maximises the reward earned in the long run. Therefore, a policy π^* for which

$$v_{\pi^*}(s_t) \geq v_\pi(s_t), \quad \forall s_t \in \mathcal{S}_t, \pi \in \Pi. \tag{3.5}$$

Such a policy is referred to as an optimal policy. There exist models where an optimal policy is not necessary; instead, an ε-optimal policy is sought. An ε-optimal policy,

for an $\varepsilon > 0$, is a policy π^*_ε with the property

$$v_{\pi^*_\varepsilon}(s_t) + \varepsilon > v_\pi(s_t), \quad \forall s_t \in \mathcal{S}_t, \quad \pi \in \Pi. \tag{3.6}$$

This being said, all optimal policies result in the same optimal state-value function, meaning that any optimal policy must achieve the maximum possible expected return from every state. Therefore, if two policies are both optimal, they must attain the same value for each state, even if they may prescribe different actions. The resulting function is unique and is denoted by v^*, where:

$$v^*(s_t) = \max_\pi v_\pi(s_t), \qquad \forall s_t \in \mathcal{S}_t, t \in \{1, 2, \ldots, T\}. \tag{3.7}$$

Furthermore, they share the optimal action-value function q^*, which can also be expressed in terms of v^* in the following way:

$$\begin{aligned} q^*(s_t, a_t) &= \max_\pi q_\pi(s_t, a_t) \\ &= \max_\pi \mathbb{E}[R_t + \gamma v_\pi(S_{t+1}) | S_t = s_t, A_t = a_t] \\ &= \mathbb{E}[R_t + \gamma \max_\pi v_\pi(S_{t+1}) | S_t = s_t, A_t = a_t] \\ &= \mathbb{E}[R_t + \gamma v^*(S_{t+1}) | S_t = s_t, A_t = a_t], \quad \forall s_t \in \mathcal{S}_t, a_t \in \mathcal{A}_{s_t,t}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad t \in \{1, 2, \ldots, T\}. \end{aligned} \tag{3.8}$$

3.4 The Bellman Optimality Equation

The value function v^* needs to satisfy the consistency condition imposed by the Bellman equation (3.3), which is that the value of a state must equal the best possible expected return from that state. Given that v^* corresponds to an optimal policy, such a condition can be rewritten without referring to other policies. Hence, the *Bellman optimality equation* is derived in Equation (3.9), for an expected total discounted reward G_t. Note that G_t can also be either an expected total reward or

an expected average reward.

$$\begin{aligned} v^*(s_t) &= \max_{a_t \in \mathcal{A}_{s_t,t}} q_{\pi^*}(s_t, a_t) \\ &= \max_{a_t \in \mathcal{A}_{s_t,t}} \mathbb{E}_{\pi^*}[G_t | S_t = s_t, A_t = a_t] \\ &= \max_{a_t \in \mathcal{A}_{s_t,t}} \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{T} \gamma^k R_{t+k} \middle| S_t = s_t, A_t = a_t\right] \\ &= \max_{a_t \in \mathcal{A}_{s_t,t}} \mathbb{E}_{\pi^*}\left[R_t + \gamma \sum_{k=0}^{T} \gamma^k R_{t+k+1} \middle| S_t = s_t, A_t = a_t\right] \\ &= \max_{a_t \in \mathcal{A}_{s_t,t}} \mathbb{E}_{\pi^*}\left[R_t + \gamma v^*(S_{t+1}) | S_t = s_t, A_t = a_t\right] \\ &= \max_{a_t \in \mathcal{A}_{s_t,t}} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1} | s_t, a_t) \left[r_t + \gamma v^*(s_{t+1})\right]. \end{aligned} \tag{3.9}$$

Additionally, the Bellman optimality equation for q^* also exists and is shown in (3.10). Note that, the equation in (3.8) is an expression of q^* in terms of v^* (shows the relationship between the two), while (3.10) is used to compute q^*:

$$\begin{aligned} q^*(s_t, a_t) &= \mathbb{E}\left[R_t + \gamma \max_{a_{t+1} \in \mathcal{A}_{S_{t+1},t+1}} q^*(S_{t+1}, a_{t+1}) | S_t = s_t, A_t = a_t\right] \\ &= \sum_{r_t \in \mathcal{R}_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1} | s_t, a_t) \left[r_t + \gamma \max_{a_{t+1} \in \mathcal{A}_{s_{t+1},t+1}} q^*(s_{t+1}, a_{t+1})\right]. \end{aligned} \tag{3.10}$$

Note that the decision maker may encounter issues of computational power and available memory when dealing with such problems. Mainly, the amount of computation it can carry out in a single time step and the amount of memory required to build up value functions, policies, and algorithms are limited. On the other hand, when dealing with small, finite state spaces, it is possible to store data in tables or arrays with one row for each state or state-action pair. However, many real-world applications deal with large amounts of states, where in such cases, the functions must be approximated by making use of a more compact parametrised function representation. To put this into perspective, consider a 50×50 grid world with 4 possible actions, for the state-value function needs to store 2500 values, while the action-value function needs to store 10000 numbers, which is manageable. However, consider a 500×500 grid world with the same amount of actions, this will have a state-value table of 250000 entries, while an action-value table of 1000000 entries.

The latter easily becomes memory-intensive, especially if one needs to store intermediate computations or policies for multiple time steps. Therefore, for large scale problems, exact tables cannot be stored, instead a value function is approximated.

After discussing optimal policies and their characteristics, it is important to study how one can obtain such policies. Section 3.5 delves into an efficient algorithm for obtaining optimal policies for MDPs, known as the Policy Iteration algorithm.

3.5 The Policy Iteration Algorithm

When dealing with a finite MDP, finding a unique solution to the problem at hand is possible. This solution is not achieved from a sole equation, rather, it is obtained from a system of equations, meaning that for each possible state there is an equation. Such equations are shown in expression (3.9). In the same manner, the set of equations found in (3.10) can be solved for q^*.

By solving Equation (3.9) and determining the optimal state-value function v^*, finding an optimal policy becomes simple. The maximum of Equation (3.9) can be derived through one or more actions for each state. A policy is considered optimal when a non zero probability is attributed exclusively to such actions. Such a process may be considered as a one-step search algorithm. Consequently, after v^* is attained, the actions which are optimal are the ones that emerge following the one step search. The aforementioned procedure is a greedy algorithm variant, in which the best v^* evaluated is considered in order to match to an optimal policy. Nevertheless, this method is shown to ensure policy optimality in the long term due to v^* accounting for every possible future behaviour by means of the return function, G_t.

Depending on the available computer, determining the best actions may be made easier if one manages to have q^*. This is because one would only have to determine an action a_t, for any state $s_t \in \mathcal{S}_t$, for which $q^*(s_t, a_t)$ would be the largest possible value. However this convenience comes at the cost of having to evaluate a function of state-action pairs, instead of just states, implying that a larger domain is encountered when compared to that of the state-value function v^*. As a result, this will increase the magnitude of the problem and the computational time necessary

to find a solution, since storing and computing q^* typically requires more memory and computational time.

For finite MDPs, there are two main approaches extensively used in the literature, namely the Policy Iteration algorithm and the Value Iteration algorithm. Apart from these approaches, finite MDPs can also be solved using either linear programming, Q-learning, SARSA (State-Action-Reward-State-Action), Monte Carlo methods or Temporal Difference Learning. The policy and value iteration algorithms have many similar features, but the main distinguishing factor is that the policy iteration works on improving policies, while the value iteration works on improving approximations of the optimal state-value function v^*. Furthermore, the value iteration algorithm finds an ε- optimal policy in contrast to the policy iteration algorithm, which finds an optimal policy. Due to the fact that the policy iteration algorithm is going to be used to find optimal policies in the application, this algorithm is going to be discussed in detail in Sections 3.5.1 to 3.5.2.

3.5.1 Algorithm Overview

The policy iteration algorithm is an interchanging sequence of policies and value functions, where the phases of the algorithm switch between *policy evaluation* and *policy improvement*. Should a better policy be discovered, the current policy will be replaced. This is done in order to preclude the adoption of any possible inferior policy. The illustration in Figure 3.1 shows the above described process. Furthermore, following the flow chart is the pseudo-code for this algorithm.

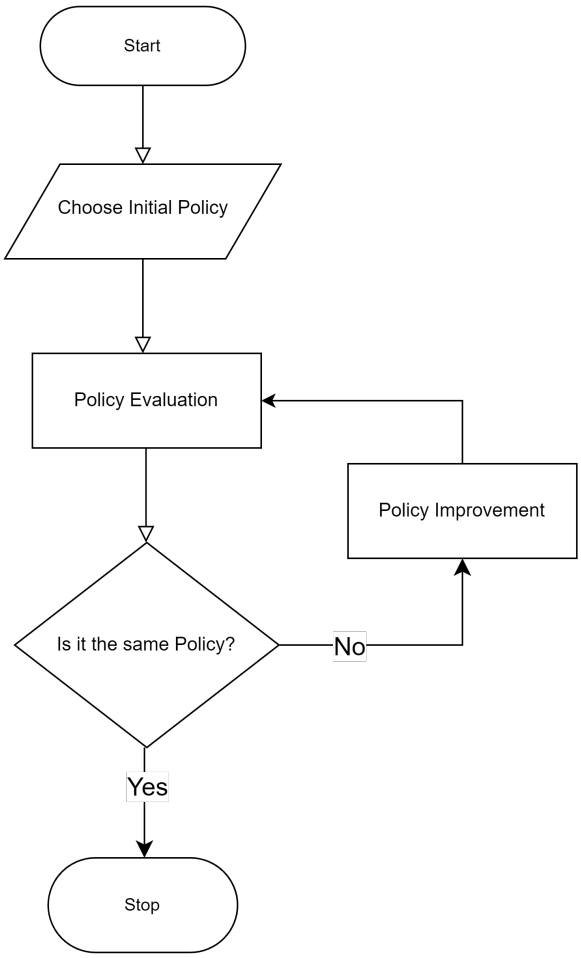

Figure 3.1: Policy Iteration Flow Chart

The Policy Iteration Algorithm

1. *Initialization*: Set $k = 1$, and select any policy $\pi_k \in \Pi$.

2. *Policy Evaluation*: Find the value of policy π_k by using the Bellman equation:

$$v_{\pi_k}(s_t) = \sum_{a_t \in \mathcal{A}_{s_t,t}} \pi(a_t|s_t) \sum_{r_t \in \mathcal{R}_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1}|s_t, a_t) \cdot [r_t + \gamma v_{\pi_k}(s_{t+1})] \quad \forall s_t \in \mathcal{S}_t.$$

3. *Policy Improvement*: Determine the new policy π_{k+1} which maximises the Bellman equation for each state $s_t \in \mathcal{S}_t$ over all possible actions $a_t \in \mathcal{A}_{s_t,t}$.

4. *Stopping Condition*: If policy $\pi_{k+1} \neq \pi_k$, then go to 2. On the other hand, if $\pi_{k+1} = \pi_k$, stop the algorithm since π_{k+1} is optimal.

In Step 2, a different value function can be taken, depending on the objective of the decision maker. For the application under study the expected total discounted reward is considered, hence making the above pseudocode applicable.

As evident above, policy iteration comprises of evaluating several policies by computing the state-value function, v_π. Furthermore, policy iteration also includes policy improvement, which enables the comparison with other state-values and hence the improvement of policies. Due to the fact that each iteration consists of a policy evaluation, which requires multiple repetitive computations over the state space, the algorithm may take a long time to converge. In such a case, an alternative to this algorithm, one that converges to an ε-optimal policy rather than a globally optimal policy, is the aforementioned Value Iteration algorithm. To understand how the policy iteration algorithm works, a short example is provided in Section 3.5.2.

3.5.2 An Illustrative Example of the Policy Iteration Algorithm

Due to the fact that most theoretical concepts can be abstract and complex, an illustrative example helps to show the practical application and the algorithm's capability. This helps since the example demonstrates the whole process of policy evaluation and improvement, step after step, making the theory easier to grasp. Furthermore, this example aims to give a comprehensive understanding of its effectiveness in real-world problems.

Let us consider a planning problem, where the state, denoted by $S_t = (x, y)$, is any of the coordinates in a 2×2 grid, and the action $A_{s_t,t}$ is the direction in which a step is taken. It is important to note that, for simplicity's sake, only the coordinates $x, y \in \{1, 2\}$ are regarded, and directions depending on which state one is in, together with staying at the same position.

For a given action $A_{s_t,t}$ in policy π, the probability of that same action taking place is 0.7, while the other actions have a remaining probability split equally between them. Furthermore, if the agent is at $(2, 2)$, the agent will stop with probability 1. A reward of 1 is given when ending up in state $(2, 2)$, while a reward of 0 is

given when ending up in other states. The discount factor is set to $\gamma = 0.9$. Figure 3.2 shows the full grid, the possible states and reward 1 in coordinate $(2, 2)$.

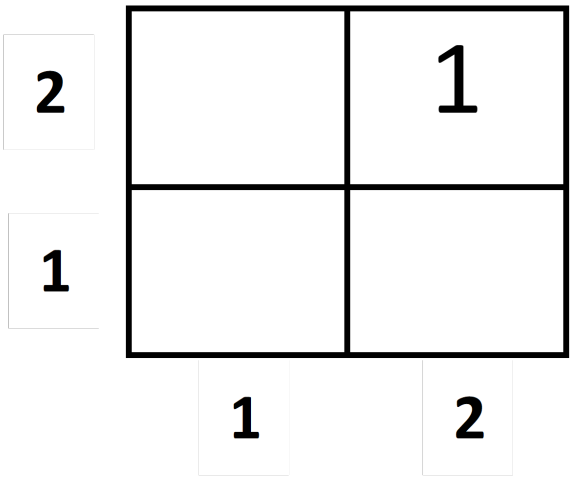

Figure 3.2: 2x2 Grid

Therefore, from state $(1, 1)$ the actions available are the directions right, up and stay, while for $(1, 2)$ the actions available are right, down, and stay. For coordinate $(2, 1)$, the actions available are left, up, and stay. On the other hand, for $(2, 2)$, the only available action is to stay (stop in that state).

Let $k = 1$. An initial policy π_1 is chosen, where $\pi_1 = [(1, 1) : \text{“}right\text{”}, (1, 2) : \text{“}down\text{”}, (2, 1) : \text{“}left\text{”}, (2, 2) : \text{“}stay\text{”}]^T$. Next, the state-value of π_1, which is $v_{\pi_1}(s_t)$, is found using the Bellman equation, which process is known as policy evaluation. Hence, the value function vector,

$$\mathbf{v}_{\pi_1} = [v_{\pi_1}\big((1, 1)\big), v_{\pi_1}\big((1, 2)\big), v_{\pi_1}\big((2, 1)\big), v_{\pi_1}\big((2, 2)\big)]^T,$$

by using the equation in the second step of the Policy iteration algorithm found in Section 3.5.1, becomes:

$$\mathbf{v}_{\pi_1} = \begin{bmatrix} 0.7(0 + 0.9v_{\pi_1}(2, 1)) + 0.15(0 + 0.9v_{\pi_1}(1, 2)) + 0.15(0 + 0.9v_{\pi_1}(1, 1)) \\ 0.15(1 + 0.9v_{\pi_1}(2, 2)) + 0.7(0 + 0.9v_{\pi_1}(1, 1)) + 0.15(0 + 0.9v_{\pi_1}(1, 2)) \\ 0.15(1 + 0.9v_{\pi_1}(2, 2)) + 0.7(0 + 0.9v_{\pi_1}(1, 1)) + 0.15(0 + 0.9v_{\pi_1}(2, 1)) \\ 1(1 + 0.9v_{\pi_1}(2, 2)) \end{bmatrix}.$$

This results in:

$$\mathbf{v}_{\pi_1} = \begin{bmatrix} 0 \\ 0.15 \\ 0.15 \\ 1 \end{bmatrix}.$$

Now that the value function vector has been found, a policy improvement step follows. The initial policy π_1 is updated to π_2 by maximising the Bellman equation for each state and over all possible actions, as follows:

$$\pi_2\big((1,1)\big) = \text{argmax} \begin{cases} \text{right: } 0.15(0.9)(0.15) + 0.7(0.9)(0.15) + 0.15(0.9)(0) = 0.115 \\ \text{up: } 0.15(0.9)(0.15) + 0.7(0.9)(0.15) + 0.15(0.9)(0) = 0.115 \\ \text{stay: } 0.15(0.9)(0.15) + 0.15(0.9)(0.15) + 0.7(0.9)(0) = 0.041 \end{cases}$$

$$\pi_2\big((1,2)\big) = \text{argmax} \begin{cases} \text{right: } 0.15(0.9)(0) + 0.7(1 + 0.9(1)) + 0.15(0.9)(0.15) = 1.350 \\ \text{down: } 0.7(0.9)(0) + 0.15(1 + 0.9(1)) + 0.15(0.9)(0.15) = 0.305 \\ \text{stay: } 0.15(0.9)(0) + 0.15(1 + 0.9(1)) + 0.7(0.9)(0.15) = 0.380 \end{cases}$$

$$\pi_2\big((2,1)\big) = \text{argmax} \begin{cases} \text{left: } 0.7(0.9)(0) + 0.15(1 + 0.9(1)) + 0.15(0.9)(0.15) = 0.305 \\ \text{up: } 0.15(0.9)(0) + 0.7(1 + 0.9(1)) + 0.15(0.9)(0.15) = 1.350 \\ \text{stay: } 0.15(0.9)(0) + 0.15(1 + 0.9(1)) + 0.7(0.9)(0.15) = 0.380 \end{cases}$$

Note that, $\pi_2\big((2,2)\big)$ remains as it is due to the problem's characteristics. Thus,

$$\pi_1 = \begin{bmatrix} (1,1) : \text{“}right\text{”} \\ (1,2) : \text{“}down\text{”} \\ (2,1) : \text{“}left\text{”} \\ (2,2) : \text{“}stay\text{”} \end{bmatrix} \rightarrow \pi_2 = \begin{bmatrix} (1,1) : \text{“}right\text{”} \\ (1,2) : \text{“}right\text{”} \\ (2,1) : \text{“}up\text{”} \\ (2,2) : \text{“}stay\text{”} \end{bmatrix}$$

Since $\pi_2 \neq \pi_1$, set $k = 2$ and go through policy evaluation and improvement again. This process is repeated until $\pi_{k+1} = \pi_k$, at which point an optimal policy π^* is found.

The convergence properties of the policy iteration algorithm illustrated in this example will be analysed and proved in Section 3.5.3.

3.5.3 Convergence of the Policy Iteration Algorithm

This section outlines a method for analysing the convergence of the policy iteration algorithm by employing what is known as the *Bellman operator*. Before delving into the proofs of the convergence of this algorithm, some preliminaries, which facilitate the mathematical rigour, are introduced.

A widely known result when proving the convergence of the policy iteration algorithm is the *Chapman-Kolmogorov Theorem*. The statement of this theorem, as per Sigman (2009), can bee seen in Theorem 3.5.1, but prior to that, the notation being used must be introduced. Consider a Markov Chain $(X_t)_{t\in\{0,1,\dots\}}$, whose transition probability matrix is denoted by P and state space by $\mathcal{S}$. Consequently, as per Sigman's notation, $P^n_{i,j}$ denotes the probability of going from state i to j in n time units.

Theorem 3.5.1 (Chapman-Kolmogorov Theorem)**.**

$$P^{n+m}_{i,j} = \sum_{k\in\mathcal{S}} P^n_{i,k} P^m_{k,j} \qquad \forall n, m \geq 0, i, j \in \mathcal{S}.$$

To prove this theorem, firstly, the state at time n is considered. The probability of being in state k following n steps is $\mathbb{P}(X_n = k|X_0 = i)$. By the Markov Property, subsequent states after time n do not depend on previous states. Thus, the probability of occupying state j after an additional m time steps is $P^m_{k,j}$. Therefore, summing over all possible $k \in \mathcal{S}$ gives us the above result.

Due to the fact that Theorem 3.5.1's result requires numerous matrix calculations, the Bellman operator can help resolve this dilemma. This mechanism maps a value function v, to another value function v'. Thus, the Bellman operator is defined as shown below as per Vajjha et al. (2021).

Definition 3.5.1 (Bellman Operator)**.** Given an MDP and policy $\pi \in \Pi$, the Bellman Operator is defined as:

$$B_\pi : (\mathcal{S}_t \to \mathbb{R}) \to (\mathcal{S}_t \to \mathbb{R}),$$

$$v(s_t) \to \sum_{a_t\in\mathcal{A}_{s_t,t}} \pi(a_t|s_t) \sum_{r_t} \sum_{s_{t+1}\in\mathcal{S}_{t+1}} p_t(r_t, s_{t+1}|s_t, a_t)[r_t + \gamma v(s_{t+1})], \quad \forall s_t \in \mathcal{S}_t$$

for any function $v : \mathcal{S}_t \to \mathbb{R}$.

Furthermore, the *Bellman optimality operator* is defined as follows:

Definition 3.5.2 (Bellman Optimality Operator)**.** For a given MDP, the Bellman optimality operator is defined as follows:

$$\hat{B} : (\mathcal{S}_t \to \mathbb{R}) \to (\mathcal{S}_t \to \mathbb{R}),$$

$$v(s_t) \to \max_{a_t \in \mathcal{A}_{s_t,t}} \sum_{r_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1}|s_t, a_t)[r_t + \gamma v(s_{t+1})], \quad \forall s_t \in \mathcal{S}_t$$

for any function $v : \mathcal{S}_t \to \mathbb{R}$.

The Bellman operator and the Bellman optimality operator both satisfy the below essential properties, where B can be either B_π, or $\hat{B}$.

1. The operator B, is monotone:

$$\forall s_t \in \mathcal{S}_t, \quad v_1(s_t) \leq v_2(s_t) \Rightarrow \forall s_t \in \mathcal{S}_t, B(v_1(s_t)) \leq B(v_2(s_t)).$$

2. B is a contraction with respect to the L_∞ norm:

$$\forall s_t \in \mathcal{S}_t, \quad ||B(v_1(s_t)) - B(v_2(s_t))||_\infty \leq \gamma ||v_1(s_t) - v_2(s_t)||_\infty.$$

Now that some preliminary results and the mathematical tools required to facilitate convergence results were provided, convergence results related to the policy iteration algorithm can be introduced. For simplicity's sake, for a given policy $\pi \in \Pi$, action value function q_π is given as:

$$q_\pi(s_t, a_t) := \sum_{r_t} \sum_{s_{t+1} \in \mathcal{S}_{t+1}} p_t(r_t, s_{t+1}|s_t, a_t)[r_t + \gamma v(s_{t+1})], \quad \forall s_t \in \mathcal{S}_t, a_t \in \mathcal{A}_{s_t,t}.$$

Definition 3.5.3 (Improved Policy)**.** Let π_k and π_{k+1} be two different policies of the problem, where k is the iteration number. Then π_{k+1} is an improvement of π_k if $\forall s_t \in \mathcal{S}_t$:

$$\pi_{k+1}(s_t) = \arg\max_{a_t \in \mathcal{A}_{s_t,t}} [q_{\pi_k}(s_t, a_t)].$$

Such a definition forms the basis for introducing the policy improvement theorem.

Theorem 3.5.2 (Policy Improvement Theorem)**.** *Let π_k and π_{k+1} be two distinct policies.*

- *If* $B_{\pi_{k+1}}(v_{\pi_k}) \geq B_{\pi_k}(v_{\pi_k})$ *, then* $v_{\pi_{k+1}} \geq v_{\pi_k}$*, or,*
- *if* $B_{\pi_{k+1}}(v_{\pi_k}) \leq B_{\pi_k}(v_{\pi_k})$ *, then* $v_{\pi_{k+1}} \leq v_{\pi_k}$*.*

Theorem 3.5.2 shows that following a better policy leads to higher rewards. This links directly to Theorem 3.5.3 by showing how these improved policies result in increased state values.

Theorem 3.5.3 (State-Value Function Improvement Theorem)**.** *If* π_k *and* π_{k+1} *are two distinct policies and* π_{k+1} *is an improvement of* π_k*, then* $v_{\pi_{k+1}} \geq v_{\pi_k}$ *when dealing with a maximization problem. On the other hand, if it is a minimization problem, then* $v_{\pi_{k+1}} \leq v_{\pi_k}$*.*

The above theorems, presented in Vajjha et al. (2021), show that the policy produced at each iteration is an improvement on the policy computed in the preceding iteration of the policy iteration algorithm. At some point, the algorithm must terminate due to the problem being finite (finite action and state spaces).

3.6 Conclusion

In conclusion, a thorough insight into the theory of MDPs and the solution technique used to find an optimal policy for the underlying problem was provided in this chapter. By having gained a deeper understanding of the theory, one is then able to apply such principles to the problem under study. Consequently, in Chapter 4, the application, namely managing a four-way intersection using MDPs, is introduced, and the theoretical concepts are applied to the problem under study.

Chapter 4

Road Infrastructure and Model

This chapter primarily deals with the application under study, that is, controlling intelligently a traffic light system in a four-way test intersection. The four-way intersection chosen by the author is one found in Gżira (Malta) near the local council building, hereinafter referred to as the test intersection. A detailed description of the different control schemes is provided in the first part of this chapter to be able to understand how different traffic light systems operate. The parameters of the intersection and the manner in which information is extracted from the test intersection are presented, thereby enabling the formulation of the MDP for this specific application.

4.1 Control Schemes

Researchers from different areas study the optimal control of traffic lights in their own way. For instance, the operations research field uses mathematical techniques to evaluate and optimise traffic light control. Another approach is that of a computer scientist, which uses Artificial Intelligence (AI) techniques to develop learning algorithms. Moreover, these approaches can also be intertwined. The main control schemes studied in the literature are: *fixed cycle* (FC), *exhaustive control*, *vehicle actuated control* and *adaptive control*. These four schemes will be discussed in detail in Sections 4.1.1 to 4.1.3.

4.1.1 Fixed Cycle

FC is a static control policy, which disregards any traffic information that changes dynamically over time (Webster, 1958). This enables FCs to have a straightforward structure on which mathematical analysis can easily be applied. When considering an FC, the order in which queues receive green-time and the duration of green-time allocated to each queue are fixed. Therefore, the time when traffic lights switch is predetermined, making the queue length meaningless. Consequently, FC can be referred to as fixed-time or pre-timed control.

A reactive version of FC can also be implemented by choosing between multiple predetermined FCs. For example, during rush hour traffic, an FC with a long *cycle length* (the time a traffic signal takes to complete one full cycle of indications) can be chosen. This is because a long cycle length gives more green-time to each traffic light in the intersection before the subsequent light changes to green. For instance, instead of priority (green) changing every 20 seconds, with a longer cycle length, priority changes every 30 seconds. Thus, giving more green-time at that instance to each direction, which prevents cars from stopping more frequently. On the other hand, short cycle lengths can be ideal for non-rush hour traffic. For example, at night, due to the decrease in the number of cars on the roads, a short cycle length will give priority to each road more quickly. This will reduce unnecessary waiting time for cars approaching the intersection.

Notwithstanding the above, a fixed-cycle approach may have its own drawbacks. For instance, since the timing pattern of an FC is predetermined, this may not accommodate the different traffic demands, thereby causing inefficiency during low or high traffic periods. An exhaustive control system aims to be a more efficient approach for traffic demand variations.

4.1.2 Exhaustive and Vehicle Actuated Control

Exhaustive control policies, as per Liu & Yu (2007), can be classified into two classes based on the exhaustiveness term: *pure exhaustive control* and *anticipative exhaustive control*. In pure exhaustive control, the green light signal changes as

soon as the queue length on a road reaches zero, meaning the green-time ends when the road is emptied. This control scheme can be cyclic, which keeps the order in which roads are given priority (green-time) fixed, or acyclic, which serves the road specified by the controller after each *phase*, discarding the cycle notion. Phases are what make up a traffic signal cycle, with each one representing a specific road or direction.

In anticipative exhaustive control, a car can leave the queue when the lights are yellow, apart from when they are green. The lights are switched from green to yellow exactly before a road is exhausted (when one or two cars remain), hence the name anticipative. Note that, this approach can also be cyclic or acyclic. However, both exhaustive control systems have the disadvantage that in heavy traffic situations, a longer waiting time due to the exhaustiveness of the priority road will be experienced by the vehicles on the roads awaiting a 'green' traffic light.

To address these limitations, a combination of FC and exhaustive control can be applied in *vehicle actuated control*. In this approach, the traffic light's green-time ends when queues are exhausted or when a maximum given time has elapsed. Moreover, with a maximum green-time, usually a minimum green-time is also applied. A detector (typically an induction loop) is placed near the stopping line (just before entering the intersection) to detect any cars which pass over it. If no cars are detected for a predetermined number of seconds, the green-time of that road ends.

A more complex and appropriate approach for continuously changing traffic conditions, is *adaptive control*. The traffic light controller by using adaptive control, is able to make real-time adjustments to traffic signal timings, thus potentially providing an improved performance compared to the ones previously discussed.

4.1.3 Adaptive Control

In adaptive control, traffic lights are manipulated based on the traffic flow conditions. When taking into account this definition, vehicle actuated control can be described as adaptive. Two extensively used methods, the Split, Cycle and Offset Optimisation Technique (SCOOT), introduced by Hunt et al. (1981), and the Sydney Co-ordinated Adaptive Traffic (SCAT), introduced by Lowrie (1982), are based

on adaptive control. Both techniques can be used to optimise the signal timings at traffic intersections based on real-time traffic conditions with the aim of improving traffic flow and minimising delays. The main difference between the two systems is that SCAT was first implemented in the Sydney metropolitan area, while SCOOT is a general term.

Under this control scheme, new information is assumed to be given to the controller every Z seconds, where Z is decided by the controller. This control scheme is said to be of high resolution if updates on the state of the system are given regularly. Thereby, adaptive control takes into consideration the number of queued cars, if any, and estimated arrival times of cars in the roads leading to the intersection.

Depending on the state of the environment, numerous actions are evaluated over some planning horizon, where the best action with respect to the controller's objective(s) is executed. The optimization process should be done in real time, meaning that at every Z seconds, the best decision or set of decisions should be found.

4.2 Traffic Modelling and Simulation

To test the different control schemes mentioned, simulations are crucial. A controller cannot experiment and test different strategies extensively on real-life systems. This is due to the accidents and traffic congestion that may occur. These systems are best examined using computer models, which yield results considerably faster and definitely safer than physical tests. Hence, simulation programs together with various modelling approaches were developed by computer scientists and transportation experts to simulate traffic systems, which as per Passos et al. (2011), are classified as follows: Microscopic, Macroscopic, Mesoscopic, or Nanoscopic.

Each of the said models portray vehicle activity and traffic accumulation differently from one another. In addition, their computational speed varies considerably, due to the different details each goes into, and hence, certain applications require a specific model.

Microscopic modelling handles vehicles individually, thus making it a highly accurate type of simulation. This approach is ideal for small models due to the long computational time a detailed simulation requires. Such models make use of car-following and lane changing concepts which help control each vehicle in the system. On the contrary, macroscopic modelling is most utilized when the traffic network under study is large, and when individual vehicles and their characteristics are not of interest. This strategy gives a comprehensive perspective of the traffic behaviour instead of information on individual vehicle management. A macroscopic model's lack of detail results in faster computations while delivering valuable information on the system.

As a means to fill the void between the mentioned modelling techniques, mesoscopic modelling was developed, which is a mix of attributes of both microscopic and macroscopic models. This approach is useful when dealing with a system of intersections, where microscopic modelling is used for the intersection of interest, while macroscopic modelling is applied to the remaining areas. Mesoscopic modelling is increasing in popularity due to the balance it creates between computational demand and the level of detail provided. Nanoscopic modelling is a relatively recent approach, and is an extension to microscopic models since it consists of driver and environmental models. As a consequence, this strategy incurs the highest computational cost since it provides the utmost level of detail. Such a modelling type is mainly used when having a self-driving vehicle application due to the level of detail needed.

Since the application being studied necessitates a certain amount of detail from vehicles without dealing with the burden of computational time, a microscopic model will be used to model the problem infrastructure. The model will be set up using Aimsun (Advanced Interactive Microscopic Simulator for Urban and Non-Urban Networks). The latter programme was developed by Transportation Simulation Systems and supports all modelling strategies mentioned, except nanoscopic. Aimsun is a traffic simulation software meant for modelling and simulating road networks, public transport and pedestrian traffic. The purpose of this software is to assist transportation engineers and controllers in analysing and optimising the functioning of urban and regional transportation infrastructure. Mainly, it is a platform for

creating realistic simulations of traffic (vehicles and pedestrians) flow by using different modelling techniques. This simulates individual vehicles, together with their interactions at an in-depth level, which allows for a proper portrayal of real world traffic conditions. In addition, it also includes tools to control traffic signal timings and control strategies, which are essential for testing the efficiency of signalised intersections and measuring traffic congestion. This can be seen further in the work of Barceló (2010).

Furthermore, Aimsun allows communication between external applications and the model by using an Application Program Interface (API). For the problem under study, it is imperative that such a feature exists, namely, because optimal policies need to be implemented. The programming language which is mainly used in this study, while being supported by the Aimsun API is Python. Prior to simulating a particular scenario, one must define the road infrastructure to be modelled. Therefore, in Section 4.3, the road infrastructure will be discussed and defined comprehensively.

4.3 Road Infrastructure

First and foremost, this study applies MDPs to improve traffic intersections. It is being emphasised that the purpose of this study is not solely to analyse and improve the test intersection. Rather, the intersection will be used as a test intersection, so that ultimately this research can be applied to any other traffic intersection. The reason behind the choice of the test intersection is mainly due to the data made available and prior research conducted on the said test intersection. Notwithstanding this, one should keep in mind that certain parameters depend upon the intersection in question, namely, inter alia, input flow of cars, number of roads leading towards the intersection, road lengths, turns and road speed limit. Moreover, the side roads leading to or exiting from the intersection are not being considered, since this study analyses the efficiency of the traffic lights at the intersection and not the efficiency of the infrastructure of the area.

The aforementioned test intersection is marked by the red circle in Figure 4.1. The figure also shows the four inputs labelled with letters A to D. Throughout this

study, the entrance/exit marked A will be referred to as Sliema, B will be referred to as Gżira, C will be referred to as Msida, while D will be referred to as Kappara.

Figure 4.1: Test Intersection and Surroundings Map with A, B, C & D referring to Sliema, Gżira, Msida & Kappara, respectively.

When a model of an intersection is being created, it is fundamental to capture its true characteristics. This way, the simulated traffic flow is aimed to be an accurate representation of the real-life situation. Therefore, the road infrastructure of this particular intersection was applied on a map of the area. As a result, this guaranteed that the lengths and turns of the roads making up the intersection are precise. In order to define more precisely the traffic intersection under study, some important definitions are provided in Section 4.3.1.

4.3.1 Defining the Test Intersection

The four-way traffic intersection being studied consists of four traffic lights (or signals), one for each direction. A *signal phase* is the configuration of a traffic light, which specifies the direction of traffic that has priority and the directions that do not. In view of this, the *cycle* of a traffic intersection is a complete sequence of the intersection's signal phases, whereas the *cycle length* is the time it takes to visit each signal phase that makes up the intersection once.

Figure 4.2 shows all the directions leading to the intersection, where Sliema, Gżira, Msida and Kappara traffic lights are the four phases that make up a cycle. Furthermore, Figure 4.2 shows that the intersection under study has four directions from where traffic approaches. These four roads, which make up the intersection, are labelled as Sliema (North), Gżira (East), Msida (South) and Kappara (West). For instance, if Sliema road is considered, the traffic approaching the intersection from the north side of the intersection is regarded, and so on. Hence, in this case, a complete cycle would be starting from the Sliema traffic phase, visiting all other phases and ending up in the Sliema phase again. Note that the way priority is given can be cyclic or acyclic, where in the former, which will be considered throughout the application, traffic priority rotates in a clockwise manner, giving green time to each road.

Currently, the Rue D'Argens intersection works with a fixed cycle of length around 92s. Inclusive in these 92s are 3s yellow time for each traffic light.

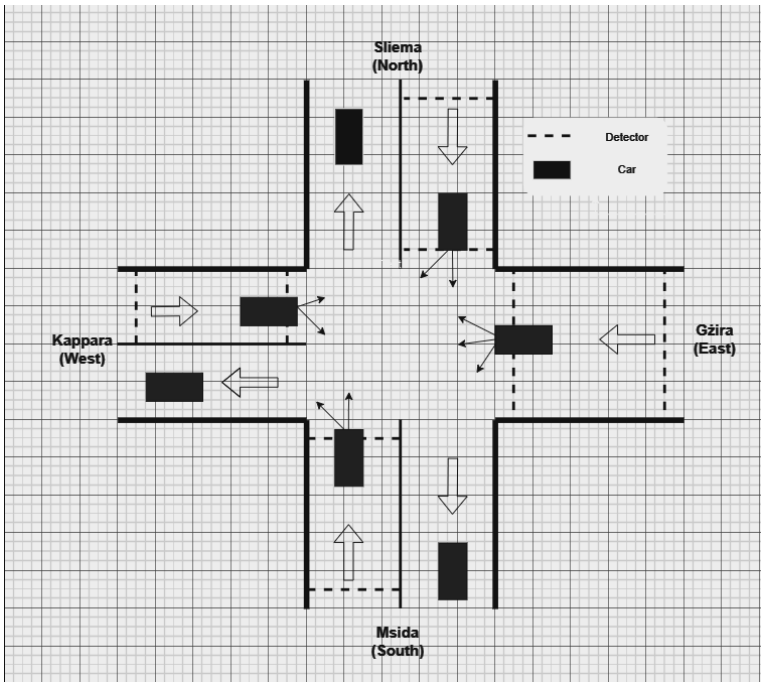

Figure 4.2: Representation of the Signalised Traffic Intersection Under Study

Apart from a precise road layout and traffic light parameters, to achieve an accurate microscopic model, the vehicle flow through the intersection needs to be correct. In order to have an extensive study, the proposed MDP driven cycle will be tested on different rush hours, namely morning and evening, and on a non-peak

hour. Thus, three different flow rates, one for each time of day, will be used. This will be delved into in Section 4.3.3. In addition, all vehicles in the simulation were chosen to be passenger vehicles, with an average length of 4.3 metres, since the input flow rates made available, presented in Table 4.1, did not have any details on the vehicle type.

In general, the junction consists of four roads; each road is composed of two lanes in opposite directions, except the East road, which has one lane flowing to the intersection. A vehicle enters the intersection and can go anywhere, except where it came from, and to the East road. Thus, the environment being considered is composed of four roads, vehicles circulating on the roads, and traffic lights for each respective road.

Vehicles communicate with the decision maker by means of detectors, which are placed on each road lane heading to the intersection. These can be seen in Figure 4.2 and will be discussed in Section 4.3.2 together with the statistics available at the detector's output.

4.3.2 Detectors

Detectors collect all types of vehicle information, the most relevant being the number of vehicles that have passed during a certain time interval. Two detectors are placed on each road lane heading to the intersection, one placed at the beginning of the road (placed at a certain stretch of road away from the stopping line) and the other at the end of the road (near the stop sign before exiting the road into the intersection).

The detectors farthest from the intersection, which record the cars entering the system, are placed at a distance from the intersection based on the authors' knowledge of the specific intersection. Hence, at a distance deemed necessary to capture the traffic caused by the traffic lights. As a consequence, *saturation* is an inevitable issue when deploying detectors in traffic management systems. Saturation is the point at which detectors cannot accurately detect and process vehicles due to the maximum measurable road capacity being attained. Detectors may fail to capture properly vehicles queuing when they become saturated, leading to imprecise traffic data. When placing such detectors, a trade-off between capturing a high traffic

volume and having an ideal detector capacity is necessary. Therefore, the detectors in the simulation are placed far enough to capture a realistic number of cars so that saturation is not a frequent issue, while being capable of capturing the cars that are directly affecting the traffic light system. In general, saturation is inevitable because detectors need to be placed somewhere in the system, regardless of where this issue persists.

The detectors closest to the intersection are placed exactly where a vehicle enters the intersection. With these two detectors in place they together acquire important traffic data for each road. The main statistic achieved is the *queue length*, which represents the amount of cars between two detectors, which is obtained by subtracting the car count of the detector closest to the intersection from the car count farthest from the intersection. Consequently, the average *waiting time* can be achieved either directly, by subtracting the entry time of a car from the exit time, or using Little's Law. The latter is a formula which states that the average waiting time can be calculated by dividing the average queue length by the arrival rate into the system. Another statistic is the *throughput*, which gives the number of cars exiting the system per hour, calculated using the vehicle exit count. There exists other information that can be extracted to help analyse the intersection, namely, the number of stops per car and vehicle speed.

On that account, the car flow rates are the foundation of the simulation. These will be provided in Section 4.3.3 for the different times of day considered.

4.3.3 Car Flows

For the morning rush hour, the majority of traffic is seen to arrive from the Kappara and Msida roads. Then, in the noon rush hour, the cars coming from Kappara decrease, while those coming from Sliema and Gżira increase significantly, with cars coming from Msida staying more or less the same. During the evening rush hour, Sliema replaces Msida as the second road with the most cars arriving in an hour. Furthermore, in non-rush hour traffic (midnight), there is a balanced input flow of cars. The input flow rates for each road leading to the intersection, for any considered hour, are provided in Table 4.1. These values are taken from Gatt,

L. (2020) and were chosen in a way that reflects the real traffic queues witnessed by this intersection. In the referred study, the values were obtained from real life observations of the Rue D'Argens junction.

Road	Morning Rush Hour (1-hour)	Noon Rush Hour (1-hour)	Evening Rush Hour (1-hour)	Midnight Hour (1-hour)
Sliema	360	550	480	72
Gżira	150	300	200	63
Msida	470	450	350	47
Kappara	800	650	600	109

Table 4.1: Input Flow Rates in veh/hr.

The input flow rates are not kept constant throughout the rush hour being considered. Rather, for the model to be more realistic, the flow rates are Gaussian distributed with mean $\frac{1}{\lambda}$, where λ is the mean input flow (vehicles/second), and variance $\sigma = 0.1(\frac{1}{\lambda})$ as tenth of the mean. This is a built-in option in Aimsun. Five minutes of warm-up time are given for each hour being considered, such that vehicles settle at the intersection when the hour being considered starts. As a result, the average arrival rates for each road leading to the intersection for each hour can be seen in Table 4.2. These are calculated by dividing the input flow rate by the amount of seconds in an hour and five minutes (warm-up time), that is 3900 seconds.

Road	Morning Rush Hour	Noon Rush Hour	Evening Rush Hour	Midnight Hour
Sliema	0.0923	0.141	0.1231	0.0185
Gżira	0.0385	0.077	0.0513	0.0162
Msida	0.1205	0.115	0.0897	0.0121
Kappara	0.2051	0.167	0.1538	0.0279

Table 4.2: Average Arrival Rates in veh/s.

The entrance of vehicles into the system is referred to as the arrival process. Vehicles are considered to have entered the system when they pass over any one of the detectors placed furthest from the intersection of each road. The rate at which vehicles enter a system is called the arrival rate. Thus, in this study, an arrival rate is denoted by $\lambda_\circ$, for each entrance into the system. Specifically, $\lambda_\circ$, where the symbol $\circ$ can be either Sliema (S), Gżira (G), Msida (M) or Kappara (K). Note that arrivals at different queues (roads), are independent.

By introducing the arrival rates above, the arrival process can be delved into. Even though the arrival rates are available, these are only averages, and what happens at particular instants throughout the hour may not be perfectly represented with these averages. Therefore, in Section 4.3.4, the manner in which the arrival of cars in the system is calculated is introduced.

4.3.4 Arrival Process

With the help of an arrival rate, one can have an idea of the number of cars arriving per unit time. However, since the arrival rate is an average, the arrival process may not be well represented by a deterministic arrival process. The drawback of considering a deterministic arrival process is that the vehicles arrive in the queue at equal intervals, when in reality, this may not be the case. Hence, as per the conventional approach (Shortle et al. 2018), a Poisson distribution will be used to manage the arrival process. Hence, in order to find the probability of a specific number of cars arriving, n_A, with arrival rate $\lambda_\circ$ (same definition as in Section 4.3.3) in a queue, the following is used:

$$P(n_A; \lambda_\circ) = \frac{(\lambda_\circ t_s)^{n_A} e^{-\lambda_\circ t_s}}{n_A!}, \tag{4.1}$$

where t_s is the time step being considered. Due to the discretisation of states in Section 4.4.1, Equation (4.1) is not enough to directly calculate the probability of a whole interval (different amount of cars arriving), for instance, between 0 and 4 cars arriving (being the probability of 0, 1, 2, 3 and 4 cars arriving). The latter is known as the Poisson cumulative distribution function (cdf), which achieves the

probability of at most n_A arrivals and is defined as:

$$F(n_A; \lambda_\circ) = \sum_{i=0}^{n_A} \frac{(\lambda_\circ t_s)^i e^{-\lambda_\circ t_s}}{i!}. \tag{4.2}$$

4.3.5 Departure Process

In contrast to arrivals, the exit of vehicles from a system is referred to as the departure process. In this case, a vehicle is considered to depart from the system when it passes the detector closest to the intersection, near the stopping line. The rate at which vehicles exit the system, is called the departure rate. Unlike the arrival rate, the departure rate is deterministic; hence, each possible exit from the system has the same departure rate. This is because a vehicle is not affected by the road it departs from.

The departure process, similarly to the arrival process, can be taken as deterministic or stochastic. A stochastic departure process unnecessarily complicates the process, owing to the fact that every vehicle is simply waiting for the traffic lights to turn green and leave the queue as quickly as possible. Thus, a deterministic departure process is preferred, even in the literature, as per Prinsen (2006) and Haijema (2008). The only thing which affects how quickly one leaves the queue is the acceleration time of the vehicle. The acceleration time of a vehicle depends on the characteristics of the vehicle, human reaction time and road conditions. Assuming that road conditions are normal (flat, dry roads), hence not considering conditions such as sloped or wet roads, and acceleration is not as relevant in the few metres being considered. Thereby, the human reaction time is the main factor when considering a deterministic departure process. The average human reaction time to a green light is considered to be between 1.5 and 2.5 seconds as per the U.S. Department of Transportation's Federal Highway Administration. Therefore, it is assumed that when a signal shows green and as long as cars are present, a vehicle exits the system every 2 seconds in a deterministic fashion which is quite common in traffic engineering. As a result, the departure rate, denoted by μ, is given by $\mu = 0.5$.

When considering the probability of cars departing from the system with rate μ, the probability is 1 for all instances. For instance, in a time step of 10 seconds, $0.5 \times 10 = 5$ vehicles exit the system every 10 seconds (if there are 5 or more vehicles

in the system) with probability 1. When the queue length is less than 5, the amount of cars in the queue leave within this time interval with a probability of 1.

After discussing in detail the application and its framework, one needs to implement the theory of MDPs to the said application. By formulating the MDP model this transition can be made.

4.4 Formulation of MDP Model

Our model is based on the following assumptions:

- The detectors, those which first interact with the vehicles leading to the intersection, had to be placed at a certain point on each road. This decision was made solely by the decision maker and was based on where the decision maker sees fit, as discussed in Section 4.3.2. Hence, only the intersection and roads leading to it are considered in the problem. As a consequence, any side-roads that exist in the real intersection are disregarded, namely 3 in the Sliema road, 2 in the Gżira road, 1 in the Msida road, and 4 in the Kappara road.
- A discrete time step is considered. This time step can be adjusted as per the needs of the controller. A discrete time step of 10 seconds is taken. Although there is considerable debate amongst traffic professionals as to the appropriate values to use for minimum green-time, a minimum green-time of 10 seconds is considered in the model being presented, which is based on suggestions made by the U.S Department of Transportation. The U.S Department of Transportation states that the green-time phase duration must be no shorter than some absolute minimum time, such as 5 to 10 seconds, and if pedestrians may be crossing during this phase, their crossing time must also be considered and included in the minimum phase length.

4.4.1 States

The state is represented by the collection of variables $S_t = \{I_t, C_t\}$, where:

- I_t is the variable corresponding to the state of the intersection at time t. There are four possible states of the junction:

$$I_t \in \Big\{\text{Sliema-Open}, \text{Gżira-Open}, \text{Msida-Open}, \text{Kappara-Open}\Big\}$$

  It is important to note that when a certain road is open, the other roads are closed. Hence, it is assumed that only one road can be open at a time.

- C_t corresponds to the state of the cars at time t, which is given with the help of the detectors. $C_t = (W_t^S, W_t^G, W_t^M, W_t^K)$ where W_t^S, W_t^G, W_t^M, $W_t^K \in \mathbb{N}_0$. Note that $\mathbb{N}_0$ is the set of natural numbers, including zero. The variables that make up C_t are the amount of waiting cars (cars in the queue) for each road composing the intersection at time t. Therefore;

$$W_t^S - \text{Amount of waiting cars on the Sliema-road at time } t \tag{4.3a}$$

$$W_t^G - \text{Amount of waiting cars on the Gżira-road at time } t \tag{4.3b}$$

$$W_t^M - \text{Amount of waiting cars on the Msida-road at time } t \tag{4.3c}$$

$$W_t^K - \text{Amount of waiting cars on the Kappara-road at time } t. \tag{4.3d}$$

In general, the state is represented as follows:

$$S_t = \{I_t, C_t\}. \tag{4.4}$$

An example of the state of the system at time t would be:

$$S_t = \{\text{`Sliema-Open'}, (k, l, m, n)\},$$

where $k, l, m, n \in \mathbb{N}_0$ are realizations of W_t^S, W_t^G, W_t^M, W_t^K, respectively.

4.4.1.1 Grouping of State Space

As discussed in Section 4.3, the detector pair of each road being considered is at a particular distance from each other. As a consequence, a specific maximum queue length exists for each road. This is calculated using the Aimsun car length, which is $4.3m$, and the detector pair distances. As a result, the maximum queue length for

each road is achieved. These are 35, 40, 50 and 100 for Sliema, Gżira, Msida and Kappara, respectively.

Considering all possible queue length combinations and the four road directions which can be open, this results in an incredibly large state space. Specifically, $4 \times 35 \times 40 \times 50 \times 100 = 28,000,000$ different states. Moreover, each state then has its own specific next possible states. This makes the problem computationally complex, both in terms of memory and time. For this reason, the state space was discretised in order to ease the computational burden while still solving the problem under study. The discretisation process consists of firstly placing queue lengths into categories with range 5, which are $0-4$,$5-9$,$10-14,\ldots$,$100-104$. For example, consider a queue length of 8, it will be represented by the category $5-9$. Additionally, these categories are given a number so that they are easily represented when coding, as shown in Tables 4.3 and 4.4.

Queue Length Category
$0-4$
$5-9$
$10-14$
$15-19$
$20-24$
$25-29$
$30-34$
$35-39^{*S}$
$40-44^{*M}$
$45-49$
$50-54^{*G}$
$\vdots$
$100-104^{*K}$

Table 4.3: Categories

$\Longrightarrow$

Category Number
1
2
3
4
5
6
7
8
9
10
11
$\vdots$
21

Table 4.4: Category No.

Note that the asterisks in the queue length category list indicate the end category of each road, $*S$ for Sliema, $*G$ for Gżira,$*M$ for Msida and $*K$ for Kappara. If it is an end category for a specific road, rather than a range of 5, the category

will contain any number from the maximum queue length upwards. For instance, category $35-39$ for Sliema becomes 35+, which includes 35 and any number larger than it. The same goes for the categories which contain the maximum queue length of other roads. Therefore, expression (4.4) stays the same but the state of the cars C_t at time t, denoted by expressions (4.3a), (4.3b), (4.3c), and (4.3d), change in the following way:

$$C_t^S - \text{Queue length category on the Sliema-road at time } t \tag{4.5a}$$

$$C_t^G - \text{Queue length category on the Gżira-road at time } t \tag{4.5b}$$

$$C_t^M - \text{Queue length category on the Msida-road at time } t \tag{4.5c}$$

$$C_t^K - \text{Queue length category on the Kappara-road at time } t. \tag{4.5d}$$

It is important to note that both definitions were shown because Aimsun outputs queue lengths and as explained, due to the large state space these values create, these are categorised, thus the latter definition of C_t. From here onwards, C_t is taken as defined in expression (4.5). This way the state space becomes much smaller, specifically $4 \times 8 \times 11 \times 9 \times 21 = 66,528$. Additionally, the endpoints are defined as follows, $k_{max} = 8$, $l_{max} = 11$, $m_{max} = 9$ and $n_{max} = 21$, which are the maximum possible category number for each road.

4.4.2 Actions

The decision maker controls the state of the intersection by taking appropriate actions. The action $A_{s_t,t}$ taken at time step t under state s_t can be either;

- *Stay the same*: Action of keeping the state of the traffic lights unchanged.
- *Change*: Action to change the state of the traffic lights to the next traffic light phase (for example, changing from 'Sliema-Open' to 'Gżira-Open').

Let us take a scenario where the current state of the intersection at time t is $s_t = \{\text{'Sliema-Open'}, (k, l, m, n)\}$. Once the state is perceived, an action out of the two presented above needs to be taken. Say it is decided that $a_t =$ Stay the same, then Sliema remains open up until time $t + 1$.

4.4.3 Transition Probabilities

Recall Equation (2.3), the state transition probability $\bar{p}(s_{t+1}|s_t, a_t)$ defines the probability of transitioning to state $S_{t+1} = s_{t+1}$ by choosing action $A_{s_t,t} = a_t$ when the system was at state $S_t = s_t$.

As shall be seen in Tables 4.5 and 4.6, transition functions $f_{I_t I_{t+1}}(\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n})$ are defined, where $I_t I_{t+1}$ denotes any one of the following transitions: Sliema-Gżira (SG), Gżira-Msida (GM), Msida-Kappara (MK), Kappara-Sliema (KS), Sliema-Sliema (SS), Gżira-Gżira (GG), Msida-Msida (MM), or Kappara-Kappara (KK). The arguments of each function are the vectors $\mathbf{k} = (k, k')^T$, $\mathbf{l} = (l, l')^T$, $\mathbf{m} = (m, m')^T$, and $\mathbf{n} = (n, n')^T$, where k, l, m and n are the queue length category at time t, while k', l', m' and n' are the queue length category at time $t + 1$, for each respective road. The first four transitions, shown in Table 4.5, give the probability of being in state $S_{t+1} = \{I_{t+1}, (k', l', m', n')\}$ having taken action 'Change' in state $S_t = \{I_t, (k, l, m, n)\}$, while the last four transitions, shown in Table 4.6, give the probability of being in state $S_{t+1} = \{I_{t+1}, (k', l', m', n')\}$ having taken action 'Stay the same' in state $S_t = \{I_t, (k, l, m, n)\}$.

I_t	I_{t+1}	$\bar{p}_t\Big(\{I_{t+1},(k',l',m',n')\}\vert\text{`Change'},\{I_t,(k,l,m,n)\}\Big)$
Sliema-Open	Sliema-Open	0
	Gżira-Open	$f_{SG}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$
	Msida-Open	0
	Kappara-Open	0
Gżira-Open	Sliema-Open	0
	Gżira-Open	0
	Msida-Open	$f_{GM}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$
	Kappara-Open	0
Msida-Open	Sliema-Open	0
	Gżira-Open	0
	Msida-Open	0
	Kappara-Open	$f_{MK}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$
Kappara-Open	Sliema-Open	$f_{KS}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$
	Gżira-Open	0
	Msida-Open	0
	Kappara-Open	0

Table 4.5: Transition Characteristics with $a_t =$ 'Change' and $S_t = \{I_t,(k,l,m,n)\}$

I_t	I_{t+1}	$\bar{p}_t\Big(\{I_{t+1},(k',l',m',n')\}\vert$'Stay the same', $\{I_t,(k,l,m,n)\}\Big)$
Sliema-Open	Sliema-Open	$f_{SS}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$
	Gżira-Open	0
	Msida-Open	0
	Kappara-Open	0
Gżira-Open	Sliema-Open	0
	Gżira-Open	$f_{GG}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$
	Msida-Open	0
	Kappara-Open	0
Msida-Open	Sliema-Open	0
	Gżira-Open	0
	Msida-Open	$f_{MM}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$
	Kappara-Open	0
Kappara-Open	Sliema-Open	0
	Gżira-Open	0
	Msida-Open	0
	Kappara-Open	$f_{KK}(\mathbf{k},\mathbf{l},\mathbf{m},\mathbf{n})$

Table 4.6: Transition Characteristics with a_t = 'Stay the same' and $S_t = \{I_t,(k,l,m,n)\}$

The zeros, in both Table 4.5 and 4.6, indicate that there is a probability of zero of ending up in a particular state $S_{t+1} = s_{t+1}$ when taking action $A_{s_t,t} = a_t$ while in state $S_t = s_t$. In other words, that particular transition cannot happen. For simplicity's sake, the Sliema (S), Gżira (G) and Msida (M) roads are sometimes referred to by X, where $X \in \{S,G,M\}$. Kappara (K) is excluded from the latter because of its substantially higher vehicle arrival rate, and hence the transition probabilities alter in structure and cannot be generalised along with the others. In addition, x will be used to represent the queue length category in some instances. This is due to the fact that transition probabilities concerning the said roads have the same structure; the only thing that changes from one to another is the arrival

rate $\lambda_\circ$, as defined in Section 4.3.3, and queue length category representation.

Assume that the information is collected every 10 seconds as described. As seen, $f_{I_t I_{t+1}}(\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n})$ is defined as:

$$f_{I_t I_{t+1}}(\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n}) = \mathbb{P}\Big(S_{t+1} = \{I_{t+1}, (k', l', m', n')\} | A_{s_t,t} = a_t, S_t = \{I_t, (k, l, m, n)\}\Big), \tag{4.6}$$

where a_t = 'Change' or a_t = 'Stay the same'. Since I_{t+1}, $C^S_{t+1} = k'$, $C^G_{t+1} = l'$, $C^M_{t+1} = m'$, $C^K_{t+1} = n'$ are all independent events, then generally

$$\mathbb{P}(X_1, X_2, \ldots, X_n | Z) = \prod_{i=1}^{n} \mathbb{P}(X_i | Z),$$

therefore, equation (4.6) results in:

$$\begin{aligned} f_{I_t I_{t+1}}(\mathbf{k}, \mathbf{l}, \mathbf{m}, \mathbf{n}) = \mathbb{P}(I_{t+1} | A_{s_t,t} = a_t, S_t = s_t) \\ \cdot \mathbb{P}(C^S_{t+1} = k' | A_{s_t,t} = a_t, S_t = s_t) \\ \cdot \mathbb{P}(C^G_{t+1} = l' | A_{s_t,t} = a_t, S_t = s_t) \\ \cdot \mathbb{P}(C^M_{t+1} = m' | A_{s_t,t} = a_t, S_t = s_t) \\ \cdot \mathbb{P}(C^K_{t+1} = n' | A_{s_t,t} = a_t, S_t = s_t). \end{aligned} \tag{4.7}$$

The roads which have red lights from time t up until $t + 1$ are going to be assumed that they can only remain the same or increase with respect to the amount of waiting cars. Thus, no cars can leave the queue when the roads have a red light, which may not be the case when you have cars parking or side roads. Moreover, since C^S_t, C^G_t, C^M_t and C^K_t are collected every 10 seconds, the difference in the queue length categories between time t and time $t+1$ is going to be assumed that it cannot be more than 2 categories, hence at most 10 cars arrive between time t and $t + 1$. Except for Kappara road which, due to its high arrival rates, can sometimes move up to 3 categories, which means at most 15 cars can arrive. These values were decided based on the arrival and departure processes described in Sections 4.3.4 and 4.3.5.

The first term of expression (4.7) denotes the probability of making a possible intersection transition based on the action 'Change' or 'Stay the same', which is equivalent to 1. For example, when taking action 'Change' while in state 'Sliema-Open', the only change in the intersection state that can happen is that to 'Gżira-Open' due to the cyclic feature of the traffic intersection. On the other hand,

when taking action 'Stay the same' while in 'Sliema-Open', the next state of the intersection can only be 'Sliema-Open'. The second till the fourth terms of (4.7) have a similar structure but can change in structure based on whether the respective road is open or not. Lastly, the fifth term in (4.7) represents Kappara's probability distribution which again changes based on whether the road is open or not. The calculation of these terms shall now be discussed.

4.4.3.1 Sliema, Gżira and Msida Roads Open

Let us assume that road $X \in \{S, G, M\}$ is open between time t and time $t+1$, and that at time t, $C_t^X = x$. The probability $\mathbb{P}[C_{t+1}^X = x' | C_t^X = x]$ is represented by three equations, one for each of the following cases: (i) $x = 1$, (ii) $1 < x < x_{max}$, and (iii) $x = x_{max}$, as follows. Following these equations is a detailed explanation on how they were derived.

Case (i): If $x = 1$:

$$P(C_{t+1}^X = x' | C_t^X = x) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(9-i;\lambda_X), & \text{if } \; x' = x \\ 1 - \frac{1}{5}\sum_{i=0}^{4} F(9-i;\lambda_X), & \text{if } \; x' = x+1 \\ 0 & \text{Otherwise.} \end{cases} \tag{4.8}$$

Case (ii): If $1 < x < x_{max}$:

$$P(C_{t+1}^X = x' | C_t^X = x) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i;\lambda_X), & \text{if } \; x' = x-1 \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(9-i;\lambda_X) & \text{if } \; x' = x \\ \qquad -F(4-i;\lambda_X)\Big], & \\ 1 - \frac{1}{5}\Bigg(\sum_{i=0}^{4}\Big[F(9-i;\lambda_X) & \text{if } \; x' = x+1 \\ \qquad -F(4-i;\lambda_X) & \\ \qquad +F(i;\lambda_X)\Big]\Bigg), & \\ 0 & \text{Otherwise.} \end{cases} \tag{4.9}$$

Case (iii): If $x = x_{max}$:

$$P(C_{t+1}^X = x' | C_t^X = x) = \begin{cases} \sum_{i=0}^{4} \omega_{4-i} F(i; \lambda_X), & \text{if } x' = x - 1 \\ 1 - \sum_{i=0}^{4} \omega_{4-i} F(i; \lambda_X), & \text{if } x' = x \\ 0 & \text{Otherwise.} \end{cases} \tag{4.10}$$

Note that $F(i; \lambda_X)$ is calculated as indicated in (4.2) and that since road X is open between time t and time $t+1$, departures from the queue are possible. Thus, equations (4.8), (4.9) and (4.10) are implicitly inclusive of the departure process. Due to the fact that the departure process is deterministic, as described in Section 4.3.5, the probability of 5 cars departing between time t and time $t+1$ is assumed to be 1 throughout equations (4.8), (4.9) and (4.10).

Equations (4.8)-(4.10) are separated since (4.8) and (4.10) correspond to the two endpoints of x, while (4.9) corresponds to what lies between the endpoints. The summation $\sum_{i=0}^{4}$ and the constant $\frac{1}{5}$ are displayed in equations (4.8) and (4.9) to take a simple average of the relevant cdf's. This is done in order to represent all five possible values that a category consists of. Hence, an assumption is made that it is equally likely to be in any position in the given range. For example for category $0-4$, the queue length can either be 0, 1, 2, 3 or 4, hence for the category to represent all of these values, an average of the cdf's is taken. Furthermore, the probability complement is present in each expression since this ensures that the total probability sums up to 1.

In (4.8) the different cases considered are that of remaining in the same queue length category or going to the next category. Only these two are considered since at category $0-4$ it is impossible to go to a less than 0 queue length category. Note that the $\frac{1}{5}$ is found in all equations discussed (4.8) onwards, and is present for the same reason as mentioned here above. Considering the first term, that of staying in the same queue length category, results in $\frac{1}{5} \sum_{i=0}^{4} F(9-i; \lambda_X)$, where the summation gives the cumulative probability of a Poisson distribution for different arrival counts and the factor in front is an averaging factor. The latter is placed so that an average probability over the five summed terms is given. The expression $9 - i$ indicates that the CDF for various arrivals is being evaluated, starting from at most 9 and decreasing to at most 5. Although at most 9 cars can arrive, hence increasing two

category steps, due to the deterministic departure of 5 cars (or less, if there is less than 5 cars on the specific road) the category can either remain the same or increase by one.

On the other hand, (4.9) has three cases: one shifts to a one-step lower queue length category, another stays in the same category and the third case goes to a one-step higher category. The first case is when the system is dealing with relatively low vehicle arrivals, which the summation shown is giving the probability of 0 to 4 arrivals. The second case is for when the queue length category remains the same. Hence, for this case, arrivals and departures are more or less the same, while the subtraction inside the formula gives the probability of between $5-i$ and $9-i$ arrivals. The last case for expression (4.9) gives the probability that a queue length category increases, signifying that arrivals exceed departures.

In (4.10), the two cases consist of shifting one-step to a lower category or staying in the same category. This is because $x = x_{max}$ is the maximum queue length category it can be in. It is important to note that the end category, x_{max}, includes any queue length above what was described in Table 4.3, hence making it an open-ended queue length category. When dealing with the open-ended queue length category, weight vector $\boldsymbol{\omega} = \Big[\alpha, \alpha(1-\alpha), \alpha(1-\alpha)^2, \alpha(1-\alpha)^3, \alpha(1-\alpha)^4, \dots\Big]^T$ was introduced, where $\alpha = 0.5$ and the vector entries add up to 1. This is done so that the model prioritises queue lengths that are closer to the endpoint category by assigning them more weight, while giving less weight to those that are further away. Note that, $\omega_0 = \alpha$, $\omega_1 = \alpha(1-\alpha), \dots, \omega_j = \alpha(1-\alpha)^j$ and so on. The latter definition is the same all throughout the different cases presented from here onwards, hence it is defined only once. As a result, when the number of waiting cars is 5 or more than the exact endpoint, there is a zero probability of ending up in category $x_{max} - 1$. This is due to the deterministic 5 cars departing every time step, thus, a lesser category cannot be attained within one time step. As a result, the probabilities for this scenario are represented as in (4.10).

4.4.3.2 Sliema, Gżira and Msida Roads Closed

In contrast, equations (4.11) - (4.17) do not take into account the departure process, hence considering no arrivals, one arrival or multiple arrivals only. Now assume that road $X \in \{S, G, M\}$ is closed between time t and time $t+1$, and that at time t, $C_t^X = x \in \{k, l, m\}$.

The probability $\mathbb{P}[C_{t+1}^X = x' | C_t^X = x]$ is represented by three equations, one for each of the following cases: (i) $1 \leq x < x_{max} - 1$, (ii) $x = x_{max} - 1$, and (iii) $x = x_{max}$. This segregation of scenarios was done such that equation (4.11) represents the situation where a category can increase up to two steps due to the arrival process. In addition, (4.12) represents being in a category exactly before x_{max}, thus an increase of two steps is impossible, while staying in the same category or increasing one step are catered for. The final scenario is that of being in x_{max} where it is impossible to go to any other category except remaining in x_{max}.

Case (i): If $1 \leq x < x_{max} - 1$:

$$P(C_{t+1}^X = x' | C_t^X = x) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i; \lambda_X), & \text{if } x' = x \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(9-i; \lambda_X) & \text{if } x' = x+1 \\ \qquad -F(4-i; \lambda_X)\Big], & \\ 1 - \frac{1}{5}\Bigg(\sum_{i=0}^{4}\Big[F(9-i; \lambda_X) & \text{if } x' = x+2 \\ \qquad -F(4-i; \lambda_X) & \\ \qquad +F(i; \lambda_X)\Big]\Bigg), & \\ 0 & \text{Otherwise.} \end{cases} \tag{4.11}$$

Case (ii): If $x = x_{max} - 1$:

$$P(C_{t+1}^X = x' | C_t^X = x) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i; \lambda_X), & \text{if } x' = x \\ 1 - \frac{1}{5}\sum_{i=0}^{4} F(i; \lambda_X), & \text{if } x' = x+1 \\ 0 & \text{Otherwise.} \end{cases} \tag{4.12}$$

Case (iii): If $x = x_{max}$:

$$P(C_{t+1}^X = x' | C_t^X = x) = \begin{cases} 1, & \text{if } x' = x \\ 0 & \text{Otherwise.} \end{cases} \tag{4.13}$$

Equation (4.11) is used when X is closed and the current queue length category x is between $1 \leq x < x_{max} - 1$ at time t. If it is the case that the queue length category at time $t+1$ remains the same, hence $x' = x$, the probability of this happening is represented by $\frac{1}{5}\sum_{i=0}^{4} F(i; \lambda_X)$. The latter gives the simple average of; at most 4 vehicles arriving if the queue length is the first value of the category, at most 3 vehicles arriving if the queue length is the second value of the category, at most 2 vehicles arriving if the queue length is the third value of the category, at most 1 vehicle arriving if the queue length is the fourth value of the category, and at most 0 vehicles arriving if the queue length is the fifth value of the category.

For the case when the queue length category at time $t+1$ increases by one, hence $x' = x + 1$, the probability of this happening is represented by $\frac{1}{5}\sum_{i=0}^{4}\Big[F(9 - i; \lambda_X) - F(4 - i; \lambda_X)\Big]$. This expression gives the simple average of: between $5 - 9$ vehicles arriving if the queue length is the first value of the category, between $4 - 8$ vehicles arriving if the queue length is the second value of the category, between $3 - 7$ vehicles arriving if the queue length is the third value of the category, between $2 - 6$ vehicles arriving if the queue length is the fourth value of the category i.e. $i = 3$, and between $1 - 5$ vehicles arriving if the queue length is the fifth value of the category.

For the case when the queue length category at time $t+1$ increases by two, hence $x' = x + 2$, the probability of this happening is represented by the complement of the probabilities of $x' = x$ and $x' = x + 1$.

In order to keep on the closed road track, we will now delve into the probabilities relating to the Kappara road when closed.

4.4.3.3 Kappara Road is Closed

For the Kappara road, the equations seen up till now differ slightly in terms of structure, due to the road being much larger and supporting a higher car arrival

rate. Since the probability of more than 9 vehicles to enter this particular road is no longer 0, a new scenario needed to be included for Kappara, which is that of $10-14$ vehicles entering the Kappara road. Here it is assumed that Kappara road is closed between time t and time $t+1$, and that at time t, $C_t^K = n$. The probability $\mathbb{P}[C_{t+1}^K = n' | C_t^K = n]$ is represented by four equations, one for each of the following cases: (i) $n < n_{max} - 2$, (ii) $n = n_{max} - 2$, (iii) $n = n_{max} - 1$, and (iv) $n = n_{max}$.

Case (i): If $n < n_{max} - 2$:

$$P(C_{t+1}^K = n' | C_t^K = n) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i;\lambda_K), & \text{if } n' = n \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \text{if } n' = n+1 \\ \qquad -F(4-i;\lambda_K)\Big], & \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(14-i;\lambda_K) & \text{if } n' = n+2 \\ \qquad -F(9-i;\lambda_K)\Big], & \\ 1 - \frac{1}{5}\Bigg(\sum_{i=0}^{4}\Big[F(14-i;\lambda_K) & \text{if } n' = n+3 \\ \qquad\qquad -F(9-i;\lambda_K)\Big] & \\ \quad + \sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \\ \qquad\qquad -F(4-i;\lambda_K)\Big] & \\ \quad + \sum_{i=0}^{4} F(i;\lambda_K)\Bigg), & \\ 0 & \text{Otherwise.} \end{cases} \tag{4.14}$$

Case (ii): If $n = n_{max} - 2$:

$$P(C_{t+1}^K = n' | C_t^K = n) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i;\lambda_K), & \text{if } n' = n \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \text{if } n' = n+1 \\ \qquad -F(4-i;\lambda_K)\Big], & \\ 1 - \frac{1}{5}\Bigg(\sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \text{if } n' = n+2 \\ \qquad\qquad -F(4-i;\lambda_K)\Big] & \\ \quad + \sum_{i=0}^{4} F(i;\lambda_K)\Bigg), & \\ 0 & \text{Otherwise.} \end{cases} \tag{4.15}$$

Case (iii): If $n = n_{max} - 1$:

$$P(C_{t+1}^K = n' | C_t^K = n) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i;\lambda_K), & \text{if } n' = n \\ 1 - \frac{1}{5}\sum_{i=0}^{4} F(i;\lambda_K), & \text{if } n' = n+1 \\ 0 & \text{Otherwise.} \end{cases} \tag{4.16}$$

Case (iv): If $n = n_{max}$:

$$P(C_{t+1}^K = n' | C_t^K = n) = \begin{cases} 1, & \text{if } n' = n \\ 0 & \text{Otherwise.} \end{cases} \tag{4.17}$$

The first two segments of equation (4.14) are similar to the ones discussed in the previous sub-sections for when road X is closed, hence, there is no need for further detail. On the contrary, the third case of (4.14) is only found in relation to the Kappara road since, due to its characteristics, an extra queue length category can be achieved, when compared to the other roads. For the case when the queue length category at time $t+1$ increases by two, hence $n' = n+2$, the probability of this happening is represented by $\frac{1}{5}\sum_{i=0}^{4}\Big[F(14-i;\lambda_K) - F(9-i;\lambda_K)\Big]$. This expression gives the simple average of: between $10-14$ vehicles arriving if the queue length is the first value of the category, between $9-13$ vehicles arriving if the queue length is the second value of the category, between $8-12$ vehicles arriving if the queue length is the third value of the category, between $7-11$ vehicles arriving if the queue length is the fourth value of the category, and between $6-10$ vehicles arriving if the queue length is the fifth value of the category. Expressions (4.15) - (4.17) are similar to (4.11) - (4.13), hence they do not need a detailed explanation like equation (4.14).

Without loss of generality, this procedure is done for f_{SG}, f_{GM},f_{KS}, f_{SS}, f_{GG}, and f_{MM}, where the departure process is considered for the road which is opened and not considered for the closed roads at time t. When it comes to Kappara some exceptions arise, which can be seen in the differences between equations (4.11) - (4.13) and (4.14) - (4.17).

4.4.3.4 Kappara Road is Open

When considering f_{MK} and f_{KK} the probability distribution of C_t^K, which considers the open road between time t and $t+1$, is represented by one of four equations, one equation for each of the following scenarios: when $n = 1$, $1 < n < n_{max} - 1$, $n = n_{max} - 1$, or $n = n_{max}$. Here the departure process is also taken into account, since these probabilities reflect an open road. Due to the fact that the departure process is deterministic, as described in Section 4.3.5, the probability of 5 cars departing between time t and time $t+1$ is assumed to be 1 throughout equations (4.18), (4.19), (4.20) and (4.21).

Case (i): If $n = 1$:

$$P(C_{t+1}^K = n' | C_t^K = n) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(9-i;\lambda_K), & \text{if } n' = n \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(14-i;\lambda_K) & \text{if } n' = n+1 \\ \qquad -F(9-i;\lambda_K)\Big], & \\ 1 - \frac{1}{5}\Bigg(\sum_{i=0}^{4}\Big[F(14-i;\lambda_K) & \text{if } n' = n+2 \\ \qquad -F(9-i;\lambda_K)\Big] & \\ \qquad + \sum_{i=0}^{4} F(9-i;\lambda_K)\Bigg), & \\ 0. & \text{Otherwise.} \end{cases} \tag{4.18}$$

Case (ii): If $1 < n < n_{max} - 1$:

$$P(C_{t+1}^K = n'|C_t^K = n) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i;\lambda_K), & \text{if } n' = n-1 \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \text{if } n' = n \\ \qquad -F(4-i;\lambda_K)\Big], & \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(14-i;\lambda_K) & \text{if } n' = n+1 \\ \qquad -F(9-i;\lambda_K)\Big], & \\ 1 - \frac{1}{5}\Bigg(\sum_{i=0}^{4}\Big[F(14-i;\lambda_K) & \text{if } n' = n+2 \\ \qquad\qquad -F(9-i;\lambda_K)\Big] & \\ \qquad + \sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \\ \qquad\qquad -F(4-i;\lambda_K)\Big] & \\ \qquad + \sum_{i=0}^{4} F(i;\lambda_K)\Bigg), & \\ 0. & \text{Otherwise.} \end{cases} \tag{4.19}$$

Case (iii): If $n = n_{max} - 1$:

$$P(C_{t+1}^K = n'|C_t^K = n) = \begin{cases} \frac{1}{5}\sum_{i=0}^{4} F(i;\lambda_K), & \text{if } n' = n-1 \\ \frac{1}{5}\sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \text{if } n' = n \\ \qquad -F(4-i;\lambda_K)\Big], & \\ 1 - \frac{1}{5}\Bigg(\sum_{i=0}^{4}\Big[F(9-i;\lambda_K) & \text{if } n' = n+1 \\ \qquad\qquad -F(4-i;\lambda_K)\Big] & \\ \qquad + \sum_{i=0}^{4} F(i;\lambda_K)\Bigg), & \\ 0. & \text{Otherwise.} \end{cases} \tag{4.20}$$

Case (iv): If $n = n_{max}$:

$$P(C_{t+1}^K = n'|C_t^K = n) = \begin{cases} \sum_{i=0}^{4} \omega_{4-i} F(i;\lambda_K), & \text{if } n' = n-1 \\ 1 - \sum_{i=0}^{4} \omega_{4-i} F(i;\lambda_K), & \text{if } n' = n \\ 0. & \text{Otherwise.} \end{cases} \tag{4.21}$$

In the case of expression (4.21), the same explanation as for equation (4.10) is relevant. For f_{MK} and f_{KK}, the probability distributions of the closed roads are taken as seen in equations (4.11) - (4.13). A detailed explanation for equation (4.19) is already given when explaining equation (4.14), while expression (4.20) is similar to (4.11). On the other hand, equation (4.18) as a whole was not experienced in the previous expressions considered. In this case, equation (4.18) is for when the queue length category of Kappara is 1. The first case is that of staying in the same queue length category, results in $\frac{1}{5}\sum_{i=0}^{4} F(9-i;\lambda_K)$, where the summation gives the cumulative probability of a Poisson distribution for different arrival counts and the factor in front is an averaging factor. The latter is placed so that an average probability over the five summed terms is given. The expression $9-i$ indicates that the CDF for various arrivals is being evaluated, starting from at most 9 and decreasing to at most 5. The second case is for when the Kappara road increases by one queue length category, where the whole term with the subtraction gives the probability that the number of arrivals is between $9-i$ and $14-i$. Lastly, the third case is the complement, hence the probability that neither of the previous two mentioned scenarios happens and that the queue length category increases by two.

To summarise, when considering f_{SG}, f_{GM},f_{KS}, f_{SS}, f_{GG},and f_{MM} expressions (4.8) - (4.17) are used to compute equation (4.7). Whereas for f_{MK} and f_{KK} expressions (4.18) - (4.21) are used for the open road, while (4.11) - (4.13) are used for the closed roads to compute equation (4.7). Notably, the equations presented in Section 4.4.3, have been formulated by the author specifically for this study.

4.4.4 Rewards

The reward function illustrates numerically the consequence of being in state S_t, taking action $A_{s_t,t}$ and ending in state S_{t+1}. Hence, similarly to what was generally defined in Section 2.2.1, it is defined as follows:

$$R : \mathcal{S}_t \times \mathcal{A}_{s_t,t} \times \mathcal{S}_{t+1} \mapsto \mathbb{R}$$

By taking into consideration the current state and the action to be executed, referred to as the state-action pair, and the next possible state, the function returns a real value. This formulation allows the reward to capture both immediate action costs

and state-dependent performance objectives, such as penalising undesirable states or encouraging transitions towards target regions of the state space. The main attribute of a state, be it the current or next, is the queue length. Since queue length categories are encountered, which are composed of five different queue lengths, a value to help us formulate a reward was needed. The queue length value of each queue length category is expressed as the midpoint of the respective category. This assumption is made to associate one numeric value to a category. Furthermore, each chosen midpoint fairly represents all category values due to the narrow categories under study. The midpoints for each road are denoted by $\tilde{k}$,$\tilde{l}$,$\tilde{m}$ and $\tilde{n}$. For example, the midpoint for $k = 1$ (category $[0 - 4]$) is $\tilde{k} = 2$, while for $k = 2$ (category $[5 - 9]$) it is $\tilde{k} = 7$ and so on. The same goes for $\tilde{l}$,$\tilde{m}$ and $\tilde{n}$.

Various reward functions can be constructed using the queue length. In the upcoming sections, different reward functions are discussed. Firstly, the main reward function used is introduced, which considers road capacity and the influence on the intersection that the current and future states have.

4.4.4.1 Weighted Capacity Reward

For this reward function, the road capacity value is primarily introduced. The road capacity at time t for each road is denoted by $\bar{k}$,$\bar{l}$,$\bar{m}$ and $\bar{n}$. In this study's case, the current capacity of a road is the percentage of queued cars in a road at time t with respect to the maximum capacity of that same road. This value is essential because it shows the true road conditions. For instance, 50 cars in Gżira road are not the same as if they were in Kappara road, due to the fact that the maximum capacity of Gżira is 50 cars, while that of Kappara is of 100 cars. Therefore, in the latter example Gżira's current capacity is that of 100%, while that of Kappara is 50%. In this example, it is clear that although queue lengths may be the same for both roads, they do not have the same effect since one results in a full capacity road, while the other results in a half capacity road.

The maximum capacity for each road, in terms of cars, is 35 for Sliema, 50 for Gżira, 40 for Msida and 100 for Kappara. Hence the capacities are calculated as

follows:

$$\bar{k} = \frac{100\tilde{k}}{35}, \qquad \bar{l} = \frac{100\tilde{l}}{50}, \qquad \bar{m} = \frac{100\tilde{m}}{40}, \qquad \bar{n} = \frac{100\tilde{n}}{100}. \tag{4.22}$$

In addition, the calculations in (4.22) are also done for the next state capacities by replacing k, l, m and n with k', l', m' and n', respectively. The latter are used in equations (4.23) and (4.24).

Apart from the capacity, the reward also considers how far the road is in the cycle. This means that the reward distinguishes between the capacity of a road that will be open next when taking action 'Change' and that of a road that will be open after two or more actions of 'Change'. In order for this feature to be implemented, the use of ranked weights was needed. The weights are defined as $w_1 = 1$, $w_2 = 2$, $w_3 = 3$ and $w_4 = 4$, with such values being chosen to give importance to roads which will be opened much later than others. For instance, consider the current intersection state as 'Sliema-Open' (not delving into queue lengths), and it is decided to take action 'Change', the next intersection state is going to be 'Gżira-Open'. So the farthest road in the cycle becomes Sliema, then Kappara, then Msida, while Gżira is the closest. Thus, the farthest road in the cycle will have w_4 multiplied by its capacity, while the closest will have w_1. The reward function is defined as follows, depending on the action taken:

if $a_t =$ 'Stay the same':

$$R(s_{t+1}|s_t, a_t) = \begin{cases} \frac{(w_1\bar{k}+w_2\bar{l}+w_3\bar{m}+w_4\bar{n})-(w_1\bar{k}'+w_2\bar{l}'+w_3\bar{m}'+w_4\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'S-O'} \\ \frac{(w_4\bar{k}+w_1\bar{l}+w_2\bar{m}+w_3\bar{n})-(w_4\bar{k}'+w_1\bar{l}'+w_2\bar{m}'+w_3\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'G-O'} \\ \frac{(w_3\bar{k}+w_4\bar{l}+w_1\bar{m}+w_2\bar{n})-(w_3\bar{k}'+w_3\bar{l}'+w_1\bar{m}'+w_2\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'M-O'} \\ \frac{(w_2\bar{k}+w_3\bar{l}+w_4\bar{m}+w_1\bar{n})-(w_2\bar{k}'+w_3\bar{l}'+w_4\bar{m}'+w_1\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'K-O'}. \end{cases} \tag{4.23}$$

or if $a_t =$ 'Change':

$$R(s_{t+1}|s_t, a_t) = \begin{cases} \frac{(w_1\bar{k}+w_2\bar{l}+w_3\bar{m}+w_4\bar{n})-(w_4\bar{k}'+w_1\bar{l}'+w_2\bar{m}'+w_3\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'S-O'} \\ \frac{(w_4\bar{k}+w_1\bar{l}+w_2\bar{m}+w_3\bar{n})-(w_3\bar{k}'+w_4\bar{l}'+w_1\bar{m}'+w_2\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'G-O'} \\ \frac{(w_3\bar{k}+w_4\bar{l}+w_1\bar{m}+w_2\bar{n})-(w_2\bar{k}'+w_3\bar{l}'+w_4\bar{m}'+w_1\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'M-O'} \\ \frac{(w_2\bar{k}+w_3\bar{l}+w_4\bar{m}+w_1\bar{n})-(w_1\bar{k}'+w_2\bar{l}'+w_3\bar{m}'+w_4\bar{n}')}{w_1+w_2+w_3+w_4}, & \text{if } I_t = \text{'K-O'}. \end{cases} \tag{4.24}$$

Note that the intersection state $I_t \in \{$'Sliema-Open', 'Gżira-Open', 'Msida-Open', 'Kappara-Open'$\}$ was abbreviated to $I_t \in \{$'S-O', 'G-O', 'M-O', 'K-O'$\}$.

Equations (4.23) and (4.24) consider the weighted capacities of four roads, in the numerator, both in the current and next state. For each road, weights are assigned based on the road's position in the cycle relative to the next opening, where w_1 is for the closest to being opened, w_2 is for the next in line, w_3 is for third in line and w_4 is for the road which is farthest from being opened. As a result, the numerator compares the total weighted capacity of the current state against that of the next state after an action. Such a comparison helps in determining the benefits of the action taken, while taking into consideration which road will be open sooner or later.

Furthermore, the denominator's role in the above equations is that of a normalising factor, whereby it ensures that the reward is scaled properly. Thus, preventing the reward from being excessively large or small in value due to the size of the weights.

4.5 Conclusion

In this chapter, a detailed description of traffic modelling and the various strategies was provided, and thereafter, the AIMSUN program was introduced. The test intersection and its surrounding area were demonstrated together with the respective tuning and input flow rates. Furthermore, the pros and cons of detectors were discussed. The second part of this chapter focused on the MDP formulation for the

problem at hand, thus delving into the applied attributes of an MDP. Chapter 5 illustrates the computational results achieved for a fixed cycle (FC) and an MDP driven cycle.

Chapter 5

Computational Experiments

This chapter provides a comprehensive analysis of the results obtained from the computational experiments. The key performance measures that will assess the proposed approach are first introduced in Section 5.1. Subsequently, in Section 5.2, a sensitivity analysis of the parameters related to the policy iteration algorithm is conducted. Thereafter, a comparison between an MDP driven cycle and an FC for different hours of the day, mainly rush hours, is provided in Section 5.3, which comparison essentially comprises the fulcrum of this chapter. Finally, the dynamics of shifting between optimal policies throughout the day are explored, demonstrating how the system adapts to changing traffic flows in order to maintain optimal performance.

5.1 Performance Measures

In order to quantify the performance of the MDP driven cycle and FC, certain performance measures had to be collected, calculated and analysed. There exist many different measures to evaluate the efficiency of a traffic intersection, however, the ones chosen are deemed as the most necessary and essential for this study. Table 5.1 introduces the four chosen performance measures and a brief description is provided therein.

Measure	Unit	Description
Throughput	veh/hr	The total number of cars exiting the intersection in an hour.
Queue Length	veh	The average queue length, i.e., the average number of cars in a queue.
Waiting Time	s	The average time in seconds a vehicle needs to wait before exiting the intersection.
Number of Stops	#/veh	The average number of stops before exiting the intersection, per vehicle.

Table 5.1: Performance Measures

Each of the measures in Table 5.1 provide particular insights. In the present case, the throughput shows the capacity of the intersection, giving the number of cars that passed through the intersection during a specific time period. A high throughput indicates that the intersection can handle a large volume of vehicles. Moreover, if the throughput between two scenarios is different, while considering the same road parameters, the scenario with the higher throughput is minimising congestion and providing smoother traffic flow. In this study, the throughput presented in the results represents the whole system and is not split by road. This measure is obtained by counting each vehicle that passes over one of the four detectors closest to the intersection. At the end of the one hour simulation, the total number of vehicles that passed through the intersection in an hour, is then extracted.

The next measure, that is the average queue length per road, portrays how efficiently the intersection is working with respect to each road. A large queue results in traffic congestion, more vehicle emissions, and delays. Thus, the queue length is an essential indicator for one to be able to obtain a good insight of the traffic flow. Firstly, the queue length was collected by subtracting the number of exiting cars from the number of entering cars of each road every 10 seconds (this time interval was taken the same as when an action is being taken) and recording

each one. Then an average was taken of all the queue lengths collected per road, which resulted in the average queue length per road. Apart from the importance of this measure to analyse the results, this statistic was fundamental to construct the state of the model, as seen in Section 4.4.1.

Waiting time is the first measure that comes to mind when considering driver satisfaction. Furthermore, it gives a good understanding of the intersection's efficiency, especially when comparing different scenarios. The waiting time is the time spent in the system before exiting the intersection; hence, it includes the time spent traversing from the detector to the stop sign before exiting to the intersection. Waiting more than usual in a traffic intersection creates frustration, aggressive driving and more emissions. Hence, minimising the waiting time significantly improves the road user's experience. Another measure, similar to waiting time but distinct from it, is delay. Even though it is not considered as one of the measures, it is important to highlight this distinction since it is a commonly used measure in the literature. Delay is the extra time spent in the system before exiting the intersection. Hence, the difference between waiting time and delay is that the former includes the traversal time to reach the stop sign before exiting to the intersection, while the latter does not.

The waiting time was collected by recording the time each car passed over the detector at the end of the road and the time that same car entered the system, and then subtracting the latter from the former to obtain the waiting time of the respective car. Since this was done for each car at the different roads, the average waiting time for each road was then calculated.

Lastly, the number of stops per vehicle indicates how smoothly traffic is moving through the roads leading to the intersection. The fewer stops a car experiences leading to the intersection, the more flowing the traffic is, which in turn, increases user satisfaction and reduces emissions and wear and tear of vehicles. Such a measure is important to assess the efficacy of traffic signal timings. The number of stops per vehicle was achieved by registering the number of times a vehicle experiences a red light. Then, as a result, the average number of stops for each road was computed.

These measures together provide a thorough evaluation of the traffic intersection,

which aids in identifying the areas that improved or require improvement. Since the MDP driven cycle is achieved from optimal policies derived from the policy iteration algorithm, a sensitivity analysis of the algorithm's parameters is performed in Section 5.2.

5.2 Sensitivity Analysis

The discount factor γ is essential to provide balance between immediate and future rewards, as seen in Chapter 2. Its selection can significantly affect the performance of the policy iteration algorithm. Therefore, a sensitivity analysis of γ is important so that the possible impact of the convergence rate, policy stability and overall efficiency of the algorithm are understood. This section analyses how a change in γ can affect the algorithm's outcomes, specifically, how different γ's can lead to different optimal policies. By presenting some plots, this section aims to identify the most suitable discount factor for the problem, which provides stability and sensitivity to long-term rewards.

Firstly, the time (in seconds) and iterations taken for an optimal policy to be found using different discount factors are analysed. The discount factors being tested throughout this section are $\gamma \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$. These selections were made to capture most of the range of values a discount factor can have. Figure 5.1 illustrates that $\gamma = 0.9$ results in an iteration count of 43 and 18 minutes of running time, which, when compared to the other γ's, is significantly high. Thus, $\gamma = 0.9$ can be seen as exaggerated when compared to the other γ's in relation to time and iterations. The higher the discount factor in Figure 5.1, the higher the iteration count, but the same cannot be said for the time taken. Although $\gamma = 0.1$ required the fewest iterations to find an optimal policy, it needed more time than $\gamma = 0.25$ and the same as $\gamma = 0.5$. Hence, it can be deduced that $\gamma = 0.9$ and $\gamma = 0.1$ are already the least favourites to be chosen due to the characteristics mentioned, based on Figure 5.1.

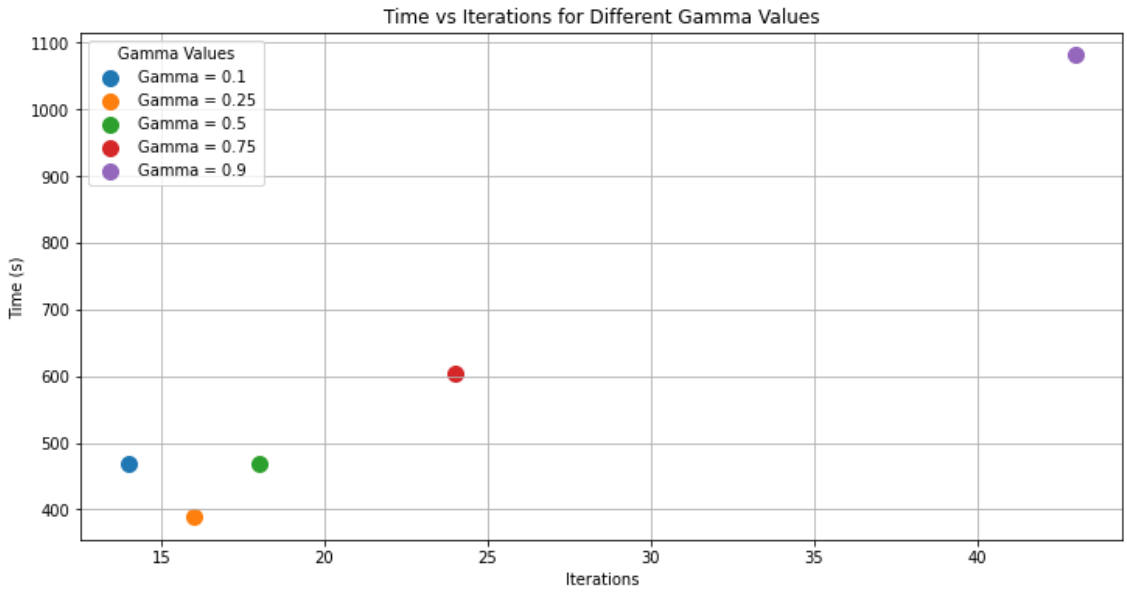

Figure 5.1: Time v. Iterations for different γ's

Even though Figure 5.1 provides a good insight, it is not sufficient by itself to choose the discount factor. As a consequence, the value function convergence for each discount factor is also compared in Figure 5.2. As can be seen, $\gamma = 0.1$ and $\gamma = 0.25$ lead to relatively stable value functions, whereby fluctuations are minimal and the value function does not vary much. Such a stable plot suggests that the model focuses on immediate rewards, which is ideal when such rewards are much more important than future ones. For $\gamma = 0.5$, the figure also shows a stable convergence, but with a bit more fluctuations than the lower discount factors. This discount factor provides a balance between immediate and future rewards.

On the other hand, for $\gamma = 0.75$ and $\gamma = 0.9$, the value function is much more fluctuating. This indicates that the model is becoming more sensitive to future rewards, while exploring a wider range of value functions. Hence, the higher the discount factor, the less the stability. Such discount factors are opted for when future rewards are nearly as important as immediate rewards.

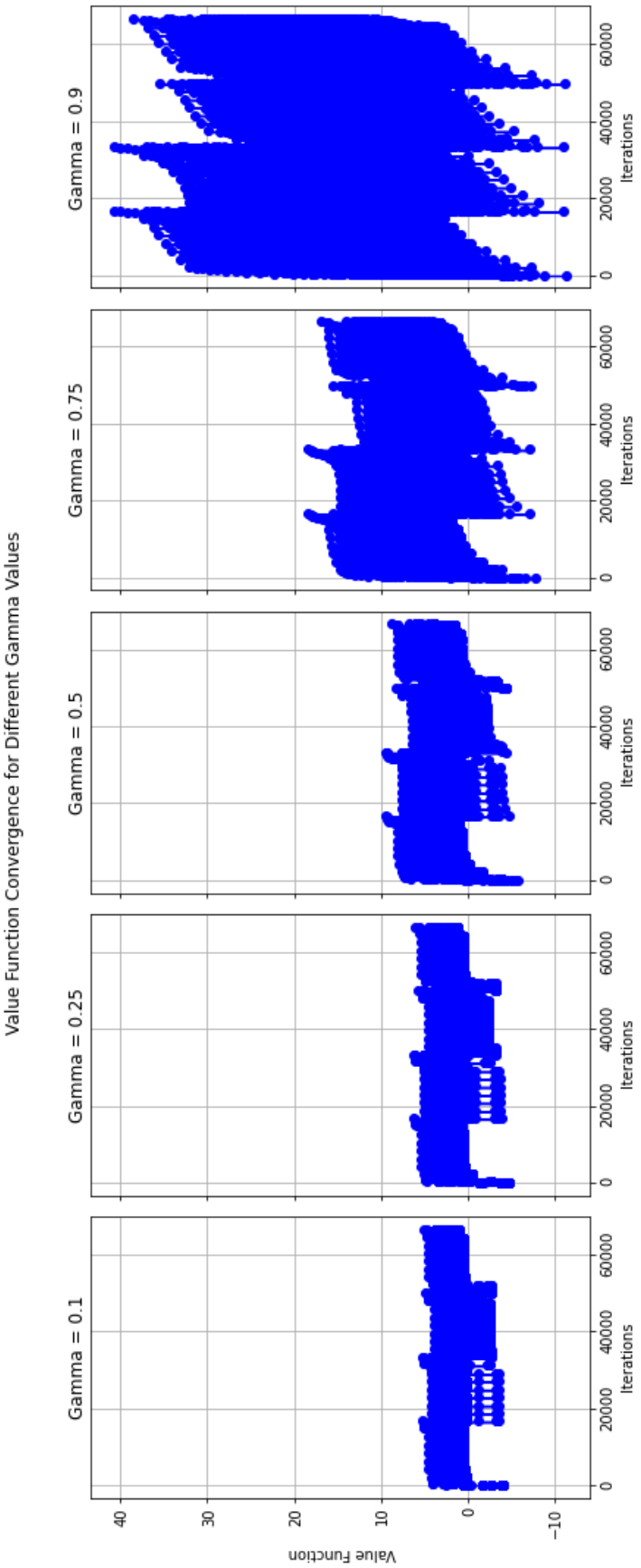

Figure 5.2: Convergence Rates of the Value Function for varying discount factors γ, highlighting the impact of the discount factor on the speed and stability of the convergence process.

The discount factors 0.1 or 0.25 are ideal if one needs a stable and consistent policy that converges quickly. Nonetheless, if the problem at hand necessitates giving importance to future rewards, a discount factor closer to 0.75 or 0.9 is more appropriate, keeping in mind that the model can tolerate some instability. In this study's case, future traffic conditions are important; thus, a discount factor greater than 0.25 is definitely needed.

Figure 5.3 is another illustration that can help to finalise the choice of the discount factor. This figure shows how the optimal policy achieved is structured by the two actions present in the problem. Figure 5.3 further rules out $\gamma = 0.9$, since the optimal policy changes drastically, compared to the other γ's. For $\gamma = 0.9$, the amount of actions 'Change' would result in an unstable intersection, one which would change priority more frequently. The remaining discount factors are all relatively similar. With this figure again confirming what the discount factor range should be, one is left to decide between two discount factors, $\gamma = 0.5$ and $\gamma = 0.75$.

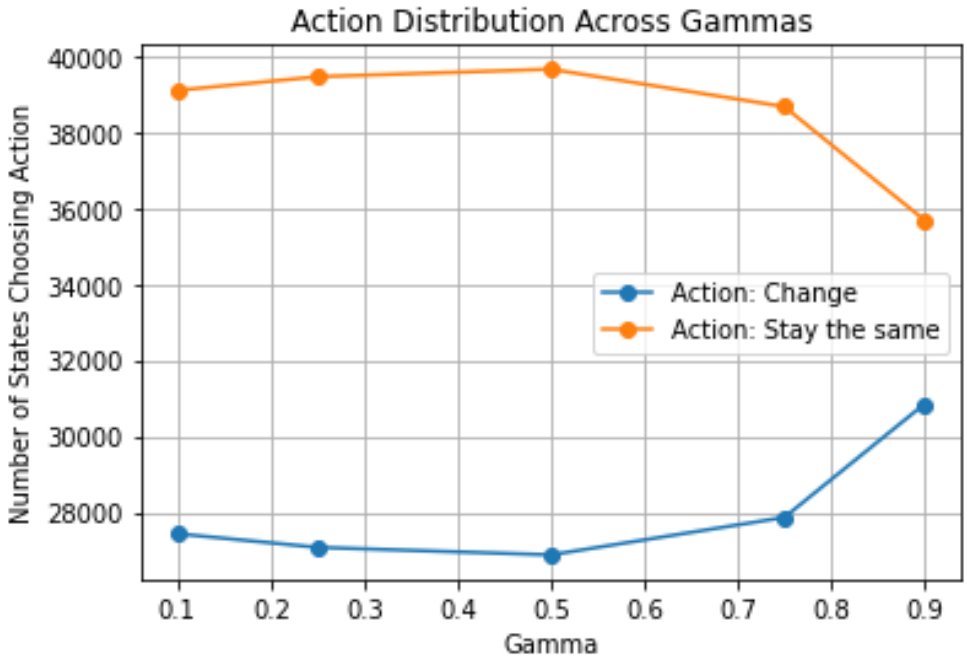

Figure 5.3: Distribution of Optimal Policy Actions for varying discount factors γ, illustrating the effect of long-term versus short-term reward considerations on the action selection process.

A discount factor of 0.5 will place more importance to immediate rewards, leading to decisions catered for optimising short term traffic flow. This latter γ may fail to

consider long term congestion. On the contrary, $\gamma = 0.75$ allows the algorithm to consider the future effects of current decisions without placing too much importance on immediate rewards. The latter is crucial in traffic management, since changing traffic light timings can have sequential effects on the flow and congestion levels further down the line. Based on the latter reasoning and Figures 5.1, 5.2 and 5.3, the discount factor chosen is $\gamma = 0.75$.

Now that a discount factor for the policy iteration algorithm has been chosen, the algorithm with the defined parameters is applied to the traffic management application. The results comparing a normal approach of dealing with traffic light timings, FC, and the dynamic approach, MDP driven cycle, are given in Section 5.3.

5.3 Fixed Cycle vs. MDP Driven Cycle

In this section, the results for the different scenarios (times of day), mentioned in Section 4.3.3, are provided. These scenarios are; morning rush hour, noon rush hour, evening rush hour and midnight hour. The author obtained the morning, noon, evening and midnight policies, so that they can be used through an API in the Aimsun simulator, by using Python to define states, actions, transition probabilities, rewards and related algorithms. This code can be found in Appendix A. Furthermore, a separate code was developed on Python, in order to interact with Aimsun through an API, thereby allowing the allocation of green time at the intersection according to the selected policy. The aforementioned code can be found in Appendix B. The results demonstrated throughout this chapter were obtained using several Monte Carlo runs. For each scenario, 100 Monte Carlo runs were done with the results shown in this section reflecting the average of the 100 simulations. Such a technique makes use of the randomness of each run to obtain a more comprehensive perspective of the network. The number of Monte Carlo runs was set at 100, as this did not have an impact on the computational time, while at the same time providing good results. Furthermore, the results, containing the measures discussed in Section 5.1, will be split for the four roads that make up the intersection under study. However, even a general overview of the intersection as a whole will be provided.

5.3.1 Morning Rush Hour

The average results for the simulations of the morning rush hour are provided in Table 5.2. The table consists of the different metrics that will be used to analyse the two approaches, with three of the metrics being separated by road. In Table 5.2 and the similar tables that will be seen in the following sections,"Impr." refers to improvement, and a value is placed there if there is an improvement from an FC to an MDP driven cycle in that particular metric.

Table 5.2 shows that for an MDP driven cycle during the morning rush hour, a vehicle on average waited much less than for an FC; around 7 minutes less on the Kappara road, while waiting around 7 minutes more than for an FC on the Gżira road. In this case, the Gżira road's performance deteriorated slightly with an MDP driven cycle, this happened since the Gżira road affects the least number of commuters in this scenario. Hence, the MDP driven cycle decided to give less green time to the road from which vehicle exits are the lowest, which resulted from balancing out the four roads. Additionally, the average waiting time for the MDP driven cycle in Msida decreased by half a minute, while in Sliema it increased by two and a half minutes when compared to an FC. The overall intersection average waiting time for an MDP improved by more than a minute and a half when compared to the FC, with the MDP driven cycle giving more balanced waiting times to the different roads that make up the intersection.

It is important to note that the overall average waiting time is the weighted average of the four roads, based on the throughput of each road, this is also relevant to the other metrics. Similarly, the average queue length improved for two of the roads while increased for the remaining. Overall, the average queue length did not improve but it almost remained the same. A positive remark on the queue length is that for an MDP driven cycle the standard deviation of the queue length values is much less and more balanced between roads when compared to that of an FC. This is also evident in the overall standard deviation, which is the square root of the weighted pooled variance.

On the other hand, the remaining metrics evidently show an improvement, where the overall average number of stops per vehicle was reduced to more than one stop.

Therefore, with an FC, a vehicle on average stops more than one time more, than with an MDP driven cycle. The most evident improvement, and one which affects the other metrics, is that of the average throughput, where an MDP driven cycle resulted in dealing with 121 (8.51%) more vehicles than an FC for the same amount of time.

Morning Rush Hour															
	Average Waiting Time			Standard Deviation of Average Waiting Time		Average Queue Length			Standard Deviation of Average Queue Length		Average Number of Stops			Standard Deviation of Average Number of Stops	
Traffic Origin	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP
Sliema	439.01	592.69	-	94.87	156.67	21.87	23.46	-	2.54	1.98	3.47	1.68	1.79	0.42	0.25
Gżira	52.75	501.48	-	12.97	66.85	18.16	15.48	2.68	6.18	1.09	0.34	2.40	-	0.12	0.34
Msida	148.99	114.81	34.18	68.27	12.62	2.33	20.35	-	0.62	2.88	1.17	0.34	0.83	0.48	0.07
Kappara	712.91	321.06	391.85	94.87	60.53	56.85	50.47	6.38	1.49	2.33	3.70	1.08	2.62	0.14	0.14
Overall	405.81	305.44	100.37	282.69	181.03	28.53	33.70	-	24.02	15.22	2.43	1.02	1.41	1.39	0.63
	Average Throughput														
	FC					MDP					Impr.				
	1422					1543					121 (8.51%)				

Table 5.2: Morning Rush Hour FC and MDP Driven Cycle Average Results for Waiting Time, Queue Length, Number of Stops, and Throughput

Due to the fact that, on average, 121 more vehicles exited the intersection for an MDP driven cycle, the other metrics in Table 5.2 are affected. This depends on the specific traffic conditions at each location, mainly, the average waiting time. All metrics presented for an FC are being calculated on 1422 vehicles, while for an MDP driven cycle on 1543 vehicles. This increase in throughput in the Sliema and Gżira roads, where the average waiting time increased under the MDP cycle, resulted in shorter green time and, consequently, vehicles had to wait longer for these roads. In general, the overall average waiting time is more than a minute and a half better. This has been attained by the system prioritising the Kappara road and marginally the Msida road.

As a result the weighted average of the waiting times shows that overall an MDP driven cycle improved average waiting time. This indicates that the MDP driven cycle was more effective when it comes to balancing the traffic flow. Therefore, an MDP driven cycle optimized the signal timings better than an FC, allowing significantly more vehicles to pass through without causing excessive delays.

5.3.2 Noon Rush Hour

As opposed to the morning rush hour scenario, the noon rush hour scenario, depicts a much clearer picture of the improvement obtained with an MDP driven cycle. Even at first sight, Table 5.3 shows the overall intersection improvement for all the metrics considered. The overall average waiting time when using an MDP driven cycle was, on average, about more than a minute and a half less than for an FC. Furthermore, the overall average queue length remained almost the same, while the overall average number of stops for a MDP driven cycle is almost two less than for an FC. The average throughput once again shows that an MDP driven cycle handles more vehicles than an FC, specifically in this case 34 (2.23%) vehicles.

Noon Rush Hour															
	Average Waiting Time			Standard Deviation of Average Waiting Time		Average Queue Length			Standard Deviation of Average Queue Length		Average Number of Stops			Standard Deviation of Average Number of Stops	
Traffic Origin	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP
Sliema	544.90	783.47	-	78.64	126.07	25.26	26.07	-	0.92	0.84	1.95	1.37	0.58	0.11	0.17
Gżira	610.15	475.99	134.16	78.77	41.83	30.24	21.30	8.94	0.87	0.88	5.07	2.09	2.98	0.38	0.12
Msida	642.67	174.11	468.56	82.39	11.75	31.47	32.45	-	2.11	1.55	3.06	0.70	2.36	0.13	0.07
Kappara	474.64	500.74	-	80.08	75.31	52.93	55.01	-	3.79	1.79	3.45	2.08	1.37	0.29	0.30
Overall	550.02	444.42	105.6	104.35	222.61	37.40	37.31	0.09	12.25	13.50	3.17	1.52	1.65	1	0.65
	Average Throughput														
	FC					MDP					Impr.				
	1526					1560					34 (2.23%)				

Table 5.3: Noon Rush Hour FC and MDP Driven Cycle Average Results for Waiting Time, Queue Length, Number of Stops, and Throughput

Overall, the comparison of the two different strategies during the noon rush hour indicates significant improvement for an MDP driven cycle. However, across different locations there is a mixture of improvements and slight non-improvements, the former being in the majority. For instance, in the Sliema road, an MDP driven cycle led to a 4 minute extra of waiting time, on average, when compared to an FC. However, this disadvantage is a consequence of having a generally more efficient intersection. A specific road, in this case, Sliema, became more congested. Nevertheless, this was imperative to achieve improvements at the intersection as a whole.

Conversely, in the Gżira road, the MDP driven cycle significantly improved traffic conditions, reducing the average waiting time by above two minutes. Moreover, the average queue length slightly improved and the average number of stops reduced by almost 3. This means that a vehicle arriving in the Gżira road stops 5 times on average for an FC and only 2 times on average for an MDP driven cycle. Evidently here the MDP driven cycle is managing to reduce the number of stops for a vehicle, which will reduce frustration at the intersection. Similar remarks can be made on the Msida road, where significant improvements were made in all performance measures, except the average queue length which remained almost the same. For the Kappara road the results were slightly mixed, with the average waiting time increasing by around 30 seconds, the average queue length remaining the same, while the stops decreased by almost one and a half, on average. Hence, a smoother traffic flow was also experienced in Kappara with an MDP driven cycle.

After analysing the results, it is unmistakable that the increase in the average waiting time in the Sliema and Kappara roads serves a greater purpose. The latter enhances the overall efficiency of the entire intersection, which shows in the significant improvement of the overall values. Even though some vehicles may experience more waiting time, overall, when compared to an FC, this calculated trade-off is far more beneficial when taking into consideration the broader context. In general, the MDP approach for the noon rush hour is prioritising the overall traffic efficiency by taking calculated risks at specific roads. Hence, ensuring that the majority of drivers are benefiting from smoother and faster travelling through the intersection.

5.3.3 Evening Rush Hour

In this subsection, the evening rush hour results, displayed in Table 5.4, will be analysed. Table 5.4 illustrates a clear advantage for the MDP driven cycle in all performance measures, hence proving further its overall superiority in optimising traffic signal timings. For the Sliema road, the MDP approach reduced the average waiting time by 47 seconds, from 354 seconds under FC to 307 seconds, while also decreasing the average queue length and the average number of stops. In this case, the Gżira road with the MDP strategy resulted in an increase in the average waiting time by around two minutes. However, the MDP approach significantly reduced the average queue length by seven vehicles and only increased the average number of stops by half a stop per vehicle. This indicates that although some vehicles might wait longer and make slightly more stops, the overall traffic flow efficiency is still tolerable. The Msida and Kappara roads saw a significant improvement with the MDP driven cycle for the average waiting time and average number of stops metrics. These roads both had the average waiting time with an MDP approach decrease by more than a minute, while stopping one time less than using an FC.

In conclusion, the MDP-driven cycle for the evening rush hour demonstrated an all round improvement for all performance measures considered, with the average waiting time of a vehicle in the intersection being 40 seconds less, one less vehicles in an average queue, and almost one stop less on average for every vehicle. Finally, the throughput further confirms the efficiency of the MDP approach, since, on average, 37 (2.42%) more cars exited the system when compared to an FC. Such improvements highlight the efficiency of the MDP driven cycle, which ultimately lead to better traffic management.

Evening Rush Hour															
	Average Waiting Time			Standard Deviation of Average Waiting Time		Average Queue Length			Standard Deviation of Average Queue Length		Average Number of Stops			Standard Deviation of Average Number of Stops	
Traffic Origin	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP
Sliema	353.91	306.83	47.08	74.20	106.79	23.63	20.87	2.76	1.51	2.45	1.78	1	0.78	0.17	0.22
Gzira	184.95	332.36	-	82.59	78.62	17.49	10.44	7.05	6.96	1.79	1.72	2.27	-	0.87	0.49
Msida	190.70	103.92	86.78	86.95	17.65	10.63	18.39	-	5.14	4.42	1.61	0.48	1.13	0.73	0.11
Kappara	313.33	237.37	75.96	76.05	12.44	44.21	39.56	4.65	6.81	1.75	2.76	1.61	1.15	0.52	0.16
Overall	278.07	238.07	40	106.06	103.19	27.07	25.85	1.22	14.64	11.21	2.08	1.26	0.82	0.77	0.62
	Average Throughput														
	FC 1529					MDP 1566					Impr. 37 (2.42%)				

Table 5.4: Evening Rush Hour FC and MDP Driven Cycle Average Results for Waiting Time, Queue Length, Number of Stops, and Throughput

5.3.4 Midnight Hour

As will be seen in Table 5.5, and contrary to what was seen in the previous subsections, the results for a low traffic flow differ from that of a busy intersection. Thus, the traffic signal approaches, that of an FC and an MDP driven cycle, demonstrate a reversal of trends during the midnight hour. In this low traffic intensity scenario, the FC approach consistently outperforms the MDP approach, basically in almost all performance metrics.

The results in Table 5.5 suggest that in low-traffic conditions, an FC is significantly better than an MDP driven approach. This can be seen in both the overall average waiting time and the overall average queue length. For an FC approach, in this scenario, a vehicle waits on average around 30 seconds less. Additionally, on average, a queue under an FC approach will consist of almost a vehicle less. On the other hand, the average number of stops and throughput are almost the same for the two approaches.

The rationale behind this shift is due to the nature of traffic during non-rush hours. When experiencing low traffic volumes, adaptive control becomes unnecessary, and the simplicity of an FC becomes beneficial. An FC approach provides a predictable and consistent timing, while the MDP approach may introduce inefficiencies during low traffic volumes due to its adaptive behaviour. Thus, as a result, for a low traffic volume scenario, an FC approach outperforms an MDP driven cycle.

Midnight Hour															
	Average Waiting Time			Standard Deviation of Average Waiting Time		Average Queue Length			Standard Deviation of Average Queue Length		Average Number of Stops			Standard Deviation of Average Number of Stops	
Traffic Origin	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP	FC	MDP	Impr.	FC	MDP
Sliema	22.36	12.27	-	5.45	5.12	0.48	0.27	0.21	0.08	0.04	0.12	0.07	0.05	0.04	0.02
Gżira	24.92	62.24	-	8.47	10.05	0.34	0.82	-	0.08	0.23	0.17	0.15	0.02	0.05	0.04
Msida	21.66	56.24	-	19.27	21.48	0.46	1.12	-	0.08	0.21	0.14	0.13	0.01	0.48	0.06
Kappara	35.70	95.46	-	9.91	11.77	1.17	3.04	-	0.14	0.23	0.29	0.34	-	0.06	0.07
Overall	27.02	59.03	-	12.56	33.03	0.65	1.41	-	0.36	1.12	0.19	0.18	0.01	0.22	0.12
	Average Throughput														
	FC					MDP					Impr.				
	316					312					-				

Table 5.5: Midnight Hour FC and MDP Driven Cycle Average Results for Waiting Time, Queue Length, Number of Stops, and Throughput

The scenarios analysed in Sections 5.3.1, 5.3.2 and 5.3.3 are showing the problem in separate silos. Although the latter still highlights the improvements an MDP driven cycle offers, traffic congestion is not confined to a single hour before disappearing. Instead, traffic congestion is a constant challenge throughout the day, with morning, noon, and evening traffic conditions affecting each other rather than existing independently. For this reason, a simulation connecting the morning, noon, and evening scenarios was created on Aimsun and will be delved into in Section 5.4.

5.4 Combined Scenarios - 9 Hour Simulation

This section offers a more realistic approach wherein the policy to be chosen is based on real-time metrics. Hence, rather than using a predetermined policy for a specific time period, a policy is chosen, out of a pool of policies, based on the real time conditions of the intersection. As a result, a policy that best represents the traffic condition at that particular time is chosen. On the other hand, in the previous sections, policy selection was strictly time-based; where the morning policy was applied to the morning scenario, the noon policy to the noon scenario, and the evening policy to the evening scenario.

In order for this to materialise, the three scenarios which are, the morning, noon and evening scenarios, had to be combined together in one simulation, where an amendment was done to the code in Appendix B to achieve the code in Appendix C, which was developed to accommodate the 9 hour simulation and its respective results. In the present case, a 9-hour simulation was developed, which simulation was split equally between the three times of day, hence 3 hours each. The times of day transition from one to another, respectively. It is important to note that while this approach is more realistic since it does not assume a single scenario, it is actually more intense than real-life morning, noon, and evening scenarios. The latter scenarios would also incorporate low intensity periods in between themselves but for reasons of complexity, these interim low-intensity periods were not considered.

One notable difference from the previous approach concerns the fact that the vehicle counts had to be triplicated to cater for the 3-hour simulation of each time of day. Furthermore, whereas before the policy was assigned to that specific hour

with no possibility of alternating policies, this present approach periodically decides which policy to apply. Every 5 minutes, the system chooses a policy based on a comparative analysis of expected and real-time metrics from the previous 5 minutes. However, it must be noted that the first policy chosen is always the morning policy, this being the first time of day. This pre-determined decision has to be made since there is no previous 5-minute metrics to be compared. The 5-minute period was chosen due to its consistency (frequent enough to be reliable but not sporadic). The policy chosen is then used until a different one is chosen, with an evaluation being made every 5 minutes as to whether the current policy is to be maintained or changed to a different one.

5.4.1 Measure for Policy Choice

Primarily, arrival rates were utilised as the main policy evaluation measure in the comparative analysis. This policy evaluation measure was used to determine which policy is most suitable for the particular simulation scenario being analysed. Thus, the results which will be shown hereunder; are a reflection of this measure's influence. Notwithstanding this, other evaluation approaches were tested, such as queue lengths and a combination of both arrival rates and queue lengths. However, the one that produced the best results was adopted, i.e., the arrival rates.

The comparison of arrival rates included expected and real-time values. The former were achieved by taking the three sets of average arrival rates from each of the 1-hour simulations of Sections 5.3.1, 5.3.2 and 5.3.3, respectively. These were taken because they best represent the respective traffic scenarios. The real-time values were achieved by recording the arrival rates every 10 seconds during the simulation. When 5 minutes elapse, the system averages the arrival rates which were recorded every 10 seconds within that 5 minute period. Thereafter, for a policy to be chosen, the expected arrival rates and real-time arrival rates are compared using a distance metric, with different distance metrics being tested.

5.4.2 Distance Metrics

The Euclidean, Manhattan and Chebyshev distance metrics, together with their weighted counterparts, were tested so that the most suitable distance metric for this approach is identified. The weighted counterparts of these distance metrics refer to the same exact metrics but modified by applying weights to the different traffic origins. This was done to give more importance to roads which contain more vehicles than others. The results demonstrated that the Euclidean and Chebyshev distance metrics yielded the best outcome, as evidenced by the results hereunder. It is to be noted that all possible combinations of distance metrics together with policy evaluation measures were evaluated to identify the overall best combination. Consequently, the Euclidean and Chebyshev distance metrics will be defined hereunder.

The Euclidean distance (L_2) gives the shortest possible distance between two points in a space, while the Chebyshev distance (L_∞) measures the maximum difference between comparable coordinates of two points. In this study's case, the two points are 4-dimensional vectors, where each dimension represents the arrival rate of one of the four traffic origins. The two points denote the expected arrival rates and the real time arrival rates, respectively. Before giving general definitions for the Euclidean and Chebyshev distances, for two points in an 4-dimensional space the following is considered, $\mathbf{E} = (e_1, e_2, e_3, e_4)^T$ and $\mathbf{R} = (r_1, r_2, r_3, r_4)^T$, representing the expected arrival rates and the *real-time* arrival rates, respectively. The Euclidean distance is defined as:

$$L_2(\mathbf{E}, \mathbf{R}) = \sqrt{\sum_{i=1}^{n} (r_i - e_i)^2}. \tag{5.1}$$

Chebyshev's distance is defined as:

$$L_\infty(\mathbf{E}, \mathbf{R}) = \max_{i=1}^{n} |r_i - e_i|. \tag{5.2}$$

5.4.3 Scenarios Considered and Results

Similar to Section 5.3, the results for an FC are presented in this section, but for a 9 hour simulation. Further to this, deterministic results will also be shown and denoted by "Det.". The deterministic scenario, as the name suggests, uses the morning policy for the first 3 hours, then the noon policy for the second 3 hours,

and then the evening policy for the last 3 hours. The latter scenario was included since it provides a benchmark for evaluation, hence enabling the author to analyse the efficiency of the dynamic approach. By monitoring how the system performs under a deterministic approach, one can grasp the impact of adaptability, and even conclude whether the dynamic approach leads to any improvements.

Finally, the results for a dynamic scenario will be demonstrated; these do not only comprise of one set of results, but two, both using arrival rates as a policy evaluation measure. The difference between the two dynamic sets of results is that one uses the Euclidean distance metric, while the other uses the Chebyshev distance metric, shown in expressions (5.1) and (5.2) respectively. The former will be denoted by "Eucl.", while the latter by "Cheb.". Therefore, altogether four sets of results are going to be demonstrated for the 9-hour simulation, showing the performance measures defined in Table 5.1. Additionally, for the dynamic approaches, the selection frequency will also be demonstrated. Such a measure is an account of how many times a particular policy (morning, noon, evening) was chosen. Similar to prior results, for each scenario, 100 Monte Carlo runs were done with the results shown in this section reflecting the average of 100 simulations.

Table 5.6 summarises two performance indicators, namely Average Throughput and Average Waiting Time (in minutes). The Average Throughput is the average of 100 simulations for each scenario. A throughput improvement, if any, is also illustrated, this being based on the FC and on one of the dynamic scenarios (depending on the better out of the two). Hence if the throughput value of a dynamic scenario is larger than that of an FC, this will be registered as an improvement. The latter is shown in the number of vehicles and a percentage over the FC scenario. Although in Section 5.3 the Average Waiting Time was presented in seconds, in this section, it is presented in minutes for ease of interpretation due to the time increase in the simulation. It is worth noting that the overall value at the end of the table for the Average Waiting Time, is the weighted average of the four traffic origins. Additionally, the standard deviation of the Average Waiting Time is also provided, to evaluate the consistency of each approach across the traffic origins. Additionally, the overall standard deviation is also calculated by evaluating the square root of the pooled variance.

9 Hour Simulation									
Overall	Average Throughput								
	FC	Det.	Eucl.	Cheb.	Impr.				
	12708	13310	13312	13313	605 (4.76%)				
	Average Waiting Time (in minutes)					Standard Deviation of Average Waiting Time			
Traffic Origin	FC	Det.	Eucl.	Cheb.	Impr.	FC	Det.	Eucl.	Cheb.
Sliema	63	85	71	71	-	4	7	6	6
Gżira	46	11	10	10	**36**	3	1	1	1
Msida	28	2	2	2	**26**	3	0.3	0.4	0.4
Kappara	85	63	69	69	**16**	3	7	6	6
Overall	60	42	41	41	**19**	23	34	33	33

Table 5.6: Results relating to Overall Average Throughput and Average Waiting Time for the 9 hour simulation of FC, Deterministic, and Dynamic approaches

The throughput in Table 5.6 demonstrates the significant improvement of a dynamic scenario when compared to an FC. In particular, a dynamic scenario makes the intersection support 605 more vehicles than an FC for the 9-hour simulation, which shows how effective a dynamic scenario is when it comes to this policy evaluation measure. Furthermore, the overall average waiting time improvement reconfirms the effectiveness of the dynamic scenario when compared to the FC.

Although the Sliema traffic origin does not show an improvement, the 8 minute setback introduced an improvement in the other traffic origins. Specifically, there was an improvement of 36, 26, and 16 minutes in Gżira, Msida and Kappara roads, respectively. This resulted in an overall average improvement of 19 minutes for a dynamic approach over an FC. It is important to note that the average waiting times for Kappara and Sliema are still considered relatively high; however, one has to keep in mind that in real-life implementation, low intensity traffic is experienced between high intensity traffic periods. Thus, the traffic build up is unlikely to be this consistent and exaggerated. Notwithstanding this, the results are proof of concept,

meaning that this approach works. The standard deviation of the latter metric is also presented to show that the system is both efficient and predictable.

The remaining performance metrics are presented in Table 5.7, where one can see the average number of stops and average queue lengths together with their respective standard deviations for the 100 simulations performed. Moreover, the overall values for these metrics are also presented.

9 Hour Simulation									
	Average Number of Stops					Standard Deviation of Average Number of Stops			
Traffic Origin	FC	Det.	Eucl.	Cheb.	Impr.	FC	Det.	Eucl.	Cheb.
Sliema	2.88	1.66	1.60	1.58	**1.3**	0.09	0.09	0.08	0.08
Gżira	5.56	2.81	2.80	2.81	**2.76**	0.24	0.09	0.10	0.10
Msida	2.02	0.51	0.50	0.50	**1.52**	0.38	0.12	0.12	0.12
Kappara	4.24	1.74	1.73	1.75	**2.51**	0.08	0.09	0.07	0.07
Overall	3.5	1.53	1.50	1.52	**2**	1.21	0.75	0.75	0.76
	Average Queue Length					Standard Deviation of Average Queue Length			
Traffic Origin	FC	Det.	Eucl.	Cheb.	Impr.	FC	Det.	Eucl.	Cheb.
Sliema	28	28	28	28	-	0.3	0.3	0.3	0.3
Gżira	26	21	21	21	**5**	1.9	0.2	0.5	0.5
Msida	27	34	34	34	-	0.3	0.5	0.6	0.7
Kappara	62	59	59	60	**3**	0.2	0.5	0.4	0.4
Overall	40	40	40	40	-	16.88	15.15	15.15	15.62

Table 5.7: Results relating to Average Number of Stops and Average Queue Length for the 9-hour simulation of FC, Deterministic, and Dynamic approaches

By utilising a dynamic scenario, an evident improvement was observed across all traffic origins in the average number of stops, during the 9-hour simulation. In Sliema, the dynamic scenario showed that, on average, slightly more than 1 stop is avoided when compared to an FC scenario. The same can be said for the other traffic origins; almost 3 less stops in Gżira, 1 and a half less stops in Msida, and 2 and a half less stops in Kappara. Generally, vehicles passing through the intersection under a dynamic scenario, experienced an average of 2 less stops, than an FC. The standard deviation of this metric indicates that the values do not vary by much, suggesting that there was a consistent number of stops.

The results of the average queue length remained unchanged overall; however, there were slight changes for the different traffic origins. This continues to indicate that the dynamic scenario processes vehicles more quickly, thereby reducing the number of stops without allowing longer queues. The average queue length primarily remained unchanged due to the consistent high intensity traffic experienced in the 9 hour simulation. This produces an average queue length which is close to the maximum of each traffic origin. If the detectors had been placed farther apart, there might have been an improvement in the average queue length. However, this demonstrates a very common limitation in traffic control problems which was already discussed in Section 4.3.2, known as saturation. Saturation is when the traffic intensity at an intersection exceeds its capacity, leading to traffic jams and delays. This takes place when the number of vehicles coming into the intersection surpasses the traffic control system's limit.

It is pertinent that one considers which policies were chosen and when. Tables 5.8, 5.9 and 5.10 represent the policy choices for each respective time period. The tables all portray the policy selected every 5 minutes for both dynamic approaches considered. Moreover, the table presents the most chosen and the second most chosen policy in the 100 simulations, denoted as (1) and (2), respectively. For ease of interpretation, 'M', 'N' and 'E' denote the morning, noon and evening policies, respectively. Blank spaces in the second most chosen policy column, denoted by (2) indicate that, for all 100 simulations in that specific time interval, the policy listed as the most chosen policy, denoted by (1), is the only policy selected. It is important to note that no transitional period (cooling down and running in of

vehicles) is considered between the different time periods.

Policy Choices - Morning Time									
Time (mins)	Eucl. (1)	Eucl. (2)	Cheb. (1)	Cheb. (2)	Time (mins)	Eucl. (1)	Eucl. (2)	Cheb. (1)	Cheb. (2)
5	M	N	M	N	95	M	-	M	-
10	M	E	M	E	100	M	-	M	-
15	M	-	M	N	105	M	-	M	-
20	M	-	M	-	110	M	-	M	-
25	M	-	M	-	115	M	-	M	-
30	M	-	M	-	120	M	-	M	-
35	M	-	M	-	125	M	-	M	-
40	M	-	M	-	130	M	-	M	-
45	M	-	M	-	135	M	-	M	-
50	M	-	M	-	140	M	-	M	-
55	M	-	M	-	145	M	-	M	-
60	M	-	M	-	150	M	-	M	-
65	M	-	M	-	155	M	-	M	-
70	M	-	M	-	160	M	-	M	-
75	M	-	M	-	165	M	-	M	-
80	M	-	M	-	170	M	-	M	-
85	M	-	M	-	175	M	-	M	-
90	M	-	M	-	180	M	-	M	-

Table 5.8: First (1) and Second (2) Preference Policy choices for the first three hours of the 9-hour simulation

Table 5.8 indicates that the most chosen policy throughout the first 3 hours of the 9-hour simulation is the morning policy, for both dynamic approaches. This suggests that the dynamic approaches are taking the morning policy during morning time. Notwithstanding this, during the first 15 minutes of some simulations, other policies were also selected. The latter variation can be attributed to the fact that, at the beginning of the simulation, traffic conditions are still stabilising while slowly forming to the morning traffic pattern.

Policy Choices - Noon Time									
Time (mins)	Eucl. (1)	Eucl. (2)	Cheb. (1)	Cheb. (2)	Time (mins)	Eucl. (1)	Eucl. (2)	Cheb. (1)	Cheb. (2)
5	M	-	M	-	95	M	N	M	N
10	M	-	M	-	100	N	M	M	N
15	M	-	M	-	105	N	M	N	M
20	M	-	M	-	110	N	M	N	M
25	M	-	M	-	115	N	M	N	M
30	M	-	M	-	120	N	M	N	M
35	M	-	M	-	125	N	M	N	M
40	M	-	M	-	130	N	M	N	M
45	M	-	M	-	135	N	M	N	M
50	M	-	M	-	140	N	M	N	M
55	M	N	M	N	145	N	M	N	M
60	M	N	M	N	150	N	M	N	M
65	M	N	M	N	155	N	M	N	M
70	M	N	M	N	160	N	-	N	M
75	M	N	M	N	165	N	-	N	M
80	M	N	M	N	170	N	-	N	-
85	M	N	M	N	175	N	-	N	-
90	M	N	M	N	180	N	-	N	-

Table 5.9: First (1) and Second (2) Preference Policy choices for the second 3 hours of the 9-hour simulation

Table 5.9 illustrates the results of the second 3 hours of the 9-hour simulation. The results show that the most chosen policies overall were morning and noon. It can be observed that such policies were almost on par, with the morning policy being the most chosen for the first 95 minutes, whereas the noon policy was the most chosen for the remaining 85 minutes. This pattern shows an important dynamic, that although the scenario switches to a noon traffic profile, initially the system continues to behave like a morning scenario. Such a lag occurs because the policy selection is being driven by the current state of the intersection, which still reflects earlier

conditions. Some time is required in order for the new arrival rate to influence the intersection, and as a consequence, the chosen policy. Although the noon policy was not the sole most chosen policy, this did not impact the policy evaluation measures presented earlier in this section. In fact, the deterministic results were presented purposely to demonstrate that these policy choices produce slightly better outcomes.

One may assume that the noon policy is being chosen in the 100^{th} minute and not earlier, since low intensity traffic does not feature between the morning time and noon time periods. Since during the morning time period there are no fluctuations in traffic intensity, thereby having only high traffic intensity, the effects are carried over in the next time period, being the noon time period.

Policy Choices - Evening Time									
Time (mins)	Eucl. (1)	Eucl. (2)	Cheb. (1)	Cheb. (2)	Time (mins)	Eucl. (1)	Eucl. (2)	Cheb. (1)	Cheb. (2)
5	N	-	N	-	95	N	M	N	M
10	N	-	N	-	100	N	M	N	M
15	N	-	N	-	105	N	M	N	M
20	N	-	N	-	110	N	M	N	M
25	N	-	N	-	115	N	M	N	M
30	N	-	N	-	120	N	M	N	M
35	N	-	N	M	125	N	M	N	M
40	N	-	N	M	130	N	M	N	M
45	N	M	N	M	135	N	M	M	N
50	N	M	N	M	140	N	M	M	N
55	N	M	N	M	145	N	M	M	N
60	N	M	N	M	150	N	M	M	N
65	N	M	N	M	155	N	M	M	N
70	N	M	N	M	160	N	M	M	N
75	N	M	N	M	165	M	N	M	N
80	N	M	N	M	170	M	N	M	N
85	N	M	N	M	175	M	N	M	N
90	N	M	N	M	180	M	N	M	N

Table 5.10: First (1) and Second (2) Preference Policy choices for the last three hours of the 9-hour simulation

Table 5.10 best illustrates the effects that are created due to the absence of fluctuations in traffic intensity, which started to be evident during the noon period. One must note that the evening policy is tailored to control less vehicles in total, than the other two policies. As can be observed in Table 5.10, the evening policy was never chosen during the evening time period, neither as a first nor as a second preference. One explanation behind this occurrence pertains to the consistently high traffic intensity experienced throughout the 9-hour simulation. The dynamic approach continued to choose policies that are better suited at controlling for higher

number of vehicles entering the intersection. Consequently, the other policies were chosen as the system was still dealing with vehicles that had entered the system and accumulated in previous time periods. Furthermore, a total of 6 hours of peak traffic, maybe even reaching the limit of such an intersection within this time period, affected the amount of vehicles around the intersection during the last 3 hours of the simulation.

In conclusion, despite the potential limitations, which be further delved into in the next chapter, under a dynamic approach, the intersection still performed significantly better than an FC. This suggests that the concept of a dynamic approach is an effective solution for improving traffic flow.

Chapter 6

Limitations and Further Ameliorations

Traffic light systems at an intersection very often operate inefficiently, especially when such systems operate in a predetermined manner in high intensity traffic scenarios; this being referred to as an FC approach. This dissertation addressed this problem thoroughly and studied different approaches which could enhance the efficiency of traffic light systems. It was established that by developing alternative decision strategies to the FC identified through an MDP, which MDP was fully developed by the author, and ultimately finding the best policies, contributes significantly to the amelioration of traffic flow. This final chapter provides a concise summary of the results attained whilst addressing the limitations relating to this study. Lastly, possible improvements are put forward.

6.1 Summary of Results

To achieve the results presented in Chapter 5, the author developed and formulated an original MDP model, which is set out in Section 4.4. The author obtained the morning, noon and evening policies, so that they can be used through an API in the Aimsun simulator, by using Python to define states, actions, transition probabilities, rewards and related algorithms. This code can be found in Appendix A. Furthermore, a separate code was developed on Python, in order to interact with Aimsun

through an API, thereby allowing the allocation of green time at the intersection according to the selected policy. The aforementioned code can be found in Appendix B. A slight addition to the latter code, found in Appendix C, was also developed to accommodate the 9 hour simulation and its respective results. Lastly, a code to transpose all the results from Aimsun and average over the 100 simulations was necessary to obtain the results which are presented in Chapter 5; such code can be found in Appendix D. Overall, the Monte Carlo simulations took around 30 minutes for the 1 hour scenario, while 2-hours for the 9 hour scenario. The computer used to run the simulations and execute the code has an Intel Core i5-7200U processor (2 cores, 4 threads, 2.5–3.1 GHz) and 8 GB of RAM.

By virtue of the above-mentioned code, the results that were attained demonstrate that the MDP driven cycle, across the morning, noon and evening scenarios, outperforms the traditional FC. This improvement was experienced in both sets of results; those results reflecting the different times of day distinctly, as demonstrated in Section 5.3, and the other results reflecting a combined 9-hour simulation integrating all times of day, as demonstrated in Section 5.4. Although there were roads that sustained a slight increase in waiting time under the MDP driven cycle, it can be concluded that these were calculated trade-offs, which ultimately contributed to better overall performance; namely, reducing congestion, improving throughput and minimising vehicle waiting time and stops. These results affirm the superiority that the MDP driven cycle holds over the FC, in that it is a more effective and adaptive tool for traffic management.

6.2 Limitations

A number of limitations were encountered throughout this dissertation. A major limitation experienced by the author, which may be addressed in future studies, is the fact that Aimsun does not have an in-built feature for cool down periods, despite having an in-built warm up feature. This affected the study in various ways; however, it was mainly encountered in the 9-hour simulation, since the traffic intensity was consistently high throughout the whole 9-hours. A cool down period would have contributed to a more realistic 9-hour traffic period, with fluctuations

in traffic intensity. Nevertheless, the 9-hour simulation was still performed, since the primary aim of this dissertation was to demonstrate that an MDP driven cycle outperforms an FC, and thus to validate the effectiveness of this concept.

Additionally, another minor limitation is that Aimsun does not cater for vehicle accidents. Nevertheless, if a vehicle stops for an extended period, the model still operates effectively.

Contrarily, something which Aimsun supports, but the data used in this study lacked, is the vehicle type. In Aimsun, public transport, heavy vehicles, bicycles and also pedestrians can be programmed to be part of the simulation. However, the data used in this study did not have this level of detail; hence, only standard vehicles were considered. Had this type of data been available, the traffic intersection could have been modelled more realistically. Notwithstanding this, the author asserts that this limitation does not undermine the study's value, since the effectiveness and viability of the MDP driven model were still demonstrated.

On the same note, the author was also limited since local traffic data of a network of intersections (e.g. 3 subsequent intersections) is non-existent. Consequently, this study focused only on one intersection. However, the model can be applied to any existing four way intersection by simply altering the parameters accordingly. Notwithstanding this, the study would not operate if one were to apply it to more than one intersection simultaneously.

Lastly, this study was also restricted by the RAM capacity of the computer's processor being used, since multiple transition probability matrices and large policy tables had to be stored. Without this constraint, the author would have been able to develop more policies, which would have provided a vaster policy selection during the 9-hour simulation. This would have enabled the model to make finer decisions leading to further improvements.

6.3 Possible Improvements

If one were to consider this study in the future, further developments can be carried out. A natural extension would be that of leveraging RL techniques to the model, since these are intended to learn optimal policies within MDP frameworks. By integrating this framework, adaptive policies can be attained which optimise the intersection's behaviour by direct environment interaction. This is the main improvement to be considered for future research, yet others exist.

In addition to the aforementioned improvement, future research could consider, in combination with RL, Artificial Neural Networks (ANNs). The amalgamation of these two powerful approaches is known as Deep Q-learning, or Deep Q-Networks (DQNs). While the results presented in this study demonstrate promising results, DQNs have the advantage of utilising deep neural networks for action-value function approximation. Such a characteristic enables the user to deal with more complex traffic scenarios and higher-dimensional state spaces. This capability is important to enable richer state descriptions, namely vehicle position and vehicle waiting times. An amelioration of this calibre would lead to more adaptive and effective policies relating to traffic signal control.

For instance, special conditions which indirectly affect traffic can be considered, such as weather conditions and road conditions. Weather conditions may influence traffic due to factors like reduced visibility and extra driver caution, leading to slower moving traffic. Similarly, road conditions, such as infrastructure defects, may also cause slower moving traffic. Although these special conditions can all be integrated in Aimsun, the author decided to consider normal everyday conditions.

Furthermore, other test intersections and differently structured intersections (3 way intersections) can be studied. This would provide an opportunity for future authors to consider networks with more than one intersection. Real-time data can also be considered rather than past data, which would provide more up to-date results. Once real-time data is comprehensively validated and tested within the simulation, it may also be more easily integrated in to a real-world traffic system.

6.4 Concluding Remarks

This dissertation explored an alternative approach to the traditional method used in traffic light systems, which nowadays, through extensive studies, is gradually becoming a thing of the past when dealing with high traffic intensities. This work bestows substantial value to the ongoing endeavour to improve traffic light systems, and offers a strong and solid foundation that future research and technologies can build upon.

BIBLIOGRAPHY

Abdoos, M., Mozayani, N. and Bazzan, A.L. (2011) 'Traffic Light Control in non-stationary environments based on Multi agent Q-learning', 2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC), pp. 1580–1585. doi:10.1109/itsc.2011.6083114.

Abdulhai, B., Pringle, R. and Karakoulas, G.J. (2003) 'Reinforcement learning for true adaptive traffic signal control', Journal of Transportation Engineering, 129(3), pp. 278–285. doi:10.1061/(asce)0733-947x(2003)129:3(278).

Althaqafi, T., 2024. A study on inventory control system for a supply chain using Markov decision processes. Edelweiss Applied Science and Technology, 8(6), pp.7846-7864.

Barceló, J., (2010). Fundamentals of traffic simulation (Vol. 145, p. 439). New York: Springer.

Bayer, P., Brown, J.S., Dubbeldam, J. and Broom, M. (2021) A Markovian decision model of adaptive cancer treatment and quality of life. Toulouse School of Economics Working Paper No. 1291

Brechtel, S., Gindele, T. and Dillmann, R. (2011) 'Probabilistic MDP-behavior planning for Cars', 2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC), pp. 1537–1542. doi:10.1109/itsc.2011.6082928.

Busoniu, L., Babuska, R., De Schutter, B. and Ernst, D., (2017). Reinforcement learning and dynamic programming using function approximators. CRC press.

Chiu, S. (1992) 'Adaptive Traffic Signal Control using fuzzy logic', Proceedings of the Intelligent Vehicles '92 Symposium, pp. 98–107. doi:10.1109/ivs.1992.252240.

De Schutter, B. and De Moor, B., (1998). Optimal traffic light control for a single intersection. *European Journal of Control, 4(3)*, pp.260-276.

Feinberg, E.A. and Shwartz, A. eds., (2012). Handbook of Markov decision processes: methods and applications (Vol. 40). Springer Science & Business Media.

Foy, M.D., Benekohal, R.F. and Goldberg, D.E., 1992. Signal timing determination using genetic algorithms. *Transportation Research Record*, (1365), p.108.

Gatt, L. (2020) Control of connected and autonomous vehicles. University of Malta.

Haijema, R. and van der Wal, J. (2008) 'An MDP decomposition approach for traffic control at isolated signalized intersections', Probability in the Engineering and Informational Sciences, 22(4), pp. 587–602. doi:10.1017/s026996480800034x.

Han, G. et al. (2022) 'Deep reinforcement learning for intersection signal control considering pedestrian behavior', Electronics, 11(21), p. 3519. doi:10.3390/electronics11213519.

Henry, J.J., Farges, J.L. and Gallego, J.L. (1998) 'Neuro-fuzzy techniques for traffic control', Control Engineering Practice, 6(6), pp. 755–761. doi:10.1016/s0967-0661(98)00081-1.

Hirulkar, P., Bajaj, P., & Deshpande, R. (2013). Optimization of traffic flow through signalized intersections using Particle Swarm Optimization (PSO). *International Journal of Advances in Computer Science and Its Applications, 1(1)*, 434–437.

Howard, R.A. (1960) Dynamic programming and Markov Processes. Cambridge, Mass: M.I.T. Press.

Hunt, P.B., Robertson, D.I., Bretherton, R.D. and Winton, R.I., (1981). *SCOOT-a traffic responsive method of coordinating signals (No. LR 1014 Monograph).*

Jafari, S., Shahbazi, Z. and Byun, Y.-C. (2021) 'Traffic control prediction design based on fuzzy logic and Lyapunov approaches to improve the performance of road intersection', Processes, 9(12), p. 2205. doi:10.3390/pr9122205.

Jian Tan, Hong Xie and Yung-Cheng Lee (1995) 'Efficient establishment of a fuzzy logic model for process modeling and Control', IEEE Transactions on Semiconductor Manufacturing, 8(1), pp. 50–61. doi:10.1109/66.350757.

Jintamuttha, K., Watanapa, B. and Charoenkitkarn, N. (2016) 'Dynamic traffic light timing optimization model using BAT algorithm', 2016 2nd International Conference on Control Science and Systems Engineering (ICCSSE), pp. 181–185. doi:10.1109/ccsse.2016.7784378.

Liang, X. et al. (2019) 'A deep reinforcement learning network for Traffic Light Cycle Control', IEEE Transactions on Vehicular Technology, 68(2), pp. 1243–1253. doi:10.1109/tvt.2018.2890726.

Liu, M., Yu, L., Guo, J., Guo, S., Guo, J. and Wen, H., (2007). Fuzzy logic-based urban traffic congestion evaluation models and applications. *In International Conference on Transportation Engineering 2007* (pp. 1169-1174).

Lim, S.H., Xu, H. and Mannor, S. (2016) 'Reinforcement learning in robust Markov Decision Processes', Mathematics of Operations Research, 41(4), pp. 1325–1353. doi:10.1287/moor.2016.0779.

Lowrie, P.R., (1982). The Sydney cooridinated adaptive traffic (SCAT) system-principles, methodology, algorithm. *In Proc. of the Second International Conference on Road Traffic Signaling, IEE* (pp. 67-70).

Minoarivelo, O.H. (2009) Application of markov decision processes to the control of ... Available at: https://www.inf.ufrgs.br/maslab/traffic/seeChapter4forSUMO.pdf (Accessed: 29 June 2024).

Passos, L.S., Rossetti, R.J. and Kokkinogenis, Z., (2011), June. Towards the next-generation traffic simulation tools: a first appraisal. *In 6th Iberian Conference on Information Systems and Technologies (CISTI 2011)* (pp. 1-6). IEEE.

Patel, M. and Ranganathan, N. (2001) 'IDUTC: An intelligent decision-making system for urban traffic-control applications', IEEE Transactions on Vehicular Technology, 50(3), pp. 816–829. doi:10.1109/25.933315.

Prinsen, L. H. A. (2006) Personal communications with Luc Prinsen. Goudappel Coffeng.

Puterman, M.L., (2014). Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons.

Shapley, L.S. (1953) 'Stochastic games', Proceedings of the National Academy of Sciences, 39(10), pp. 1095–1100. doi:10.1073/pnas.39.10.1953.

Shortle, J.F., Thompson, J.M., Gross, D. and Harris, C.M., (2018). Fundamentals of queueing theory. John Wiley & Sons.

Sigman, M. (2009). Urban traffic signal control using reinforcement learning agents. *IET Intelligent Transport Systems, 3(2)*, 153–162. https://doi.org/10.1049/iet-its.2009.0096

Steingröver, M., Schouten, R., Peelen, S., Nijhuis, E., & Bakker, B. (2005). Reinforcement learning of traffic light controllers adapting to traffic congestion. *Proceedings of the 17th Belgium-Netherlands Conference on Artificial Intelligence (pp. 216–223).* Brussels, Belgium

Sutton, R.S. and Barto, A.G., (1999). Reinforcement learning. *Journal of Cognitive Neuroscience, 11(1)*, pp.126-134.

Sutton, R.S., Bach, F. and Barto, A.G. (2018) Reinforcement learning: An introduction. Massachusetts: MIT Press Ltd.

Teo, K.T.K., Kow, W.Y. and Chin, Y.K. (2010) 'Optimization of traffic flow within an urban traffic light intersection with genetic algorithm', 2010 Second International Conference on Computational Intelligence, Modelling and Simulation [Preprint]. doi:10.1109/cimsim.2010.95.

Thorpe, T.L. and Anderson, C.W., (1996). Traffic light control using sarsa with three state representations. Technical report, Citeseer.

U.S. Department of Transportation's Federal Highway Administration. (no date) Federal Highway Administration. Available at: https://highways.dot.gov/ (Accessed: 30 June 2024).

Vajjha, K., Shinnar, A., Trager, B., Pestun, V. and Fulton, N., (2021). CertRL: formalizing convergence proofs for value and policy iteration in Coq. *In Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (pp. 18-31).

Webster, F.V., (1958). Traffic signal settings (No. 39).

Wiering, M. et al. (2004) 'Simulation and optimization of traffic in a City', IEEE Intelligent Vehicles Symposium, 2004, pp. 453–458. doi:10.1109/ivs.2004.1336426.

Appendix A

Model and Policy Evaluation Code

```
#%% Importing and some initial parameters
import math
import numpy as np
import time
from scipy.stats import poisson
import itertools
import pandas as pd

## Official as per Aimsun ##
max_k = 35
max_l = 50
max_m = 40
max_n = 100

max_kV2 = 8
max_lV2 = 11
max_mV2 = 9
max_nV2 = 21

# Define I_t
I_t = ['Sliema Open', 'Gzira Open', 'Msida Open', 'Kappara Open']
```

```
N = 4*max_k*max_l*max_m*max_n

#%% Defining some functions and state space

def states(I_t,max_k,max_l,max_m,max_n):
    # Create an empty matrix to store the combinations
    matrix = []
    # Create state_next by combining I_t with each combination
    # in the matrix
    All_states = []
    # Iterate over all possible combinations of k, l, m, n
    for k in range(max_k + 1):
        for l in range(max_l + 1):
            for m in range(max_m + 1):
                for n in range(max_n + 1):
                    # Append the current combination to the matrix
                    matrix.append([k, l, m, n])

    for i_t in I_t:
        # Extend state_next with combinations
        # for the current element in I_t
        All_states.extend([[i_t] + combination for combination in matrix])

    return All_states

matrix = []
# Create state_next by combining I_t with each combination in the matrix
All_states = []
# Iterate over all possible combinations of k, l, m, n
for k in range(max_k+1):
    for l in range(max_l+1):
        for m in range(max_m+1):
            for n in range(max_n+1):
                # Append the current combination to the matrix
```

```
                matrix.append([k, l, m, n])

for i_t in I_t:
    # Extend state_next with combinations
    # for the current element in I_t
    All_states.extend([[i_t] + combination for combination in matrix])

converted_states = []
range_size = 5
seen_states = set()

for state in All_states:
    converted_state = [state[0]]  # Assuming the first element is a string
    for value in state[1:]:
        # Convert numerical values to ranges
        start_range = (value // range_size) * range_size
        end_range = start_range + range_size - 1
        range_number = start_range // range_size + 1
        converted_state.append(range_number)

    # Convert the converted_state list to a tuple for set comparison
    tuple_state = tuple(converted_state)

    # Check if the state has been seen before
    if tuple_state not in seen_states:
        seen_states.add(tuple_state)
        converted_states.append(converted_state)


def possnext_statesV2(action, state):
    all_next_states = []
    k, l, m, n = state[1:]
    It = state[0]
```

```
if action == 'Stay the same':
    if state[0] == 'Sliema Open':
        q_range = range(max(1, k - 1), min(k+1,max_kV2)+1)
        x_range = range(l, min(l+2,max_lV2)+1)
        y_range = range(m, min(m+2,max_mV2)+1)
        z_range = range(n, min(n+3,max_nV2)+1)
    elif state[0] == 'Gzira Open':
        q_range = range(k, min(k+2,max_kV2)+1)
        x_range = range(max(1, l - 1), min(l+1,max_lV2)+1)
        y_range = range(m, min(m+2,max_mV2)+1)
        z_range = range(n, min(n+3,max_nV2)+1)
    elif state[0] == 'Msida Open':
        q_range = range(k, min(k+2,max_kV2)+1)
        x_range = range(l, min(l+2,max_lV2)+1)
        y_range = range(max(1, m - 1), min(m+1,max_mV2)+1)
        z_range = range(n, min(n+3,max_nV2)+1)
    else:
        q_range = range(k, min(k+2,max_kV2)+1)
        x_range = range(l, min(l+2,max_lV2)+1)
        y_range = range(m, min(m+2,max_mV2)+1)
        z_range = range(max(1, n - 1), min(n+2,max_nV2)+1)

elif action == 'Change':
    if state[0] == 'Sliema Open':
        q_range = range(k, min(k+2,max_kV2)+1)
        x_range = range(max(1, l - 1), min(l+1,max_lV2)+1)
        y_range = range(m, min(m+2,max_mV2)+1)
        z_range = range(n, min(n+3,max_nV2)+1)
        It = 'Gzira Open'
    elif state[0] == 'Gzira Open':
        q_range = range(k, min(k+2,max_kV2)+1)
        x_range = range(l, min(l+2,max_lV2)+1)
        y_range = range(max(1, m - 1), min(m+1,max_mV2)+1)
        z_range = range(n, min(n+3,max_nV2)+1)
```

```
            It = 'Msida Open'
        elif state[0] == 'Msida Open':
            q_range = range(k, min(k+2,max_kV2)+1)
            x_range = range(l, min(l+2,max_lV2)+1)
            y_range = range(m, min(m+2,max_mV2)+1)
            z_range = range(max(1, n - 1), min(n+2,max_nV2)+1)
            It = 'Kappara Open'
        else:
            q_range = range(max(1, k - 1), min(k+1,max_kV2)+1)
            x_range = range(l, min(l+2,max_lV2)+1)
            y_range = range(m, min(m+2,max_mV2)+1)
            z_range = range(n, min(n+3,max_nV2)+1)
            It = 'Sliema Open'

    for q in q_range:
        for x in x_range:
            for y in y_range:
                for z in z_range:
                    all_next_states.append([It, q, x, y, z])

    return all_next_states

#%% Reward Function
rew_dict = {index: (index-1) * 5 + 2 for index in range(1, 22)}

def RewofCap(state,next_state):

    It = state[0]
    Itp1 = next_state[0]
    k,l,m,n = state[1:]
    mid_k,mid_l,mid_m,mid_n = rew_dict[k],rew_dict[l],
                              rew_dict[m],rew_dict[n]
    kcap,lcap,mcap,ncap = (rew_dict[k]/max_k)*100,
                          (rew_dict[l]/max_l)*100,
```

```
                        (rew_dict[m]/max_m)*100,
                        (rew_dict[n]/max_n)*100
ttl_cap = kcap+lcap+mcap+ncap
#####
k_p, l_p, m_p, n_p = next_state[1:]
mid_k_p,mid_l_p,mid_m_p,mid_n_p = rew_dict[k_p], rew_dict[l_p],
                                    rew_dict[m_p], rew_dict[n_p]
k_pcap, l_pcap, m_pcap, n_pcap = (rew_dict[k_p]/max_k)*100,
                                    (rew_dict[l_p]/max_l)*100,
                                    (rew_dict[m_p]/max_m)*100,
                                    (rew_dict[n_p]/max_n)*100
ttl_pcap = k_pcap + l_pcap + m_pcap + n_pcap
deccap_k = min(mid_k,5)/max_k
deccap_l = min(mid_k,5)/max_k
deccap_m = min(mid_k,5)/max_k
deccap_n = min(mid_k,5)/max_k

if It == Itp1:
    if It == 'Sliema Open':
        w_1,w_2,w_3,w_4 = 1,2,3,4
        reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                    - (k_pcap*w_1+ l_pcap*w_2 + m_pcap*w_3 + n_pcap*w_4))
                        /(w_1+w_2+w_3+w_4)
    elif It == 'Gzira Open':
        w_1,w_2,w_3,w_4 = 4,1,2,3
        reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                    - (k_pcap*w_1+ l_pcap*w_2 + m_pcap*w_3 + n_pcap*w_4))
                        /(w_1+w_2+w_3+w_4)
    elif It == 'Msida Open':
        w_1,w_2,w_3,w_4 = 3,4,1,2
        reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                    - (k_pcap*w_1+ l_pcap*w_2 + m_pcap*w_3 + n_pcap*w_4))
                        /(w_1+w_2+w_3+w_4)
    else:
```

```
            w_1,w_2,w_3,w_4 = 2,3,4,1
            reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                        - (k_pcap*w_1+ l_pcap*w_2 + m_pcap*w_3 + n_pcap*w_4))
                            /(w_1+w_2+w_3+w_4)

    else:
        if It == 'Sliema Open':
            w_1,w_2,w_3,w_4 = 1,2,3,4
            reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                        - (k_pcap*w_4+ l_pcap*w_1 + m_pcap*w_2 + n_pcap*w_3))
                            /(w_1+w_2+w_3+w_4)
        elif It == 'Gzira Open':
            w_1,w_2,w_3,w_4 = 4,1,2,3
            reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                        - (k_pcap*w_4+ l_pcap*w_1 + m_pcap*w_2 + n_pcap*w_3))
                            /(w_1+w_2+w_3+w_4)
        elif It == 'Msida Open':
            w_1,w_2,w_3,w_4 = 3,4,1,2
            reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                        - (k_pcap*w_4+ l_pcap*w_1 + m_pcap*w_2 + n_pcap*w_3))
                            /(w_1+w_2+w_3+w_4)
        else:
            w_1,w_2,w_3,w_4 = 2,3,4,1
            reward = ((kcap*w_1 + lcap*w_2 + mcap*w_3 + ncap*w_4)
                        - (k_pcap*w_4+ l_pcap*w_1 + m_pcap*w_2 + n_pcap*w_3))
                            /(w_1+w_2+w_3+w_4)

    return reward

#%% Transition Probabilties

def transition_funcNEW(action, state,state_next, lambda_s,lambda_g,
                        lambda_m,lambda_k):
    k,l,m,n = state[1:]
```

```
k_prime, l_prime, m_prime, n_prime = state_next[1:]
z=5
aph = 0.5
It = state[0]
I_tp1 = state_next[0]
w = [aph, aph*(1-aph)**1,aph*(1-aph)**2,aph*(1-aph)**3,aph*(1-aph)**4]

if (action == 'Stay the same' and It == 'Sliema Open'
        and I_tp1 == 'Sliema Open')
            or (action == 'Change' and It == 'Kappara Open'
                and I_tp1 == 'Sliema Open'):
    if l == max_lV2 and l_prime == l:
        p_2 = 1
    elif l == max_lV2 -1:
        if l_prime == l:
            p_2 = 1/5 * (sum(poisson.cdf(i,lambda_g)
                                for i in range(z)))
        else:
            p_2 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_g)
                    for i in range(z)))
    elif l < max_lV2 - 1:
        if  l_prime == l:
            p_2 = 1/5 * (sum(poisson.cdf(i,lambda_g)
                                for i in range(z)))
        elif l_prime == l +1:
            p_2 = 1/5 * (sum((poisson.cdf(9-i,lambda_g)
                            - poisson.cdf(4-i,lambda_g))
                                for i in range(z)))
        else:
            p_2 = 1 - 1/5*( (sum((poisson.cdf(9-i,lambda_g)
                                - poisson.cdf(4-i,lambda_g))
                                        for i in range(z)))
                                + (sum(poisson.cdf(i,lambda_g)
                                        for i in range(z))))
```

```
else:
    p_2 = 0
############
if m == max_mV2 and m_prime == m:
    p_3 = 1
elif m == max_mV2 -1:
    if m_prime == m:
        p_3 = 1/5 * (sum(poisson.cdf(i,lambda_m)
                         for i in range(z)))
    else:
        p_3 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_m)
                for i in range(z)))
elif m < max_mV2 - 1:
    if  m_prime == m:
        p_3 = 1/5 * (sum(poisson.cdf(i,lambda_m)
                         for i in range(z)))
    elif m_prime == m +1:
        p_3 = 1/5 * (sum((poisson.cdf(9-i,lambda_m)
                         - poisson.cdf(4-i,lambda_m))
                                for i in range(z)))
    else:
        p_3 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_m)
                                - poisson.cdf(4-i,lambda_m))
                                        for i in range(z))
                                + sum(poisson.cdf(i,lambda_m)
                                        for i in range(z)))
else:
    p_3 = 0
############
if n == max_nV2 and n_prime == n:
    p_4 = 1
elif n == max_nV2 -1:
    if n_prime == n:
        p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
```

```
                            for i in range(z)))
        else:
            p_4 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_k)
                    for i in range(z)))
    elif n < max_nV2 - 2:
        if  n_prime == n:
            p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                            for i in range(z)))
        elif n_prime == n +1:
            p_4 = 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                        - poisson.cdf(4-i,lambda_k))
                                    for i in range(z)))
        elif n_prime == n + 2:
            p_4 = 1/5 *(sum((poisson.cdf(14-i,lambda_k)
                        - poisson.cdf(9-i,lambda_k))
                                    for i in range(z)))
        else:
            p_4 = 1 - 1/5 * ((sum((poisson.cdf(14-i,lambda_k)
                            - poisson.cdf(9-i,lambda_k))
                                    for i in range(z)))
                            + sum((poisson.cdf(9-i,lambda_k)
                            - poisson.cdf(4-i,lambda_k))
                                    for i in range(z))
                            + sum(poisson.cdf(i,lambda_k)
                                    for i in range(z)))
    elif n == max_nV2 - 2:
        if  n_prime == n:
            p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                            for i in range(z)))
        elif n_prime == n +1:
            p_4 = 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                        - poisson.cdf(4-i,lambda_k))
                                for i in range(z)))
        else:
```

```
            p_4 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                                    - poisson.cdf(4-i,lambda_k))
                                            for i in range(z))
                                    + sum(poisson.cdf(i,lambda_k)
                                            for i in range(z)))
    else:
        p_4 = 0
    ############
    if k ==1:
        if k_prime == k:
            p_1 = 1/5 * (sum(poisson.cdf(9-i,lambda_s)
                                for i in range(z)))
        else:
            p_1 = 1- 1/5 * (sum(poisson.cdf(9-i,lambda_s)
                                for i in range(z)))
    elif k > 1 and k < max_kV2:
        if k_prime == k-1:
            p_1 = 1/5 * (sum(poisson.cdf(i,lambda_s)
                                for i in range(z)))
        elif k_prime == k:
            p_1 = 1/5 * (sum(poisson.cdf(9-i,lambda_s)
                            - poisson.cdf(4-i,lambda_s)
                                    for i in range(z)))
        else:
            p_1 = 1 - 1/5 * (sum(poisson.cdf(9-i,lambda_s)
                                    - poisson.cdf(4-i,lambda_s)
                                            for i in range(z))
                                    +sum(poisson.cdf(i,lambda_s)
                                            for i in range(z)))
    elif k == max_kV2:
        if k_prime == k:
            p_1 = 1- sum(w[4-i]*poisson.cdf(i,lambda_s)
                                    for i in range(z))
        else:
```

```
            p_1 = sum(w[4-i]*poisson.cdf(i,lambda_s)
                                    for i in range(z))
    else:
        p_1 = 0

elif (action == 'Stay the same' and It == 'Gzira Open'
            and I_tp1 == 'Gzira Open')
                or (action == 'Change' and It == 'Sliema Open'
                        and I_tp1 == 'Gzira Open'):
    if k == max_kV2 and k_prime == k:
        p_1 = 1
    elif k == max_kV2 -1:
        if k_prime == k:
            p_1 = 1/5 * (sum(poisson.cdf(i,lambda_s)
                                for i in range(z)))
        else:
            p_1 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_s)
                                    for i in range(z)))
    elif k < max_kV2 - 1:
        if  k_prime == k:
            p_1 = 1/5 * (sum(poisson.cdf(i,lambda_s)
                                for i in range(z)))
        elif k_prime == k +1:
            p_1 = 1/5 * (sum((poisson.cdf(9-i,lambda_s)
                                - poisson.cdf(4-i,lambda_s))
                                            for i in range(z)))
        else:
            p_1 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_s)
                                    - poisson.cdf(4-i,lambda_s))
                                            for i in range(z))
                                    +sum(poisson.cdf(i,lambda_s)
                                            for i in range(z)))
    else:
        p_1 = 0
```

```
############
if m == max_mV2 and m_prime == m:
    p_3 = 1
elif m == max_mV2 -1:
    if m_prime == m:
        p_3 = 1/5 * (sum(poisson.cdf(i,lambda_m)
                                for i in range(z)))
    else:
        p_3 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_m)
                                for i in range(z)))
elif m < max_mV2 - 1:
    if  m_prime == m:
        p_3 = 1/5 * (sum(poisson.cdf(i,lambda_m)
                                for i in range(z)))
    elif m_prime == m +1:
        p_3 = 1/5 * (sum((poisson.cdf(9-i,lambda_m)
                            - poisson.cdf(4-i,lambda_m))
                                for i in range(z)))
    else:
        p_3 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_m)
                            - poisson.cdf(4-i,lambda_m))
                                    for i in range(z))
                            + sum(poisson.cdf(i,lambda_m)
                                    for i in range(z)))
else:
    p_3 = 0
############
if n == max_nV2 and n_prime == n:
    p_4 = 1
elif n == max_nV2 -1:
    if n_prime == n:
        p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                                for i in range(z)))
    else:
```

```
            p_4 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_k)
                                    for i in range(z)))
    elif n < max_nV2 - 2:
        if  n_prime == n:
            p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                                    for i in range(z)))
        elif n_prime == n +1:
            p_4 = 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                                - poisson.cdf(4-i,lambda_k))
                                            for i in range(z)))
        elif n_prime == n + 2:
            p_4 = 1/5 *(sum((poisson.cdf(14-i,lambda_k)
                                - poisson.cdf(9-i,lambda_k))
                                            for i in range(z)))
        else:
            p_4 = 1 - 1/5 * ((sum((poisson.cdf(14-i,lambda_k)
                                - poisson.cdf(9-i,lambda_k))
                                            for i in range(z)))
                                + sum((poisson.cdf(9-i,lambda_k)
                                - poisson.cdf(4-i,lambda_k))
                                            for i in range(z))
                                + sum(poisson.cdf(i,lambda_k)
                                            for i in range(z)))
    elif n == max_nV2 - 2:
        if  n_prime == n:
            p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                                    for i in range(z)))
        elif n_prime == n +1:
            p_4 = 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                                - poisson.cdf(4-i,lambda_k))
                                            for i in range(z)))
        else:
            p_4 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                                    - poisson.cdf(4-i,lambda_k))
```

```
                                                for i in range(z))
                                    + sum(poisson.cdf(i,lambda_k)
                                                for i in range(z)))
        else:
            p_4 = 0
        ############
        if l ==1:
            if l_prime == l:
                p_2 = 1/5 * (sum(poisson.cdf(9-i,lambda_g)
                                    for i in range(z)))
            else:
                p_2 = 1- 1/5 * (sum(poisson.cdf(9-i,lambda_g)
                                        for i in range(z)))
        elif l > 1 and l < max_lV2:
            if l_prime == l-1:
                p_2 = 1/5 * (sum(poisson.cdf(i,lambda_g)
                                    for i in range(z)))
            elif l_prime == l:
                p_2 = 1/5 * (sum(poisson.cdf(9-i,lambda_g)
                                - poisson.cdf(4-i,lambda_g)
                                                for i in range(z)))
            else:
                p_2 = 1 - 1/5 * (sum(poisson.cdf(9-i,lambda_g)
                                    - poisson.cdf(4-i,lambda_g)
                                                for i in range(z))
                                    +sum(poisson.cdf(i,lambda_g)
                                                for i in range(z)))
        elif l == max_lV2:
            if l_prime == l:
                p_2 = 1- sum(w[4-i]*poisson.cdf(i,lambda_g)
                                        for i in range(z))
            else:
                p_2 = sum(w[4-i]*poisson.cdf(i,lambda_g)
                                        for i in range(z))
```

```
        else:
            p_2 = 0


    elif (action == 'Stay the same' and It == 'Msida Open'
            and I_tp1 == 'Msida Open')
                or (action == 'Change' and It == 'Gzira Open'
                    and I_tp1 == 'Msida Open'):
        if k == max_kV2 and k_prime == k:
            p_1 = 1
        elif k == max_kV2 -1:
            if k_prime == k:
                p_1 = 1/5 * (sum(poisson.cdf(i,lambda_s)
                                 for i in range(z)))
            else:
                p_1 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_s)
                                 for i in range(z)))
        elif k < max_kV2 - 1:
            if  k_prime == k:
                p_1 = 1/5 * (sum(poisson.cdf(i,lambda_s)
                                 for i in range(z)))
            elif k_prime == k +1:
                p_1 = 1/5 * (sum((poisson.cdf(9-i,lambda_s)
                                 - poisson.cdf(4-i,lambda_s))
                                           for i in range(z)))
            else:
                p_1 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_s)
                                      - poisson.cdf(4-i,lambda_s))
                                           for i in range(z))
                                      +sum(poisson.cdf(i,lambda_s)
                                           for i in range(z)))
        else:
            p_1 = 0
        ############
        if l == max_lV2 and l_prime == l:
```

```
    p_2 = 1
elif l == max_lV2 -1:
    if l_prime == l:
        p_2 = 1/5 * (sum(poisson.cdf(i,lambda_g)
                            for i in range(z)))
    else:
        p_2 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_g)
                            for i in range(z)))
elif l < max_lV2 - 1:
    if  l_prime == l:
        p_2 = 1/5 * (sum(poisson.cdf(i,lambda_g)
                            for i in range(z)))
    elif l_prime == l +1:
        p_2 = 1/5 * (sum((poisson.cdf(9-i,lambda_g)
                        - poisson.cdf(4-i,lambda_g))
                                    for i in range(z)))
    else:
        p_2 = 1 - 1/5*( (sum((poisson.cdf(9-i,lambda_g)
                                - poisson.cdf(4-i,lambda_g))
                                        for i in range(z)))
                                + (sum(poisson.cdf(i,lambda_g)
                                        for i in range(z))))
else:
    p_2 = 0
############
if n == max_nV2 and n_prime == n:
    p_4 = 1
elif n == max_nV2 -1:
    if n_prime == n:
        p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                                for i in range(z)))
    else:
        p_4 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_k)
                                for i in range(z)))
```

```
elif n < max_nV2 - 2:
    if  n_prime == n:
        p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                                for i in range(z)))
    elif n_prime == n +1:
        p_4 = 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                        - poisson.cdf(4-i,lambda_k))
                                for i in range(z)))
    elif n_prime == n + 2:
        p_4 = 1/5 *(sum((poisson.cdf(14-i,lambda_k)
                        - poisson.cdf(9-i,lambda_k))
                                for i in range(z)))
    else:
        p_4 = 1 - 1/5 * ((sum((poisson.cdf(14-i,lambda_k)
                            - poisson.cdf(9-i,lambda_k))
                                    for i in range(z)))
                            + sum((poisson.cdf(9-i,lambda_k)
                            - poisson.cdf(4-i,lambda_k))
                                    for i in range(z))
                            + sum(poisson.cdf(i,lambda_k)
                                    for i in range(z)))
elif n == max_nV2 - 2:
    if  n_prime == n:
        p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                            for i in range(z)))
    elif n_prime == n +1:
        p_4 = 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                        - poisson.cdf(4-i,lambda_k))
                                for i in range(z)))
    else:
        p_4 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_k)
                            - poisson.cdf(4-i,lambda_k))
                                    for i in range(z))
                            + sum(poisson.cdf(i,lambda_k)
```

```
                                    for i in range(z)))
else:
    p_4 = 0
############
if m ==1:
    if m_prime == m:
        p_3 = 1/5 * (sum(poisson.cdf(9-i,lambda_m)
                            for i in range(z)))
    else:
        p_3 = 1- 1/5 * (sum(poisson.cdf(9-i,lambda_m)
                            for i in range(z)))
elif m > 1 and m < max_mV2:
    if m_prime == m-1:
        p_3 = 1/5 * (sum(poisson.cdf(i,lambda_m)
                            for i in range(z)))
    elif m_prime == m:
        p_3 = 1/5 * (sum(poisson.cdf(9-i,lambda_m)
                        - poisson.cdf(4-i,lambda_m)
                            for i in range(z)))
    else:
        p_3 = 1 - 1/5 * (sum(poisson.cdf(9-i,lambda_m)
                            - poisson.cdf(4-i,lambda_m)
                                for i in range(z))
                            +sum(poisson.cdf(i,lambda_m)
                                for i in range(z)))
elif m == max_mV2:
    if m_prime == m:
        p_3 = 1-sum(w[4-i]*poisson.cdf(i,lambda_m)
                        for i in range(z))
    else:
        p_3 = sum(w[4-i]*poisson.cdf(i,lambda_m)
                        for i in range(z))
else:
    p_3 = 0
```

```
elif (action == 'Stay the same' and It == 'Kappara Open'
        and I_tp1 == 'Kappara Open')
            or (action == 'Change' and It == 'Msida Open'
                    and I_tp1 == 'Kappara Open'):
    if k == max_kV2 and k_prime == k:
        p_1 = 1
    elif k == max_kV2 -1:
        if k_prime == k:
            p_1 = 1/5 * (sum(poisson.cdf(i,lambda_s)
                                for i in range(z)))
        else:
            p_1 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_s)
                                for i in range(z)))
    elif k < max_kV2 - 1:
        if  k_prime == k:
            p_1 = 1/5 * (sum(poisson.cdf(i,lambda_s)
                                for i in range(z)))
        elif k_prime == k +1:
            p_1 = 1/5 * (sum((poisson.cdf(9-i,lambda_s)
                            - poisson.cdf(4-i,lambda_s))
                                for i in range(z)))
        else:
            p_1 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_s)
                                - poisson.cdf(4-i,lambda_s))
                                            for i in range(z))
                                +sum(poisson.cdf(i,lambda_s)
                                            for i in range(z)))
    else:
        p_1 = 0
    ############
    if l == max_lV2 and l_prime == l:
        p_2 = 1
    elif l == max_lV2 -1:
```

```
        if l_prime == l:
            p_2 = 1/5 * (sum(poisson.cdf(i,lambda_g)
                                for i in range(z)))
        else:
            p_2 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_g)
                                for i in range(z)))
    elif l < max_lV2 - 1:
        if  l_prime == l:
            p_2 = 1/5 * (sum(poisson.cdf(i,lambda_g)
                                for i in range(z)))
        elif l_prime == l +1:
            p_2 = 1/5 * (sum((poisson.cdf(9-i,lambda_g)
                            - poisson.cdf(4-i,lambda_g))
                                    for i in range(z)))
        else:
            p_2 = 1 - 1/5*( (sum((poisson.cdf(9-i,lambda_g)
                                - poisson.cdf(4-i,lambda_g))
                                        for i in range(z)))
                                + (sum(poisson.cdf(i,lambda_g)
                                        for i in range(z))))
    else:
        p_2 = 0
    ############
    if m == max_mV2 and m_prime == m:
        p_3 = 1
    elif m == max_mV2 -1:
        if m_prime == m:
            p_3 = 1/5 * (sum(poisson.cdf(i,lambda_m)
                                    for i in range(z)))
        else:
            p_3 = 1 - 1/5 * (sum(poisson.cdf(i,lambda_m)
                                    for i in range(z)))
    elif m < max_mV2 - 1:
        if  m_prime == m:
```

```
            p_3 = 1/5 * (sum(poisson.cdf(i,lambda_m)
                                    for i in range(z)))
        elif m_prime == m +1:
            p_3 = 1/5 * (sum((poisson.cdf(9-i,lambda_m)
                        - poisson.cdf(4-i,lambda_m))
                                    for i in range(z)))
        else:
            p_3 = 1 - 1/5 * (sum((poisson.cdf(9-i,lambda_m)
                                - poisson.cdf(4-i,lambda_m))
                                        for i in range(z))
                                +sum(poisson.cdf(i,lambda_m)
                                        for i in range(z)))
    else:
        p_3 = 0
    ############
    if n ==1:
        if n_prime == n:
            p_4 = 1/5 * (sum(poisson.cdf(9-i,lambda_k)
                                    for i in range(z)))
        elif n_prime == n+1:
            p_4 = 1/5 *(sum((poisson.cdf(14-i,lambda_k)
                        - poisson.cdf(9-i,lambda_k))
                                    for i in range(z)))
        else:
            p_4 = 1- 1/5 * (sum((poisson.cdf(14-i,lambda_k)
                                - poisson.cdf(9-i,lambda_k))
                                        for i in range(z))
                                +sum(poisson.cdf(9-i,lambda_k)
                                        for i in range(z)))
    elif n > 1 and n < max_nV2 -1:
        if n_prime == n-1:
            p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                                    for i in range(z)))
        elif n_prime == n:
```

```
            p_4 = 1/5 * (sum(poisson.cdf(9-i,lambda_k)
                            - poisson.cdf(4-i,lambda_k)
                                    for i in range(z)))
        elif n_prime == n+1:
            p_4 = 1/5 *(sum((poisson.cdf(14-i,lambda_k)
                            - poisson.cdf(9-i,lambda_k))
                                    for i in range(z)))
        else:
            p_4 = 1 - 1/5 * ((sum((poisson.cdf(14-i,lambda_k)
                                - poisson.cdf(9-i,lambda_k))
                                        for i in range(z)))
                                + sum((poisson.cdf(9-i,lambda_k)
                                - poisson.cdf(4-i,lambda_k))
                                        for i in range(z))
                                + sum(poisson.cdf(i,lambda_k)
                                        for i in range(z)))
    elif n == max_nV2 -1:
         if n_prime == n-1:
             p_4 = 1/5 * (sum(poisson.cdf(i,lambda_k)
                                    for i in range(z)))
         elif n_prime == n:
             p_4 = 1/5 * (sum(poisson.cdf(9-i,lambda_k)
                            - poisson.cdf(4-i,lambda_k)
                                    for i in range(z)))
         else:
             p_4 = 1 - 1/5 * (sum(poisson.cdf(9-i,lambda_k)
                                - poisson.cdf(4-i,lambda_k)
                                            for i in range(z))
                                + sum(poisson.cdf(i,lambda_k)
                                            for i in range(z)))
    elif n == max_nV2:
        if n_prime == n:
            p_4 = 1-sum(w[4-i]*poisson.cdf(i,lambda_k)
                                        for i in range(z))
```

```
            else:
                p_4 = sum(w[4-i]*poisson.cdf(i,lambda_k)
                                            for i in range(z))
        else:
            p_4 = 0

    else:
        p_1,p_2,p_3,p_4 = 0,0,0,0

    ttl_p = p_1*p_2*p_3*p_4
    return ttl_p


#%% Creating Dictionaries of all possible states, actions and next states

### Action - Change Dictionary ###

start_time = time.time()
#Creating dictionary for action change
chngprobdict = {}
# Record the start time
start_time = time.time()
loop_counter = 0
# Generate tuples for state transitions and calculate probabilities
for current_state in converted_states:
    loop_counter += 1
    print(loop_counter)
    all_next_states = possnext_statesV2('Change', current_state)
    for next_state in all_next_states:
        probability = transition_funcNEW('Change', current_state,
                                next_state,1.41,0.77,1.15,1.67)
        transition = (tuple(current_state), tuple(next_state))
        ##### Rewards ############
        #reward = reward_funV2('Change',current_state,0.25)
```

```
        reward = RewofCap(current_state, next_state)
        if probability > 0:
            chngprobdict[transition] = (probability,reward)
        else:
            continue

# Record the end time
end_time = time.time()
# Calculate the runtime
runtime = end_time - start_time
# Print the runtime
print(f"Runtime: {runtime} seconds")



### Action - Stay the same Dictionary ###

#Creating dict for action stay the same
stayprobdict = {}
# Record the start time
start_time = time.time()
loop_counter = 0
# Generate tuples for state transitions and calculate probabilities
for current_state in converted_states:
    loop_counter += 1
    print(loop_counter)
    all_next_states = possnext_statesV2('Stay the same', current_state)
    for next_state in all_next_states:
        probability = transition_funcNEW('Stay the same', current_state
                            ,next_state, 1.41,0.77,1.15,1.67)
        ### Rewards ###
        #reward = reward_funV2('Stay the same',current_state,0.25)
        reward = RewofCap(current_state, next_state)
        transition = (tuple(current_state), tuple(next_state))
        if probability > 0:
```

```
            stayprobdict[transition] = (probability,reward)
        else:
            continue

# Record the end time
end_time = time.time()
# Calculate the runtime
runtime = end_time - start_time
# Print the runtime
print(f"Runtime: {runtime} seconds")

#### To save a dictionary ####
import json
# Convert tuple keys to strings
converted_stayprobdict = {str(key): value for key
                                    , value in stayprobdict.items()}
# Convert tuple keys to strings
converted_chngprobdict = {str(key): value for key
                                    , value in chngprobdict.items()}

# Save the dictionary to a JSON file
with open('converted_staydictNoon_NoYTNEW25_07.json', 'w')
               as json_file: json.dump(converted_stayprobdict
                                          , json_file)

# Save the dictionary to a JSON file
with open('converted_chngdictNoon_NoYTNEW25_07.json', 'w')
               as json_file: json.dump(converted_chngprobdict
                                          , json_file)

#%% Importing Dictionaries
## To retrieve a saved sictionary and redo its format from string
import json
```

```
# Load the dictionary from the JSON file
with open('converted_staydictNoon_NoYTNEW25_07.json', 'r')
                as json_file: loaded_stayprobdict = json.load(json_file)

with open('converted_chngdictNoon_NoYTNEW25_07.json', 'r')
                as json_file: loaded_chngprobdict = json.load(json_file)

import ast
loaded_stayprobdict = {ast.literal_eval(key): value for key
                        , value in loaded_stayprobdict.items()}
loaded_chngprobdict = {ast.literal_eval(key): value for key
                        , value in loaded_chngprobdict.items()}

#%% Updating Rewards depending on function needed to be used
for key in loaded_stayprobdict:
    #print(key)
    #reward = RewardQLnxtStStand('Stay the same',key[1])
    reward = RewofCap(key[0],key[1])
    #reward = RewardDeltaQLWeights(key[0],key[1])
    loaded_stayprobdict[key][1] = reward

for key in loaded_chngprobdict:
    #reward = RewardQLnxtStStand('Change',key[1])
    reward = RewofCap(key[0],key[1])
    #reward = RewardDeltaQLWeights(key[0],key[1])
    loaded_chngprobdict[key][1] = reward

#%% Policy Iteration Algorithm
actions = ['Stay the same', 'Change']

def R(s,a,s_prime):
    #s_prime = list(s)
    s_prime = tuple(s_prime)
    try:
```

```
        if a == 'Stay the same':
            reward = loaded_stayprobdict[(s,s_prime)][1]
        else:
            # if s[0] == 'Sliema Open':
            #     s_prime[0] = 'Gzira Open'
            # elif s[0] == 'Gzira Open':
            #     s_prime[0] = 'Msida Open'
            # elif s[0] == 'Msida Open':
            #     s_prime[0] = 'Kappara Open'
            # else:
            #     s_prime[0] = 'Sliema Open'

            #s_prime = tuple(s_prime)
            reward = loaded_chngprobdict[(s,s_prime)][1]

    except KeyError:
        # If KeyError occurs, set the reward to
        reward = -9999

    return reward

def P(s_next, s, a):
    s_next = tuple(s_next)
    try:
        if a == 'Stay the same':
            prob = loaded_stayprobdict[(s, s_next)][0]
        else:
            prob = loaded_chngprobdict[(s, s_next)][0]
    except KeyError:
        # If KeyError occurs, set the probability to 0
        prob = 0

    return prob
```

```
def policy_evaluation(policy,S, gamma,V):
    S_tuples = [tuple(s) for s in S]
    for s in S_tuples:
        a = policy[s]
        #V[s]= R(s,a) + gamma*sum(P(s_next,s,a)*V[tuple(s_next)]
                        for s_next in possnext_statesV2(a,s))
        V[s]= sum(P(s_next,s,a)*(R(s,a,s_next) + gamma*V[tuple(s_next)])
                        for s_next in possnext_statesV2(a,s))
        #V[tuple(s)]= sum( P(s_next,s,a)*(R(s,a) + gamma*oldV[tuple(s_next)])
                        for s_next in possnext_statesV2(a,s))
        #if S_tuples.index(s) == 16:
            #print(str(s) + ":::" + str(V[s]))

    return V

import heapq
def policy_improv(V,S,A,gamma,policy):
    S_tuples = [tuple(s) for s in S]
    for s in S_tuples:
        Q = {}
        for a in A:
            Q[a] = sum(P(s_next,s,a)*(R(s,a,s_next)
                        + gamma*V[tuple(s_next)])
                                    for s_next in possnext_statesV2(a,s))
        policy[s] = max(Q, key = Q.get)
    return policy

from itertools import islice
import random

def policy_iterations(S,A):
    loop_counter = 0
    #policy = {tuple(s): A[0] for s in S}
    V = {tuple(s):0 for s in S}
```

```
    policy = {tuple(s): random.choice(A)
                        if random.random() < 0.5 else A[1] for s in S}
    gamma = 0.75

    while True:
        loop_counter += 1
        print(loop_counter)
        old_policy = policy.copy()
        V = policy_evaluation(policy,S,gamma,V)
        #print(":::::V:" + str(list(islice(V.values(), 10))))
        policy = policy_improv(V,S,A,gamma,policy)

        if all(old_policy[tuple(s)] == policy[tuple(s)] for s in S):
            break
    return policy

OptimalPol = policy_iterations(converted_states,actions)

#%% Exporting Optimal Policy
# Convert tuple keys to strings
converted_OptimalPol = {str(key): value for key
                                      , value in OptimalPol.items()}
with open('converted_OptimalPol_Noon_NoYT_DF075NEW_25_07.json', 'w')
            as json_file: json.dump(converted_OptimalPol , json_file)

#%% Importing optimal Policy
import json
with open(r'C:\Users\jform\Desktop
                \converted_OptimalPol_Morning_NoYT_DF075NEW_25_07.json', 'r')
            as json_file: loaded_optPol = json.load(json_file)

import ast
loaded_optPol = {ast.literal_eval(key): value for key
                                            , value in loaded_optPol.items()}
```

Appendix B

Python Code for Controlling Simulation through the Aimsun API - 1 hour simulation

```
from AAPI import *
from array import *
import math
import json
import ast
#import pandas as pd
import csv
import os
#import openpyxl
#import xlsxwriter

def initValues():
    global countall_list, det_list,count_list,countall,queue_len
            ,quelenallv1, quelenallv2, quelenallv3,quelenallv4
            ,loaded_optPol, Greensim_time, ResultsAll
            , EntranceTimes, ExitTimes, Traveltimes
            , Waitingtimes, veh_list, StopResults
    #countall should be 0 times the number of detectors
```

```
    #queue_len should have length equal to
    #the number of roads which have detectors
    countall_list= [0]*8
    det_list = []
    count_list = []
    ResultsAll = []
    veh_list = []
    StopResults = []
    EntranceTimes = [[], [], [], []]
    ExitTimes = [[], [], [], []]
    Traveltimes = [[], [], [], []]
    Waitingtimes = [[], [], [], []]
    countall = 0
    quelenallv1 = 0
    quelenallv2 = 0
    quelenallv3 = 0
    quelenallv4 = 0
    queue_len = [0]*4
    with open(r'C:\Users\jform\Desktop
                    \converted_OptimalPol_Morning_NoYT_DF075NEW_25_07.json'
                        , 'r')
                as json_file: loaded_optPol = json.load(json_file)
    loaded_optPol = {ast.literal_eval(key): value for key
                                              , value in loaded_optPol.items()}
    Greensim_time = -1


def AAPILoad():
    AKIPrintString("AAPILoad")
    return 0


def AAPIInit():
    AKIPrintString("AAPIInit")
    initValues()
    return 0
```

```
def AAPIManage(time, timeSta, timeTrans, acycle):
    global countall_list, det_list, count_list,countall,queue_len
            ,quelenallv1, quelenallv2, quelenallv3,quelenallv4,Greensim_time
            , ResultsAll, EntranceTimes, ExitTimes, Traveltimes, Waitingtimes
            , veh_list, StopResults
    detectors = AKIDetGetNumberDetectors()
    det_list = []
    count_list = []
    for i in range(detectors):
        det_list.insert(i,AKIDetGetIdDetector(i))
        count_list.insert(i,AKIDetGetCounterCyclebyId(det_list[i], 0))
        if count_list[i] <0:
            count_list[i] = 0
        countall_list[i] = countall_list[i] + count_list[i]
    count = AKIDetGetCounterCyclebyId(1167, 0)
    simtime = AKIGetCurrentSimulationTime()
    if count < 0:
        count = 0
    countall = countall + count
    nodeID = ECIGetJunctionId(0)
    vecofexits = range(ECIGetNumberSignalGroups(nodeID))
    IDsofsignal = [ECIGetAimsunIdofSignalGroup(nodeID, 1)
                   ,ECIGetAimsunIdofSignalGroup(nodeID, 2)
                   ,ECIGetAimsunIdofSignalGroup(nodeID, 3)
                   ,ECIGetAimsunIdofSignalGroup(nodeID, 4)]
    Phases = ECIGetNumberPhases(nodeID)
    phaseno = range(1,Phases +1)
    stateKap = ECIGetStateSem(nodeID, 0)
    stateSlm = ECIGetStateSem(nodeID, 1)
    stateGzr = ECIGetStateSem(nodeID, 2)
    stateMsd = ECIGetStateSem(nodeID, 3)

    if AKIDetGetCounterCyclebyId(det_list[1], 0) == 1:
```

```
    EntranceTimes[0].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[5], 0) == 1:
    EntranceTimes[1].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[3], 0) == 1:
    EntranceTimes[2].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[7], 0) == 1:
    EntranceTimes[3].append(simtime)

if AKIDetGetCounterCyclebyId(det_list[0], 0) == 1:
        ExitTimes[0].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[4], 0) == 1:
        ExitTimes[1].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[2], 0) == 1:
        ExitTimes[2].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[6], 0) == 1:
        ExitTimes[3].append(simtime)

if stateKap != 0:
    statesem = "Kappara Open"
elif stateSlm != 0:
    statesem = "Sliema Open"
elif stateGzr != 0:
    statesem = "Gzira Open"
elif stateMsd != 0:
    statesem = "Msida Open"
else:
    statesem = "Yellow"

## Sliema Queue
queue_len[0] = countall_list[1]-countall_list[0]
## Msida Queue
queue_len[1] = countall_list[3]-countall_list[2]
## Gzira Queue
queue_len[2] = countall_list[5]-countall_list[4]
```

```
## Kappara Queue
queue_len[3] = countall_list[7]-countall_list[6]
queue_len = [queue_len[0],queue_len[2],queue_len[1],queue_len[3]]

def state_changer_AimV2(state, max_values, range_size):
    # Create a copy to avoid modifying the original state
    changed_state = state[::]
    for i in range(1, 5):
        current_value = state[i]
        max_limit = max_values[i - 1]

        if current_value < max_limit:
            start_range = (current_value // range_size) * range_size
            range_number = start_range // range_size + 1
            changed_state[i] = range_number
        else:
            # Set a default range_number
            # when the value exceeds the max_limit
            default_range_numbers = [8, 11, 9, 21]
            changed_state[i] = default_range_numbers[i - 1]

    changed_state = tuple(changed_state)
    return changed_state

def current_state(statesem,queue_len):
    state = []
    state.append([statesem,queue_len])
    newform = [state[0][0]]
    newform.extend(state[0][1])
    state = state_changer_AimV2(newform,[35,50,100,40],5)
    #state=newform

    return state
```

```
Greenofphase = simtime - ECIGetStartingTimePhase(nodeID)

if simtime == 0:
      ECIChangeTimingPhase(nodeID, phaseno[0], 100, timeSta)
      ECIChangeTimingPhase(nodeID, phaseno[2], 100, timeSta)
      ECIChangeTimingPhase(nodeID, phaseno[4], 100, timeSta)
      ECIChangeTimingPhase(nodeID, phaseno[6], 100, timeSta)

## 2 Sliema, 3 Gzira , 4 Msida, 1 Kappara
#AKIPrintString(str(ECIGetCurrentStateofSignalGroup(nodeID, 2)))

slmlight = ECIGetCurrentStateofSignalGroup(nodeID, 2)
gzrlight = ECIGetCurrentStateofSignalGroup(nodeID, 3)
msdlight = ECIGetCurrentStateofSignalGroup(nodeID, 4)
kaplight = ECIGetCurrentStateofSignalGroup(nodeID, 1)
if simtime == 0:
    StopResults = [[simtime,slmlight,gzrlight,msdlight,kaplight]]
elif simtime > 0:
    StopResults.append([simtime,slmlight,gzrlight,msdlight,kaplight])

if Greenofphase != 0 and statesem != "Yellow":
    cur_state = current_state(statesem, queue_len)
    takeaction = loaded_optPol[cur_state]

    remgrnsimtime = Greenofphase%10
    if remgrnsimtime == 0:

        #AKIPrintString("Queue length: " + str(queue_len))
        lightnos = ECIGetNumberSem(nodeID)-1

        if takeaction == 'Stay the same':
            if current_state(statesem, queue_len)[0] == 'Sliema Open':
                ECIChangeTimingPhase(nodeID, phaseno[0]
                                          , Greenofphase + 100, timeSta)
```

```
                elif current_state(statesem, queue_len)[0] == 'Gzira Open':
                    ECIChangeTimingPhase(nodeID, phaseno[2]
                                         , Greenofphase + 100, timeSta)
                elif current_state(statesem, queue_len)[0] == 'Msida Open':
                    ECIChangeTimingPhase(nodeID, phaseno[4]
                                         , Greenofphase + 100, timeSta)
                elif current_state(statesem, queue_len)[0] == 'Kappara Open':
                    ECIChangeTimingPhase(nodeID, phaseno[6]
                                         , Greenofphase + 100, timeSta)
            else:
                if current_state(statesem, queue_len)[0] == 'Sliema Open':
                    if takeaction == 'Change':
                        ECIChangeTimingPhase(nodeID, phaseno[6], 150, timeSta)
                        ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                            ,phaseno[2],timeSta,simtime,1.0)
                elif current_state(statesem, queue_len)[0] == 'Gzira Open':
                    if takeaction == 'Change':
                        ECIChangeTimingPhase(nodeID, phaseno[0], 150, timeSta)
                        ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                            ,phaseno[4],timeSta,simtime,1.0)
                elif current_state(statesem, queue_len)[0] == 'Msida Open':
                    if takeaction == 'Change':
                        ECIChangeTimingPhase(nodeID, phaseno[2], 150, timeSta)
                        ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                            ,phaseno[6],timeSta,simtime,1.0)
                elif current_state(statesem, queue_len)[0] == 'Kappara Open':
                    if takeaction == 'Change':
                        ECIChangeTimingPhase(nodeID, phaseno[4], 150, timeSta)
                        ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                            ,phaseno[0],timeSta,simtime,1.0)
    # Read the number of sections present on the road network
    num_sections = AKIInfNetNbSectionsANG()
    idofsec = [922,906,932,919]
    for i in range(len(idofsec)):
```

```
        # Read the total number of vehicles in a section.
        num_vehicles = AKIVehStateGetNbVehiclesSection(idofsec[i], True)
        #AKIPrintString("num_vehicles: " + str(num_vehicles))

        for j in range(num_vehicles):
            # Read the information of a vehicle in a section
            infVehS = AKIVehStateGetVehicleInfSection(idofsec[i], j)
            veh_id = infVehS.idVeh
            #AKIPrintString("vehID: " + str(veh_id))
            GenT = infVehS.SystemGenerationT
            SysT = infVehS.SystemEntranceT
            VQ = SysT - GenT
            #AKIPrintString("VQ: " + str(VQ))
            if idofsec[i] == 922:
                loc = "Sliema"
            elif idofsec[i] == 906:
                loc = "Gzira"
            elif idofsec[i] == 932:
                loc = "Msida"
            else:
                loc = "Kappara"

            veh_list.append([loc, veh_id, VQ ])

    CheckAV = simtime%10

    columns = ['SimTime','QueueLenSlm','CapSlm','QueueLenGzr'
                ,'CapGzr','QueueLenMsd','CapMsd','QueueLenKap','CapKap']
    if simtime != 0:
        if CheckAV == 0:
            ResultsAll.append([simtime,queue_len[0],(queue_len[0]/35)*100
                               ,queue_len[2],(queue_len[2]/50)*100
                               ,queue_len[1],(queue_len[1]/40)*100
                               ,queue_len[3],(queue_len[3]/100)*100])
```

```
            #AKIPrintString("Departing cars: " + str(ExitTimes[0]))


    if simtime == 3899:
        ## Sliema
        carcntv1 = countall_list[1]
        AvArrRtev1 = carcntv1/3900
        Deprtv1 = countall_list[0]/3900
        ## Msida
        carcntv2 = countall_list[3]
        AvArrRtev2 = carcntv2/3900
        Deprtv2 = countall_list[2]/3900
        ## Gzira
        carcntv3 = countall_list[5]
        AvArrRtev3 = carcntv3/3900
        Deprtv3 = countall_list[4]/3900
        ## Kappara
        carcntv4 = countall_list[7]
        AvArrRtev4 = carcntv4/3900
        Deprtv4 = countall_list[6]/3900
        AvQlnv1 = quelenallv1/3900
        AvQlnv2 = quelenallv2/3900
        AvQlnv3 = quelenallv3/3900
        AvQlnv4 = quelenallv4/3900
        AvWaitTmev1 = AvQlnv1/AvArrRtev1
        AvWaitTmev2 = AvQlnv2/AvArrRtev2
        AvWaitTmev3 = AvQlnv3/AvArrRtev3
        AvWaitTmev4 = AvQlnv4/AvArrRtev4
        AllroundAvg = (AvWaitTmev1 + AvWaitTmev2
                        + AvWaitTmev3 + AvWaitTmev4)/4


        ## Sliema Delay
        for i in range(max(len(EntranceTimes[0]), len(ExitTimes[0]))):
            if i < len(EntranceTimes[0]) and i < len(ExitTimes[0]):
                Delay = (ExitTimes[0][i] - EntranceTimes[0][i])-10
```

```
            WaitTme = ExitTimes[0][i] - EntranceTimes[0][i]
            Waitingtimes[0].append(WaitTme)
            if Delay >= 0:
                Traveltimes[0].append(Delay)
            else:
                Traveltimes[0].append(0)
        else:
            break

    ## Gzira Delay
    for i in range(max(len(EntranceTimes[1]), len(ExitTimes[1]))):
        if i < len(EntranceTimes[1]) and i < len(ExitTimes[1]):
            Delay = (ExitTimes[1][i] - EntranceTimes[1][i])-15
            WaitTme = ExitTimes[1][i] - EntranceTimes[1][i]
            Waitingtimes[1].append(WaitTme)
            if Delay >= 0:
                Traveltimes[1].append(Delay)
            else:
                Traveltimes[1].append(0)
        else:
            break

    ## Msida Delay
    for i in range(max(len(EntranceTimes[2]), len(ExitTimes[2]))):
        if i < len(EntranceTimes[2]) and i < len(ExitTimes[2]):
            Delay = (ExitTimes[2][i] - EntranceTimes[2][i])-12
            WaitTme = ExitTimes[2][i] - EntranceTimes[2][i]
            Waitingtimes[2].append(WaitTme)
            if Delay >= 0:
                Traveltimes[2].append(Delay)
            else:
                Traveltimes[2].append(0)
        else:
            break
```

```
## kap delay
for i in range(max(len(EntranceTimes[3]), len(ExitTimes[3]))):
    if i < len(EntranceTimes[3]) and i < len(ExitTimes[3]):
        Delay = (ExitTimes[3][i] - EntranceTimes[3][i])-31
        WaitTme = ExitTimes[3][i] - EntranceTimes[3][i]
        Waitingtimes[3].append(WaitTme)
        if Delay >= 0:
            Traveltimes[3].append(Delay)
        else:
            Traveltimes[3].append(0)

    else:
        break
#filename = 'ResultsNewMornRushHour.csv'
Othercolumns = ['CarNo','EntrSlm','EntrGzr','EntrMsd','EntrKap'
               ,'WaitSlm','WaitGzr','WaitMsd','WaitKap']
Carno = [i + 1 for i in range(max(len(sublist)
                    for sublist in Waitingtimes))]
#TransCarno = [[item] for item in Carno]
#transposed_Traveltimes = list(map(list, zip(*Traveltimes)))
#transposed_Waitingtimes = list(map(list, zip(*Waitingtimes)))
for i in range(max(len(sublist) for sublist in Waitingtimes)):
    try:
        EntrSlm = EntranceTimes[0][i]
    except (ValueError, TypeError,IndexError):
        EntrSlm = None
    try:
        EntrGzr = EntranceTimes[1][i]
    except (ValueError, TypeError,IndexError):
        EntrGzr = None
    try:
        EntrMsd = EntranceTimes[2][i]
    except (ValueError, TypeError,IndexError):
        EntrMsd = None
```

```
        try:
            EntrKap = EntranceTimes[3][i]
        except (ValueError, TypeError,IndexError):
            EntrKap = None
        ###################
        try:
            WaitSlm = Waitingtimes[0][i]
        except (ValueError, TypeError,IndexError):
            WaitSlm = None
        try:
            WaitGzr = Waitingtimes[1][i]
        except (ValueError, TypeError,IndexError):
            WaitGzr = None
        try:
            WaitMsd = Waitingtimes[2][i]
        except (ValueError, TypeError,IndexError):
            WaitMsd = None
        try:
            WaitKap = Waitingtimes[3][i]
        except (ValueError, TypeError,IndexError):
            WaitKap = None
        if i == 0:
            OtherResults = [[Carno[i],EntrSlm,EntrGzr,EntrMsd
                                ,EntrKap,WaitSlm,WaitGzr,WaitMsd,WaitKap]]
        else:
            OtherResults.append([Carno[i],EntrSlm,EntrGzr,EntrMsd
                                    ,EntrKap,WaitSlm,WaitGzr
                                    ,WaitMsd,WaitKap])

    #Open the file in write mode and create a CSV writer object
    Colvehs = ['Location','CarID','Virtual Queue Time']
    Colstops = ['Simtime','Slm','Gzr', 'Msd','Kap']
    blankrow = []
    base_path = r'C:\Users\jform\Desktop\RawAimsunResultsNew\Morning_MDP'
```

```
        file_name = 'MorningMDP' + str(ANGConnGetReplicationId())
        extension = '.csv'
        ResPath = os.path.join(base_path, file_name + extension)
        with open(ResPath, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(columns)
            writer.writerows(ResultsAll)
            writer.writerow(blankrow)
            writer.writerow(Othercolumns)
            writer.writerows(OtherResults)
            writer.writerow(blankrow)
            writer.writerow(Colvehs)
            writer.writerows(veh_list)
            writer.writerow(blankrow)
            writer.writerow(Colstops)
            writer.writerows(StopResults)

    return 0

def AAPIPostManage(time, timeSta, timeTrans, acycle):
    #AKIPrintString("AAPIPostManage")
    return 0

def AAPISimulationReady():
return 0

def AAPIFinish():
    AKIPrintString("AAPIFinish")
    return 0

def AAPIUnLoad():
    AKIPrintString("AAPIUnLoad")
    return 0
```

Appendix C

Python Code for Controlling Simulation through the Aimsun API - Combined 9 hour simulation

```
from AAPI import *
from array import *
import math
import json
import ast
#import numpy as np
#import pandas as pd
import csv
import os
#import openpyxl
#import xlsxwriter

def initValues():
    global countall_list, det_list,count_list,countall,queue_len
            ,quelenallv1, quelenallv2, quelenallv3,quelenallv4,loaded_optPol
            ,loaded_optPol_Morn,loaded_optPol_Noon,loaded_optPol_Even
            , Greensim_time, ResultsAll,EntranceTimes, ExitTimes, Traveltimes
            , Waitingtimes, veh_list, StopResults, mornRt,noonRt, evenRt
```

```
        , ArRt_list,Chosen_One,AvG, Test_list, mornQls
        , noonQls, evenQls, last_policy,Chosen_Pols
#countall should be 0 times the number of detectors
#queue_len should have length equal to
#the number of roads which have detectors
countall_list= [0]*8
det_list = []
count_list = []
ResultsAll = []
veh_list = []
StopResults = []
EntranceTimes = [[], [], [], []]
ExitTimes = [[], [], [], []]
Traveltimes = [[], [], [], []]
Waitingtimes = [[], [], [], []]
ArRt_list = [[], [], [], []]
Test_list = [[], [], [],[]]
Chosen_Pols = [[],[], [], []]
AvG = [0,0,0,0]
Chosen_One = None
countall = 0
quelenallv1 = 0
quelenallv2 = 0
quelenallv3 = 0
quelenallv4 = 0
queue_len = [0]*4
# Define a dictionary of file paths with custom variable names as keys
file_paths = {'loaded_optPol_Morn': r'C:\Users\jform\Desktop
                \converted_OptimalPol_Morning_NoYT_DF075NEW_25_07.json',
              'loaded_optPol_Noon': r'C:\Users\jform\Desktop
                \converted_OptimalPol_Noon_NoYT_DF075NEW_25_07.json',
              'loaded_optPol_Even': r'C:\Users\jform\Desktop
                \converted_OptimalPol_Even_NoYT_DF075NEW_25_07.json',}
# Loop over each file path and custom variable name
```

```
    for var_name, file_path in file_paths.items():
        with open(file_path, 'r') as json_file:
            loaded_optPol = json.load(json_file)
            # Convert keys to tuples if necessary
            loaded_optPol = {ast.literal_eval(key): value for key
                                , value in loaded_optPol.items()}
            # Assign data to custom variable name
            globals()[var_name] = loaded_optPol

    mornRt = [0.064358974, 0.04025641 , 0.136153846 , 0.173846154]
    noonRt = [0.078717949,0.075641026, 0.12974359, 0.138205128]
    evenRt = [0.122307692,0.059487179 , 0.097435897, 0.148717949]

    mornQls = [23.76463307,16.10079576,19.38443855,52.30742706]
    noonQls = [26.88744474,22.32793988,32.79177719,57.23032714]
    evenQls = [22.8755084,10.93244916,21.75579134,40.05747126]

# Now each file's data can be accessed using its custom variable name,
#like loaded_optPol_A, loaded_optPol_B, etc.
    Greensim_time = -1
    last_policy = None

def AAPILoad():
    AKIPrintString("AAPILoad")
    return 0

def AAPIInit():
    AKIPrintString("AAPIInit")
    initValues()
    return 0

def AAPIManage(time, timeSta, timeTrans, acycle):
    global countall_list, det_list, count_list,countall,queue_len
            ,quelenallv1, quelenallv2, quelenallv3,quelenallv4
```

```
        ,Greensim_time, ResultsAll, EntranceTimes, ExitTimes
        , Traveltimes, Waitingtimes, veh_list, StopResults, mornRt
        , noonRt, evenRt, ArRt_list, Chosen_One,AvG, Test_list
        , mornQls, noonQls, evenQls, last_policy,Chosen_Pols
detectors = AKIDetGetNumberDetectors()
det_list = []
count_list = []
for i in range(detectors):
    det_list.insert(i,AKIDetGetIdDetector(i))
    count_list.insert(i,AKIDetGetCounterCyclebyId(det_list[i], 0))
    if count_list[i] <0:
        count_list[i] = 0
    countall_list[i] = countall_list[i] + count_list[i]
count = AKIDetGetCounterCyclebyId(1167, 0)
simtime = AKIGetCurrentSimulationTime()
if count < 0:
    count = 0
countall = countall + count
nodeID = ECIGetJunctionId(0)
vecofexits = range(ECIGetNumberSignalGroups(nodeID))
IDsofsignal = [ECIGetAimsunIdofSignalGroup(nodeID, 1)
               ,ECIGetAimsunIdofSignalGroup(nodeID, 2)
               ,ECIGetAimsunIdofSignalGroup(nodeID, 3)
               ,ECIGetAimsunIdofSignalGroup(nodeID, 4)]
Phases = ECIGetNumberPhases(nodeID)
phaseno = range(1,Phases +1)
stateKap = ECIGetStateSem(nodeID, 0)
stateSlm = ECIGetStateSem(nodeID, 1)
stateGzr = ECIGetStateSem(nodeID, 2)
stateMsd = ECIGetStateSem(nodeID, 3)

if AKIDetGetCounterCyclebyId(det_list[1], 0) == 1:
    EntranceTimes[0].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[5], 0) == 1:
```

```
    EntranceTimes[1].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[3], 0) == 1:
    EntranceTimes[2].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[7], 0) == 1:
    EntranceTimes[3].append(simtime)


if AKIDetGetCounterCyclebyId(det_list[0], 0) == 1:
        ExitTimes[0].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[4], 0) == 1:
        ExitTimes[1].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[2], 0) == 1:
        ExitTimes[2].append(simtime)
if AKIDetGetCounterCyclebyId(det_list[6], 0) == 1:
        ExitTimes[3].append(simtime)


if stateKap != 0:
    statesem = "Kappara Open"
elif stateSlm != 0:
    statesem = "Sliema Open"
elif stateGzr != 0:
    statesem = "Gzira Open"
elif stateMsd != 0:
    statesem = "Msida Open"
else:
    statesem = "Yellow"


## Sliema Queue
queue_len[0] = countall_list[1]-countall_list[0]
## Msida Queue
queue_len[1] = countall_list[3]-countall_list[2]
## Gzira Queue
queue_len[2] = countall_list[5]-countall_list[4]
## Kappara Queue
queue_len[3] = countall_list[7]-countall_list[6]
```

```
queue_len = [queue_len[0],queue_len[2],queue_len[1],queue_len[3]]

def state_changer_AimV2(state, max_values, range_size):
    # Create a copy to avoid modifying the original state
    changed_state = state[::]

    for i in range(1, 5):
        current_value = state[i]
        max_limit = max_values[i - 1]

        if current_value < max_limit:
            start_range = (current_value // range_size) * range_size
            range_number = start_range // range_size + 1
            changed_state[i] = range_number
        else:
            #Set a default range_number when the value exceeds max_lim
            default_range_numbers = [8, 11, 9, 21]
            changed_state[i] = default_range_numbers[i - 1]

    changed_state = tuple(changed_state)

    return changed_state

def current_state(statesem,queue_len):

    state = []
    state.append([statesem,queue_len])
    newform = [state[0][0]]
    newform.extend(state[0][1])
    state = state_changer_AimV2(newform,[35,50,100,40],5)
    #state=newform

    return state
```

```
Greenofphase = simtime - ECIGetStartingTimePhase(nodeID)

if simtime == 0:
      ECIChangeTimingPhase(nodeID, phaseno[0], 100, timeSta)
      ECIChangeTimingPhase(nodeID, phaseno[2], 100, timeSta)
      ECIChangeTimingPhase(nodeID, phaseno[4], 100, timeSta)
      ECIChangeTimingPhase(nodeID, phaseno[6], 100, timeSta)

## 2 Sliema, 3 Gzira , 4 Msida, 1 Kappara
#AKIPrintString(str(ECIGetCurrentStateofSignalGroup(nodeID, 2)))

slmlight = ECIGetCurrentStateofSignalGroup(nodeID, 2)
gzrlight = ECIGetCurrentStateofSignalGroup(nodeID, 3)
msdlight = ECIGetCurrentStateofSignalGroup(nodeID, 4)
kaplight = ECIGetCurrentStateofSignalGroup(nodeID, 1)
if simtime == 0:
    StopResults = [[simtime,slmlight,gzrlight,msdlight,kaplight]]
elif simtime > 0:
    StopResults.append([simtime,slmlight,gzrlight,msdlight,kaplight])

simtime_Ten = simtime%10
simtime_fivemins = simtime%300
simtime_tenmins = simtime%600
simtime_hour = simtime%3600
simtime_twominhalf = simtime%150

if simtime_Ten == 0 and simtime != 0 :

#     ## Sliema
     carcntv1 = countall_list[1]
     AvArrRtev1 = carcntv1/simtime
#     ## Gzira
     carcntv3 = countall_list[5]
     AvArrRtev3 = carcntv3/simtime
```

```
#     ## Msida
      carcntv2 = countall_list[3]
      AvArrRtev2 = carcntv2/simtime
#     ## Kappara
      carcntv4 = countall_list[7]
      AvArrRtev4 = carcntv4/simtime
      ####################
      ArRt_list[0].append(AvArrRtev1)# Add 1 to the first sublist
      Test_list[0].append(queue_len[0])
      ArRt_list[1].append(AvArrRtev3)  # Add 2 to the second sublist
      Test_list[1].append(queue_len[2])
      ArRt_list[2].append(AvArrRtev2)  # Add 3 to the third sublist
      Test_list[2].append(queue_len[1])
      ArRt_list[3].append(AvArrRtev4)
      Test_list[3].append(queue_len[3])
      #AKIPrintString(str(ArRt_list))
      # if simtime_fivemins == 0 and simtime != 0:
      #     AvG = [sum(sublist) / len(sublist) for sublist in ArRt_list]
      #     AKIPrintString(str(AvG))
      if simtime_fivemins == 10 and simtime != 10 :
           ArRt_list = [[], [], [], []]
           Test_list = [[], [], [], []]
      # if simtime_hour == 0 and simtime != 0 :
      #      ArRt_list = [[], [], [], []]
      #AKIPrintString(str(ArRt_list))


cur_state = current_state(statesem, queue_len)


if simtime < 300:
    takeaction = loaded_optPol_Morn[cur_state]
    checker = "Simtime less than 300, hence Morn policy"
    last_policy = "Morn policy"
else:
    if simtime_fivemins ==0 and simtime != 0:
```

```
        AvG = [sum(sublist) / len(sublist) for sublist in ArRt_list]
        AvG_QL = [sum(sublist) / len(sublist) for sublist in Test_list]

        def euclidean_distance(list1, list2):
            return sum((a - b) ** 2
                       for a, b in zip(list1, list2)) ** 0.5

        dist_morn = euclidean_distance(AvG, mornRt)
        dist_noon = euclidean_distance(AvG, noonRt)
        dist_even = euclidean_distance(AvG, evenRt)

        dist_morn_QL = euclidean_distance(AvG_QL, mornQls)
        dist_noon_QL = euclidean_distance(AvG_QL, noonQls)
        dist_even_QL = euclidean_distance(AvG_QL, evenQls)

        distances = {"morn": dist_morn, "noon": dist_noon
                     , "even": dist_even}
        closest_list = min(distances, key=distances.get)

        if closest_list == "morn":
            takeaction = loaded_optPol_Morn[cur_state]
            last_policy = "Morn policy"
            checker = "Morn policy"
        elif closest_list == "noon":
            takeaction = loaded_optPol_Noon[cur_state]
            last_policy = "Noon policy"
            checker = "Noon policy"
        elif closest_list == "even":
            takeaction = loaded_optPol_Even[cur_state]
            last_policy = "Even policy"
            checker = "Even policy"

    else:
        if last_policy == "Morn policy":
```

```
            takeaction = loaded_optPol_Morn[cur_state]
        elif last_policy == "Noon policy":
            takeaction = loaded_optPol_Noon[cur_state]
        else:
            takeaction = loaded_optPol_Even[cur_state]

        checker = "Previous chosen policy being used"

# # Choosing policy based on time. (cheating)
# if simtime < 11100:
#     takeaction = loaded_optPol_Morn[cur_state]
#     checker = "Morn policy"
# elif simtime >= 11100 and simtime < 21900:
#     takeaction = loaded_optPol_Noon[cur_state]
#     checker = "Noon policy"
# else:
#     takeaction = loaded_optPol_Even[cur_state]
#     checker = "Even policy"

# if simtime_fivemins == 0 and simtime != 0:
#     if simtime < 11100:
#         Chosen_Pols[0].append(simtime)
#         Chosen_Pols[1].append(checker)
#     elif simtime >= 11100 and simtime < 21900:
#           Chosen_Pols[2].append(checker)
#     else:
#           Chosen_Pols[3].append(checker)

if Greenofphase != 0 and statesem != "Yellow":

    remgrnsimtime = Greenofphase%10

    if remgrnsimtime == 0:
```

```
    lightnos = ECIGetNumberSem(nodeID)-1


    if takeaction == 'Stay the same':
        if current_state(statesem, queue_len)[0] == 'Sliema Open':
            ECIChangeTimingPhase(nodeID, phaseno[0]
                                 , Greenofphase + 100, timeSta)
        elif current_state(statesem, queue_len)[0] == 'Gzira Open':
            ECIChangeTimingPhase(nodeID, phaseno[2]
                                 , Greenofphase + 100, timeSta)
        elif current_state(statesem, queue_len)[0] == 'Msida Open':
            ECIChangeTimingPhase(nodeID, phaseno[4]
                                 , Greenofphase + 100, timeSta)
        elif current_state(statesem, queue_len)[0] == 'Kappara Open':
            ECIChangeTimingPhase(nodeID, phaseno[6]
                                 , Greenofphase + 100, timeSta)
    else:
        if current_state(statesem, queue_len)[0] == 'Sliema Open':
            if takeaction == 'Change':
                ECIChangeTimingPhase(nodeID, phaseno[6], 150, timeSta)
                ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                        , phaseno[2],  timeSta,  simtime,  1.0)
        elif current_state(statesem, queue_len)[0] == 'Gzira Open':
            if takeaction == 'Change':
                ECIChangeTimingPhase(nodeID, phaseno[0], 150, timeSta)
                ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                        , phaseno[4],  timeSta,  simtime,  1.0)
        elif current_state(statesem, queue_len)[0] == 'Msida Open':
            if takeaction == 'Change':
                ECIChangeTimingPhase(nodeID, phaseno[2], 150, timeSta)
                ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                        , phaseno[6],  timeSta,  simtime,  1.0)

        elif current_state(statesem, queue_len)[0] == 'Kappara Open':
            if takeaction == 'Change':
```

```
                    ECIChangeTimingPhase(nodeID, phaseno[4], 150, timeSta)
                    ECIChangeDirectPhaseWithInterphaseTransition(nodeID
                            , phaseno[0], timeSta, simtime, 1.0)

# Read the number of sections present on the road network
num_sections = AKIInfNetNbSectionsANG()
#idofsec = AKIInfNetGetSectionANGId(0)
idofsec = [922,906,932,919]
for i in range(len(idofsec)):
    # Read the total number of vehicles in a section.
    num_vehicles = AKIVehStateGetNbVehiclesSection(idofsec[i], True)
    #AKIPrintString("num_vehicles: " + str(num_vehicles))

    for j in range(num_vehicles):
        # Read the information of a vehicle in a section
        infVehS = AKIVehStateGetVehicleInfSection(idofsec[i], j)
        veh_id = infVehS.idVeh
        #AKIPrintString("vehID: " + str(veh_id))
        GenT = infVehS.SystemGenerationT
        SysT = infVehS.SystemEntranceT
        VQ = SysT - GenT
        #AKIPrintString("VQ: " + str(VQ))
        if idofsec[i] == 922:
            loc = "Sliema"
        elif idofsec[i] == 906:
            loc = "Gzira"
        elif idofsec[i] == 932:
            loc = "Msida"
        else:
            loc = "Kappara"

        veh_list.append([loc, veh_id, VQ ])

CheckAV = simtime%10
```

```
columns = ['SimTime','QueueLenSlm','CapSlm','QueueLenGzr','CapGzr'
                ,'QueueLenMsd','CapMsd','QueueLenKap','CapKap']
if simtime != 0:
    if CheckAV == 0:
        ResultsAll.append([simtime,queue_len[0],(queue_len[0]/35)*100,queue


if simtime == 32699:
    ## Sliema
    carcntv1 = countall_list[1]
    ## Msida
    carcntv2 = countall_list[3]
    ## Gzira
    carcntv3 = countall_list[5]

    ## Kappara
    carcntv4 = countall_list[7]
    AKIPrintString("Morning Choices:" + str(Chosen_Pols[1]))
    AKIPrintString("Noon Choices:" + str(Chosen_Pols[2]))
    AKIPrintString("Evening Choices:" + str(Chosen_Pols[3]))

    ## Sliema Delay
    for i in range(max(len(EntranceTimes[0]), len(ExitTimes[0]))):
        if i < len(EntranceTimes[0]) and i < len(ExitTimes[0]):
            Delay = (ExitTimes[0][i] - EntranceTimes[0][i])-10
            WaitTme = ExitTimes[0][i] - EntranceTimes[0][i]
            Waitingtimes[0].append(WaitTme)
            if Delay >= 0:
                Traveltimes[0].append(Delay)
            else:
                Traveltimes[0].append(0)
        else:
            break
```

```
## Gzira Delay
for i in range(max(len(EntranceTimes[1]), len(ExitTimes[1]))):
    if i < len(EntranceTimes[1]) and i < len(ExitTimes[1]):
        Delay = (ExitTimes[1][i] - EntranceTimes[1][i])-15
        WaitTme = ExitTimes[1][i] - EntranceTimes[1][i]
        Waitingtimes[1].append(WaitTme)
        if Delay >= 0:
            Traveltimes[1].append(Delay)
        else:
            Traveltimes[1].append(0)
    else:
        break

## Msida Delay
for i in range(max(len(EntranceTimes[2]), len(ExitTimes[2]))):
    if i < len(EntranceTimes[2]) and i < len(ExitTimes[2]):
        Delay = (ExitTimes[2][i] - EntranceTimes[2][i])-12
        WaitTme = ExitTimes[2][i] - EntranceTimes[2][i]
        Waitingtimes[2].append(WaitTme)
        if Delay >= 0:
            Traveltimes[2].append(Delay)
        else:
            Traveltimes[2].append(0)
    else:
        break

## kap delay
for i in range(max(len(EntranceTimes[3]), len(ExitTimes[3]))):
    if i < len(EntranceTimes[3]) and i < len(ExitTimes[3]):
        Delay = (ExitTimes[3][i] - EntranceTimes[3][i])-31
        WaitTme = ExitTimes[3][i] - EntranceTimes[3][i]
        Waitingtimes[3].append(WaitTme)
        if Delay >= 0:
            Traveltimes[3].append(Delay)
```

```
            else:
                Traveltimes[3].append(0)


        else:
            break


    #filename = 'ResultsNewMornRushHour.csv'
    Othercolumns = ['CarNo','EntrSlm','EntrGzr','EntrMsd'
                        ,'EntrKap','WaitSlm','WaitGzr'
                        ,'WaitMsd','WaitKap']
    Carno = [i + 1 for i in range(max(len(sublist)
                            for sublist in Waitingtimes))]
    for i in range(max(len(sublist) for sublist in Waitingtimes)):
        try:
            EntrSlm = EntranceTimes[0][i]
        except (ValueError, TypeError,IndexError):
            EntrSlm = None
        try:
            EntrGzr = EntranceTimes[1][i]
        except (ValueError, TypeError,IndexError):
            EntrGzr = None
        try:
            EntrMsd = EntranceTimes[2][i]
        except (ValueError, TypeError,IndexError):
            EntrMsd = None
        try:
            EntrKap = EntranceTimes[3][i]
        except (ValueError, TypeError,IndexError):
            EntrKap = None
        try:
            WaitSlm = Waitingtimes[0][i]
        except (ValueError, TypeError,IndexError):
            WaitSlm = None
        try:
```

```
            WaitGzr = Waitingtimes[1][i]
        except (ValueError, TypeError,IndexError):
            WaitGzr = None
        try:
            WaitMsd = Waitingtimes[2][i]
        except (ValueError, TypeError,IndexError):
            WaitMsd = None
        try:
            WaitKap = Waitingtimes[3][i]
        except (ValueError, TypeError,IndexError):
            WaitKap = None


        if i == 0:
            OtherResults = [[Carno[i],EntrSlm,EntrGzr,EntrMsd
                                ,EntrKap,WaitSlm,WaitGzr
                                ,WaitMsd,WaitKap]]
        else:
            OtherResults.append([Carno[i],EntrSlm,EntrGzr,EntrMsd
                                      ,EntrKap,WaitSlm,WaitGzr
                                      ,WaitMsd,WaitKap])


    #Open the file in write mode and create a CSV writer object
    Colvehs = ['Location','CarID','Virtual Queue Time']
    Colstops = ['Simtime','Slm','Gzr', 'Msd','Kap']
    ColPolChosen = ['Simtime','MornPols','NoonPols','EvenPols']
    blankrow = []
    base_path = r'C:\Users\jform\Desktop\Results9h\DET'
    file_name = 'DET_' + str(ANGConnGetReplicationId())
    extension = '.csv'
    ResPath = os.path.join(base_path, file_name + extension)
    with open(ResPath, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(columns)
        writer.writerows(ResultsAll)
```

```
            writer.writerow(blankrow)
            writer.writerow(Othercolumns)
            writer.writerows(OtherResults)
            writer.writerow(blankrow)
            writer.writerow(Colvehs)
            writer.writerows(veh_list)
            writer.writerow(blankrow)
            writer.writerow(Colstops)
            writer.writerows(StopResults)

    return 0

def AAPIPostManage(time, timeSta, timeTrans, acycle):
    #AKIPrintString("AAPIPostManage")
    return 0

def AAPISimulationReady():
return 0

def AAPIFinish():
    AKIPrintString("AAPIFinish")
    return 0

def AAPIUnLoad():
    AKIPrintString("AAPIUnLoad")
    return 0
```

Appendix D

Transforming Aimsun Results Code

```
import pandas as pd
import numpy as np
from openpyxl import load_workbook
from openpyxl import Workbook
import os

def TransResults(df, file):

    column_names = df.columns

    #print(df.dtypes)

    def convert_to_numeric(val):
        try:
            return pd.to_numeric(val)
        except ValueError:
            return val

    # Apply the conversion function to the entire DataFrame
    df = df.applymap(convert_to_numeric)
```

```
for i in range(len(df['QueueLenSlm'])):
    if isinstance(df['QueueLenSlm'][i],str) == True:
        split_index = i
        first_df = df.iloc[:split_index]
        dfpt2 = df.iloc[split_index:].copy()
        dfpt2.columns = dfpt2.iloc[0]
        dfpt2.drop(dfpt2.index[0], inplace=True)
        dfpt2.reset_index(drop=True, inplace=True)
        break

    else:
        next

# indices = dfpt2.index
# indices_list = indices.tolist()

for i in range(len(dfpt2)):
    if isinstance(dfpt2['CarNo'][i],str) == True:
        split_index = i
        second_df = dfpt2.iloc[:split_index]
        dfpt3 = dfpt2.iloc[split_index:].copy()
        dfpt3.columns = dfpt3.iloc[0]
        dfpt3.drop(dfpt3.index[0], inplace=True)
        dfpt3.reset_index(drop=True, inplace=True)
        break

    else:
        next

for i in range(len(dfpt3)):
    if dfpt3['Location'][i] == 'Simtime':
        print(i)
        split_index = i
```

```
        third_df = dfpt3.iloc[:split_index]
        fourth_df = dfpt3.iloc[split_index:].copy()
        fourth_df.columns = fourth_df.iloc[0]
        fourth_df.drop(fourth_df.index[0], inplace=True)
        fourth_df.reset_index(drop=True, inplace=True)
        break

    else:
        next

QL_Cap_df = first_df

WT_EntrT_perCar_df = second_df

Throughput = WT_EntrT_perCar_df['WaitSlm'].count()
               + WT_EntrT_perCar_df['WaitGzr'].count()
               + WT_EntrT_perCar_df['WaitMsd'].count()
               + WT_EntrT_perCar_df['WaitKap'].count()

Throughput_df = pd.DataFrame({'Throughput': [Throughput]})

Loc_carid_VQ_df = third_df
# Identify columns with NaN in their name
columns_to_drop = [col for col in Loc_carid_VQ_df if pd.isna(col)]
# Drop columns with NaN in their name
Loc_carid_VQ_df.drop(columns=columns_to_drop, inplace=True)
Loc_carid_VQ_df = Loc_carid_VQ_df.drop_duplicates()
Loc_carid_VQ_df.reset_index(drop=True, inplace=True)

Trafficlight_df = fourth_df
columns_to_drop = [col for col in Trafficlight_df if pd.isna(col)]
# Drop columns with NaN in their name
Trafficlight_df.drop(columns=columns_to_drop, inplace=True)
```

```
#### 1 Green ##### 0 Red ####### 2 Yellow ###
### Processing tables to create ###
#### Total Waiting Time per car ###
# Initialize newVQT as a list of lists
newVQT = [
    [],  # SlmId
    [],  # SlmWQT
    [],  # GzrId
    [],  # GzrWQT
    [],  # MsdId
    [],  # MsdWQT
    [],  # KapId
    []   # KapWQT
]

# Iterate over the DataFrame
for i in range(len(Loc_carid_VQ_df['Location'])):
    # Append CarID to SlmId
    if Loc_carid_VQ_df['Location'][i] == 'Sliema':
        newVQT[0].append(Loc_carid_VQ_df['CarID'][i])
        # Append Virtual Queue Time to SlmWQT
        newVQT[1].append(Loc_carid_VQ_df['Virtual Queue Time'][i])
    elif Loc_carid_VQ_df['Location'][i] == 'Gzira':
        # Append CarID to GzrId
        newVQT[2].append(Loc_carid_VQ_df['CarID'][i])
        # Append Virtual Queue Time to GzrWQT
        newVQT[3].append(Loc_carid_VQ_df['Virtual Queue Time'][i])
    elif Loc_carid_VQ_df['Location'][i] == 'Msida':
         # Append CarID to MsdId
        newVQT[4].append(Loc_carid_VQ_df['CarID'][i])
        # Append Virtual Queue Time to MsdWQT
        newVQT[5].append(Loc_carid_VQ_df['Virtual Queue Time'][i])
    else:
        # Append CarID to KapId
```

```
            newVQT[6].append(Loc_carid_VQ_df['CarID'][i])
            # Append Virtual Queue Time to KapWQT
            newVQT[7].append(Loc_carid_VQ_df['Virtual Queue Time'][i])
    print(newVQT)

    Otherdict = {
        'SlmId': newVQT[0],
        'SlmWQT': newVQT[1],
        'GzrId': newVQT[2],
        'GzrWQT': newVQT[3],
        'MsdId': newVQT[4],
        'MsdWQT': newVQT[5],
        'KapId': newVQT[6],
        'KapWQT': newVQT[7]
    }

    # Get the maximum length of the lists
    max_length = max(len(lst) for lst in newVQT)

    # Pad each list with NaN values to make them equal length
    padded_lists = [lst + [np.nan] * (max_length - len(lst))
                                            for lst in newVQT]

    # Create the DataFrame
    newVQT_df = pd.DataFrame({
        'SlmId': padded_lists[0],
        'SlmWQT': padded_lists[1],
        'GzrId': padded_lists[2],
        'GzrWQT': padded_lists[3],
        'MsdId': padded_lists[4],
        'MsdWQT': padded_lists[5],
        'KapId': padded_lists[6],
        'KapWQT': padded_lists[7]
    })
```

```
WText = WT_EntrT_perCar_df[['CarNo','WaitSlm','WaitGzr'
                                ,'WaitMsd','WaitKap']]

### ADDING WAITING TIME TABLE ###
ResultTtlWT = pd.concat([newVQT_df, WText], axis=1)

ResultTtlWT = ResultTtlWT.drop(columns=['SlmId','GzrId','MsdId','KapId'])

max_row_len = len(ResultTtlWT)

# Specify the starting value for filling NaN
# Adjust this as needed
start_value = ResultTtlWT['CarNo'].count() +1

# Generate a list of increasing natural numbers
fill_values = list(range(start_value, max_row_len+1))

nan_count = ResultTtlWT['CarNo'].isnull().sum()

# Fill NaN values in 'CarNo' column with fill_values
ResultTtlWT.loc[ResultTtlWT['CarNo'].isnull()
                    , 'CarNo'] = fill_values[:nan_count]

Tttls_df = pd.DataFrame(np.nan, index=range(max_row_len)
                            , columns=['TtlWTSlm', 'TtlWTGzr'
                                        , 'TtlWTMsd', 'TtlWTKap'])

ResultTtlWT = pd.concat([ResultTtlWT, Tttls_df], axis=1)

for j in range(max_row_len):
    ## Adding non Nan values
    ResultTtlWT['TtlWTSlm'][j] =  ResultTtlWT['SlmWQT'][j]
                                    +  ResultTtlWT['WaitSlm'][j]
```

```
    ResultTtlWT['TtlWTGzr'][j] =  ResultTtlWT['GzrWQT'][j]
                                    +  ResultTtlWT['WaitGzr'][j]
    ResultTtlWT['TtlWTMsd'][j] =  ResultTtlWT['MsdWQT'][j]
                                    +  ResultTtlWT['WaitMsd'][j]
    ResultTtlWT['TtlWTKap'][j] =  ResultTtlWT['KapWQT'][j]
                                    +  ResultTtlWT['WaitKap'][j]

### Total Waiting Times in ResultTttlWT table - FINAL TABLE ####
### STOPS ###
CalcStopsTable = WT_EntrT_perCar_df[['CarNo','EntrSlm','EntrGzr'
                        ,'EntrMsd','EntrKap']]


ExitTable = pd.DataFrame(np.nan, index=range(len(CalcStopsTable))
                            , columns=['ExitSlm', 'ExitGzr'
                                        , 'ExitMsd', 'ExitKap'
                                        ,'StopsSlm', 'StopsGzr'
                                        , 'StopsMsd', 'StopsKap'])

CalcStopsTable  = pd.concat([CalcStopsTable , ExitTable], axis=1)

for j in range(len(CalcStopsTable)):
    # Calculate Exit values
    CalcStopsTable.loc[j, 'ExitSlm'] =
            WT_EntrT_perCar_df.loc[j, 'EntrSlm']
            + WT_EntrT_perCar_df.loc[j, 'WaitSlm']
    CalcStopsTable.loc[j, 'ExitGzr'] =
            WT_EntrT_perCar_df.loc[j, 'EntrGzr']
            + WT_EntrT_perCar_df.loc[j, 'WaitGzr']
    CalcStopsTable.loc[j, 'ExitMsd'] =
            WT_EntrT_perCar_df.loc[j, 'EntrMsd']
            + WT_EntrT_perCar_df.loc[j, 'WaitMsd']
    CalcStopsTable.loc[j, 'ExitKap'] =
            WT_EntrT_perCar_df.loc[j, 'EntrKap']
```

```
            + WT_EntrT_perCar_df.loc[j, 'WaitKap']

    # Handle NaN values in Wait columns
    if pd.isna(WT_EntrT_perCar_df.loc[j, 'WaitSlm'])
        and not pd.isna(WT_EntrT_perCar_df.loc[j, 'EntrSlm']):
        CalcStopsTable.loc[j, 'ExitSlm'] = 3900
    if pd.isna(WT_EntrT_perCar_df.loc[j, 'WaitGzr'])
        and not pd.isna(WT_EntrT_perCar_df.loc[j, 'EntrGzr']):
        CalcStopsTable.loc[j, 'ExitGzr'] = 3900
    if pd.isna(WT_EntrT_perCar_df.loc[j, 'WaitMsd'])
        and not pd.isna(WT_EntrT_perCar_df.loc[j, 'EntrMsd']):
        CalcStopsTable.loc[j, 'ExitMsd'] = 3900
    if pd.isna(WT_EntrT_perCar_df.loc[j, 'WaitKap'])
        and not pd.isna(WT_EntrT_perCar_df.loc[j, 'EntrKap']):
        CalcStopsTable.loc[j, 'ExitKap'] = 3900

for j in range(len(CalcStopsTable)):
    # Filter Trafficlight_df based on simulation times for each column
    filtered_dfSlm =
            Trafficlight_df[(Trafficlight_df['Simtime']
                             >= CalcStopsTable.loc[j, 'EntrSlm'])
            & (Trafficlight_df['Simtime']
                             <= CalcStopsTable.loc[j, 'ExitSlm'])]
    filtered_dfSlm.reset_index(drop=True, inplace=True)

    filtered_dfGzr =
            Trafficlight_df[(Trafficlight_df['Simtime']
                             >= CalcStopsTable.loc[j, 'EntrGzr'])
            & (Trafficlight_df['Simtime']
                             <= CalcStopsTable.loc[j, 'ExitGzr'])]
    filtered_dfGzr.reset_index(drop=True, inplace=True)

    filtered_dfMsd =
            Trafficlight_df[(Trafficlight_df['Simtime']
```

```
                                >= CalcStopsTable.loc[j, 'EntrMsd'])
            & (Trafficlight_df['Simtime']
                                <= CalcStopsTable.loc[j, 'ExitMsd'])]
    filtered_dfMsd.reset_index(drop=True, inplace=True)

    filtered_dfKap =
            Trafficlight_df[(Trafficlight_df['Simtime']
                                >= CalcStopsTable.loc[j, 'EntrKap'])
            & (Trafficlight_df['Simtime']
                                <= CalcStopsTable.loc[j, 'ExitKap'])]
    filtered_dfKap.reset_index(drop=True, inplace=True)

    # Calculate StopsSlm if EntrSlm and ExitSlm are not NaN
    if not pd.isna(CalcStopsTable.loc[j, 'EntrSlm'])
            and not pd.isna(CalcStopsTable.loc[j, 'ExitSlm']):
        CalcStopsTable.loc[j, 'StopsSlm'] =
            ((filtered_dfSlm['Slm'].shift(1) == 2)
                & (filtered_dfSlm['Slm'] == 0)).sum()

    # Calculate StopsGzr if EntrGzr and ExitGzr are not NaN
    if not pd.isna(CalcStopsTable.loc[j, 'EntrGzr'])
            and not pd.isna(CalcStopsTable.loc[j, 'ExitGzr']):
        CalcStopsTable.loc[j, 'StopsGzr'] =
            ((filtered_dfGzr['Gzr'].shift(1) == 2)
                & (filtered_dfGzr['Gzr'] == 0)).sum()

    # Calculate StopsMsd if EntrMsd and ExitMsd are not NaN
    if not pd.isna(CalcStopsTable.loc[j, 'EntrMsd'])
            and not pd.isna(CalcStopsTable.loc[j, 'ExitMsd']):
        CalcStopsTable.loc[j, 'StopsMsd'] =
            ((filtered_dfMsd['Msd'].shift(1) == 2)
                & (filtered_dfMsd['Msd'] == 0)).sum()

    # Calculate StopsKap if EntrKap and ExitKap are not NaN
```

```
    if not pd.isna(CalcStopsTable.loc[j, 'EntrKap'])
            and not pd.isna(CalcStopsTable.loc[j, 'ExitKap']):
        CalcStopsTable.loc[j, 'StopsKap'] =
            ((filtered_dfKap['Kap'].shift(1) == 2)
                & (filtered_dfKap['Kap'] == 0)).sum()

### CalcStopsTable FINAL TABLE ###

# Define the data
SummaryData = {
"Location": ["Sliema", "Gzira", "Msida", "Kappara"],
"Av. Waiting Time": [ResultTtlWT['TtlWTSlm'].mean()
                        , ResultTtlWT['TtlWTGzr'].mean()
                        , ResultTtlWT['TtlWTMsd'].mean()
                        , ResultTtlWT['TtlWTKap'].mean()],
"Av. Queue Length": [QL_Cap_df['QueueLenSlm'].mean()
                        , QL_Cap_df['QueueLenGzr'].mean()
                        , QL_Cap_df['QueueLenMsd'].mean()
                        , QL_Cap_df['QueueLenKap'].mean()],
"Av. Number of Stops": [CalcStopsTable['StopsSlm'].mean()
                        , CalcStopsTable['StopsGzr'].mean()
                        , CalcStopsTable['StopsMsd'].mean()
                        , CalcStopsTable['StopsKap'].mean()]
}

# Create the DataFrame
Summary_df = pd.DataFrame(SummaryData)

base_path = r'C:\Users\jform\Desktop\MscResults_Latest\Evening_MDP'
#file_name =  "'" + file + "'"
extension = '.xlsx'
# Create a Pandas Excel writer using openpyxl as the engine
output_path = os.path.join(base_path, file + extension)
#output_path = r'C:\Users\jform\Desktop\MscResults\output.xlsx'
```

```
    with pd.ExcelWriter(output_path, engine='openpyxl') as writer:
        # Write df1 to the first part of the sheet
        QL_Cap_df.to_excel(writer, sheet_name='Sheet1', startrow=0
                           , startcol=0, index=False)

        # Write df2 to the same sheet, below df1
        Throughput_df.to_excel(writer, sheet_name='Sheet1', startrow=0
                               , startcol= QL_Cap_df.shape[1] + 2
                               , index=False)

        Summary_df.to_excel(writer, sheet_name='Sheet1', startrow=0
                           , startcol= QL_Cap_df.shape[1]
                                       + Throughput_df.shape[1] + 4
                           , index=False)

        # Write df3 to the same sheet, below df2
        ResultTtlWT.to_excel(writer, sheet_name='Sheet1', startrow= 0
                           , startcol= QL_Cap_df.shape[1]
                                       + Summary_df.shape[1]
                                       + Throughput_df.shape[1]+ 6
                           , index=False)

        # Write df4 to the same sheet, below df3
        CalcStopsTable.to_excel(writer, sheet_name='Sheet1', startrow=0
                           , startcol= QL_Cap_df.shape[1]
                                       + Summary_df.shape[1]
                                       + Throughput_df.shape[1]
                                       +ResultTtlWT.shape[1]+ 8
                           , index=False)

### Loop throughout all csv files ###

directory = r'C:\Users\jform\Desktop\RawAimsunResultsNew\Evening_MDP'
```

```
# Loop through each file in the directory
for filename in os.listdir(directory):
    df = pd.read_csv(os.path.join(directory, filename))
    file = os.path.splitext(filename)[0]
    TransResults(df, file)

#%% Output Grand statistics File

# Directory containing the Excel files
directory = r'C:\Users\jform\Desktop\MscResults_Latest\Evening_MDP'

# Initialize an empty list to store DataFrames
dfs = []

# Loop through each file in the directory
for filename in os.listdir(directory):
    if filename.endswith('.xlsx'):
        file_path = os.path.join(directory, filename)
        # Read the Excel file
        df = pd.read_excel(file_path)
        # Append the DataFrame to the list
        dfs.append(df)

# Concatenate all DataFrames
all_data = pd.concat(dfs, ignore_index=True)

# Compute the aggregated statistics
stats = all_data.groupby('Location').agg({
    'Av. Waiting Time': ['mean', 'sum', 'min', 'max', 'std'],
    'Av. Queue Length': ['mean', 'sum', 'min', 'max', 'std'],
    'Av. Number of Stops': ['mean', 'sum', 'min', 'max', 'std']
}).reset_index()

# Flatten the MultiIndex columns
```

```
stats.columns = ['_'.join(col).strip() for col in stats.columns.values]
stats.rename(columns={'Location_': 'Location'}, inplace=True)

average_throughput = all_data['Throughput'].mean()

# Save the results to a new Excel file
output_path = r'C:\Users\jform\Desktop\MscResults_Latest
                    \Evening_MDP\GrandStatsThro_Evening_MDP.xlsx'
with pd.ExcelWriter(output_path, engine='openpyxl') as writer:
    stats.to_excel(writer, sheet_name='Statistics', index=False)

    throughput_df = pd.DataFrame({'Average Throughput':
                                    [average_throughput]})
    # Replace with the desired row number
    start_row = 0
    # Replace with the desired column number
    #(0 for column A, 1 for column B, etc.)
    start_col = 17

    # Write the throughput DataFrame to the desired location in the sheet
    throughput_df.to_excel(writer, sheet_name='Statistics', index=False
                           , startrow=start_row, startcol=start_col)

print("Aggregated statistics have been saved to", output_path)
```