Signal Optimization via Heuristic Search and Traffic Simulation

by

Abdullah

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Traffic congestion is a severe problem in many cities. One way to reduce it is by optimizing traffic signal timings. Experts spend a lot of time analyzing traffic patterns to produce good handcrafted timing schedules. However, these timing schedules can be less responsive when there is a sudden change in traffic flow. In this thesis, a novel way to formulate the traffic signal optimization problem as a signal-player game is proposed. The model uses a heuristic search algorithm called Monte Carlo Tree Search (MCTS) which is incorporated with a traffic simulator called Simulation of Urban MObility (SUMO) to approximate the optimal traffic signal timings. The model is tested against handcrafted timing schedules across different types of road networks, such as interconnected intersections, a long corridor of intersections, and intersections with Light Rail Transit (LRT) crossings. Experimental results show that our model performs significantly better in most cases when compared to our handcrafted policies. For instance, in one of the networks, our search model outperforms the handcrafted policy by 29% in all performance measures we considered. Moreover, in a real-world scenario with LRT crossings, MCTS surpassed the handcrafted policy by 18%. The strength of our model is that it can foresee changes in traffic flow patterns through simulations and react accordingly. Therefore, MCTS along with simulations is a viable alternative to experts handcrafting traffic light timing policies manually.

Acknowledgements

I am grateful to my supervisor Dr. Michael Buro for all his help and support. His guidance and valuable comments helped me think more critically. I am also thankful to Autonomous Systems Initiative (ASI) for supporting this project financially.

Most importantly, I would like thank my wife, Suzana for all her love and support through out my masters. She motivated me when I felt depressed and always encouraged me to give my best. I also want to thank my parents for their love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Throughout the last decade, there has been a drastic growth in population in most major cities of the world. Similarly, the number of vehicles on the road has risen considerably [31]. Thus, traffic jams are causing serious problems for many cities because they have an adverse impact on people and the environment. Waiting increases fuel consumption, which leads to an increased amount of money spent on fuels [30]. Moreover, traffic congestion can negatively affect a city's economy in many ways. For instance, congestion can stress people out because it makes people late to work. This can badly affect their mental state and as a result, their performances can get worse. In addition, important deliveries such as food or medical supplies can arrive late.

Traffic congestion can be mitigated in various ways. Two of the most common approaches are building new roads or expanding existing roads. Both are expensive and require a lot of time to implement. Another way to tackle traffic congestion is to optimize traffic signal timings for increasing the traffic flow. Optimizing the traffic timings can save money since it takes an existing traffic infrastructure and tries to increase traffic throughput in the network. Therefore, we want to create a timing schedule that can adapt and react to increasing traffic within a network of roads. This can greatly improve the traffic movements without having to alter any road infrastructure [66]. Since the roads in a city are inter-connected with alleys and other narrow roads, optimizing only the major intersections is insufficient as this may increase the flow of traffic towards smaller intersections. Hence, optimization should be

done on all intersections of a given network. Optimizing traffic signal timings can improve network flow while reducing other factors such as fuel consumption or waiting time. To improve traffic flow, different objective functions can be optimized [24]. Popular optimization objectives are: increasing the total distance traveled or the number of completed trips or reducing the number of stops.

Traffic optimization techniques try to reduce traffic congestion by creating optimized timing schedules for traffic signals. They can effectively minimize delays, fuel consumption and improve flow, safety, and cost management in the traffic system. Previous work in this domain is heavily based on mathematical optimization or evolutionary techniques. However, mathematical optimization techniques require precise mathematical domain models. In addition, real-world schedules are handwritten, which requires extensive analysis of traffic patterns. In this thesis, we propose a model that uses a search algorithm along with traffic simulation to approximate the optimal timings for traffic signals without requiring to analyze any traffic patterns. Search algorithms explore different possibilities to identify patterns in traffic flow by themselves.

In this work, we model the traffic light optimization problem as a single-player game and use a search algorithm to approximate the optimal timings of traffic signals. In particular, given an objective function f that maps a traffic history of fixed duration T, to a real number such as total distance driven, our goal is to find a sequence of traffic light duration that maximizes f. A more detailed formulation is given in Ch. 3. To optimize traffic light schedules we use a search algorithm called Monte Carlo Tree Search (MCTS) to achieve traffic signal coordination across several traffic intersections. MCTS takes the current state of the game and builds a search tree by choosing moves from a set of possible actions and performing simulations. These simulations help MCTS determine the outcome or result of choosing particular action sequences. The reason behind using a search algorithm along with simulations is that it gives us the ability to look ahead and find changes in traffic movements. To perform traffic simulations, we use a traffic simulator called Simulation of Urban MObility (SUMO) [36]. For this research, we have written our own MCTS al-

gorithm that uses the SUMO simulator. We tested the performance of MCTS across various traffic networks, including real-world scenarios containing railway crossings. To measure the performance of MCTS, we used four different traffic metrics: the total distance traveled in the network by all the vehicles, the total number of stops, total delays, and the number of completed trips. We compare MCTS's performance with handwritten traffic timing schedules. Our experiments show that MCTS either performs better or equally well to the handwritten schedules we designed, which indicates that search algorithms can be beneficial in this domain.

The contributions of our research work are:

1. We formulated the traffic signal optimization problem as a single-player game, which allows us to use standard tree search algorithms, such as MCTS, for optimization

2. We created a model that takes LRT schedules into consideration and looks ahead into the future and plans accordingly. Previous work on traffic optimization only considers current data or historic traffic data.

3. We use MCTS to make decisions for multiple intersections in a network by considering major traffic participants (like LRTs) to plan better. Monte Carlo Tree Search has been used previously for a single intersection but not for a whole network. .

The remainder of the thesis is organized as follows. Ch. 2 will focus on related work and background knowledge on traffic signal optimization and heuristic search. Ch. 3 will discuss the formulation of traffic light optimization as a single-player game and describes how a search algorithm can be combined with a traffic simulator. Ch. 4 will present our experimental setup. Subsequently, Ch. 5 will describe our simulation results as well as their analysis. Finally, Ch. 6 will provide a summary and a discussion of future work.

Chapter 2

Background

This chapter provides some necessary background knowledge to better understand the thesis. First, important terminology related to traffic signals is introduced. Then we discuss previous work on traffic signal optimization and Monte Carlo Tree Search.

2.1 Terminology

There are three terms that are fundamental to traffic signal coordination: cycle length, split, and offset. The definition of each term is taken from the literature [61].

1. **Cycle Length** is defined as the time it takes to iterate over a set of distinct phases. For example, a traffic signal has three phases, yellow (Y), red (R), and green (G), and they are arranged in the following sequence: YRG. The duration of the yellow, red, and green phases are 5, 10, and 15 seconds, respectively. Then the cycle length is 30 seconds. To coordinate the phases between two adjacent intersections, their cycle lengths need to be the same [45].

2. **Split** are commonly referred to as the length of the green phase. In other words, it is the amount of time set aside for this phase. The split for the example described above will be 15 seconds.

3. **Offsets** refer to the difference in time between the same phase of two adjacent traffic signals. It defines how far behind or ahead the starting

time of a traffic signal's phase is with respect to a reference point, which can either be a traffic signal or a master clock. For example, if there are two traffic signals, A and B, and A is the reference point then the offset of B is how far behind B's green phase is to A's green phase.

2.2 Traffic Signal Optimization

Finding a good sequence of timings for traffic signals in a short period of time is difficult because it is a combinatorial problem. To get real-time solutions, different optimization algorithms can be used to achieve coordination within traffic networks [18]. Existing traffic control systems can be divided into two major groups: fixed timing systems and responsive timing systems.

2.2.1 Fixed Timing Systems

In fixed timing systems, all parameters, such as cycle lengths, offsets, splits, and phases are kept fixed for certain time period during the day based on historic traffic data [21]. To accommodate for changing traffic flow throughout the day, different hand-crafted timing schedules are used that are created based on traffic flow [66]. The shortcoming of fixed timing systems is that they can not react to sudden changes in traffic flow.

2.2.2 Responsive Timing Systems

Since the 1970s, a variety of responsive traffic control systems have been implemented for improving traffic signal coordination systems (e.g., [4], [34] and [20]). For example, many cities around the world are using systems such as SCOOT (Split Cycle Offset Optimization Technique) [6], SCATS (Sydney Coordinated Adaptive Traffic System) [57], and TRANSYT (TRAffic Network StudY Tool) [47] to optimize their traffic control systems. SCATS is one of the earliest traffic control systems that significantly improved traffic movement at a low cost. To minimize the degree of saturation of roads, SCATS chooses a plan from a set of predefined signal plans for different times of the day. Another widely used system is SCOOT. It reduces the number of stops

and delays in a network by continuously measuring traffic density on all approaches of an intersection and making small changes to phase lengths, offsets, and cycle lengths. TRANSYT is based on a hill-climbing type of optimization technique that tries to balance total delay and the number of stops in a network. Responsive traffic control systems can be further split into two categories: actuated and adaptive traffic control systems.

Actuated Traffic Control Systems

In actuated traffic control systems, the phase lengths are constantly adjusted by utilizing traffic sensors. They use simple calculations to come up with signal timings. These systems do not predict traffic flow but rather detect changes in traffic using historical and current data. There are two types of actuated traffic control systems based on sensor usage. The first one is a fully actuated traffic control system and the other one is semi-actuated. In the semi-actuated system, the sensors are used only in the minor roads, whereas in the fully actuated system, all roads at different intersections have traffic sensors [10].

Adaptive Traffic Control Systems

Adaptive traffic control systems optimize by predicting traffic movement and planning accordingly to improve vehicle movement in arterial roads, which are high-capacity urban roads [4], [17]. Different algorithms have been proposed for adaptive traffic control systems, which are discussed next.

1. **Webster**

   Webster [29], calculates a cycle length that reduces the waiting time of vehicles, which is the delay at an intersection. The cycle length is computed based on the number of vehicles and the time loss. Time loss is the time required to accelerate and leave the intersection. Once the cycle length is determined, the green split is calculated based on the ratio of the volume of vehicles per phase. Since Webster only works with a single intersection, it cannot coordinate multiple intersections.

2. **Green Wave**

   Another common signal scheduling technique is called a green wave. It minimizes vehicle delays by syncing the traffic signals along a corridor, ensuring smooth progression of the vehicles [4]. This strategy drastically reduces waiting times and pollution and at the same time increases traffic movement throughout a city. To achieve a green wave, the cycle lengths for all intersections need to be the same. The offsets for the green phases are calculated based on the distance between intersections and the average speed of vehicles. Many optimization algorithms, such as Particle Swarm Optimization and Genetic Algorithms can be used to find good green wave schedules. The green waves can greatly benefit vehicles traveling in one direction due to the continuous sequence of green signals but might not be beneficial in the opposite direction [63].

   An algorithm was proposed in [5], which calculates the starting timing of green phases to create a green wave. The authors calculated the amount of time required for an intersection to stay green so that an emergency vehicle can pass through without stopping. The formula was the queue length at an intersection multiplied with the time it takes for one car to pass that intersection. Furthermore, to ensure coordination among consecutive intersections for green waves, the authors suggested that one traffic light should turn green before another traffic light depending on the route of the emergency vehicle. In addition, the difference in their timings should be the time it takes for a car to travel from one intersection to another.

3. **MAXBAND**

   Using mixed-integer linear programming, a traffic control system called MAXBAND was introduced in [35]. MAXBAND utilizes fixed cycle lengths and determines the splits and offsets to find the maximum bandwidth for the green phases to synchronize intersections along a corridor. In another study [19] the authors improved the MAXBAND method by proposing the MULTI-BAND method, which not only consid-

ers traffic movement in one direction but also incorporates dedicated left turns. Moreover, in [60] the authors modified the MAXBAND method by dividing a large network into smaller sub-networks. Then, by using MAXBAND, the model finds the maximum green bandwidth solution for these small networks. Afterwards, these sub-solutions are used to build a solution for the whole network prioritizing the peak traffic direction.

4. **Actuated and Self-Organizing Control Systems**

   Actuated and self-organizing control systems use sensor data along with a set of rules to determine phase lengths (e.g., [29] and [14]). The phase lengths are determined based on requests, which are defined by how many cars are near or passing through an intersection. Upon receiving requests, the control system looks at preset rules and determines whether to change to the next phase or extend the length of the current phase. The difference between actuated and self-organizing systems lies in the request. For the actuated system, if vehicles are approaching the intersection during a green phase, a request is generated, whereas, for the self-organizing system, a request is generated when the number of vehicles approaching an intersection crosses a limit.

5. **Other Optimization Techniques**

   A comparative study on particle swarm optimization and social learning particle swarm optimization (SL-PSO) was conducted in [10]. In SL-PSO, at every iteration, the randomly generated particles update their positions based on the values of the best particles. The authors used an objective function that measures total travel time and the constraints were defined in terms of cycle length, the length of the green phase, and delay. The paper results showed that both optimization techniques performed similarly. Another algorithm was proposed in [52], in which the authors suggested that the duration of the first intersection of a corridor depends on the queue length multiplied with the time taken for vehicles to leave the intersection plus a time loss for acceleration.

Timings for the subsequent intersections are calculated similarly. The offset then is the distance between the intersection divided by the average speed. A greedy algorithm was proposed in [2], which gathers vehicular information at each intersection through mobile GPS, and based on the longest queue, the timing for the green phase is assigned. The system relies heavily on vehicle GPS information which is collected through a mobile application. For a scenario in which no drivers have the mobile application, the system will choose a static cycle length.

2.2.3 Different Types of Objective Functions

The most commonly used objective functions in the literature are delay, travel time and the number of stops (e.g., [37], [58], [62], [65] and [40]). Apart from these, there exist other objective functions, such as throughput and throughput-minus-queue. Throughput is the number of vehicles that have passed through the network and throughput-minus-queue is the difference between the total number of vehicles passing through the network and the sum of the queue lengths at every intersection. Maximizing throughput-minus-queue rather than throughput, can produce better results in over-saturated conditions since it reduces gridlocks in the network (e.g., [25], [43] and [11]).

Different objective functions (delay, travel time, throughput, number of trips, and weighted trips) were compared to find out which was most effective in traffic signal timing optimization [24]. The authors of [24] used different demand patterns (traffic scenarios) on a network and then evaluated 8 performance measures to compare the objective functions. The demand patterns were symmetric under-saturated, symmetric over-saturated, asymmetric under-saturated, and asymmetric partially over-saturated. The paper found that all objective functions performed similarly in the under-saturated scenario, but for the over-saturated scenario, throughput and weighted trip maximization outperformed the others.

A different objective function was explored in [33], in which the authors maximized the proportion of non-stopping vehicles to the total number of vehicles traveling through a corridor of intersections. A linear combination of

this ratio and the average time traveled was considered and using an iterative signal control algorithm, the values for the splits were determined.

2.3 Monte Carlo Tree Search

Tree search algorithms, such as breath-first search and mini-max search are often used in games to select moves. These algorithms explore all branches in the tree to ensure that all possible move variations are checked within the search horizon. However, they require exponential time in the worst case to visit all the branches. In 1987, Bruce Abramson combined mini-max search with the Monte Carlo method (an expected-outcome model based on random game playouts) and found that it significantly improved the search time [1]. Monte Carlo (MC) methods have been initially used in statistical physics and mathematics for optimization and numerical integration. The concept of using randomness to solve deterministic problems (gives the same output for the same initial input) is quite useful in solving higher-dimensional integrals. MC chooses random values to evaluate the integral [64]. The combination of tree search and MC is known as Monte Carlo Tree Search (MCTS) which was introduced by Rémi Coulom in [15]. MCTS has been highly successful as a search algorithm framework. It has been used in many games such as Go, Chess, and Shogi (e.g., [8], [41] and [54]) to create world-class game playing systems.

MCTS has also been applied to single-agent games. One of the earliest examples is the work on SameGame in [50], in which MCTS was able to obtain a high score on standardized tests. This is remarkable because the size of the game tree is approximately $21.1^{62.2}$, where 21.1 is the average branching factor and 62.2 is the average game length. The authors allowed an equal computational budget over multiple runs and created a tree per move rather than a tree per game.

In MCTS, a single iteration consists of four distinct steps: Selection, Expansion, Simulation, and Backpropagation, as shown in Fig. 2.1. In each iteration, MCTS expands the tree and stores the previous search information

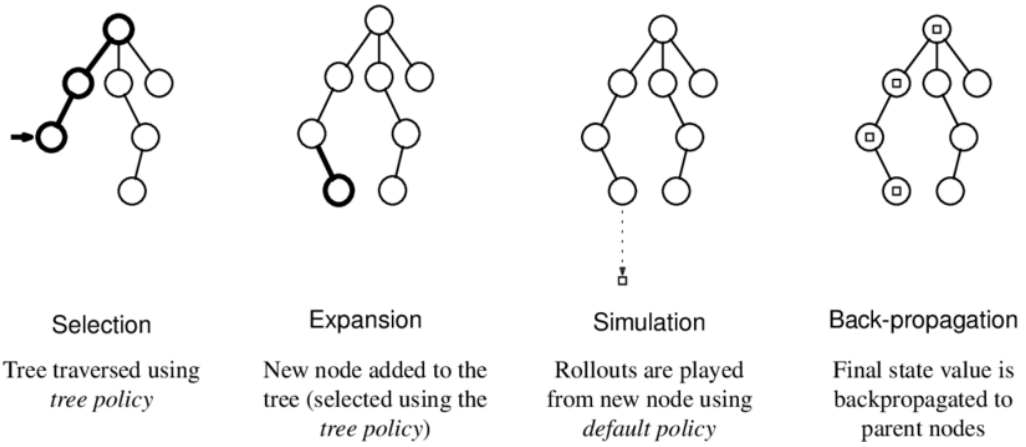


Figure 2.1: Steps of Monte Carlo Tree Search [49]

in nodes and edges. Nodes store search information including the state of the game, whereas edges store search information. With this information, MCTS can choose edges more efficiently and steer the direction of the search towards better actions. This is part of the selection process which is crucial since the tree can grow large depending on the branching factor. Hence, it becomes quite expensive in terms of memory and time to explore branches of the tree that have a low expected reward. Through the selection process, we can ensure a balance between exploitation and exploration of branches. Details about the MCTS phases are discussed next.

2.3.1 Selection

The selection process controls how the tree is traversed. Exploration refers to testing or trying out unseen or seemingly non-optimal possible actions for finding a global optimum. More exploration implies that the tree will grow wider in size, whereas exploitation greedily chooses the action with the best-estimated reward. An exploitation strategy will make the tree grow deeper as we traverse certain edges more often and expand only those edges. If MCTS explores more branches then finding a good solution in a short amount of time may be difficult. However, if MCTS exploits more, then our search is most likely to find sub-optimal solutions as it will not visit other moves.

One problem that closely resembles this predicament is the Multi-Armed

Bandit Problem. In this problem, a gambler is presented with an n-armed slot machine and the gambler has to pull a set number of arms to gain money. The gambler then repeatedly pulls an arm and receives a reward, with the goal of maximizing his overall reward. A popular solution to this is the Upper Confidence Bound (UCB1) algorithm [3]. UCB1 is an optimistic algorithm in which actions are evaluated based on observed performance and future potential. The formula for UCB1 is as follows:

$$UCB1(a) = Q(a) + \sqrt{\frac{2 \log t}{N(a)}} \tag{2.1}$$

where t is the total number of trails so far, $N(a)$ is the number of times actions a has been explored, and $Q(a)$ is the average reward for action a. An action's value decreases the more it is chosen. This encourages the algorithm to explore other actions with higher values. UCB1 requires exploring the actions at least once before selection takes place [42]. It assumes that the rewards gained from each action are between [0,1].

In [28], the authors developed UCT (Upper Confidence Bounds for Trees) algorithm, which is a well-known MCTS variant that is based on UCB1 [3], in which the selection process was biased towards the moves with high rewards. This can be achieved by adding an exploration value to the estimated value, which is the expected reward of an action, and calculated by dividing the total reward gained by playing a move by the total number of times the move has been visited [23]. It associates a bandit problem with each search node. The modified action evaluation equation for UCT is as follows:

$$UCT(s, a) = Q(s, a) + C \sqrt{\frac{\log N(s)}{N(s, a)}} \tag{2.2}$$

Here, $N(s)$ is the number of times the state s has been visited, $N(s, a)$ is the number of times action a has been chosen, and C is a constant which controls the amount of exploration and exploitation. For high values of C, UCT will explore more, and with lower C values UCT will exploit more. Therefore, with a high C value, the UCT value of less-visited edges will surpass the most visited edges and MCTS will be forced to explore those edges. UCT was implemented

in MoGo (a computer program for playing Go), and in 2008, MoGo became a master in 9×9 Go.

Using UCT requires the search to visit all actions in a node at least once. This can be costly if the branching factor is high. To overcome this, a popular type of UCT search called Predictor + UCT (PUCT) was introduced in [48]. Later in 2015, Google Deepmind developed AlphaGo based on PUCT, which was able to defeat a professional Go player on the regular 19×19 board for the first time [53]. In 2017, AlphaGoZero [56], and AlphaZero[55] was introduced, which was able to learn Chess, Shogi, and Go from scratch and bested many world champion programs. The predictor acts as a guide that informs MCTS about how good or how bad an action is. This helps MCTS to divert its search towards better actions and not waste resources while exploring actions that will not yield a good end game reward. The updated formula for PUCT is

$$PUCT(s,a) = Q(s,a) + CP(s,a)\frac{\sqrt{N(s)}}{1+N(s,a)} \tag{2.3}$$

Here, $P(s,a)$ is the predictor value that corresponds to prior probability of an action. It acts a guide and helps to ensure that, initially, MCTS does not explore all actions but rather use the search budget on actions with high PUCT values and low visits. Later on, actions with higher $Q(s,a)$ values are exploited more.

2.3.2 Expansion

Expansion is about how nodes are added to the search tree in each iteration. One strategy is to add a fixed number of nodes in every iteration but this can lead to the tree expanding more rapidly and consuming more memory. Another way is to expand a node only when there have been at least k visits, which ensures some level of confidence in the node. For our experiments, we used a strategy outlined in [59]. The strategy expands a single node in every iteration because our computation is not tree-size bound, but rather simulation-time bound.

2.3.3 Rollouts

Simulations are performed based on rollouts. Rollouts generate sequences of actions that advance the game to a terminal state that can be easily evaluated. Rollouts select actions based on specific policies. For example, in a random policy, actions are selected randomly from a uniform distribution, whereas more informed policies can either be handcrafted or generated by a learning algorithm. At the end of a rollout, the result of the state is calculated and passed back up the tree.

2.3.4 Backpropagation

The results of rollouts are propagated up the tree along the same path traversed in the selection step. During backpropagation, the edge statistics are updated in the following way:

$$N(s,a) \leftarrow N(s,a) + 1 \tag{2.4}$$

$$Q(s,a) \leftarrow (Q(s,a) * (N(s,a) - 1) + R(s,a))/N(s,a) \tag{2.5}$$

Here, we increment the number of visits and we update the total edge reward by adding the reward calculated from the rollout. This is done on every edge up to the root node. The newly updated statistics are then used by the selection step in the next iteration.

2.4 MCTS and Traffic Optimization

Due to MCTS's immense success, many researchers have applied it in different domains including traffic optimization [44]. However, previous work has only used MCTS for a single intersection, whereas we are using MCTS for multiple intersections. It is the tree expansion technique that makes MCTS a great algorithm to make decisions without having to have extensive knowledge on a domain [7]. In [39], several algorithms including Covariance Matrix Adaptation Evolution, Genetic Algorithm, Particle Swarm Optimization, Differential Evolution, Monte Carlo, and Archipelago were compared based on the number of stops and completed trips. The authors found that the evolution method

using the covariance matrix adaptation strategy performed best. In [13], UCT was used in conjunction with the fuel consumption metric to compute actions for connected autonomous vehicles (CAVs). In this work, MCTS decides either to maintain a constant speed or accelerate or decelerate. The decisions are made based on whether a CAV can cross at a traffic light safely or not.

Chapter 3

Signal Optimization via Heuristic Search and Traffic Simulation

This chapter is focused on the formulation of the traffic optimization problem as a single-player game. We also describe the process of incorporating MCTS with a traffic simulator by using a small network as an example.

3.1 Formulating the Traffic Signal Optimization Problem as a Single-player Game

In this section, we define the game state, actions, and terminal state for our game. Games can be categorized into the following: single-player vs. multiplayer, deterministic vs. stochastic, and perfect information vs. imperfect-information, etc. In a traffic network, one or more traffic signals need to make decisions about the length of their next phase. Traffic signal optimization can be viewed as a cooperative multiplayer game in which each traffic light is an individual player, acting by itself in choosing its next phase length. Traffic lights making their own decisions can be regarded as decentralized control. But in our case, we view the problem as a centralized control problem in which a centralized control agent chooses the phase lengths for all signals. This makes coordinated planning between traffic signals much easier, allowing us to model the traffic light optimization problem as a single-agent optimization problem. Furthermore, in the real world, traffic networks are stochastic, as there is al-

ways randomness in car movements. In our simulations, the routes for each car are set before it enters the network. Therefore, the vehicles' destinations are assumed to be known, but the movements of vehicles are stochastic. Therefore, states reached by a given move sequence are unique. This means that our game is a perfect information game, and thus amenable to straightforward MCTS search.

In what follows, we will formalise our single-player traffic game by defining the game state, actions, state transition and terminal state. The game state at time t is defined as follows:

$$
\begin{aligned}
s_t = [\ & \text{current time } t, \\
& \text{all vehicles' positions along their trajectories,} \\
& \text{all vehicles' speeds,} \\
& \text{current phase of all traffic lights,} \\
& \text{remaining duration of each phase}]
\end{aligned}
$$

It is possible to simulate a game by using a game state and a traffic network. The simulator can position the vehicles in the network by using the existing information from the game state and subsequently, apply actions to the state to change signal phases. Applying a move and simulating the game forward is done by the method `make_move(move)`. This method takes a move in the form of $(junction\_id, time)$ and changes the phase of junction $junction\_id$ to the next phase, and assigns the $time$ to this junction. The pseudo-code of `make_move(move)` is described in Alg. 1. The $nextPhase$ of a junction can be found from the sequence of phases assigned to an intersection. So, if the current phase was "red", then the next phase would be "green". If there are no actions to execute then the method would progress the game forward until a traffic light has 0 seconds remaining for its current phase or the game has reached its terminal state.

Algorithm 1 Pseudo-code for make_move

```
function MAKE_MOVE(move)
    junc_id ← move[0]
    time ← move[1]
    if junc_id ≠ −1 then
        apply time to junction junc_id in state s
    end if
    while not is_terminal() do
        if another signal has 0 time remaining then
            break
        end if
        simulate game until a signal is changing or terminal state is reached
    end while
end function
```

A game is simulated forward by moving the vehicles along their paths and reducing the remaining times in all the traffic signals. When a traffic light has 0 seconds remaining for its current phase then the method, `generate_moves()` is called. This method looks at the next phase of the traffic signal and generates a list of actions that can be assigned to that traffic signal. If there are no traffic signals that are changing their phases, then `generate_moves()` returns a special move $(-1, 0)$, which tells `make_move` to simulate forward without applying any signal timing changes. In a scenario, in which two traffic signals have reached 0 seconds at the same time, the following steps are executed. At first, `generate_moves()` generates a list of moves for one of the traffic signals. Then one of the actions from the list is chosen and `make_move(move)` is called, which then assigns the chosen timing to the traffic signal, but will not progress the game forward since there is another traffic light that is at 0 seconds. Therefore, `generate_moves()` is called again to generate a list of actions for that traffic light. Again, a move is chosen from the list for which `make_move(move)` is called. This time, `make_move(move)` assigns the timing to the second traffic light and simulates the game forward. A set of possible actions in the game can be defined as a set of possible timings such as,

$$actionList(s) = \begin{cases} ((-1, 0)) & \text{if no changes} \\ ((j, t)) & \text{if nextPhase}(s,j) = \text{“y”} \\ ((j, t_1), (j, t_2), ..., (j, t_n)) & \text{if nextPhase}(s,j) = (\text{“r”} \, or \, \text{“g”}) \end{cases}$$

where s is the current simulation state, $j \geq 0$ is the id of a junction whose phase is about to change, t and t_i are traffic signal phase timings, and the yellow, red, and green phases are denoted by y, r, and g, respectively. The timing values of the actions can be continuous or discrete. Using a continuous set of timings is preferable as it would be possible to find the best timing. However, MCTS cannot work with an infinite set of actions. Therefore, we opted to use a discrete set of timings. Lastly, the game time T is defined before the game begins. Therefore, we can use the following definition to check whether a game has reached a terminal state or not:

$$isTerminal(s) = \begin{cases} True & \text{if } s.t = T \\ False & \text{otherwise} \end{cases}$$

Once in the course of a simulation, a terminal state is reached, the reward for the entire action sequence up to this point is computed. For example, if the total travel distance is chosen as a reward function, in terminal states the distances driven by all cars in the network up to this point are added. With these definitions, traffic light optimization can now be described as finding a sequence of light phase actions (j_i, t_i) that generate the highest reward when reaching terminal states at time T. Details about the objective function will be discussed in Ch. 4.

3.2 Combining MCTS with SUMO

In this section, we present pseudo-code to illustrate how MCTS works together with a traffic simulator. Looking at Alg. 2, we can see that the Game class, which refers to the game definition presented above, has several functions including `generate_moves()`, `make_move(move)`, `get_result()` and `is_terminal()`. In the beginning, a game object is initialized with a traffic scenario (start_state) and the duration (T) which specifies how long the game would last. After the game starts, we wait for a traffic signal to change its phase. Then we call method `generate_moves()` to give us the list of possible timings for the traffic light that is changing its phase.

Algorithm 2 Pseudo-code of the main program and MCTS

```
function MAIN(start_state)
    main_game ← initialize_game(start_state, T)
    while not main_game.is_terminal() do
        simulate game forward until a traffic signal is changing phase or
        terminal state reached
        moves ← game.generate_moves()
        if length(moves) = 1 then
            game.make_move(moves)
        else
            save current state of the game
            move ← MCTS(saved_state)
            game.make_move(move)
        end if
    end while
end function

function MCTS(saved_state, n)
    mini_game ← initialize_game(saved_state, horizon_time)
    tree ← create root of the search tree
    for i ← 1, n do
        propagate down the tree to reach a leaf node
        moves ← main_game.generate_moves()
        tree.add_edges(moves)
        move ← select an edge
        mini_game.make_move(move)
        while not mini_game.is_terminal() do
            moves ← mini_game.generate_moves()
            select a move based on rollout policy from moves
            mini_game.make_move(move)
        end while
        reward ← mini_game.get_result()
        tree.add_node()
        tree.backpropagate_reward(reward)
    end for
    move ← tree.select_root_move()
    return move
end function
```

For example, if traffic light A is changing its phase from red to green then `generate_moves()` would return the list of possible timings for the green phase. If only one timing is returned then it means that the next phase is yellow. Therefore, instead of calling MCTS, we can assign the timing to the traffic light and move the simulation forward. This is done by method `make_move(move)` that takes timing and assigns it to a traffic signal and moves the game forward until another traffic light is changing its phase. If two traffic lights are changing their phases simultaneously then `make_move(move)` would not simulate the game forward. It would just assign the timing to a traffic light and stop.

If `generate_moves()` returns multiple timings, then the state of the game, s_t is saved and passed on to MCTS, which then starts from this saved state and builds the search tree based on the results obtained from rollouts. The rollout is a step in MCTS in which a sequence of actions is played based on a policy to play the game until a terminal state is reached. Method `get_result()` is called to retrieve the value of the objective function in a terminal state.

To demonstrate how MCTS would work alongside a traffic simulator, we will use a simple network with two intersections. As shown in Fig. 3.1, intersection Jn01, which is about to change from yellow to red, and intersection Jn02 has 14 seconds remaining on its current phase.

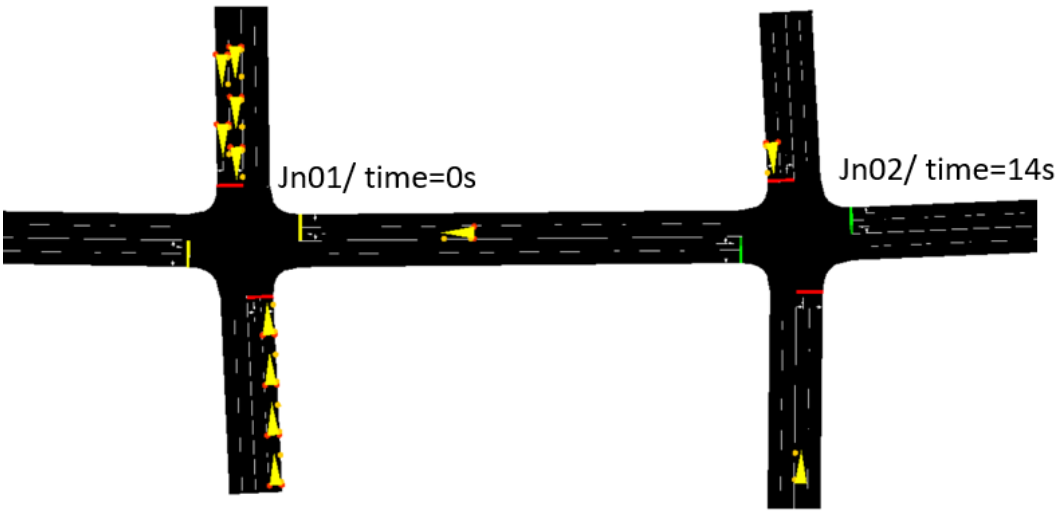


Figure 3.1: Saved state before calling MCTS

This network state is saved and passed on to MCTS to determine a good timing for the next phase for intersection Jn01. According to [29], the minimum length of the green phase for major arterial roads is 10 seconds. For our experiments, we used the following list of timings: 10, 15, 20, and 25 seconds. The reason to choose longer timings such as 20 and 25 is to allow the search algorithm to reduce long queues as quickly as possible. The lower timing values help to achieve synchronization among intersections.

MCTS takes this saved state and creates the search tree. Fig. 3.2 depicts the root node and its edges for MCTS. The root node represents the current state of the game and expanding the root node gives us the possible timings that can be assigned to the next phase for intersection Jn01. Next, we select an edge (10 seconds in this case), and follow this edge and check whether we have reached a leaf node. After that, we perform a rollout, which simulates the game for a set time period (until the simulation reaches 100 seconds in this instance). Once the rollout ends, the final reward is retrieved and back-propagated up the tree. After performing the same steps for the next two edges, we end up with a tree that is shown in Fig. 3.3. Here, we can see that the edge with 10 seconds achieved the best reward. Therefore, we select this edge and apply it to the junction and simulate forward until another junction is changing its phase or the game time is reached.

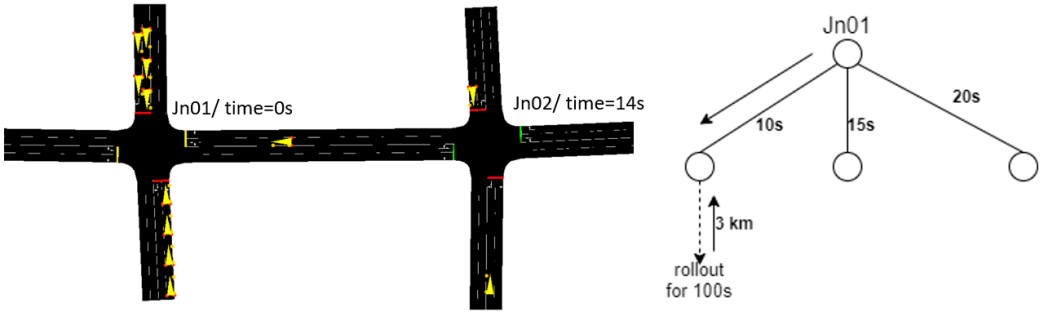


Figure 3.2: Root node and its edges corresponding to the saved state

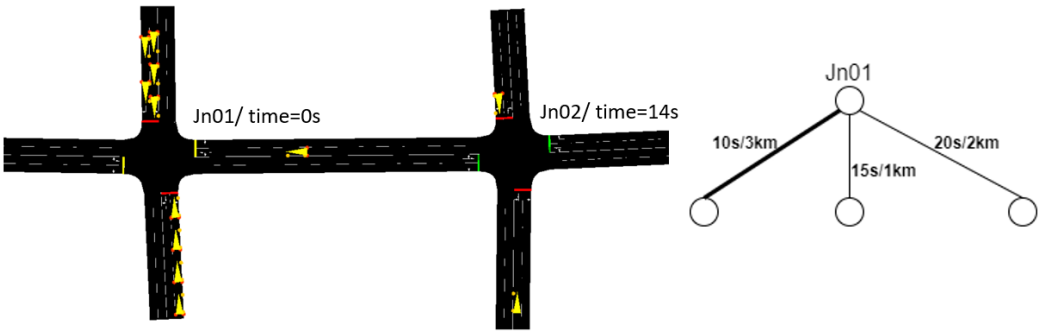


Figure 3.3: Edge rewards after exploring all edges

After simulating the game for 10 seconds, we can see that intersection Jn01 is changing its phase and 4 seconds are left on intersection Jn02. Consequently, expanding the tree adds one edge with the value of 5 seconds since the next phase for intersection Jn01 is yellow as shown in Fig. 3.4. We repeat the steps described above and back-propagate the results up the tree. After the update, we can see that the edge with 20 seconds has the next best reward and we explore and update the tree accordingly. This is illustrated in Fig. 3.5. Here, we choose the edge with value 10 again and move down the tree. After that, we apply this edge value to the intersection and move the simulation forward for 14 seconds.

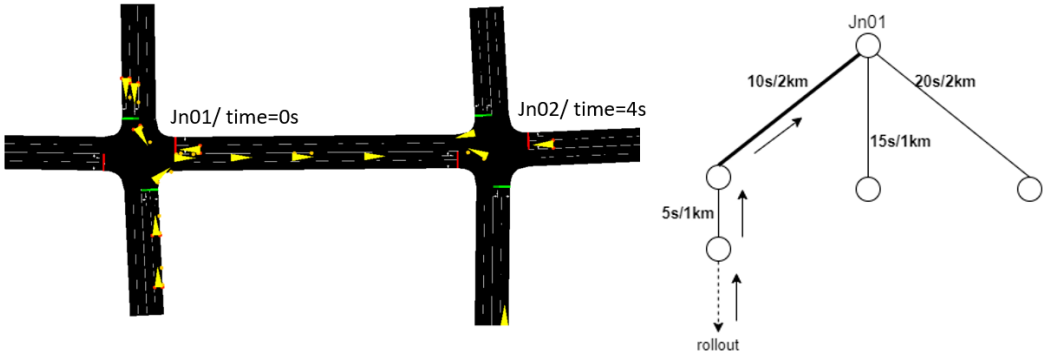


Figure 3.4: Edge rewards after selecting and exploring the first edge

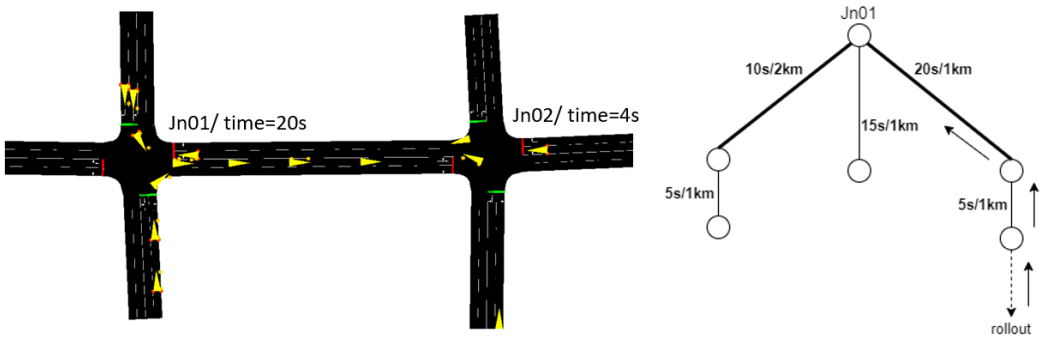


Figure 3.5: Edge rewards after selecting and exploring the third edge

After 14 seconds, intersection Jn02 is changing its phase. Hence, we expand the tree to get three new edges, as shown in Fig. 3.6. The selection, expansion, rollout, and back-propagation are continued until the maximum number of iterations, n is reached. Finally, the edge with the best reward at the root node is returned by MCTS. For instance, choosing the edge with the value 10 gives us the highest distance traveled, 2km.

Information such as vehicles' speed, routes, distance moved, etc. are retrieved from the simulator. We assume that this information is available in the real world through sensors. For instance, a vehicle's speed can be easily calculated using sensors that are embedded in the road. Similarly, a vehicle's position can be determined through cameras. We can also analyze a series of footage from several cameras and calculate the distance traveled by a vehicle. Additionally, routing information is available in the simulator but it can be quite difficult to determine in the real world.

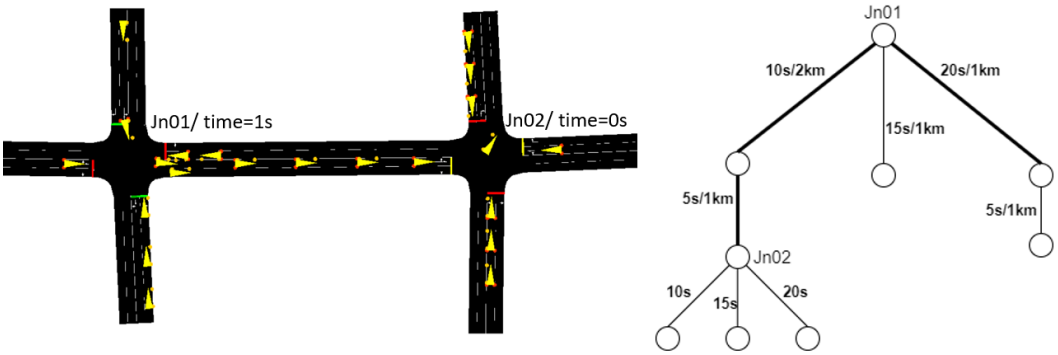


Figure 3.6: Tree expansion when intersection Jn02 is changing phase

However, we can estimate a vehicle's route in the real world by using several techniques. One technique is to use GPS data along with a distribution chart to determine the probability of a car going straight or taking a left or a right turn. Also, if people are using navigation applications such as Google maps, then the route information can be retrieved from these applications.

Chapter 4

Experimental Setup

In this chapter, we will discuss the algorithms, policies, and simulation software that we used in our experiments.

4.1 MCTS Setup for Experiments

UCT requires rewards to be bounded for convergence but in many applications, tight value bounds may not be available. Therefore, normalization methods have been developed. There are many ways to normalize Q values. One way is to divide it by a large number but to come up with a feasible value, one requires domain knowledge or trial and error experiments [51]. Instead, we use the Max-Min Normalization method introduced in [51], in which the selection process uses the following modified PUCT equation:

$$PUCT^*(s,a) = Q_{norm}(s,a) + CP(s,a)\frac{\sqrt{N(s)}}{1+N(s,a)} \tag{4.1}$$

with

$$Q_{norm}(s,a) = \frac{Q(s,a) - \min_{a'} Q(s,a^{'})}{\max_{a'} Q(s,a^{'}) - \min_{a'} Q(s,a^{'})} \tag{4.2}$$

Here, $a^{'}$ iterates over all possible actions that can be taken in state s. In case of ties between two or more actions, we randomly choose one of them. In the Expansion step, we expand a node in every iteration. As for simulation, we used SUMO to generate the rollouts, and 3 minutes was used for the horizon time (the end time of a rollout). For each network, we have different vehicle population policies, which are described in Secs. 4.3.2, 4.3.3 and 4.3.4.

Algorithm 3 Pseudo-code for Rollouts

```
function ROLLOUT(state)
    game.load_state(state)
    while not game.is_terminal() do
        move_list ← game.generate_moves()
        move ← rollout_policy(move_list)
        game.make_move(move)
    end while
    game.get_result()
end function

function ROLLOUT_POLICY(moves)
    return random.choice(moves)
end function
```

Since our simulations take a long time to complete, we could not use longer horizon times. However, within 3 minutes we can at least expect 6 phase changes on different traffic signals as our list of timings are 10, 15, 20, and 25 seconds. Therefore, with 3 minutes MCTS will be able to foresee any changes in traffic flow. The pseudo-code in Alg. 3 illustrates how the rollout method works with a random policy. At first, we load the game state from a leaf node. Then a list of actions is generated from which an action is chosen randomly and applied to the game. This is repeated until a terminal state is reached and the final result is returned. For this research work, we do not have enough data to train an informed rollout policy. Therefore, we either randomly selected actions from a uniform policy or based on hand-crafted policies. Lastly, for backpropagation we utilized equations (2.4) and (2.5).

4.2 SUMO

For simulating traffic movements, we used SUMO (Simulation of Urban MObility) [36]. It was developed by the German Aerospace Center and community users and was made an open-source software in 2001. SUMO can simulate large road networks with different types of vehicles such as cars, buses, LRT, etc. It has a network editor with a Graphical User Interface (GUI) application. The network editor can be used to create, modify or import road networks.

4.2.1 Configuring SUMO

There are four files that are required to simulate a scenario in SUMO. The first file is a configuration file that specifies which network and route file SUMO uses. The configuration file also specifies other optional parameters, such as random seeds or the length of the simulation in seconds. The network file defines the road network in XML format. This file is generated by the network editor after creating a network and saving it. The route file contains the information of all routes created by us (users). Each route is given an ID and a list of edges from the road network that forms a path for vehicles. The last file is a script file that can be used to control the simulation. SUMO is designed to work as a server-client application. It acts as a server that starts the simulation on a TCP network port. SUMO listens on this port for incoming connections. To connect to SUMO and control the simulation, a control interface called Traffic Control Interface (Traci) is used. By using Traci, we can control the simulations step by step. We can pause the simulation at every time step and do calculations when required. Traci also allows us to add vehicles to the network as the simulation proceeds. This is helpful since we can create different traffic flows in the network. For example, we can create heavy traffic in one direction and after a certain time period, we can change the direction of traffic flow. This also means that we can create a scheduled LRT that passes an intersection at regular time intervals.

4.2.2 Computing Objective Function and Performance Metrics

SUMO provides an XML file called tripinfo which contains information about vehicles that have reached their destinations. This file contains values, such as departure and arrival time, duration and length of the journey, number of stops, and any delays during the journey. We can easily parse this file to evaluate our objective function. However, the file does not provide information about vehicles that have not finished their journeys at the end of the simulation but the statistics of such vehicles are of interest as well. We can retrieve this

information by using different methods provided by SUMO. Since our results are based on the values obtained from SUMO, we wanted to know how accurate these values are compared to the real world. In [38], the authors compared the accuracy of SUMO's traffic simulation with the real-world traffic data. They gathered traffic data of Jianghan Zone in Wuhan, China, from the public domain and found that the simulated traffic data was close to that of the real world. For our model, we retrieve the following values from the simulator: the total distance traveled, the total number of stops, the total number of trips completed, and the total delay in the network, which we now describe in more detail in turn.

Total Distance Traveled

The total distance traveled is calculated by adding the distance covered by all vehicles up to time point T. We want to maximize the total distance traveled as this will increase the throughput in the network. Total distance traveled is a suitable objective function for optimizing the traffic flow, since in the real world we want each vehicle to cover as much distance as possible in a given amount of time. It is also correlated with the number of stops, the number of trips completed, and the total delay in the network. Increasing the distance traveled means vehicles will reach their destinations faster with fewer number of stops. As a result, the delay in the network will also decrease. Also, in congested intersections, the rate of flow increases with the increase of total distance traveled [32]. Additionally, we can easily calculate the total distance traveled in the real world using cellular signals and techniques such as handsoff and location update [26]. The total distance traveled is calculated in the following way:

$$total\_distance = \sum_{i=1}^{n} d_{v_i}$$

Here, n is the total number of vehicles in the network, v_i is the i^{th} vehicle, and d_{v_i} is the distance traveled by vehicle i. From SUMO's tripinfo file, we can retrieve the distance traveled by a vehicle that has completed its trip. For vehicles that are still in the network, we can use SUMO functions to get their

distance information.

Total Number of Stops

The total number of stops is the number of times the vehicles have to stop at all intersections. So, we want this value to be low since stopping at multiple intersections increases delay and carbon dioxide (CO_2) emission. We calculated the number of stops rather than the CO_2 emission since in the real world it will be difficult to get an accurate measurement of CO_2 emission from all vehicles. The number of stops can be read from the tripinfo file for the vehicles that have completed their trips. For the vehicles that have not finished their trips, we can check how many times their speed has gone below a threshold. If a vehicle's speed is below this threshold then we consider that the vehicle is coming to a stop. For our experiments, the threshold was $0.1m/s$.

Total Number of Trips

The total number of trips is calculated by counting the number of vehicles that have reached their destinations. We want more trips to be completed since this implies that more cars are exiting the network in the real world. Thus, more number of completed trips means less congestion and more new cars can enter into the network.

Total Delay

Total Delay is the difference between the actual and the ideal time taken to complete a trip. The ideal time is the time that a vehicle would take to complete its trip if there are no traffic signals or other vehicles. In the real world, no one likes to wait for a long period of time or travel slowly because they want to reach their destinations as quickly as possible. There can be situations in which vehicles in minor traffic flows are waiting for a longer period of time compared to the vehicles in major traffic flows. We do not want our algorithm to ignore minor traffic flows and only concentrate on improving major arterial flows. Therefore, measuring this parameter will give us an insight into how much time a vehicle spends waiting in the network.

4.2.3 Driving Policies in SUMO

The default driving policy in SUMO closely resembles real-world driving behavior. For example, cars that are about to take a left turn at an intersection will wait until it is safe to do so. Similarly, cars will only change a lane if there are no cars beside them. According to SUMO's documentation, there are two types of car dynamics used in SUMO: Euler dynamics and ballistic dynamics. In Euler dynamics, the speed of a car is considered to be constant at any given time step and any change in speed is done instantly in the next time step. For example, if the speed of a car is $25m/s$ at time step t and it applies brakes to decelerate to $4m/s$ then at time step $t+1$, the speed of the car will be $4m/s$. For ballistic dynamics, cars accelerate and decelerate over several time steps to come to a complete stop. Therefore, if a car is decelerating at $2m/s^2$ then at time step $t+1$, the speed of the car will be $(25-2)m/s$. We used ballistic dynamics in all of our experiments as it models real-world physics better.

4.2.4 Incorporating MCTS with SUMO

To use SUMO with MCTS, we have to save and load simulation states multiple times. We start our main simulation and proceed forward until a traffic signal is about to change its phase. If the traffic signal is changing to a yellow phase then we do not need to call MCTS as yellow phases have a constant time of 5 seconds. But if the next phase is a red or green phase then we save the current state of the simulation before calling MCTS. The saved state includes car positions and the state of the traffic lights. it is passed on to MCTS to perform a search. In this way, the saved state is used as the starting point for the simulations of every iteration. Furthermore, if we stop the main simulation and let MCTS start a new simulation for every iteration then the system becomes very slow. Because if we use Traci to stop the simulation and start the simulation again, SUMO will reload the entire network. That means it would re-create the entire network which is time-consuming. This was extremely slow and prevented us from running multiple experiments.

4.2.5 Parallel MCTS with SUMO

To speed up the simulation time of our experiments, we implemented parallel MCTS in which we had four individual threads (we refer to them as workers) that would perform rollouts and return the results to a server. The server traversed down the MCTS tree until it reached a leaf node and then it passed the game state at that leaf node to an available worker. While traversing down the tree we incremented a parameter called virtual loss [12]. The virtual loss parameter helps MCTS to avoid traversing the same path multiple times in a concurrent environment. In the selection process of MCTS, the action a with the highest value according to the following equation:

$$a = \underset{a'}{\operatorname{argmax}}(PUCT^*(s, a') - L * Q_{norm}(s, a')) \tag{4.3}$$

However, this implementation had its issues with SUMO, since we require 4 instances of SUMO to be running on the same computer. Unfortunately, SUMO does not allow multiple instances of itself to run on a single computer. Therefore, we ended up using non-parallel MCTS and decided to use a different set of built-in functions of SUMO to speed up our simulations.

4.2.6 Speeding up SUMO Simulations

To prevent SUMO from reloading the entire network in every iteration, we do not stop or close the simulation. Rather we just load the saved state which only changes the vehicles' positions. Furthermore, we keep track of traffic signal states so that we can automatically reset their states along with the simulation state. This ensures that SUMO does not construct the network again, rather only changes the vehicles' positions. We also noticed that increasing MCTS's iterations increased memory usage considerably. Through memory profiling, we found that versions of SUMO prior to v1.9.0 have a memory leakage bug in the function which is responsible for loading a state. Since at the beginning of every iteration a saved state is loaded, running more iterations would increase memory usage. For example, if we use 100 iterations for MCTS, the memory usage was around 4 gigabytes. When we increased the number of iterations to

500, the memory usage increased to approximately 40 gigabytes. This bug was reported to the SUMO team and they were able to fix it in the 1.9.0 version. This finally allowed us to run multiple experiments in parallel using Compute Canada resources.

4.3 Road Networks in SUMO

This section will describe the 3 different types of road networks used in our experiments. Before that, we will also discuss how our model can be implemented in the real world.

4.3.1 Real World Implementation

To implement our model in the real world, we first need to create an accurate representation of the road network. This can be done in two ways: by creating the whole network by hand in the SUMO simulator or by using OpenStreetMap API to export the map of the city and then importing it into SUMO. After that, traffic phase sequences can be defined for each traffic signal in the network. Once the network and the traffic lights are set up, vehicles can be introduced into the network. To match the traffic flow of the real world, data from sensors like cameras can be used to control the density of vehicles in different areas. In the real world, we can not run MCTS when the traffic lights are changing phases. Thus, we have to run MCTS ahead of time. For instance, if the next signal change happens in 15 seconds then we can run MCTS for 15 seconds to decide the timing for the next phase. Once MCTS has decided on the timing, a message can be sent to the signal controller on-site to set the timing for the upcoming phase. So, we can see that this model is easy to implement and parts of it can be swapped to enhance performance. For instance, we can use a different simulator (if needed), use neural network based policies in-tree or in rollouts, or use different objective functions.

The average runtimes of each of the major methods in our model as shown in Table 4.1. We can observe that most methods of MCTS run quickly except for the selection and the rollout methods, which are dependent on the

Table 4.1: Runtimes for different methods

Methods	Average Time
MCTS Selection	19.6 ms
MCTS Rollout	173.6 ms
MCTS Expansion	70 μs
MCTS Backpropagation	20 μs
Game load_state	73.9 ms
Game generate_moves	40 μs
Game make_move	2.3 ms
Game get_results	5.2 ms
MCTS 1 complete iteration	267.3 ms
MCTS with 600 iterations	160 s

game's *make_move* method. The whole selection process calls the make_move method multiple times as it traverses down the tree (multiple actions). The *make_move* method is slow since Traci relies on TCP connections to communicate with SUMO to control the simulation. Also, loading a network state and getting results from SUMO takes a considerable amount of time as both methods need to parse XML files. As far as the memory requirement is concerned, we only require 45 Megabytes to run a simulation, which is quite low. If we can improve the runtimes of loading a state and making a move then it will be possible to implement this model in the real world. Different ways of improving the simulation are discussed in Ch. 6.

4.3.2 Interconnected Intersections

Network 1 consists of 6 intersections in a 2-by-3 grid format as shown in Fig. 4.1. The reason to choose such a network is that it represents a small section of a city with interconnected roads. This network has short and long corridors with heavy traffic flow in a particular direction. The ten edges on the perimeter are used to insert vehicles into the network. Each intersection has four-phase traffic signals. There are two distinct yellow phases, one for vertical direction and the other for horizontal direction. The other phases are alternating between green and red. For instance, if it is green for the vertical direction then it is red for the horizontal direction and vice versa.

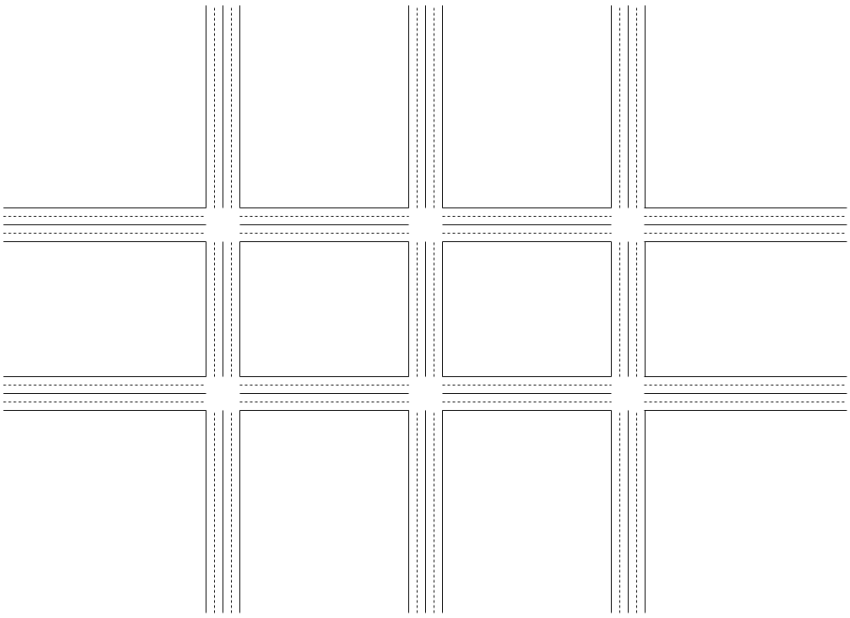
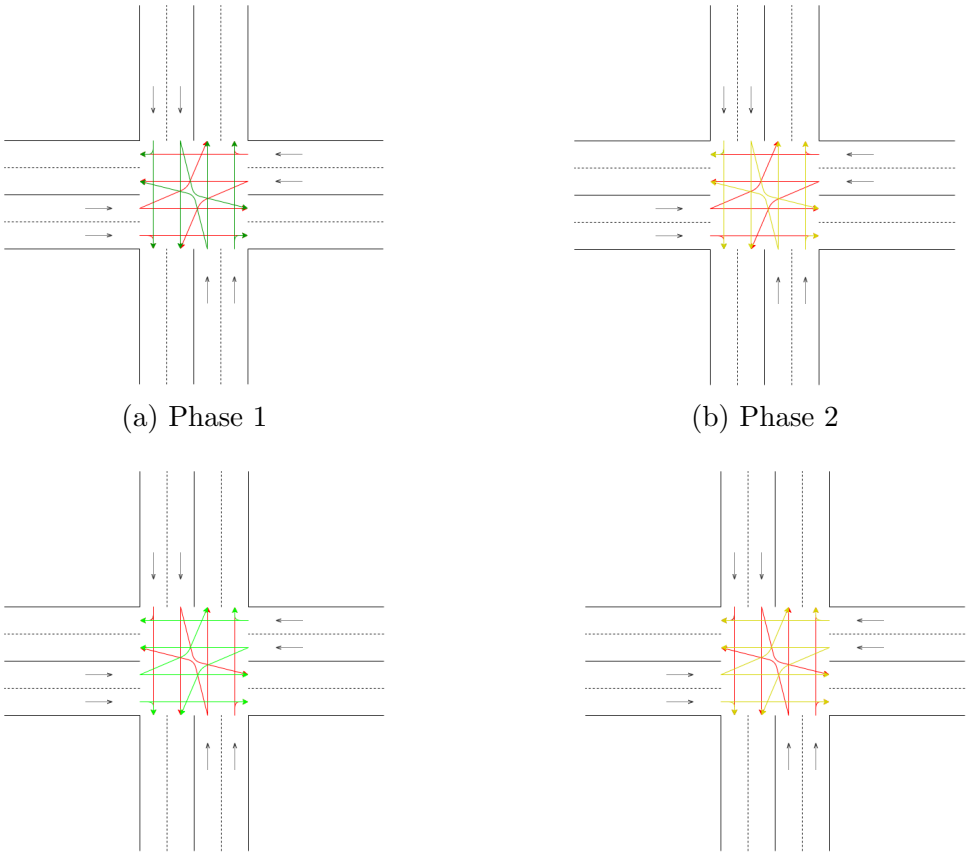


Figure 4.1: Network 1: a road network with 6 intersections

Each phase and the trajectories of cars through an intersection are shown in Fig. 4.2. Fig. 4.2a displays the green phase for the north to south and south to north direction, whereas Fig. 4.2c displays the green phase for the west to east and east to west direction. There are no dedicated left-turn phases in this network. Furthermore, left turns and right turns have lower priorities compared to the other directional flows. Moreover, for this network heavy traffic flow is introduced from the north traveling to the south.

To run tests and compare the results, 106 different starting scenarios were created. At first, we started a simulation with no vehicles and then added cars to the network as the simulation progressed. We let the simulation run for 24 hours (wall-clock time). Then, at every 15 minutes interval, we saved the network state, which included the cars' positions, the traffic lights' phases, and their corresponding duration. Each of these saved states became the starting point of our 106 scenarios.



(a) Phase 1 (b) Phase 2

(c) Phase 3 (d) Phase 4

Figure 4.2: Four signal phases of an intersection

4.3.3 Long Corridor of Intersections

Network 2 contains 4 intersections lined up next to each other, as illustrated in Fig. 4.3. This network represents a long road stretch controlled by several traffic signals. Vehicles traveling through this network will greatly benefit from a green wave. Again, a green wave is achieved when the green phases of several traffic signals are aligned, so that traffic can flow through the signals continuously. We are using this road network to test MCTS's capability to come up with a green wave schedule. The car insertion policy for this network varies as the simulation progresses. We tried to model the rush hours of a real-world scenario with this network. In the real world, we have more cars going downtown in the morning, while in the evening it is the opposite.

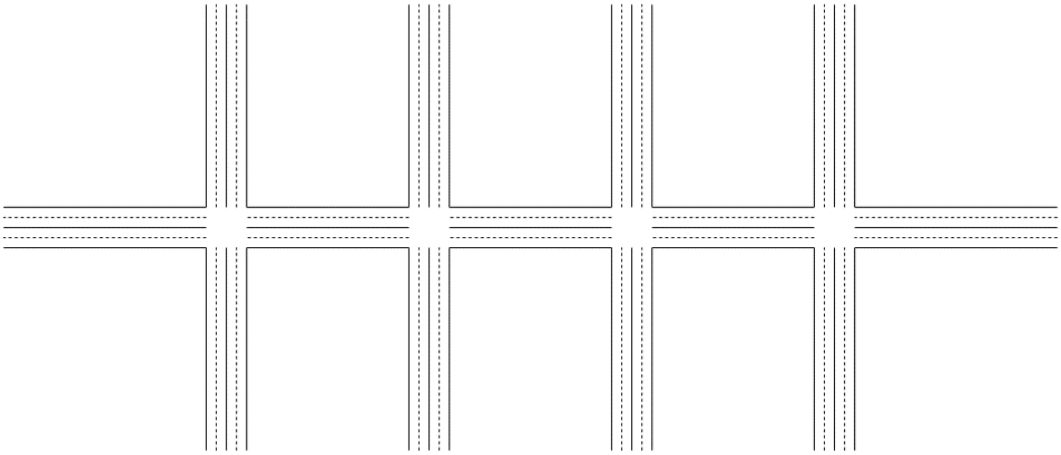


Figure 4.3: Network 2: a corridor of intersections

To model this, we started with heavy traffic movements from the west to east direction and light traffic from the other three directions i.e. south to north, north to south, and east to west. After a while, we switch the heavy traffic flow from east to west and light traffic from east to west (adding more vehicles from east to west and fewer vehicles from west to east). The time interval at which the switch occurs is not constant. By doing this, we can test MCTS on how quickly it can adapt to sudden changes in traffic flow and provide an updated schedule. However, in the real world, there can be situations in which the simulation rollout thinks that the change will happen at time t, when in fact the changes will happen earlier or later. Since the system receives real-time data from sensors, the simulations can be updated quickly. Also, MCTS is being called at every phase change, so there would be at least one phase change at every minute. Therefore, MCTS would be able to readjust the timings according to the updated simulations. In Ch. 6, we will discuss a technique that will be able to address this issue.

4.3.4 Real World Intersections with Railway Crossings

The architecture of the final network is taken from a real-world location. We wanted to test how viable MCTS's solutions are in a traffic scenario with LRTs. Network 3 (depicted in Fig. 4.4) contains two parallel roads with LRT tracks running between them.

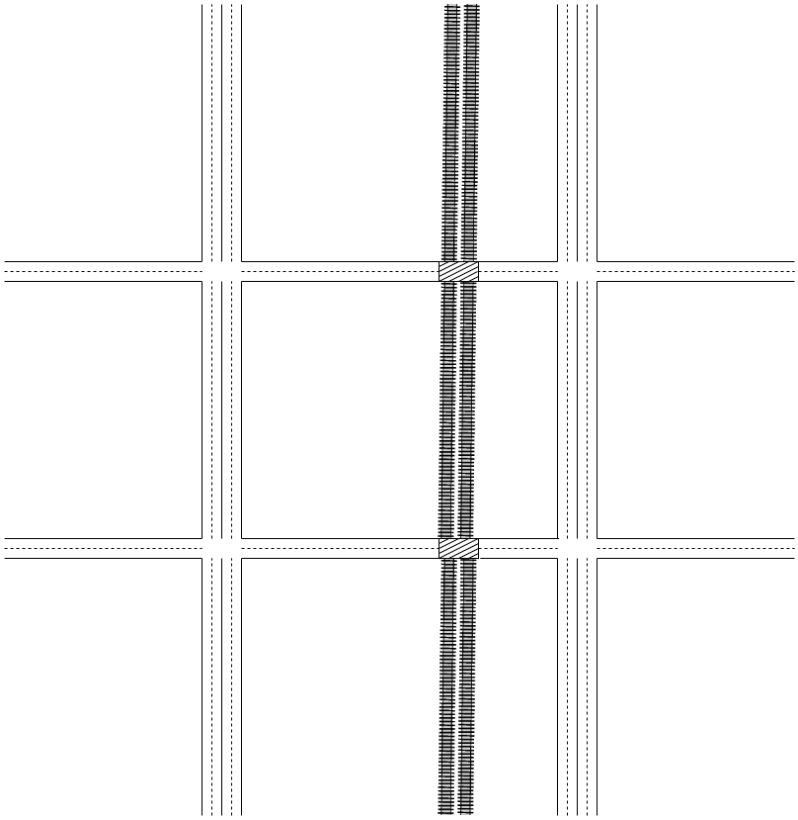


Figure 4.4: Network 3: a road network with LRT crossings

There are two rail crossings where cars can cross the LRT tracks to move from east to west or vice versa. Vehicles are added in a balanced way in all directions. The objective of this network is to test MCTS's capability to foresee the movement of LRTs and schedule the traffic lights in such a way that the movements of traffic from either east to west or west to east do not coincide with LRTs passing through intersections. This network represents the intersections at 111 St. near the Southgate Mall in Edmonton. We did a field study to record the traffic signal's phases and their corresponding timings of each intersection and implemented it in SUMO. This network acts as a baseline to test how adaptable MCTS is in the real world.

In this chapter, we have discussed the experimental setup for MCTS and how it is used with SUMO. We also described different road networks that were used in the experiments. The next chapter will describe and discuss the experimental results we obtained.

Chapter 5

Experimental Results

In this chapter, we discuss the results yielded from the previously described experiments. We have compared MCTS with handwritten policies for all performance metrics. We also performed experiments to optimize the parameters of MCTS such as the number of iterations, the exploration constant, and the rollout policy. Finding a globally optimal parameter vector is a hard combinatorial problem in this domain, which doesn't offer much analytical help, such as gradients. We, therefore, resort to a simple, but popular hill-climbing method, that locally optimizes a single parameter, while holding the others fixed, and repeating the process. We are aware that this approach can only approximate good solutions, and we'll leave finding better optimization methods to future research. For our experiments, we began optimizing the C value by setting the number of iterations to 200 and using a random rollout policy. The reason we decided to use 200 iterations is that the experiments would finish in a reasonable amount of time. Also, we were not sure how MCTS would behave with the other rollout policies and therefore, opted to use the random rollout policy. We found that there was a huge performance gain by using $C = 1$ to $C = 2$. After that, the performance gains from higher C values were not significant. Since our experiments take a long time to complete (roughly 7 to 8 hours for one experiment), we decided to optimize our number of iterations first. This helped us to find the highest number of iterations that we could perform in a reasonable amount of time. Then we found the best value for the exploration constant for that given number of iterations and explored

different rollout policies to maximize the performance of MCTS.

5.1 Optimizing the Number of Iterations for MCTS

The first experiments were used to decide the maximum number of iterations that will be used in MCTS. For this experiment, we used the 106 starting scenarios for Network 1 whose generations are described in Section 4.3.2. Fig. 5.1 and Fig. 5.2 show two examples of these scenarios. We performed 9 sets of experiments in which the number of iterations was varied from 50 to 800. In addition, the exploration constant, C was set to 2 and the uniform random rollout policy was used. Each scenario was simulated for 10 minutes and the total distance traveled in the network was calculated. We used total distance for comparison because it was our objective function and it has a positive relationship with the other performance metrics. For example, higher distance traveled means vehicles are less likely to sit at intersections for a long time. This means less total delay, less number of stops. The results for comparing the different number of iterations are shown in Table 5.1.

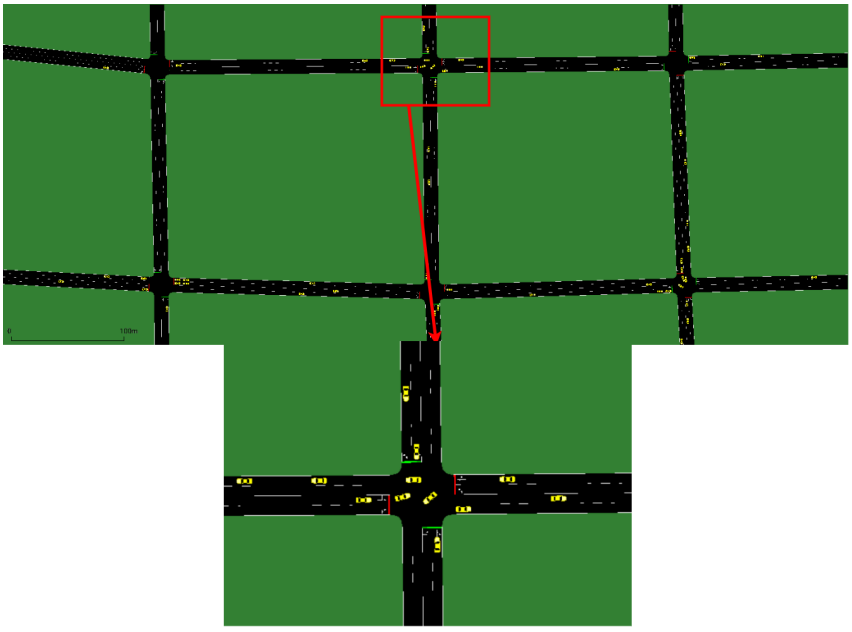


Figure 5.1: The first of the 106 starting scenario with the top-middle intersection zoomed

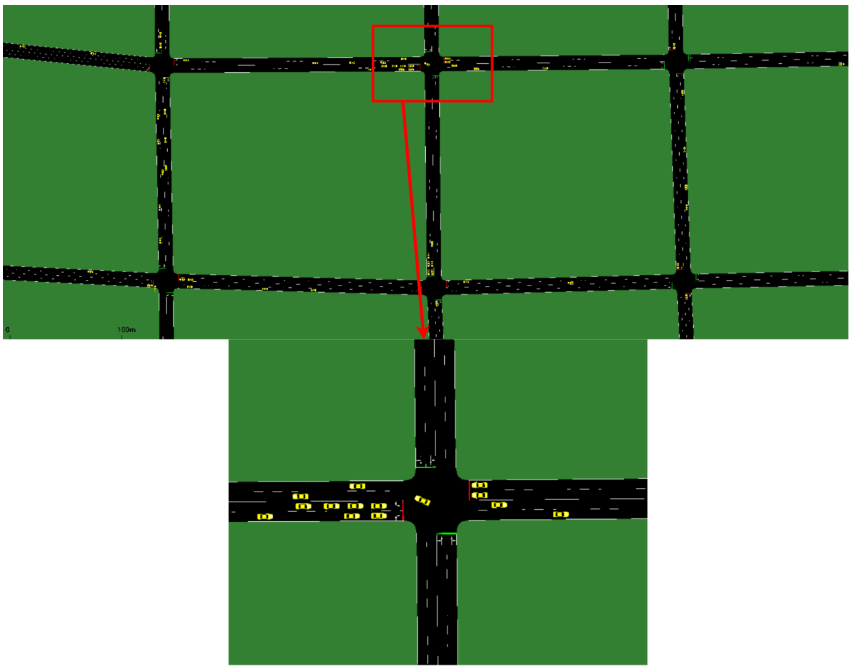


Figure 5.2: The second of the 106 starting scenario with the top-middle intersection zoomed

Table 5.1: Total distance traveled in the network for different MCTS iterations and the random baseline policy

Iterations	**Avg. Total Distance (m)**	**Standard Error**
50	$3.131 \cdot 10^5$	$3.20 \cdot 10^3$
100	$3.156 \cdot 10^5$	$3.20 \cdot 10^3$
200	$3.191 \cdot 10^5$	$3.20 \cdot 10^3$
300	$3.203 \cdot 10^5$	$3.30 \cdot 10^3$
400	$3.208 \cdot 10^5$	$3.20 \cdot 10^3$
500	$3.212 \cdot 10^5$	$3.30 \cdot 10^3$
600	$3.216 \cdot 10^5$	$3.30 \cdot 10^3$
700	$3.215 \cdot 10^5$	$3.20 \cdot 10^3$
800	$3.215 \cdot 10^5$	$3.20 \cdot 10^3$
Random	$2.447 \cdot 10^5$	$2.40 \cdot 10^4$

We used a statistical significance test to decide whether the results are significantly different or not. A popular parametric test for performance comparisons is the paired t-test [16], which helps to decide whether there is a significant difference between the mean of two populations. A null hypothesis states that parameters of population, such as the mean or the standard

deviation are equal to a hypothesized value, whereas the alternative Hypothesis states the opposite. There are a few assumptions that a dataset needs to meet for the paired t-test to be viable. The average value pair differences are normally distributed and the data are observed independently. Because we only use 106 samples, we opted to use Wilcoxon signed-rank test [9], which is a non-parametric test that can be used as an alternative to the paired t-test which is more robust since it works with medians rather than averages. When the Wilcoxon test is performed on the results between any two number of iterations, it returns a p-value. If the p-value is less than 0.05, i.e. the 5% significance level, then we accept the alternate hypothesis, or else we accept the null hypothesis. We used 0.05 because it is the most commonly used default cutoff value [46].

Using the Wilcoxon test on the above results showed that there is a significant difference between the results except if the number of iterations is between 600 and 700. It is to be noted that the graph in Fig. 5.3 does not indicate statistical significance but is mainly used to illustrate graphically trends in the data and that the paired tests are more powerful because they consider result pairs.

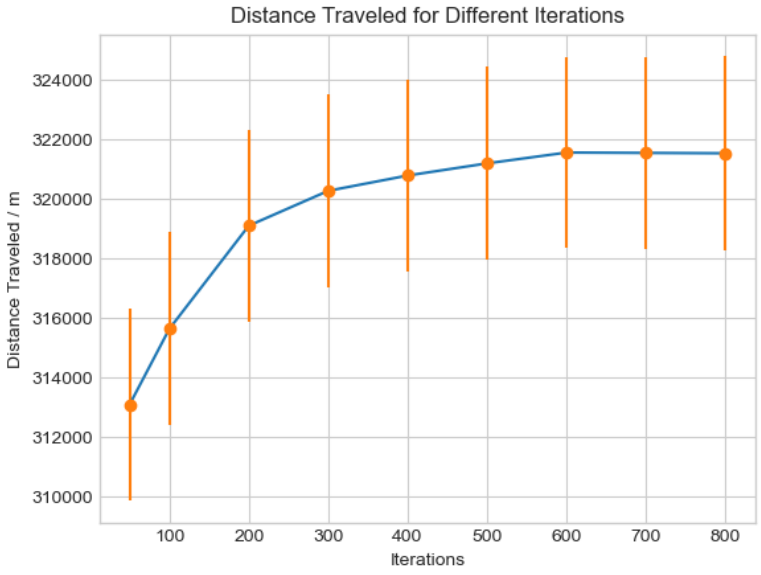


Figure 5.3: Total distance traveled in the network for different iterations

The orange bars in the graph represent the measured standard errors. From the graph, we can observe that the total distance traveled increases with more iterations. After 600 iterations the graph levels out. Since the highest result was achieved using 600 iterations which is marked in red in Table 5.1 and increasing the number of iterations further would result in a diminishing return, we decided to use 600 as the number of iterations for MCTS for the following experiments. In addition, we can also observe that for any number of iterations, MCTS defeats the random move selector.

5.2 Optimizing the Exploration Constant for MCTS

The second set of experiments were performed to find an optimal value for the exploration constant C that is used in the PUCT selection step. The objective function that was maximized was the total distance traveled by all vehicles in the network. There were 600 iterations performed by MCTS with 10 minutes of game time allocated to each iteration. We used Network 1 and the traffic policy described in Sec. 4.3.2. The value of C ranges from 0 to 11. The average result of the 106 experiments for each C is shown in Table 5.2. We found that there was a significant difference between the results of $C = 9$ and $C = 10$. After performing the Wilcoxon test on the results obtained from using C values 9 and 10, the p-value was 0.0011. Hence, there is a significant difference between the mean values for these cases. Since the mean value of $C = 9$ is the highest in comparison with all other C values as shown in Fig. 5.4, we fixed $C = 9$ for our subsequent experiments.

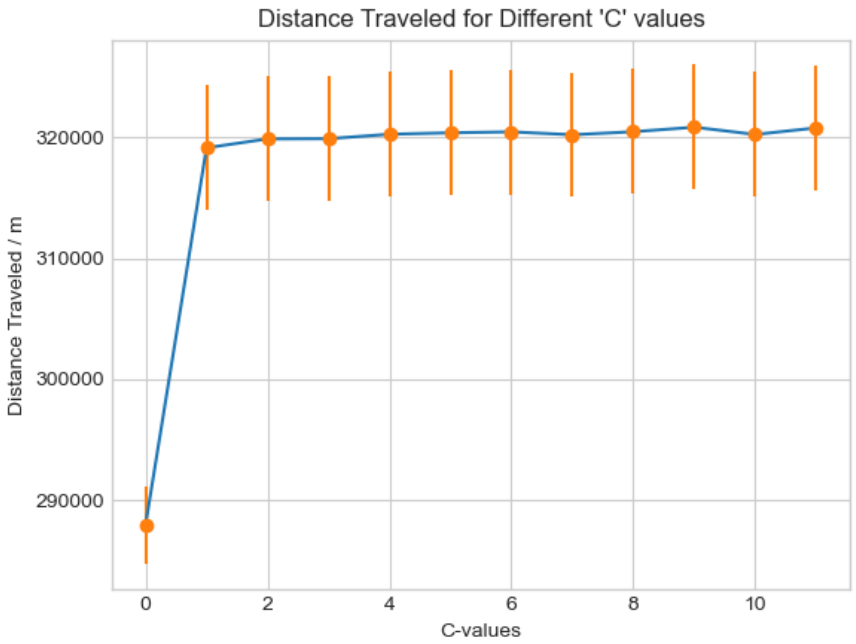


Figure 5.4: Average total distance traveled in the network for different C values

Table 5.2: Total distance traveled in the network for different C values together with their standard errors

C	Avg. Total Distance (m)	Standard Error
0	$2.880 \cdot 10^5$	$3.20 \cdot 10^3$
1	$3.191 \cdot 10^5$	$5.20 \cdot 10^3$
2	$3.199 \cdot 10^5$	$5.20 \cdot 10^3$
3	$3.199 \cdot 10^5$	$5.20 \cdot 10^3$
4	$3.203 \cdot 10^5$	$5.20 \cdot 10^3$
5	$3.204 \cdot 10^5$	$5.20 \cdot 10^3$
6	$3.205 \cdot 10^5$	$5.20 \cdot 10^3$
7	$3.202 \cdot 10^5$	$5.30 \cdot 10^3$
8	$3.205 \cdot 10^5$	$5.30 \cdot 10^3$
9	$3.208 \cdot 10^5$	$5.20 \cdot 10^3$
10	$3.202 \cdot 10^5$	$5.10 \cdot 10^3$
11	$3.207 \cdot 10^5$	$5.10 \cdot 10^3$
Random	$2.447 \cdot 10^5$	$2.40 \cdot 10^4$

5.3 Selecting a Rollout Policy for MCTS

To decide which rollout policy would give the best result for MCTS, we experimented with 5 different rollout policies on the 106 different starting scenarios of Network 1 with the same setting as mentioned in Sec. 4.3.2.

1. **Random Rollout Policy**: In this policy, the actions or timings are selected randomly from $[10, 15, 20, 25]$. All actions have an equal chance of being chosen.

2. **Psychic with Sloping Distribution Rollout Policy**: In this policy, the actions are ordered according to the flow of traffic and the direction of the green phase. The pseudo-code for Psychic with Sloping Distribution Policy is shown in Alg. 4. For example, if there is heavy traffic from the north direction and the next phase is green for the north direction then the timing order for the green phase is $[25, 20, 15, 10]$, and the probability distribution will be $\{25 : 0.4, 20 : 0.3, 15 : 0.2, 10 : 0.1\}$. The reason behind calling the policy Psychic with Sloping Distribution Policy is that the policy knows about the traffic flow and the probability distribution decreases monotonically.

3. **Psychic with Step Distribution Rollout Policy**: Similar to Psychic with Sloping Distribution Policy, here the timings are sorted accordingly

Algorithm 4 Pseudo-code for Psychic with Sloping Distribution Policy

```
function SLOPE_ROLLOUT_POLICY(move_list)
    if move_list.length = 1 then
        return move_list[0]
    end if
    if traffic flow direction = direction of green phase then
        sort move_list in descending order
    else
        sort move_list in ascending order
    end if
    return random.choice(move_list, [0.4,0.3,0.2,0.1])
end function
```

to the flow of traffic and the direction of the green phase. But the probability distribution is set in the following manner, $\{25 : 0.35, 20 : 0.35, 15 : 0.15, 10 : 0.15\}$. The reason we called it Psychic with Step Distribution Policy is that if the probability distribution is plotted then it will give us a step graph. The pseudo-code for Psychic with Step Distribution Policy is shown in Alg. 5.

4. **Psychic with Best-Two Rollout Policy**: Again, the timings are sorted according to the flow of traffic and the direction of the green phase but only the top two timings are used. The pseudo-code for Psychic with Best-Two Distribution Policy is shown in Alg. 6. For example, if the sorted timings were, $[25, 20, 15, 10]$ then one of the top two best timings is chosen randomly, i.e., *random.choice*($[25, 20]$). The reason for using the best two timings instead of just the best timing is to give MCTS the ability to create offsets among adjacent traffic signals to achieve a green wave.

Algorithm 5 Pseudo-code for Psychic with Step Distribution Policy

```
function STEP_ROLLOUT_POLICY(move_list)
    if move_list.length = 1 then
        return move_list[0]
    end if
    if traffic flow direction = direction of green phase then
        sort move_list in descending order
    else
        sort move_list in ascending order
    end if
    return random.choice(move_list, [0.35,0.35,0.15,0.15])
end function
```

Algorithm 6 Pseudo-code for Psychic with Best-two Distribution Policy

```
function BEST_ROLLOUT_POLICY(move_list)
    if move_list.length = 1 then
        return move_list[0]
    end if
    if traffic flow direction = direction of green phase then
        sort move_list in descending order
    else
        sort move_list in ascending order
    end if
    return random.choice(move_list[0:2])
end function
```

5. **Handwritten Rollout Policy**: Our handwritten policy tries to create a green wave along the major traffic flow direction as quickly as possible. To create our handwritten timing schedule, we had to observe several simulations and try out different timing combinations until we were able to maximize our objective function. Our handwritten policies surpass random move selectors in all networks we considered. In the real world, engineers would do something similar: they would observe traffic patterns through an intersection and decide which timing combinations would provide better traffic flow. This process of handcrafting timing schedules is quite a time intensive. The pseudo-code for the handwritten policies for each road network will be illustrated in Secs. 5.4.1..3.

At first, we wanted to observe how the policies perform by themselves, without using MCTS. Each policy was run on the 106 different starting scenarios of Network 1 for 10 minutes and different performance metrics were recorded, such as total distance traveled, the total number of stops, total number of trips completed, and total delay in the network. The results in Table 5.3 show that the handwritten policy outperforms all other policies. Moreover, combining MCTS with all the policies improved their performances by at least 26%. As for MCTS, we used 600 iterations and the value of C was 9. For each policy, the performance values are shown in Table 5.4. Please note that the standard errors are shown in parenthesis. We can observe that the informed policies performed worse than the random policy, which is odd. In [22], the authors

Table 5.3: Comparison between different Rollout Policies without MCTS with standard errors in parentheses

Policies	Total Distance (m)	Stops	Trips	Delays (s)
Random	$2.447 \cdot 10^5$ $(2.38 \cdot 10^4)$	488	389	$5.03 \cdot 10^5$
Sloping	$2.456 \cdot 10^5$ $(3.44 \cdot 10^3)$	487	392	$5.05 \cdot 10^5$
Step	$2.448 \cdot 10^5$ $(3.87 \cdot 10^3)$	488	385	$5.07 \cdot 10^5$
Best-Two	$2.460 \cdot 10^5$ $(4.53 \cdot 10^3)$	484	398	$5.08 \cdot 10^5$
Handwritten	$2.495 \cdot 10^5$ $(3.20 \cdot 10^3)$	478	456	$4.69 \cdot 10^3$

Table 5.4: Comparison between different Rollout Policies with MCTS with standard errors in parentheses

Policies	Total Distance (m)	Stops	Trips	Delays (s)
Random	$3.222 \cdot 10^5$ $(3.21 \cdot 10^3)$	311	486	$3.49 \cdot 10^5$
Sloping	$3.209 \cdot 10^5$ $(3.20 \cdot 10^3)$	304	484	$3.58 \cdot 10^5$
Step	$3.218 \cdot 10^5$ $(3.21 \cdot 10^3)$	314	485	$3.54 \cdot 10^5$
Best-Two	$3.208 \cdot 10^5$ $(3.20 \cdot 10^3)$	299	484	$3.64 \cdot 10^5$
Handwritten	$3.136 \cdot 10^5$ $(3.20 \cdot 10^3)$	352	470	$3.92 \cdot 10^5$

encountered something similar, in which their heavily informed policies for UCT performed worse than the default policy of MoGo. They theorized that each policy might be biased in its way, which can lead to a different distribution of simulations in MCTS. If these distributions are skewed towards different outcomes then these policies can worsen the performance of the search. The authors of [27], investigated this phenomenon by plotting the expected values of random, biased, and inversely-biased policies while optimizing an objective function. Their plots showed that random rollouts were able to better represent the objective function when compared to the other policies.

After performing the Wilcoxon-signed-rank test on the total distance traveled results, we found that all performance values are significantly different from one another except for the Psychic with Sloping Distribution and the Psychic with Best-Two policies. Those policies are similar to one another. In the Psychic with Best-Two rollout policy, there is an equal chance of choosing the first timing and in the Psychic with Sloping Distribution, the first timing has the highest probability of being chosen. Therefore, both policies might be

choosing the first available timing frequently. The Random Policy has done better than the other policies according to the total distance traveled, the number of trips, and the total delays. The reason behind this could be that choosing timings randomly encourages more exploration because at the end of every MCTS iteration the result of the rollout is backpropagated. Therefore, if the move selection for rollout only varies between two actions then the end result will not differ by much. This means it might be harder for MCTS to distinguish between a good move and a bad move during the selection process. As for the handwritten policy, we speculate that the reason why it did not perform well is that it only focuses on the main traffic flow direction. According to the results shown in Table 5.4, we used the Random Rollout Policy for our experiments.

5.4 MCTS vs. Handwritten Time Schedules

5.4.1 Interconnected Intersections

After establishing the number of iterations, the C value, and the rollout policy, we want to compare how MCTS does against a handcrafted policy, Alg. 7, on Network 1. The handwritten policy starts by assigning a timing and a phase to each intersection. Since there is heavy traffic flow from north to south, an offset (offset_1) is used to calculate the time when the north traffic signals will be turning green. As the vehicles approach the signals, they turn green and stay green for as long as possible. The south traffic signals turn green when these sets of cars are approaching it. The timing that the south-side traffic signals need to turn green can be calculated by adding another offset (offset_2) to the timing of the north signal's green phase. The offset value was found out by dividing the distance of the road by the speed of the vehicles. As for MCTS, we used the C value of 9 and the Random Rollout Policy with 600 iterations. The goal of MCTS was to maximize the objective function which is the total distance traveled in the network. Again, each iteration was given 10 minutes of (in-game time) simulation time. The average results of the 106 experiments are shown in Table 5.5.

Table 5.5: Comparison between MCTS and Handwritten Policy on Network 1 with standard errors in parentheses

Methods	Total Distance (m)	Stops	Trips	Delays (s)
MCTS	$3.223 \cdot 10^5$ $(3.22 \cdot 10^3)$	311	486	$0.349 \cdot 10^6$
Handwritten	$2.495 \cdot 10^5$ $(3.20 \cdot 10^3)$	478	456	$0.469 \cdot 10^6$

Algorithm 7 Pseudo-code for Handwritten Policy for Network 1

```
function HANDWRITTEN_POLICY(junc_to_change, move_list)
    juncID ← junc_to_change.id
    if juncID is a north junction then
        if green phase for major traffic flow then
            return max(move_list) + offset_1
        else if red phase for major traffic flow then
            return min(move_list)
        end if
    else if juncID is a south junction then
        if green phase for major traffic flow then
            time ← north junction's green_phase timing
            return time + offset_2
        else if red phase for major traffic flow then
            return min(move_list)
        end if
    end if
end function
```

We can observe from Table 5.5 that MCTS was able to beat out handwritten policy in total distance, number of stops, number of trips, and total delay by $29\%, 37\%, 45\%$ and 39%, respectively. One reason why the handwritten policy did not perform well is that it tries to achieve green waves only in the major traffic flow (north to south), whereas MCTS might be able to discover better timing combinations that allow it to achieve green waves not only in the north-south direction but also in the east-west direction. To further investigate this, we calculated the total number of times the cars stopped in the east-west direction for both MCTS and the handwritten policy. For our handwritten policy, the total number of stops in the east-west directions was 169 and for MCTS it was 99. This indicates that just using MCTS with a random policy yields a better result than studying traffic patterns to come

up with a simple timing schedule that only focuses on the major traffic flow direction.

5.4.2 Long Corridor of Intersections

We also wanted to see how MCTS performs if there is a sudden change in traffic. For this, we are considering a real-world rush hour scenario in which at a certain time of the day there is more traffic in one direction, and then after some time the flow changes in the other direction. For example, in the morning there is more traffic flowing towards downtown and in the evening the traffic flows away from downtown. To model such a scenario, we used Network 2 shown in Fig. 4.3. In this network, heavy traffic starts from the north and travels down towards the south. After some time, the flow is reversed with more traffic coming from the south and traveling towards the north. The time interval for switching the direction of heavy flow is not constant. We change the direction of the heavy flow by adding more cars in that opposite direction. The reason for using non-constant time intervals is to observe whether MCTS is using simulations to detect changes in traffic flow and choose the current signal phase length in such a way that it will be easier to accommodate future traffic changes. Through this experiment, we are trying to find out whether MCTS just prioritizes the major traffic flow or it also takes other minor (east-west) traffic flows into consideration. Again, we designed a handwritten policy that tries to align the green wave in the north and south directions depending on the traffic flow, according to the pseudo-code shown in Alg. 8. We ran MCTS to maximize the total distance traveled.

Algorithm 8 Pseudo-code for Handwritten Policy for Network 2

```
function HANDWRITTEN_POLICY(junc_to_change)
    juncID ← junc_to_change.id
    # junctions are number from top to bottom
    if heavy traffic flow from north to south then
        move_list ← get_north_bound_timings(juncID)
        offsets ← get_north_bound_offsets(juncID)
    else if heavy traffic flow from south to north then
        move_list ← get_south_bound_timings(juncID)
        offsets ← get_south_bound_offsets(juncID)
    end if
    if green phase for major traffic flow then
        return max(move_list) + offset
    else if red phase for major traffic flow then
        return min(move_list)
    end if
end function

function GET_NORTH_BOUND_TIMINGS(junction)
    return pre-defined set of timing for junction
end function

function GET_SOUTH_BOUND_TIMINGS(junction)
    return pre-defined set of timing for junction
end function

function GET_NORTH_OFFSETS(junction)
    return pre-defined offset for junction
end function

function GET_SOUTH_OFFSETS(junction)
    return pre-defined offset for junction
end function
```

The results are listed in Table 5.6, which shows that MCTS was able to perform better than our handwritten policy but only by 2% in terms of total distance driven. If we look at the other performance metrics, MCTS was able to increase the number of trips by 3% and reduce the total delay in the network by 7%. But MCTS failed to reduce the total number of stops in the network. This can indicate that MCTS was able to achieve a higher total distance traveled by allowing more cars to travel in the east-west direction. This also explains why MCTS was able to achieve more trips and reduced delays since the routes in the east-west directions are shorter. But this number is not large since the difference in the number of trips is only 3%. Therefore, this reinforces our previous claim that rather than spending time and resources to handcraft traffic schedules, we can use MCTS instead to achieve better or similarly good performance.

5.4.3 Real World Intersections with Railway Crossings

In our final experiment, we wanted to measure the performance of MCTS with respect to a timing schedule used at a complex real-world intersection that contains LRT tracks. In this experiment, MCTS can exploit the information about the LRT's timing schedule through simulations, whereas it can be hard to incorporate this information in the handwritten policy. The intersection we choose corresponds to 111 street near Southgate LRT station in Edmonton. The road network for this particular intersection is shown in Fig. 4.4 in Sec. 4.3.4. This intersection contains 4 traffic signals with a rail track between two major traffic roads. There are two railway crossings that are controlled by LRT traffic. To acquire the timing schedules of these traffic signals, we observed the intersections for half an hour and recorded the phase

Table 5.6: Comparison between MCTS and Handwritten Policy on Network 2

Methods	**Total Distance (m)**	**Stops**	**Trips**	**Delays (s)**
MCTS	$1.186 \cdot 10^5$ $(2.30 \cdot 10^3)$	579	609	$0.810 \cdot 10^6$
Handwritten	$1.161 \cdot 10^5$ $(2.12 \cdot 10^3)$	509	591	$0.871 \cdot 10^6$
Random	$1.045 \cdot 10^5$ $(2.26 \cdot 10^4)$	683	531	$1.243 \cdot 10^6$

sequences and their corresponding lengths. We noticed that the top right and the bottom left intersections had the same phase sequences and phase lengths. We will refer to both of these intersections as Group A. Similarly, the top left and the bottom right had the same phase lengths and sequences and they will be referred to as Group B.

Group A has three main green phases, in which the first phase is for the main arterial traffic in the north-south directions. The second phase controls traffic in the east-west directions, and finally, these intersections have a dedicated phase for left turns from the north-south directions. As for Group B, it has two green phases, one for the north-south directions and the other for the east-west directions. The timings used for Group A are 60 seconds for the first phase, 40 seconds for the second phase, and 15 seconds for the third phase. For Group B, the timings are 45 seconds and 20 seconds for the first and second phases, respectively. Whenever the LRT gates are closed, the first phase of the two groups is active. As soon as the LRT leaves, the dedicated left-turn phase of Group A and the second phase of Group B are activated. After that, the third phase of Group A is activated and the cycle repeats. We recreated the scenario in SUMO and scheduled the LRT to pass the intersections at irregular intervals. We simulated for 15 minutes (in-game time) with two different sets of possible timings. One set contains the timings obtained from the field research, and the other set of timings ranged from 10 to 25 seconds. The results obtained from all three experiments are shown in Table 5.7.

As we can see, MCTS in both cases performs better than the city's timing

Table 5.7: Comparison between MCTS and Handwritten Policy on Network 3

Schemes	Methods	Total Distance (m)	Stops	Trips	Delays (s)
City's timings	**MCTS**	$1.101 \cdot 10^5$ $(3.12 \cdot 10^3)$	593	720	$2.505 \cdot 10^6$
	City's schedule	$0.927 \cdot 10^3$ $(2.93 \cdot 10^3)$	493	601	$6.117 \cdot 10^6$
	Random	$0.911 \cdot 10^5$ $(3.81 \cdot 10^4)$	509	589	$3.573 \cdot 10^6$
Our timings	**MCTS**	$1.266 \cdot 10^5$ $(3.20 \cdot 10^3)$	627	828	$1.424 \cdot 10^6$
	City's schedule	$1.196 \cdot 10^5$ $(3.07 \cdot 10^3)$	697	780	$2.127 \cdot 10^6$
	Random	$0.976 \cdot 10^5$ $(6.17 \cdot 10^4)$	622	632	$2.194 \cdot 10^6$

schedules. MCTS (with our set of timings) was successful at increasing the total distance traveled and the number of completed trips by 6%, compared to the city's timing schedules, and reducing the number of stops and delays in the network by 15% and 33%, respectively. Although MCTS (with the city's set of timings) did not perform as well as the city's timing schedules in terms of the number of stops, it did significantly improve the total distance by 18%, the number of trips by 22% and the delay by 59%. All in all, MCTS performed considerably better in comparison with the timing schedules obtained from the field research. This indicates that even in real-world scenarios we can use MCTS to obtain a timing schedule without having to study traffic patterns extensively. Also, MCTS helps to experiment with different sets of timings easily. Thus, cities can use MCTS as a guideline to optimize traffic signal timings and handcraft high-performance traffic signal policies faster, or control complex intersections with MCTS directly.

Chapter 6

Conclusion and Future Work

In this thesis, we investigated a novel approach to define the whole network traffic signal optimization problem as a single-player game. MCTS was used along with the simulator SUMO to perform a heuristic search (using the PUCT values of each action) to approximate the optimal traffic signal timings. The results of MCTS were compared to that of handwritten policies across three different types of road networks, which represented different real-world scenarios. The results of our experiments show that MCTS performs well in all road networks we considered. In the first network, that models interconnected intersections of a city, MCTS performed significantly better than our handwritten policy across all the performance metrics. Moreover, in the second network which models a long corridor of intersections, MCTS performed equally better when compared to our handwritten policy. The third network has LRT rail crossings and MCTS was able to outperform the city's timing schedules that we obtained by observing traffic. This is quite impressive since MCTS was able to use the LRT's timing schedules to plan better before the LRT arrived. This shows that MCTS is capable of producing better timing schedules with currently available data on traffic. Therefore, for any intersection, new or old, we can deploy MCTS to achieve good timings quickly rather than spending a considerable amount of time constructing timing policies manually. Even in the scenarios in which MCTS performs equally well to a handwritten timing schedule, it is still better to use MCTS, as it does not require any prior knowledge of the intersections. It will be faster to deploy MCTS than to analyze

different traffic patterns and come up with an optimum timing schedule. Additionally, MCTS is not dependent on the design of an intersection as it can work with any kind of intersection. Thus, using MCTS on newly constructed intersections can be beneficial. There have been many scenarios in which artificial intelligent models help experts in their domains, such as Procedural Content Generation, Github's Copilot, etc. Similarly, MCTS along with a simulator can help traffic experts to approximate good traffic signal timings effectively.

This preliminary research was done to see how viable MCTS is in the traffic optimization domain. Possible future research could focus on the following:

1. **Simulator**: Since SUMO is slow, we can try to build a simulator that would not require TCP connections to control simulations, but rather gives MCTS direct access to state variables. Also, instead of using XML files, we can use JSON to store and read information about the network. JSON data are much faster to read and write especially when there is a lot of information. Creating a simulator with these features would help us achieve real-time performance.

2. **Parallel MCTS**: Implementing parallel MCTS would help us to perform more number of iterations, thus, giving us better timing schedules. It would also help us increase the horizon time and allow MCTS to get observe further into future traffic development.

3. **Accurate Simulations**: Our model relies on simulations to approximate optimal traffic signal timings. However, in Section 4.3.3, we discussed that there can be situations in which the simulations are inaccurate. A possible improvement would be to augment search trees with chance nodes that model several likely traffic developments and then use MCTS tailored to such trees for optimization.

4. **Better Policies**: By spending more time on creating better rollout policies, we can try to decrease the number of nodes that MCTS has to explore to achieve the same performance. In addition, we can create

a system similar to AlphaZero by training a policy network to predict the in-tree PUCT action probabilities and training a state evaluator for non-terminal nodes. With the policy network and the state evaluation network we can cut down rollouts and increase the performance considerably, similar to AlphaGo. Using a lookup table can be costly in terms of memory as there can be a vast number of traffic states.

Bibliography

[1] B. Abramson, "Expected-outcome: A general model of static evaluation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 12, no. 2, pp. 182–193, 1990.

[2] V. Astarita, V. P. Giofrè, G. Guido, and A. Vitale, "The use of adaptive traffic signal systems based on floating car data," *Wireless Communications and Mobile Computing*, vol. 2017, 2017.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[4] B. Beak, K. L. Head, and Y. Feng, "Adaptive coordination based on connected vehicle technology," *Transportation Research Record*, vol. 2619, no. 1, pp. 1–12, 2017.

[5] L. Bieker-Walz and M. Behrisch, "Modelling green waves for emergency vehicles using connected traffic data," *EPiC Series in Computing*, vol. 62, pp. 10–20, 2019, ISSN: 23987340. DOI: 10.29007/sj1m.

[6] D. Bretherton, K. Wood, and N. Raha, "Traffic monitoring and congestion management in the scoot urban traffic control system," *Transportation research record*, vol. 1634, no. 1, pp. 118–122, 1998.

[7] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[8] B. Brügmann, "Monte Carlo Go," Citeseer, Tech. Rep., 1993.

[9] S. Carolina, "Wilcoxon signed-rank test," no. 7, pp. 7–9, 2008.

[10] S. A. Celtek, A. Durdu, and M. E. M. Alı, "Real-time traffic signal control with swarm optimization methods," *Measurement: Journal of the International Measurement Confederation*, vol. 166, p. 108 206, 2020, ISSN: 02632241. DOI: 10.1016/j.measurement.2020.108206. [Online]. Available: https://doi.org/10.1016/j.measurement.2020.108206.

[11] T.-H. Chang and G.-Y. Sun, "Modeling and optimization of an oversaturated signalized network," *Transportation Research Part B: Methodological*, vol. 38, no. 8, pp. 687–707, 2004.

[12] G. M.-B. Chaslot, M. H. Winands, and H. J. van Den Herik, "Parallel Monte-Carlo tree search," in *International Conference on Computers and Games*, Springer, 2008, pp. 60–71.

[13] Y. Cheng, X. Hu, Q. Tang, H. Qi, and H. Yang, "Monte Carlo tree search-based mixed traffic flow control algorithm for arterial intersections," *Transportation research record*, vol. 2674, no. 8, pp. 167–178, 2020.

[14] S.-B. Cools, C. Gershenson, and B. D'Hooghe, "Self-organizing traffic lights: A realistic simulation," in *Advances in applied self-organizing systems*, Springer, 2013, pp. 45–55.

[15] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Proceedings of International conference on computers and games*, Springer, 2006, pp. 72–83.

[16] B. Efron, "Student's t-Test under Symmetry Conditions," *Journal of the American Statistical Association*, vol. 64, no. 328, pp. 1278–1302, 1969. DOI: 10.1080/01621459.1969.10501056.

[17] Y. Feng, M. Zamanipour, K. L. Head, and S. Khoshmagham, "Connected vehicle–based adaptive signal control and applications," *Transportation Research Record*, vol. 2558, no. 1, pp. 11–19, 2016.

[18] 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.

[19] N. H. Gartner, S. F. Assman, F. Lasaga, and D. L. Hou, "A multi-band approach to arterial traffic signal optimization," *Transportation Research Part B: Methodological*, vol. 25, no. 1, pp. 55–74, 1991.

[20] N. H. Gartner, J. D. Little, and H. Gabbay, "Optimization of traffic signal settings by mixed-integer linear programming: Part I: The network coordination problem," *Transportation Science*, vol. 9, no. 4, pp. 321–343, 1975.

[21] N. H. Gartner, F. J. Pooran, and C. M. Andrews, "Optimized policies for adaptive control strategy in real-time traffic adaptive control systems: Implementation and field testing," *Transportation Research Record*, vol. 1811, no. 1, pp. 148–156, 2002.

[22] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," pp. 273–280, 2007.

[23] ——, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

[24] A. Hajbabaie and R. F. Benekohal, "Traffic signal timing optimization," *Transportation Research Record*, no. 2355, pp. 10–19, 2013, ISSN: 03611981. DOI: 10.3141/2355-02.

[25] A. Hajbabaie, J. C. Medina, R. F. Benekohal, *et al.*, "Traffic signal coordination and queue management in oversaturated intersections," NEXTRANS Center (US), Tech. Rep., 2011.

[26] H. Han, L. Peng, A. Teng, C. Wang, and T. Z. Qiu, "Evaluation of freeway travel speed estimation using anonymous cellphones as probes: A field study in china," *Canadian Journal of Civil Engineering*, vol. 48, no. 7, pp. 859–867, 2021. DOI: 10.1139/cjce-2019-0628.

[27] S. James, B. S. Rosman, and G. Konidaris, "An investigation into the effectiveness of heavy rollouts in UCT," 2016.

[28] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *European conference on machine learning*, Springer, 2006, pp. 282–293.

[29] P. Koonce and L. Rodegerdts, "Traffic signal timing manual.," United States. Federal Highway Administration, Tech. Rep., 2008.

[30] E. Korkmaz and A. P. AKGÜNGÖR, "Delay estimation models for signalized intersections using differential evolution algorithm," *Journal of Engineering Research*, vol. 5, no. 3, 2017.

[31] P. Krecl, C. Johansson, A. C. Targino, J. Ström, and L. Burman, "Trends in black carbon and size-resolved particle number concentrations and vehicle emission factors under real-world conditions," *Atmospheric Environment*, vol. 165, pp. 155–168, 2017.

[32] S. Långström and E. Fridsäll, *Optimizing traffic flow on congested roads*, 2019.

[33] F. Lian, B. Chen, K. Zhang, L. Miao, J. Wu, and S. Luan, "Adaptive traffic signal control algorithms based on probe vehicle data," *Journal of Intelligent Transportation Systems: Technology, Planning, and Operations*, vol. 25, no. 1, pp. 41–57, 2021, ISSN: 15472442. DOI: 10.1080/15472450.2020.1750384. [Online]. Available: https://doi.org/10.1080/15472450.2020.1750384.

[34] J. D. Little, "The synchronization of traffic signals by mixed-integer linear programming," *Operations Research*, vol. 14, no. 4, pp. 568–594, 1966.

[35] J. D. Little, M. D. Kelson, and N. H. Gartner, "MAXBAND: A versatile program for setting signals on arteries and triangular networks," 1981.

[36] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic Traffic Simulation using SUMO," in *Proceedings of IEEE International Conference on Intelligent Transportation Systems*, IEEE, 2018. [Online]. Available: https://elib.dlr.de/124092/.

[37] D. E. Lucas, P. B. Mirchandani, and K. Larry Head, "Remote simulation to evaluate real-time traffic control strategies," *Transportation Research Record*, vol. 1727, no. 1, pp. 95–100, 2000.

[38] X. Ma, X. Hu, T. Weber, and D. Schramm, "Evaluation of Accuracy of Traffic Flow Generation in SUMO," *Applied Sciences*, vol. 11, no. 6, p. 2584, 2021.

[39] K. Małecki, P. Pietruszka, and S. Iwan, "Comparative Analysis of Selected Algorithms in the Process of Optimization of Traffic Lights," Feb. 2017, ISBN: 978-3-319-54429-8. DOI: 10.1007/978-3-319-54430-4_48.

[40] P. Mirchandani and L. Head, "A real-time traffic signal control system: architecture, algorithms, and analysis," *Transportation Research Part C: Emerging Technologies*, vol. 9, no. 6, pp. 415–432, 2001.

[41] S. G.-Y. W.-R. Munos and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," *Technical Report RR-6062*, vol. 32, 2006.

[42] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search," in *Proceedings IEEE Conference on Computational Inteligence in Games (CIG)*, IEEE, 2013, pp. 1–8.

[43] R. Putha and L. Quadrifoglio, "Using ant colony optimization for solving traffic signal coordination in oversaturated networks," *Transportation Research Board 89th Annual Meeting*, 2010.

[44] H. Qi and X. Hu, "Monte carlo tree search-based intersection signal optimization model with channelized section spillover," *Transportation Research Part C: Emerging Technologies*, vol. 106, pp. 281–302, 2019.

[45] F. Qiao, X. Tan, and F. A. Tobi, "Optimization of bidirectional green wave of traffic systems on urban arterial road," *The 9th International Conference on Modelling, Identification and Control*, no. Icmic, 2017.

[46] K. Rasmussen, *Encyclopedia of measurement and statistics*. Sage, 2007, vol. 1.

[47] D. I. Robertson, "TRANSYT: a traffic network study tool," 1969.

[48] C. D. Rosin, "Multi-Armed Bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[49] A. Santos, P. Santos, and F. Melo, "Monte Carlo tree search experiments in hearthstone," Aug. 2017, pp. 272–279. DOI: 10.1109/CIG.2017.8080446.

[50] M. P. Schadd, M. H. Winands, M. J. Tak, and J. W. Uiterwijk, "Single-player Monte-Carlo tree search for SameGame," *Knowledge-Based Systems*, vol. 34, pp. 3–11, 2012.

[51] A. Seify, "Single-agent optimization with monte-carlo tree search and deep reinforcement learning," 2020.

[52] Y. Shi, J. Li, Q. Han, and L. Lv, "A Coordination Algorithm for Signalized Multi-Intersection to Maximize Green Wave Band in V2X Network," *IEEE Access*, vol. 8, pp. 213 706–213 717, 2020.

[53] 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, pp. 484–489, 2016.

[54] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[55] ——, "A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[56] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[57] A. G. Sims and K. W. Dobinson, "The Sydney coordinated adaptive traffic (SCAT) system philosophy and benefits," *IEEE Transactions on vehicular technology*, vol. 29, no. 2, pp. 130–137, 1980.

[58] D. Sun, R. F. Benekohal, and S. T. Waller, "Bi-level programming formulation and heuristic solution approach for dynamic traffic signal optimization," *Computer-Aided Civil and Infrastructure Engineering*, vol. 21, no. 5, pp. 321–333, 2006.

[59] M. Świechowski, J. Mańdziuk, and Y. S. Ong, "Specialization of a UCT-based general game playing program to single-player games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 218–228, 2015.

[60] Z. Tian, T. Urbanik, C. Messer, K. Balke, and P. Koonce, "A system partition approach to improve signal timing," in *Transportation Research Board*, 2003.

[61] T. Urbanik, A. Tanaka, B. Lozner, E. Lindstrom, K. Lee, S. Quayle, S. Beaird, S. Tsoi, P. Ryus, D. Gettman, *et al.*, *Signal timing manual*. Transportation Research Board Washington, DC, 2015, vol. 1.

[62] R. T. Van Katwijk, "Multi-agent look-ahead traffic-adaptive control," 2008.

[63] H. Wei, G. Zheng, V. Gayah, and Z. Li, "A survey on traffic signal control methods," *arXiv preprint arXiv:1904.08117*, 2019.

[64] S. Weinzierl, "Introduction to Monte Carlo methods," *arXiv preprint hep-ph/0006269*, 2000.

[65] L. Zhang, Y. Yin, and Y. Lou, "Robust signal timing for arterials under day-to-day demand variations," *Transportation Research Record*, vol. 2192, no. 1, pp. 156–166, 2010.

[66] P. Zhou, Z. Fang, H. Dong, J. Liu, and S. Pan, "Data analysis with multi-objective optimization algorithm: A study in smart traffic signal system," *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 307–310, 2017. DOI: 10.1109/SERA.2017.7965743.