B.E. PROJECT ON

Adaptive Traffic Light Control Using Reinforcement Learning

Submitted by

Rohit Gohri	348/CO/13
Shubham Arora	369/CO/13
Sukriti Bisht	174/EC/13
Vaishali Sharma	183/EC/13

Under the Guidance of
Dr. Sushama Nagpal

A project in partial fulfillment of requirement for the award of
B.E. in
Computer Engineering

Division of Computer Engineering
NETAJI SUBHAS INSTITUTE OF TECHNOLOGY
(UNIVERSITY OF DELHI)
NEW DELHI-110078

ACKNOWLEDGEMENTS

We are sincerely thankful to Dr. Sushama Nagpal for making this project possible and constantly motivating us. We would like to articulate our deep gratitude for her guidance, advice and constant support in the project work. We would like to thank her for being our guide here at Netaji Subhas Institute of Technology, Delhi. We would also like to thank all faculty members and staff of the department of Computer Engineering (COE) for their generous help in various ways for this project.

Last but not the least, our sincere thanks to all of our family and friends who have patiently extended us their support.

We are grateful to our parents for supporting us through the course of this project.

Rohit Gohri (348/CO/13)

Shubham Arora (369/CO/13)

Sukriti Bisht (174/EC/13)

Vaishali Sharma (183/EC/13)

Bachelor of Engineering

Division of Computer Engineering

Netaji Subhas Institute of Technology

University of Delhi.

ABSTRACT

Adaptive traffic signal control (ATSC) has the potential to significantly reduce traffic congestion as opposed to the commonly used pre-timed control systems for isolated intersections. These are also referred to as Smart Traffic Lights. This work presents an implementation of ATSC by using Reinforcement Learning - a machine learning algorithm that has great self-learning potential.

Specifically, this work uses model-free Reinforcement Learning methods like Q-Learning and SARSA to train our Traffic Lights. These are coupled with different action selection strategies and phasing schemes.

The traffic data used for training is generated by an open source, microscopic simulator named "Simulation of Urban MObility" (SUMO). It also provides an environment to run simulations using these algorithms. The improvements gained in queue lengths and waiting times, over pre timed control, are measured.

CERTIFICATE

This is to certify that the project entitled, **Adaptive Traffic Light Control Using Reinforcement Learning**, submitted by:

1. Rohit Gohri (348/CO/13)

2. Shubham Arora (369/CO/13)

3. Sukriti Bisht (174/EC/13)

4. Vaishali Sharma (183/EC/13)

is a record of bona fide work carried out by them, in the department of Computer Engineering, Netaji Subhas Institute of Technology, New Delhi, under the supervision and guidance of the undersigned in partial fulfillment of requirement of the degree of Bachelor of Engineering in Computer Engineering, University of Delhi in the academic year 2016-2017.

Dr. Sushama Nagpal
Department of Computer Engineering
Netaji Subhas Institute of Technology
New Delhi 110078
Date: 02nd June, 2017
Place: New Delhi

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Purpose

The purpose of this project is to develop an Adaptive Traffic Signal Control (ATSC) to reduce traffic congestions as well as delays caused by them. These are intelligent systems that can adapt to changing traffic patterns and can make decisions based on current traffic situation. This project makes use of Reinforcement Learning (RL) to implement these systems.

The aim is to implement an RL based traffic light agent and simulate it in a real-world like environment. Simulation of Urban Mobility (SUMO) is an open source, microscopic traffic simulator. SUMO used for simulating the traffic environment. The simulated traffic resembles the properties and composition of real-world traffic.

1.2 Scope

The RL based traffic light agent interacts with its environment and learns on the go. Agent observes the environment and receives a reward and accordingly takes an action. Thus RL agent functions an ATSC system with self learning capabilities.

The RL agent developed in this project adopts different learning techniques, action selection methods, state definitions, reward definitions and phasing sequences. These help in achieving the objective of reducing traffic congestion, some more than other.

This project develops an intelligent RL traffic light agent that is not only responsive to changing traffic patterns but is also sensitive to emergency vehicles. The agent gives higher priority to emergency vehicles like ambulance, fire brigades, police vehicles etc. This leads to a smaller cumulative delay for emergency vehicles and enables them to reach

their destination in a shorter amount of time, as compared to pre-timed traffic lights.

CHAPTER 2

REINFORCEMENT LEARNING

Reinforcement Learning is an area of Machine Learning that allows machines or software agents to automatically determine best possible behaviour within a given context. This behaviour usually defines what actions an agent should take in order to maximize a notion of cumulative reward.

The problem of Reinforcement learning is studied in many disciplines, such as control theory, multi-agent systems, swarm intelligence, game theory, statistics and genetic algorithms. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. Further, there is a focus on on-line performance, which involves finding a balance between exploration and exploitation.

The basic reinforcement learning system consists of :

- Agent - It is the brain that acts in the world in a way to maximize its reward. It can observe a part of the environment and takes action accordingly
- Model - A representation that mimics the behaviour of the environment
- Policy - It defines the agent's behaviour, mapping from state to action

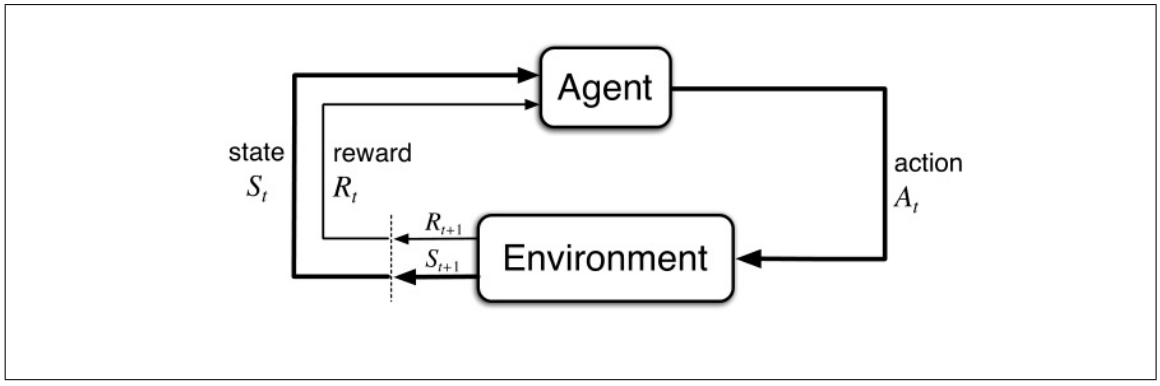


Figure 2.1: The agent environment interaction in reinforcement learning

- Reward Function - Used to determine a scalar immediate reward of a state or state-action pair
- Observation - Describes what part of the environment the agent observes
- Value Function - Defines the expected total reward resulting from a state. It quantizes how good it is to be in a state

2.1 Markov Decision Processes

2.1.1 Markov Property

In the reinforcement learning framework, the agent makes its decisions as a function of a signal from the environment called the environment's state. A process has the Markov property if the future states of the process depends only upon the present state, not on the sequence of events that preceded it. A state that satisfies the Markov Property is called the Markov State.

Consider how a general environment might respond at time t+1 to the action taken at time t. In the most general, causal case this response may depend on everything that has happened earlier. In this case the dynamics can be defined only by specifying the complete probability distribution:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{2.1}$$

for all *r, s'*, and all possible values of the past events: $S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t$. On the other hand if the state signal has the Markov property, then the environment's response at $t+1$ depends only on the state and action representations at t, in which case the environment's dynamics can be defined by specifying only :

$$Pr\{R_{t=1} = r, S_{t+1} = s' | S_t, A_t\} \tag{2.2}$$

for all r, s', S_t and A_t If an environment has the Markov property, then its one-step dynamics enable us to predict the next state and expected next reward given the current state and action. It can be shown that, by iterating this equation, one can predict all future states and expected rewards from knowledge of only the current state. It also follows that Markov states provide the best possible basis for choosing actions. That is, the best policy for choosing actions as a function of a Markov state is just as good as the best policy for choosing actions as a function of complete histories. The Markov property is important in reinforcement learning because decisions and values are assumed to be a function of only the current state. In order for these to be effective and informative, the state representation must give the necessary information about the current state of the system.

2.1.2 Markov Decision Processes (MDP)

A reinforcement learning task that satisfies the Markov property is called a Markov Decision Process or MDP. If the state and action spaces are finite, then it is called a finite MDP. A particular finite MDP is defined by its state and action sets and one-step dynamics of the environment.

Given any state, action (s,a), the probability of each possible next state, s', is called transition probability and is given as follows:

$$p(s'|s,a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\} \tag{2.3}$$

Similarly, given any current state, action (s,a) together with any next state, s', the expected value of the next reward is:

$$r(s,a,s') = E[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] \tag{2.4}$$

These quantities, $p(s'|s,a)$ and $r(s,a,s')$, completely specify the important aspects of dynamics of a finite MDP.

2.2 Value Functions

Almost all reinforcement learning algorithms involve estimating value functions - functions of states (or state-action pairs) that estimate how good it is for the agent to be in a given state (or perform a given action for a given state). The notion of "how good" is defined in terms of expected future rewards. A policy, π, is a mapping from each state, $s \in S$, and action, $a \in A(s)$, to the probability of taking action a when in state s. The value of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define $v_\pi(s)$ formally as:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma R_{t+k+1}|S_t = s] \tag{2.5}$$

where $E\pi[\Delta]$ denotes the expected value given that the agent follows policy π, and t is any time step. We call the function v_π the state-value function for policy π. Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma R_{t+k+1}|S_t = s, A_t = a] \tag{2.6}$$

We call q_π the action-value function for policy π. These value functions can be estimated from experience. For example, if an agent follows policy π and maintains an average of the actual returns for each state encountered that have followed that state, then the average will converge to the state's value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. Similarly if separate averages are kept for each action taken in a state, then these averages will converge to the action values, $q_\pi(s,a)$.

A fundamental property of value functions used throughout reinforcement learning is that they satisfy particular recursive relationships. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible

successor states:

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|S_t = s] \\ &= E_\pi[\sum_{k=0}^{\infty} \gamma R_{t+k+1}|S_t = s] \\ &= E_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma R_{t+k+2}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{a'} p(s'|s,a)[r(s,a,s') + \gamma E_\pi[\sum_{k=0}^{\infty} \gamma R_{t+k+2}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{a'} p(s'|s,a)[r(s,a,s') + \gamma v_\pi(s')] \end{aligned} \tag{2.7}$$

(2.7) is the Bellman equation for v_π . It expresses a relationship between the value of a state and the values of its successor states. The Bellman equation averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way. The value function v_π is the unique solution to its Bellman equation.

2.3 Optimality

Solving a reinforcement learning task roughly means finding a policy that achieves a lot of reward over the long run. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v'_\pi(s) \forall s \in S$. There is always at least one policy that is better than or equal to all other policies. This is called the optimal policy. The optimal policies(may be more than one) are denoted by $\pi^\star$. They share the same state-value function, called the optimal state-value function, denoted $v_\star$, and defined as (2.8).

$$v_\star(s) = \max_\pi v_\pi(s) \forall s \in S \tag{2.8}$$

Optimal policies also share the same optimal action-value function, denoted by $q_\star$, and defined as equation.

$$q_\star(s,a) = \max_\pi q_\pi(s,a) \forall s \in S \text{ and } \forall a \in A(s) \tag{2.9}$$

For the state–action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write $q_\star$ in terms of $v_\star$ as equation.

$$q_\star(s,a) = E[R_{t+1} + \gamma v_\star(S_{t+1}) | S_t = s, A_t = a] \tag{2.10}$$

Because $v_\star$ is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (2.7). This is the Bellman equation for $v_\star$, or the Bellman optimality equation. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state.

$$\begin{aligned}
v_\star(s) &= \max_{a \in A(s)} q_{\pi\star}(s,a) \\
&= \max_a E_{\pi\star}[G_t | S_t = sA_t = a] \\
&= \max_a E_{\pi\star}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a] \\
&= \max_a E_{\pi\star}[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a] \\
&= \max_a E[R_{t+1} + \gamma v_\star(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_{a \in A(s)} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_\star(s')]
\end{aligned} \tag{2.11}$$

The Bellman optimality equation for $q_\star$ is equation.

$$\begin{aligned}
q_\star(s,a) &= E[R_{t+1} + \gamma \max_{a'} q_\star(S_{t+1}, a') | S_t = s, A_t = a] \\
&= \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \max_{a'} q_\star(s', a')]
\end{aligned} \tag{2.12}$$

The backup diagrams for these are Fig 2.2

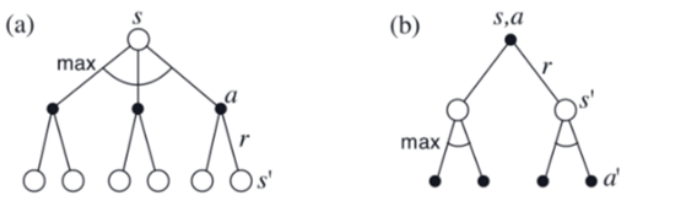


Figure 3.7: Backup diagrams for (a) v_* and (b) q_*

Figure 2.2: Backup Diagrams for optimality equations

CHAPTER 3

RL-BASED ADAPTIVE TRAFFIC SIGNAL CONTROL

A generic RL software agent is developed using the Python programming language in such a way that different learning methods, state representations, phasing sequence, reward definition, and action selection strategies can be tested for any control task.

The interactions between the agent and the traffic simulation environment is represented in Fig.(3.1)

The agent component implements the control algorithm. The agent is the learner and the decision maker (or action selector) that interacts with the environment by first receiving the system's state and the reward and then selecting an action accordingly. The simulation environment component models the traffic environment. We use SUMO, a microscopic traffic simulator, to model the traffic environment.

3.1 Temporal Difference (TD) Learning Methods

Due to the stochasticity associated with the traffic flow approaching signalized intersections, the ATSC problem can been modeled as a stochastic control problem in which the goal is to obtain a policy (mapping between each intersection traffic state s and signal action a) that maximizes the expected cumulative discounted reward (e.g. reduction in delay time) at each traffic state. The expected cumulative discounted reward at state s is defined as the value function of the state–action pair Q(s,a), also called Q-value. The traffic signal control agent updates the estimate of the value functions over time until it converges to the optimal values, and consequently, the optimal policies.

The temporal difference is a class of RL methods that update the estimates of the state values immediately after visiting the state in which the value function is estimated based on the difference between temporally successive estimations, which is called TD error,

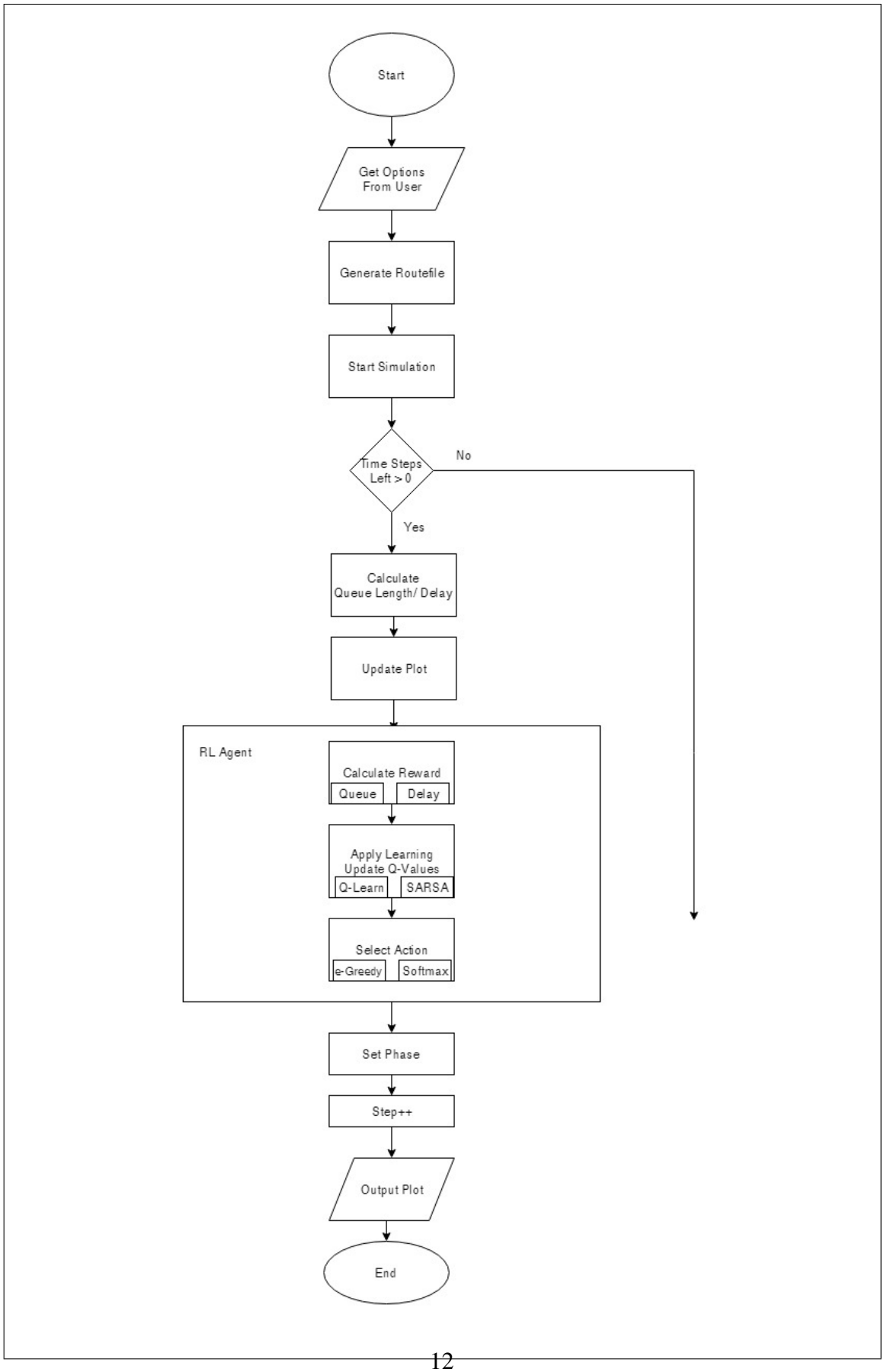


Figure 3.1: RL blocks, traffic agent and environment

denoted TD^k, where α and γ are the learning rate and the discount factor, respectively.

$$TD^t(s^{t-1}) = [r^t + \gamma V^{t-1}(s^t) - V^{t-1}(s^{t-1})] \tag{3.1}$$

$$V^t(s^{t-1}) - V^{t-1}(s^{t-1}) + a^t TD^t(s^{t-1}) \tag{3.2}$$

At time t, the value of state s, $V^t(s^{t-1})$, is updated using the observed reward R^t and the estimate $V^{t-1}(s^t)$. Eq. (1) represents the simplest form of TD known as TD(0). It is also known as 1-step TD. In TD(0), the traffic signal agent looks back one step in time and updates the value functions based on the reward R^t.

TD(0) can represent either an on-policy or an off-policy algorithm, depending on the updating procedure for the value functions.

3.1.1 Q-Learning: Off-Policy Algorithm

Q-Learning updates the estimated value functions (Q-values) using greedy actions i.e. the action that has the highest value, independent of the policy being followed which may involve exploratory actions. Hence Q-Learning is an off-policy algorithm as the agent attempts to improve a policy while following another one's actions. The equation for Q-Learning is given in (3.3)

$$Q^t(s^{t-1}, a^{t-1}) = Q^{t-1}(s^{t-1}, a^{t-1}) + \alpha[R^t + \gamma \max_{(a' \in A)} Q^{t-1}(s^t, a^t) - Q^{t-1}(s^{t-1}, a^{t-1})] \tag{3.3}$$

3.1.2 SARSA: On-Policy Algorithm

In SARSA, the Q-value functions are updated using actions (a') determined by a policy that may involve exploratory actions. SARSA algorithm is considered as an on-policy TD method since it is learning and improving the same policy that is being followed. The equation for Q-Learning is given in (3.4).

$$Q^t(s^{t-1}, a^{t-1}) = Q^{t-1}(s^{t-1}, a^{t-1}) + \alpha[R^t + \gamma Q^{t-1}(s^t, a^t) - Q^{t-1}(s^{t-1}, a^{t-1})] \tag{3.4}$$

3.2 State Representation

The observation that the agent makes of the environment is the state of the environment. Different state representations can be adopted to facilitate different manipulations. In this project, state is represented by a vector of N components, where N is the number of phases of the traffic light. Two representations used are covered in the following sections.

3.2.1 Queue Length

This definition is represented by a vector of N components that are the maximum queue lengths associated with each phase. This is shown in (3.5).

$$s_t^i = \max_{(l \in L_i)} q_l^t \forall i \ \in \ 1,2,...N \tag{3.5}$$

where q_l^t is the number of queued vehicles in lane l at time k. The maximum queue is taken over all lanes that belong to the lane group corresponding to phase i, L_i. Vehicle (v) is considered to be in a queue if its speed is below certain threshold speed ($Sp_T hr$). In this project, the threshold speed ($Sp_T hr$) is taken to be 0.1m/s.

3.2.2 Cumulative Delay

The vehicle cumulative delay for phase I is the summation of the cumulative delay of all the vehicles that are currently traveling on lane group L_i. Vehicles leave the system once they clear the stop line at the intersection. This state is also represented by N components where each component is the cumulative delay of the corresponding phase, as shown in equation x.

$$s_t^i = \sum_{l \in L_i} \sum_{v \in V_l^t} Cd_v^t \forall i \in 1,2,...,N \tag{3.6}$$

where Cd_v^t is the total time spent by the vehicle v in a queue upto time step t

3.3 Phasing Sequence

Two phasing sequences are covered

- Fixed phasing sequence - order of phases is followed is fixed
- Variable phasing sequence - phasing sequence is not predetermined or fixed

3.3.1 Fixed Phasing Sequence

The action is a binary number that indicates either extend the current phase by one time step $(a = 0)$ or terminate the current green phase and move to the next phase $(a = 1)$.

$$a_t = i, i \in 0, 1 \tag{3.7}$$

3.3.2 Variable Phasing Sequence

In this scheme, the phasing sequence is not fixed. The action is the phase that should be in effect next and can hence take N values, where N is the number of phases. If the action is the same as the current green phase, then the green time for that phase will be extended by one time step.

$$a_t = i, i \in 1, 2, ..., N \tag{3.8}$$

3.4 Reward Definition

This project consider two definitions for the immediate reward that the agent receives. These definitions are discussed in the following sections.

3.4.1 Maximizing the Reduction in the Total Cumulative Delay

The immediate reward is defined as the reduction (saving) in the total cumulative delay, that is, the difference between the total cumulative delays of two successive decision points.

The total cumulative delay at time k is the summation of the cumulative delay, up to time k, of all the vehicles that are currently in the system. If the reward has a positive value, this means that the delay is reduced by this value after executing the selected action. However, a negative reward value indicates that the action results in an increase in the total cumulative delay :

$$R^t = \sum_{i \in N} \sum_{i \in L_i} \Big(\sum_{v \in V_l^{t-1}} Cd_v^{t-1} - \sum_{v \in V_l^t} Cd_v^t \Big) \tag{3.9}$$

3.4.2 Reward Definition 2: Minimizing and Balancing Queue Length

Reducing the cumulative delay does not guarantee preventing queue spilling back and affecting minor streets. A reward can be defined as the reduction in the sum of the squared maximum queues. This leads to minimization and equalization of queue lengths across different phases.

$$R^t = \sum_{i \in N} (\max_{I \in L_i} q_l^{t-1})^2 - \sum_{i \in N} (\max_{I \in L_i} q_l^t)^2 \tag{3.10}$$

3.5 Action Selection Strategy

While accumulating the maximum reward requires the traffic signal control agent to exploit the best experienced actions, it also has to explore new actions to discover better selection of actions for the future. Exploration allows visiting a larger number of state–action pairs, which is the condition of convergence to optimal policy (i.e. visit each state–action pair, theoretically, an infinite number of times). To balance the exploration and exploitation in RL, algorithms such as ε-greedy and softmax are typically used. These are discussed in the following sections.

3.5.1 ε-Greedy

In each iteration, ε-greedy method is used for action selection in which the agent selects the greedy action most of the time except for ε amount of time, when it selects a random action uniformly. This project adopts an ε-greedy exploration method with a gradually decreasing rate of exploration. In the beginning the agent mostly explores, as it does not know much about the environment yet. Exploitation increases with time as the agent converges to the optimal policy. Gradual decreasing rate is required to help the agent covering the entire state space in less learning time.

The gradually decreasing rate of exploration can be represented by an exponentially decreasing function, as shown in equation x.

$$\varepsilon = e^{-En} \tag{3.11}$$

where E is a constant (taken to be 0.01) and n is the age of the agent.

3.5.2 Softmax

One disadvantage of the ε-greedy selection method is that it treats all exploratory actions equally, irrespective of the estimated value function of each action. This means that it is as likely to chose the worst action as it is to chose the next-to-best action. To overcome this limitation, that may adversely affect the real-life applications, Softmax makes action selection probabilities proportional to the estimated values. The Softmax method used in this project is a Boltzmann distribution. The resulting action selection probabilities of an action, a, at any state, s, is given by equation x.

$$P_s(a) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_{b \in A} e^{\frac{Q(s,b)}{\tau}}} \tag{3.12}$$

where A is the set of possible actions, τ is called the temperature.

When τ is high, each action will have approximately the same probability to be selected (more exploration). When τ is low, actions will be selected proportionally to their estimated payoff (more exploitation). The expression of τ used in this project is given in equation x

$\tau = e^{-Cn}$

where C is inversely proportional to the number of cars in the system.

3.5.3 Learning Rate and Discount Rate

The learning rate, α, determines the rate at which learning takes place. It determines the extent of change in Q values in each iteration. The value of α is kept between 0 and 1. Its value for this project is taken to be 0.4.

The discount rate, γ, signifies discounting of past rewards. We discount rewards by a factor of γ at each time step into the past. The value of γ is kept between 0 and 1. Smaller value of signifies that we care more about immediate rewards than future rewards. It is chosen to be 0.8 for this project.

CHAPTER 4

SUMO

Simulation of Urban Mobility (SUMO), a microscopic traffic simulator has been used for simulating vehicular traffic. It is an open source, highly portable, microscopic and continuous road traffic simulation package designed to handle large road networks. It models individual elements of transportation systems as well as their interactions. SUMO provides functionality for the following :

- Controls traffic behaviour and composition by modelling vehicles and public transport explicitly
- Sets time schedules of traffic lights
- Imports different network formats, like open street maps, to build road networks
- Generates routes programmatically for all vehicles
- Controls the simulation using Traffic Control Interface (TraCI)
- Measures performance characteristics like average waiting time, average queue length

4.1 Components of SUMO Road Network

A road network in SUMO is defined by the network file. It contains information about the roads and intersections on which the simulated vehicles run. It is encoded in XML format and contains information of the following :

- Nodes (Junctions) which represent intersections
- Edges which represent roads or streets. All edges are unidirectional

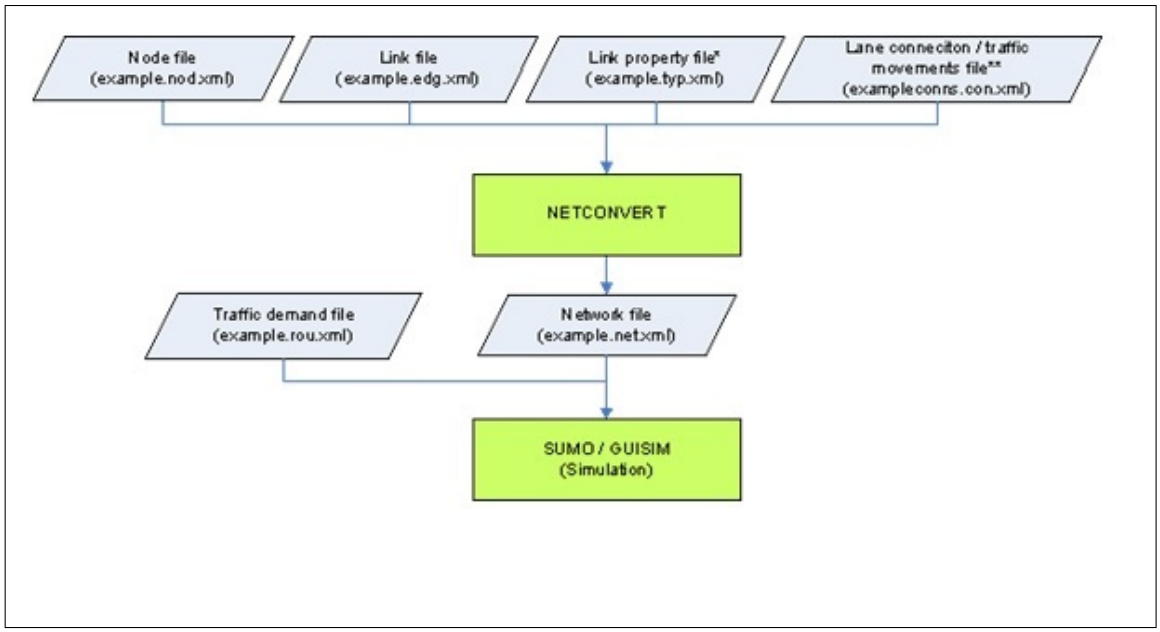


Figure 4.1: The agent environment interaction in reinforcement learning

- Every edge is a collection of lanes. Each lane has information about its position, shape, and speed limit
- Traffic light logic is referenced by junctions
- Connections define connectivity between different lanes at junctions

It is constructed as shown in fig. 4.1.

4.1.1 Node

A node in SUMO net file as

<node id="<STRING>" x="<FLOAT>" y="<FLOAT>" [type="<TYPE>"]/>

4.1.2 Edge

An edge in SUMO net file is defined as

<edge id="<STRING>"from="<NODE_ID>"to="<NODE_ID>" [type="<STRING>"][nu

Table 4.1: Description of a node

Attribute Name	Value Type	Description
id	id(string)	The name of the node
x	float	x-pos of node on the plane(meters)
y	float	y-pos of node on the plane(meters)
type	$enum(\text{"}traffic_light\text{"},\text{"}unregulated\text{"})$	optional type for the node

Table 4.2: Description of an edge

Attribute Name	Value Type	Description
id	id(string)	The id of the edge (must be unique)
from	referenced node id	Name of a node within the nodes-file the edge starts at
to	referenced node id	Name of a node within the nodes-file the edge ends at
numLanes	int	Number of lanes of the edge(int)
speed	float	The maximum speed allowed on the edge in m/s
length	int	The length of the edge in meter

4.1.3 Connection

A connection in SUMO net file is defined as

```
<connection from="<FROM_EDGE_ID>" to="<T0_EDGE_ID>" fromLane="<INT_1>"
```

Table 4.3: Description of a connection

Attribute Name	Value Type	Description
from	referenced edge id	The name of the edge the vehicles leave
to	referenced edge id	Name of the edge vehicles may reach when leaving "from"
fromLane	int	The lane index of the incoming lane
toLane	int	The lane index of the outgoing lane

4.1.4 Traffic Light

A traffic light in SUMO net file is defined as

```
<tlLogic id="0" type="static" programID="custom" offset="0">
    <phase duration="500" state="grrrrgrrrrGGGGGgrrrr"/>
</tlLogic>
```

Table 4.4: Description of a traffic light

Attribute Name	Value Type	Description
id	id (string)	Id of the traffic light. Must be a valid id in the .net.xml file
duration	time (int)	The duration of the phase
state	signal states	The traffic light states for this phase

4.2 Generation of Road Network

4.2.1 Single Intersection

The network file of single intersection has been generated by using NETCONVERT (a command line application) which takes .node.xml, .edges.xml, .rou.xml, .con.xml as input and generates .net.xml as output.

4.2.2 Abstract Road Networks

Abstract road networks are generated using NETGENERATE. It requires command line parameters and outputs a SUMO road network. Using NETGENERATE, we can generate grid, spider and random networks. NETGENERATE has been used to generate a 2*2 grid network and the network has been edited using NETEDIT.

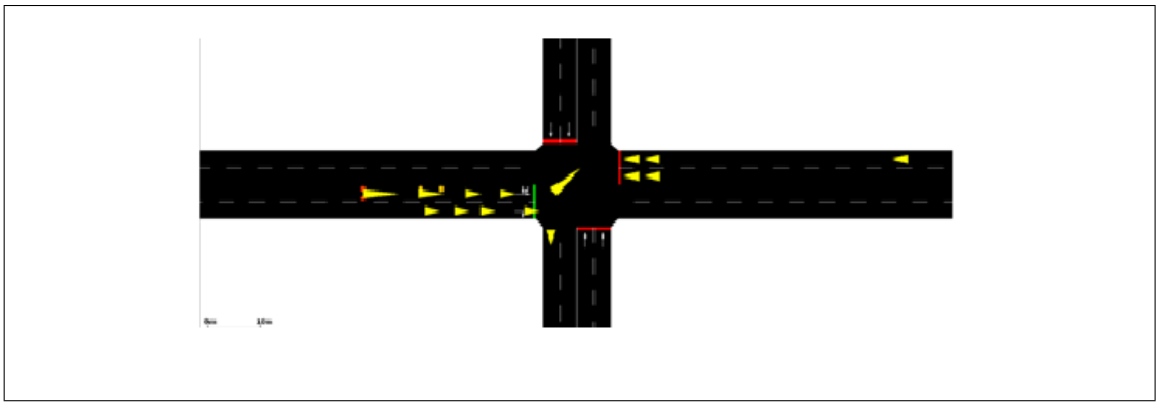


Figure 4.2: Simulation of Single Intersection in SUMO

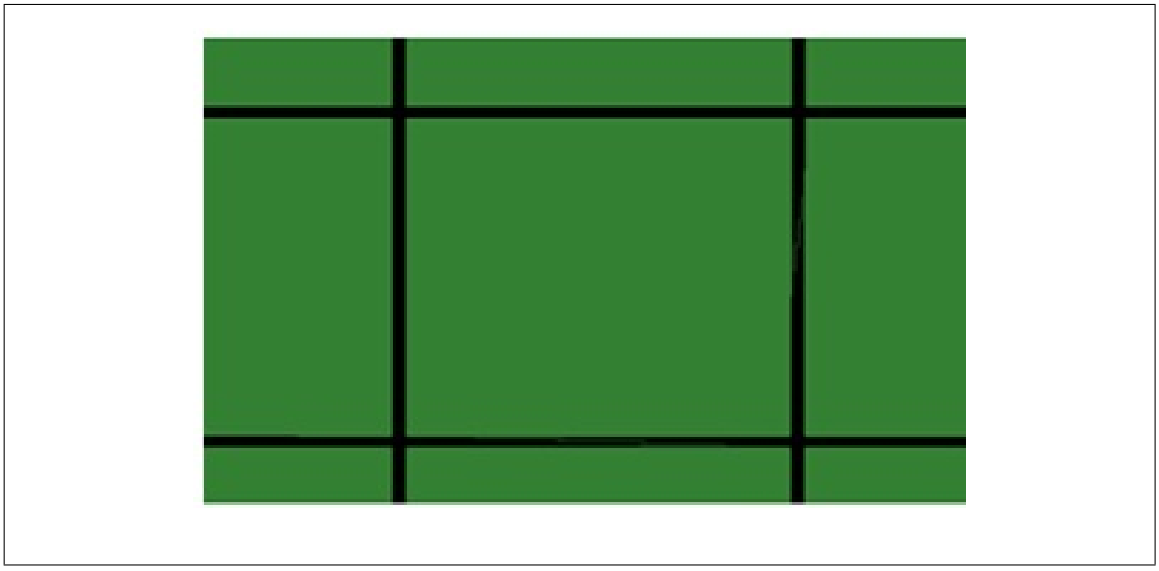


Figure 4.3: Simulation of Multiple Intersection in SUMO

4.3 Vehicle Composition

Sumo allows us to manage vehicular demand using

- Vehicle type which describes the vehicle's physical properties
- Route taken by a vehicle

All the properties of vehicles are defined in .rou.xml file.

4.3.1 Traffic Light

A traffic light in SUMO net file is defined as

```
<tlLogic id="0" type="static" programID="custom" offset="0">
    <phase duration="500" state="grrrrgrrrrGGGGGgrrrr"/>
</tlLogic>
```

Table 4.5: Description of a vehicle in SUMO

Attribute Name	Value Type	Description
id	id(string)	The name of the vehicle
type	id	The id of the vehicle type to use for this vehicle
vClass	class(enum)	An abstract vehicle class e.g taxi, bus, motorcycle
route	id	The id of the route the vehicle shall drive along
accel	float	The acceleration ability of vehicles of this type (in m/s^2). Default $value = 2.6m/s^2$
decel	float	The deceleration ability of vehicles of this type (in m/s^2). Default value=4.5 m/s^2
sigma	float	Car-following model parameter. Default value=0.5
maxSpeed	float	The vehicle's maximum velocity (in m/s). Default value=70 m/s
lcStrategic	int	Eagerness to perform strategic lane changing. Higher values result in more lane-changing.Default=1.0
lcCooperative	float	Willingness to perform cooperative lane changing. Lower values result in reduced cooperation.Default=1.0
Length	float	Vehicles length (in m)

Using the above attributes, a vehicle in SUMO has been defined as

```
<vType id="v2" maxSpeed="50" vClass="passenger" probability="0.31"
length="5" accel="4.5" decel="4.0" sigma="0.7"
```

TABLE 5.7 PERCENTAGE DISTRIBUTION OF REGISTERED VEHICLES TO THE TOTAL

Year	Personal Vehicles		Commercial Vehicles				Total
	Cars & Jeeps	M. Cycles/ Scooters	Auto rickshaw	Taxis	Buses*	Good Vehicles	
1980-81	22.48	64.13	3.83	1.20	1.52	6.85	100.00
1990-91	21.74	67.51	3.51	0.57	1.06	5.61	100.00
2000-01	26.64	64.53	2.52	0.53	1.20	4.59	100.00
2006-07	30.66	63.64	1.43	0.48	0.90	2.89	100.00
2007-08	30.73	63.58	1.33	0.54	0.93	2.85	100.00
2008-09	30.92	63.17	1.40	0.66	0.91	2.91	100.00
2009-10	31.21	62.85	1.34	0.70	0.89	2.99	100.00
2010-11	31.34	62.64	1.27	0.84	0.89	3.02	100.00
2011-12	31.50	62.43	1.19	0.94	0.86	3.08	100.00
2012-13	31.83	63.83	1.11	0.90	0.51	1.81	100.00
2013-14	31.70	63.88	1.11	0.95	0.49	1.86	100.00

* Ambulance & other passenger vehicles are included.
Source: Transport Department, GNCT of Delhi.

Figure 4.4: Composition of Vehicular traffic

lcStrategic="6" lcCooperative="0.4"/>

SUMO offers a variety of abstract vehicle classes namely passenger cars, taxi, bus, truck, motorcycle, emergency vehicle, bicycle etc. The composition of vehicles has been generated programmatically using these abstract classes in order to simulate a traffic scenario similar to Delhi. SUMO uses Krauss Model of Car Following to simulate vehicular traffic. This model allows vehicles to drive as fast as possible while maintaining perfect safety. It is able to avoid a collision if the leader starts braking within leader and follower maximum acceleration bounds. It provides different deceleration capabilities among the vehicles and these are handled without violating safety

4.4 Routes

The route file consists of a set of random trips for a given network . The source and destination edge are selected either uniformly or with a modified distribution. The resulting trips are stored in an XML file with .rou.xml as extension.The trips are distributed evenly in an interval. The number of trips is defined by the repetition rate in second.In addition to

```
234     fileDir = os.path.dirname(os.path.realpath('__file__'))
235     filename = os.path.join(fileDir, 'data/cross.net.xml')
236     os.system("python randomTrips.py -n " + filename
237         + " --weights-prefix " + os.path.join(fileDir, 'data/cross')
238         + " -e " + str(options.numberCars)
239         + " -p  4" + " -r " + os.path.join(fileDir, 'data/cross.rou.xml')
240         + " --fringe-factor " + str(fringeFactor)
241         + " --trip-attributes=\"type=\"" + vType + "\"\""
242         + " --additional-file "  +  os.path.join(fileDir, 'data/type.add.xml')
243         + " --edge-permission emergency passenger taxi bus truck motorcycle bicycle"
244         )
```

Figure 4.5: Generating vehicle routes using randomTrips.py

this, additional parameters have been given to the generated vehicles such as max speed, vehicle class etc.

4.5 Sublane Model

This model has been used to simulate the following

- multiple 2-wheeled vehicles driving in parallel on a single lane
- vehicles overtaking a bicycle on a single lane
- formation of virtual lanes in dense traffic (i.e. 3 vehicles driving in parallel on 2 lanes)

This model is activated using the option –lateral-resolution <FLOAT>.

In this model, the regular lanes of the road network are divided into sublanes with a minimum width of the given resolution (–lateral-resolution). The default lane-width of SUMO is 3.2m so a lateral resolution of 0.8 will created exactly 4 sublanes of that width per lane.The vehicles use Krauss Model to implement car following.This model also implements lane changing. Lane changing takes place at the sublane level and uses attributes namely lcStrategic and lcCooperative to show the likelihood of a vehicle changing its sublane.

CHAPTER 5

INTERACTION BETWEEN SUMO AND RL AGENT

This project uses TraCI (Traffic Control Interface) as an interface between SUMO and Reinforcement Learning Agent. With the help of TraCI, we can retrieve, interpret and manipulate the value of simulated objects. TraCI is based on the client server model wherein SUMO behaves as the server and the program as the client

5.1 User Interaction with the system

User can choose from and specify from various options:

- the number of cars generated for simulation
- learning method
- traffic state representation to be used
- phasing scheme

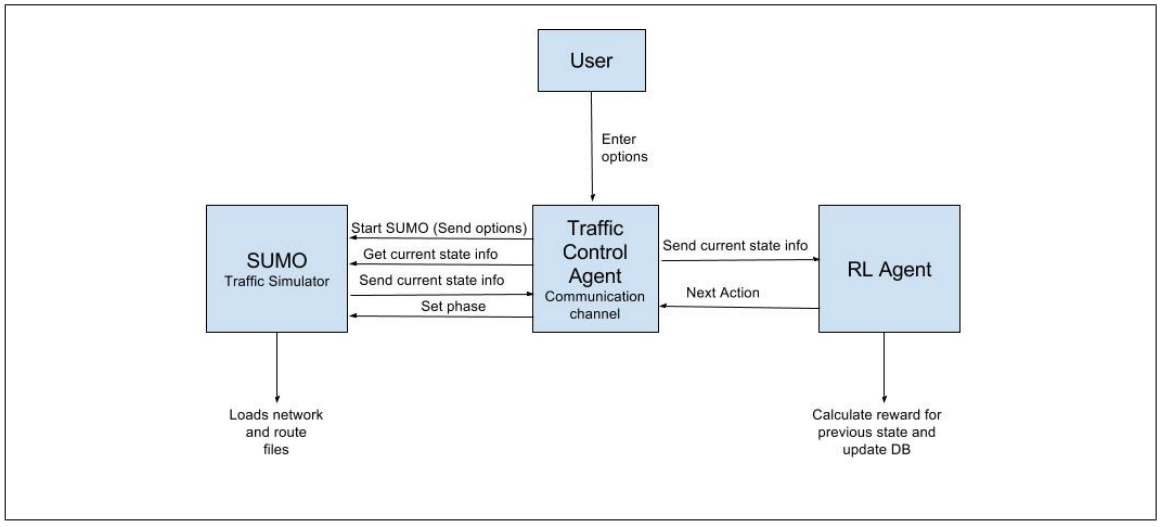


Figure 5.1: Interfacing of different system components

```
def get_options():
    optParser = optparse.OptionParser()
    optParser.add_option("--nogui", action="store_true",
                         default=False, help="run the commandline version of sumo")
    optParser.add_option("--cars", "-C", dest="numberCars", default=100000, metavar="NUM",
                         help="specify the number of cars generated for simulation")
    optParser.add_option("--bracket", dest="bracket", default=10, metavar="BRACKET",
                         help="specify the number with which to partition the range of queue length/cumulative delay")
    optParser.add_option("--learning", dest="learn", default='1', metavar="NUM", choices= ['0', '1', '2'],
                         help="specify learning method (0 = No Learning, 1 = Q-Learning, 2 = SARSA)")
    optParser.add_option("--state", dest="stateRep", default='1', metavar="NUM", choices= ['1', '2'],
                         help="specify traffic state representation to be used (1 = Queue Length, 2 = Cumulative Delay)")
    optParser.add_option("--phasing", dest="phasing", default='1', metavar="NUM", choices= ['1', '2'],
                         help="specify phasing scheme (1 = Fixed Phasing, 2 = Variable Phasing)")
    optParser.add_option("--action", dest="actionSel", default='1', metavar="NUM", choices= ['1', '2'],
                         help="specify action selection method (1 = epsilon greedy, 2 = softmax)")
    optParser.add_option("--sublane", dest="sublaneNumber", default=2, metavar="FLOAT",
                         help="specify number of sublanes per edge (max=6) ")
    options, args = optParser.parse_args()
    return options
```

Figure 5.2: user interaction with Traci

- action selection method
- number of sublanes per edge

This is implemented using the function as shown in figure 5.2

5.2 Traffic Control Interface

After receiving inputs from the user, the program starts SUMO using the command line options and binds TraCI to it. SUMO acts as a server and waits for clients to connect to it. It only prepares the simulation and waits for all external applications to connect to it and take control. SUMO runs until the client demands that the simulation end.

The work of TraCI in this project has been covered in the following sections.

5.2.1 Starting SUMO with command line options

TraCI starts SUMO and provides configuration file, network file and routes file as inputs. In addition, some additional files defining the traffic light phasing scheme, lateral resolution (to implement sublane model) are also provided. Queue output and trip info files are generated as output.

```
88          # get average queue length for current time step
89          queueLength=[]
90          avgQLCurr[i] = 0
91          for lane in lanes:
92              queueLength.append(traci.lane.getLastStepHaltingNumber(lane))
93              avgQLCurr[i] += traci.lane.getLastStepHaltingNumber(lane)
94          avgQLCurr[i] = avgQLCurr[i]/(len(lanes)*1.0)
95
96          # get average queue length till now
97          avgQL[i] = (avgQL[i]*step + avgQLCurr[i])/((step+1)*1.0)
```

Figure 5.3: starting SUMO using Traci

```
118         if (options.stateRep == '1'):
119             # generate current step's phase vector - with queueLength
120             phaseVector[0] = int(round(max(queueLength[4], queueLength[5])/options.bracket))
121             phaseVector[1] = int(round(max(queueLength[0], queueLength[4])/options.bracket))
122             phaseVector[2] = int(round(max(queueLength[0], queueLength[1])/options.bracket))
123             phaseVector[3] = int(round(max(queueLength[3], queueLength[2])/options.bracket))
124             phaseVector[4] = int(round(max(queueLength[2], queueLength[6])/options.bracket))
125             phaseVector[5] = int(round(max(queueLength[6], queueLength[7])/options.bracket))
```

Figure 5.4: Calculating queue length at current time step

```
118         if (options.stateRep == '1'):
119             # generate current step's phase vector - with queueLength
120             phaseVector[0] = int(round(max(queueLength[4], queueLength[5])/options.bracket))
121             phaseVector[1] = int(round(max(queueLength[0], queueLength[4])/options.bracket))
122             phaseVector[2] = int(round(max(queueLength[0], queueLength[1])/options.bracket))
123             phaseVector[3] = int(round(max(queueLength[3], queueLength[2])/options.bracket))
124             phaseVector[4] = int(round(max(queueLength[2], queueLength[6])/options.bracket))
125             phaseVector[5] = int(round(max(queueLength[6], queueLength[7])/options.bracket))
```

Figure 5.5: creating queue length state representation

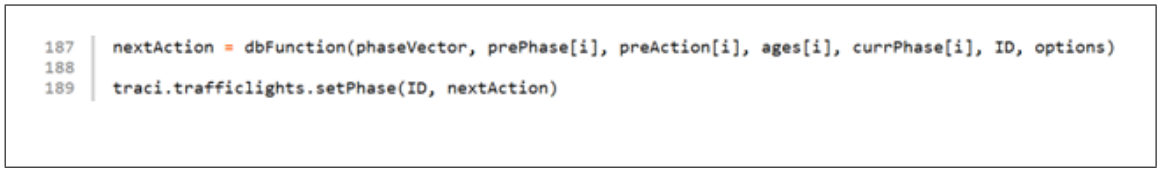
5.2.2 Get Current State Information from SUMO

Traci is used to retrieve and manipulate the value of simulated objects. TraCI queries the state information from SUMO in order to calculate either of the two state representations - Queue length and Cumulative Delay - as specified by the user in the beginning.

```
                j = 0
                for lane in lanes:
                    listVehicles = traci.lane.getLastStepVehicleIDs(lane)
                    for veh in listVehicles:
                        vehicles[j][veh] = oldVehicles[j][veh]
                        if (traci.vehicle.isStopped(veh)):
                            vehicles[j][veh] += 1
                            cumulativeDelay[j] += 1
                    vehToDelete = oldVehicles[j] - vehicles[j]
                    for veh, vDelay in vehToDelete.most_common():
                        cumulativeDelay[j] -= vDelay
                    oldVehicles[j] = vehicles[j]
                    j+=1
                cumuDelay[i] = cumulativeDelay
                oldVeh[i] = oldVehicles

                # generate current step's phase vector - with cumulativeDelay
                phaseVector[0] = int(round(cumulativeDelay[4] + cumulativeDelay[5])/options.bracket)
                phaseVector[1] = int(round(cumulativeDelay[0] + cumulativeDelay[4])/options.bracket)
                phaseVector[2] = int(round(cumulativeDelay[0] + cumulativeDelay[1])/options.bracket)
                phaseVector[3] = int(round(cumulativeDelay[3] + cumulativeDelay[2])/options.bracket)
                phaseVector[4] = int(round(cumulativeDelay[2] + cumulativeDelay[6])/options.bracket)
                phaseVector[5] = int(round(cumulativeDelay[6] + cumulativeDelay[7])/options.bracket)
```

Figure 5.6: Cumulative Delay state representation



```
nextAction = dbFunction(phaseVector, prePhase[i], preAction[i], ages[i], currPhase[i], ID, options)

traci.trafficlights.setPhase(ID, nextAction)
```

Figure 5.7: Set next phase

5.2.3 Setting next traffic light phase

TraCI is used to change the current traffic light phase to next phase according to the action taken by the Reinforcement Learning Traffic Light Agent. For this, the program sends current state information to agent and then receives an action. TraCI is then used to set phase according to this action, as shown in figure 5.7.

5.3 Reinforcement Learning Agent

RL Agent is the main traffic light agent that learns by interacting with the environment and takes action based on reward and current state of the environment.

```
5   def queueBalanceReward(pre, curr, N):
6       # STATE REPRESENTATION IS QUEUE LENGTH
7
8       reward = 0
9       # print(curr, pre, N, "CURR PRE N")
10      for i in range(0, N):
11          reward = reward - (curr[i]**2 - pre[i]**2)
12      #print(state reward, "Reward")
13      return reward
```

Figure 5.8: Minimizing and Balancing Queue Lengths Reward

```
15  def delayReward(pre, curr, N):
16      # STATE REPRESENTATION IS CUMULATIVE DELAY
17
18      reward = 0
19      # print(curr, pre, N, "CURR PRE N")
20      for i in range(0, N):
21          reward = reward - (curr[i] - pre[i])
22      #print(delay reward, "Reward")
23      return reward
```

Figure 5.9: Maximizing Reduction in Total Cumulative Reward

5.3.1 Calculate the reward of the previous state action pair

This project makes use of two types of reward definitions, as discussed in section 3.4, corresponding to the two state representations, as discussed in section 3.2.

Minimizing and Balancing Queue Lengths Reward definition is used along with Queue Length state representation.

Similarly Maximizing Reduction in Total Cumulative Reward definition is used with Cumulative Delay state representation.

```
6   def qLearning(currQ, alpha, gamma, reward, nextMaxQ):
7       """ nextMaxQ =   max possible reward from next step i.e. max over Q of (st+1, at+1)
8           scan DB for next possible actions of curr state and find nextMaxQ
9       """
10      newQ = currQ + alpha*(reward + gamma*nextMaxQ - currQ)
11      #update currQ to newQ in DB
12      return newQ
```

Figure 5.10: Implementation of Qlearning

```
14  def sarsa(currQ, alpha, gamma, reward, nextQ):
15      """ nextQ = Q(curr, nextAction)
16                  where nextAction is selected according to policy - eGreedy or softmax
17      """
18      newQ = currQ + alpha*(reward + gamma*nextQ - currQ)
19      #update currQ to newQ in DB
20      return newQ
```

Figure 5.11: Implementation of SARSA

```
94      if (options.learn == '1'):
95          newQ = qLearning(currQ, globals.alpha, globals.gamma, reward, nextMaxQ)
96      else:
97          nextQ = 0
98          if (options.actionSel=='1'):
99              nextQ = currBSON[eGreedy(globals.numActions, globals.E, age, currBSON)]['qVal']
100         else:
101             nextQ = currBSON[softmax(globals.numActions, options.numberCars, age, currBSON)]['qVal']
102         newQ = sarsa(currQ, globals.alpha, globals.gamma, reward, nextQ)
103
104     qValues.find_one_and_update({"state": pre, "action": preAction}, {'$set': {"qVal": newQ}})
```

Figure 5.12: Updating the q value of the previous state

5.3.2 Update the Q value of the previous state

On reaching the current state, new Q value of the previous state action pair is computed by using the specified learning method. This project uses two types of learning methods as described in section 3.1.

This newly calculated value of previous state-action pair i.e. Q(pre, preAction) is then updated in the database.

```python
def eGreedy(numActions, E, age, currBSON):
    epsilon = math.exp(-1*E*age)
    randNum = random.uniform(0,1)

    """ split into 2 probability ranges separated by epsilon ie 0<=p<=epsilon and epsilon<p<=1
        if randNum<=epsilon, random action
        else greedy action
    """
    randAction = random.randint(0,numActions-1)
    # print(randAction, "randAction")

    if randNum<=epsilon:
        return randAction
    else:
        nextMaxQ = currBSON[0]['qVal']
        greedyAction = 0
        for i in currBSON:
            # print(i, "currG")
            if nextMaxQ < i['qVal']:
                nextMaxQ = i['qVal']
                greedyAction = i['action']
        # ensures selection of random action if multiple actions have qVal = nextMaxQ
        # i.e. when there are multiple candidates for greedy action
        selAction=[]
        for i in currBSON:
            if(i['qVal']==nextMaxQ):
                selAction.append(i['action'])
        greedyAction = selAction[random.randint(0,len(selAction)-1)]

        return greedyAction
```

Figure 5.13: ε-greedy action selection

5.3.3 Determines the next action

RL Traffic Light agent takes action based on learned Q values of state-action pair, current state of environment and reward received. This project makes use of two action selection strategies, as described in section 3.5, namely ε-greedy and Softmax.

```
43  def softmax(numActions, numberCars, age, currBSON):
44      temperature = math.exp(-1*age/int(numberCars))
45      options = numActions*[None]
46      selProb = numActions*[None]
47      selProbSum = 0
48      for i in range(0, numActions):
49          options[i] = currBSON[i]['action']
50          selProb[i] = math.exp(currBSON[i]['qVal']/temperature)
51          selProbSum += selProb[i]
52
53      for i in range(0, numActions):
54          selProb[i] /= selProbSum
55
56      return int(numpy.random.choice(options, p=selProb))
```

Figure 5.14: softmax action selection

CHAPTER 6

RESULTS AND CONCLUSION

6.1 Results

The results for this project are obtained by designing a set of experiments to test all possible combinations of RL agent parameters. This is done to compare them amongst themselves as well as with a pre-timed, non learning traffic light agent. This pre-timed agent is the benchmark for each experiment.

The design of these experiments is summarized in table 6.1.

Table 6.1: Design of Experiments

Experiment	Phasing	State	Learning	Action	Figure
Benchmark	Fixed	-	No Learning	Pre-timed	Fig. 6.1, 6.2
1	Fixed	Queue length	Q learning	$\varepsilon greedy$	Fig. 6.3, 6.4
2	Fixed	Queue length	Q learning	softmax	Fig. 6.5, 6.6
3	Fixed	Queue length	SARSA	$\varepsilon greedy$	Fig. 6.7, 6.8
4	Fixed	Queue length	SARSA	softmax	Fig. 6.9, 6.10
5	Fixed	Delay	Q learning	$\varepsilon greedy$	Fig. 6.11, 6.12
6	Fixed	Delay	Q learning	softmax	Fig. 6.13, 6.14
7	Fixed	Delay	SARSA	$\varepsilon greedy$	Fig. 6.15, 6.16
8	Fixed	Delay	SARSA	softmax	Fig. 6.17, 6.18
9	Variable	Queue length	Q learning	$\varepsilon greedy$	Fig. 6.19, 6.20
10	Variable	Queue length	Q learning	softmax	Fig. 6.21, 6.22
11	Variable	Queue length	SARSA	$\varepsilon greedy$	Fig. 6.23, 6.24
12	Variable	Queue length	SARSA	softmax	Fig. 6.25, 6.26
13	Variable	Delay	Q learning	$\varepsilon greedy$	Fig. 6.27, 6.28
14	Variable	Delay	Q learning	softmax	Fig. 6.29, 6.30
15	Variable	Delay	SARSA	$\varepsilon greedy$	Fig. 6.31, 6.32
16	Variable	Delay	SARSA	softmax	Fig. 6.33, 6.34

Each of these experiments and obtaining their plots has been automated by using a python script. Each of these experiments are iterated three time, each iteration of 2lakh vehicles. The results for 1st and 3rd iterations of these experiments are shown in the figure 6.1 to figure 6.34.

6.1.1 Experiment Benchmark

Figure 6.1: Experiment Benchmark: Average Waiting Time

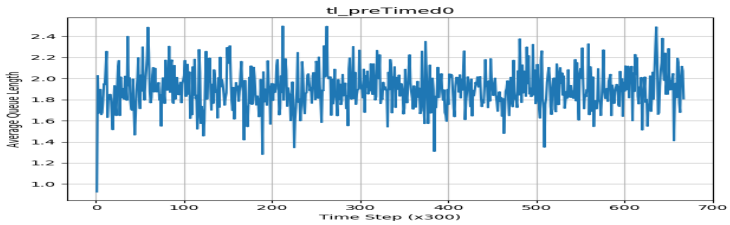
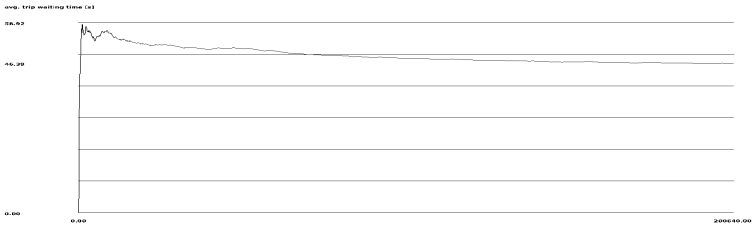
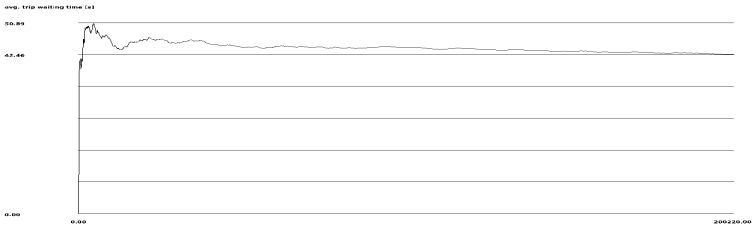


Figure 6.2: Experiment Benchmark: Average Queue Length

6.1.2 Experiment 1

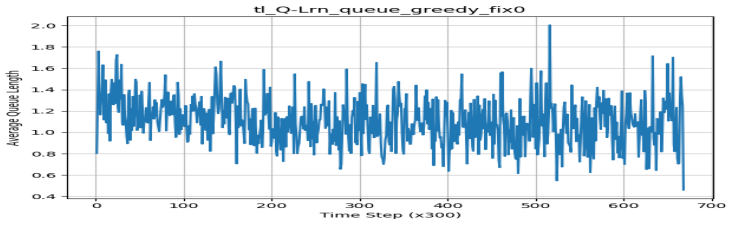


(a) 1st Iteration

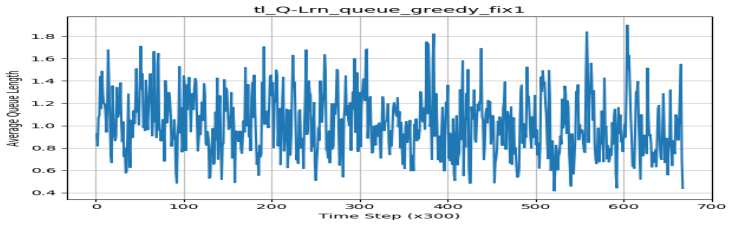


(b) 3rd Iteration

Figure 6.3: Experiment 1: Average Waiting Time



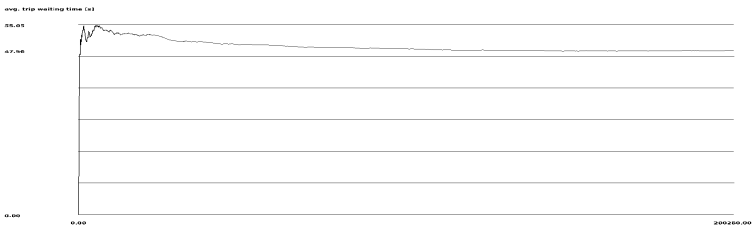
(a) 1st Iteration



(b) 3rd Iteration

Figure 6.4: Experiment 1: Average Queue Length

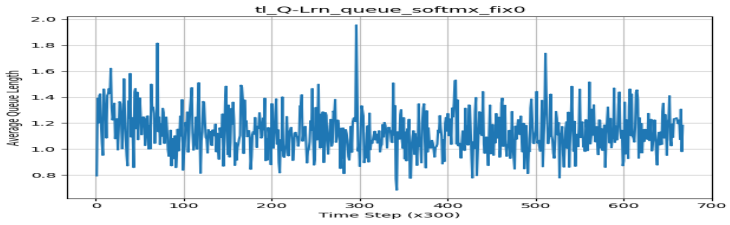
6.1.3 Experiment 2



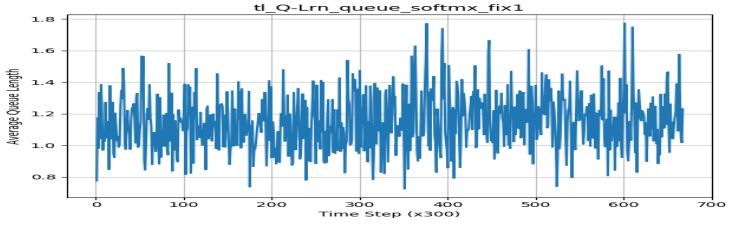
(a) 1st Iteration

(b) 3rd Iteration

Figure 6.5: Experiment 2: Average Waiting Time



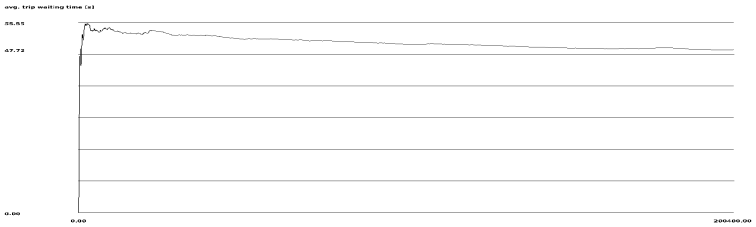
(a) 1st Iteration



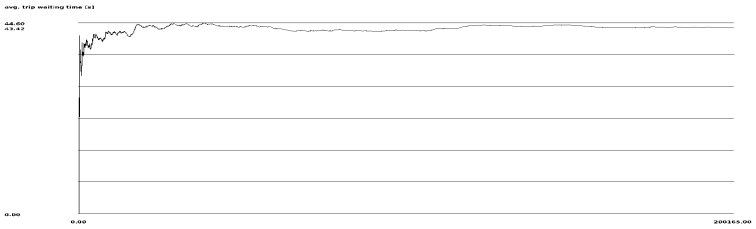
(b) 3rd Iteration

Figure 6.6: Experiment 2: Average Queue Length

6.1.4 Experiment 3

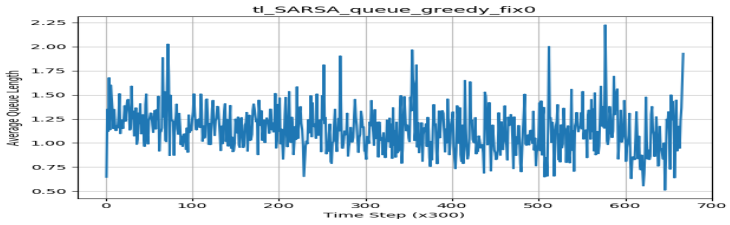


(a) 1st Iteration

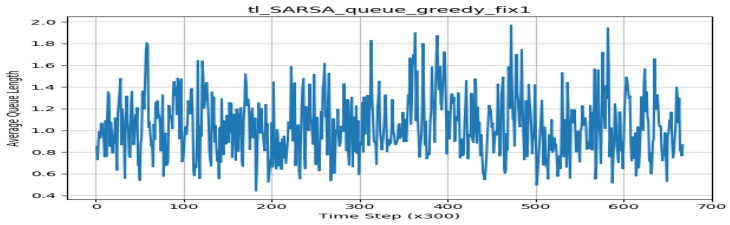


(b) 3rd Iteration

Figure 6.7: Experiment 3: Average Waiting Time



(a) 1st Iteration

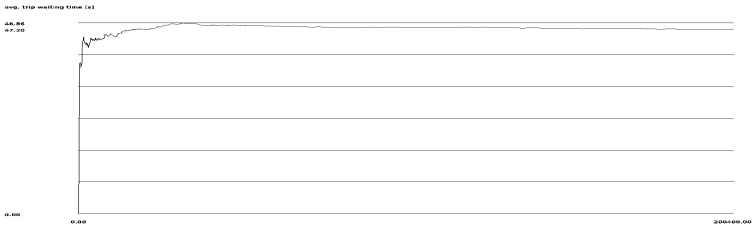


(b) 3rd Iteration

Figure 6.8: Experiment 3: Average Queue Length

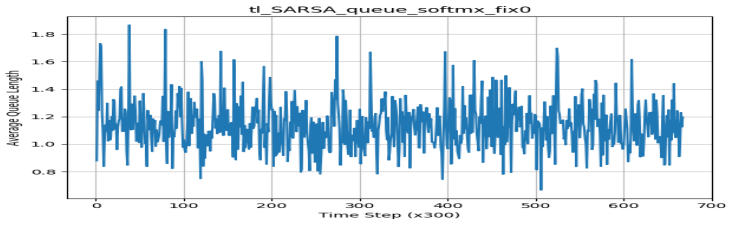
6.1.5 Experiment 4

(a) 1st Iteration

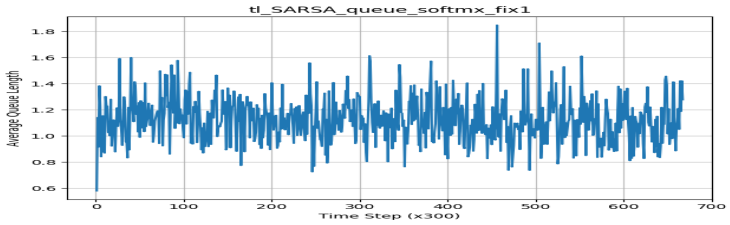


(b) 3rd Iteration

Figure 6.9: Experiment 4: Average Waiting Time



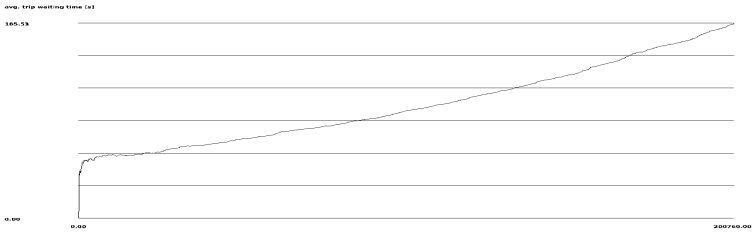
(a) 1st Iteration



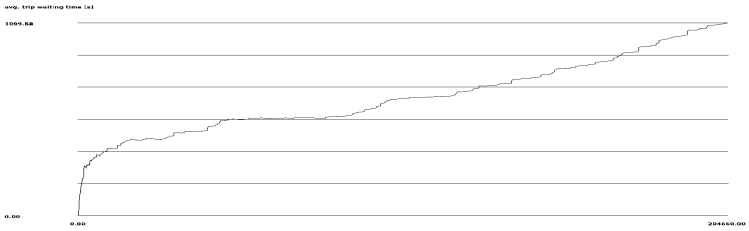
(b) 3rd Iteration

Figure 6.10: Experiment 4: Average Queue Length

6.1.6 Experiment 5

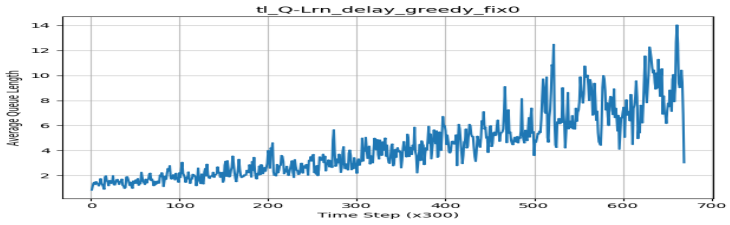


(a) 1st Iteration

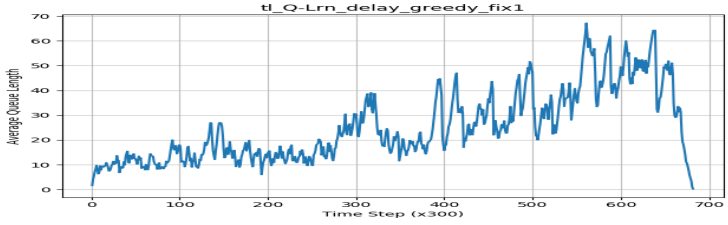


(b) 3rd Iteration

Figure 6.11: Experiment 5: Average Waiting Time



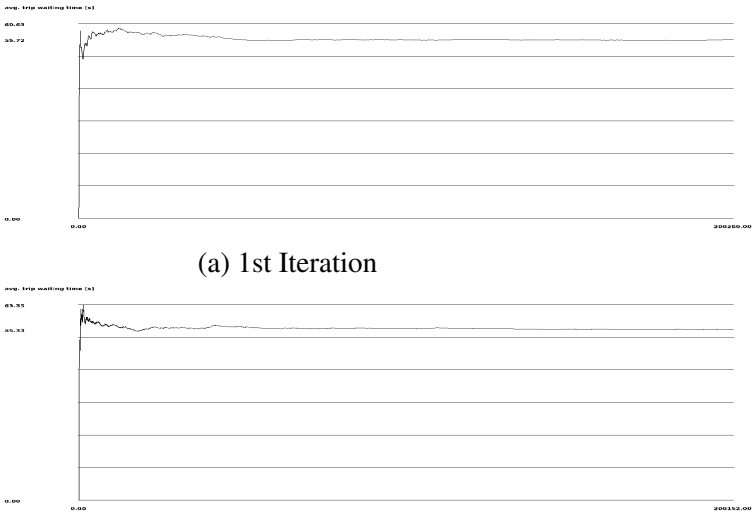
(a) 1st Iteration



(b) 3rd Iteration

Figure 6.12: Experiment 5: Average Queue Length

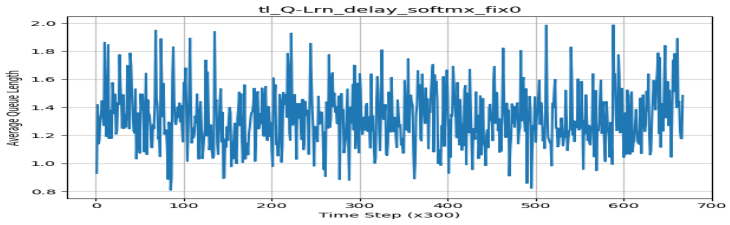
6.1.7 Experiment 6



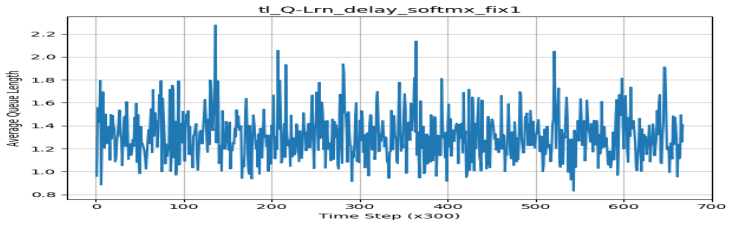
(a) 1st Iteration

(b) 3rd Iteration

Figure 6.13: Experiment 6: Average Waiting Time



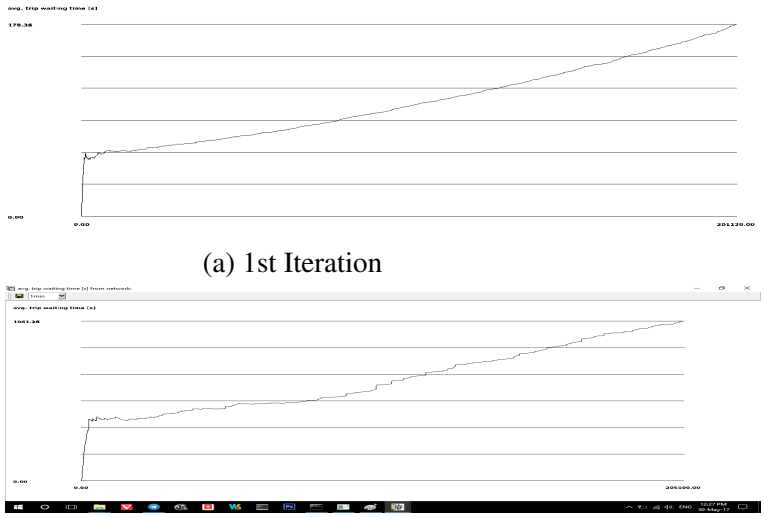
(a) 1st Iteration



(b) 3rd Iteration

Figure 6.14: Experiment 6: Average Queue Length

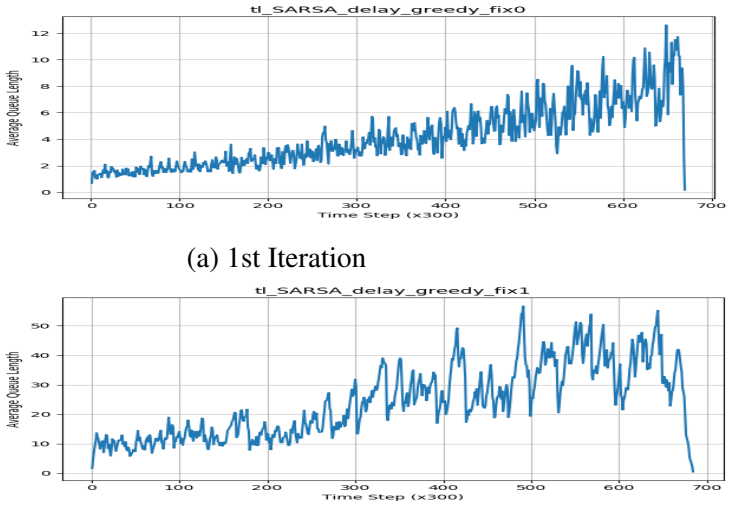
6.1.8 Experiment 7



(a) 1st Iteration

(b) 3rd Iteration

Figure 6.15: Experiment 7: Average Waiting Time

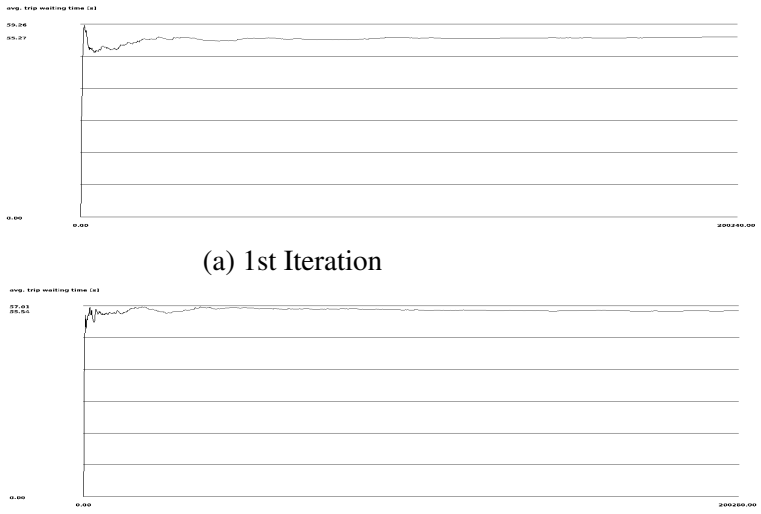


(a) 1st Iteration

(b) 3rd Iteration

Figure 6.16: Experiment 7: Average Queue Length

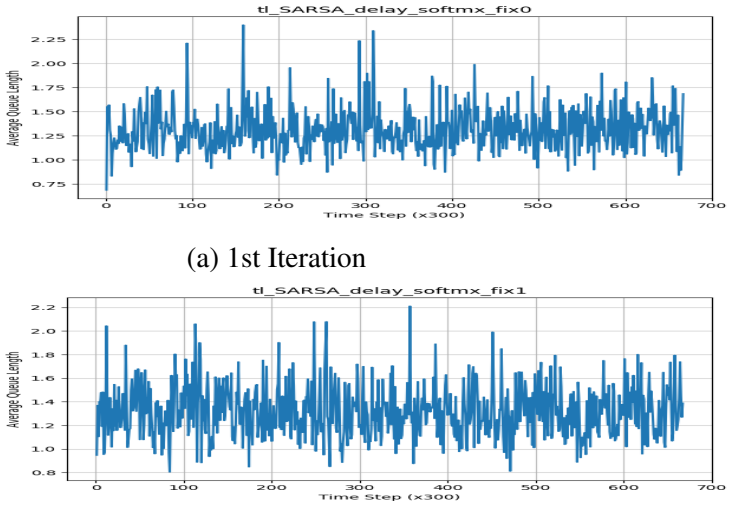
6.1.9 Experiment 8



(a) 1st Iteration

(b) 3rd Iteration

Figure 6.17: Experiment 8: Average Waiting Time

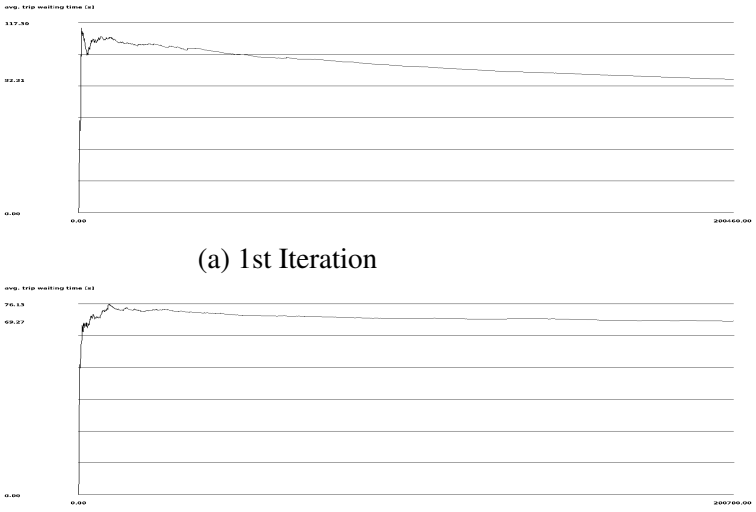


(a) 1st Iteration

(b) 3rd Iteration

Figure 6.18: Experiment 8: Average Queue Length

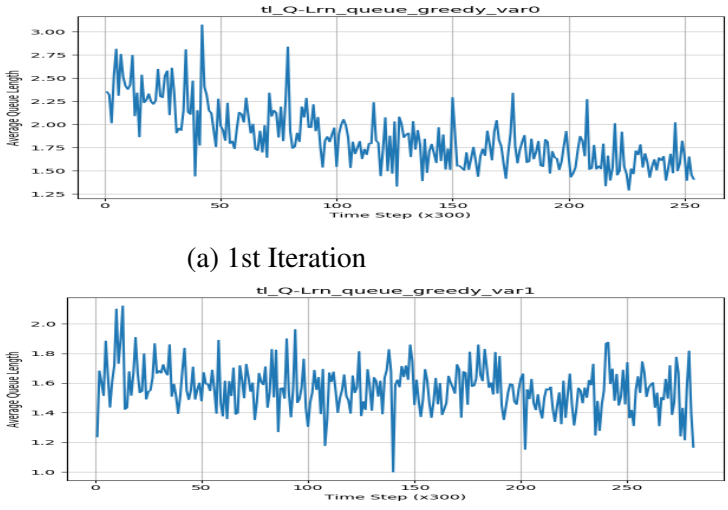
6.1.10 Experiment 9



(a) 1st Iteration

(b) 3rd Iteration

Figure 6.19: Experiment 9: Average Waiting Time

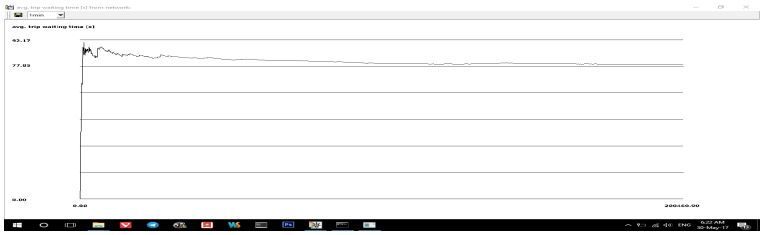


(a) 1st Iteration

(b) 3rd Iteration

Figure 6.20: Experiment 9: Average Queue Length

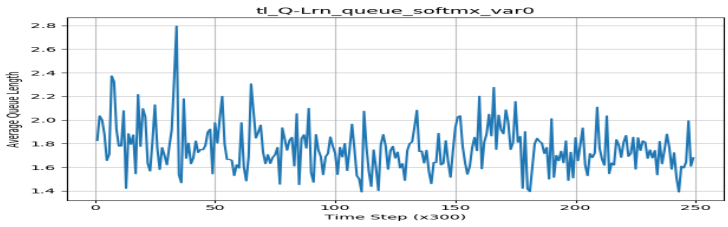
6.1.11 Experiment 10



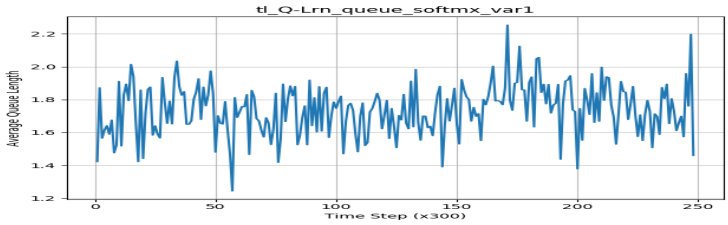
(a) 1st Iteration

(b) 3rd Iteration

Figure 6.21: Experiment 10: Average Waiting Time



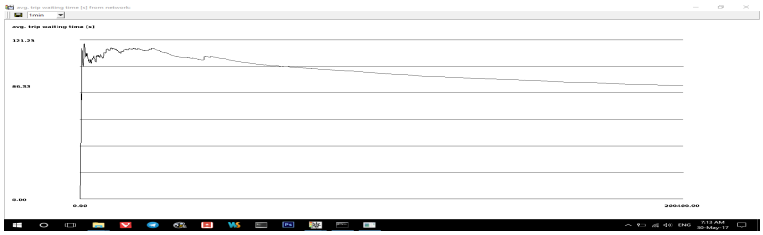
(a) 1st Iteration



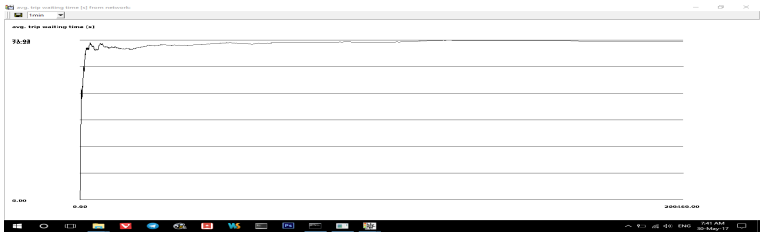
(b) 3rd Iteration

Figure 6.22: Experiment 10: Average Queue Length

6.1.12 Experiment 11

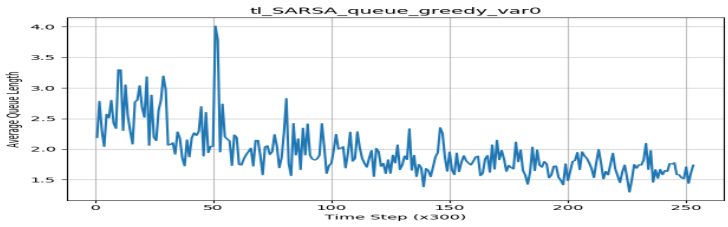


(a) 1st Iteration

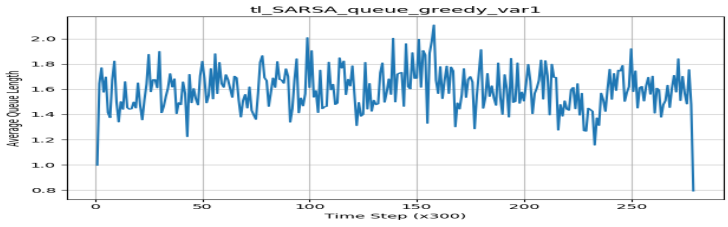


(b) 3rd Iteration

Figure 6.23: Experiment 11: Average Waiting Time



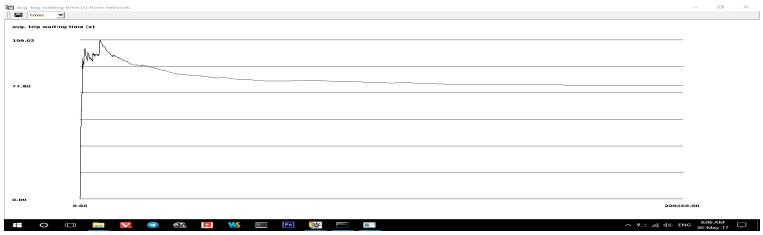
(a) 1st Iteration



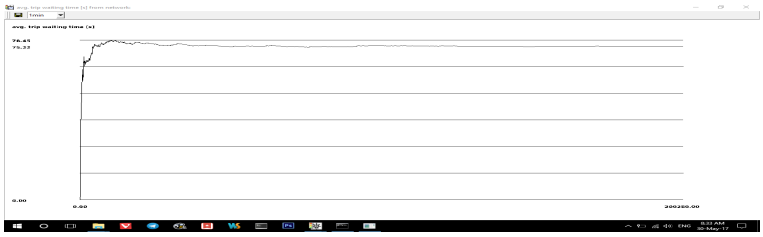
(b) 3rd Iteration

Figure 6.24: Experiment 11: Average Queue Length

6.1.13 Experiment 12

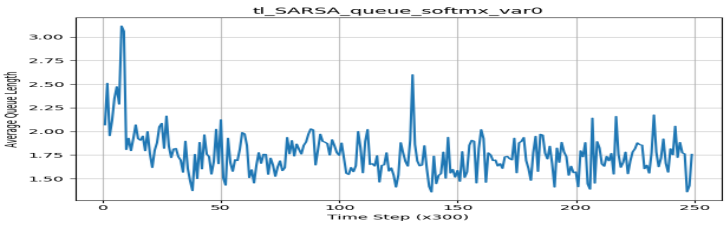


(a) 1st Iteration

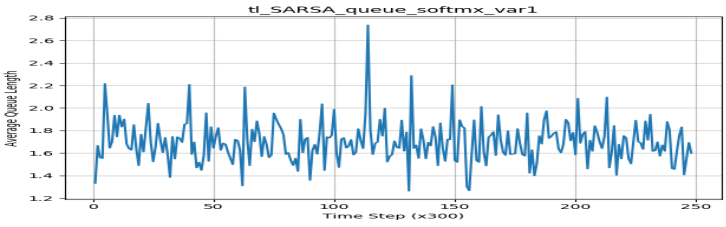


(b) 3rd Iteration

Figure 6.25: Experiment 12: Average Waiting Time



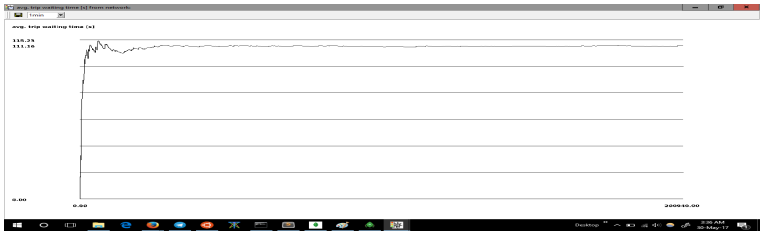
(a) 1st Iteration



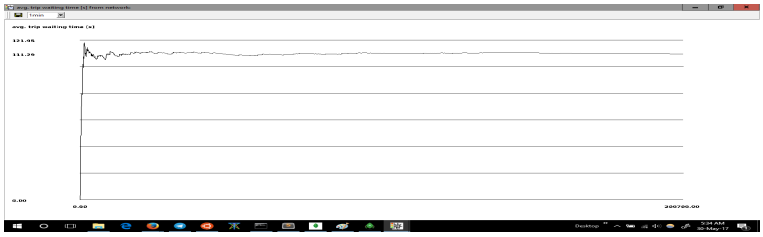
(b) 3rd Iteration

Figure 6.26: Experiment 12: Average Queue Length

6.1.14 Experiment 13

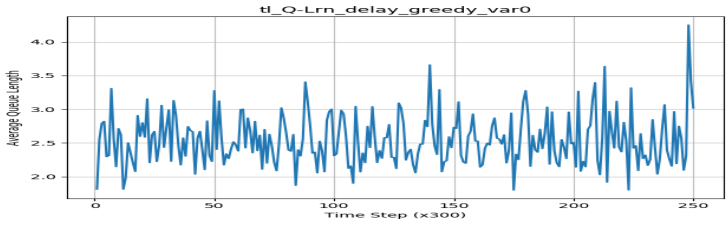


(a) 1st Iteration

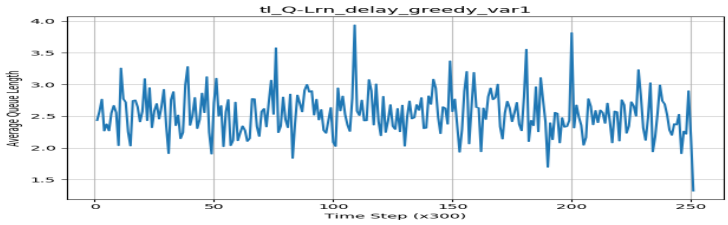


(b) 3rd Iteration

Figure 6.27: Experiment 13: Average Waiting Time



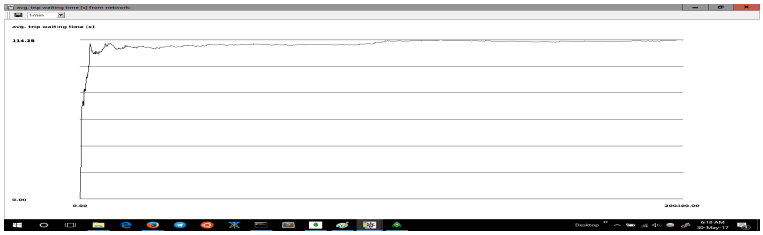
(a) 1st Iteration



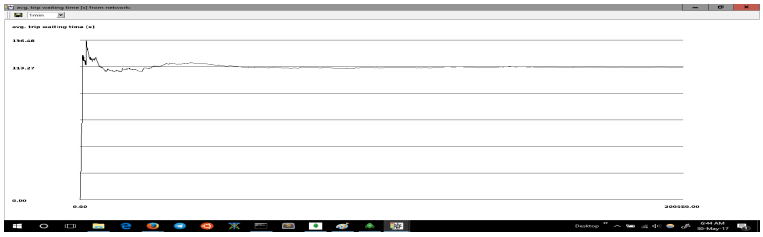
(b) 3rd Iteration

Figure 6.28: Experiment 13: Average Queue Length

6.1.15 Experiment 14

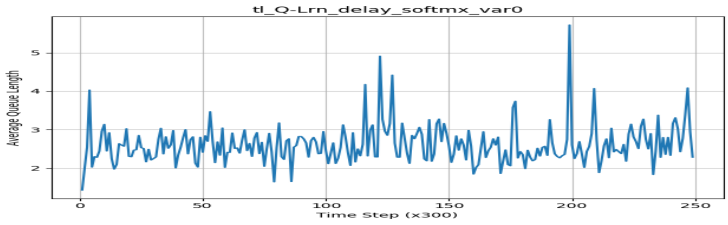


(a) 1st Iteration

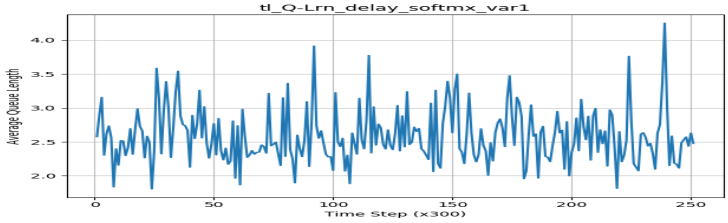


(b) 3rd Iteration

Figure 6.29: Experiment 14: Average Waiting Time



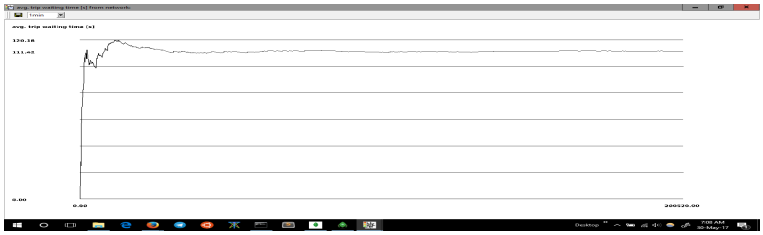
(a) 1st Iteration



(b) 3rd Iteration

Figure 6.30: Experiment 14: Average Queue Length

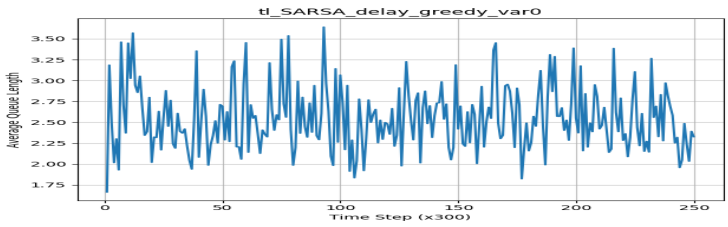
6.1.16 Experiment 15



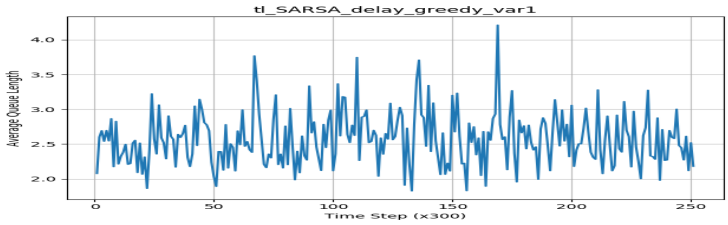
(a) 1st Iteration

(b) 3rd Iteration

Figure 6.31: Experiment 15: Average Waiting Time



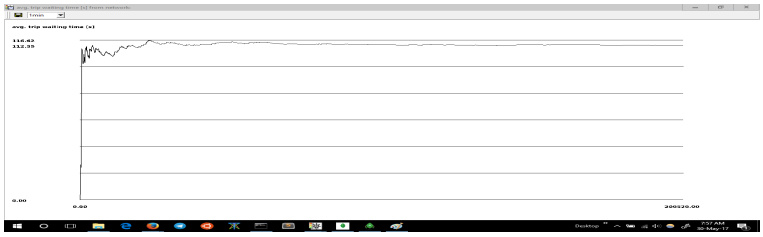
(a) 1st Iteration



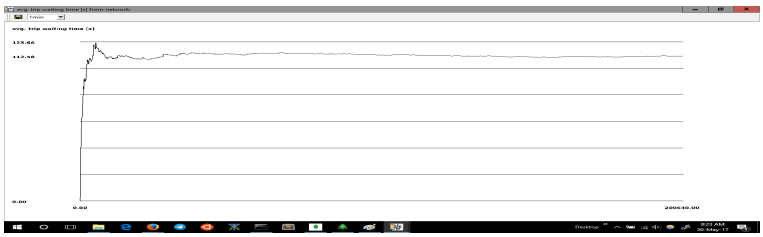
(b) 3rd Iteration

Figure 6.32: Experiment 15: Average Queue Length

6.1.17 Experiment 16

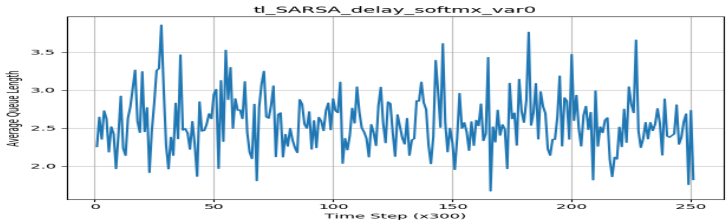


(a) 1st Iteration

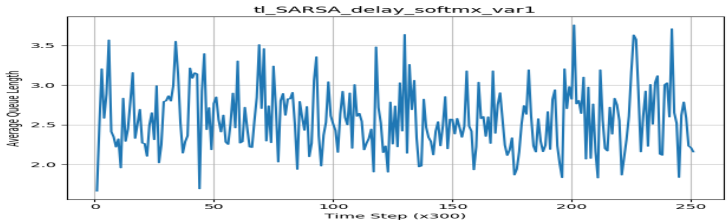


(b) 3rd Iteration

Figure 6.33: Experiment 16: Average Waiting Time



(a) 1st Iteration



(b) 3rd Iteration

Figure 6.34: Experiment 16: Average Queue Length

6.2 Conclusion

This section presents the conclusions of this project. These conclusions are drawn from results of experiments specified in the previous section, presented in figures 6.1 to 6.34.

6.2.1 Comparison of phasing sequences

On an average, variable phasing performs better than fixed phasing. Variable phasing doesn't give worst case results and its performance is relatively constant, unlike experiments 5 and 7 for Fixed phasing. This can be shown by comparing figures of experiment 5,7 and experiments 13,15.

Fixed phasing performs better than the benchmark i.e. no learning, pre-timed case (Experiment Benchmark). Performance of Variable phasing is either comparable or slightly worse than the no learning case but the added advantage of variability of phasing sequence compensates for this error.

6.2.2 Comparison of state/reward definitions

Queue length state definition (along with minimizing and balancing queue lengths reward) performs better than Cumulative delay state definition (along with maximizing reduction in Cumulative delay reward) for every case.

This is due to the fact that cumulative delay representation/reward does not guarantee prevention of queue spilling back and therefore affecting other intersections and minor streets.

6.2.3 Comparison of learning methods

On an average, performance of both Q-Learning and SARSA learning methods is comparable most of the times. Q-learning performs minutely better than SARSA in all cases except for cumulative delay reward and variable phasing where performance of SARSA is slightly better than Q-learning. This is evident on comparing results of experiment 13 to 16, as shown in figures 6.27 to 6.34.

6.2.4 Comparison of action selection techniques

Softmax performs better than e-greedy as its performance is constant and does not result into worst case scenarios, unlike experiment 5 and 7 for e-greedy. This is clear from figures 6.11 to 6.16.

The performance of e-greedy improves as it runs for more time, as is evident from comparing figures 6.19a to 6.19b and 6.23a to 6.23b.

CHAPTER 7

SYSTEM DIAGRAMS

7.1 Entity Relationship (ER) Diagram

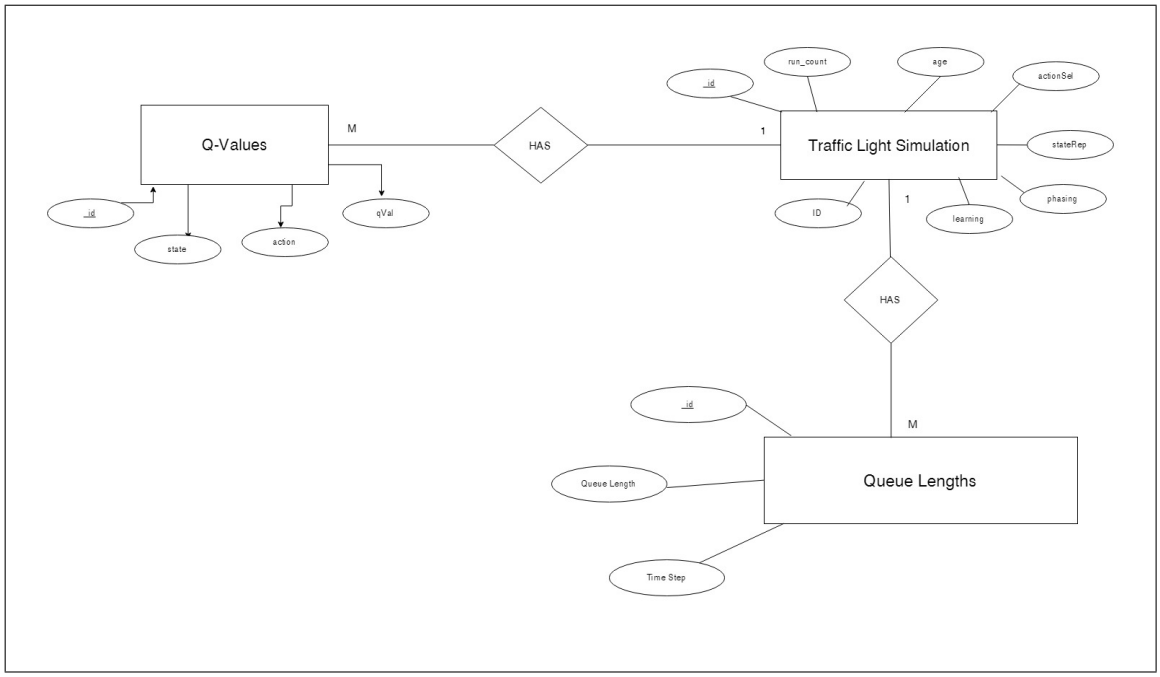


Figure 7.1: Entity Relationship Diagram

7.2 Data Flow Diagram (DFD) Diagram

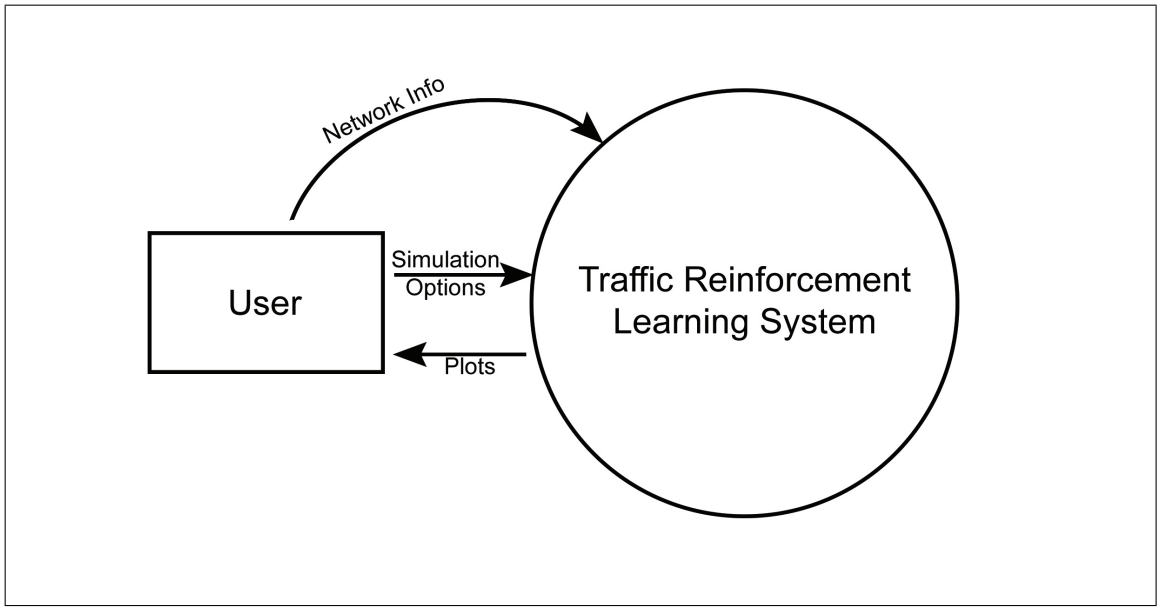


Figure 7.2: Level 0 DFD

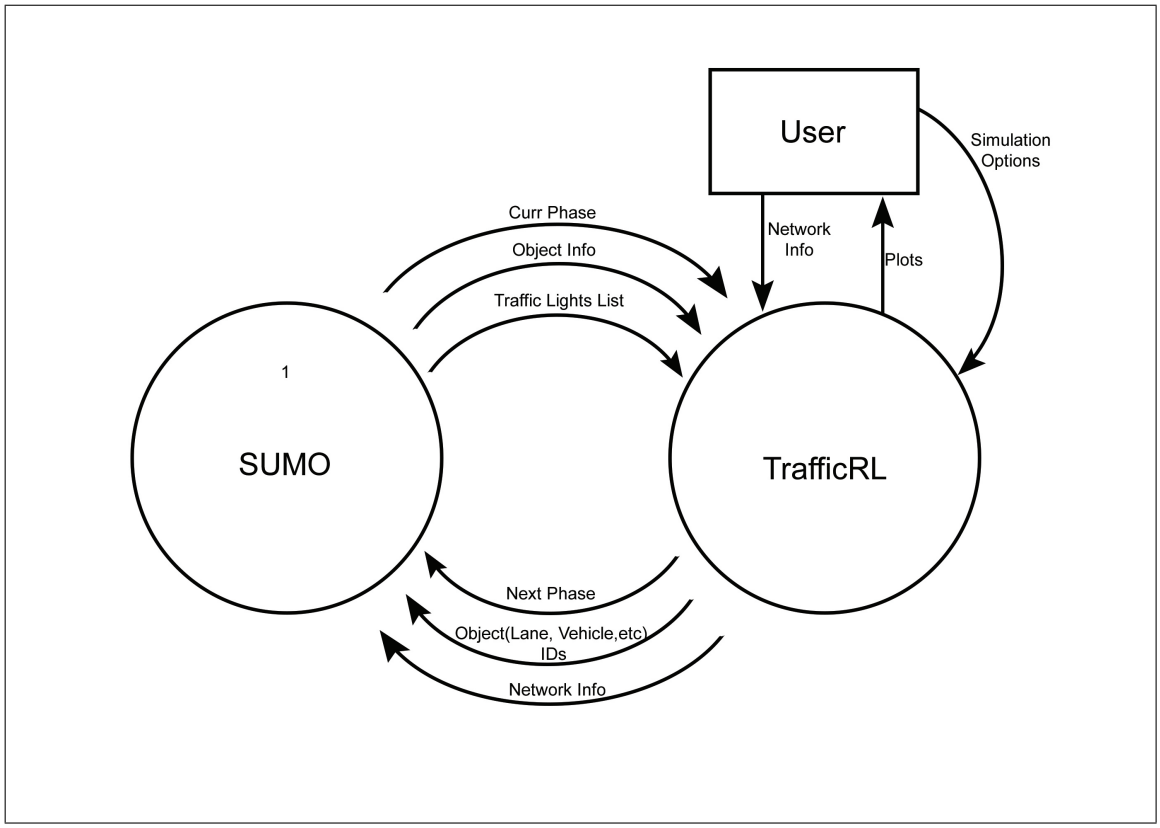


Figure 7.3: Level 1 DFD

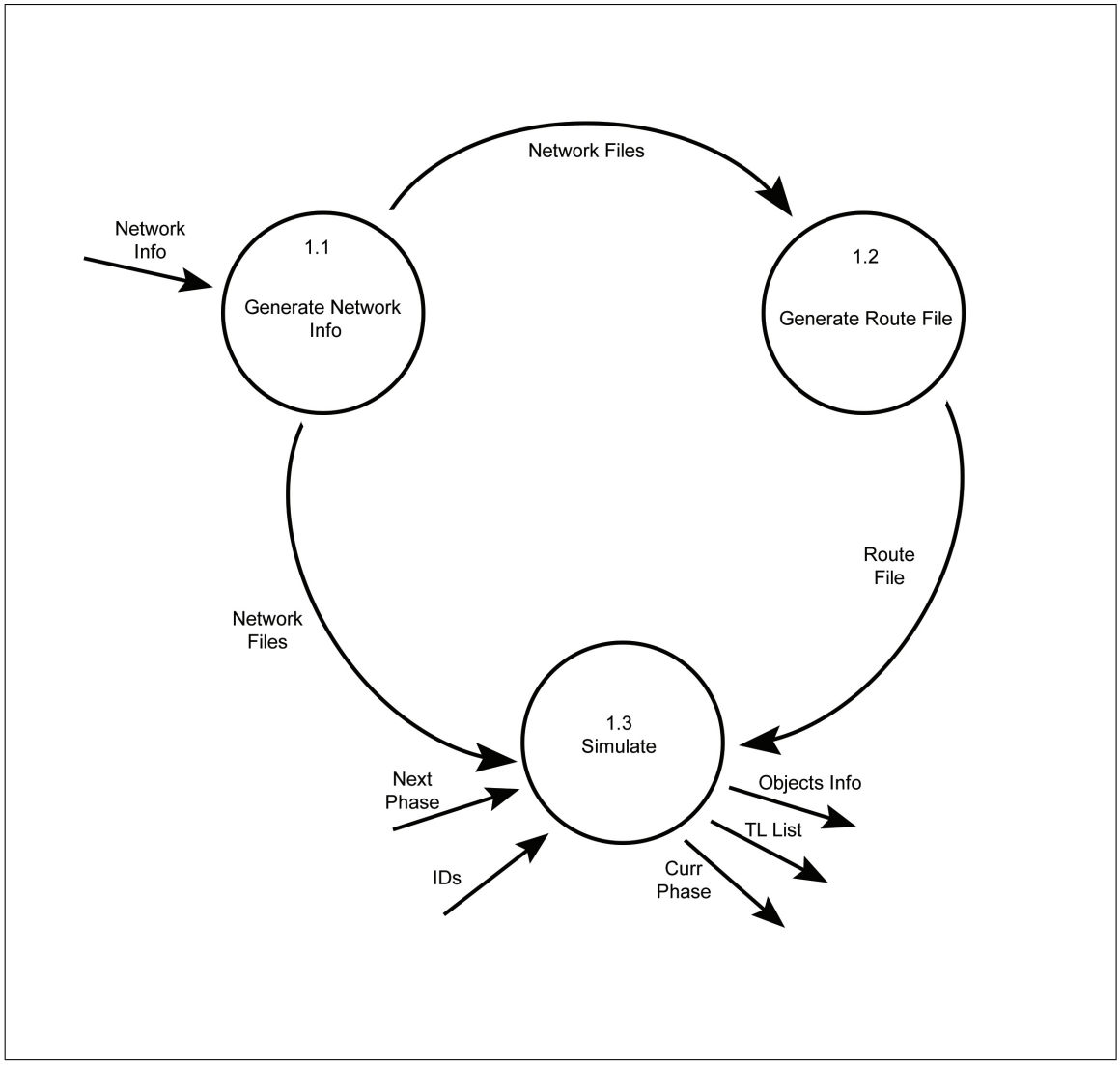


Figure 7.4: Level 2 DFD - Figure 1

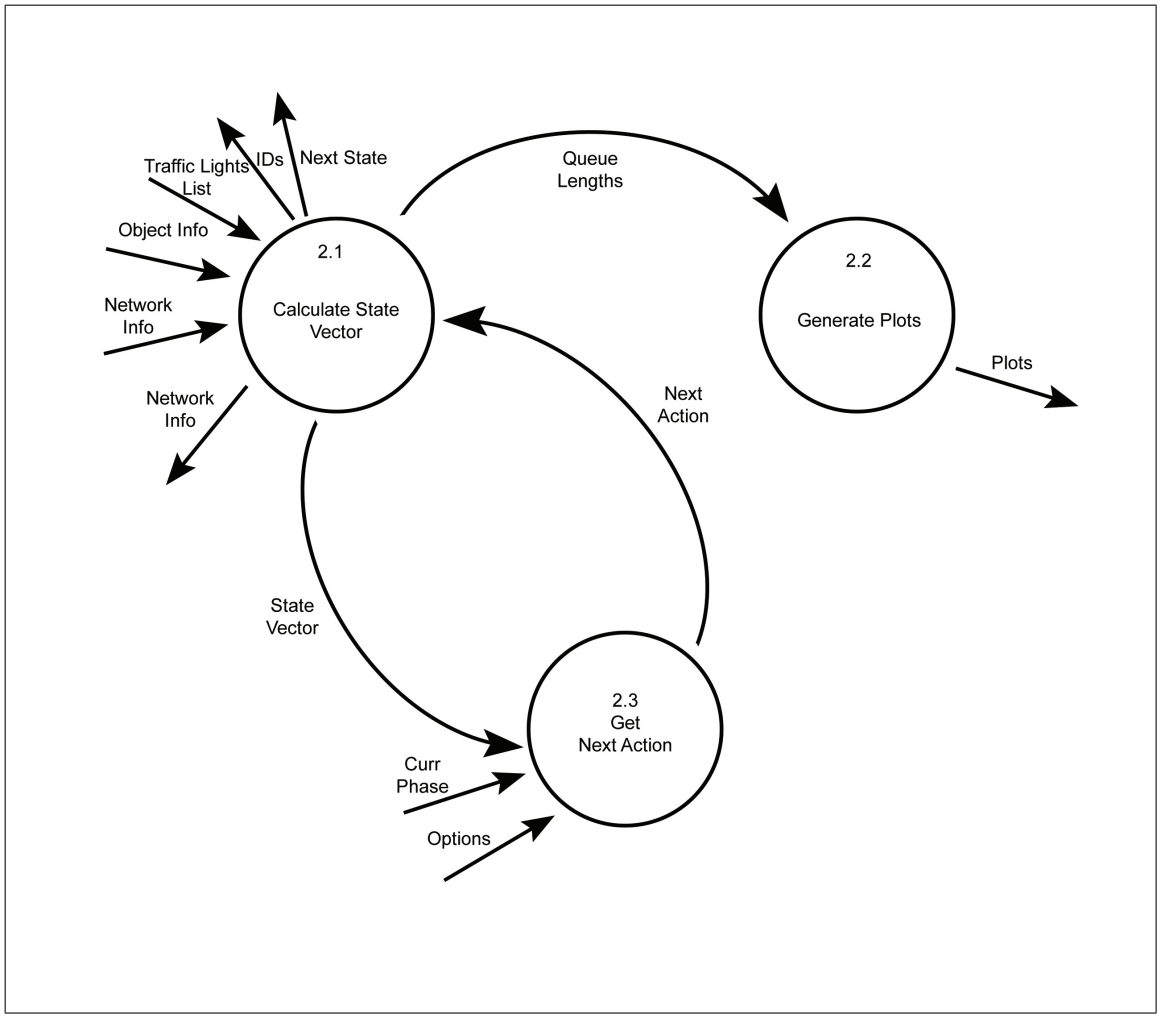


Figure 7.5: Level 2 DFD - Figure 2

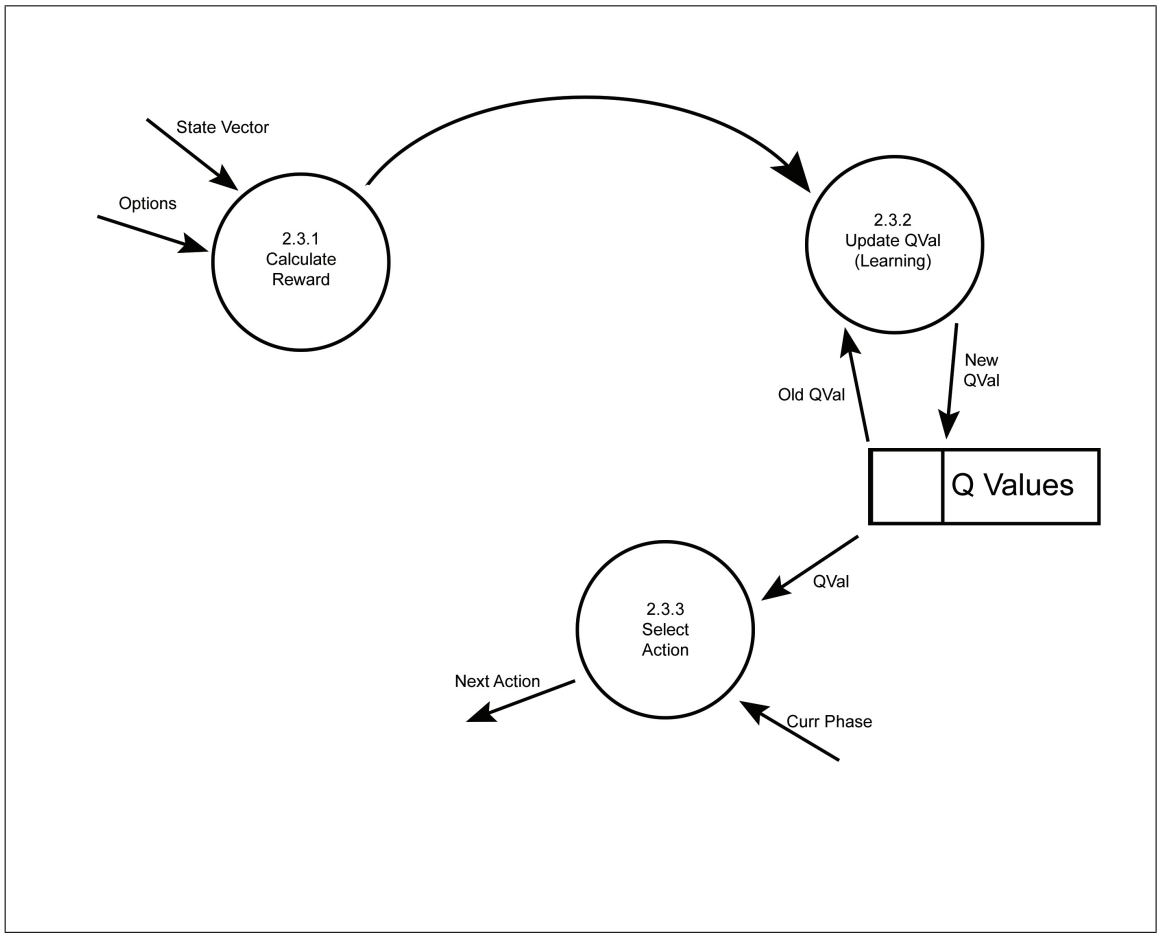


Figure 7.6: Level 3 DFD

7.3 Use Case Diagram

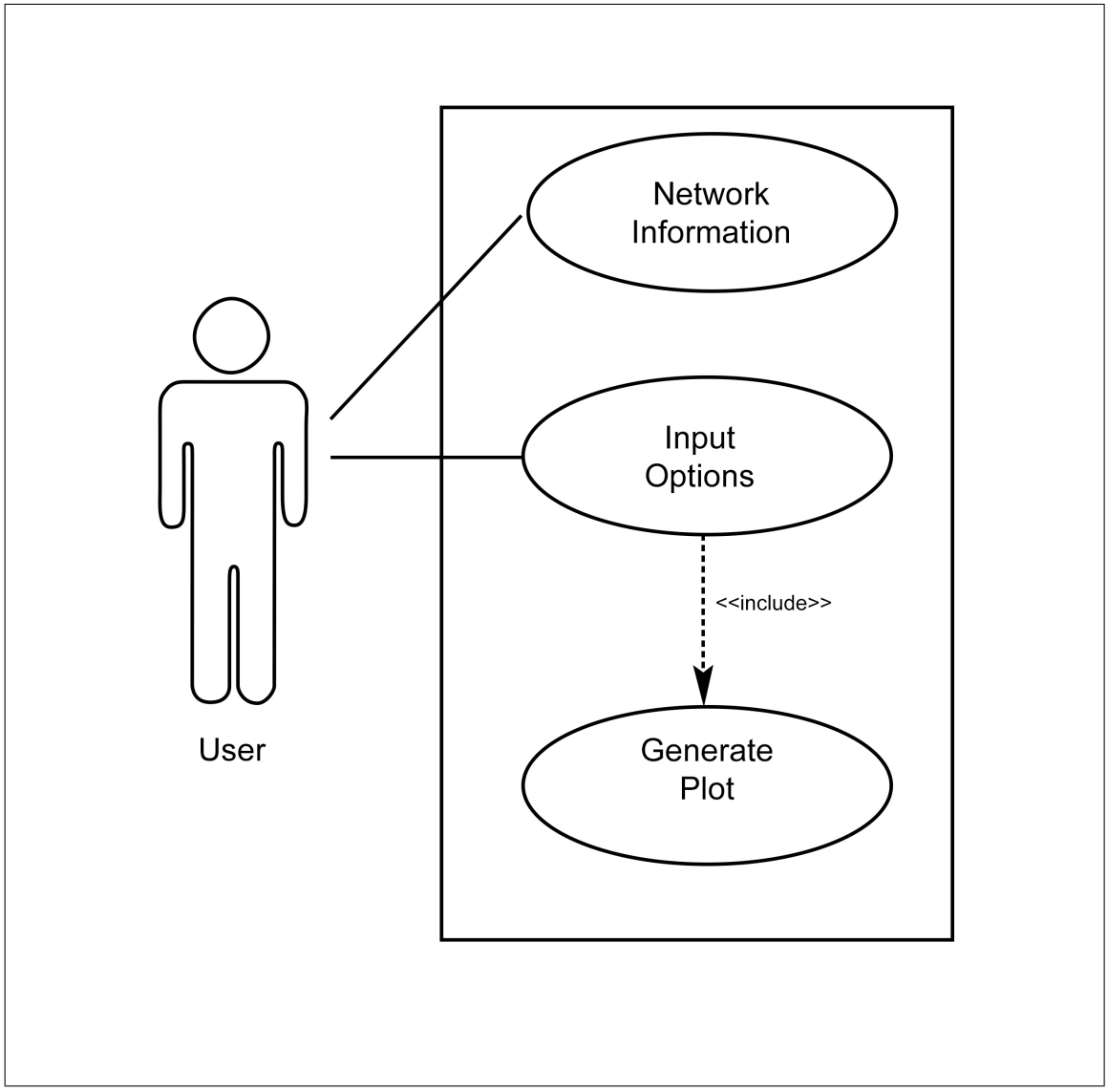


Figure 7.7: Use Case Diagram

APPENDICES

APPENDIX A

PYTHON MATHEMATICAL LIBRARIES

Numpy

It is the fundamental package for scientific computing with Python. It contains features like a powerful N-dimensional array object,useful linear algebra, Fourier transform, and random number capabilities Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Scipy

SciPy is an open source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering. SciPy builds on the NumPy array object and is part of the NumPy stack which includes tools like Matplotlib, pandas and SymPy, and an expanding set of scientific computing libraries. This NumPy stack has similar users to other applications such as MATLAB, GNU Octave, and Scilab. The NumPy stack is also sometimes referred to as the SciPy stack

APPENDIX B

DATABASE TOOLS

MongoDB

MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas. MongoDB is developed by MongoDB Inc. and is free and open-source, published under a combination of the GNU Affero General Public License and the Apache License. The database is used to store Q-values of traffic Lights.

PyMongo

PyMongo is a Python library containing tools for working with MongoDB, and is the recommended way to work with MongoDB from Python. It is an open source distribution and is avtively maintained on Github. It provides easy tools to read, write and manipulate the MongoDB database.

APPENDIX C

SOURCE CODE

The source code written for this project is available freely online at

https://github.com/codemerlin19/trafficRL

BIBLIOGRAPHY

[1] SUMO main Website -

http://sumo.dlr.de/wiki/Simulation_of_Urban_MObility_-_Wiki

[2] Python web documentation -

https://www.python.org/doc/

[3] Delhi Government report on vehicular distribution -

http://www.delhi.gov.in/wps/wcm/connect/DoIT_Planning/planning/economic+survey+of+dehli/content/transport

[4] DeepMind Reinforcement Learning Tutorials -

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

[5] UC Berkeley Lectures -

http://ai.berkeley.edu/lecture_videos.html

[6] El-Tantawy, Samah, Baher Abdulhai, and Hossam Abdelgawad. "Design Of Reinforcement Learning Parameters For Seamless Application Of Adaptive Traffic Signal Control". Journal of Intelligent Transportation Systems 18.3 (2014): 227-245. Web. 30 May 2017.