The Use of Computational Intelligence for Traffic Light Control

Amr Muhammad Alameen Khalifa
135058

Supervisor: Dr. Ahmed Kheiri

Faculty of Engineering
University of Khartoum

A thesis submitted in partial fulfillment of the requirements for the degree of
B.Sc (Hons) Electrical and Electronic Engineering
(Software Engineering)

September 2018

I dedicate this work to my family because of their unconditional love and continuous support. I also dedicate this work to the soul of my grandfather who had passed away few days before the submission. He had always been encouraging me to keep moving forward.

Acknowledgements

I would like to express my sincere acknowledgement for my supervisor Dr.Ahmed Kheiri, for his continuous support and revision. We have learned a lot from him. Dr. Ahmed has been a true inspiration, extremely helpful, and considerate.
I would also like to thank my partner, colleague and dear friend Eltayeb. He has shown a great deal of hard working, team collaboration, passion, commitment, dedication, and cleverness. I have enjoyed working with him I'm looking forward to continue working with him in the future.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Amr Muhammad Alameen Khalifa

135058
September 2018

Abstract

The intelligent control of traffic lights is a research topic of high interest due to its ability to help road traffic networks cope with the ever-growing increase in vehicle traffic. A well designed traffic light controller can provide a significant increase in network throughput. There are two main types of traffic light controllers, pre-timed controllers which operate on predetermined timing cycles and traffic responsive controllers that change phase in reaction to real-time input from sensors describing the state of the road. Since road traffic is an extremely complex non-linear stochastic system the use of traditional analytical modelling techniques to design either type of controller is extremely difficult and yields modestly performing results. However, the existence of real-life like simulators makes the problem of traffic light controller design a very good candidate for computational intelligence algorithms. When designing an intelligent traffic light controller it follows that the controller will give higher priority to the phase with higher traffic, however if you optimise over-aggressively the controller might make cars from the other phases wait disproportionately long. To measure this phenomenon we devised two metrics for traffic fairness, the VSSD and JTSD. In this work we tested multiple computational intelligence algorithms for both pre-timed and traffic responsive controllers. For optimising pre-timed controllers we applied genetic algorithms, and two selection-move acceptance hyper-heuristics SR-IE and SR-SA. We did a head-on-head comparison between the three algorithms and found the genetic algorithm to perform best, closely followed by SR-SA with SR-IE lagging behind in the quality of its yielded solutions. To design traffic responsive controllers we used the RL techniques Q-learning and policy gradient. We experimented with two reward functions for both algorithms, mean vehicle speed and a novel function FPMVS which combines mean vehicle speed with a fairness penalty term. Out of the four RL controllers the best performance in all criteria, both fairness and throughput, was achieved by the policy gradient controller trained with mean vehicle speed reward. It achieved a 21.7% decrease in mean journey time compared to the classical traffic responsive controller LQF. Three of the four RL controllers outperformed the LQF algorithm.

المستخلص

لقد أصبحت متحكمات إشارات المرور الذكية من المواضيع البحثية الرائجة نسبة لإرتفاع كثافة حركة مرور السيارات. إن بإمكان متحكم إشارة مرور مصمم بصورة جيدة زيادة سعة شبكة مرور سيارات بصورة ملحوظة. يوجد نوعين أساسيين من من متحكمات إشارات المرور، المتحكمات ذات الزمن الثابت والتي تتغير بعد دورات زمنية ثابتة محددة مسبقاً، والمتحكمات التفاعلية مع المرور والتي تتغير طبقاً لمدخلات لحظية عن حالة الطريق آتية من متحسسات. بما أن سلوك حركة المرور نظام معقد غير خطي ويمتاز بدرجة من العشوائية فإن استخدام طرق النمذجة التحليلية التقليدية في تصميم أي نوع من متحكمات إشارات المرور على درجة عالية من الصعوبة، وديدنها الفشل في احراز نتائج جيدة. إن ظهور المحاكيات الواقعية لحركة المرور قد جعلت اسخدام طرق الذكاء المحوسب خياراً ملائماً لتطوير متحكمات إشارات المرور. عند تصميم إشارات المرور الذكية فإنه من الحتمي أن يعطي الإشارة الأولوية للاتجاه ذو الكثافة المرورية الأعلى، ولكن عند التركيز المفرط على تحسين سعة التقاطع، فإن الإشارة قد تجعل الذين هم آتون من الاتجاه ذو الكثافة المرورية الأقل ينتظرون فترات طويلة بصورة غير عادلة، لوصف هذه الظاهرة فإننا اقترحنا مقياسين جديدين، الانحراف المعياري لسرعة السيارات و الإنحراف المعياري لزمن رحلات السيارات. لقد قمنا في عملنا هذا بإختبار عدد من خوارزميات الذكاء المحوسب لتصميم كل من المتحكمات ذات الزمن الثابت، والمتحكمات التفاعلية. لتحسين المتحكمات ذات الزمن الثابت استخدمنا الخوارزمية الجينية، بالإضافة إلى اثنين من خوارزميات استدلال الاستدلالات، خورازمية الإختيار العشوائي مع القبول المساوي أو التحسني، وخوازمية الإختيار العشوائي مع القبول المحاكي للتصلب. كان الأداء الأفضل من بين الخوارزميات الثلاثة للخوارزمية الجينية، وتليها عن قرب خوارزمية الإختيار العشوائي مع القبول المحاكي للتصلب، وتأخرت خوارزمية الإختيار العشوائي مع القبول المساوي أو التحسني من حيث جودة النتائج. لتصميم المتحكمات التفاعلية استخدمنا كل من طريقة تعلم دالة الجودة وطريقة منحدر السياسة الآتيتين من مجال التعلم التعزيزي. قمنا بإختبار كل من الطريقتين مع دالتي جزاء مختلفتين، دالة السرعة اللحظية المتوسطة، ودالة السرعة اللحظية مع العقوبات العادلة. كان الأداء الأفضل في كل المقاييس (السعة والعدالة) لطريقة منحدر السياسة عند استخدام دالة جزاء السرعة المتوسطة والتي حققت نقصان 21.7% في متوسط زمن الرحلة مقارنة بطريقة التحكم التفاعلية الصف الأطول أولا. تفوقت ثلاث متحكمات من أصل الأربعة متحكمات التفاعلية المقترحة على خوارزمية الصف الأطول أولا.

Table of contents

Dedication **ii**

Acknowlegdements **iii**

Abstract **v**

Abstract (Arabic) **vi**

List of figures **ix**

List of tables **x**

Nomenclature **xi**

1 Introduction **1**

1.1 Background 1

1.2 Research Objectives 2

1.3 Academic Publications 2

1.4 Thesis Structure 2

2 Literature Review **4**

2.1 The Traffic Signal Control Problem 4

2.2 Genetic Algorithms 6

2.2.1 The Fundamental Theory of Genetic Algorithms 8

2.2.2 The Use of Genetic Algorithms in Traffic Signalling Optimisation . 9

2.3 Hyper-Heuristics 10

2.3.1 The Use of Hyper-Heuristics in Traffic Signalling Optimisation . . 14

2.4 Reinforcement Learning 15

2.4.1 Reinforcement Learning Theory 15

2.4.2 Reinforcement Learning as Class of Machine Learning 15

2.4.3 The Standard RL Model 16
2.4.4 Modelling Optimal Behaviour 17
2.4.5 Measuring Learning Performance 18
2.4.6 Markov Decision Processes (MDPs) 18
2.4.7 Solving Markov Decision Processes 19
2.4.8 Finding the Optimal Policy 20
2.4.9 Learning to Solve MDPs 22
2.4.10 Model-free Reinforcement Learning Strategies 23
2.4.11 Policy Search 25
2.4.12 The State of the Art Reinforcement Learning 27
2.4.13 The Use of RL in Traffic Optimisation 28

3 Methodology **30**
3.1 Problem Definition 30
3.1.1 Traffic Description Metrics 30
3.1.2 The Scenario 32
3.2 Pre-timed Traffic Light Optimisation 32
3.2.1 The Selection - Move Acceptance Hyper-Heuristics 33
3.2.2 The Genetic Algorithm 35
3.3 Traffic Responsive Control 36
3.3.1 Q-Learning 38
3.3.2 Policy Gradient 40

4 Results and Discussion **43**
4.1 Overview 43
4.2 Pre-timed Traffic Control 43
4.2.1 The performance of the Low Level Heuristics 45
4.3 Traffic adaptive control 46

5 Conclusion **53**
5.1 Summary of Work 53
5.2 Future Work 54

Appendix A Code Listings **56**

Appendix B The Scenario **97**

References **108**

List of figures

2.1 The agent-environment abstraction in a reinforcement learning model taken from [1] 16

3.1 A screenshot of the simulation of the problem scenario. 32
3.2 This figure shows (a) a sample of traffic and the corresponding DTSE (b) boolean and (c) real vectors taken from [2] 37
3.3 An illustration of the neural network used to approximate the Q-function. . 39
3.4 An illustration of the neural network used to approximate policy. 41

4.1 The change in the throughput metrics during optimisation for the GA, SR-IE and SR-SA, averaged over the 10 runs 45
4.2 Percentage of improving moves causes by each low-level heuristic. 46
4.3 Changes in mean vehicle speed during training the RL models without filtering 47
4.4 Changes in mean vehicle speed during training the RL models. 48
4.5 Changes in mean journey time during training RL models. 49
4.6 Changes in mean time loss during training the RL models. 50
4.7 Changes in mean waiting time during training the RL models. 51
4.8 Changes in reward for while training the RL models with $\beta ! = 0$ 52

List of tables

3.1 Parameters of the genetic algorithm 35
3.2 The hyper-parameters of the Q-learning's Q-function approximation neural network. 39
3.3 The hyper-parameters of the policy gradient's policy approximation neural network. 41

4.1 Summary of intersection throughput for GA, SR-IE and SR-SA. Best values are highlighted in bold. 44
4.2 Pairwise performance comparison of SR-IE, SR-SA and GA. 44
4.3 VSSD (Vehicle Speed Standard deviation) and JTSD (Journey Time Standard Deviation) for pre-timed traffic light optimization. Best values are highlighted in bold. 45
4.4 Throughput metrics for traffic responsive control. Best values are highlighted in bold. 47
4.5 VSSD (Vehicle Speed Standard deviation) and JTSD (Journey Time Standard Deviation) for traffic responsive controllers. Best values are highlighted in bold. 47

Nomenclature

Roman Symbols

$\mathbb{R}_+$	The set of all positive real numbers
d	Vehicle waiting time
j	Vehicle journey time
T	Temperature
v	Vehicle speed
$\mathbf{A}$	Set of all possible actions
$\mathbf{r}$	Set of rewards
$\mathbf{S}$	Set of all possible states
E	Expectation
P	Probability
V	Value function

Greek Symbols

α	Learning rate
β	A parameter in a novel reward function that controls the weight of the fairness term
Δ	The change in
ε	The fraction of time a random exploratory move is selected
γ	Discount rate

∇	The gradient of
π	A policy (a mapping from states to actions)

Acronyms / Abbreviations

AM	All moves
CI	Computational intelligence
DRL	Deep reinforcement learning
DTSE	Discrete traffic state encoding
EMC	Exponential Monte Carlo
EMCQ	Exponential Monte Carlo with counter
FPMVS	Fairness penalised mean vehicle speed
GA	Genetic algorithm
JTSD	Journey time standard deviation
LLH	Low-level heuristic
LMC	Linear Monte Carlo
LQF	Longest queue first
MDP	Markov decision process
OI	Only Improvements
POMDP	Partially observable Markov decision process
PtMS	Parallel transportation management system
Q-function	Quality function
RHODES	Real-time hierarchical optimized distributed effective system
RL	Reinforcement learning
SCATS	Sydney coordinated adaptive traffc system
SCOOT	Split cycle offset optimization technique

SGA	Simple genetic algorithm
SR-IE	Simple Random - Improve or Equals
SR-SA	Simple Random - Simulated Annealing
SUMO	Simulation of Urban Mobility
TD	Temporal differences
TRANSYT	Traffic network study tool
UTOPIA	Urban traffc optimization by integrated automation
VSSD	Vehicle speed standard deviation

Chapter 1

Introduction

1.1 Background

Today about 55% of the world's population live in cities, with that number expected to rise over next few decades. This increase in population density is accompanied with an increase in the number of vehicles on the road. Adapting the existing road traffic infrastructure to accommodate this increase is in most cases prohibitively expensive, so the search for ways to maximize the use of currently existing infrastructure is underway. One of the main areas of interest is optimising the control of traffic lights. There are two main types of traffic light controllers, pre-timed controllers that operate based on timers with fixed time cycles and traffic adaptive controllers that operate based on real-time input from sensors that feed the controller with instantaneous information about the current state of the road. Since traffic is complex, non-linear and stochastic it is difficult to design controllers (pre-timed or adaptive) by deriving analytical models of traffic and using them to optimise the flow of traffic, however since realistic traffic simulators exist, this makes methods that can learn from experience very well suited for the problem, such algorithms are known as computational intelligence algorithms. Previous literature covers the use of many different algorithms for both problems, however there is a definite lack of head-on-head comparative studies where multiple algorithms are tested on the same scenarios. Furthermore, implementations of many of these computational intelligence techniques are available from specialized software vendors but the prices are exorbitantly expensive with licenses for software packages for traffic light controller systems prices ranging from the tens of thousands of dollars to multiple millions of dollars. In this work we attempted to implement state of the art computational intelligence techniques and tailor them to the design of pre-timed and traffic responsive traffic light controllers and ran head-on-head comparisons between them. For the pre-timed controllers we focused on the meta-heuristic genetic algorithm and selection-move acceptance hyper-heuristic algorithms. For traffic responsive control we focused on reinforcement learning techniques

1.2 Research Objectives

The objectives of this work are to:

1. Conduct a thorough literature review on the state of the art traffic light control methods in general, and the techniques used in this work specifically, namely the genetic algorithm, hyper-heuristics and reinforcement learning
2. Optimise pre-timed traffic light cycles using the genetic algorithm and hyper-heuristic algorithms (SR-IE & SR-SA) and conduct comparisons between the different timing plans using a real-world like microscopic simulator.
3. To create reinforcement learning based (Q-learning and policy gradient) agents that perform traffic responsive control and compare there performance with traditional traffic adaptive traffic light control methods.

1.3 Academic Publications

- **ICCCEEE Conference Paper:** Eltayeb Ahmed, Amr Khalifa, Ahmed Kheiri *Evolutionary Computation for Static Traffic Light Cycle Optimisation* International Conference on Computer, Control, Electrical and Electronics Engineering: Artificial Intelligence Track, ICCCEEE, Khartoum, Sudan
- **Deep Learning Indaba 2018 Poster:** Amr Khalifa, Eltayeb Ahmed, Ahmed Kheiri *Generating Optimized Traffic Light Controllers using Reinforcement Learning* Deep Learning Indaba 2018, Stellenbosch, South Africa.

1.4 Thesis Structure

This thesis is made up of five chapters. Chapter 2 contains a literature review about genetic algorithms, selection-move acceptance hyper-heuristics, reinforcement learning and the traffic light optimisation problem. Chapter 3 describes the problem tackled in this research in detail as well as all the representations, models and assumptions used in the application of the different algorithms to the problem. Chapter 4 shows the results of the experiments that were run on the algorithms outlined in Chapter 3. Chapter 5 contains a brief summary of the work along with the key findings and suggestions for future work.

Chapter 2

Literature Review

2.1 The Traffic Signal Control Problem

Road traffic networks are large nonlinear systems with complex interactions of stochastic nature taking place between its various components. Traffic networks have many various parameters that can be tuned in order to reduce congestion, this has been the subject of many different studies because reductions in traffic congestion will save time and fuel and reduce environmental pollution. There are many causes of congestion including the rising number of vehicles on the road that are outgrowing the capacity of the infrastructure, inadequate planning and lack of large scale solutions. *Traffic signal control* is one of the most important parameters that can be configured in a road traffic network because it has a very profound effect on overall congestion and it can be re-tuned without incurring large costs due to change to infrastructure. A study in Phoenix, Arizona found the implementation of an intelligent traffic signal control system would cut down on vehicle collisions by 6.7% and vehicle travel time by 11.4% [3]. *Traffic signal control* has gone through various phases, all of which are still in use to various degrees. They are *pre-timed control*, *traffic responsive control* and *intelligent control*. *Pre-timed control* uses a predetermined cycle time and green portion. It is the oldest technique, with methods for tuning the cycle time and green portion dating back to the late 50's [4, 5]. The oldest techniques for tuning the cycles were based upon macro-models for traffic most famously including Webster's model which is still widely used and cited in literature today [4]. Multiple fixed-time traffic signal configurations can be prepared for different times of day, however since traffic is very dynamic any system based on fixed-time plans is inherently limited. *Traffic responsive control* is based upon the use of real-time feedback from sensors to make decisions based upon the current state of the road. *Actuated control* is one of the most popular *Traffic responsive methods*. It selects the next traffic light *phase* (colour) or decides to extend the time of the current phase based upon the real-time traffic and a set of predetermined rules. This method takes into account only vehicles in the currently moving lanes so it cannot achieve optimality in its use of resources and is only appropriate for conditions with low or medium traffic saturation and high traffic randomness. There are many examples of *traffic responsive control* systems currently in use around the world such as the *Sydney coordinated adaptive traffic system (SCATS)* in Australia that designates plans for individual traffic lights to maximise the traffic flow through the network. Another system in use is the *Split cycle offset optimisation technique (SCOOT)* system in the UK that incrementally reactively adapts the traffic light's timing in response to real-time signals. Italy uses a system called *urban traffic optimisation by integrated automation (UTOPIA)/system for priority and optimisation of traffic (SPOT)* that is a distributed system with special consideration for complex public transport systems. The US also uses a distributed system, the system *RHODES* has proved its efficiency in field tests in Arizona. And last but not least China uses a newly developed system called the *parallel transportation management system (PtMS)* [3].

Traffic system's behaviour are influenced by many factors such as the types of vehicles, the infrastructure, the weather and the behaviour of users of the road. This makes them large complex nonlinear stochastic systems that analytical methods or classic control theories fail to capture effectively. This has risen to the use of multiple techniques that fall under the umbrella of *computational intelligence (CI)*. Computational intelligence techniques shine well in such conditions where precise mathematical models are difficult and impractical to formulate. The CI methods used in the field of traffic signal control are widely varied but include *fuzzy systems*, *artificial neural networks*, *evolutionary computation and swarm intelligence*, *reinforcement learning* and *agent and game techniques*.

Fuzzy Systems are systems based upon mathematical models that allow uncertainty. In contrast to *crisp logic* where statements can only be either true or false *fuzzy logic* allows statements to be true or false to a certain degree. Fuzzy systems are governed by the laws of *fuzzy sets* that allow elements to have a "degree" of membership between 0 and 1 as opposed to absolute membership or lack of thereof as in traditional *crisp sets* [6]. In comparison to other computational intelligence methods fuzzy logic has the advantage that it is easy to incorporate expert human judgement into its systems. The use of fuzzy logic in traffic light control dates back to the 70's and is the first form of computational intelligence to be implemented in a traffic signal controller. This can be traced back to the work of Pappis et al. where they designed a traffic signal controller for a single intersection [7]. Lee et al. developed on this method and designed a fuzzy logic based controller for traffic lights that shared information with neighbouring controllers and achieved good results [8]. Chou and Teng designed a fuzzy traffic light controller that scales to multiple lanes or multiple intersections without modification, the controller they designed had only 9 fuzzy logic rules and achieved good performance in life-like simulation scenarios [9]. Qiao proposed a two-stage traffic controller based upon fuzzy logic [10]. The first stage was to select the next lane to assign a green signal. The second stage decides the duration of the green signal. The controller aimed to optimise both the efficiency and the fairness of the intersection. The controller's fuzzy logic rules and membership functions were tuned using a genetic algorithm offline. The performance of the controller was evaluated both from the perspectives of efficiency and fairness and was found to outperform Pappis et al. [7] in both metrics. Gokulan and Srinivasan designed a multi-agent geometric fuzzy based system for controlling traffic signals [11] and achieved good results. The system developed outperformed multiple other techniques currently in use around the world. The system performed very well at handling planned and unplanned road obstructions and other exceptional incidents.

The *agent and game* paradigm has also been applied to the traffic signal control problem to design agent-based controllers. Some approaches are based on *reinforcement learning* and are covered in great detail in Section 2.4.13. Others approaches are based on the application traditional *game theory*. Examples of such approaches include Alvarez et al. where they modelled each intersection as a Markov chain [12]. The target of optimising the flow of the traffic was then viewed as a non-cooperative multi-player game. They placed a player at each intersection. Each player tried to minimise the length of the queue at their intersection. They then concluded that the best overall traffic flow would be achieved when all agents reached a state of Nash equilibrium. Cheng et al. followed an approach where they followed the game-theoretical paradigm of fictitious play to search for a timing plan for a a set of intersections [13]. The algorithm they created was highly parallel and could make use of large number of CPU's. They tested the algorithm on a large real world scenario containing 75 signalised intersections. The proposed algorithm outperformed hill climbing algorithms when both were given the same amount of computational wall clock time.

The use of genetic algorithms in traffic signal controllers is widely spread and is covered in detail in Section 2.2.2. The use of swarm intelligence and other hyper-heuristics has also seen a fair amount of attention in the research and is covered in Section 2.3.1.

2.2 Genetic Algorithms

Genetic algorithms were first proposed by Holland in the 1970's as method for solving various optimisation problems inspired by biological evolution [14]. Genetic Algorithms are part of a larger family of algorithms known as evolutionary algorithms. What differentiates genetic algorithms from other evolutionary algorithms is the use of crossover in the search process. In genetic algorithms each possible solution in the search space is represented by a chromosome, with each individual characteristic of the solution represented by a gene in the chromosome. The genetic algorithm first starts with a random *population* (group) of solutions known as a gene pool. On each iteration of the algorithm a new gene pool is generated by the application of crossover and mutation to the members of the existing gene pool. Genetic algorithms that replace the entire population on each iteration are known as *generational genetic algorithms* while genetic algorithms that replace only a portion of population are known as *steady-state genetic algorithms*. The objective function that is the target of the optimisation is usually re-scaled to fit in the range 0 to 1 and is called the *fitness function*. The fitness function is used to evaluate the performance of each individual solution. The first step of each iteration of the genetic algorithm is *selection*. Selection is modelled after nature's survival-of-the-fittest. Each member of the solution with fitness f_i is allocated $f_i/\bar{f}$ offspring where $\bar{f}$ is the average fitness. In this scheme the solutions with better values of the objective function are given higher representation in the next generation of solutions and are more likely to pass down their traits. After selection comes *crossover*. Pairs of members are selected to undergo crossover. Two new individual solutions (known as offspring) are produced by exchanging genes between the originally selected individuals. The newly generated individuals are then subject to *mutation* where randomly selected genes are subjected to random perturbation. This process is then repeated until the termination criteria is reached. This criteria could be a number of generations or the reaching of an individual with a certain target fitness or the reaching of a certain degree of homogeneity [15].

The Simple Genetic Algorithm

The simple genetic algorithm (SGA) is the algorithm originally proposed by Holland in the 1970's [15]. Holland's genetic algorithm represents each chromosome by a string with binary characters. Continuous variables can be mapped to integers with are then represented with binary strings. The SGA uses *roulette wheel selection* where each population member is allocated an angle equal to $2\pi f_i/\bar{f}$ of a roulette wheel which is then spun to select a member of the population. This method may result in large sampling errors where the actual number of the allocated offspring allocated differs significantly from the expected number. The SGA uses one-point crossover. A random point in the string representing each member is selected and the parts of the strings after the selected point are swapped. For mutation the SGA randomly selects a bit from the string and flips it. The SGA depends mainly on crossover to search the problem space and uses mutation mainly to recover lost genetic material.

Since the proposition of the SGA different methods for selection, crossover and mutation have been proposed. Selection methods such as *rank-based selection, elitist strategies* and *tournament selection* have all been presented. Crossover techniques such as *two-point crossover, multi-point crossover* and *uniform* crossover have all been considered as replacements for single-point crossover.

2.2.1 The Fundamental Theory of Genetic Algorithms

The exact mechanism of the effectiveness of GA's is still not known, however *schema theory* and *the building block* hypothesis give a general idea of how they work. A schema (plural schemata) is a template that fits a group of strings, for instance in the domain of chromosomes with 5 bit binary representations, the schema **110 fits all strings with the bits 110 as the final 3 bits, namely the strings 00110, 01110, 10110, 11110. Each string that fits the schema is called an instance of the schema. Since a schema is associated with a group of solutions we can calculate a mean fitness value for a schema. It is thought that good solutions are the combination of multiple independent schemata each individual schema with a high average fitness. A schema with a high average fitness is called a building block, and the hypothesis that a good performing solution can be reached by combining multiple building blocks is known as *the building block hypothesis*. Disregarding crossover, a schema with high average fitness will dominate the population exponentially, but crossover is a disturbing process. Schema with shorter defining lengths are less likely to be destroyed by crossover and have a higher chance of dominating the competition [15]. However the schema theorem and building block hypothesis have both faced some degree of scepticism. The schema theorem is based on the relationship between a schema, its average fitness and its probability of destruction. This relationship is stochastic in nature so generalising conclusions from it to beyond a single generation is problematic. As for building block theory, Holland himself created functions that were designed to exhibit the building block hypothesis and faced problematic results when non-genetic methods outperformed genetic algorithms in optimising them [16].

In the early generations of the genetic algorithm most individuals have low fitness with a few individuals having relatively high fitness, These individuals are allocated a large number of offspring and tend to dominate the population quickly causing premature convergence of the the genetic algorithm. In the late generations a different problem arises, when most individuals tend to have close values of fitness. This causes most individuals to be allocated an almost equal number of offspring. When this happens the survival-of-the-fittest mechanism (which is what pushes the members of the population towards better solutions) tends to saturate, slowing down or halting any further improvement.

Another problem that faces genetic algorithms is the problem of optimising deceptive functions. The notion that the combination of multiple good independent schemata will lead to a new solution better than any of the previous solutions (building block theory) is not true for all functions. Such functions are said to be deceptive functions and tend to be hard to optimise by genetic algorithms, however literature shows that deception is neither necessary nor sufficient for a function to be hard to optimise using genetic algorithms.

Another obstacle facing genetic algorithms is the existence of *Hamming cliffs*. A Hamming cliff is the existence of a large distance in representation between two logically close solutions. An example would be the large distance in representation between the number 15 and 16 when using binary representation (01111 and 10000 respectively). The use of *gray code* based representation resolves the problem partially but does not eliminate it.

Different crossover operators exhibit different kinds of bias. If an operator has a bias towards exchanging genes at certain positions then it is said to exhibit positional bias, on the other hand if it has bias towards exchanging a certain number of genes then it is said to exhibit distributional bias. Different operators fall in different positions on this spectrum, on one end of the spectrum we have one-point crossover which exhibits high positional bias and low distributional bias, and on the other end we have uniform crossover which exhibits low positional bias and high distributional bias [15].

2.2.2 The Use of Genetic Algorithms in Traffic Signalling Optimisation

Multiple pieces of literature research the use of genetic algorithms in pre-timed traffic light optimisation [17, 18]. The genetic algorithm is now also used by commercially available tools such as TRANSYT-T7F [19]. Kalganova et al. used to the genetic algorithm to optimise the timings of a multi-intersection traffic light system [17]. The study aimed to optimise both the cycle time and the portion of green time. Total vehicle delay in the traffic network was used as the objective function. The timings as well the relative offset of each traffic light were represented as a single chromosome. A uniform mutation where a random gene would be assigned a randomly generated integer value from a specific range was used. The use of roulette wheel selection and binary tournament selection was investigated and multiple methods of crossover were compared, they were uniform, one-point and two-point. The best result was obtained using uniform crossover however the behaviour of the mean fitness was better when using two-point crossover. The fastest convergence of the genetic algorithm was when one-point crossover was used. Tournament selection yielded better results than roulette wheel selection. An adaptive mutation rate where the mutation rate was increased when the population fitness reached a local maxima seemed to improve the final achieved solution. The timings achieved were then compared to the timings predicted by well known analytical method known such as the Webster method. The cycles times of the solutions were significantly shorter than the cycle times predicted by the Webster method, but the green portion (ratio of green time to total cycle time) was similar to the Webster method. The study claimed that effect green portion on traffic light performance was more profound than the effect of cycle time. It was then noted that in the representation used in the study the green portion could not be varied independently from cycle time and suggested that representations where green portion and cycle time are independently encoded could be worth exploring. It was then concluded that the genetic algorithm produced results similar to but slightly better than the Webster model and that the choice of genetic algorithm parameters heavily affected its performance.

Rophail et al. studied the use of genetic algorithms in optimising the traffic light timings of a 9 traffic light traffic network in the city of Chicago. A custom objective function called modified network queue time was designed. The function depends mainly on overall queue time but also includes a factor to penalise long queues. A microscopic stochastic traffic simulator was used to calculate the objective function for the genetic algorithm. Queue time was used instead of overall delay due to a limitation in the simulation software. Initially the genetic algorithm failed to converge, so the search space was reduced to the space surrounding a solution that calculated using Websters model. Consequently the model converged and a large reduction in average delay was reached. The resulting timing profile was then compared to the performance of 12 macro-model based optimisation techniques and the genetic algorithm was found to outperform all of them. 100 runs in the microscopic simulator was used as basis for the comparison. The simulator includes a degree of stochasticity to model real world behaviour of drivers, so some degree of variance between runs of the same traffic light timing profile is expected. The variance of the profile produced by the genetic algorithm was found to be lower, meaning the profile performed well more consistently than the macroscopic-model based timing profiles.

2.3 Hyper-Heuristics

Heuristics are rules of thumb in optimisation that will possibly - but without guarantee - generate a good solution. Heuristics are used to generate good solutions to computationally hard optimisation problems. The term *hyper-heuristic* was first coined in 2000 by Cowling to refer to heuristics for choosing heuristics in combinatorial optimisation, however the concept can be traced back to the 1960's. Techniques to automatically configure parameters for evolutionary algorithms can be considered as forerunners to modern hyper-heuristics. Heuristics are problem specific, but hyper-heuristics are general problem-independent guidelines to designing heuristic based optimisation algorithms [20]. Most hyper-heuristics are algorithms that take a group of low-level heuristics and apply them to an instance of the problem to yield a good solution. In most of the literature hyper-heuristics refers to such algorithms but there also exists another class of hyper-heuristics are known as *generational hyper-heuristics* and they are used to algorithmically generate low-level heuristics. Generational hyper-heuristics typically take a template of a generic algorithm and automatically tailors it to fit a specific problem. The Teacher system is a machine-learning based example of such systems and has been successfully applied to many problems such as circuit placement, process mapping and load balancing [21]. Generational hyper-heuristics have a special appeal for use, for if they are run once the generated low-level heuristics are directly usable of different instances of the problem, in contrast to traditional hyper-heuristics which need to be re-run on each instance of the problem. This means the use of generational hyper-heuristics could lead to long-term performance increases and time savings.

For a hyper-heuristic to be effective at finding solutions it must maintain a balance between covering large areas of the search space (*exploration*) and conducting local search to hone in on a specific solution (*intensification*). Hyper-heuristics can be classified into perturbative search algorithms and constructive algorithms. A constructive algorithm will take a partial solution with missing components and use heuristics to fill in the gaps to construct a solution whereas perturbative algorithms will start with a complete candidate solution and repeatedly modify it using heuristics to reach a better solution. A hyper-heuristic is a learning algorithm if it receives some sort of feedback from the search process. Learning hyper-heuristics attempt to learn when to apply which heuristic to achieve good results. The learning can be offline, done before the algorithm is used on specific instance of the problem, or online where the algorithm learns while attempting to solve a specific problem instance. In online learning approaches the algorithm conducts two simultaneous searches, one over the sequence of heuristics and the other over the solution-space of the problem.

Perturbative hyper-heuristics are either *single-point* where the algorithm maintains throughout execution only one candidate solution that it tries to improve, or *multi-point* where the algorithm maintains multiple independent solutions and attempts to improve them simultaneously. Perturbative hyper-heuristics that perform single-point search are known as *Selection-Move Acceptance* hyper-heuristics because these types of hyper-heuristics have two roles, one each iteration they first select a heuristic to apply to the current solution. They then make a decision whether to keep the newly generated solution as the new current solution or to discard the newly generated solution and keep the old current solution. Some perturbative selection-move acceptance hyper-heuristics attempt to learn the selection process from the experience gained while solving an instance of the problem. Some assign each heuristic a score based upon its performance. To do this the hyper-heuristic must first assign all heuristics an initial score, then it must implement some sort of score update for the heuristics based on whether its use improved or worsened the solution. It must then also have some sort of memory policy deciding on how scores change over time. Finally it must implement a strategy to select heuristics based on their scores, such as a maximum score strategy or a roulette wheel strategy. Other hyper-heuristics opt for a reinforcement learning scheme where the hyper-heuristic punishes heuristics for worsening moves and rewards them for improving moves. A hyper-heuristic that does not attempt to learn the selection process must use either an exhaustive selection process or a random selection process [22].

Move acceptance methods are either deterministic or non-deterministic. Most non-deterministic move acceptance strategies keep track of some extra parameters such as time or temperature that are factored into the move acceptance decision. Examples of hyper-heuristics that use deterministic move acceptance rules are *simple random descent, random gradient, random permutation, random permutation gradient, greedy* and *choice function*. *Simple random descent* selects a heuristic at random on each iteration and accepts the new solution only if it is an improvement on the current solution. *Random gradient* selects a heuristic at random and repeatedly applies it until it stops resulting in improvements. *Random permutation* generates a permutation of the heuristics each iteration and applies them in that order. *Random permutation gradient* generates a permutation of heuristics and applies them in that order repeatedly until it no longer results in improvement, only then does it change the permutation. *Greedy* applies all of the heuristics to the current solution to generate candidate solutions and then selects the best candidate. Greedy is considered to be a simple learning heuristic selection method with extremely short memory. The *choice function* selection scheme assigns each heuristic a score based on how well it has performed in general, how well it has performed as a successor for the previously applied heuristic and how long it has been since it has last been used. Roulette wheel or maximum strategies are then applied each iteration to select a heuristic based on the scores. Two move acceptance strategies have been tested for use with the choice function, namely *All moves (AM)* and *Only Improvements (OI)*. *Reinforcement learning* based move selection has also been coupled with *All Move* acceptance and has yielded good results in the logistics domain [22].

Other hyper-heuristics use non-deterministic move-acceptance coupled with no learning. Examples of such hyper-heuristics are *Monte Carlo based move acceptance, the great deluge acceptance criteria, simulated annealing* and *late acceptance*. *Monte-Carlo acceptance strategies* that accept all improving moves and accept non-improving moves with a calculated probability have been proposed, they include *linear (LMC), exponential (EMC)* and the more sophisticated *EMCQ* move acceptance. The Monte Carlo move acceptance criteria were tested on an electronic component placement problem and compared against the combinations of simple random and choice function selection coupled with all move and only improving move acceptance. Simple Random-EMCQ was the top performer on the problem. In the *great deluge* acceptance criteria any move is accepted as long as it is not worse than an acceptable objective value, known as level and the level is linearly updated on each iteration until reaches the target value. In *simulated annealing* move acceptance all improving moves are accepted and non-improving moves are accepted with a probability that depends on the objective function difference and the current *temperature*. Simple random coupled with simulated annealing hyper-heuristics outperform simple random, greedy and choice function in many cases and is widely used. Late acceptance is a memory-based method that makes the decision to accept a move or not based upon the previous L moves where L is a configurable parameter [22].

Another group of hyper-heuristics couple online learning with non-deterministic move acceptance. An example of such algorithms would be a hybrid algorithm between reinforcement learning with Tabu Search and simulated annealing with reheating. This algorithm was used to generate experimental data for a cosmetics company from real-world data. The algorithm outperformed a simple local search strategy (random descent).

The use of *multi-point* based hyper-heuristics (population based hyper-heuristics) has also gained momentum. *Ant colony optimisation* is a multi-agent based algorithm where each agent incrementally and non-deterministically constructs a solution [23]. This construction is modelled by tracing out a path in the graph of the solution space. Like real ants each agent deposits *pheromones* along the path it traces in an amount proportional to the performance of the solution it built. The pheromone concentrations are then used to build a probabilistic model that influences the behaviour of future agents increasing the probability that they will produce good solutions. Ant colony optimisation has been used to tackle sports timetabling and personnel scheduling with good results [22]. *Particle swarm optimisation* is another *multi-point* hyper-heuristic that is inspired from socio-psychological principles [24]. It is inspired from the models that describe the overall behaviour consistency of a crowd of people in the presence of individual differences. Each member of the population represents a solution and corresponds to a point that moves in the space of possible solutions (usually Euclidean). The population is connected together in a topological network where each point has a number of neighbours. The population is iteratively improved by each point moving both individually and in coordination with its neighbours [24].

The scientific level of understanding of different hyper-heuristics has still not reached a point where there is a definitive guide to which hyper-heuristics work best on each problem. In 2006 Bilgin et al. conducted a study on 35 selection-move acceptance hyper-heuristics on 14 benchmark functions and 21 exam time-tabling problem instances. Different hyper-heuristics did well on different problems and no single hyper-heuristic dominated across all the tested problems [25].

The low-level heuristics used by the selection-move acceptance family of hyper-heuristics are classified either as *hill climbers* (memes) that improve upon an existing solution or as *mutational* heuristics that randomly perturb the solution in a certain way regardless of whether the solutions quality is increased or reduced.

2.3.1 The Use of Hyper-Heuristics in Traffic Signalling Optimisation

The use of multiple different hyper-heuristics in traffic signal optimisation has been studied with promising results. Dong et al. studied the use of multiple hyper-heuristics in optimising the timings of a 3 by 3 grid network of traffic lights. The algorithms they tested were *chaos-genetic algorithm, chaos-particle swarm optimisation, catastrophe-particle swarm optimisation* which are modified versions of the genetic algorithm and particle swarm optimisation algorithm. They also used a *simulated annealing-particle swarm optimisation* hybrid algorithm [26]. The modified versions of the genetic algorithm and particle swarm optimisation were found to be superior to the classical genetic algorithm and particle swarm algorithm and all algorithms tested were found to achieve good results. Burvall and Olegard conducted a comparison between genetic algorithms and simulated annealing for configuring the timings of a single intersection and found them to achieve similar results [27]. Simulated annealing was found to scale better as the traffic flow and simulator run time were increased, however the genetic algorithm scaled better with respect to increases in search space size. Garcia-Neito et al. conducted a study on the use of particle swarm optimisation for optimising the timings of two traffic networks taken from the cities of Bahia Blanca (Argentina) and Malaga (Spain) [28]. They proposed a customised version of the particle swarm optimisation algorithm and found it to scale well in the increase of the number of traffic lights. The performance of the proposed custom algorithm was then compared with the performance of the standard particle swarm algorithm and the differential evolution algorithm. The proposed algorithm was found to be better than the other two tested algorithms by a statistically significant margin. Saba et al. looked into the use of a memetic algorithm for online optimisation of traffic lights timings for networks taken from Brisbane (Australia) and Plock (Poland) [29]. The system used was designed to generate traffic light timings from readings taken from induction loops in an online adaptive manner. The memetic algorithm used in this study was an algorithm that combines the genetic algorithm with local search to improve performance and speed up convergence. An novel indicator scheme was proposed to help balance the memetic algorithms behaviour between diversifying the solutions (exploring) and increasing their quality (intensifying). The proposed memetic algorithm was found to be statistically better that the genetic algorithm.

2.4 Reinforcement Learning

2.4.1 Reinforcement Learning Theory

Reinforcement Learning (RL) is a technique for solving sequential decision making problems [30] with roots in artificial intelligence and computer science. The primary goal of reinforcement learning is to solve the sequential decision making problem by designing and training autonomous agents that interact with the environment to learn the optimal behaviour, these agents improve over time through trial and error. Thus reinforcement learning methods provide a mathematical framework for experience-driven autonomous learning [31].

In general RL is learning through interaction. An RL agent interacts with its environment, upon observing the consequences of its actions it learns to alter its own behaviour in response to rewards received. The concept of reinforcement learning has been adopted from studies on human behavioural psychology [32].

2.4.2 Reinforcement Learning as Class of Machine Learning

Machine learning is a sub-field of computer science that utilises statistical techniques and data analysis to give computers the ability to identify patterns and learn from data (to improve performance progressively) without being explicitly programmed [33]. Supervised machine learning is a branch of machine learning where we try to solve the problem of taking a vector $x = (x_1, ..., x_n)$ of inputs and producing a vector $y = (y_1, ..., y_m)$ of outputs, and the task is to find a function g such that $y = g(x)$ for all inputs. Supervised machine learning problems can be regression problems or classification problems. In classification problems, the entries $x_1, ..., x_n$ are features of an item to be classified, and the corresponding output y is the class label for that item. Typically, we are satisfied if we can approximate the 'true' function g to sufficient accuracy by a function f of some particular form. These forms can be polynomials whose coefficients need to be determined or neural networks whose parameters need to be determined [34].

There is a major difference between reinforcement learning and supervised learning. Generally in reinforcement learning, the learning is not in the form of input/output pairs, instead, whenever an agent chooses an action, it is given a reward, and the agent learns which action to take in each state based on the rewards. In supervised learning, we intend to learn $a = \pi(s)$ from previous inputs (a, s) where in reinforcement learning we intend to learn $a = \pi(s)$ from a reward r that is related to s. s, a, r stand for state,action, and reward respectively, $\pi(s)$ gives the optimal action to take at a state s.

2.4.3 The Standard RL Model

In the standard RL model, an agent interacts with its environment via perceptions (observations) and actions. The the agent interacts with the environment in discrete time steps. At each time step the agent receives an input that carries information about the environment at that time step, this input is modelled by the state of the environment. Then the agent applies an action to the environment and receives a real valued reward that tells it how good the action was, this action drives the environment into a new state. The agent guided by RL algorithms learns from the the states, actions, and rewards the optimal state-action mapping. The whole process is repeated over and over until the agent learns to act optimally so that it will choose the action that maximises the long term sum of rewards.

Formally, the RL model consists of

- A set of environment states, $\boldsymbol{S}$;
- A set of agent actions, $\boldsymbol{A}$;
- A set of scalar reinforcement rewards, $\boldsymbol{r}$;
- A policy π that maps states to actions describing the agent's behaviour.

The goal of reinforcement learning is to find a policy π, which will maximise the long term collection of reinforcements (rewards).

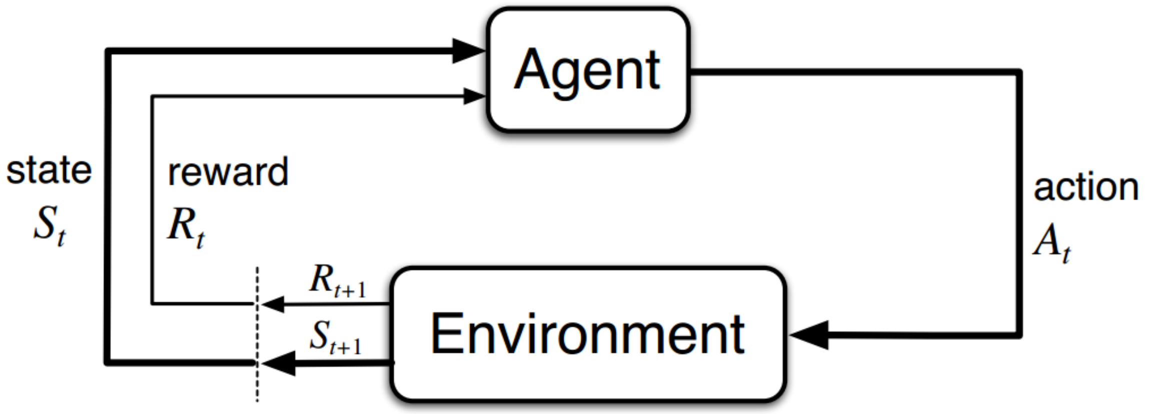


Fig. 2.1 The agent-environment abstraction in a reinforcement learning model taken from [1]

Reinforcement Learning Environment

The environment in the reinforcement learning model is assumed to be non-deterministic meaning that taking the same action from the same state on different occasions may lead to different future states or different reward values. However the environment is assumed to be stationary, i.e. the probabilities of making a state transitions are fixed overtime [31].

Delayed Rewards

In reinforcement learning problems, the actions the agent performs on the environment determine not only its immediate reward, but also the next state of the environment(probabilistically). Thus the agent must always take into account future states as well as the immediate reward when choosing among actions. The agent must be able to learn and then decide which of its actions are desirable based on rewards that can take place arbitrarily far in the future.

2.4.4 Modelling Optimal Behaviour

Modelling the optimal behaviour is essential to develop the learning algorithms for reinforcement learning. Therefore modelling how the agent should take the future rewards into account is crucial as it affects the way it takes decisions now. Many models for taking the future into account have been developed. It's worth mentioning that the choice of optimality model changes the optimal policy which the agent strives to learn.

The Finite Horizon Model

The simplest model is the finite horizon model; at any point of time, the agent should optimise the expected rewards for the next n steps:

$$E[\sum_{t=0}^{n} r_t] \tag{2.1}$$

Any future time steps after n are not taken into account, i.e. the agent doesn't care about what will happen after that. The limitation of this model, is the choice of the number of steps in the future n that the agent takes into account, is rather arbitrary, and it's usually difficult to be determined appropriately [35].

The Infinite Horizon Model

This model takes into account all of the long run rewards. However, rewards received in the future are geometrically discounted,

$$E[\sum_{t=0}^{\infty} \gamma^t r_t] \tag{2.2}$$

γ is the discounting rate, it falls in the range (0,1) and it favours immediate rewards over future rewards. It also serves to bound the infinite sum [35]. This is the most widely used model.

2.4.5 Measuring Learning Performance

It is necessary to assess and evaluate the quality of reinforcement learning algorithms to be able to design better algorithms. To evaluate the quality of the learning algorithms, several measures have been introduced. The most importance measures are eventual convergence to the optimum and the speed of convergence [35].

Eventual Convergence to Optimality

This measure is about describing if the algorithm eventually converges to optimality or not. Many algorithms have been proven to converge to optimal behaviour. This measure is not practically used, because eventual convergence to optimality doesn't give sufficient information about how good the algorithm is [35].

Speed of Convergence to Optimality

This measure is about how fast the algorithm reaches optimality. But in practice it is the speed of convergence to near-optimality. A related measure that is used is the level of performance after a given time [35].

2.4.6 Markov Decision Processes (MDPs)

To model the problems of reinforcement learning mathematically, Markov decision processes (MDPs) are used. Markov decision processes formally describe an environment for reinforcement learning [36]. The Markov property states that the future is independent of the past given the present. Mathematically a state S_t is Markov if and only if

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1,...,S_t]$$

In other words if given the current state no further information about the present can be extracted from the previous history. A Markov decision process is a group of tuples (S, A, P, R, γ) where,

- S is a finite set of states.

- A is a finite set of actions.
- P is a state transition probability, $P^a_{ss'} = P[S_{t+1} = S^{'} | S_t = s, A_t = a]$
- R is a reward function, $R^a_s = E[R_{t+1} | S_t = s, A_t = a]$
- γ is a discount factor $\gamma \in [0,1]$

To solve reinforcement learning problems, first we model them as Markov decision process problems. i.e. developing the states, actions, rewards and transition scheme. The second step is to solve the MDP that models the problem.

2.4.7 Solving Markov Decision Processes

Solving a Markov decision process, is finding the optimal policy π^* that gives the optimal mapping form states to actions.

A policy in an MDP defines the behaviour of the agent at each step. Formally a policy is described as a distribution over actions given states [30], mathematically an MDP policy is defined as

$$\pi(a|s) = P[A_t = a | S_t = s] \tag{2.3}$$

Many algorithms have been introduced to solve MDPs. When we're given a model, dynamic programming techniques are used to derive the optimal policy. When a model isn't given, other techniques are used to learn the optimal policy. In the following sections, all the derivations for optimal policies consider only the infinite horizon discounted model. However it has been proven that for any finite horizon model, there exists an optimal stationary policy [36].

The Value Function and The Q-Function

Before discussing how to derive the optimal policy, we need to introduce a new term, namely the *value* of a state, or the *utility* of a state. It is the expected infinite discounted sum of rewards for an agent starting from state s and acting with a policy π. The value function represent how good a state is for an agent to be in. The value function depends on the policy by which the agent picks actions to perform [30]. Mathematically the value function is expressed by

$$V(s) = E[\sum_{t=0}^{\infty} \gamma^t r_t] \tag{2.4}$$

Among all possible value-functions, there exist an optimal value function V^* in which the agent starts at state s and acts optimally thereafter. This value function is unique and it is found by solving the simultaneous non-linear equations

$$V^*(s) = \max_{\pi}(\sum_{t=0}^{\infty} \gamma^t r_t) \tag{2.5}$$

The the optimal policy π^* then, is the policy that corresponds to the optimal value function. Given the optimal value function, we can mathematically express the optimal policy as

$$\pi^* = \arg\max_{\pi}(V^{\pi}(S)) \tag{2.6}$$

It is clear that the value function is very instrumental to finding the optimal policy in reinforcement learning problems modelled as MDPs. However, in practise, another function is used, a modified version of the value function known as the Q-function or the Quality function. The difference between the value function and the Q-function is that the Q-function states the value of the state action pair when taking a policy π, i.e. it states what is the value of arriving at state s and leaving via action a and acting optimally thereafter [30]. Mathematically the Q-function can be described as,

$$Q^{\pi} = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_t = s, a_t = a] \tag{2.7}$$

The relationship between the value function, and the Q-function can be described by the equation

$$V(s) = \max_{a} Q(s,a) \tag{2.8}$$

and the relationship between the Q-function and the optimal policy can be described with

$$\pi^*(s) = \arg\max_{a} Q^*(s,a) \tag{2.9}$$

Where $Q^*(s,a)$ is the optimal Q-function.[30]

2.4.8 Finding the Optimal Policy

When a correct model for the environment is given, i.e. the states, actions, rewards and the state transition probabilities are given, finding the optimal policy is straight forward using several methods. These methods will help as a foundation to develop other techniques to learn the optimal policy when the correct model is not given.

In order to find the optimal policy π^*, first we need to express the optimal Q-function as

$$Q^*(s,a) = R(s,a) + \gamma(\sum_{s' \in S} P(s'|s,a)V(s')) \tag{2.10}$$

This is the famous Bellman equation [37], the value function can then be found by solving the simultaneous equations

$$V^*(s) = \max_a Q^*(s,a) \tag{2.11}$$

The optimal value function is unique and can be used to derive the optimal policy using,

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{2.12}$$

Thus finding the value function is the key to solving an MDP. However, due to the non-linearity of the equations above solving them is difficult, hence other techniques need to be used, namely value iteration and policy iteration which are numerical algorithms used to estimate the the optimal value function and the optimal policy respectively.

Value Iteration

It is a method to find the optimal policy, also known as Backward Induction. In value iteration we iteratively use the Bellman equation to find the optimal value function, you begin with a random value function and then improve that value function in an iterative process, until the optimal value function is reached, the exact process shown in Algorithm 1.

Algorithm 1: Value Iteration Algorithm

```
1 repeat
2     foreach $s \in S$ do
3         foreach $a \in A$ do
4             $Q(s,a) \leftarrow E[r|s,a] + \gamma(\sum_{s' \in S} P(s'|s,a)V(s')$
5             $V(s) \leftarrow \max_a Q(s,a)$
6 until $V(s)$ converges;
```

It is clear that the policy used here is the greedy policy with respect to $V(s)$. Finding a stopping criterion can be challenging. Williams & Baird et al. (1993) developed an effective stopping criterion which can be used [38], also Puterman (1994) introduced another stopping criterion based on the span semi-norm that helped to allow earlier termination [1]. It is worth mentioning that the greedy policy is guaranteed to reach optimality even before the convergence of the value function [31] and this result is used in practice.

Policy Iteration

In policy iteration, you begin with a random policy, then on each step you compute the value function of that policy, then you find a new (improved) policy based on the computed value function, and so on. It is proven that each policy is guaranteed to be an improvement over the previous one unless optimality has been reached [38]. The value function is found using Bellman equation and is used to find the improved policy on the next step. In policy iteration, it is clear that we directly modify the policy instead of acting greedily based on the optimal value function to derive the policy. The algorithm for finding the policy is shown in Algorithm 2.

Algorithm 2: Policy Iteration Algorithm

```
1 Initialise with an arbitrary policy $\pi'$ **repeat**
2     $\pi \leftarrow \pi'$
3     Find the value function using the policy $\pi$ by solving the linear equations
        $V_\pi(s) = R(s,\pi(s)) + \gamma(\sum_{s' \in S} P(s,\pi(s),s')V_\pi(s'))$
4     Improve the policy by computing
        $\pi'(s) \leftarrow \arg\max_a(R(s,a) + \gamma(\sum_{s' \in S} P(s,a,s')(V_\pi(s'))))$
5 **until** $\pi = \pi'$;
```

2.4.9 Learning to Solve MDPs

The value iteration and policy iteration techniques were used to find the optimal policy to solve an MDP when a model is given, i.e. when the state transitions and reward functions are known in advance. In this case the solution can be found before the agent actually begins interacting with the environment. This is known in AI as planning, and the algorithms above are classic planning algorithms to solve Markov decision processes, and the policy iteration and value iteration are dynamic programming algorithms, these algorithms require a perfect model of the environment to work well [1].

Generally reinforcement learning problems require treatment that differs from planning problems in the sense that in RL problems the model isn't known, i.e. the state transition probabilities and the reward function are not known and the agent doesn't know neither how its actions might change the environment nor the reward it might get. The main concern in RL is to find the optimal policy directly without knowing in advance the response of the environment, so the agent interacts with the environment in order to compile the information needed to figure out the optimal policy [1].

There are two approaches to deal with these kind of RL problems, either the agent learn the dynamics of the environment, and develops a model to use to find the optimal policy and this approach is known as model-based reinforcement learning, else the agent completely relies on trial and error, (actions and rewards) to infer how to directly estimate the Q-function of each action in each state as in Q-learning algorithm, or the agent develops a scheme to map states to actions as in policy search algorithms. This approach is known as model-free reinforcement learning. Generally the model-free approach is more widely used due to the difficulty of developing a model for the environment and model-free approaches are more robust when dealing with continuous states and actions [1].

2.4.10 Model-free Reinforcement Learning Strategies

In reinforcement learning problems, the actions we take might result in a reward arbitrarily far in the future and it can be difficult sometimes to know whether the action taken is good or bad this is known as the *Temporal Credit Assignment Problem*. In episodic scenarios, Monte Carlo techniques can be used to decide upon actions, the agent waits until the end of the episode and decides from the result if the actions taken were good. However this technique is only useful in an episodic scenarios where the agent runs for a finite amount of time steps. It is not easy to implement Monte Carlo techniques in non-episodic scenarios and ongoing tasks when there is no end and the agent keeps interacting with the environment, also this technique is only useful when the episode requires a small deal of memory. An alternative to Monte Carlo techniques is a class of algorithms derived from value iteration they are called temporal differences learning.

Temporal Differences Learning

In the model-free approach of reinforcement learning, a class of algorithms share the notion of temporal differences which in its essence refers to the process where an agent learns an estimate of values based on previously learned estimates (as in statistical bootstrapping). Temporal differences learning is a process where the agent on each environment time step generates a learning estimate based on the immediate reward received and uses the information from the previous estimate and the difference between the temporally successive estimates [39]. The temporal differences was first introduced by Richard S. Sutton and it was influenced by the work on animal learning theories [40]. The name came from the use of the difference in predictions and estimates over time. The main idea of temporal differences is, at every time step the values are updated to get closer to the same value in the next time step. Just like dynamic programming, updates are performed based on the current estimates without waiting for the final outcome.

TD algorithms have advantages over dynamic programming methods, they learn directly from experience without the need for a model. Also TD methods have an advantage over Monte Carlo methods, they can be applied on-line, i.e. the updates are applied after each step, so that the algorithms can be applied in a continuous environment (non-episodic), they can learn from an incomplete sequence, and the most important advantage is the fact the memory needed in TD learning is minimal. However, temporal differences learning is more sensitive to initial conditions, but in general temporal differences learning is more efficient than Monte Carlo approaches [39].

Q-learning

Q-learning is the most widely used model free technique of reinforcement learning, it is a specific TD algorithm used to learn the Q-function. Because Q-learning is a model-free technique, in the Q-learning framework, an agent performs an action at a particular state and evaluates its consequences in terms of the immediate reward it receives and the estimate of the value of the state to which it has been taken to. By trying all the actions in all the states repeatedly, it learns which are best overall, judged by long-term discounted reward [41]. In Q-Learning the state-action pairs Q-function is approximated from the samples of $Q(s,a)$ that are observed during interaction with the environment. It is proven that in Q-learning, the learned Q-function converges to the true Q function and hence the optimal policy can be used [42]. The optimal policy can then be directly derived from Equation 2.9.

Function-approximation

In model-free learning algorithms, we have assumed so far that a state value function, or a state-action value function (Q-function) is easily defined. These functions can be implemented using look-up tables. However in practice, for most of the problems the state-space and the action-space might too large to be represented tabularly and learning it would be too slow, in other words to learn the value of each state individually is too expensive, also, we might face a problem when the action-space and/or the state-space are continuous and hence, representing the value functions comprehensively is neither easy nor efficient. Hence, the need for functions that adequately approximate the value function remains essential.

There are numerous ways to approximate the value function. Linear combinations of features, neural networks, decision trees, Fourier and wavelet bases have been used [43]. The most celebrated function approximators are the neural networks, it has been empirically demonstrated that using neural networks especially deep neural networks as function approximators is very useful. When using deep neural networks as function approximators they may potentially become unstable. But several techniques have been implemented to reduce the instability, and to speed up the convergence [43].

Using neural networks in Q-learning

Almost all of modern reinforcement learning agents that use the Q-learning algorithm today use neural networks to approximate the Q-function. At the beginning of the learning processes, the neural network parameters are randomly initialised. The feedback from the environment in terms of the difference between the expected reward and the observed reward is used to adjust the parameters of the network to improve the future interpretation of similar state-action pairs. A major issue with using neural networks to approximate the true Q-function is that, we can get stuck at a sub-optimal behaviour [43]. Several techniques have been developed to avoid this problem and they are discussed in the next sub-section.

Exploitation Vs Exploration

In Q-learning and other online model-free learning strategies, it is important to balance between exploration and exploitation while trying to learn the optimal behaviour. Exploitation is choosing the action believed to be best given the current information. Exploration on the other hand is choosing an action not currently believed to be the best action in-order to develop a better estimate of the optimal policy.

In the context of Q-learning, many strategies have been used to achieve the appropriate balance. The most widely used approach is the ε-greedy approach in which the best action (the action that maximises the Q-function i.e. the greedy action) is chosen in all but a fraction of ε of the times and a random action is chosen otherwise, where $0 \leq \varepsilon \leq 1$. ε varies from one application to another, and its value can be changed during the learning process. Other techniques like the temperature approach where exploration is very spacious at the beginning of the learning process and decreases with time, as well as optimism in the face of uncertainty are also used [31]. A general heuristic for devising a trade-off between exploration and exploitation is to make use fact that the best long-term solution may involve short-term sacrifices, and gathering more information is useful to making optimal decisions in the long run.

2.4.11 Policy Search

The previously introduced Q-learning method and the temporal differences learning algorithms attempt to find the optimal state value function or the optimal state-action value function and then derive the optimal policy. Another approach to find the optimal policy is to find it without the use a value function, this is known as the policy search. This is achieved by choosing a parameterised policy π_θ, the parameters of this policy are adjusted so that eventually the policy will generate the optimal overall expected returns. The adjustment of the parameters can stem from either gradient-based or gradient-free optimisation (using heuristics like genetic algorithms) [44]. The most popular parameterisation and encoding of the policy function is the approximation using neural networks, neural networks are very useful because of their ability to encode a large number of parameters and the advancements in deep learning techniques have led to better performance. Although gradient-free optimisation can be easier to implement and can optimise non-differentiable policies, gradient based optimisation has been found to be more efficient and has produced better results in practice, therefore it is the most celebrated approach to optimise policies function [44].

Knowing that a policy is a distribution from states to actions, a policy neural network directly outputs the probabilities of taking a certain action. A certain state is fed as an input to the policy network and the reward received is used to favour or disfavour the action current (i.e. increase the likelihood if the reward is high and and reduce the likelihood of the action if the reward is low). For continuous actions the output can be the parameters of the distribution the actions belongs to (e.g. mean, standard deviation), for discrete actions the output is the individual probabilities of the actions [44].

Policy search methods specially when gradient optimisation is used, offer better convergence and are able to learn stochastic policies, but they generally they converge to local optima.

Policy Gradient Method

Policy gradient is a policy search method to learn the optimal behaviour in a model-free approach by optimising the policy (parameterised and approximated with any form of function approximation) using the discounted rewards by means of gradient descent. In the policy gradient framework, we have

$$\pi_\theta = P[a|s] \tag{2.13}$$

In order to find the best policy, we deal with the problem of finding this policy as an optimisation problem, and we define an objective function that is a function of the parameters of the policy, the objective function can take many forms, but the most widely used is the discounted sum of rewards.

$$J(\theta) = E_{\pi\theta}[\sum r^\gamma] \tag{2.14}$$

In the policy gradient method we traverse the search space to find the parameters that lead to a local optima by following the gradient of $J(\theta)$ and updating the parameters as in

$$\Delta\theta = \alpha\nabla_{\theta}J(\theta) \tag{2.15}$$

α is the learning rate and $\nabla_{\theta}J(\theta)$ is a vector of partial derivatives of the objective function with respect to the policy function parameters. In practice, for differentiable policies the gradient of the objective function is dealt with by using the negative log likelihood operation, and the policy gradient is defined as

$$\nabla_{\theta}J(\theta) = E_{\pi\theta}[\nabla(log\pi(s,a,\theta))R] \tag{2.16}$$

where R is the observed discounted reward, the derivation of this relationship is provided in [45]. This equation can be interpreted as if the operation tries to modify the policy function to increase the probabilistic likelihood of an action if the action results in a high discounted reward, and decreases the probabilistic likelihood of an action if the action results in a low discounted reward. The relationship above is for a version of policy gradient known as *reinforce policy gradient*, another technique uses the value function instead of the reward and is known as *actor-critic policy gradient*[46]. The policy gradient technique is a Monte Carlo technique, and the parameters are updated after each training episode.

2.4.12 The State of the Art Reinforcement Learning

Over the past few years, reinforcement learning has been the state of the art technique to solve decision making problems in which agents interact with environments. Being heavily adopted in games and optimal control, reinforcement learning has shown great performance, surpassing the human level performance in many cases. The most important breakthroughs in reinforcement learning were due to the advent of deep learning, deep learning has had a huge impact on machine intelligence in the past decade. Deep learning has made it very easy to extract the low-level representation of features from high-dimensional data [35]. Deep learning has boosted the progress in reinforcement learning, giving rise to deep reinforcement learning (DRL) which is the result of using the aid of deep learning to tackle reinforcement learning problems.

The very first major contribution of deep reinforcement learning was the development of an agent that could play Atari games at super human level, it was deciding what are the actions to take directly from raw images [47]. This agent has demonstrated that RL agents can learn quiet well from high dimensional representations of states and rewards. The algorithms used in creating this agent have provided a solution to the instability caused by the function approximation. The most celebrated break through of DRL came with AlphaGo which is a computer agent that plays the board game Go [48]. This agent has defeated the world Champion of Go. This agent used a combination of reinforcement learning along with classical artificial intelligence search heuristics.

The applications of reinforcement learning are not limited to agents playing games, it has been used for optimal control & navigation in robotics [49]. In each case, DRL has been an excellent tool to learn directly from images or from other high-dimensional representations of data.

2.4.13 The Use of RL in Traffic Optimisation

The use of reinforcement learning in traffic light optimisation is an emerging trend. In the past two decades, RL optimisation has been applied to traffic light control, however it was rudimentary and the techniques at that time used tabular Q-learning which is very difficult to use with complex state-action spaces, also the processing power at the time didn't permit the use of sophisticated function approximation like neural networks and deep reinforcement learning. Thus in that time small-size state space and limited information were used in the modelling. El-Tantawy et al. summarise the research in the past two decades [50]. With the advent of powerful function approximation, more complex forms of information in traffic control have been modelled, represented and processed by RL algorithms, It is also worth mentioning the fact that the development of efficient traffic simulators has made it easier to learn in virtual environments, giving rise to more advanced and intelligent agents.

The majority of recent studies work in a slightly similar way. Wade Gendersa and Saiedeh Razavib [51] proposed a traffic light controller that used reinforcement learning, they introduced an effective state space representation, the discrete traffic state encoding, which encodes the information of the location and speed of vehicles on the road, their agent reduced the average queue length by 66% and reduced the average travel time by 20% in comparison to a previously used technique. They used Simulation of Urban Mobility (SUMO) simulator to evaluate their solution. Another example of traffic light optimisation using RL techniques in single intersection networks is the work of Xiaoyuan Liang and Xusheng Du [52]. In their research they used deep reinforcement learning to reduce the cumulative delay of all the vehicles in a road network with a single intersection along with reducing the waiting time of each individual vehicle. They also used the SUMO simulator to evaluate their solution. In their solution the traffic signal is split into fixed-time intervals, and they manipulate the duration of the green/red intervals in each cycle. They used the change in cumulative waiting time between the cycles as a reward signal. A convolutional neural network was used as their function approximator. The state-space was comprised of the positions and speeds of the vehicles at the intersection. Their results produced a 20% reduce of the average waiting time. Marco Wiering et al. [53] also introduced an RL algorithm to control traffic lights in a multiple intersection network, they developed their own simulator namely the Green Light District (GLD) simulator to evaluate their proposed solution, Their simulations showed that the RL algorithms reduced average waiting times by more than 25% compared to pre-timed or static controllers. Most of the related work uses the Q-learning algorithm due to it's simplicity, reinforce policy gradient algorithms, to the best of authors' knowledge have yet to be studied and tested.

Chapter 3

Methodology

3.1 Problem Definition

The focus of this work is to look into different methods to control traffic lights with the aim of optimising the flow of traffic and reducing congestion. The metrics used to describe the flow of traffic are listed and explained in section 3.1.1. In this work we studied two different modes of traffic light control, *pre-timed* control where the traffic light operates on a timer and changes from each phase to the next after spending a specific duration of time in each phase and *traffic-responsive* control where the traffic light takes real-time input from some sort of sensor and makes an independent decision at each moment of time to select which phase (combination of colours) to enact. In pre-timed control the traffic light timings are tuned according to the expected flow of traffic. This can be estimated by counting vehicles at the time of day that you want to estimate the traffic for. In traffic-responsive control there is no need to estimate the flow of traffic as the aim is to design a controller that adapts to the state of the road in real-time. To study both modes of control a traffic simulator SUMO [54] was used. SUMO is a microscopic traffic simulator, meaning it simulates the behaviour of each individual vehicle and driver. It does not resort to the use of equations that describe the flow of traffic as a whole. SUMO is available free of charge as it is open source software. It is widely used in industry and academia and the developers of SUMO organise an annual conference for its users for them to come and talk about their work.

3.1.1 Traffic Description Metrics

The flow of traffic can be described in two main different ways. The first kind of metrics describes the **throughput** of the traffic or the efficiency of the road, these metrics aim to capture both the number of vehicles passing through an intersection in a unit of time and the delay experienced by each vehicle. The other way traffic can be described is the **fairness** of the traffic. When trying to optimise the throughput the controller could potentially favour lanes with high traffic at the expense of the lanes with low traffic, this could lead to high traffic throughput but at the cost of making a small portion of vehicles wait disproportionately long times. The fairness metrics attempt to capture this imbalance and can be used to compare the fairness of two different traffic light control schemes. The metrics used for **throughput** in this work were:

1. **Mean Journey Time**: defined as the time taken by each vehicle to complete its journey from start to finish averaged over all vehicles in the simulation scenario.

2. **Mean Vehicle Speed**: defined as the average speed of each vehicle throughout its journey averaged over all vehicles in the simulation scenario.

3. **Mean Time Loss**: defined as the difference between the time the vehicles journey actually took and the time it would have taken had the vehicle been travelling at the maximum speed allowed on each road for the entire journey, averaged over all vehicles in the simulation scenario.

4. **Mean Vehicle Waiting time**: defined as the time each vehicle spent at a speed of less than $0.1m/s$ throughout its journey averaged over all vehicles in the simulation scenario.

These metrics are used to measure throughput across the literature are readily available from the simulator SUMO via the python API. There is less of a consensus of how fairness can be quantified so we designed two metrics for fairness. The first metric we devised is called the **Vehicle Speed Standard Deviation** (VSSD) and is defined in Equation 3.1. The second metric is the **Journey Time Standard Deviation** (JTSD) and is defined in Equation 3.2. These metrics can be used to compare the fairness of two different traffic light controllers when used on the same traffic simulation scenario.

$$VSSD = \frac{\sqrt{\sum_{i=1}^{n}(v_i - \bar{v})^2}}{n} \tag{3.1}$$

Where v_i is the average speed of the vehicle i throughout its journey, $\bar{v}$ is the average vehicle speed averaged over all vehicles and n is the total number of vehicles in the scenario.

$$JTSD = \frac{\sqrt{\sum_{i=1}^{n}(j_i - \bar{j})^2}}{n} \tag{3.2}$$

Where j_i is the journey time of vehicle i, $\bar{j}$ is the average vehicle journey time and n is the total number of vehicles in the scenario.

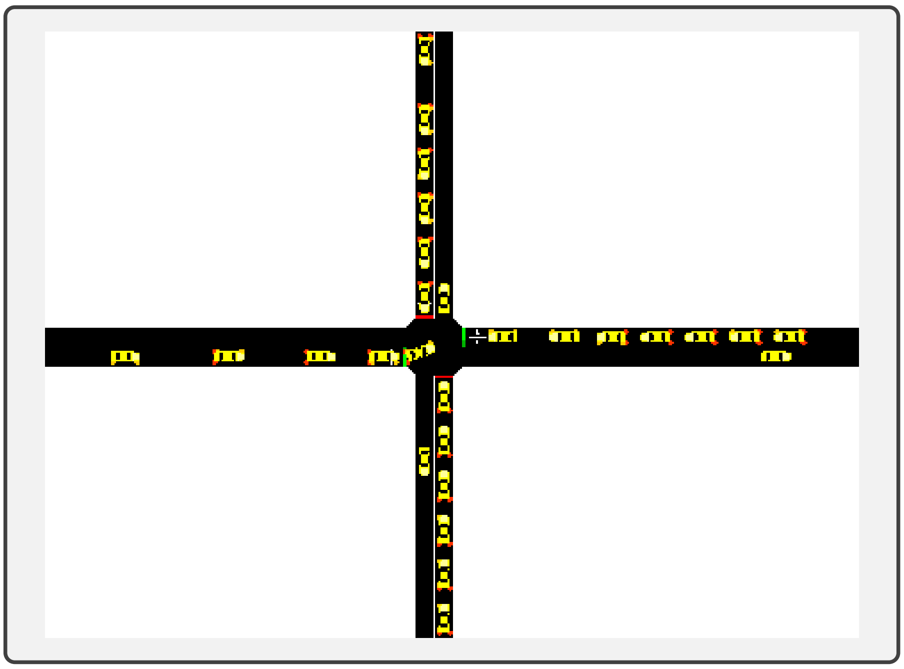


Fig. 3.1 A screenshot of the simulation of the problem scenario.

3.1.2 The Scenario

The scenario used was a single traffic intersection between two roads controlled by a traffic light. The traffic consisted of 528 vehicles departing at semi-random times between 0 seconds and 800 seconds. The simulation is run until each vehicle reaches its final destination. The simulation is cut off after a fixed amount of simulation time that should be sufficient for all vehicles to have arrived, even if there are any vehicles that are still on the map. This occurs when a non-functioning or very poorly function traffic light controller is used, such as a controller that only allows traffic in one direction to pass and completely blocks traffic in the perpendicular direction. An illustration of the scenario is shown in Figure 3.1. The simulation configuration files for the scenario are in Appendix B.

3.2 Pre-timed Traffic Light Optimisation

To find the optimal pre-timed timing plan for the scenario described in Section 3.1.2 three different optimisation techniques were used and they were:

1. Simple Random Improve or Equal (SR-IE)

2. Simple Random Simulated Annealing (SR-SA)

3. Genetic Algorithm (GA)

The traffic light was configured to cycle through all 8 phases in turn. 6 of the 8 phases were transitional phases (containing yellow) needed to facilitate the switch of flow of traffic from one direction to the other or were phases which only allowed turning traffic. Since the intersection had only one lane in each direction they were somewhat redundant. These 6 phases were set to have a fixed time of 3 seconds each. The duration of the two main phases *North-South all green* and *East-West all green* were optimised by the above mentioned algorithms. The search space for the optimisation algorithms was ${\mathbb{R}_+}^2$. Out of the 4 metrics we defined, we choose **mean journey time** as the objective function we sought to minimise, since it out of the 4 throughput metrics is the most tangible benefit to users of the road. To account for the random components of all 3 algorithms each of the 3 algorithms was run 10 times then statistical indicators of their performance were computed. The results were compared to each other and to the simulators default configuration of the traffic light.

3.2.1 The Selection - Move Acceptance Hyper-Heuristics

The first two algorithms used **SR-IE** and **SR-SA** are from the family of *Heuristic Selection - Move Acceptance* hyper-heuristic algorithms. This family of algorithms starts with an initial solution (usually a greedy or random valid solution) then iteratively improves it. On each iteration these hyper-heuristic first select a *Low-Level Heuristic* from a predefined set of heuristics and applies one of them to perturb the solution in some way to obtain a new solution. The hyper-heuristic then based on some criteria either accepts the newly generated solution as the current solution or discards it. A general framework for Selection-Move Acceptance hyper-heuristics is defined in Algorithm 3. For both hyper-heuristics used in this work the following set of low-level heuristics were defined

- **LLH$_1$**: Increase or decrease a randomly selected component of the solution (i.e. the amount of time the traffic light spends in the randomly selected phase) by a random number between [0, 5]
- **LLH$_2$**: Increase or decrease a randomly selected component of the solution by 10
- **LLH$_3$**: Increase or decrease a random number of components by a random number between [0, 1]
- **LLH$_4$**: Swap the two components of the solution

In both algorithms the solution was randomly initialised on each run to vector of two random integers sampled from the range [50,100]. Both algorithms were run for 1000 iterations

Algorithm 3: Selection hyper-heuristic algorithm

```
1  Let $LLH$ represent the set of low level heuristics
2  Let $S_{candidate}$ represent the current solution
3  Let $S_{best}$ represent the best solution
4  $S_{initial} \leftarrow$ initialise();
5  $S_{best} \leftarrow S_{initial}$;
6  $S_{candidate} \leftarrow S_{initial}$;
7  repeat
8      $LLH_i \leftarrow$ Select($LLH$);
9      $S_{new} \leftarrow$ ApplyHeuristic($LLH_i, S_{candidate}$);
10     $S_{best} \leftarrow$ updateBestSolution($S_{new}$);
11     if $Accept(S_{candidate}, S_{new})$ then
12         $S_{candidate} \leftarrow S_{new}$;
13     end
14 until termination criterion is satisfied;
15 return $S_{best}$;
```

Simple Random Improve or Equal

Simple Random Improve or Equal (**SR-IE**) uses simple random for heuristic selection where in each iteration a low-level heuristic is selected at random. It uses improve or equal for move acceptance. In this mode of move acceptance only solutions with an objective function value equal to or better than the current solution are accepted. Solutions that are worse than the current solution are discarded.

Simple Random Simulated Annealing

Simple Random Simulated Annealing (**SR-SA**) uses simple random heuristic selection identical to that of SR-IE. It uses *simulated annealing* move acceptance. Simulated annealing is inspired by the process of annealing used to condition metals where the metals are heated then cooled slowly. In simulated annealing move acceptance if the newly generated solution is better than the current solution it is accepted. If it worse it is accepted with a probability p calculated as follows:

$$p = e^{-\frac{\Delta f}{T}} \tag{3.3}$$

where Δf is the change in the objective function and T is the *temperature* a parameter that is a function of the iteration number. In this work the temperature was given an initial value of 1 and was multiplied by a *cooling factor* of 0.99 on each iteration.

3.2.2 The Genetic Algorithm

The genetic algorithm is a meta-heuristic algorithm used for optimising hard problems and is considered to be the state-of-the-art for many difficult problems. The the exact flavor of the GA used in this work is outlined in Algorithm 4. The parameters of the GA used in this work are shown in Table 3.1. The mutation used in this work was to add or subtract

Algorithm 4: Genetic algorithm

```
let $f(x)$ be the objective we seek to minimise
$population \leftarrow$ initialise();
**repeat**
    $parents \leftarrow$ select($population$, 2);
    $offspring \leftarrow$ crossover($parents$);
    $offspring \leftarrow$ mutation($offspring$);
    $worstChromosome \leftarrow$getWorstSolution($population$)
    **if** $f(offspring) < f(worstChromosome)$ **then**
        population.pop($worstChromosome$);
        population.insert($offspring$);
**until** *termination criterion is satisfied*;
**return** getBestSolution($population$);
```

Table 3.1 Parameters of the genetic algorithm

Parameter	Value
Population size	4
Crossover type	one-point
Mutation rate	50%
Selection	random selection

a random number between 0 and 15 from a random component of the solution, while also insuring the modified component remains non-negative. The genetic algorithm was run for 1000 generations before taking the best solution from the population as the final result. To recap the exact procedure used in this work the flow of the algorithm is described as follows:

1. An initial populations of candidate solutions of size 4 is generated randomly and the fitness of each population member is calculated.

2. Each iteration (generation) two members are selected from the population at random. A one point crossover is then used to generate one new offspring from the selected pair

3. With probability 50% the offspring is mutated otherwise it is left unchanged.
4. The offspring is evaluated, if its fitness is better than any of the population members then the worst population member is discarded and the offspring is inserted into the population, otherwise the offspring is discarded.
5. Steps 2 - 4 are repeated 1000 times.
6. The best member of the population is returned as the result.

3.3 Traffic Responsive Control

In this part of the work we sought to optimise the flow of traffic in the scenario described in Section 3.1 by designing traffic light controllers that responded in real-time to the state of the intersection. To achieve this goal we implemented two reinforcement learning based algorithms, namely:

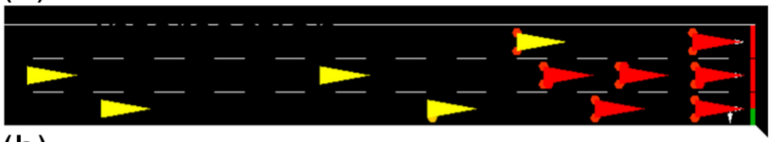
1. Q-Learning
2. Policy Gradient

The performance of the two algorithms was compared, they were also both compared against a classical non machine learning traffic light control algorithm called *longest queue first* (LQF). The LQF algorithm is a greedy algorithm used in scheduling optimisation problems, and the problems that deal with optimising the process of serving queues to reduce queue length and increase the throughput. This algorithms although its simplicity has proven to be very effective and thus it is the most used algorithm in traffic responsive optimisation.

In our implementation, we calculate the number of stopped vehicles in each direction. In the intersection the opposite directions together form one queue, and the length of the queue is determined by the sum of the stopped vehicles in the both directions, i.e. the length of $queue_1$ is the number of stopped vehicles entering the intersection from the North-South direction and the length of $queue_2$ is the number of vehicles entering the intersection from the East-West direction. On each time step, the decision is either to allow the vehicles in the North-South lanes to move, or is to allow the vehicles in the East-West lanes to move depending on the corresponding lengths of the queues.

Both RL algorithms are built around the notion training an agent to act in an environment. At each time step the agent is in a specific state s, the agent takes a specific action a from a set of predefined actions A. The agent then depending on the current state and the action it took transitions to a next state s'. Upon this transition the agent receives a reward r. If the reward

(a)



(b)

0	0	0	0	0	0	0	0	1	0	0	1
1	0	0	0	1	0	0	0	1	0	1	1
0	1	0	0	0	0	1	0	0	1	0	1

(c)

0	0	0	0	0	0	0	0	0.2	0	0	0
0.8	0	0	0	0.5	0	0	0	0	0	0	0
0	0.6	0	0	0	0	0.4	0	0	0	0	0

Fig. 3.2 This figure shows (a) a sample of traffic and the corresponding DTSE (b) boolean and (c) real vectors taken from [2]

received on the the next i^{th} action is r_i then the aim of the agent on each turn is to choose the action which maximises the long-term reward. The long-term reward is represented by the *discounted reward*. The discounted reward is calculated by $\sum_{i=0}^{\infty} \gamma^i r_i$ where γ is the discount factor and $0 < \gamma < 1$. In this work the value of γ was taken to be 0.9 .

In this work the the state was represented by the discrete traffic state encoding (DTSE) [2]. The DTSE takes the segment of road leading up to the intersection and breaks it into cells. Two vectors are then computed from the cells, each vector has one component corresponding to each block. The first vector is a vector of Booleans where each value represents the presence or absence of a vehicle in the corresponding cell. The second vector is a vector of reals, its components are equal to the value of the speed of the vehicle in the corresponding cell if the cell is occupied, 0 otherwise. An illustration of this is shown in Figure 3.2 taken from the authors of the DTSE's original paper [2]. We combined the DTSE representations of each the 4 segments of 112.5 meters of road leading up to the intersection to create the our state vector. We used a cell size of 7.5 meters. We computed the DTSE from the simulator API, however in reality the DTSE could be implemented by applying image processing to live video streams from cameras placed at the traffic light.

We experimented with two reward functions. The first reward function we selected was the **mean vehicle speed**. This is because the mean vehicle speed is affected more rapidly by traffic light and provides fast feedback to the agent. For instance if mean journey time was used as a reward function and the agent started performing poorly, that would only be reflected in the reward after the vehicles currently at the intersection completed their journey and contributed to the mean journey time. The mean vehicle speed penalise long queues by the virtue that queuing vehicles have a speed of 0 and bring down the mean vehicle speed. The second reward function that was tested was a novel function of this work. It is the **fairness penalised mean vehicle speed**. It is defined as follows:

$$FPMVS = \bar{v} - \beta \frac{\sum_{i=0}^{n} d_i}{n} \tag{3.4}$$

where $\bar{v}$ is the average speed of all the vehicles in the previous time-step, d_i is the number of contiguous seconds vehicle i has been still (has had a speed of less than $0.1m/s$) if it is still and zero otherwise, n is the number of vehicles currently in the simulation and β is a real-valued positive-valued configurable tuning parameter. This function is designed to penalise making vehicles waiting for long times. The longer the a vehicle is left waiting the higher the value of d_i and the lower the reward. This is to promote fairness.

The actions were choosing the next traffic light phase, the available phases were phases 0 and 4 (North-South all green and East-West all green). So on each time step (equal to one second of simulation time) the agent is given the current state then chooses which phase to give right of way to, the decision is then enacted by the traffic light.

3.3.1 Q-Learning

The Q-function is maximum possible discounted reward from a possible state if the agent takes optimal actions in the current state and all the following subsequent states. It is defined in environments with deterministic transitions as follows:

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a') \tag{3.5}$$

where $Q(s,a)$ is the Q-function, $r(s,a)$ is the reward function, γ is the discount factor, a is the current action, s is the current state, s' is the next state and a' is the next action. If the Q-function is obtained then choosing the next action is simple. The optimal policy would be to choose the action with the highest Q-value. In this work we attempted to approximate the Q-function with a neural network. The neural network is fed the computed state (DTSE vectors) and it estimates the Q-values for each of the two available actions. We experimented with multiple neural network architectures and decided on a neural network with 120 inputs (the size of the DTSE) two hidden layers with 100 neurons each and an output layer with 2 neurons. The hidden layers used *ReLU* activation functions and the output layer used a linear activation function. The network is illustrated in Figure 3.3 and its hyper-parameters are listed in Table 3.2. The networks weights were initialised to random values. To train a network neural network to fit a certain function, another helper function called the loss

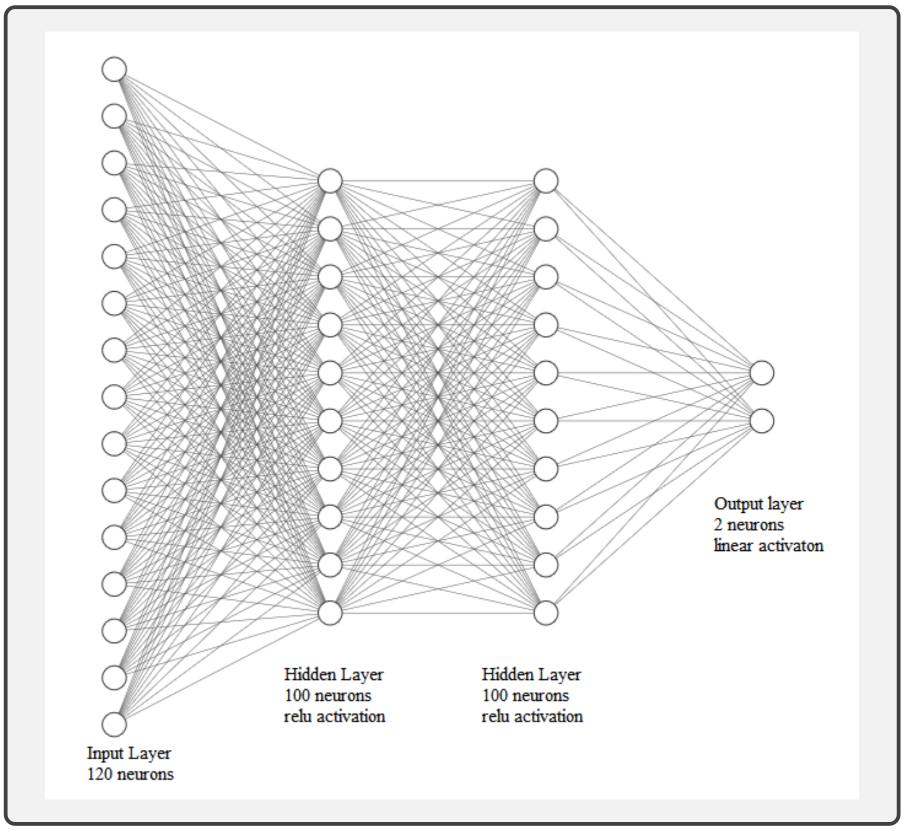


Fig. 3.3 An illustration of the neural network used to approximate the Q-function.

Input Layer	120 *neurons*
1st Hidden layer	100 *neurons* , ReLU activation
2nd Hidden layer	100 *neurons* , ReLU activation
Output layer	2 *neurons* , linear activation
Optimiser	Adam

Table 3.2 The hyper-parameters of the Q-learning's Q-function approximation neural network.

function must be defined. The loss function evaluates the performance of the neural network and tells it how good the current set of weights are, the lower the loss function the better the fit. Since when training the network to fit the Q-function the the actual Q-function is not known we use a function known as the *temporal difference* to calculate the loss the function. The temporal difference is the difference between the Q-value estimated by the network for a certain action from a certain state before taking the action and an improved estimate that uses the information obtained from the environment (the instantaneous reward for the previous state-action pair). When the temporal difference is zero the Q-function has been perfectly fitted, hence we used the temporal difference squared as the loss function. The exact loss function was as follows:

$$loss = (Q(s,a) - (r(s,a) + \gamma \max_{a'} Q(s',a')))^2 \tag{3.6}$$

where s is the previous state before taking an action, a is the action taken, $r(s,a)$ is the reward received, s' is the next state reached by the action taken and a' is selected from set of all possible actions from the next state. Psuedo-code for the Q-learning algorithm is shown in Algorithm 5.

Algorithm 5: Q-learning Algorithm

```
1 Initialise Q(s,a) arbitrarily;
2 repeat
3     Initialise s; repeat
4         Choose action $a$ from $s$ using policy derived from Q;
5         Take action $a$, observe $r, s'$;
6         $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$;
7         $s \leftarrow s'$;
8     until s is a terminal state;
9 until All episodes are done;
```

The temporal difference was calculated after each simulation time step then minimised. To minimise the loss we used the python machine learning library TensorFlow [55]. Specifically the *AdamOptimizer* was used to minimise the loss function. To diversify the the experience of the network to help it learn the optimal policy we used an ε-greedy policy during training, meaning on each time step a random action would be selected with probability ε and with probability $1-\varepsilon$ the action with the maximum Q-value would be selected. An initial value of of 0.4 was chosen for ε and it was decayed (multiplied) by a factor of 0.99 on each iteration. Each episode of training was on the map defined in the scenario in Section 3.1.2 but with traffic that was randomised before each episode with similar density and parameters as that of the traffic in Section 3.1. The exact traffic in Section 3.1.2 was reserved for testing and was not used during training. Training was carried out for 1500 iteration.

3.3.2 Policy Gradient

When using policy gradient to learn the optimal policy the policy is searched for directly without the need to infer it from the value function. To achieve this objective, we approximated the policy function using a neural network whose hyper-parameters are provided in

Table 3.3, an illustration of the network is shown in Figure 3.4. The flavor of policy gradient used in this work is known as reinforce policy gradient. Pseudo-code for this algorithm is shown in Algorithm 6

Input Layer	120 *neurons*
1st Hidden layer	100 *neurons* , ReLU activation
2nd Hidden layer	100 *neurons* , ReLU activation
Output layer	2 *neurons* , softmax activation
Optimiser	Adam

Table 3.3 The hyper-parameters of the policy gradient's policy approximation neural network.

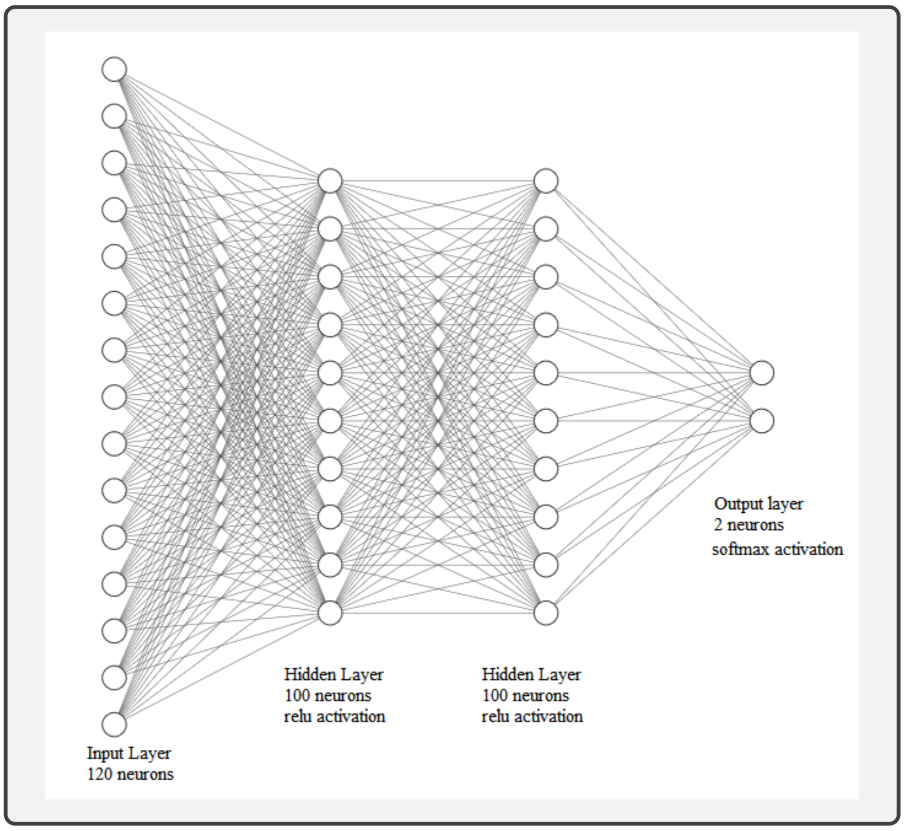


Fig. 3.4 An illustration of the neural network used to approximate policy.

Algorithm 6: Policy gradient Algorithm

```
1 Initialise the policy parameters $\theta$ arbitrarily
2 foreach episode do
3     foreach (s,a,r) do
4
5         $\theta \leftarrow \theta + \alpha \nabla log \pi_\theta(s,a)R$
```

The neural network parameters were randomly initialised. The network was fed with the same DTSE used in Q-learning. Each of the two outputs of the network express the probability of taking the corresponding action. We randomly choose a action based on these probabilities. It is clear that the policy network supplies a probability distribution for the actions, and we modify this distribution so that we increase the likelihood of the actions that produce high discounted reward being sampled in the future and eventually find the optimal policy. In practice to increase the likelihood of an action to be sampled, it is favourable to minimise the negative log likelihood of that action. To obtain the best values of the network parameters that result in the optimal policy function, we used the loss function

$$J(\theta) = E_{\pi\theta}[-log\pi_\theta(s,r,\theta)R] \tag{3.7}$$

Then we modified the network parameters, in the direction that minimises this loss. This can be expressed mathematically as

$$\Delta\theta = \alpha\nabla_\theta J(\theta) \tag{3.8}$$

The policy gradient approach we used in this experiment, known as reinforce policy gradient, is a Monte Carlo approach that runs in an episodic manner. On each episode, we store the states, the actions, and the rewards observed in the episode at each time step. We then calculated the loss of the episode and we carried out one optimisation step (parameter update) based upon the stored information using the policy gradient parameters update formula in Equation 3.8. The gradient of the function was calculated using the relationship

$$\nabla_\theta J(\theta) = E_{-\pi\theta}[\nabla(log\pi(s,a,\theta))R] \tag{3.9}$$

In the experiment the rewards were mean normalised before being used for training, and the training was run for 1500 episodes. To optimise the weights of the neural network the python machine learning library PyTorch was used. Again, the exact traffic in Section 3.1.2 was reserved for testing and was not used during training.

Chapter 4

Results and Discussion

4.1 Overview

This chapter presents the results of the experiments outlined in Chapter 3. The results of the proposed methods for pre-timed traffic light optimisation (GA, SR-IE, SR-SA) are presented in Section 4.2. They are compared against each other. The default timing for the traffic light as generated by SUMO is provided for reference. The results of the traffic responsive traffic light controllers are provided in Section 4.3. The results of the proposed RL based controllers are compared against each other as well as the classical LQF algorithm. Since any traffic responsive control method would most likely beat even the most sophisticated pre-timed method, due to the fact traffic responsive controllers have access to real time data that pre-timed controllers don't, no direct comparison has been made between the two different classes pre-timed and responsive

4.2 Pre-timed Traffic Control

The experiments regarding the pre-timed traffic controllers optimisation were conducted on an i7-4500U CPU with a memory of 8GB RAM, Table 4.1 shows the results. The algorithms were run for 10 runs each run had 1000 iteration. The table lists the average value, the standard deviation, and the best value for the four metrics. The best value is the best value scored to optimise the objective function only, namely the mean journey time.
It is very clear that the algorithms produced a solution that has by far achieved much better performance compared to the default simulator settings. To compare the results the famous Wilcoxon singed rank test was used with a confidence interval of 95%. The results are shown in table 4.2. To understand the table. Given two algorithms M_a, M_b, if $M_a > M_b$ this indicates

Table 4.1 Summary of intersection throughput for GA, SR-IE and SR-SA. Best values are highlighted in bold.

	Mean Journey Time			Mean Vehicle Speed			Mean Time Loss			Mean Vehicle Waiting Time		
Algorithm	Best	Avg.	Std.	Best	Avg.	Std.	Best	Avg.	Std.	Best	Avg.	Std.
SUMO Default	163.41	-	-	2.94	-	-	134.60	-	-	92.79	-	-
SR-IE	137.98	141.52	2.57	3.83	3.88	0.15	109.17	112.71	2.57	**75.36**	79.73	2.40
SR-SA	138.13	140.36	1.81	3.87	3.90	0.16	109.31	111.54	1.81	77.41	78.94	1.26
GA	**136.91**	**138.68**	1.22	**4.38**	**4.05**	0.25	**108.10**	**109.86**	1.21	77.50	**77.65**	1.12

Table 4.2 Pairwise performance comparison of SR-IE, SR-SA and GA.

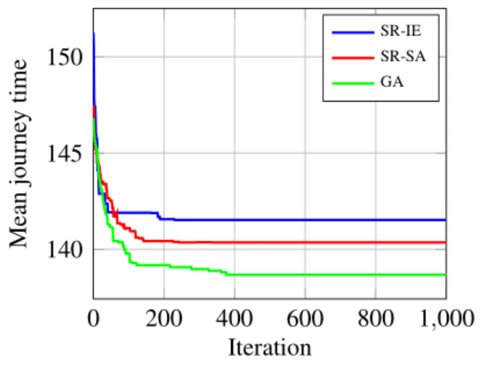
	SR-IE	SR-SA	GA
SR-IE	–	$\simeq$	$<$
SR-SA	$\simeq$	–	$\simeq$
GA	$>$	$\simeq$	–

that M_a outperformes M_b by a statistically significant margin. $M_a < M_b$ means that M_b outperforms M_a by a statistically significant margin. $M_a \simeq M_b$ indicates that the difference between the two algorithms isn't statistically significant. The GA was the top scorer, the difference between GA& SR-IE was statistically significant. But the difference between the GA and SR-SA wasn't. We hypothesise that the GA was the best due to the fact that the solutions are very sensitive to the initial conditions. Due to the nature of the GA, we have a population of solutions, this population allows more solutions to be perturbed and used and thus more regions on the search space are discovered leading to finding better solutions and escaping local minima. By same reasoning we can conclude that the probabilistic acceptance of worsening moves in SR-SA lead to escaping local minimas and hence producing better solutions than with SR-IE. Figure 4.1 shows the scores of the three algorithms on the traffic metrics.

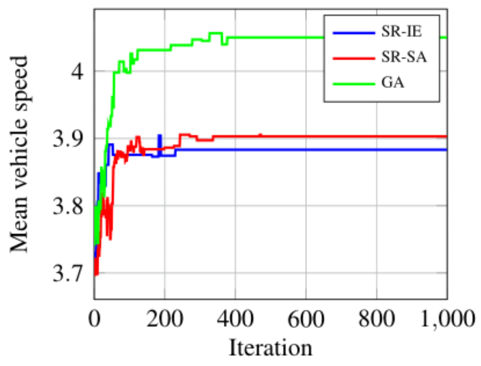
Regarding the fairness of the produced solutions, Table 4.3 shows the results. It is important to mention that the fairness is not an objective of the optimisation. It is clear that the difference between the algorithms isn't large. We can notice that the GA has produced a less fair solution and this is very reasonable because of the fact that the GA tend to optimise the mean journey time without giving any consideration for the fairness, and there is a trade-off between the fairness and the mean journey time.

Table 4.3 VSSD (Vehicle Speed Standard deviation) and JTSD (Journey Time Standard Deviation) for pre-timed traffic light optimization. Best values are highlighted in bold.

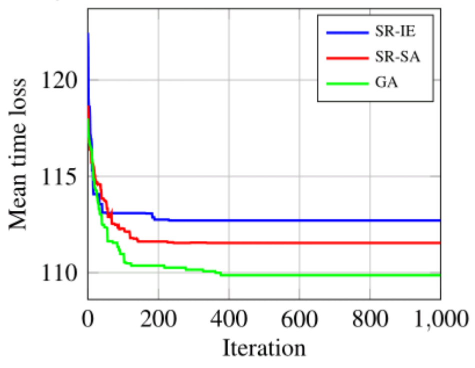
	VSSD		JTSD	
Algorithm	Best	Avg.	Best	Avg.
SUMO Default	**2.09**	-	81.04	-
SR-IE	2.32	2.66	**73.21**	75.41
SR-SA	2.62	**2.49**	77.27	**74.18**
GA	3.20	2.80	84.50	76.69



(a) Average of changes in the objective function value (mean journey time) (the lower the better).



(b) Average of changes in the mean vehicle speed (the higher the better).



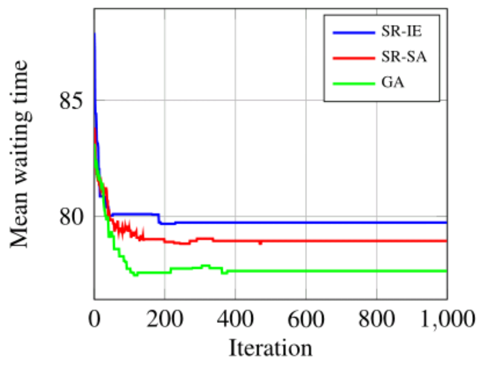


Fig. 4.1 The change in the throughput metrics during optimisation for the GA, SR-IE and SR-SA, averaged over the 10 runs

4.2.1 The performance of the Low Level Heuristics

Figure 4.2 shows the percentage of improvement introduced by each of the low level heuristics. In Both SR-IE and SR-SA. LLH1 had the biggest contribution to the improvement of both SR-IE and SR-SA. In SR-IE LLH2 came the second, and LLH3 came the third. In SR-SA LLH3 came the second and LLH2 came the third. LLH4 came the last in both algorithms.

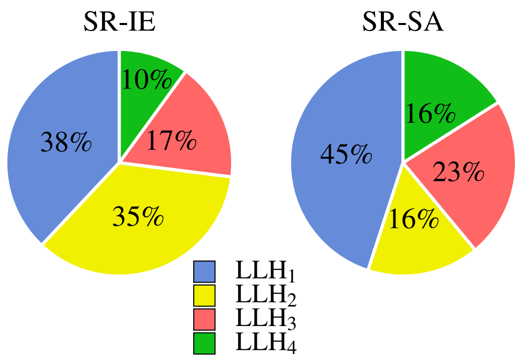


Fig. 4.2 Percentage of improving moves causes by each low-level heuristic.

4.3 Traffic adaptive control

The results of the Q-learning algorithm and the policy gradient are presented shown in Table 4.4. For the training of the agents, random traffic was used, it was generated completely randomly on every episode the training repeated for 1500 episode. The agent was tested on the traffic in scenario 3.1. The top scorer for the adaptive controllers for all the metrics was the policy gradient agent with $\beta = 0.5$. The agent has outperformed the LQF agent by 21% regarding the mean journey time which is the optimisation objective.The second best scorer was the policy gradient with $\beta = 0.0$, however the second highest mean vehicle speed was achieved by the LQF. The Q-learning with $\beta = 0.5$ came the third, and at last, the Q-learning with $\beta = 0.0$. Figs 4.3-4.8 show the progress of the four algorithms during training.
The training of the agents has been very noisy, this is expected because of the use of function approximation for both the Q-learning function and for the policy function, also the Monte Carlo nature of the policy gradient has introduced more nosiness to the performance. We have used an averaging filter of window size = 20 to smooth out the curves, figure 4.3 shows the results of change in mean vehicle speed of an RL agent during training without filtering. It is very clear that the policy gradient agent had reached a high performance from the very first episodes, and then the increase in performance processed slowly. We hypothesise that this is due to the fact that policy gradient algorithms starts by a random policy that favours no policy and gives a 50-50 chance for both directions this might be near optimal. Also we think that the first episode contained enough cases for the policy network to estimate a good policy directly, unlike in Q-learning where a lot of training is needed to first fit a proper q-network. The Q-learning agent took about 300 episodes to reach the peak in performance.

Table 4.4 Throughput metrics for traffic responsive control. Best values are highlighted in bold.

Algorithm	Mean Journey Time	Mean Vehicle Speed	Mean Time Loss	Mean Waiting Time
LQF	57.30	8.36	28.52	5.43
Q-Learning $\beta = 0$	81.73	6.99	52.96	12.88
Q-Learning $\beta = 0.5$	51.54	8.14	22.73	2.73
Policy Gradient $\beta = 0$	**44.85**	**9.24**	**16.11**	**2.09**
Policy Gradient $\beta = 0.5$	50.62	8.30	21.82	2.90

Table 4.5 VSSD (Vehicle Speed Standard deviation) and JTSD (Journey Time Standard Deviation) for traffic responsive controllers. Best values are highlighted in bold.

Algorithm	VSSD	JTSD
LQF	2.80	34.27
Q-Learning $\beta = 0$	3.45	60.15
Q-Learning $\beta = 0.5$	1.97	13.88
Policy Gradient $\beta = 0$	**1.93**	**11.78**
Policy Gradient $\beta = 0.5$	2.00	13.89

Fig. 4.3 Changes in mean vehicle speed during training the RL models without filtering

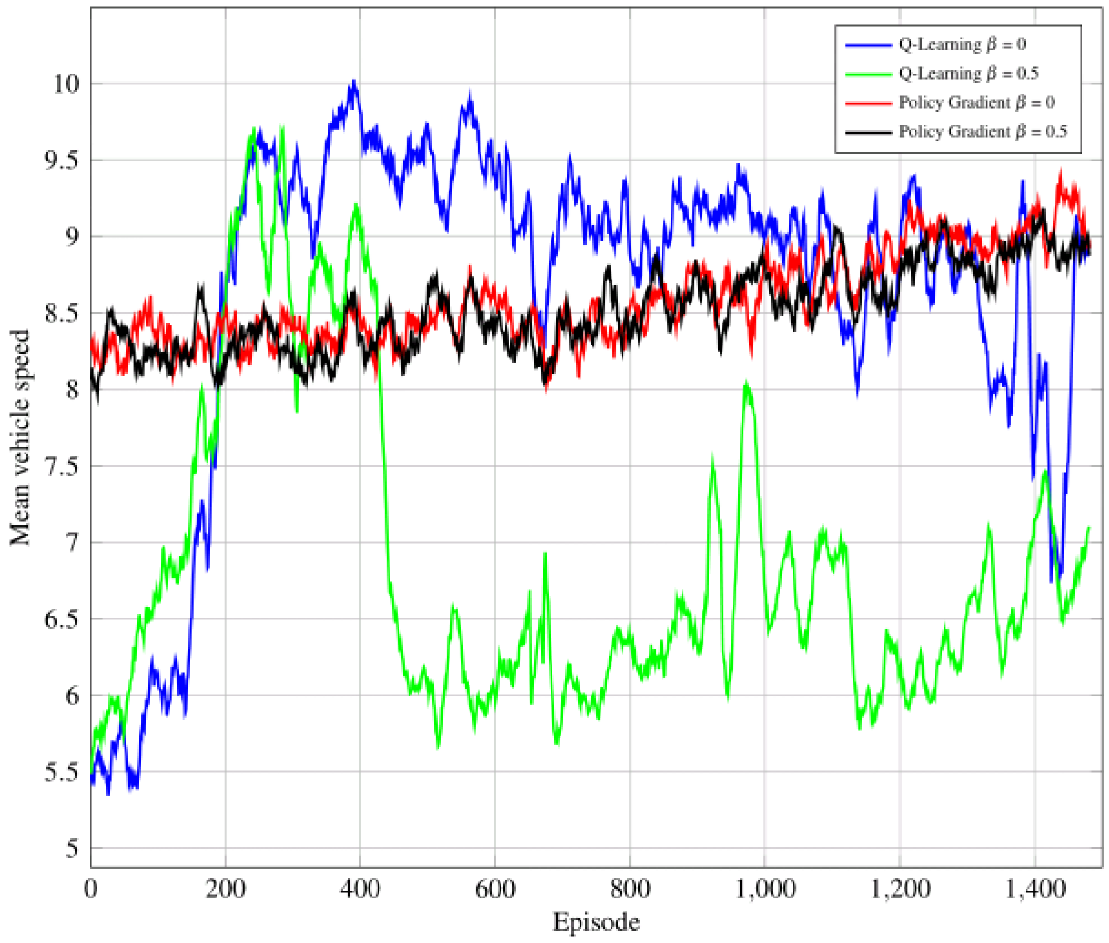


Fig. 4.4 Changes in mean vehicle speed during training the RL models.

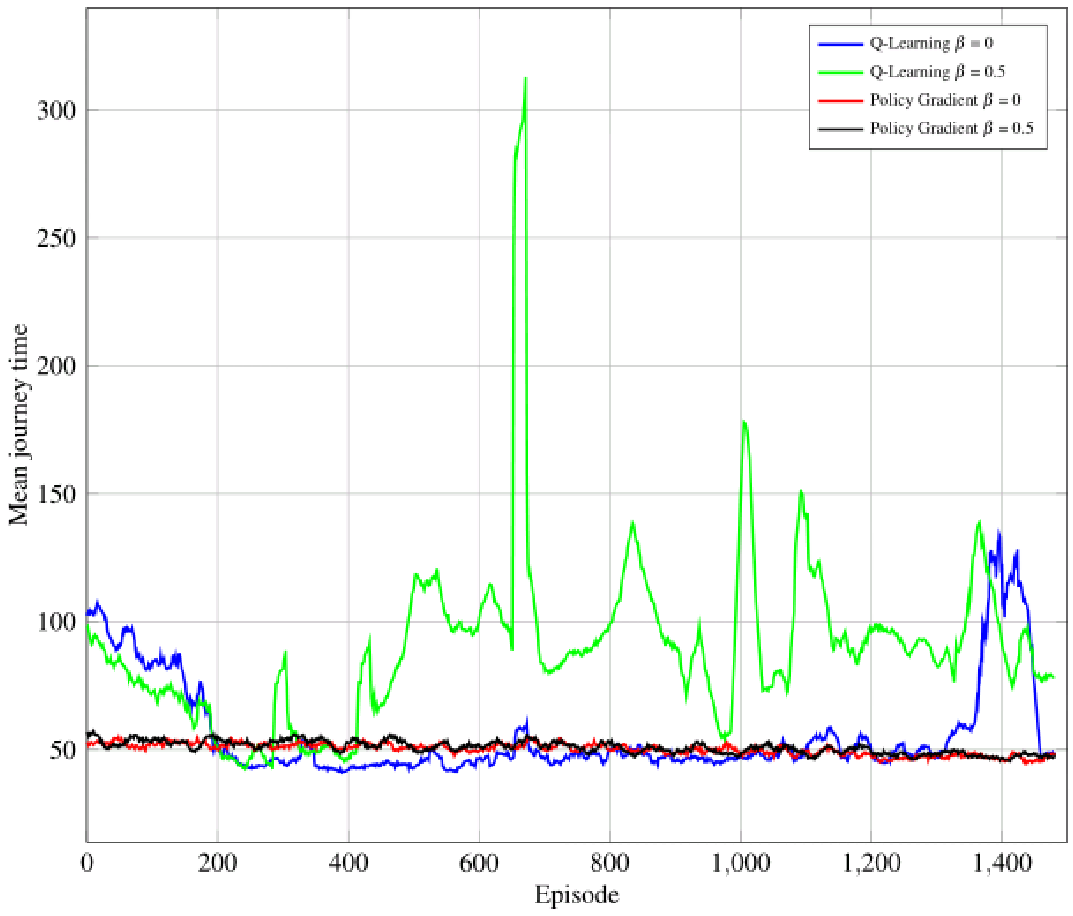


Fig. 4.5 Changes in mean journey time during training RL models.

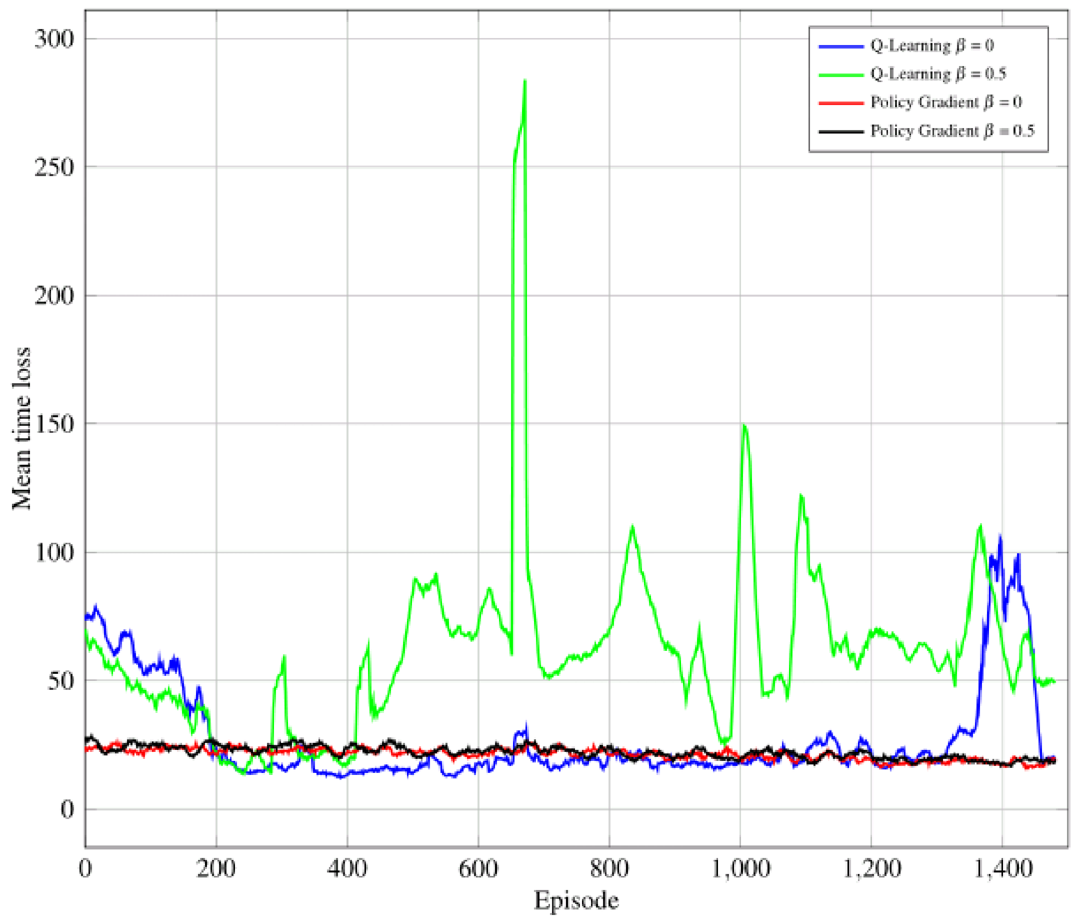


Fig. 4.6 Changes in mean time loss during training the RL models.

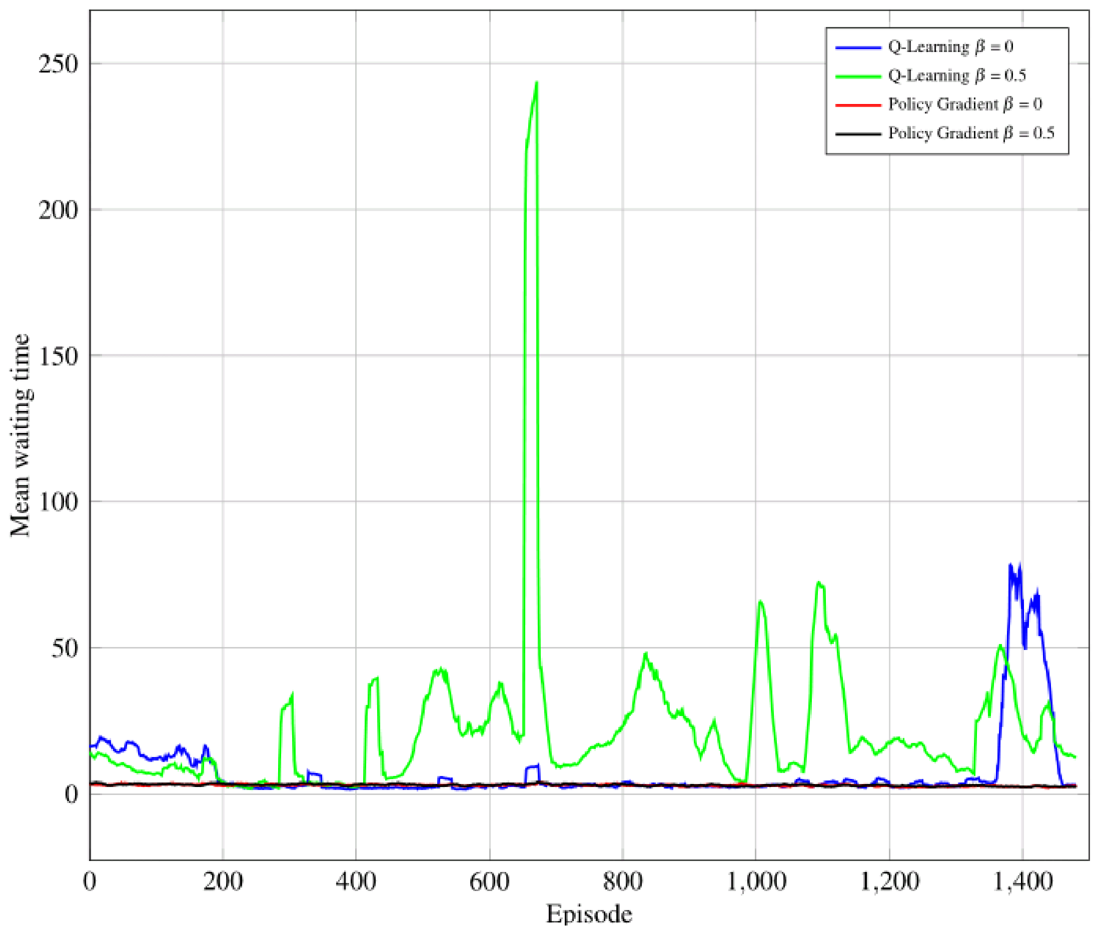


Fig. 4.7 Changes in mean waiting time during training the RL models.

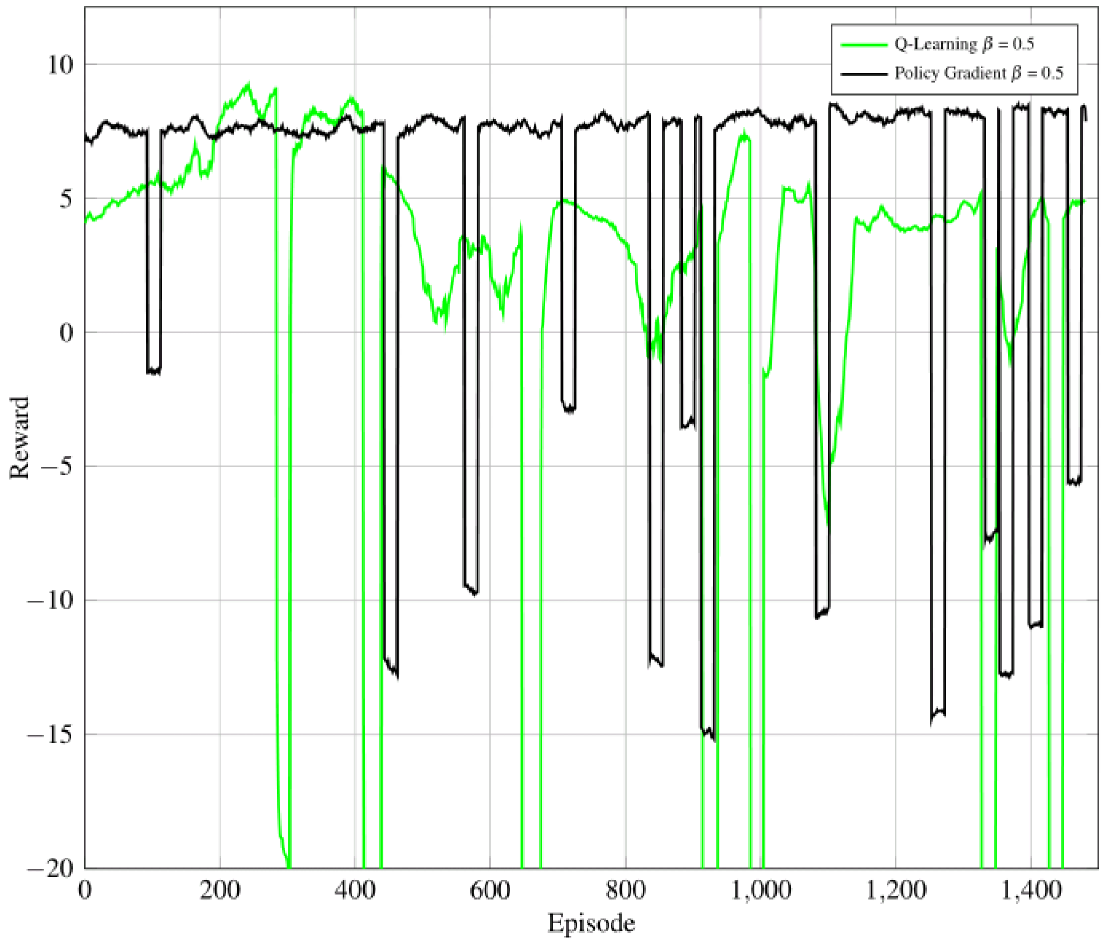


Fig. 4.8 Changes in reward for while training the RL models with $\beta ! = 0$

Chapter 5

Conclusion

5.1 Summary of Work

In our project, we have studied and empirically analysed the use of computational intelligence for traffic light control. We have proposed three solutions for the static traffic light optimisation problem where traffic lights are pre-timed in a single intersection network. Genetic algorithm, simple random improve or equal hyper-heuristic, and simple random with simulated annealing hyper-heuristic. Genetic algorithms has remarkably outperformed the simple random improve or equal hyper-heuristic, also the genetic algorithm has outperformed the simple random improve or equal hyper-heuristic by a statistically insignificant margin. In this problem we have used the mean journey time of the vehicles as the objective function that we intend to optimise. Other metrics to compare the performance of the algorithms have been used, such as: mean vehicles speed, mean waiting time and mean time loss. In our work, we have introduced two measure for the fairness of the flow namely, vehicle speed standard deviation (VSSD) and journey time standard deviation (JTSD). Our work wasn't limited to optimising pre-timed traffic lights, we have proposed reinforcement learning solutions to optimise the traffic lights in an adaptive control scheme. We have used the Q-learning algorithms and the Reinforce Policy Gradient algorithm to optimise the traffic flow in a single intersection network to adaptively and dynamically adjust the timings of the traffic lights to satisfy to our objective function which in this case was the mean journey time in one problem instance, and a combination between the mean journey time the journey time vehicle speed in another problem instance. The algorithms were benchmarked against the well know Longest Queue First algorithm. In both the Q-learning and Policy gradient, a neural network was used to approximate the Q-function and the policy function. The problem was modelled as a Markov Decision Process problem where the states space and the action space are discrete. The states are the discrete encoding of the locations and the relative velocities of the vehicles.

The actions are either to allow the vehicles in the North-South direction to move, or the vehicles in the East-West direction to move. We had two different reward signals, the direct reward was the vehicles speed and the fairness adjust reward which accounts for the fairness of the traffic.
The results shows a significant improvement of the objective function, the policy gradient was the top scorrer followed by the Q-learning algorithm, the policy gradient algorithm has a high initial score on the objective function, we think this is due to the fact that the policy gradient starts by a random policy and improves this policy. This policy gives a 50-50 chance for the two direction, and this is a good initial condition for the traffic control using the given traffic properties. The detailed results are in chapter 4.
All our evaluation and testing was performed using the simulation of urban mobility (SUMO) simulator, using a randomly generated traffic.

5.2 Future Work

The future work will include the following tasks:

- Using different reward function that are going to include different information and study the effects of this usage.
- Studying how tuning the hyper-parameters might change the performance of the proposed solutions.
- Using genetic algorithms to find the optimal neural network architectures for both the policy network and the Q-function network.
- Using the other policy gradient based techniques like Actor-critic and natural policy gradient.
- Applying the already trained model in a multi-intersection network where each intersection is controlled by a single agent.
- Training the agents in a multi-intersection network and comparing the results with the already trained agents.
- Training the agents in a multi-intersection network where the agents can exchange information that encode their states and history between each other, and comparing the results with the other multi-intersection optimisation schemes.

- Using recurrent neural networks instead of feed-forward neural networks because they're more suitable for dealing with the tasks where temporal changes occur.

Appendix A

Code Listings

Code for Interfacing the Simulator

simulation.py

```
import traci
import time
import pickle
import abc

class Simulator:
    def __init__(self):
        self._tickables = []
        self._post_run_funcs = []
        self._sim_components = []
        self._tick_freq = {}
        self._sim_step = 0
        self.results = {}
        self.output_file = "tripinfo.xml"
    def tick(self):
        traci.simulationStep()
        for tickable in self._tickables:
            if self._sim_step % self._tick_freq[tickable] == 0:
                tickable.tick()
        for component in self._sim_components:
            if self._sim_step % self._tick_freq[component] == 0:
                component.tick()
        self._sim_step += 1

    def run(self, path_to_cfg, time_steps="run_to_completion" ,gui=False):
        start_time = time.time()
        if gui:
            sumoBinary = "sumo-gui"
        else:
            sumoBinary = "sumo"
        sumoCmd = [sumoBinary, "-c"]
        sumoCmd.append(path_to_cfg)
        sumoCmd.append("--tripinfo-output")
        sumoCmd.append(self.output_file)


        traci.start(sumoCmd)
        if time_steps == "run_to_completion":
            while traci.simulation.getMinExpectedNumber() > 0 :
                self.tick()
        else:
            while self._sim_step < time_steps and traci.simulation.
                ↪ getMinExpectedNumber() > 0:
                self.tick()

        traci.close()
        for func in self._post_run_funcs:
            func()
        for component in self._sim_components:
            component.post_run()
        print("Runtime: %.3f"%(time.time()-start_time))
        if self._sim_step == time_steps:
            return True
        return False

    def add_tickable(self, tickable, freq=1):
        self._tickables.append(tickable)
        self._tick_freq[tickable] = freq

    def add_postrun(self, func):
        self._post_run_funcs.append(func)
    def add_simulation_component(self, Component, freq=1, *args, **kwargs):
        component = Component(self, *args, **kwargs)
        self._sim_components.append(component)
        self._tick_freq[component] = freq

    def save_results(self, filename):
        file_io = open(filename + ".pkl", 'wb')
        results = pickle.dump(self.results, file_io)
        file_io.close()

class Tickable:
    @abc.abstractmethod
    def tick(self):
        pass


class SimulationComponent(Tickable):
    @abc.abstractmethod
    def post_run(self):
        pass
```

stats/output_parser.py

```
from simulation import SimulationComponent
from xml.dom import minidom
import numpy as np

class SimulationOutputParser (SimulationComponent):
    def __init__(self, simulation):
        self.mean_journey_time =0
        self._simulation = simulation

    def tick(self):
        pass

    def post_run(self):
        results = {"mean_speed":[], "duration":[], "waiting_time":[], "
            ↪ time_loss":[]}
        xmldoc = minidom.parse(self._simulation.output_file)
        tripinfo_list = xmldoc.getElementsByTagName("tripinfo")
        for entry in tripinfo_list:
            duration = float(entry.getAttribute("duration"))
            mean_speed = float(entry.getAttribute("routeLength"))/duration
            results["mean_speed"].append(mean_speed)
            results["duration"].append(duration)
            results["waiting_time"].append(float(entry.getAttribute("waitSteps
                ↪ ")))
            results["time_loss"].append(float(entry.getAttribute("timeLoss")))

        for category, values in results.items():
            self._simulation.results [category] = np.array(values)
```

action.py

```
import traci


class PhaseModifier:
    def __init__(self, junctionID):
        self._junction_id = junctionID

    def set_phase(self,phase):
        traci.trafficlights.setPhase(self._junction_id, phase)
```

Code for Pre-timed light optimization

hyperheuristics/static_controller.py

```
class StaticTrafficLightController:
    def __init__(self, controller, sequence, timings):
        self._controller = controller
        assert (len(sequence) == len(timings))
        self._n_phases = len(sequence)
        self._phase_sequence = sequence
        self._timings = timings
        self._time_elapsed = 0
        self._phase_number = 0

    def tick(self):
        self._time_elapsed += 1
        self._controller.set_phase(self._phase_sequence[self._phase_number])
        if self._time_elapsed == self._timings[self._phase_number]:
            self._phase_number += 1
            self._phase_number %= self._n_phases
            self._time_elapsed = 0
```

SR-IE

hyperheuristics/random_descent.py (Runnable File)

```
from simulation import Simulator
from stats import SimulationOutputParser
from action import PhaseModifier
from .static_controller import StaticTrafficLightController
import os
import random
import pandas as pd
import pickle


sumocfg1 = "..\\..\\test_environments\\single_intersection_random_trips\\
    ↪ newnet.sumocfg"
sumocfg2 = "..\\..\\test_environments\\grid_map\\4by4.sumocfg"


def define_data_frame():
    simulation_dataFrame = pd.DataFrame({'iteration': [0],
                                         'mean_speed': [0],
                                         'duration': [0],
                                         'waiting_time': [0],
                                         'time_loss': [0]})
    simulation_dataFrame.set_index('iteration', inplace=True)
    return simulation_dataFrame


def generate_iteration_data_frame(iteration_no,mean_speed_result,
    ↪ duration_result,waiting_time,time_loss):
    iteration_dataFrame = pd.DataFrame(({'iteration': [iteration_no],
                                         'mean_speed': [mean_speed_result],
                                         'duration': [duration_result],
                                         'waiting_time': [waiting_time],
                                         'time_loss': [time_loss]}))
    iteration_dataFrame.set_index('iteration',inplace= True)
    return iteration_dataFrame


def concat_frames(f1,f2):
    frames = [f1, f2]
    frame = pd.concat(frames)
    return frame


def save_dataframe2CSV(f1,index,file):
    #f1.set_index(index, inplace=True)
    f1.to_csv(file)


def initial_timings():
    return random.sample(range(50,100), 2)


def evaluate_timing(timing):
    traffic_light = PhaseModifier("node1")
    full_timings = [3]*8
    full_timings[0] = timing[0]
    full_timings[4] = timing[1]
    controller = StaticTrafficLightController(controller=traffic_light,
        ↪ sequence=list(range(8)), timings=full_timings)
    sim = Simulator()
    sim.add_simulation_component(SimulationOutputParser)
    sim.add_tickable(controller)
    if not sim.run(sumocfg1, time_steps=2000, gui=False):
        return sim.results
    return False

def save_timing_performance(timing, filename):
    full_timings = [3]*8
    full_timings[0] = timing[0]
    full_timings[4] = timing[1]
    traffic_light = PhaseModifier("node1")
    controller = StaticTrafficLightController(controller=traffic_light,
        ↪ sequence=list(range(8)), timings=full_timings)
    sim = Simulator()
    sim.add_simulation_component(SimulationOutputParser)
    sim.add_tickable(controller)
    sim.run(sumocfg1, gui=False)
    sim.save_results(filename)

def OI(old_objective, new_objective):
    return new_objective < old_objective


def IE(old_objective, new_objective):
    return new_objective <= old_objective

def llh(h, timing):
    if h==0:
        return mutate_timing(timing,5)
    elif h==1:
        return mutate_timings2(timing,2)
    elif h==2:
        return mutate_timings3(timing,1)
    elif h==3:
        return mutate_timings4(timing)

def mutate_timing(timings, magnitude):
    n = len(timings)
    timing_to_modify = random.randrange(0,n)
    result = timings[:]
    result[timing_to_modify] += random.randrange(-magnitude, magnitude+1)
    result[timing_to_modify] = max(0, result[timing_to_modify])
    return result


def mutate_timings2(timings, magnitude):
    n = len(timings)
    result = timings[:]
    timing_to_modify = random.randrange(0, n)
    result[timing_to_modify] += random.choice([-10, 10])
    result[timing_to_modify] = max(0, result[timing_to_modify])
    return result


def mutate_timings3(timings, magnitude):
    #select at random n cells and and mutate them
    n = len(timings)
    number_of_indices_to_perturb = random.randrange(n)
    indices_to_perturb = random.sample(range(n), number_of_indices_to_perturb
        ↪ )
    new_timings = timings[:]
    for i in indices_to_perturb:
        new_timings[i] += random.randrange(-magnitude, magnitude +1)
        new_timings[i] = max(0, new_timings[i])
    return new_timings


def mutate_timings4(timings):
    n = len(timings)
    new_timings = timings[:]
    a,b = random.sample(range(n), 2)
    new_timings[a], new_timings[b] = new_timings[b], new_timings[a]
    return new_timings

def run_10_rand_desc():
    results_dir = os.path.dirname(__file__)
    results_dir = os.path.dirname(results_dir)
    results_dir = os.path.join(results_dir, "results")
    results_dir = os.path.join(results_dir, "random_descent")
    for run in range(10):
```

```
results = False
while not results:
    print("generating initial timings")
    current_timing = initial_timings()
    results = evaluate_timing(current_timing)
previous_objective = results["duration"].mean()
metrics = define_data_frame()
improved = {0: 0, 1: 0, 2: 0, 3: 0}
called = {0: 0, 1: 0, 2: 0, 3: 0}
for i in range(1000):
    new_results = False
    while not new_results:
        h = random.randrange(4)
        new_timings = llh(h, current_timing)
        new_results = evaluate_timing(new_timings)

    objective = new_results["duration"].mean()
    called[h] += 1
    if OI(previous_objective, objective):
        improved[h] += 1
        current_timing = new_timings
        previous_objective = objective
        results = new_results
    x = generate_iteration_data_frame(i, results["mean_speed"].mean(),
        ↪  results["duration"].mean(),
                                      results["waiting_time"].mean(), results
                                         ↪ ["time_loss"].mean())
    metrics = concat_frames(metrics, x)

heuristic_report = {"called": called, "improved": improved}
save_dataframe2CSV(metrics, "iteration",os.path.join(results_dir,"
    ↪ rd_runtime" + str(run) + ".csv"))
save_timing_performance(current_timing, os.path.join(results_dir,"
    ↪ rd_final_iteration" + str(run)))
file_io = open(os.path.join(results_dir,r"rd_heuristic_report" + str(
    ↪ run) + ".pkl"), 'wb')
pickle.dump(heuristic_report, file_io)
file_io.close()
```

```
        file_io = open(os.path.join(results_dir,"rd_final_iteration_timings" +
            ↪  str(run) + ".txt"), "w")
        file_io.write(str(current_timing))
        file_io.close()

if __name__== "__main__":
    run_10_rand_desc()
```

SR-SA

hyperheuristics/simulated_annealing.py (Runnable File)

```
from simulation import Simulator
from stats import SimulationOutputParser
from action import PhaseModifier
from static_controller import StaticTrafficLightController
import matplotlib.pyplot as plt
import os
import random
import pandas as pd
import pickle


from hyperheuristics.random_descent import define_data_frame
from hyperheuristics.random_descent import generate_iteration_data_frame,
    ↪ concat_frames, save_dataframe2CSV
from hyperheuristics.random_descent import initial_timings, evaluate_timing,
    ↪ save_timing_performance, OI, llh
import random
import math

sumocfg1 = "..\\..\\test_environments\\single_intersection_random_trips\\
    ↪ newnet.sumocfg"
sumocfg2 = "..\\..\\test_environments\\grid_map\\4by4.sumocfg"


def simulated_annealing_accept(previous_objective, objective, temperature):
    if objective < previous_objective:
        return True
    if temperature < 0.00000001:
        return False
    accept_worse = math.exp((previous_objective - objective) / temperature) >
        ↪  random.random()
    return accept_worse



def run_10_sim_anneal():

    results_dir = os.path.dirname(__file__)
    results_dir = os.path.dirname(results_dir)
    results_dir = os.path.join(results_dir, "results")
    results_dir = os.path.join(results_dir, "simulated_annealing")
    for run in range(10):
        results = False
        T = 1
        alpha = 0.99
        while not results:
            print("generating initial timings")
            current_timing = initial_timings()
            results = evaluate_timing(current_timing)
        previous_objective = results["duration"].mean()
        metrics = define_data_frame()
        accepted = {0: 0, 1: 0, 2: 0, 3: 0}
        called = {0: 0, 1: 0, 2: 0, 3: 0}
        improved = {0: 0, 1: 0, 2: 0, 3: 0}
        improved_best_of_run = {0: 0, 1: 0, 2: 0, 3: 0}
        best_of_run_objective = 100000000
        for i in range(1000):
            new_results = False
            while not new_results:
                h = random.randrange(4)
                new_timings = llh(h, current_timing)
                new_results = evaluate_timing(new_timings)

            objective = new_results["duration"].mean()
            called[h] += 1
            if simulated_annealing_accept(previous_objective, objective, T):
                if objective < previous_objective:
                    improved[h] +=1
                if objective < best_of_run_objective:
                    improved_best_of_run[h] +=1
                    best_of_run_objective = objective
                accepted[h] += 1
                current_timing = new_timings
                previous_objective = objective
                results = new_results
            x = generate_iteration_data_frame(i, results["mean_speed"].mean(),
                ↪  results["duration"].mean(),
                                     results["waiting_time"].mean(), results
                                         ↪ ["time_loss"].mean())
            metrics = concat_frames(metrics, x)
            T *= alpha
        heuristic_report = {"called": called, "accepted": accepted, "improved"
            ↪ :improved, "improved_best_of_run":improved_best_of_run}
        save_dataframe2CSV(metrics, "iteration",os.path.join(results_dir,"
            ↪ rd_runtime" + str(run) + ".csv"))
        save_timing_performance(current_timing, os.path.join(results_dir,"
            ↪ rd_final_iteration" + str(run)))
        file_io = open(os.path.join(results_dir,r"rd_heuristic_report" + str(
            ↪ run) + ".pkl"), 'wb')
        pickle.dump(heuristic_report, file_io)
        file_io.close()
        file_io = open(os.path.join(results_dir,"rd_final_iteration_timings" +
            ↪  str(run) + ".txt"), "w")
        file_io.write(str(current_timing))
        file_io.close()

if __name__ == "__main__":
    run_10_sim_anneal()
```

GA

genetic_algorithms/running_GA.py (Runnable File)

```
import random
import time

import numpy as np
import pandas as pd
from action import PhaseModifier
from genetic_algorithms.genetic_tools import Chromosome
from genetic_algorithms.genetic_tools import GAOpertations
from genetic_algorithms.tripinfo_extract import XMLDataExtractor
from hyperheuristics.static_controller import StaticTrafficLightController
from simulation import Simulator
from stats.output_parser import SimulationOutputParser

sumocfg1 = "..\\..\\test_environments\\single_intersection_random_trips\\
    ↪ newnet.sumocfg"
path = "tripinfo.xml"
fitness_list = []
timings_list_ = []


def define_data_frame():
    simulation_dataFrame = pd.DataFrame({'iteration': [0],
                                         'mean_speed': [0],
                                         'duration': [0],
                                         'waiting_time': [0],
                                         'time_loss': [0]})
    simulation_dataFrame.set_index('iteration', inplace=True)
    return simulation_dataFrame


def generate_iteration_data_frame(iteration_no,mean_speed_result,
    ↪ duration_result,waiting_time,time_loss):
    iteration_dataFrame = pd.DataFrame(({'iteration': [iteration_no],
                                         'mean_speed': [mean_speed_result],
                                         'duration': [duration_result],
                                     'waiting_time': [waiting_time],
                                     'time_loss': [time_loss]}))
    iteration_dataFrame.set_index('iteration',inplace= True)
    return iteration_dataFrame


def concat_frames(f1,f2):
    frames = [f1, f2]
    frame = pd.concat(frames)
    return frame


def save_dataframe2CSV(f1,file):
    f1.to_csv(file)

#
    ↪ ////////////////////////////////////////////////////////////////////////////////////
    ↪



for iteration in range (10):

    # ///////////////////////// initializing the population
        ↪ ///////////////////////////////////


    simulation_dataFrame = define_data_frame()

    time1 = time.time()
    timing_list = []

    for _ in range(5):
        timing_list.append(random.sample(range(50, 100), 2))

    population = []

     # initializing the chromosomes #
    for timing in timing_list:
```

```
    print(timing)
    chromosome = Chromosome(timing, fitness= 0)
    configuration = []
    for ii in range (8):
        configuration.append(3)
        if ii == 0:
            configuration[ii] = timing [0]
        if ii == 4:
            configuration[ii] = timing [1]
    chromosome_controller = StaticTrafficLightController(PhaseModifier("
        ↪ node1"),list(range(0,8)),configuration)
    print(chromosome._phases_steps)
    print(configuration)

    sim = Simulator()
    sim.add_tickable(chromosome_controller)
    sim.run(sumocfg1, gui=False)

    fitness = XMLDataExtractor(path).get_data()
    chromosome.set_fitness(fitness)
    population.append(chromosome)

for member in population:
    print(member._fitness)


best_initial_solution = min(population, key=lambda x: x._fitness)

configuration = []
for ii in range(8):
    configuration.append(3)
    if ii == 0:
        configuration[ii] = best_initial_solution._phases_steps[0]
    if ii == 4:
        configuration[ii] = best_initial_solution._phases_steps[1]

best_initial_solution_controller = StaticTrafficLightController(
    ↪ PhaseModifier("node1"), list(range(0,8)),configuration)
```

```
sim = Simulator()
sim.add_simulation_component(SimulationOutputParser)
sim.add_tickable(best_initial_solution_controller)
sim.run(sumocfg1, gui=False)

fitness = XMLDataExtractor(path).get_data()
best_initial_solution.set_fitness(fitness)
print("fitness of the best is: " ,fitness)

mean_speed_result = (np.mean(sim.results['mean_speed']))
duration_result = (np.mean(sim.results['duration']))
waiting_time = (np.mean(sim.results['waiting_time']))
time_loss = (np.mean(sim.results['time_loss']))
iteration_dataFrame = generate_iteration_data_frame(0, mean_speed_result,
    ↪  duration_result, waiting_time, time_loss)

simulation_dataFrame = concat_frames(simulation_dataFrame,
    ↪ iteration_dataFrame)


# ///////////////////////////// carrying out GA operations
    ↪ /////////////////////////////

print("*="*15)
print("performing genetic algorithm ....")

i = 0
j = 0

while i < 1000:

    print("iteration: ", i)
    ga_operator = GAOpertations()

    chromosome_1, chromosome_2 = ga_operator.select_simply(population)

    # crossover #

    offspring = ga_operator.point_corssover(chromosome_1,chromosome_2)
```

```
# mutation on the offspring #

mutated_offspring = ga_operator.mutate(offspring)

# determining offspring's fitness #
configuration = []


for ii in range(8):
    configuration.append(3)
    if ii == 0:
        configuration[ii] = mutated_offspring._phases_steps[0]
    if ii == 4:
        configuration[ii] = mutated_offspring._phases_steps[1]


offspring_chromosome_controller = StaticTrafficLightController(
    ↪ PhaseModifier("node1"), list(range(0, 8)),configuration)
sim = Simulator()
sim.add_simulation_component(SimulationOutputParser)
sim.add_tickable(offspring_chromosome_controller)

sim.run(sumocfg1, gui=False)

fitness = XMLDataExtractor(path).get_data()
mutated_offspring.set_fitness(fitness)

print("the offspring's fitness is: ", mutated_offspring._fitness)

# survival of the fittest #

best_solution = min(population, key=lambda x: x._fitness)

for chromosome in population:
    if chromosome._fitness > mutated_offspring._fitness:
        population.remove(max(population, key=lambda x: x._fitness))
        population.append(mutated_offspring)
        break
        if mutated_offspring._fitness < best_solution._fitness:

            mean_speed_result = (np.mean(sim.results['mean_speed']))
            duration_result = (np.mean(sim.results['duration']))
            waiting_time = (np.mean(sim.results['waiting_time']))
            time_loss = (np.mean(sim.results['time_loss']))

            j+=1
            iteration_dataFrame = generate_iteration_data_frame(j,
                ↪ mean_speed_result, duration_result, waiting_time,
                                                                    time_loss)

            simulation_dataFrame = concat_frames(simulation_dataFrame,
                ↪ iteration_dataFrame)

        else:
            j+=1
            simulation_dataFrame.tail(1)['iteration']=j
            simulation_dataFrame = concat_frames(simulation_dataFrame,
                ↪ simulation_dataFrame.tail(1))


        best_solution = min(population, key=lambda x: x._fitness)
        fitness_list.append(best_solution._fitness)
        timings_list_.append(best_solution._phases_steps)

        i += 1

    print("The fitness list is: ",fitness_list)
    timings_array = np.asarray(timings_list_)
    np.savetxt(str("timings_list"+str(iteration)+".csv"),timings_array,
        ↪ delimiter=",")

    time2 = time.time()
    print("="*10)

    print("iteration were performed in: ",time2-time1," seconds.")
    sim.save_results("ga_results"+str(iteration))
    save_dataframe2CSV(simulation_dataFrame,"ga_results"+str(iteration)+".csv
        ↪ ")

    print(simulation_dataFrame.head())
```

genetic_algorithms/genetic_tools.py

```
import random


class Chromosome(object):
    def __init__(self, phases_steps,fitness):
        self._phases_steps = phases_steps
        self._fitness = fitness

    def set_fitness(self, fitness):
        self._fitness = fitness

    def get_data(self):
        return self._phases_steps,self._fitness


class GAOpertations(object):
    def __init__(self):
        pass

    def select_ranked(self,population):
        population.sort(key=lambda x: x._fitness, reverse=True)
        return population[0],population[1]
        pass

    def select_simply(self,population):
        rand1 = random.randint(0, len(population) - 1)
        rand2 = random.randint(0, len(population) - 1)
        return population[rand1], population[rand2]
```

```
def point_corssover(self,first_chromosome,second_chromosome):

    portion = random.randint(0,len(first_chromosome._phases_steps))
    first_part = first_chromosome._phases_steps[:portion]
    second_part = second_chromosome._phases_steps[portion:]

    return Chromosome(first_part + second_part,fitness=0)


def uniform_corssover(self,first_chromosome,second_chromosome):

    phase_steps = []
    for i in range(len(first_chromosome._phases_steps)):
        rand = bool(random.randint(0,1))
        if(rand):
            phase_steps.append(first_chromosome._phases_steps[i])
        else:
            phase_steps.append(second_chromosome._phases_steps[i])
    return Chromosome(phase_steps,fitness=0)


def mutate(self,chromosome):
    mutated_chromosome = chromosome._phases_steps

    mutation_position = random.randint(0,len(mutated_chromosome)-1)
    plus_minus = bool(random.randint(0,1))

    if(plus_minus):
        mutated_chromosome[mutation_position] += random.randint(0,15)
    else:
        if mutated_chromosome[mutation_position] >10 :
            mutated_chromosome[mutation_position] -= random.randint(0,10)
        elif mutated_chromosome[mutation_position] > 5 and
            ↪ mutated_chromosome[mutation_position]< 10 :
            mutated_chromosome[mutation_position] -= random.randint(0, 5)
        else:
            mutated_chromosome[mutation_position] = 0

    return Chromosome(mutated_chromosome,fitness = 0)
```

genetic_algorithms/tripinfo_extract.py

```
from xml.dom import minidom

class XMLDataExtractor(object):

    def __init__(self,path_to_file):
        self.meanJtime = 0
        self.file = path_to_file

    def get_data(self):
        try:
            xmldoc = minidom.parse(self.file)
            tripinfo_list = xmldoc.getElementsByTagName("tripinfo")
            accumlated_JTime =0
            for i,tripinfo in enumerate(tripinfo_list):
                accumlated_JTime += (float(tripinfo.getAttribute("duration")))

            meanJtime = accumlated_JTime/i
            return meanJtime
        except:
            print("error while parsing the file , make sure the path is valid
                ↪ and the attributes are ok! ")
            print("please make sure the file is closed before reopening it !!!
                ↪ ")
```

Code for Traffic Responsive Traffic Light Control

reward.py

```
import traci
EPS = 0.1

class RewardCalculator:
    def __init__(self, alpha=0.5, log=True):
        self.vehicles = {}
        self._alpha = alpha
        self._is_logging = log
        self._log = []

    def tick(self):
        if self._alpha > 0:
            current_vehicle_ids = traci.vehicle.getIDList()
            old_vehicle = list(self.vehicles.keys())
            for vehID in old_vehicle:
                if vehID not in current_vehicle_ids:
                    # vehile has departed simulation, remove entry from vehicle
                        ↪  dictionary
                    self.vehicles.pop(vehID)
            for vehID in current_vehicle_ids:
                if vehID not in self.vehicles.keys():
                    self.vehicles[vehID] = 0
            for vehID in traci.simulation.getDepartedIDList():
                traci.vehicle.subscribe(vehID, [traci.constants.VAR_SPEED])
            subscription_results = traci.vehicle.getSubscriptionResults()
            for vehID in current_vehicle_ids:
                speed = subscription_results[vehID][traci.constants.VAR_SPEED]
                if speed < EPS:
                    self.vehicles[vehID] += 1
                else:
                    self.vehicles[vehID] = 0

    def get_reward(self):
        avg_speed = 0
        n_cars = 0
        for vehID in traci.vehicle.getIDList():
            avg_speed += traci.vehicle.getSpeed(vehID)
            n_cars += 1
        if n_cars == 0:
            avg_speed = 0
        else:
            avg_speed /= n_cars

        if abs(self._alpha ) < .000001:
            r = avg_speed
        else:
            if n_cars > 0:
                delay_penalty = float(sum(self.vehicles.values()))/n_cars
            else:
                delay_penalty = 0
            r = avg_speed - self._alpha*delay_penalty

        if self._is_logging:
            self._log.append(r)

        return r

    def get_log(self):
        if not self._is_logging:
            raise Exception("Loggging is not turned on")
        return self._log

    def reset(self):
        self._log = []
        self._vehicles = {}
```

Q-Learning

rl/rl.py

```
from reward import RewardCalculator
from simulation import Simulator
from stats.output_parser import SimulationOutputParser
from action import PhaseModifier
import numpy as np
import dtse
import subprocess
import time
import os

def randomize_traffic():
    os.chdir(r"C:\Users\eltay\Documents\grad_project\test_environments\
        ↪ single_intersection_no_traffic")
    p = subprocess.Popen(["python", r'C:\Program Files (x86)\DLR\Sumo\tools\
        ↪ randomTrips.py', '-n', 'newnet.net.xml',
                            '-e', '800', '-p', '1.2', '-r', 'newnet.rou.xml'],
                                ↪ stdout=subprocess.PIPE)
    out, err = p.communicate()
    time.sleep(.1)

class ActionRepresentation:
    def __init__(self, actions):
        self._eye = np.eye(len(actions))
        self._indices = dict(zip(actions, range(len(actions))))
        self._action_numbers = actions

    def get_one_hot(self, action_number):
        return self._eye[self._indices[action_number]]

    def get_phase_number(self, one_hot_vec):
        hot = np.argmax(one_hot_vec)
        return self._action_numbers[hot]

    def get_phases(self):
        return self._action_numbers

class StateGenerator:
    def __init__(self):
        pass

    def get_state(self):
        # TODO add traffic light phase to state
        state = np.array(())
        for lane,direction in zip(("-road1_0", "road2_0", "road3_0", "road4_0"
            ↪ ), ("in", "in", "in", "in")):
            a, b = dtse.DTSE_Generator.get_traffic_state(lane, direction, 15,
                ↪ 7.5)
            a, b = np.asarray(a), np.asarray(b)
            state = np.hstack((state, a.ravel(), b.ravel()))
        state = np.hstack((state))
        return state


class Controller:
    def __init__(self, phase_modifier, action_reprentation_handler,
        ↪ transitions_dict, transition_time):
        self._phase_modifier = phase_modifier
        self._action_representation = action_reprentation_handler
        self._current_phase = action_reprentation_handler.get_phases()[0]
        self._transitions_dict = transitions_dict
        self._transition_time = transition_time

        self._previous_phase = self._current_phase
        self._is_in_transition = False
        self._ticks_elapsed_in_transition = 0

    def do_action(self, action):
        applied_action = self._action_representation.get_phase_number(action)
        if applied_action != self._previous_phase:
            self._is_in_transition = True
            self._previous_phase = self._current_phase
            self._current_phase = applied_action


    def tick(self):
        if self._is_in_transition:
            self._phase_modifier.set_phase(self._transitions_dict[self.
                ↪ _previous_phase])
        self._phase_modifier.set_phase(self._current_phase)

    def reset(self):
        self._current_phase = self._action_representation.get_phases()[0]
        self._ticks_since_changed = 0

class SimpleController:
    def __init__(self, phase_modifier, action_reprentation_handler):
        self._phase_modifier = phase_modifier
        self._ticks_since_changed = 0
        self._action_representation = action_reprentation_handler
        self._current_phase = action_reprentation_handler.get_phases()[0]

    def do_action(self, action):
        self._current_phase = self._action_representation.get_phase_number(
            ↪ action)
        self._phase_modifier.set_phase(self._current_phase)

    def tick(self):
        self._phase_modifier.set_phase(self._current_phase)
        self._ticks_since_changed += 1

    def reset(self):
        self._current_phase = self._action_representation.get_phases()[0]
        self._ticks_since_changed = 0

class RLAgent:
    def __init__(self, reward_calc, state, controller, actions, training=True
        ↪ ):
        self._reward_calc = reward_calc
        self._state_calc = state
        self._controller = controller
        self._actions = actions
        self._previous_state = None
        self._previous_action = None
        self._is_training = training

    def tick(self):
        state = self._state_calc.get_state()[np.newaxis, :]
        reward = self._reward_calc.get_reward()
        if self._previous_state is not None and self._is_training:
            self._update_model(self._previous_state, self._previous_action,
                ↪ reward, state)
        action = self._select_action(state)
        self._controller.do_action(action)
        self._previous_action = action
        self._previous_state = state

    def reset(self):
        self._previous_state = None
        self._previous_action = None

    def set_is_training(self, training):
        self._is_training = training

    def _select_action(self, state):
        pass

    def _update_model(self, state, action, reward, new_state):
        pass
```

rl/tensor_flow_single_intersect.py (Runnable File)

```
from rl import RLAgent, StateGenerator, SimpleController,
    ↪ ActionRepresentation,randomize_traffic
from reward import RewardCalculator
from simulation import Simulator
from action import PhaseModifier
from stats import SimulationOutputParser
import tensorflow as tf
import numpy as np
import os


N_ACTIONS = 3

class SingleIntersectQAgent (RLAgent):
    def __init__(self, reward_calc, state, controller, actions):
        super().__init__(reward_calc, state, controller, actions)
        self._build_model_graph()
        self._discount = .9
        self._explore_prob = 0.2

    def _select_action(self, state):
        if np.random.rand(1) < self._explore_prob:
            action = np.eye(N_ACTIONS)[np.random.randint(N_ACTIONS)]
            return action
        output = self._session.run(self._model_output, feed_dict={self.
            ↪ _model_input: state})
        action_number = np.argmax(output)
        one_hot = np.eye(output.size)[action_number]
        return one_hot

    def set_explore_probability(self, explore_probability):
        self._explore_prob = explore_probability

    def _build_model_graph(self):
        self._model_input = tf.placeholder(dtype=tf.float64, shape=[1,120])
        layer1 = tf.layers.dense(inputs=self._model_input,units=100,
            ↪ activation=tf.nn.relu)
        layer2 = tf.layers.dense(inputs=layer1,units=100, activation=tf.nn.
            ↪ relu)
        layer3 = tf.layers.dense(inputs=layer2, units=3)
        self._model_output = layer3

        self._calculated_q = tf.placeholder(dtype=tf.float64, shape=(1,
            ↪ N_ACTIONS))
        self._loss = tf.reduce_sum(tf.square(self._calculated_q - self.
            ↪ _model_output))
        self._optimizer = tf.train.AdamOptimizer().minimize(self._loss)

        g_initializer = tf.global_variables_initializer()
        l_initializer = tf.local_variables_initializer()

        self._session = tf.Session()
        self._saver = tf.train.Saver(max_to_keep=10000)
        self._session.run(g_initializer)
        self._session.run(l_initializer)


    def _update_model(self, state, action, reward, new_state):
        initial_q = self._session.run(self._model_output, feed_dict={self.
            ↪ _model_input: state})
        next_q = self._session.run(self._model_output, feed_dict={self.
            ↪ _model_input: new_state})
        max_next_q = np.max(next_q)
        action_index = np.argmax(action)
        calculated_q = initial_q
        calculated_q[0,action_index] = reward + self._discount*max_next_q
        self._session.run(self._optimizer, feed_dict={self._model_input: state
            ↪ , self._calculated_q: calculated_q})

    def save_model(self, save_path):
        self._saver.save(self._session, save_path)

    def restore_model(self, save_path):
        self._saver.restore(self._session, save_path)

class SingleIntersectQAgentRunOnly(SingleIntersectQAgent):
    def _update_model(self, state, action, reward, new_state):
        pass

def train_model(model_name, alpha):
    sumocfg = "..\\..\\test_environments\\single_intersection_no_traffic\\
        ↪ newnet.sumocfg"
    rc = RewardCalculator(alpha=alpha)
    action_translator = ActionRepresentation([0, 3, 4])
    controller = SimpleController(PhaseModifier("node1"), action_translator)
    state_gen = StateGenerator()
    agent = SingleIntersectQAgent(rc, state_gen, controller, np.eye(3))
    explore_prob = .4
    results_means = []
    save_base_dir = "./rl_models/" + model_name + "/"
    save_interval = 50
    n_episodes = 1500
    for episode in range(n_episodes):
        controller.reset()
        rc.reset()
        agent.reset()

        explore_prob *= .99
        agent.set_explore_probability(explore_prob)

        randomize_traffic()
        sim = Simulator()
        sim.add_tickable(rc)
        sim.add_tickable(controller)
        sim.add_tickable(agent)
        sim.add_simulation_component(SimulationOutputParser)
        sim.run(sumocfg, time_steps=3000, gui=False)
        mean_speed = sim.results["mean_speed"].mean()
        mean_dur = sim.results["duration"].mean()
        mean_waiting_time = sim.results["waiting_time"].mean()
        mean_time_loss = sim.results["time_loss"].mean()
        mean_reward = np.asarray(rc.get_log()).mean()
        results_means.append((mean_speed, mean_dur, mean_waiting_time,
            ↪ mean_time_loss, mean_reward))
        print("Episode %i : avg speed %.3f, mean journey %.3f mean reward %.2f
            ↪ "%
              (episode, mean_speed, mean_dur, mean_reward))

        if episode% save_interval == 0 :
            os.chdir(os.path.dirname(__file__))
            print("saving.........")
            save_dir = save_base_dir + ("episode%i"%episode) + "/"
            agent.save_model(save_dir + model_name + str(episode))
            file_io = open(save_dir + model_name + "_train.txt", "w+")
            file_io.write(str(results_means))
            file_io.close()
            print("saved")
            print(list(zip(*results_means))[0])

    os.chdir(os.path.dirname(__file__))
    print("saving.........")
    save_dir = save_base_dir + ("episode%i" % n_episodes) + "/"
    agent.save_model(save_dir + model_name + str(n_episodes))
    file_io = open(save_dir + model_name + "_train.txt", "w+")
    file_io.write(str(results_means))
    file_io.close()
    print("saved")
    print(list(zip(*results_means))[0])


if __name__ == "__main__":
```

```
import time
for model_name, alpha in [("zero_alpha", 0),("half_alpha", 0.5)]:
    start = time.time()
    train_model(model_name, alpha)
    print("trained in %.2f seconds"%(time.time() - start))
```

Policy Gradient

policy_gradient/running_PG.py (Runnable File)

```
import numpy as np
from SARS import RewardCollector
from SARS import StateObserver
from SARS import Actor
from torch_network import PolicyNetwork
from reward import RewardCalculator
from simulation import Simulator
from stats import SimulationOutputParser
import subprocess
import time
import os
import pandas as pd

sumocfg1 = "..\\..\\test_environments\\single_intersection_no_traffic\\newnet.
    ↪ sumocfg"
roads_list = [["-road1_0","road3_0"],["road2_0","road4_0"]]
lossings = []
rewards = []
policy_network = PolicyNetwork()

#################################
def define_data_frame():
    simulation_dataFrame = pd.DataFrame({'iteration': [0],
                                     'mean_speed': [0],
                                     'duration': [0],
                                     'waiting_time': [0],
                                     'time_loss': [0],
                                      'vssd':[0],
                                       'jtsd':[0],
                                        'reward':[0]})
    simulation_dataFrame.set_index('iteration', inplace=True)
    return simulation_dataFrame


def generate_iteration_data_frame(iteration_no,mean_speed_result,
    ↪ duration_result,waiting_time,time_loss,vssd,jtsd,reward):
    iteration_dataFrame = pd.DataFrame(({'iteration': [iteration_no],
                                        'mean_speed': [mean_speed_result],
                                        'duration': [duration_result],
                                        'waiting_time': [waiting_time],
                                        'time_loss': [time_loss],
                                       'vssd':[vssd],
                                       'jtsd':[jtsd],
                                        'reward':[reward]}))
    iteration_dataFrame.set_index('iteration',inplace= True)
    return iteration_dataFrame


def concat_frames(f1,f2):
    frames = [f1, f2]
    frame = pd.concat(frames)
    return frame

def save_dataframe2CSV(f1,file):
    f1.to_csv(file)
##########################################

simulation_dataFrame = define_data_frame()


gamma = .9
def discount_and_norm_rewards(rewards):
    discounted_episode_rewards = np.zeros_like(rewards)
    cumulative = 0
    for t in reversed(range(len(rewards))):
        cumulative = cumulative * gamma + rewards[t]
        discounted_episode_rewards[t] = cumulative
    discounted_episode_rewards -= np.mean(discounted_episode_rewards)
    discounted_episode_rewards /= np.std(discounted_episode_rewards)
    return discounted_episode_rewards


def shift(seq, n):
    n = n % len(seq)
    return seq[n:] + seq[:n]

def randomize_traffic():
    os.chdir(
        r"D:\My study\5th year\Graduation Project\traffic-optimization\
            ↪ test_environments\single_intersection_no_traffic")
    p = subprocess.Popen(["python", r'C:\Program Files (x86)\DLR\Sumo\tools\
        ↪ randomTrips.py', '-n', 'newnet.net.xml',
                          '-e', '800', '-p', '1.2', '-r', 'newnet.rou.xml'],
                              ↪ stdout=subprocess.PIPE)
    out, err = p.communicate()
    time.sleep(.1)


iterations = 1500
for i in range (iterations):

    rc = RewardCalculator(alpha= 0.5)
    rewardCollector = RewardCollector(rc)

    sim = Simulator()
    randomize_traffic()
    observer = StateObserver(sim)
    sim.add_simulation_component(observer)
    actor = Actor(observer,policy_network)
    sim.add_tickable(actor)
    sim.add_tickable(rc)
    sim.add_tickable(rewardCollector)

    parser = SimulationOutputParser(sim)
    sim.add_tickable(parser)
    sim.add_postrun(parser.post_run())
    sim.run(sumocfg1, gui=False,time_steps = 2000)
    sim.results['rewards'] = rewardCollector.get_reward_log()
    sim.results['actions']= actor.get_actions_list()

    mean_speed_result = (np.mean(sim.results['mean_speed']))
    duration_result = (np.mean(sim.results['duration']))
    waiting_time = (np.mean(sim.results['waiting_time']))
    time_loss = (np.mean(sim.results['time_loss']))
    vssd = sim.results['mean_speed'].var() ** .5
    jtsd = sim.results['duration'].var() ** .5
    reward = np.mean(sim.results['rewards'])

    iteration_dataFrame = generate_iteration_data_frame(i, mean_speed_result,
        ↪  duration_result, waiting_time, time_loss,vssd,jtsd,reward)
    simulation_dataFrame = concat_frames(simulation_dataFrame,
        ↪ iteration_dataFrame)

    shifted_rewards = shift(sim.results['rewards'],1)

    discounted_normed_rewards = discount_and_norm_rewards(shifted_rewards)

    s = sim.results['states'][0]
    a = sim.results['actions']
    r = discounted_normed_rewards

    loss, reward= policy_network.train(np.array(s),a,np.array(r),i)
    lossings.append(loss)
    rewards.append(reward)


######################################################################

rc = RewardCalculator(alpha= 0.5)
rewardCollector = RewardCollector(rc)

sim = Simulator()
randomize_traffic()
observer = StateObserver(sim)
```

```
sim.add_simulation_component(observer)
actor = Actor(observer,policy_network)
sim.add_tickable(actor)
sim.add_tickable(rc)
sim.add_tickable(rewardCollector)

sim.add_simulation_component(SimulationOutputParser(sim))

sim.run(sumocfg1, gui=False,time_steps = 2000)
sim.results['rewards'] = rewardCollector.get_reward_log()
sim.results['actions']= actor.get_actions_list()

mean_speed_result = (np.mean(sim.results['mean_speed']))
duration_result = (np.mean(sim.results['duration']))
waiting_time = (np.mean(sim.results['waiting_time']))
time_loss = (np.mean(sim.results['time_loss']))
vssd = sim.results['mean_speed'].var() ** .5
jtsd = sim.results['duration'].var() ** .5

iteration_dataFrame = generate_iteration_data_frame(55555, mean_speed_result,
    ↪  duration_result, waiting_time, time_loss,vssd,jtsd,reward)
simulation_dataFrame = concat_frames(simulation_dataFrame,
    ↪ iteration_dataFrame)

shifted_rewards = shift(sim.results['rewards'],1)

discounted_normed_rewards = discount_and_norm_rewards(shifted_rewards)

s = sim.results['states'][0]
a = sim.results['actions']
r = discounted_normed_rewards

loss, reward= policy_network.train(np.array(s),a,np.array(r),55555)
lossings.append(loss)
rewards.append(reward)

#############################################################
```

```
save_dataframe2CSV(simulation_dataFrame,"D:\My study\\5th year\Graduation
    ↪ Project\\traffic-optimization\\rl_model\policy_gradient\pg_results.csv
    ↪ ")

policy_network.saveModel()
print(rewards)
```

policy_gradient/SARS.py

```
from xml.dom import minidom
from action import PhaseModifier
import numpy as np
import reward
from dtse import DTSE_Generator
from simulation import SimulationComponent
import torch


class StateObserver(SimulationComponent):

    def __init__(self, simulation):

        self._simulation = simulation
        self._roads_list = [["-road1_0", "road3_0"], ["road2_0", "road4_0"]]
        self._states = []
        self._current_state = 0



    def tick(self):
        state = self.get_state()
        self._states.append(state)
        pass

    def post_run(self):
        results = {"states": [], "actions": [], "rewards": []}
        results["states"].append(self._states)
        self._simulation.results = results
        pass
    def get_state(self):

        phase_1 = self._roads_list[0]
        phase_2 = self._roads_list[1]

        exist_1 = []
        exist_2 = []
        exist_3 = []
        exist_4 = []

        exist_1, speed_1 = DTSE_Generator.get_traffic_state(phase_1[0],
            ↪ direction="in", state_size=15, cell_size=7)
        exist_2, speed_2 = DTSE_Generator.get_traffic_state(phase_1[1],
            ↪ direction="in", state_size=15, cell_size=7)

        for road in phase_2:

            exist_3,speed_3 = DTSE_Generator.get_traffic_state(phase_2 [0],
                ↪ direction="in",state_size=15,cell_size =7)
            exist_4, speed_4 = DTSE_Generator.get_traffic_state(phase_2[1],
                ↪ direction="in", state_size=15, cell_size=7)

        state = np.concatenate([exist_1,exist_2,exist_3,exist_4,speed_1,
            ↪ speed_2,speed_3,speed_4])
        self._current_state = state
        return state


class RewardCollector:
    def __init__(self,reward_calculator):
        self._reward_calculator = reward_calculator
        self._reward_log = []
        pass

    def tick(self):
        self._reward_log.append(self._reward_calculator.get_reward())
        pass
    def get_reward_log(self):
        return self._reward_log

class Actor:
    def __init__(self,StateObserver,network):
        self._network = network
        self._actions_list = []
        self._StateObserver = StateObserver

        pass

    def tick(self):
        # actions probabilities (using softmax)
        modifier = PhaseModifier("node1")
        self._modifier = modifier
        action = self._network.get_action(self._StateObserver._current_state)
        self._actions_list.append(action)

        if action.data.item() == 0:
            self._modifier.set_phase(0)
        else:
            self._modifier.set_phase(4)
        pass

    def get_actions_list(self):
        return self._actions_list
```

policy_gradient/torch_network.py

```
import pandas as pd
import torch
import torch.nn as nn
from torch import autograd
import numpy as np
from torch.autograd import Variable
from torch.distributions import Categorical
from torch.distributions import Bernoulli
import torch.nn.functional as F
```

```
path = "D:\\My study\\5th year\\Graduation Project\\traffic-optimization\\
    ↪ rl_model\\policy_gradient\model.pt"

class PolicyNetwork:

    def __init__(self):
        self.model = self.NeuralNet()
        self.model = self.model.double()
        # Loss and optimizer
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=self.
            ↪ model.learning_rate)
        #self.optimizer = torch.optim.SGD(self.model.parameters(), lr=self.
            ↪ model.learning_rate)


        pass

    def get_action(self,state):
        #returns a torch tensor
        state = torch.from_numpy(state)
        #actions = self.model(state)
        probs = self.model(state)
        # Note that this is equivalent to what used to be called multinomial
        m = Categorical(probs)
        action = m.sample()

        return action

    def saveModel(self):



        torch.save(self.model, path)

    def loadModel(self):
        self.model = torch.load(path)



    def train(self, s, a, r,iteration):
        self.optimizer.zero_grad()

        states = torch.from_numpy(s)
        actions = a
        rewards = torch.from_numpy(r)

        for i in range (len(states)):
            state = states[i]
            action = actions[i]
            reward = rewards[i]

            probs = self.model(state)
            m = Categorical(probs)
            loss = -m.log_prob(action.double()) * reward
            loss.backward()





        self.optimizer.step()
        torch.save(self.model,
                    "D:\\My study\\5th year\\Graduation Project\\traffic-
                        ↪ optimization\\rl_model\\policy_gradient\model_alpha_0
                        ↪ .5\model_" + str(iteration) + ".pt")
        return(loss.item(),np.sum(rewards.numpy()))
        pass

    class NeuralNet(nn.Module):

        n_features = 120
        n_classes = 2
        layer1_neurons = 100
        layer2_neurons = 100
        learning_rate = .0001

        def __init__(self):
            super().__init__()
            self.fc1 = nn.Linear(self.n_features, self.layer1_neurons)
        self.fc2 = nn.Linear(self.layer1_neurons, self.layer2_neurons)
        self.fc3 = nn.Linear(self.layer2_neurons, self.n_classes)


        torch.set_default_tensor_type('torch.DoubleTensor')

    def forward(self, x):
        out = self.fc1(x)
        out = F.relu(out)
        out = self.fc2(out)
        out = F.relu(out)
        out = self.fc3(out)

        return F.softmax(out)
```

Appendix B

The Scenario

The simulation scenario used in this work was a SUMO simulation scenario and it consists of a network file (.net.xml), a sample traffic file (.rou.xml) and a configuration file file (.sumocfg)

Network File

This file was generated using the SUMO tool *NetEdit* and can also be edited with *NetEdit*.

`newnet.net.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- generated on 4/28/2018 4:52:19 PM by Netedit Version 0.32.0
<?xml version="1.0" encoding="UTF-8"?>

<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
   ↪ noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/
   ↪ netconvertConfiguration.xsd">

    <input>
        <sumo-net-file value="D:\My%20study\5th%20year\Graduation%20Project\
           ↪ traffic-optimization\test_environments\
           ↪ single_intersection_random_trips\newnet.net.xml"/>
    </input>

    <output>
```

```
        <output-file value="D:\My%20study\5th%20year\Graduation%20Project\
            ↪ traffic-optimization\test_environments\
            ↪ single_intersection_random_trips\newnet.net.xml"/>
    </output>

    <processing>
        <no-turnarounds value="true"/>
        <offset.disable-normalization value="true"/>
        <lefthand value="false"/>
        <junctions.corner-detail value="0"/>
        <rectangular-lane-cut value="false"/>
        <walkingareas value="false"/>
    </processing>

</configuration>
-->

<net version="0.27" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi
    ↪ :noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/net_file.xsd">

    <location netOffset="0.00,0.00" convBoundary="
        ↪ -250.00,-150.00,150.00,250.00" origBoundary="
        ↪ -10000000000.00,-10000000000.00,10000000000.00,10000000000.00"
        ↪ projParameter="!"/>

    <edge id=":node1_0" function="internal">
        <lane id=":node1_0_0" index="0" speed="13.89" length="5.00" shape="
            ↪ -51.65,54.75 -51.84,53.39 -52.42,52.43 -53.39,51.84
            ↪ -54.75,51.65"/>
    </edge>
    <edge id=":node1_1" function="internal">
        <lane id=":node1_1_0" index="0" speed="13.89" length="9.50" shape="
            ↪ -51.65,54.75 -51.65,45.25"/>
    </edge>
    <edge id=":node1_2" function="internal">
        <lane id=":node1_2_0" index="0" speed="13.89" length="5.28" shape="
            ↪ -51.65,54.75 -51.25,51.95 -50.05,49.95 -49.95,49.89"/>
    </edge>
    <edge id=":node1_12" function="internal">
```

```
        <lane id=":node1_12_0" index="0" speed="13.89" length="5.04" shape="
           ↪ -49.95,49.89 -48.05,48.75 -45.25,48.35"/>
    </edge>
    <edge id=":node1_3" function="internal">
        <lane id=":node1_3_0" index="0" speed="13.89" length="5.00" shape="
           ↪ -45.25,51.65 -46.61,51.84 -47.57,52.42 -48.16,53.39
           ↪ -48.35,54.75"/>
    </edge>
    <edge id=":node1_4" function="internal">
        <lane id=":node1_4_0" index="0" speed="13.89" length="9.50" shape="
           ↪ -45.25,51.65 -54.75,51.65"/>
    </edge>
    <edge id=":node1_5" function="internal">
        <lane id=":node1_5_0" index="0" speed="13.89" length="5.28" shape="
           ↪ -45.25,51.65 -48.05,51.25 -50.05,50.05 -50.11,49.95"/>
    </edge>
    <edge id=":node1_13" function="internal">
        <lane id=":node1_13_0" index="0" speed="13.89" length="5.04" shape="
           ↪ -50.11,49.95 -51.25,48.05 -51.65,45.25"/>
    </edge>
    <edge id=":node1_6" function="internal">
        <lane id=":node1_6_0" index="0" speed="13.89" length="5.00" shape="
           ↪ -48.35,45.25 -48.16,46.61 -47.58,47.57 -46.61,48.16
           ↪ -45.25,48.35"/>
    </edge>
    <edge id=":node1_7" function="internal">
        <lane id=":node1_7_0" index="0" speed="13.89" length="9.50" shape="
           ↪ -48.35,45.25 -48.35,54.75"/>
    </edge>
    <edge id=":node1_8" function="internal">
        <lane id=":node1_8_0" index="0" speed="13.89" length="5.28" shape="
           ↪ -48.35,45.25 -48.75,48.05 -49.95,50.05 -50.05,50.11"/>
    </edge>
    <edge id=":node1_14" function="internal">
        <lane id=":node1_14_0" index="0" speed="13.89" length="5.04" shape="
           ↪ -50.05,50.11 -51.95,51.25 -54.75,51.65"/>
    </edge>
    <edge id=":node1_9" function="internal">
```

```
        <lane id=":node1_9_0" index="0" speed="13.89" length="5.00" shape="
            ↪ -54.75,48.35 -53.39,48.16 -52.43,47.58 -51.84,46.61
            ↪ -51.65,45.25"/>
    </edge>
    <edge id=":node1_10" function="internal">
        <lane id=":node1_10_0" index="0" speed="13.89" length="9.50" shape="
            ↪ -54.75,48.35 -45.25,48.35"/>
    </edge>
    <edge id=":node1_11" function="internal">
        <lane id=":node1_11_0" index="0" speed="13.89" length="5.28" shape="
            ↪ -54.75,48.35 -51.95,48.75 -49.95,49.95 -49.89,50.05"/>
    </edge>
    <edge id=":node1_15" function="internal">
        <lane id=":node1_15_0" index="0" speed="13.89" length="5.04" shape="
            ↪ -49.89,50.05 -48.75,51.95 -48.35,54.75"/>
    </edge>

    <edge id="-road1" from="node2" to="node1" priority="1">
        <lane id="-road1_0" index="0" speed="13.89" length="195.25" shape="
            ↪ 150.00,51.65 -45.25,51.65"/>
    </edge>
    <edge id="-road2" from="node1" to="gneJ13" priority="1">
        <lane id="-road2_0" index="0" speed="13.89" length="195.25" shape="
            ↪ -51.65,45.25 -51.65,-150.00"/>
    </edge>
    <edge id="-road3" from="node1" to="gneJ15" priority="1">
        <lane id="-road3_0" index="0" speed="13.89" length="195.25" shape="
            ↪ -54.75,51.65 -250.00,51.65"/>
    </edge>
    <edge id="-road4" from="node1" to="gneJ14" priority="1">
        <lane id="-road4_0" index="0" speed="13.89" length="195.25" shape="
            ↪ -48.35,54.75 -48.35,250.00"/>
    </edge>
    <edge id="road1" from="node1" to="node2" priority="1">
        <lane id="road1_0" index="0" speed="13.89" length="195.25" shape="
            ↪ -45.25,48.35 150.00,48.35"/>
    </edge>
    <edge id="road2" from="gneJ13" to="node1" priority="1">
    <lane id="road2_0" index="0" speed="13.89" length="195.25" shape="
        ↪ -48.35,-150.00 -48.35,45.25"/>
</edge>
<edge id="road3" from="gneJ15" to="node1" priority="1">
    <lane id="road3_0" index="0" speed="13.89" length="195.25" shape="
        ↪ -250.00,48.35 -54.75,48.35"/>
</edge>
<edge id="road4" from="gneJ14" to="node1" priority="1">
    <lane id="road4_0" index="0" speed="13.89" length="195.25" shape="
        ↪ -51.65,250.00 -51.65,54.75"/>
</edge>

<tlLogic id="node1" type="static" programID="1" offset="0">
    <phase duration="33" state="GGgrrrGGgrrr"/>
    <phase duration="3" state="yygrrryygrrr"/>
    <phase duration="6" state="rrGrrrrrGrrr"/>
    <phase duration="3" state="rryrrrrryrrr"/>
    <phase duration="33" state="rrrGGgrrrGGg"/>
    <phase duration="3" state="rrryygrrryyg"/>
    <phase duration="6" state="rrrrrGrrrrrG"/>
    <phase duration="3" state="rrrrryrrrrry"/>
</tlLogic>

<junction id="gneJ13" type="dead_end" x="-50.00" y="-150.00" incLanes="-
    ↪ road2_0" intLanes="" shape="-49.95,-150.00 -53.25,-150.00
    ↪ -50.05,-150.00"/>
<junction id="gneJ14" type="dead_end" x="-50.00" y="250.00" incLanes="-
    ↪ road4_0" intLanes="" shape="-50.05,250.00 -46.75,250.00
    ↪ -49.95,250.00"/>
<junction id="gneJ15" type="dead_end" x="-250.00" y="50.00" incLanes="-
    ↪ road3_0" intLanes="" shape="-250.00,49.95 -250.00,53.25
    ↪ -250.00,50.05"/>
<junction id="node1" type="traffic_light" x="-50.00" y="50.00" incLanes="
    ↪ road4_0 -road1_0 road2_0 road3_0" intLanes=":node1_0_0 :node1_1_0 :
    ↪ node1_12_0 :node1_3_0 :node1_4_0 :node1_13_0 :node1_6_0 :node1_7_0
    ↪ :node1_14_0 :node1_9_0 :node1_10_0 :node1_15_0" shape="-53.25,54.75
    ↪  -46.75,54.75 -45.25,53.25 -45.25,46.75 -46.75,45.25 -53.25,45.25
    ↪ -54.75,46.75 -54.75,53.25">
```

```
        <request index="0" response="000000000000" foes="000100010000" cont="0
            ↪ "/>
        <request index="1" response="000000000000" foes="111100110000" cont="0
            ↪ "/>
        <request index="2" response="000011000000" foes="110011110000" cont="1
            ↪ "/>
        <request index="3" response="000010000000" foes="100010000000" cont="0
            ↪ "/>
        <request index="4" response="000110000111" foes="100110000111" cont="0
            ↪ "/>
        <request index="5" response="011110000110" foes="011110000110" cont="1
            ↪ "/>
        <request index="6" response="000000000000" foes="010000000100" cont="0
            ↪ "/>
        <request index="7" response="000000000000" foes="110000111100" cont="0
            ↪ "/>
        <request index="8" response="000000000011" foes="110000110011" cont="1
            ↪ "/>
        <request index="9" response="000000000010" foes="000000100010" cont="0
            ↪ "/>
        <request index="10" response="000111000110" foes="000111100110" cont="
            ↪ 0"/>
        <request index="11" response="000110011110" foes="000110011110" cont="
            ↪ 1"/>
    </junction>
    <junction id="node2" type="dead_end" x="150.00" y="50.00" incLanes="
        ↪ road1_0" intLanes="" shape="150.00,50.05 150.00,46.75 150.00,49.95"
        ↪ />

    <junction id=":node1_12_0" type="internal" x="-49.95" y="49.89" incLanes=
        ↪ ":node1_2_0 road2_0" intLanes=":node1_4_0 :node1_5_0 :node1_6_0 :
        ↪ node1_7_0 :node1_10_0 :node1_11_0"/>
    <junction id=":node1_13_0" type="internal" x="-50.11" y="49.95" incLanes=
        ↪ ":node1_5_0 road3_0" intLanes=":node1_1_0 :node1_2_0 :node1_7_0 :
        ↪ node1_8_0 :node1_9_0 :node1_10_0"/>
    <junction id=":node1_14_0" type="internal" x="-50.05" y="50.11" incLanes=
        ↪ ":node1_8_0 road4_0" intLanes=":node1_0_0 :node1_1_0 :node1_4_0 :
        ↪ node1_5_0 :node1_10_0 :node1_11_0"/>
```

```
<junction id=":node1_15_0" type="internal" x="-49.89" y="50.05" incLanes=
    ↪ ":node1_11_0 -road1_0" intLanes=":node1_1_0 :node1_2_0 :node1_3_0 :
    ↪ node1_4_0 :node1_7_0 :node1_8_0"/>

<connection from="-road1" to="-road4" fromLane="0" toLane="0" via=":
    ↪ node1_3_0" tl="node1" linkIndex="3" dir="r" state="o"/>
<connection from="-road1" to="-road3" fromLane="0" toLane="0" via=":
    ↪ node1_4_0" tl="node1" linkIndex="4" dir="s" state="o"/>
<connection from="-road1" to="-road2" fromLane="0" toLane="0" via=":
    ↪ node1_5_0" tl="node1" linkIndex="5" dir="l" state="o"/>
<connection from="road2" to="road1" fromLane="0" toLane="0" via=":
    ↪ node1_6_0" tl="node1" linkIndex="6" dir="r" state="o"/>
<connection from="road2" to="-road4" fromLane="0" toLane="0" via=":
    ↪ node1_7_0" tl="node1" linkIndex="7" dir="s" state="o"/>
<connection from="road2" to="-road3" fromLane="0" toLane="0" via=":
    ↪ node1_8_0" tl="node1" linkIndex="8" dir="l" state="o"/>
<connection from="road3" to="-road2" fromLane="0" toLane="0" via=":
    ↪ node1_9_0" tl="node1" linkIndex="9" dir="r" state="o"/>
<connection from="road3" to="road1" fromLane="0" toLane="0" via=":
    ↪ node1_10_0" tl="node1" linkIndex="10" dir="s" state="o"/>
<connection from="road3" to="-road4" fromLane="0" toLane="0" via=":
    ↪ node1_11_0" tl="node1" linkIndex="11" dir="l" state="o"/>
<connection from="road4" to="-road3" fromLane="0" toLane="0" via=":
    ↪ node1_0_0" tl="node1" linkIndex="0" dir="r" state="o"/>
<connection from="road4" to="-road2" fromLane="0" toLane="0" via=":
    ↪ node1_1_0" tl="node1" linkIndex="1" dir="s" state="o"/>
<connection from="road4" to="road1" fromLane="0" toLane="0" via=":
    ↪ node1_2_0" tl="node1" linkIndex="2" dir="l" state="o"/>

<connection from=":node1_0" to="-road3" fromLane="0" toLane="0" dir="r"
    ↪ state="M"/>
<connection from=":node1_1" to="-road2" fromLane="0" toLane="0" dir="s"
    ↪ state="M"/>
<connection from=":node1_2" to="road1" fromLane="0" toLane="0" via=":
    ↪ node1_12_0" dir="l" state="m"/>
<connection from=":node1_12" to="road1" fromLane="0" toLane="0" dir="l"
    ↪ state="M"/>
<connection from=":node1_3" to="-road4" fromLane="0" toLane="0" dir="r"
    ↪ state="M"/>
    <connection from=":node1_4" to="-road3" fromLane="0" toLane="0" dir="s"
        ↪ state="M"/>
    <connection from=":node1_5" to="-road2" fromLane="0" toLane="0" via=":
        ↪ node1_13_0" dir="l" state="m"/>
    <connection from=":node1_13" to="-road2" fromLane="0" toLane="0" dir="l"
        ↪ state="M"/>
    <connection from=":node1_6" to="road1" fromLane="0" toLane="0" dir="r"
        ↪ state="M"/>
    <connection from=":node1_7" to="-road4" fromLane="0" toLane="0" dir="s"
        ↪ state="M"/>
    <connection from=":node1_8" to="-road3" fromLane="0" toLane="0" via=":
        ↪ node1_14_0" dir="l" state="m"/>
    <connection from=":node1_14" to="-road3" fromLane="0" toLane="0" dir="l"
        ↪ state="M"/>
    <connection from=":node1_9" to="-road2" fromLane="0" toLane="0" dir="r"
        ↪ state="M"/>
    <connection from=":node1_10" to="road1" fromLane="0" toLane="0" dir="s"
        ↪ state="M"/>
    <connection from=":node1_11" to="-road4" fromLane="0" toLane="0" via=":
        ↪ node1_15_0" dir="l" state="m"/>
    <connection from=":node1_15" to="-road4" fromLane="0" toLane="0" dir="l"
        ↪ state="M"/>

</net>
```

Traffic File

The traffic file was generated with the SUMO supplied script *randomTrips.py* and can be generated by opening a terminal, changing to the directory of the network file and executing the following command

```
python <sumo_install_directory>\tools\randomTrips.py -n newnet.net.xml
-e 800 -p 1.2 -r newnet.rou.xml
```

The command will generate the output file *newnet.rou.xml* as well as 2 other files *trips.trips.xml* and *newnet.rou.alt.xml*. A sample of the final output file *newnet.rou.xml* is included below

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- generated on 04/25/18 14:04:36 by SUMO duarouter Version 0.32.0
<?xml version="1.0" encoding="UTF-8"?>

<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
   ↪ noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/
   ↪ duarouterConfiguration.xsd">

    <input>
        <net-file value="newnet.net.xml"/>
        <route-files value="trips.trips.xml"/>
    </input>

    <output>
        <output-file value="newnet.rou.xml"/>
        <alternatives-output value="newnet.rou.alt.xml"/>
    </output>

    <processing>
        <ignore-errors value="true"/>
    </processing>

    <time>
        <begin value="0"/>
        <end value="800.0"/>
    </time>

    <report>
        <no-warnings value="true"/>
        <no-step-log value="true"/>
    </report>

</configuration>
-->

<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
   ↪ noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/routes_file.xsd">
```

```
<vehicle id="1" depart="1.20">
    <route edges="road3 road1"/>
</vehicle>
<vehicle id="4" depart="4.80">
    <route edges="road3 -road4"/>
</vehicle>
<vehicle id="5" depart="6.00">
    <route edges="-road1 -road2"/>
</vehicle>
<vehicle id="6" depart="7.20">
    <route edges="-road1 -road3"/>
</vehicle>
<vehicle id="7" depart="8.40">
    <route edges="road3 road1"/>
</vehicle>
<vehicle id="8" depart="9.60">
    <route edges="road3 -road2"/>
</vehicle>
<vehicle id="9" depart="10.80">
    <route edges="road3 road1"/>
</vehicle>
<vehicle id="10" depart="12.00">
    <route edges="-road1 -road2"/>
</vehicle>

............................

<vehicle id="662" depart="794.40">
    <route edges="road4 -road2"/>
</vehicle>
<vehicle id="664" depart="796.80">
    <route edges="road2 -road3"/>
</vehicle>
<vehicle id="665" depart="798.00">
    <route edges="road3 -road2"/>
</vehicle>
<vehicle id="666" depart="799.20">
    <route edges="road3 road1"/>
</vehicle>
```

```
</routes>
```

Configuration File

This file tells the simulator which network and traffic files to use

`newnet.sumocfg`

```
<?xml version="1.0" encoding="iso-8859-1"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://sumo.sf.net/xsd/sumoConfiguration.xsd"
    ↪ >
    <input>
        <net-file value="newnet.net.xml"/>
        <route-files value="newnet.rou.xml"/>
    </input>
    <time>
        <begin value="0" />
        <end value="10000"/>
    </time>
    <time-to-teleport value="-1"/>
</configuration>
```

References

[1] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," 2011.

[2] W. Genders and S. Razavi, "Using a deep reinforcement learning agent for traffic signal control," *CoRR*, vol. abs/1611.01142, 2016.

[3] D. Zhao, Y. Dai, and Z. Zhang, "Computational intelligence in urban traffic signal control: A survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 4, pp. 485–494, July 2012.

[4] F. V. Webster, "Traffic signal setting." Road Res. Lab., HMSO, London, U.K., Tech.: Springer Berlin Heidelberg, 1958, pp. 1–44.

[5] A. J. Miller, "Settings for fixed-cycle traffic signals," vol. 14, pp. 373–386, 12 1963.

[6] L. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338 – 353, 1965. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S001999586590241X

[7] C. P. Pappis and E. H. Mamdani, "A fuzzy logic controller for a trafc junction," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 7, no. 10, pp. 707–717, 1977.

[8] J.-H. Lee and H. Lee-Kwang, "Distributed and cooperative fuzzy controllers for traffic intersections group," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 29, no. 2, pp. 263–271, May 1999.

[9] C.-H. Chou and J.-C. Teng, "A fuzzy logic controller for traffic junction signals," *Information Sciences*, vol. 143, no. 1-4, pp. 73–97, 2002.

[10] J. Qiao, N. Yang, and J. Gao, "Two-stage fuzzy logic controller for signalized intersection," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 41, no. 1, pp. 178–184, 2011.

[11] B. P. Gokulan and D. Srinivasan, "Distributed geometric fuzzy multiagent urban traffic signal control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 11, no. 3, pp. 714–727, 2010.

[12] I. Alvarez, A. Poznyak, and A. Malo, "Urban traffic control problem via a game theory application," in *Decision and Control, 2007 46th IEEE Conference on*. IEEE, 2007, pp. 2957–2961.

[13] S.-F. Cheng, M. A. Epelman, and R. L. Smith, “Cosign: A parallel algorithm for coordinated traffic signal control,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 7, no. 4, pp. 551–564, 2006.

[14] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[15] M. Srinivas and L. M. Patnaik, “Genetic algorithms: a survey,” *Computer*, vol. 27, no. 6, pp. 17–26, June 1994.

[16] C. Reeves, *Genetic Algorithms*. Boston, MA: Springer US, 2003, pp. 55–82. [Online]. Available: https://doi.org/10.1007/0-306-48056-5_3

[17] T. Kalganova, G. Russell, and A. Cumming, “Multiple traffic signal control using a genetic algorithm,” in *Artificial Neural Nets and Genetic Algorithms*. Vienna: Springer Vienna, 1999, pp. 220–228.

[18] N. M. Rouphail, B. B. Park, and J. Sacks, “Direct signal timing optimization: Strategy development and results,” In XI Pan American Conference in Traffic and Transportation Engineering, Tech. Rep., 2000.

[19] “TRANSYT-7F™ – McTrans.” [Online]. Available: https://mctrans.ce.ufl.edu/mct/index.php/hcs/transyt-7f/

[20] A. Kheiri, “Multi-stage hyper-heuristics for optimisation problems,” Ph.D. dissertation, dec 2014. [Online]. Available: http://eprints.nottingham.ac.uk/29960/

[21] B. W. Wah and A. Ieumwananonthachai, *Teacher: A Genetics Based System for Learning and Generalizing Heuristics*. London: Springer London, 2001, pp. 179–211. [Online]. Available: https://doi.org/10.1007/978-1-4471-0687-6_8

[22] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, “Hyper-heuristics: a survey of the state of the art,” *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, Dec 2013. [Online]. Available: https://doi.org/10.1057/jors.2013.71

[23] M. Dorigo and M. Birattari, *Ant Colony Optimization*. Boston, MA: Springer US, 2010, pp. 36–39. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_22

[24] J. Kennedy, *Particle Swarm Optimization*. Boston, MA: Springer US, 2010, pp. 760–766. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_630

[25] B. Bilgin, E. Özcan, and E. E. Korkmaz, “An experimental study on hyper-heuristics and exam timetabling,” in *Practice and Theory of Automated Timetabling VI*, E. K. Burke and H. Rudová, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 394–412.

[26] C. Dong, S. Huang, and X. Liu, “Comparative study of several intelligent optimization algorithms for traffic control applications,” in *2011 International Conference on Electronics, Communications and Control (ICECC)*, Sept 2011, pp. 4219–4223.

[27] B. Burvall and J. Olegård, "A comparison of a genetic algorithm and simulated annealing applied to a traffic light control problem: A traffic intersection optimization problem," 2015.

[28] J. Garcia-Nieto, A. C. Olivera, and E. Alba, "Optimal cycle program of traffic lights with particle swarm optimization," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 6, pp. 823–839, 2013.

[29] N. R. Sabar, L. M. Kieu, E. Chung, T. Tsubota, and P. E. M. de Almeida, "A memetic algorithm for real world multi-intersection traffic signal optimisation problems," *Engineering Applications of Artificial Intelligence*, vol. 63, pp. 45 – 53, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197617300854

[30] R. S. Sutton, A. G. Barto *et al.*, *Reinforcement learning: An introduction.* MIT press, 1998.

[31] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[32] E. L. E. L. Thorndike, *Animal intelligence, experimental studies,.* New York,The Macmillan company,, 1911, https://www.biodiversitylibrary.org/bibliography/1201 — The study of consciousness and the study of behavior.–Animal intelligence.–The instinctive reactions of young chicks.–A note on the psychology of fishes.–The mental life of the monkeys.–Law and h. [Online]. Available: https://www.biodiversitylibrary.org/item/16001

[33] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, July 1959.

[34] D. Lynch, M. Fenton, S. Kucera, H. Claussen, and M. O'Neill, "Scheduling in heterogeneous networks using grammar-based genetic programming," in *European Conference on Genetic Programming*. Springer, 2016, pp. 83–98.

[35] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *CoRR*, vol. abs/1708.05866, 2017. [Online]. Available: http://arxiv.org/abs/1708.05866

[36] R. Bellman, "A markovian decision process," *Indiana Univ. Math. J.*, vol. 6, pp. 679–684, 1957.

[37] A. K. Dixit, *Optimization in economic theory.* Oxford University Press on Demand, 1990.

[38] D. P. Bertsekas, D. P. Bertsekas, D. P. Bertsekas, and D. P. Bertsekas, *Dynamic programming and optimal control.* Athena scientific Belmont, MA, 2005, vol. 1, no. 3.

[39] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[40] A. G. Barto, R. S. Sutton, and C. J. Watkins, "Learning and sequential decision making," in *Learning and computational neuroscience*. Citeseer, 1989.

[41] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[42] F. S. Melo, “Convergence of q-learning: A simple proof,” *Institute Of Systems and Robotics, Tech. Rep*, pp. 1–4, 2001.

[43] F. S. Melo, S. P. Meyn, and M. I. Ribeiro, “An analysis of reinforcement learning with function approximation,” in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 664–671.

[44] M. P. Deisenroth, G. Neumann, J. Peters *et al.*, “A survey on policy search for robotics,” *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[45] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[46] V. R. Konda and J. N. Tsitsiklis, “Actor-critic algorithms,” in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[47] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, p. 529, 2015.

[48] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, no. 7587, p. 484, 2016.

[49] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, “Target-driven visual navigation in indoor scenes using deep reinforcement learning,” in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 3357–3364.

[50] S. El-Tantawy and B. Abdulhai, “An agent-based learning towards decentralized and coordinated traffic signal control,” in *13th International IEEE Conference on Intelligent Transportation Systems*, Sept 2010, pp. 665–670.

[51] W. Genders and S. Razavi, “Using a deep reinforcement learning agent for traffic signal control,” *arXiv preprint arXiv:1611.01142*, 2016.

[52] X. Liang, X. Du, G. Wang, and Z. Han, “Deep reinforcement learning for traffic light control in vehicular networks,” *arXiv preprint arXiv:1803.11115*, 2018.

[53] M. Wiering, J. Vreeken, J. Veenen, and A. Koopman, “Simulation and optimization of traffic in a city,” in *IEEE Intelligent Vehicles Symposium (IV’04)*, 2004.

[54] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, “Recent development and applications of SUMO - Simulation of Urban MObility,” *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3&4, pp. 128–138, December 2012.

[55] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/