Application of Markov Decision Processes to the Control of a Traffic Intersection

Onivola Henintsoa Minoarivelo (ony@aims.ac.za)
Supervised by: Doctor Jesús Cerquides Bueno
University of Barcelona, Spain

22 May 2009
Submitted in partial fulfillment of a postgraduate diploma at AIMS

Abstract

With the rapid development of technology in urban area, the number of circulating vehicles is surprisingly increasing. This leads to a difficulty in the coordination of the traffic network. This essay approaches the problem of managing the functioning of a junction by setting up an efficient decision strategy taken by the traffic controller regarding the time the road users can waste at a crossing. For this purpose, reinforcement learning method will be used. The system will be modelled as Markov decision processes, and the value iteration algorithm will be implemented to solve the optimisation problem. We will conduct a simulation to see the performance of the proposed decision strategy.

Declaration

I, the undersigned, hereby declare that the work contained in this essay is my original work, and that any work done by others or by myself previously has been acknowledged and referenced accordingly.

Onivola Henintsoa Minoarivelo, 22 May 2009

Contents

Abstract i

1 Introduction 1

2 Reinforcement Learning 2

2.1 Introduction . . . 2

2.2 Elements of Reinforcement Learning . . . 2

2.3 Markov Decision Processes . . . 3

2.3.1 Definition and Characteristics . . . 3

2.3.2 Policy and Value Functions . . . 4

2.4 Reinforcement Learning Methods . . . 6

2.4.1 Solving a MDP by Value Iteration Algorithm . . . 6

2.4.2 Q-learning . . . 6

2.4.3 SARSA . . . 8

2.4.4 Model-based Reinforcement Learning . . . 8

3 Related Work to Traffic Control 10

3.1 Reinforcement Learning Approach . . . 10

3.2 Other Approaches . . . 11

3.2.1 Fuzzy Logic . . . 11

3.2.2 Artificial Neural Network . . . 11

3.2.3 Evolutionary Algorithms . . . 12

4 SUMO 13

4.1 Overview and Definition . . . 13

4.2 Building a Simulation with SUMO . . . 13

4.2.1 Building the Network . . . 13

4.2.2 Route Generation . . . 17

4.3 Detectors . . . 20

4.4 Intelligent Traffic Light . . . 21

5 Formalisation of a Traffic System 22

5.1 Problem Statement 22
5.2 Formalisation of the Problem as a Markov Decision Processes 22
5.2.1 Junction model 22
5.2.2 Junction Control through MDPs 23

6 Solution and Experimentation **29**
6.1 The Decision Strategy 29
6.2 Simulation 29
6.3 Results and Interpretations 31
6.3.1 Results 31
6.3.2 Interpretation 32

7 Conclusion **34**

A Implementation of the Value Iteration Code for the Traffic Control Decision Strategy **35**

References **42**

1. Introduction

In modern society, efficiently managing traffic becomes more and more of a challenge. Due to the progressive evolution of road traffic nowadays, traffic management is still a evolving problem. For decades, many researchers in engineering, as well as in computer science, have approached the problem in different ways. In urban areas, the traffic is controlled, in large, by traffic lights. Unfortunately, some traffic light configurations are still based on a 'fixed-cycle' decision. That is, lights are green during a fixed period of time, and red during the next period. This can lead to a long waiting time for some vehicles at one side of the crossing, even if when there are no vehicles passing on the opposite open road. And a long traffic jam may appear easily.

This work is especially devoted to the optimisation of the decision strategy applied to control a traffic light system at a crossroad. The goal is to minimise the total waiting time of vehicles around the junction. For this purpose, the entity which takes the decision, called an 'agent' has to learn the behaviour and the evolution of the state of the crossing. In this way, reinforcement learning, which consists of solving learning problems by studying the system through mathematical analysis or computational experiments, is considered to be the most adequate approach. As in solving any real life problem, setting a formalisation is an important first step. Being applicable to a large range of problems, Markov decision processes have been commonly used as a mathematical model for reinforcement learning. They will serve as a framework to formalise the system in this work. Once the decision strategy is set up, its performance should be checked. In this essay, SUMO, an open source package for a traffic simulation, will be adapted to simulate a crossroad working under the control of the 'intelligent traffic light'. Computational experiments will be conducted in order to compare our model with other decision strategies.

Therefore, in order to carry out this work, we will provide the theoretical background for reinforcement learning and Markov decision processes. We will regard some overviews, definitions and furthermore, explain the functioning of algorithms that solve Markov decision processes. Then, a review of related work will be presented. We will consider other approaches which deal with the decision problem of traffic light control, especially those using reinforcement learning and Markov decision processes. Because the whole simulation is constructed by the means of SUMO, chapter 4 will exclusively talk about SUMO. More precisely, it will consist of the description of the way SUMO is working. Chapter 5 explains, in detail, the formalisation of the problem of traffic light control as a Markov decision process. And finally, the last chapter will talk about the way we can put the formalisation into experiments, via implementation of a python script and via SUMO. In this last chapter, we will also interpret the results of the experiments.

2. Reinforcement Learning

2.1 Introduction

Every day, we interact with our environment and learn something from it. In some situations, we are supposed to take decisions, regarding the state of our environment. Reinforcement learning explore a computational approach to learning from this interaction. Precisely, it can be defined as a sub-area of machine learning concerned with how to take a decision according to a specific situation in order to maximise the reward obtained by taking the decision. In reinforcement learning, the entity which takes the decision is called *agent*. It can be a human, a robot, a part of a machine or anything susceptible to take a decision, and the *environment* is what is all around the agent and which can influence its decision. A good way to understand the concept of reinforcement learning is to see some examples of situations at which it can be applied.

In a chess game, the agents are the players and they act by considering a goal which is to win. So, every action of a player is considered as a decision which influences its environment by putting the game at a new state. Reinforcement learning can be applied to learn how to make good decision in every possible states of the game.

Reinforcement learning can also be used in a more technical situation, like the approach of learning a performant decision strategy to control the functioning of a road intersection. We will discuss this example of use of reinforcement learning in this work.

In these examples, the agent has a goal to be reached and acts according to it and the available state of its environment.

Reinforcement learning has two main features [SB98]:

- Trial and error search:

  The agent has to discover by itself which actions lead to the biggest reward, by trying them.

- Delayed reward:

  It considers more strongly the goal of a long term reward , rather than maximising the immediate reward.

2.2 Elements of Reinforcement Learning

Reinforcement learning has four principal elements:

1. The policy: It is a map from the perceived states of the environment to the set of actions that can be taken.
2. The reward function: It is a function that assigns a real number to each pair (s, a) where s is a state in the set of possible states S and a is an action in the actions set A.

$$\begin{aligned} R : S \times A &\longrightarrow \mathbb{R} \\ (s, a) &\longmapsto R(s, a) \end{aligned}$$

$R(s,a)$ represents the reward obtained by the agent when performing the action a at state s. Generally, we note the value of the reward at time t: r_t.

3. The value function: It is the total amount of expected rewards from a specific state of the system, from a specific time, and over the future. It is an evaluation of what is good in the long term. The value taken by this function is generally named *return* and is denoted R_t at time t.

   We have two ways to count the return:

   - We can sum up all the rewards. This is especially used for episodic tasks, when there is a time stop T. In that case:
   $$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$
   - Concept of discounting: It is used to weight present reward more heavily than future ones with a discount rate γ, $0 < \gamma \leq 1$. The discount rate γ is necessary to have a present value of the future rewards. In this case, the return is given by the infinite sum:
   $$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$
   Since the discount rate γ is less than 1 and the series of rewards r_k bounded, this infinite sum has a finite value. If γ tends to 0, this agent is said to be myopic: it sees only immediate rewards. When γ tends to 1, the future rewards are more taken into account.

4. A model of environment: It is set to plan the future behaviour of the environment.

The most important task of reinforcement learning is determining a good policy which maximises the return. Markov Decision Processes (MDPs) are a mathematical model for reinforcement learning that is commonly used nowadays. Current theories of reinforcement learning are usually restricted to finite MDPs.

2.3 Markov Decision Processes

2.3.1 Definition and Characteristics

A learning process is said to have the *Markov property* when it is retaining all relevant information about the future. Precisely, when all needed information to predict the future can be available at the present, and the state of the system at the present can resume past information. So, in that case, we can predict the future values of the rewards simply by iteration. This statement can be written as:

$$P\left\{S^{t+1} = s', r_{t+1} = r | S^t, a^t, r_t, S^{t-1}, a^{t-1}, r_{t-1}, \dots\right\} = P\left\{S^{t+1} = s', r_{t+1} = r | S^t, a^t, r_t\right\}$$

where $P\left\{S^{t+1} = s', r_{t+1} = r | S^t, a^t, r_t, S^{t-1}, a^{t-1}, r_{t-1}, \dots\right\}$ is the probability that at time $(t+1)$, we are at the state s', wining a reward r given that we took the action $a^t, a^{t-1}, a^{t-2}, \dots$ at the states $S^t, S^{t-1}, S^{t-2}, \dots$ respectively.

A decision making which has this property can be treated by MDPs. When the space of the states is finite, the process is referred to as *finite MDP*, and consists of a 4-tuple:(S, A, R, P)

1. The set of states S: It is the finite set of all possible states of the system.

2. The finite set of actions A.
3. The reward function: $R(s,a)$ depends on the state of the system and the action taken. We assume that the reward function is bounded.
4. The Markovian transition model: It is represented by the probability of going from a state s to another state s' by doing the action a: $P(s'|s,a)$.

2.3.2 Policy and Value Functions

Recall that the main task of a reinforcement learning is finding a policy that optimises the value of rewards. This policy can be represented by a map π:

$$\begin{aligned} \pi : S &\longrightarrow A \\ s &\longmapsto \pi(s) \end{aligned}$$

where $\pi(s)$ is the action the agent takes at the state s.

Most MDP algorithms are based on estimating value functions. The state-value function according to a policy π is given by:

$$\begin{aligned} \nu^\pi : S &\longrightarrow \mathbb{R} \\ s &\longmapsto \nu^\pi(s) \end{aligned}$$

If we note here and in the remaining of this work $N = |S|$, the cardinal of the set of states S, the state-value function can be represented as a vector $V \in \mathbb{R}^N$ where each component corresponds respectively to the value $\nu^\pi(s)$ at each state $s \in S$.

Let us note $E_\pi\{R_t\}$, the expected value of the return R_t when the policy π is followed in the process. The value of the state-value function $\nu^\pi(s)$ at a particular state s is in fact the expected discounted return when starting at the state s and following the policy π.

$$\nu^\pi(s) = E_\pi\left\{R_t|S^t = s\right\} = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|S^t = s\right\}$$

The return R_t has the recursive property that:

$$\begin{aligned} R_t =& \sum_{k=0}^{\infty}\gamma^k r_{t+k+1} = r_{t+1} + \sum_{k=1}^{\infty}\gamma^k r_{t+k+1} \\ =& r_{t+1} + \sum_{k=0}^{\infty}\gamma^{k+1} r_{t+(k+1)+1} = r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+(k+1)+1} \\ =& r_{t+1} + \gamma R_{t+1} \end{aligned}$$

Thus,

$$\begin{aligned} \nu^\pi(s) =& E_\pi\left\{R_t|S^t = s\right\} = E_\pi\left\{r_{t+1} + \gamma R_{t+1}|S^t = s\right\} \\ =& E_\pi\left\{r_{t+1}|S^t = s\right\} + \gamma E_\pi\left\{R_{t+1}|S^t = s\right\} \end{aligned} \tag{2.1}$$

But $E_\pi \{r_{t+1}|S^t = s\}$ is the expected value of the reward r_{t+1} when the policy π is followed and such that at time t, the state is s. This is the same as the reward obtained by doing the action $\pi(s)$ at the state s : $R(s, \pi(s))$.

Therefore, equation 2.1 becomes:

$$\nu^\pi(s) = R(s, \pi(s)) + \gamma E_\pi \{R_{t+1}|S^t = s\} \tag{2.2}$$

But $E_\pi \{R_{t+1}|S^t = s\}$ is the expected value of R_{t+1} known that $S^t = s$. and by definition of the conditional expectation, we get:

$$E_\pi \{R_{t+1}|S^t = s\} = \sum_{s'} \left[P(s'|s, \pi(s)) E_\pi \{R_{t+1}|S^{t+1} = s'\}\right] \tag{2.3}$$

Putting the equation 2.3 into the equation 2.2 yields to:

$$\nu^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} \left[P(s'|s, \pi(s)) E_\pi \{R_{t+1}|S^{t+1} = s'\}\right] \tag{2.4}$$

And in fact,

$$E_\pi \{R_{t+1}|S^{t+1} = s'\} = \nu^\pi(s')$$

Finally, we define the state-value function:

$$\nu^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} \left[P(s'|s, \pi(s)) \nu^\pi(s')\right] \tag{2.5}$$

One can also define another form of value function called *state-action value function* according to a policy π: $Q^\pi(s, a)$ which depends not only on the state but on the action taken as well.

Definition 2.3.1. *[GKPV03] The decision process operator T_π for a policy π is defined by:*

$$T_\pi \nu(s) = R(s, \pi(s)) + \gamma \sum_{s'} \left[P(s'|s, \pi(s)) \nu(s')\right]$$

It yields from the definition of the state-value function in equation 2.5 that $\nu^\pi(s)$ is the fixed point of the decision process operator T_π. That is to say, $T_\pi \nu^\pi(s) = \nu^\pi(s)$

Definition 2.3.2. *[GKPV03] The Bellman operator T^* is defined as:*

$$T^* \nu(s) = \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) \nu(s')\right]$$

We denote ν^*, the fixed point of the Bellman operator. So, $T^* \nu^*(s) = \nu^*(s)$. And $\nu^*(s)$ is the maximum, according to actions, of $\nu(s)$ for a particular state s. When considering this maximum for every state $s \in S$, we can therefore define ν^*, which is called the optimal state-value function.

Then, finding the appropriate action a which maximises the reward at each state is equivalent to find an action a verifying $T^* \nu^*(s) = \nu^*(s)$. We denote this action $greedy(\nu)(s)$.

$$greedy(\nu)(s) = arg \max_a \left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) \nu(s')\right] \tag{2.6}$$

It is worth nothing that for each state s of the system, we have one equation like 2.6.

Therefore, finding the policy π to get the right action to do for every state of the system is now equivalent to solving N equations, each given by 2.6, with N unknowns.

Solving a MDP consists of looking for an efficient algorithm to solve this system.

2.4 Reinforcement Learning Methods

2.4.1 Solving a MDP by Value Iteration Algorithm

Recall that solving a MDP is equivalent to solving a system of N Bellman equations with N unknowns, where N is the number of possible states of the system. That is solving non linear equations. Then, iterative approach is suitable. Value iteration, also called *Backward induction* is one of the simplest method to solve a finite MDP. It was established by Bellman in 1957. Because our main objective in MDP is to find an optimal policy, value iteration works on it by finding the optimal value function first, and then the corresponding optimal policy.

The basic idea is to set up a sequence of state-value functions $\{\nu_k(s)\}_k$ such that this sequence converges to an optimal state-value function $\nu^*(s)$. In fact, a state-value function ν can be written as a vector in $\mathbb{R}^N$: each component corresponds to $\nu(s)$ where s is a particular state. Recall that the Bellman operator T^* is defined by:

$$T^*\nu(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a)\nu(s') \right]$$

The Bellman operator is in fact a max-norm contraction with contraction factor γ [SL08].

It implies that $\|T^*\nu - T^*\mu\|_\infty \leq \|\nu - \mu\|_\infty \ \forall \nu$ and μ state-value functions $\in \mathbb{R}^N$.

Theorem 2.4.1. *Banach fixed point theorem (contraction mapping principle):*

Let T be a mapping $T : X \longrightarrow X$ from a non-empty complete metric space X to itself. If T is a contraction, it admits one and only one fixed point.

Furthermore, this fixed point can be found as follows: starting with an arbitrary element $x_0 \in X$, an iteration is defined by : $x_1 = T(x_0)$ and so on $x_{k+1} = T(x_k)$. This sequence converges and its limit is the fixed point of T.

In our case, $\mathbb{R}^N$ is a non-empty complete metric space and the Bellman operator T^* is a contraction. So, from theorem 2.4.1, the sequence $\{\nu_k\}_k$ converges to the fixed point ν^* of Bellman equation. We have already proven in section 2.3.2 that the fixed point of the Bellman operator is in fact the optimal policy for our MDP. That is why the value iteration converges to the optimal policy. The whole algorithm is described in Algorithm 1.

2.4.2 Q-learning

Proposed by Watkings in 1989 [SB98], Q-learning is one of the most widely used reinforcement learning method due to its simplicity. It deals only with the iteration of the action-state value function $Q(s,a)$, independent of the policy, and does not need a complete model of the environment. Q-learning is used for episodic tasks. The whole algorithm is presented in Algorithm 2.

The learning rate α is needed in the iteration $(0 \leq \alpha < 1)$. It can take the same value for all pairs (s,a). When α is 0, there is no update of the state-action value function Q, so , the algorithm learns nothing. Setting it near to 1 accelerates the learning procedure.

Algorithm 1 Value Iteration Algorithm

Input: set of states S, set of actions A, vectors of rewards for every action $R(s,a)$, transition probability matrices for every action $P(s'|s,a)$, the discount rate γ, a very small number θ near to 0
Initialise ν_0 arbitrarily for every state $s \in S$. (Example: $\nu_0(s) = 0 \ \forall s \in S$)
while $|\nu_k(s) - \nu_{k-1}(s)| < \theta \ \forall s \in S$ **do**
  for all s in S **do**
    for all a in A **do**
      $Q_{k+1}(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a)\nu_k(s')$
    end for
    $\nu_{k+1}(s) = \max_a Q_{k+1}(s,a)$
    $\pi_k(s) = arg\max_a Q_{k+1}(s,a)$ {gives the action a that has given ν_{k+1} }
  end for
end while
Output: ν_k {takes the ν_k at the last iteration of the WHILE loop: which is the optimal state-value function}
Output: π_k {takes the π_k at the last iteration of the WHILE loop: which is the optimal policy}

Algorithm 2 Q-learning algorithm

Input: the set of states S, the vector of rewards $R(s,a)$, the set of actions A
Input: the discount rate γ, and the learning rate α
Initialise $Q(s,a)$ arbitrarily for every state s and every action a
for each episode **do**
  Initialise s
  repeat {For each step of the episode}
    Choose one possible action a for the state s, derived from Q (for example, choose the greedy action, that is to say the one which has the maximum reward)
    Take the action a, and the corresponding reward $R(s,a)$, and the next state s'
    $Q(s,a) \longleftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s' \longleftarrow s$
  until s is terminal
end for

2.4.3 SARSA

SARSA algorithm stands for *State-Action-Reward, State-Action*. This algorithm also learns a state-action value function $Q(s, a)$ instead of a state-value function $\nu(s)$. The main difference between it and the Q-learning algorithm is that SARSA does not use the maximum according to action to update the value function Q. Instead, it takes again another action a' according to the policy corresponding to the value function Q. This means that the update is made using the quintuple (s, a, r, s', a'). The name SARSA is derived from that fact. The algorithm is presented in Algorithm 3.

Algorithm 3 SARSA algorithm

Input: the set of states S, the vector of rewards $R(s, a)$, the set of actions A
Input: the discount rate γ, and the learning rate α
  Initialise $Q(s, a)$ arbitrarily for every state s and every action a
  for each episode **do**
    Initialise s
    Choose one possible action a for the state s, derived from Q (for example, choose the greedy action)
    repeat {For each step of the episode}
      Take the action a, and observe the obtained reward $R(s, a)$ and the next state s'
      Choose one possible action a' for the state s', derived from Q (for example, choose the greedy action)
      $Q(s, a) \longleftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a)\right]$
      $s \longleftarrow s'$
      $a \longleftarrow a'$
    until s is terminal
  end for

Q-learning and SARSA algorithms have been proven to converge to an optimal policy and an action-state value function as long as all pairs (s, a) are visited a large number of times during the iteration. In the limit, the policy converges to the greedy policy $greedy(\nu)$.

2.4.4 Model-based Reinforcement Learning

Q-learning and SARSA target to have an optimal policy without knowing the model, and regardless of the transition probability. Due to that, they require a great focus on experiments in order to reach a good performance. Model-based reinforcement learning also does not know the model in advance, but learns it. It consists of building the transition probability and the reward function by experiments. Methods used for that purpose are variable and can be subjective according to the kind of problem. Nevertheless, generally, all model-based reinforcement learning methods consist of modelling and then planning.

The model is used to see more closely the behaviour of the environment after the agent makes an action. Given a starting state, the agent makes an action. It gives a next state and a specific reward. And by trying every possible actions at every state, the transition probability and the reward function can be estimated.

After having a complete information about the environment, the planning procedure is elaborated. It takes the model as input and produces a policy. For this purpose, a simulated experience is set up

according to the complete model. Then, the value function can be calculated from the experience by backups operations. This leads to the search of an optimal policy. The method can be summarised as follows:

$$Model \longrightarrow Simulated\ experience \xrightarrow{backups} Value\ function \longrightarrow Policy$$

Algorithm 4 is an example of model-based reinforcement learning method. It uses the updates of the state-action value function $Q(s, a)$, like in the Q-learning method, to compute the policy.

Algorithm 4 Example of a model-based reinforcement learning algorithm

Input: the set of states S and the actions set A
Input: the discount rate γ and the learning rate α
loop
Select a state s, and an action a at random
Apply the action a at the state s to the model, and obtain the next state s' and a reward r
Compute the action-state value function by the iteration:
$Q(s, a) \longleftarrow Q(s, a) + \alpha\left[r + \gamma \max a' Q(s', a') - Q(s, a)\right]$
end loop

After presenting some algorithms to solve reinforcement learning problem, we are going to see in the next chapter their applications in the traffic control.

3. Related Work to Traffic Control

Many attempts have been made to manage urban traffic signal control. With a rapid development in computer technology, artificial intelligence methods such as fuzzy logic, neural networks, evolutionary algorithms and reinforcement learning have been applied to the problem successively. Since its first exploitation in urban traffic control in 1996, the approach using reinforcement learning is still in development [Liu07]. In this chapter, we will regard some work related to the urban traffic signal control, mainly those using reinforcement learning.

3.1 Reinforcement Learning Approach

The first approach to the problem of traffic control using reinforcement learning was made by Thorpe and Anderson [TA96] in 1996. The state of the system was characterised by the number and positions of vehicles in the north, south, east and west lanes approaching the intersection. The actions consist of allowing either the vehicles on the north-south axis to pass, or those on East-West axis to pass. And the goal state is when the number of waiting cars is 0. A neural network was used to predict the waiting time of cars around a junction. And the SARSA algorithm (see chapter 2, section 2.4.3) was applied to control that junction. The model was then duplicated at every junction of a traffic network with 4×4 intersections. The choice of the state variables implies that the number of state is large. However, simulations underlined that the model provides a near optimal performance [TA96].

In 2003, Abdulhai et al. have shown that the use of reinforcement learning, especially the use of Q-learning (see chapter 2, section 2.4.2) is a promising approach to solve the urban traffic control problem [Abd03]. The state includes the duration of each phase and the queue lengths on the roads around the crossing. The actions were either extending the current phase of the traffic light or changing to the next phase. This approach has shown a good performance when used to control isolated traffic lights [Abd03].

In 2004, M. Weiring also used multi-agent reinforcement learning to control a system of junctions [WVK04]. A model-based reinforcement learning is used in this approach, and the system is modeled in its microscopic representation, that is to say, they considered the behaviour of individual vehicles. In the model-based reinforcement learning, They counted the frequency of every possible transitions, and the sum of received rewards, corresponding to a specific action taken. Then, a maximum likelihood model is used for the estimation. The set of states was car-based: it included the road where the car is, its direction, its position on a queue, and its destination address. The goal was to minimise the total waiting time of all cars at each intersection, at every time step. An optimal control strategy for area traffic control can be obtained from this approach.

In 2005, M. Steingröver presented an approach to solve traffic congestion using also model-based reinforcement learning [SSP+05]. They considered the system of an isolated junction and represented it as a MDP. Their idea was to include in the set of states an extra information which is the amount of traffic at neighbouring junctions. This approach was expected to optimise the global functioning of the whole network but not only a local network composed of an isolated junction. The transition probability was estimated using a standard maximum likelihood modelling method. As expected, experiments showed the benefits of considering the extra parameter [SSP+05].

Denise de Oliveira et al. used reinforcement learning in the control of traffic lights too [dOBdS+06].

They emphasised and exploited the fact of a non-stationary environment, and used particularly context-detection reinforcement learning. They considered a system for which the states set depends on the context. But a set of states corresponding to a specific context is viewed to be a stationary environment. Their results showed that this method performed better than the Q-learning and the classical greedy (see [SB98]) methods because real traffic situation cannot be predetermined and should be considered as a non-stationary scenario.

3.2 Other Approaches

3.2.1 Fuzzy Logic

Fuzzy logic is the same as 'imprecise logic'. It is a form of multi-valued logic, derived from fuzzy set theory which can treat approximate logic rather than precise logic.

The earliest attempt to use fuzzy logic in traffic control was made by C.P. Pappis and E.H. Mamdi in 1977. It considers only one junction consisting of two phases: green or red. The rules were as the following: from some fixed seconds of green light state, the fuzzy logic controllers take decision to change the phase or not, according to the traffic volume on all the roads of the intersection.

In 1995, Tan et al. also used fuzzy logic to control a single junction [KKY96]. The fuzzy logic controller is built to know the period of time the traffic light should stay at a phase before changing to another one. The order of phases is predetermined but the controller can skip a phase. The number of arriving and waiting vehicles is represented as a fuzzy variables namely *many*, *medium*, and *none*. The decision is taken according to these variables.

Some extension to two and many intersections have used fuzzy logic too. That is the case of Lee et al. in 1995 [LB95]. Controllers receive information from the next and previous junctions to coordinate the frequency of green phases at a junction.

Chiu also used fuzzy logic to adjust parameters like the degree of saturation on each intersection, the cycle time, which were taken into account in the decision strategy [Chi92].

All these work, after being put into experiments showed good results. Fuzzy logic controllers seem to be more flexible than fixed controllers. Nevertheless, according to a survey of intelligence methods in urban traffic signal control [Liu07], fuzzy logic is difficult to apply in the case of multiple intersections control because a complex system can be difficultly described using qualitative knowledge.

3.2.2 Artificial Neural Network

The term 'neural network' is used in artificial intelligent as a simplified model inspired by neural processing in the brain.

In general, there are three classes of applications of artificial neural network in traffic signal control [Liu07]. It can be used for modelling, learning and controlling. It can also be used as a complementary of another method. For example, it can help a fuzzy algorithm control method in order to increase precision. Finally, Neural network is commonly used as a forecast model.

Liu Zhiyong et al. have applied a self-learning control method based on artificial neural network to manage the functioning of an isolated intersection [Liu97]. Two neural networks, which are always

alternatively in the states of learning or working, were set up during the self-learning process. Then, after this process, the neural network begins to be the controller.

Patel and Ranganathan also modelled the traffic light problem using artificial neural network approach [PR01]. This approach aimed to predict the traffic lights parameters for the next time period and to compute the adjusted duration of this period. A list of data collected from censors around the junction serves as input of the artificial neural network, and the outputs are the timing for the red and the green lights.

In certain cases, artificial neural network and fuzzy logic are combined to model a traffic light control. That is for example the case in J.J. Henry et al. work [HFG98]. Methods using artificial neural network in traffic control have been shown to be efficient. But they lack of flexibility to be generalised, and it is difficult to adapt them in real-life system.

3.2.3 Evolutionary Algorithms

Evolutionary algorithm is a subset of evolutionary computation which uses some mechanics inspired by biological evolution. The advantage of evolutionary algorithms is that they are flexible and can solve optimisation of a non-linear programming problem, which traditional mathematical methods cannot solve. Generally, the application of evolutionary algorithms in traffic control is in managing the time duration of a phase of the traffic lights. It is the case in the work of M. D. Foy et al. [FBG92]. They used genetic algorithm (a branch of evolutionary algorithms) to control a network consisting of four intersections.

In 1998, Taale et al. used evolutionary algorithm to evolve a traffic light controller for a single intersection [TBP+98]. They compared their results with the one using the common traffic light controller in the Netherlands. Considerable results were noticed. But unfortunately, they did not extend their work to multiple junctions.

4. SUMO

4.1 Overview and Definition

Since there was considerable progress in the area of urban mobility, important research was lead in this area. SUMO started as an open project in 2001. It stands for *Simulation of Urban MObility*. It is mainly developed by employees of the Institute of Transportation Systems at the German Aero-Space Centre in Germany.

The first idea was to provide to the traffic research community a common platform to test and compare different models of vehicles behaviour, traffic light optimisation. SUMO was constructed as a basic framework containing methods needed for a traffic simulation. It is an open source package designed to handle large road networks.

Two thoughts were considered in the release of the package as a open source. First, every traffic research organisation has his own concern to be simulated: for example, some people are interested in traffic light optimisation, others try to simulate an accident. So, one can simply implement SUMO and modify it according to needs. The second idea is to make a common test bed for models, to make them comparable.

From the beginning, SUMO was designed to be highly portable and to run as fast as possible. In order to reach those goals, the very first versions were developed to run from a command line only, and without graphical interface. The software was split into several parts that can run individually. This allows an easier extension of each application. In addition, SUMO uses the standard C^{++}-functions, and can be run under Linux and Windows.

SUMO uses a microscopic model developed by Stefan Krauβ [KWG97, Kra98] for traffic flow. The model consists of a space continuous simulation, that is, each vehicle has a certain position described by a floating point number. SUMO also uses a *Dynamic User Assignment* (DUA) [Gaw98] by Christian Gawron to approach the problem of congestion along the shortest path in a network. The time used in SUMO is counted discretely, according to a user-defined unit of time.

4.2 Building a Simulation with SUMO

To create a simulation, several steps have to be followed.

4.2.1 Building the Network

One first needs to create a network in which the traffic to simulate takes place. There are two possible ways to generate a network.

From an existing file

The network can be imported from an existing map, which is often the case when we want to simulate a real-world network. In that case, the module *NETCONVERT* will be used for the conversion of the

file into a file with extension $.net.xml$ which is readable by SUMO.

Randomly

The module *NETGEN* can also generate a basic, abstract road map.

By hand

Networks can also be built by hand according to the research being carried out. To do this, at least, two files are needed: one file for nodes, and another one for edges.

1. **Nodes**:

   In SUMO, junctions are represented by nodes. Every node is characterised by:

   - *id*: the identity of the node. It can be any character string.
   - *x*: the x-position of the node on the plane by unit of meter. This should be a floating variable.
   - *y*: the y-position of the node on the plane (per meter). It is also a floating variable.
   - *type*: the type of the node. This is an optional attribute. Possible types are:
     - *priority* : this is the default option. And follows the right priority: vehicles have to wait until vehicles right to them have passed the junction.
     - *traffic light*: when a traffic light is set up to control the junction. (We will see it in section [4.4]).

   The following example is what a node file looks like:

```
<nodes>
    <node id="nod1" x="-500.0" y="0.0" type="traffic_light" />
    <node id="nod2" x="+500.0" y="0.0"  type="traffic_light" />
    <node id="nod3" x="+500.0" y="=+500.0"  type="priority" />
    <node id="nod4" x="+500.0" y="=-500.0"  type="priority" />
</nodes>
```

   A node file is saved with an extension $.nod.xml$

2. **Edges**:

   Edges are the roads that link the junctions (nodes). An edge has the following features:

   - *id*: The identity of the edge. It can be any character string.
   - The origin and the destination of the edge which include:
     - *fromnode*: it is the identity of the node, as given in the node file, at which the edge starts.
     - *tonode*: the identity of the node at which the edge ends.
     - *xfrom*: the x-position of the starting node.
     - *yfrom*: the y-position of the starting node.

  - *xto*: the x-position of the ending node.
  - *yto*: the y-position of the ending node.
- *type*: the name of a type file, if there is one.(We will see it later in the section [3])
- *nolanes*: the number of lanes in the edge (it is an integer).
- *speed*: the maximum allowed speed on the edge (in $m.s^{-1}$). It is a floating variable.
- *priority*: the priority of the edge, compared to other edge.It depends for example on the number of lanes in the edge, the maximum speed allowed on the edge,.... It must take an integer value.
- *length*: the length of the edge (in meters). It takes a floating value.
- *shape*: each position is encoded in x and y coordinates. And the start and the end nodes are omitted from the shape definition.
- *allow/disallow*: which describes allowed vehicle types.

Here is an example of an edge file:

```
<edges>
 <edge id="edg1" fromnode="nod1" tonode="nod2" priority="2"
nolanes="2" speed="13.889"/>
 <edge id="edg2" fromnode="nod2" tonode="nod3" priority="4"
nolanes="4" speed="11.11"/>
 <edge id="edg3" fromnode="nod3" tonode="nod4" priority="4"
nolanes="4" speed="11.11"/>
 <edge id="edg4" fromnode="nod4" tonode="nod1" priority="2"
nolanes="2" speed="13.889"/>
</edges>
```

An edge file is saved with the extension $.edg.xml$.

The built network can be more specific as well, by the creation of other files such as the type file or the connection file.

3. Types: The type file is used to avoid repetition of some of the attributes in the edge file. To avoid an explicit definition of each parameter for every edge, one combines it into a type file. Its attributes are:

   - *id*: the name of the road type. It can be any character string.
   - *priority*: the priority of the corresponding edge.
   - *nolanes*: the number of lanes of the corresponding edge.
   - *speed*: the maximum speed allowed on the referencing edge (in $m.s^{-1}$).

   The following is an example of a type file:

```
 <types>
   <type id="a" priority="4" nolanes="4" speed="11.11"/>
   <type id="b" priority="2" nolanes="2" speed="13.889"/>
 </types>
```

The type file is saved with the extension $.typ.xml$. If we consider this file, the edge file in the previous example becomes:

```
<edges>
   <edge id="edg1" fromnode="nod1" tonode="nod2" type="b"/>
   <edge id="edg2" fromnode="nod2" tonode="nod3" type="a"/>
   <edge id="edg3" fromnode="nod3" tonode="nod4" type="a"/>
   <edge id="edg4" fromnode="nod4" tonode="nod1" type="b"/>
</edges>
```

4. **Connections**: The connection file specifies which edges outgoing from a junction may be reached by certain edges incoming into this junction. More precisely, it determines which lane is allowed to link another lane. It has the attributes:

   - *from*: the identity of the edge the vehicle leaves.
   - *to*: the identity of the edge the vehicle may reach.
   - *lane*: the numbering of connected lanes.

   An example of a connection file looks like the following:

```
<connections>
   <connection from="edg1" to="edg2" lane="0:0"/>
   <connection from="edg1" to="edg2" lane="0:1"/>
   <connection from="edg2" to="edg1" lane="2:1"/>
   <connection from="edg2" to="edg1" lane="3:1"/>
</connections>
```

   $< connection\ from = "edg2"\ to = "edg1"\ lane = "3:1"/ >$ means that only *lane3* of the *edg2* can be connected to *lane1* of the *edg1*.

After building the files needed for the network, the module *NETCONVERT* is used to create the network file with the extension $.net.xml$.

Figure 4.1 shows the the construction of a network by hand: after constructing the node file, the edge file, the type file and the connection file, they serve as an input in the a configuration file that uses the module *NETCONVERTER* and outputs the netwok file.

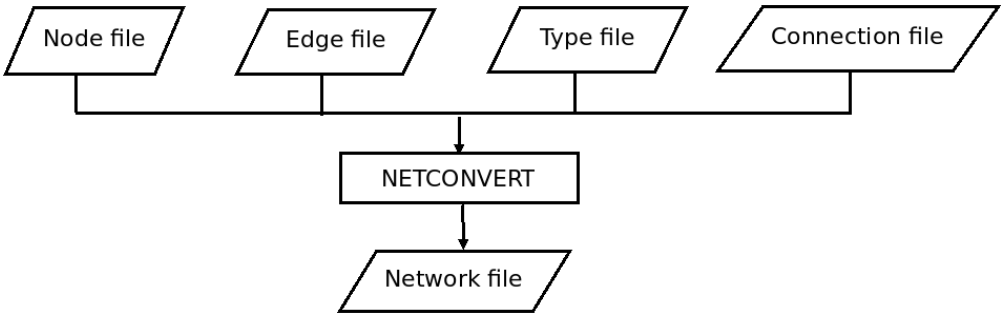

Figure 4.1: Construction of a network in SUMO

4.2.2 Route Generation

Routes give the description about the vehicles circulating in the network and their respective trajectories. There are several ways to generate routes. The most familiar ones include trip definitions, flow definitions, turning ratios with flow definitions, random generation, importation of existing routes, or manual creation of routes.

1. Trip definitions:

   A Trip definition describes the trajectory of a single vehicle. The origin and the destination edges are given, via their identity. So, a trip has the following parameters:

   - *id*: it defines the identity of the route. It can be any character string.
   - *depart*: it indicates the departure time of vehicles corresponding to the route.
   - *from*: indicates the name of the edge to start the trajectory.
   - *to*: indicates the name of the edge to end the trajectory.

   Route has also some optional attributes:

   - *type*: name of the type of the vehicle.
   - *period*: the time after which another vehicle with the same route is emitted in the network.
   - *repno*: number of vehicles to emit that share the same route.

   An example of a trip file is the following:

   ```
    <tripdef id="<rout1>" depart="<0>" from="<edg1>" to="<edg2>"
   [type="<type2>"] [period="<3>" repno="<20>"] />
   ```

   Then, the trip definition is supplied to the module *DUAROUTER* to get a file with the extension $.rou.xml$.

2. Flow definitions:

   Flow definitions share most of the parameters with trip definitions. The only difference is that the vehicle does not take a certain departure time because flow definitions describe not only one vehicle. The parameters $begin$, end which define the beginning and the end time, respectively, for the described interval, and no which indicates the number of vehicles that shall be emitted during the time interval, are added to the attributes. Apart from these extra parameters, id, $from$, to, $type$ remain for flow definitions.

   Example of a flow definitions:

   ```
    <interval begin="0" end="1000">
      <flow id="0" from="edg1" to="edg2" no="100"/>
   </interval>
   ```

3. Junction Turning Ratio-Router:

   Junction Turning Ratio-Router (*JTRROUTER*) is a routing application which uses flows and turning percentages at junctions as inputs. A further file has to be built to describe the turn definitions. For each interval and each edge, one has to give the percentages to use a certain possible following edge. Here is an example of a file containing turn definitions:

```
<turn-defs>
  <interval begin="0" end="1000">
     <fromedge id="edg1">
        <toedge id="edg2" probability="0.2"/>
        <toedge id="edg3" probability="0.8"/>
  </interval>
</turn-defs>
```

This example describes that during the time interval 0 to 3600, the probability to come from the *edg1* and continuing to the *edg2* is 0.2, and for continuing to the *edg3* is 0.8.

The definition of the flow is the same as before (in flow definitions), but the only difference is that we do not know the route followed by the vehicle, as it is computed randomly. Therefore, the attributes to is omitted.

Then, the turn and the flow definitions are supplied to *JTRROUTER* in order to generate the route file.

4. Random route:

   Random route is the easiest way to put circulating vehicles in the network; but it is known as the least accurate one. The application *DUAROUTER* needs to be called for this, or the *JTRROUTER*. The syntax to generate a random route is as follows:

   ```
   duarouter --net=<network_file> -R <float> --output-file=routefile.rou.xml
     -b <starting_time> -e <ending_time>
   ```

   where $-R < float >$ represents the average number of vehicles emitted into the network every time step. It can be a floating point. For example, $-R < 0.5 >$ means that a vehicle is emitted each twice time step.

5. By hand:

   This is feasible only if the number of routes is not too high. A route is determined by the properties of each vehicle, and the route taken by each vehicle. To define the vehicle properties, first, we define a type of it, and then the descriptions of the vehicle itself. A particular type of vehicle is described by:

   - *id*: indicates the name of the vehicle type.
   - *accel*: the acceleration ability of vehicles of the corresponding type (in $m.s^{-2}$).
   - *decel*: the deceleration ability of vehicles of the corresponding type (in $m.s^{-2}$).
   - *sigma*: it is an evaluation of the imperfection of the driver, and it is evaluated from 0 to 1.
   - *length*: the vehicle length (in meters).
   - *maxspeed*: the maximum velocity of the vehicle.

   Then, we build the vehicle of the type called above:

   - *id*: a name given to the vehicle.
   - *type*: the type mentionned above.
   - *depart*: the time the vehicle starts to circulate in the network.

- *color*: colour of the vehicle. It is given in the RGB scheme. It is coded in three numbers between 0 and 1, indicating the distribution of the colours red, green and blue, respectively.

Then, the route is constructed with the following attributes:

- *id*: identity of the route.
- *edges*: lists the edges the route is composed of.

The following examples is what a route file constructed by hand should look like.

```
 <routes>
   <vtype id="type1" accel="0.8" decel="4.5" sigma="0.7" length="3"
maxspeed="70"/>
   <route id="route0" edges="beg middle end rend"/>
   <vehicle id="0" type="type1" route="route0" depart="0" color="1,0,0"/>
   <vehicle id="1" type="type1" route="route0" depart="0" color="0,1,0"/>
 </routes>
```

SUMO also has a tool to generate routes and vehicle types, based on a specific distribution. It can be used instead of defining those attributes explicitly. The following example shows how it works

```
 <routes>
   <vtypeDistribution id="typedist1">
      <vtype id="type1" accel="0.8" decel="4.5" sigma="0.7" length="3"
maxspeed="70" probability="0.9"/>
      <vtype id="type2" accel="1.8" decel="4.5" sigma="0.5" length="2"
maxspeed="50" probability="0.1"/>
   </vtypeDistribution>
   <routeDistribution id="routedist1">
     <route id="route0" color="1,1,0" edges="beg middle end rend"
probability="0.5"/>
     <route id="route1" color="1,2,0" edges="beg middle end"
probability="0.5"/>
   </routeDistribution>
</routes>
```

A route file has the extension $.rou.xml$

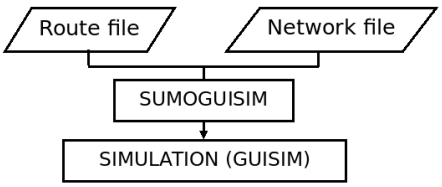

Figure 4.2: Construction of a Simulation

Figure (4.2) summarises the construction of a simulation with SUMO: after following the procedure to construct the network file (see Figure 4.1), it serves, with the route file, as inputs of a configuration file which outputs the simulation itself by means of the module SUMOGUISIM.

4.3 Detectors

Following the progress in traffic systems, SUMO is always updated according to actual changes in this field. Nowadays, most highways are well equipped with induction loops, and SUMO can provide them. There are three types of detectors used in SUMO: E_1 detector or induction loop. E_2 detector or lane-based detector, and E_3 detector or multi-origin / multi-destination detector.

E_1 detector or induction loop:

The induction loop is a slice plane that can be put through a road-lane and collect information about vehicles passing it. To define this detector, one needs to specify its identity, the lane on which it is posed, its position on a lane, and the period over which collected values should be put together. This includes the attributes: id, $lane$, pos, $freq$. Another attribute is the name of the file in which outputs will be printed: $file$.

E_1 detector gives outputs about the collected vehicles. Precisely, it gives:

- The first and the last time steps the values are collected: $begin$ and end.
- The identity of the detector: id.
- The number of vehicles that have completely passed the detector during an interval of time ($nVehContrib$), and the same number, but extrapolated to one hour ($flow$).
- The percentage of the time the vehicle was at the detector: $occupancy$.
- The mean velocity of all collected vehicles: $speed$.
- The mean length of all completely collected vehicles: $length$.

E_2 detector or lane-based detector:

E_2 detector is a sensor lying along a lane or a part of a lane, and describes the vehicles on that part. It is defined by the identity of the detector, the identity of the lane the detector is lying, the position of the detector on the lane, the length of the detector, the period at which collected values shall be summed up, and the name of its output-file.

This detector has outputs concerning information about what is going on on the part of lane covered by it. We can list:

- The first and the last time steps where values are collected.
- The number of states of a vehicle that was on the detector area.

- The mean velocity of all vehicles on the detector area.
- The percentage of the area of the detector occupied by vehicles during a time step.
- The sum of the length of jams detected on the area (in meters, and in number of vehicles).
- The mean halting duration of vehicles that entered the area and are still inside, counted for each time step, and then counted for all the time duration of the simulation.

E_3 detector multi-origin / multi-destination detector:

An E_3 detector measures vehicles circulating between a set of entry points and a set of exit points. As in the case of E_1 and E_2 detectors, E_3 detector is defined by an identity, the frequency of time to collect results, the identity of all lanes where there are entry points and exit points, and the positions of the entry and exit points of the detector on these lanes. An E_3 detector gives similar outputs as an E_2 detector but the only difference is that all values concern all the lanes where the detectors are lying.

4.4 Intelligent Traffic Light

A junction can be controlled by a Traffic Control Interface (TraCI) in SUMO. TraCI uses a TCP-based (Transmission Control Protocol) client/server architecture to provide access to SUMO.

SUMO acts as a server, and the client is a Python script. After building the network in SUMO, the Python script receives the information collected by the detectors lying on some lanes in the network, and sends instructions back to control the system. The simulation can also be executed in the python script. The python script can, for example, follow the following procedure.

First, we connect with a SUMO server by a specific port. Then, a route generation procedure is constructed. For this purpose, one can use a loop which terminates after a given terminal step, to generate the cars and update the route file. This is a good way to generate routes when we want the generation of cars to follow a certain pattern. This kind of loop cannot be feasible if routes are directly created by hand or one by one.

This procedure complete the construction of the simulation, and the script to launch SUMO can now be included.

Then, a procedure to communicate with the induction loops is created. This function takes as argument the list of the detectors, known by their identity, which the user is interested in, and it returns a list of information of the vehicles that have completely passed each detector.

The possible states of the traffic light are then defined. They are assigned to a variable which represent the state. States could be for instance *North-South green*, *East-West red*,...

Then, a loop is constructed until there are no more generated vehicles, which means until the simulation is terminated. Within the loop, the control of putting the traffic lights at a specific state is elaborated. It can be made according to the decision strategy, and taking into account the list of information provided by the induction loops.

After the time of the simulation expires, the communication with the SUMO simulation is closed.

5. Formalisation of a Traffic System

5.1 Problem Statement

Reinforcement learning has a wide range of applications. In this work, we will attempt a reinforcement learning model to optimise the performance of a traffic road network.

5.2 Formalisation of the Problem as a Markov Decision Processes

5.2.1 Junction model

As mentioned before, our aim is to optimise the performance of a traffic road network. The agent is the entity which controls the states of the system of traffic lights. Our model is based on the following assumptions:

- A junction consists of the intersection of two roads; each road is composed of two lanes of opposite directions. A vehicle enters the intersection and goes straight onto the next road-lane. Therefore, our system is composed of the two roads, the vehicles circulating on the roads, and the traffic lights.
- Vehicles communicate with the agent by means of sensors placed on a road-lane. Sensors are used to collect informations of the vehicles passing it. It collects information such as the number of vehicles that have passed during a certain time interval. Two sensors are placed on each road-lane: one is placed at a far distance from the crossing, precisely at the beginning of the lane, in order to keep an account of the number of vehicles that enter on the lane. Another one is placed just after the junction in order to keep an account of the number of vehicles that have passed the intersection. Figure 5.1 shows the description of the junction that we consider.

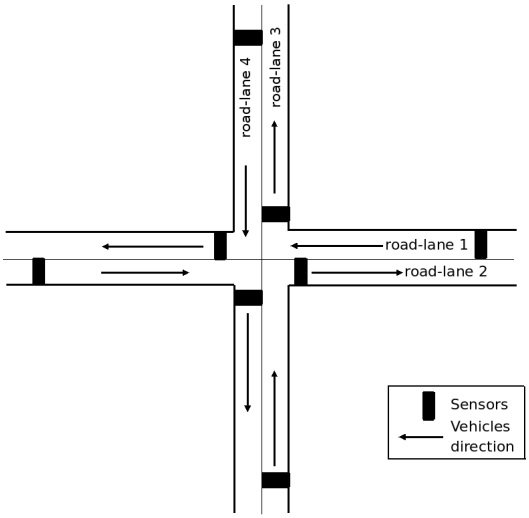

Figure 5.1: Representation of an isolated junction

- The length of the two roads we are interested in is precisely limited.
- And, we consider a discrete time step. An interval of time corresponds to a specific state of the system.

5.2.2 Junction Control through MDPs

Now, let us characterise the Markov Decision Processes:

1. Set of states:

   The state is represented by the state variable $S = \{Cr, C\}$:

   - Cr is the variable corresponding to the state of the crossing. More precisely, it is the state of the traffic light with all the components of the junction. We have two possible states of the crossing:
     - The North-South (N-S) is open and the East-West (E-W) road blocked;
     - The opposite of that, which is : the E-W road open and the North-South N-S road closed;
   - C is the state of cars.

     The state of cars is given by what is received from the sensors. Here, we need only the information concerning the number of cars that pass the censor during a certain time. Let us consider the number of 'waiting cars' on the two roads N-S and E-W. This number is given by the difference between the number of cars collected by the sensors placed before the junction and the number of cars given by the other two sensors placed after the junction. So, we have two values of 'waiting cars': one for each road composing the intersection. We can represent these values by the two variables: C_1 and C_2. C_1 is the number of 'waiting cars' on the N-S road, and C_2 is the number of 'waiting cars' on the E-W road. C_1 and C_2 take values in $\mathbb{N}$. We have then, $C = \{C_1, C_2\}$.

   To summarise, an example of a state of the system looks like :

   $$S = \{C = \text{'N-S road open'}, (C_1 = m, C_2 = n)\} \text{ where } m, n \in \mathbb{N}^+$$

2. Set of actions A:

   The agent controls the state of the crossing. So, the possible actions are:

   - *Stay the same* : This action consists of keeping the state of the traffic lights unchanged.
   - *Change* : To change the state of the crossing.

   The frequency of doing an action depends on the state of the system. That is to say, since the values of m and n are given, an action has to be taken.

3. Set of rewards:

   The reward is a function which depends on the state of the system and on the action taken, and takes values in $\mathbb{R}$.

   $$R : (A, S) \longrightarrow \mathbb{R}$$

Assume that at time t, the system is at the state:

$$S^t = \{C_r = \text{N-S open}, (C_1 = m, C_2 = n)\}$$

and if $n \geq 2m$, that means that the number of waiting cars on the closed road is more than twice the number of waiting cars on the open road. We should change the state of the traffic lights in order to decrease the jam on the blocked road. That is to say, the action *'Change'* has a higher reward than the action *'Stay the same'*. And if $n < 2m$, keeping the traffic light at the same state is still a good action.

If the state of the traffic light at time t is '*E-W open*' and $m \geq 2n$, changing the state of the crossing is better than keeping it.

Thus, let's define the reward function as the following:

$$R : (A, S) \longrightarrow \mathbb{R}$$

$$(\text{'Change'}, \{\text{N-S open}, (C_1 = m, C_2 = n)\}) \longmapsto \begin{cases} 1 & \text{if } n \geq 2m \\ -1 & \text{otherwise} \end{cases}$$

$$(\text{'Change'}, \{\text{E-W open}, (C_1 = m, C_2 = n)\}) \longmapsto \begin{cases} 1 & \text{if } m \geq 2n \\ -1 & \text{otherwise} \end{cases}$$

$$(\text{'Stay the same'}, \{\text{N-S open}, (C_1 = m, C_2 = n)\}) \longmapsto \begin{cases} 0 & \text{if } n \geq 2m \\ 1 & \text{otherwise} \end{cases}$$

$$(\text{'Stay the same'}, \{\text{E-W open}, (C_1 = m, C_2 = n)\}) \longmapsto \begin{cases} 0 & \text{if } m \geq 2n \\ 1 & \text{otherwise} \end{cases}$$

A discount factor $\gamma = 0.8$ will be used to give more importance to present and future rewards compared to past rewards.

4. Transition probability model:

   The transition probability model P is the probability of being in the state $S^{t+1} = s'$ by selecting a certain action a^t when the system was at the state $S^t = s$: $P(S^{t+1} = s' | a^t, S^t = s)$. Let us see more closely how to define the transition probability by first considering each possible action, and by considering only the state of the crossing.

Table 5.1: $a^t =$'*Change*' and $S^t = s = (C_r, C_1 = m, C_2 = n)$

State of the crossing at t	**State of the crossing at** $(t+1)$	$P(S^{t+1} = s' \| a^t, S^t = s)$
N-S open	N-S open	0
	E-W open	$f(m, n)$
E-W open	N-S open	$f(m, n)$
	E-W open	0

Table 5.2: $a^t =$'*Stay the same*' and $S^t = s = (C_r, C_1 = m, C_2 = n)$

State of the crossing at t	**State of the crossing at** $(t+1)$	$P(S^{t+1} = s' \mid a^t, S^t = s)$
N-S open	N-S open	$u(m,n)$
	E-W open	0
E-W open	N-S open	0
	E-W open	$v(m,n)$

Now, let's see more closely how the functions f, u, v are defined.

- $f(m,n)$
  The system is at the state $S^t = s = (C_r, (C_1 = m, C_2 = n))$ and the action is $a^t =$'*Change*'. The values of m and n change every time an update from the sensors are received. Let's assume that the information is collected from the sensors every second. This gives only a small number of possible states if we take into account the previous state of the system. Even if the action is '*Change*', the next values taken by C_1 and C_2 can vary at most with a difference of one waiting car from the previous state. And the state at which the values of C_1 and C_2 remain the same is more probable than to have a variation. Regarding to these assumptions, the distribution probability of C_1 is given by:

$$P(C_1^{t+1} = k | (C_1^t = m)) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = m \\ (1 - \frac{1}{\sqrt{2}})\frac{1}{m_1} & \text{if } k = m+1 \\ (1 - \frac{1}{\sqrt{2}})\frac{1}{m_1} & \text{if } k = m-1 \\ 0 & \text{otherwise} \end{cases}.$$

  where $m_1 = 1$ or 2, depending on the possible values of C_1. For example if $C_1^t = 0$, it is impossible to have $C_1^{t+1} = 0 - 1$, then $m_1 = 1$.
  By the same procedure, we can estimate the distribution of the probability of C_2 by:

$$P(C_2^{t+1} = k | (C_2^t = n)) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = n \\ (1 - \frac{1}{\sqrt{2}})\frac{1}{n_1} & \text{if } k = n+1 \\ (1 - \frac{1}{\sqrt{2}})\frac{1}{n_1} & \text{if } k = n-1 \\ 0 & \text{otherwise} \end{cases}$$

  where $n_1 = 1$ or 2, depending on the possible values of C_2.
  Since the three events composing the state of the system are independent from each other, then,

$$\begin{aligned} P(S^{t+1} = \{C_r, (C_1 = m', C_2 = n')\} \, | a^{(t)}, S^t = s) \\ = P(C_r^{t+1} | a^t, S^t = s) \cdot P(C_1 = m' | a^t, S^t = s) \cdot P(C_2 = n' | a^t, S^t = s) \end{aligned}$$

  Therefore,

$$\begin{aligned} f(m,n) = & P(S^{t+1} = \{C_r^{t+1}, (C_1^{t+1} = m', C_2^{t+1} = n')\} \\ & | a^{(t)} =' \text{ Change}', S^t = \{C_r^t, (C_1^t = m, C_2^t = n)\}) \\ = & \begin{cases} P(C_1^{t+1} = m+i).P(C_2^{t+1} = n+i) & \text{if } (m' = m+i) \text{ and } (n' = n+i) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where $i = 0, -1, 1, 2, -2$

- $u(m, n)$
  In the case when the action is '*Stay the same*', the probability transition depends not only on the number of waiting cars but also on the state of the crossing.
  Because $C_r^t =$ 'N-S open', by keeping the same state of the traffic light, it is likely probable that the number of waiting cars on the N-S road (m) will decrease and that on the E-W road (n) will increase. Nevertheless, because values of C_1 and C_2 are collected every second, the difference cannot be more than two vehicles. Let's make an estimation of the fact and set up that:

$$P((C_1^{t+1} = m \text{ or } m-1 \text{ or } m-2)|(C_1^t = m)) = \frac{3}{4}$$
$$\text{and } P((C_1^{t+1} = m \text{ or } m+1 \text{ or } m+2)|(C_1^t = m)) = \frac{1}{4}$$

  Given our choice of time step, it is less probable to have a difference of two cars on the lane than having only one or when the number of cars is the same. Thus, we will assume that:

$$P((C_1^{t+1} = m-2|(C_1^t = m))) = \frac{1}{4}\left(\frac{3}{4}\right) = \frac{3}{16}$$
$$P((C_1^{t+1} = m-1|(C_1^t = m))) = \frac{3}{8}\left(\frac{3}{4}\right) = \frac{9}{32}$$
$$P((C_1^{t+1} = m|(C_1^t = m))) = \frac{3}{8}\left(\frac{3}{4}\right) = \frac{9}{32}$$

  and then,

$$P((C_1^{t+1} = m+2|(C_1^t = m))) = \frac{1}{3}\left(\frac{1}{4}\right) = \frac{1}{12}$$
$$P((C_1^{t+1} = m+1|(C_1^t = m))) = \frac{2}{3}\left(\frac{1}{4}\right) = \frac{1}{6}$$

  To summarise, the probability distribution of C_1 is given by:

$$P(C_1^{t+1} = k|(C_1^t = m)) = \begin{cases} \frac{3}{16} & \text{if } k = m-2 \\ \frac{9}{32} & \text{if } k = m-1 \text{ or } k = m \\ \frac{1}{12} & \text{if } k = m+2 \\ \frac{1}{6} & \text{if } k = m+1 \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

  For the case of $P(C_2^{t+1}|C_2^t)$, it is highly unlikely that the number of cars on the closed road (E-W road) is decreasing. After estimation, we can approximate the distribution probability of C_2 as follows:

$$P((C_2^{t+1} = n \text{ or } n+1 \text{ or } n+2)|(C_2^t = n)) = \frac{19}{20}$$
$$\text{and } P((C_2^{t+1} = n-1 \text{ or } n-2)|(C_1^t = m)) = \frac{1}{20}$$

Then,

$$P((C_2^{t+1} = n-2|(C_2^t = n))) = \frac{1}{3}\left(\frac{1}{20}\right) = \frac{1}{60}$$
$$P((C_2^{t+1} = n-1|(C_2^t = n))) = \frac{2}{3}\left(\frac{1}{20}\right) = \frac{1}{30}$$
$$P((C_2^{t+1} = n|(C_2^t = n))) = \frac{3}{8}\left(\frac{19}{20}\right) = \frac{57}{160}$$
$$P((C_2^{t+1} = n+1|(C_2^t = n))) = \frac{3}{8}\left(\frac{19}{20}\right) = \frac{57}{160}$$
$$P((C_2^{t+1} = n+2|(C_2^t = n))) = \frac{1}{4}\left(\frac{19}{20}\right) = \frac{19}{80}$$

We can summarise the transition probability of the variable C_2 as follows:

$$P(C_2^{t+1} = k|(C_2^t = n)) = \begin{cases} \frac{1}{60} & \text{if } k = n-2 \\ \frac{1}{30} & \text{if } k = n-1 \\ \frac{19}{80} & \text{if } k = n+2 \\ \frac{57}{160} & \text{if } k = n+1 \text{ or } k = n \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

At some states, some situations cannot occur when taking an action. For example, in the case of $C_1^t = 1$, the state $C_1^{t+1} = 1-2$ does not exist. Similarly to the case of taking the action *Change*, the probability will be set to $\frac{1}{N}$ for possible states, where N is the number of possible states, with respect to the action taken.

And finally, we have

$$\begin{aligned} u(m,n) =& P(S^{t+1} = \{C_r^{t+1} = \text{'N-S open'}, (C_1^{t+1} = k, C_2^{t+1} = l)\} \\ & |a^{(t)} = \text{' Stay the same'}, S^t = \{C_r^t = \text{'E-W open'}, (C_1^t = m, C_2^t = n)\}) \\ =& P(C_1^{t+1} = k|(C_1^t = m)) \cdot P(C_2^{t+1} = l|(C_2^t = n)) \end{aligned} \tag{5.3}$$

where $P(C_1^{t+1} = k|(C_1^t = m))$ and $P(C_2^{t+1} = l|(C_2^t = n))$ are given in equations 5.1 and 5.2.

- $v(m,n)$

  In the case $C_r =$'E-W open', the probability of being at a certain state is computed exactly in the same way as for $Cr_=$'N-S open', but instead of having a high probability that C_1 is increasing and C_2 decreasing, we have a high probability that C_2 is increasing and C_1 decreasing. Then, we get:

$$P(C_1^{t+1} = k|(C_1^t = m)) = \begin{cases} \frac{1}{60} & \text{if } k = m-2 \\ \frac{1}{30} & \text{if } k = m-1 \\ \frac{19}{80} & \text{if } k = m+2 \\ \frac{57}{160} & \text{if } k = m+1 \text{ or } k = m \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

And

$$P(C_2^{t+1} = l|(C_1^t = n)) = \begin{cases} \frac{3}{16} & \text{if } l = n-2 \\ \frac{9}{32} & \text{if } l = n-1 \text{ or } l = n \\ \frac{1}{12} & \text{if } l = n+2 \\ \frac{1}{6} & \text{if } l = n+1 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

Finally,

$$v(m,n) = P(C_1^{t+1} = k|(C_1^t = m)) \cdot P(C_2^{t+1} = l|(C_1^t = n) \tag{5.6}$$

6. Solution and Experimentation

Once we have formalised the problem of controlling a junction as a Markov decision process, the decision strategy can be elaborated. In this chapter, we will present a solution to the decision problem of the traffic system. We will describe in detail the way we have conducted the work, and explain the results of our proposed strategy.

6.1 The Decision Strategy

The fact that the environment has the Markov property provides us a complete information about the system. So, applying the value iteration algorithm seems to be the simplest way to solve the decision making.

For this purpose, we have written a Python program. First, all required items have been constructed.

- The set of states has been represented as a list in which each component is a particular state. The length of the list is then the number of possible states N of the system. If N_1 and N_2 are the maximum possible values taken by the variables C_1 and C_2, respectively, (recall that C_1 and C_2 are the number of waiting cars on the N-S road and the E-W road respectively), the number of possible states of the system is $N = 2 \times (N_1 + 1) \times (N_2 + 1)$. After estimation and making sure that all possible states of the system are included in the set of states, we have set up $N_1 = N_2 = 20$. It gives a list of length $N = 882$ for the states set.
- The set of action has been represented by a list of length 2 containing the two action '*Change*' and '*Stay the same*'.
- The set of rewards has been represented by two arrays of length N, one for each action.
- The transition model is represented by two matrices; one for each action. The previous states of the system are on the rows and the next states are on the columns.
- The discount rate γ is set to 0.8 in order to give more importance to immediate rewards.

Then a function containing the value iteration has been constructed. It returns the optimal policy π. Because the policy is a function from the set of states to the set of actions, it has been represented by a dictionary in which keys are the states and the values are the corresponding action to be taken.

The whole program can be found in the appendix A of this work.

6.2 Simulation

After acquiring the correspondence state-action from the Python program, it has been used in the simulation using SUMO to control a traffic light at an isolated junction.

For this purpose, we first created the network. It is composed of only one junction, which consists of two intersecting roads of length $1km$. Four induction loops have been placed on the four entering

road-lanes at a distance of $150m$ from the crossing. Another four induction loops have been placed on the other four exiting road-lanes, just after the junction.

Once the network is created, the generation of routes has been done in a Python script. In order to focus only on the decision strategy and avoid errors coming from another external parameters, all the cars have been generated with the same type (the same maximum velocity of 60km/h, the same acceleration, the same deceleration, the same length of vehicles). The generation of cars has been made randomly following a Poisson distribution approximated by a binomial distribution with parameter $p = \frac{1}{10}$.

Then, in another Python script, the control of the traffic light has been elaborated. After connecting with the SUMO server via a port and launching SUMO, a function has been created to communicate with the height detectors. It gives, as outputs, a list of length 8, of the number of cars that have passed the detectors, each component corresponding to a detector. Then, the number of waiting cars is calculated every time data is collected from the detectors. Recall that the number of waiting cars on a specific road is given by the difference between the total number of cars that have passed the two detectors lying on the entering road-lanes, and the total number of vehicles that have passed the two other detectors lying on the road-lanes exiting the junction.

At the beginning of the simulation, the state of the traffic light is set up to '*N-S open*', before a decision is made. Then, a function of the decision, which has arguments the state of the traffic light, the number of waiting cars on each road, and the dictionary of state-action correspondence, have been built. This function return the new state of the traffic light according to the decision taken. The state of the traffic light is therefore modified according to the new state given by the decision. All the process is iterated within a loop which stops when the simulation is terminated. After the simulation is finished, the communication with SUMO is closed. Figure 6.1 shows a SUMO simulation window.

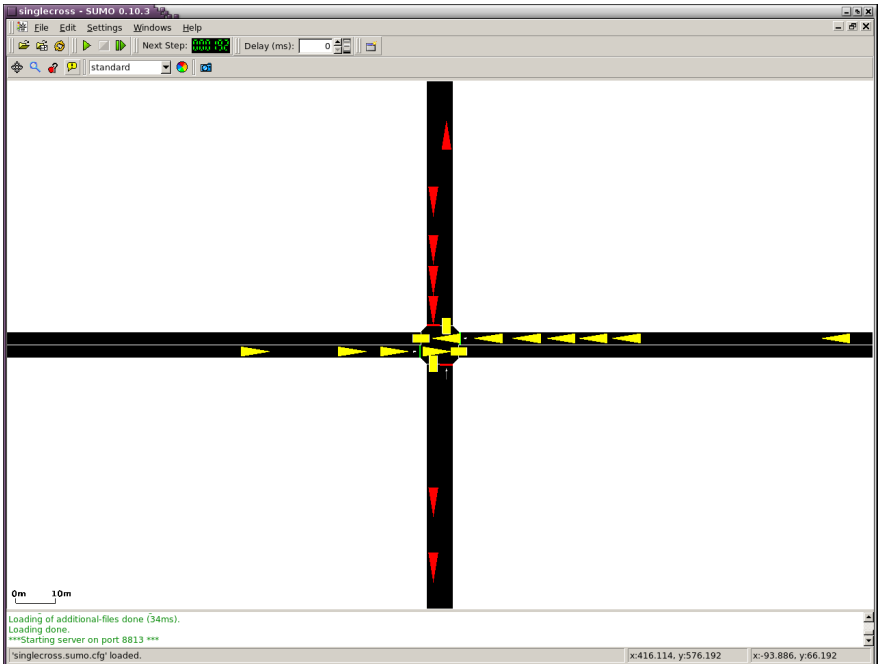

Figure 6.1: Simulation of a crossing in SUMO

6.3 Results and Interpretations

To evaluate the performance of our algorithm, we compared it with two different types of decision, namely a fixed-cycle decision and an adaptive rule-based mechanism. The fixed-cycle decision consists of changing the state of the traffic light every 30 steps in the simulation; whereas, the adaptive rule-based mechanism lies on the following rules:

The N-S road is open first and kept open unless:

- There are no more vehicles waiting in the N-S road and there some in E-W road, or
- The number of waiting cars on the E-W road is greater than, or equal to twice the number of waiting cars on the N-S road.

We apply symmetric rules when the E-W road is open.

It is worth to note that during the whole process for every simulation, the configuration of the experiment stayed the same for the three algorithms. Only the function which deals with the decision of changing the state of the traffic light is changed, according to the decision strategy from which we want to get results.

In order to get data, we have performed 50 simulations of 500 steps for each type of decision. During each simulation, we have collected two kinds of data:

- The number of waiting cars every five time-steps, that is for the $0, 5, 10, 15, \cdots$ time-steps, for each simulation and for each type of decision. This can give an approximation of the length of the queue at the intersection at a particular time.
- The cumulative number of waiting cars every five time steps. Here, the cumulative number of waiting cars is calculated by adding the number of waiting cars at every time step of the simulation. This can give a global view of the performance of the algorithm.

Then, the mean of the 50 values from the 50 simulations has been calculated at every five steps, for each decision type, for the number of waiting cars as well as for the cumulative number of waiting cars.

6.3.1 Results

Tables 6.1 and table 6.2 shows these mean values for some steps.

	100^{th} **step**	200^{th} **step**	300^{th} **step**	400^{th} **steps**	500^{th} **steps**
MDP	46.08	53.20	55.04	55.90	52.38
adaptive rule based mechanism	47.66	51.58	49.22	48.26	52.46
fixed-cycle decision	53.74	55.26	66.08	61.08	57.56

Table 6.1: Means of waiting cars at some steps of the simulation

	100^{th} **step**	200^{th} **step**	300^{th} **step**	400^{th} **steps**	500^{th} **steps**
MDP	386.26	1391.36	2482.56	3581.24	4687.48
adaptive rule based mechanism	385.72	1422.36	2478.58	3487.15	4514.26
fixed-cycle decision	440.46	1593.24	2739.72	4023.74	5237.80

Table 6.2: Means of the cumulative waiting cars at some steps of the simulation

Figure 6.2 shows the mean length of the queue with respect to the time-steps of the simulation.

Figure 6.3 shows the mean value of the cumulative number of waiting cars, and the standard deviation for the 50 simulations, for the three above types of decision making strategies, with respect to the time. Notice that the standard deviation is plotted every 20 steps for all the three types of decision, but we have shifted a little the bar plot of the Markov decision and the fixed-cycle decision in order to make them clearly visible.

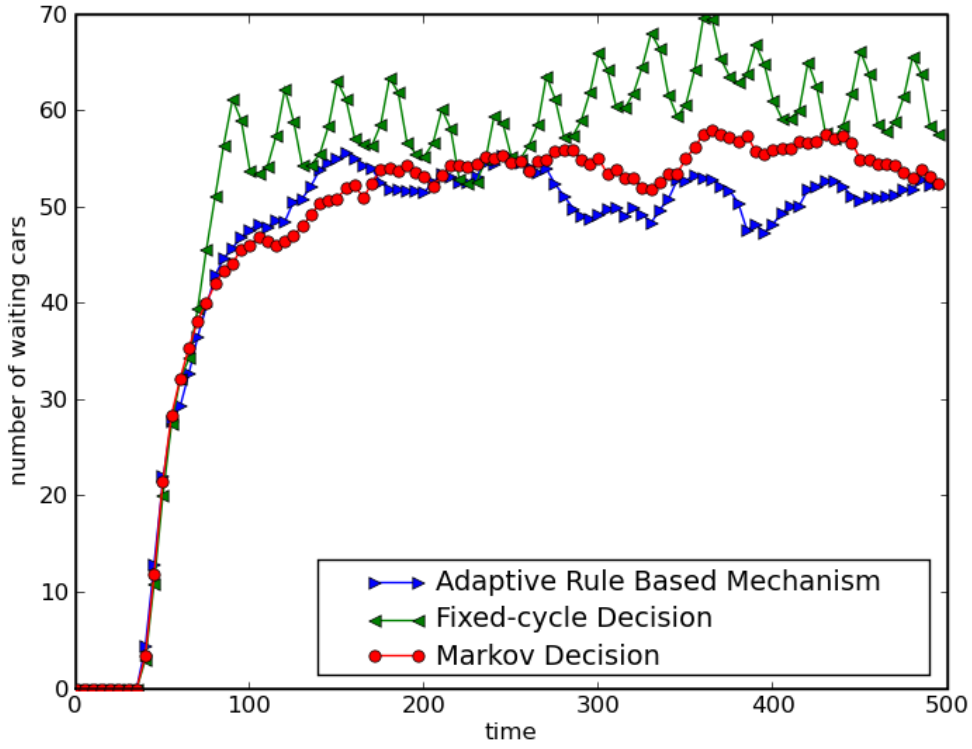

Figure 6.2: Mean number of waiting cars with respect to time

6.3.2 Interpretation

Table 6.1 shows that even if the number of waiting cars for a traffic junction controlled by our decision is higher than the number of waiting cars for a junction controlled by the adaptive rule-based mechanism, there is not a big difference. We can also notice that our decision outperforms the fixed-cycle decision.

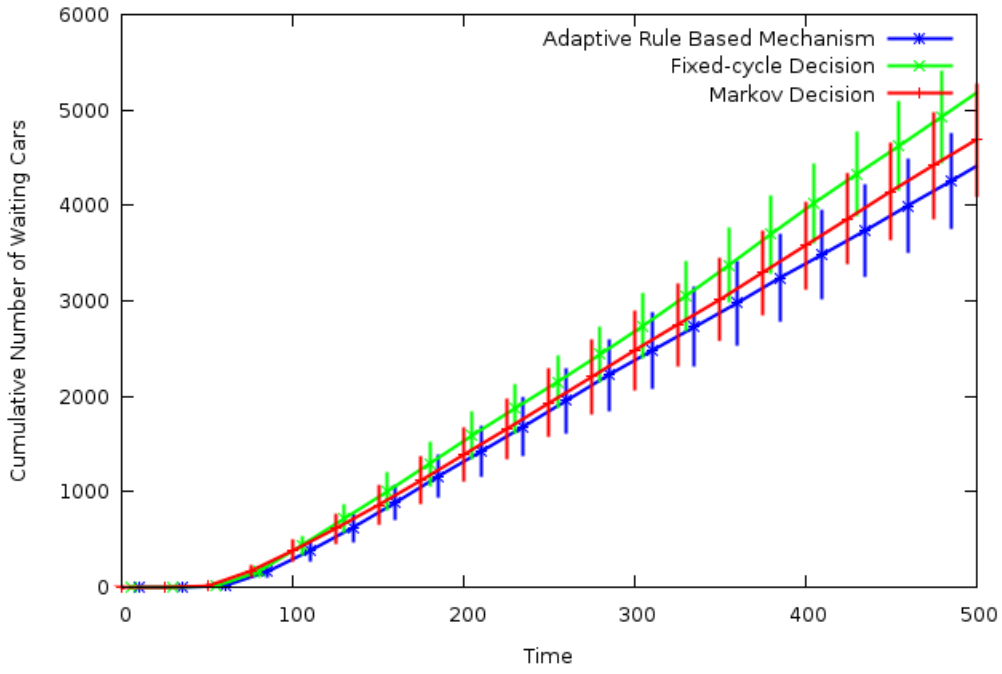

Figure 6.3: Mean and Standard deviation the cumulative number of waiting cars with respect to time

Graph 6.2 reaffirms the results presented in table 6.1. From this graph, it is more conceivable that the curve representing the performance of our decision and the adaptive rule-based mechanism are not far from each others. We can also notice that the graph representing the fixed-cycle decision is very variable from a time step to another one.

Table 6.2 gives a global estimation of the performance of our algorithm. The difference between the values given by the three methods is higher here. That is because we are considering cumulative values. Graph 6.3 also shows that the standard deviation for the 3 methods are approximately the same. Nevertheless, the number of waiting cars for the fixed-cycle decision is very variable, shown by the length of the bar of its standard deviation. Another remark is that as the simulation goes along, the number of waiting cars increases. This is a normal behaviour regarding the fact that the number of waiting cars is added up at every time step of the simulation. Nevertheless, the curve representing the fixed-cycle decision is increasing faster than the other two.

To summarise, even if our decision does not perform better than the adaptive rule based mechanism, it is performing well compared to the fixed-cycle based decision that many traffic lights still use nowadays.

7. Conclusion

In this report, we have first explained that reinforcement learning is a process in which an agent learns about the environment in order to make the appropriate decision for a specific situation. MDP provides an interesting framework to model the decision making, and the compact representation of the system provided by a MDP makes the decision strategy easier to implement. Value iteration algorithm, as well as another reinforcement learning methods like Q-learning, SARSA, and model-based reinforcement learning, mainly consist of finding the policy π which maps each state of the system to the action to be taken. Then, the concept of decision-making has been applied to a real-life problem which is the control of a traffic system. A brief review of different attempts to solve the problem has been presented. The review covered different approaches using reinforcement learning and MDP representation, as well as other approaches namely Fuzzy logic, artificial neural network and evolutionary algorithms.

After explaining all the background needed in the work, we have started to see the practical side by describing how an isolated traffic junction can be modelled as a finite MDP. We have gone on to show how the value iteration algorithm can be used to solve our decision problem. A python program has been used to elaborate the representation of the system as a finite MDP and to implement the value iteration algorithm in order to set up our decision strategy. We then performed simulations of the crossing controlled by our intelligent traffic light using SUMO, after constructing the network, generating the vehicles and implementing the traffic light decision. We have conducted series of experiments for different simulation time. The count of waiting cars permitted us to evaluate the performance of our algorithm compared to two other decision strategies, namely a fixed-cycle decision and an adaptive rule-based mechanism. The fixed-cycle decision consists changing the state of the crossing every 30 steps of the simulation, whereas, the adaptive rule-based mechanism changes the state of the traffic light when the number of waiting cars on the blocked road is twice the number of waiting cars on the closed road. Finally, results have shown that despite the fact that our decision strategy is not the most performant, it follows very closely the performance of the rule-based mechanism, and outperforms the fixed-cycle decision.

Even the approach using value iteration algorithm has been rarely used in the domain of controlling a traffic system, this work, by showing its performance, has made it an attractive one to pursue. Nevertheless, some possible perspectives to this work could be to develop the model more in order to make it more realistic by avoiding some assumptions such as the uniformity of vehicles types, the possibility of turning of the cars at the junction. The model can also be improved in order to gain more performance. Regarding the rapid development of traffic nowadays, this essay is dealing only with a small part of the whole urban traffic control problem. It can be extended to a larger problem, by considering first the control of multiple junctions which can be modelled as a multi-agents cooperative reinforcement learning problem.

Appendix A. Implementation of the Value Iteration Code for the Traffic Control Decision Strategy

This implementation summarises the main part of our work. It includes the modelisation of the system as a MDP and the value iteration algorithm, in order to get the decision stategy represented here by the dictionary of state-action correspondence.

```
""" value iteration to solve the decision making for the problem of traffic junction """

from pylab import zeros
from scipy import factorial
from math import exp,fabs
import random
m=20             #the maximum value of m (the number of waiting cars at the NS road)
n=20             #the maximum value of n (the number of waiting cars at the WE road)
N=2*(m+1)*(n+1)  #the number of possible states of the system
gamma=0.8        # discout factor
theta=10e-10     #small number to stop the loop in the value iteration
Cross_state=['N-S open','E-W open']


#-----------------------------------------------------------------------------------------------------
#definition of the set of states
def states(m,n):

        '''The entire state of the system is represented by  a list composed of each
        possible state of the system. Then, each possible state is a list composed of
        the state of the cross, the number of circulating cars, and the number of
        waiting cars '''

        state=[]
        for elt in Cross_state:
                for non_stop in range (m+1):
                        for stop in range (n+1):
                                state.append([elt,non_stop,stop])

        return state
#-----------------------------------------------------------------------------------------------------


#-----------------------------------------------------------------------------------------------------
#definition of the reward function
def reward_function(state):

        '''The reward is represented as a vector. Each component is the reward that one has for each state.
        This function is returning two such vectors, according to the two possible actions taken'''

        reward_change=[]
        reward_stay=[]

        for i in range (len(state)/2):              #for all the states such that N-S is open
                if (state[i][2]>=2*(state[i][1])):
                        reward_change.append(1)
                        reward_stay.append(0)
                else:
                        reward_change.append(-1)
                        reward_stay.append(1)
        for i in range ((len(state))/2,len(state)):  #for all the states such that W_E is open
                if (state[i][1]>=2*(state[i][2])):
                        reward_change.append(1)
                        reward_stay.append(0)
                else:
                        reward_change.append(-1)
                        reward_stay.append(1)


        return reward_change,reward_stay
#-----------------------------------------------------------------------------------------------------



#-----------------------------------------------------------------------------------------------------
#definition of the transition probability for the action change
def transistion_probability_for_change(state):

        '''The transition probability is a matrix for each action to be taken.
        The columns are the following possible state if taking an action, and
        the rows are the previous states of the system. This function returns
        the matrix when taking the action Change'''
        c=[]
        for i in range (len(state)):
```

```
                ci=0
                for j in range(len(state)):
                        if state[i][0]!=state[j][0]:
                                if (state[j][1]==state[i][1] and state[j][2]==state[i][2]+1)==True:
                                        ci=ci+1
                                if (state[j][1]==state[i][1] and state[j][2]==state[i][2]-1)==True:
                                        ci=ci+1
                                if (state[j][1]==state[i][1]+1 and state[j][2]==state[i][2]+1)==True:
                                        ci=ci+1
                                if (state[j][1]==state[i][1]+1 and state[j][2]==state[i][2]-1)==True:
                                        ci=ci+1
                                if (state[j][1]==state[i][1]+1 and state[j][2]==state[i][2])==True:
                                        ci=ci+1
                                if (state[j][1]==state[i][1]-1 and state[j][2]==state[i][2]+1)==True:
                                        ci=ci+1
                                if (state[j][1]==state[i][1]-1 and state[j][2]==state[i][2]-1)==True:
                                        ci=ci+1
                                if (state[j][1]==state[i][1]-1 and state[j][2]==state[i][2])==True:
                                        ci=ci+1
                c.append(ci)

        P_change=zeros([len(state),len(state)])
        for i in range (len(state)):
                for j in range (len(state)):
                                if state[i][0]==state[j][0]:
                                        P_change[i,j]=0
                                else:
                                        if (state[j][1]==state[i][1] and state[j][2]==state[i][2]):
                                                P_change[i,j]=1.0/2
                                        else:
                                                if (state[j][1]==state[i][1] and
                                                state[j][2]==state[i][2]+1)
                                                or(state[j][1]==state[i][1] and
                                                state[j][2]==state[i][2]-1) or
                                                (state[j][1]==state[i][1]+1 and
                                                state[j][2]==state[i][2]+1) or
                                                (state[j][1]==state[i][1]+1 and
                                                state[j][2]==state[i][2]-1) or
                                                (state[j][1]==state[i][1]+1 and
                                                state[j][2]==state[i][2]) or
                                                (state[j][1]==state[i][1]-1 and
                                                state[j][2]==state[i][2]+1) or
                                                (state[j][1]==state[i][1]-1 and
                                                state[j][2]==state[i][2]-1) or
                                                (state[j][1]==state[i][1]-1 and
                                                state[j][2]==state[i][2]):
                                                        P_change[i,j]=1.0/(2*c[i])
                                                else:
                                                        P_change[i,j]=0

                return P_change
#-----------------------------------------------------------------------------------------------------------


#-----------------------------------------------------------------------------------------------------------
#definition of the transition probability for the action Stay
def transistion_probability_for_stay(state):

        '''The transition probability is a matrix for each action to be taken.The columns are
        the following possible state if taking an action, and the rows are the previous states
        of the system. This function returns the matrix when taking the action Stay'''
        d=[]
        for i in range (len(state)):
                di=0
                for j in range(len(state)):
                        if state[i][0]==state[j][0]:
                                if (state[j][1]==state[i][1]-2 and
                                (state[j][2]==state[i][2]-2 or
                                state[j][2]==state[i][2]-1 or
                                state[j][2]==state[i][2] or
                                state[j][2]==state[i][2]+2 or
                                state[j][2]==state[i][2]+1))==True:
                                        di=di+1
                                if (state[j][1]==state[i][1]-1 and
                                (state[j][2]==state[i][2]-2 or
                                state[j][2]==state[i][2]-1 or
                                state[j][2]==state[i][2] or
                                state[j][2]==state[i][2]+2 or
                                state[j][2]==state[i][2]+1))==True:
                                        di=di+1
                                if (state[j][1]==state[i][1] and
                                   (state[j][2]==state[i][2]-2 or
                                   state[j][2]==state[i][2]-1 or
                                   state[j][2]==state[i][2] or
                                   state[j][2]==state[i][2]+2 or
                                   state[j][2]==state[i][2]+1)) ==True:
                                        di=di+1
                                if (state[j][1]==state[i][1]+2 and
                                   (state[j][2]==state[i][2]-2 or
                                   state[j][2]==state[i][2]-1 or
                                   state[j][2]==state[i][2] or
                                   state[j][2]==state[i][2]+2 or
```

```
                            state[j][2]==state[i][2]+1)) ==True:
                                di=di+1
                        if (state[j][1]==state[i][1]+1 and
                            (state[j][2]==state[i][2]-2 or
                            state[j][2]==state[i][2]-1 or
                            state[j][2]==state[i][2] or
                            state[j][2]==state[i][2]+2 or
                            state[j][2]==state[i][2]+1)) ==True:
                                di=di+1
        d.append(di)


P_stay=zeros([len(state),len(state)])
for i in range (len(state)):
        for j in range (len(state)):
                if (2<=state[i][1]<=m-2 and 2<=state[i][2]<=n-2 ):
                        if state[i][0]!=state[j][0]:
                                P_stay[i,j]=0
                        else:
                                if state[i][0]=='N-S open':
                                        P_m=0
                                        P_n=0
                                        if state[j][1]==(state[i][1]-2):
                                                P_m=3.0/16

                                        if state[j][1]==state[i][1]-1:
                                                P_m=9.0/32

                                        if state[j][1]==state[i][1]:
                                                P_m=9.0/32

                                        if state[j][1]==state[i][1]+2:
                                                P_m=1.0/12

                                        if state[j][1]==state[i][1]+1:
                                                P_m=2.0/12

                                        if state[j][2]==state[i][2]-2:
                                                P_n=(1.0/3)*(1.0/20)

                                        if state[j][2]==state[i][2]-1:
                                                P_n=(2.0/3)*(1.0/20)

                                        if state[j][2]==state[i][2]:
                                                P_n=(3.0/8)*(19.0/20)

                                        if state[j][2]==state[i][2]+2:
                                                P_n=(1.0/4)*(19.0/20)

                                        if state[j][2]==state[i][2]+1:
                                                P_n=(3.0/8)*(19.0/20)

                                        P_stay[i][j]=P_n*P_m

                                else:
                                        P_m=0
                                        P_n=0
                                        if state[j][2]==(state[i][2]-2):
                                                P_n=3.0/16

                                        if state[j][2]==state[i][2]-1:
                                                P_n=9.0/32

                                        if state[j][2]==state[i][2]:
                                                P_n=9.0/32

                                        if state[j][2]==state[i][2]+2:
                                                P_n=1.0/12

                                        if state[j][2]==state[i][2]+1:
                                                P_n=2.0/12

                                        if state[j][1]==state[i][1]-2:
                                                P_m=(1.0/3)*(1.0/20)

                                        if state[j][1]==state[i][1]-1:
                                                P_m=(2.0/3)*(1.0/20)

                                        if state[j][1]==state[i][1]:
                                                P_m=(3.0/8)*(19.0/20)

                                        if state[j][1]==state[i][1]+2:
                                                P_m=(1.0/4)*(19.0/20)

                                        if state[j][1]==state[i][1]+1:
                                                P_m=(3.0/8)*(19.0/20)

                                        P_stay[i][j]=P_m*P_n


                else:
```

```
                                if state[i][0]!=state[j][0]:
                                        P_stay[i,j]=0
                                else:
                                        if (state[j][1]==state[i][1]-2 and
                                                (state[j][2]==state[i][2]-2 or
                                                state[j][2]==state[i][2]-1 or
                                                state[j][2]==state[i][2] or
                                                state[j][2]==state[i][2]+2 or
                                                state[j][2]==state[i][2]+1)) or (
                                        state[j][1]==state[i][1]-1 and
                                                (state[j][2]==state[i][2]-2 or
                                                state[j][2]==state[i][2]-1 or
                                                state[j][2]==state[i][2] or
                                                state[j][2]==state[i][2]+2 or
                                                state[j][2]==state[i][2]+1)) or (
                                        state[j][1]==state[i][1] and
                                                (state[j][2]==state[i][2]-2 or
                                                 state[j][2]==state[i][2]-1 or
                                                 state[j][2]==state[i][2] or
                                                 state[j][2]==state[i][2]+2 or
                                                 state[j][2]==state[i][2]+1)) or(
                                        state[j][1]==state[i][1]+2 and
                                                (state[j][2]==state[i][2]-2 or
                                                state[j][2]==state[i][2]-1 or
                                                state[j][2]==state[i][2] or
                                                state[j][2]==state[i][2]+2 or
                                                state[j][2]==state[i][2]+1)) or(
                                        state[j][1]==state[i][1]+1 and
                                                (state[j][2]==state[i][2]-2 or
                                                state[j][2]==state[i][2]-1 or
                                                state[j][2]==state[i][2] or
                                                state[j][2]==state[i][2]+2 or
                                                state[j][2]==state[i][2]+1)):

                                                P_stay[i][j]=1.0/d[i]

        return P_stay
#-------------------------------------------------------------------------------------------------------------


#-------------------------------------------------------------------------------------------------------------
#value iteration algorithm
def value_iteration(state,reward1,reward2,probability_change,probability_stay,action):

        '''This is computing the policy which is a map from each possible states of the system
        to the action that should be taken. It returns a dictionary in which keys are the states
        and the values are the corresponding action to take'''


        V = []
        v0 = []

        for ii in range(len(state)):   #construction of V[0]
                v0.append(0)
        V.append(v0)

        i=1
        Stop=False
        while (Stop==False):           #construction of V[i] until convergence
                vi = []
                diff=[]
                action_taken=[]
                condition=True
                for j in range (len(state)): #construction of each component of v[i],
                                             #each component corresponds to one state
                        sum_change=0
                        sum_stay=0
                        for k in range (len(state)):#calculation of the sumation
                            sum_change = sum_change + ((probability_change[j][k])*(V[i-1][k]))
                            sum_stay = sum_stay + ((probability_stay[j][k])*(V[i-1][k]))

                        Q_change=reward1[j]+(gamma*sum_change) #the value function for the action 'change'
                        Q_stay=reward2[j]+(gamma*sum_stay)   #the value function for the action 'stay'

                        #finding the maximum of the value functions according to the two actions
                        if (Q_change>Q_stay):
                                vi.append(Q_change)
                                action_taken.append(action[0])
                        if (Q_stay>Q_change):
                                vi.append(Q_stay)
                                action_taken.append(action[1])
                        if (Q_change==Q_stay):
                                tmp=random.choice([Q_change,Q_stay])
                                vi.append(tmp)
                                if tmp==Q_change:
                                        action_taken.append(action[0])
                                else:
                                        action_taken.append(action[1])

                        diff.append(fabs(vi[j]-V[i-1][j]))  #construction of a vector of the difference
                        condition=condition and fabs(vi[j]-V[i-1][j])<theta
```

```
                V.append(vi)
                if (condition==True):
                        print 'The iteration has stopped at step',i
                        Stop=True

                        #construction of the dictionary of policy
                        decision={}
                        for i in range (len(state)):
                                decision[str(state[i])]=action_taken[i]
                                #write the correspondence state-action in a file
                                decisionmaking=open('decisionfile.dat','a')
                                text=str(state[i][0])+','+ str(state[i][1])+','+str(state[i][2])+','+str(action_taken[i])+'\n'
                                decisionmaking.write(text)
                                decisionmaking.close()

                        print decision
                else:
                        i=i+1


        return decision



# main ================================================
if __name__ == "__main__":
        set_of_state=states(m,n)
        print "Set of states"
        print set_of_state
        print 'The number of possible state is',len(set_of_state)
        action=['Change','Stay the same']
        set_of_rewardch,set_of_rewardst=reward_function(set_of_state)
        print 'Vector of rewards for the action change'
        print set_of_rewardch
        print 'Vector of rewards for the action stay'
        print set_of_rewardst
        P_change=transistion_probability_for_change(set_of_state)
        print 'transition probability for change'
        print P_change
        P_stay=transistion_probability_for_stay(set_of_state)
        print 'Transition probability for stay'
        print P_stay
        d=value_iteration(set_of_state,set_of_rewardch,set_of_rewardst,P_change,P_stay,action)
```

Acknowledgements

First of all, I would like to express my heartiest thanks to the Almighty God for His enduring grace and mercy over my life.

I am very thankful towards my supervisor Doctor Jesús Cerquides Bueno who spent his invaluable time and effort guiding me in the proper direction throughout this work. Without him, all my effort would have been in vain.

Special thanks are due to my tutor Mihaja Ramanantoanina for her advices, and corrections in writing this essay. My profound gratitude is expressed to Prof. Neil Turok through whose initiative was AIMS instituted, to Prof. Fritz Hahne the director of AIMS, to Jan Groenewald and to Frances Aron.

Finally, I would like to thank my family and all my friends for their encouragement which have helped me to successfully accomplish this essay.

References

[Abd03] B. Abdulhai, *Reinforcement Learning for the True Adaptive Traffic Signal Control*, Journal of Transportation Engineering **129** (2003), 278–285.

[Chi92] S. Chiu, *Adaptive Traffic Signal Control Using Fuzzy Logic*, Proceedings of the IEEE Intelligent Vehicles Symposium (1992), 98–107.

[dOBdS+06] D. de Oliveira, A. Bazzan, B. da Silva, W. E. Basso, and L. Nunes, *Reinforcement Learning based Control of Traffic Lights in Non-stationary Environments: A Case Study in a Microscopic Simulator*, European Workshop on Multi-Agent Systems, 2006.

[FBG92] M. D. Foy, R. F. Benokohal, and D. E. Goldberg, *Signal timing determination using genetic algorithms*, Transportation Research Record (1992), no. 1365, 108–115.

[Gaw98] C. Gawron, *Simulation-Based Traffic Assignment*, Phd, University of Cologne, 1998, Available from `http://sumo.sourceforge.net/docs/GawronDiss.pdf`.

[GKPV03] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman, *Efficient Solution Algorithm for Factored MDPs*, Journal of Artificial Intelligence **19** (2003), 399–468.

[Hai00] O. Haitao, *Urban Intelligent Traffic Control System Based on Multi-agent Technology*, Acta Electronica Silica (2000), 52–55.

[HFG98] J. J. Henry, J. L. Frages, and J. L. Gallego, *Neuro-Fuzzy Techniques for Traffic Control*, Control Engineering **6** (1998), 755–761.

[KKY96] T. K. Khiang, M. Khalid, and R. Yusof, *Intelligent Traffic Lights Control By Fuzzy Logic*, Malaysian Journal of Computer Science **9** (1996), no. 2, 29–35.

[Kra98] S. Krauß, *Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics*, Ph.D. thesis, University of Cologne, 1998, Available from `http://sumo.sourceforge.net/docs/KraussDiss.pdf`, p. 116.

[KWG97] S. Krauß, P. Wagner, and C. Gawron, *Metastable States in a Microscopic Model of Traffic Flow*, Physical Review E **55** (1997), 55–97.

[LB95] K. Lee and Z. Bien, *A Corner Matching Algorithm Using Fuzzy Logic*, Available from `http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/papers/files/lee.ps.gz`.

[Liu97] Z. Liu, *Hierarchical Fuzzy Neural Network Control for Large Scale Urban Traffic Systems*, Information and Control **26** (1997), no. 6, 441–448.

[Liu07] ———, *A Survey of Intelligent Methods in Urban Traffic Signal Control*, IJCSNS International Journal of Computer Science and Network Security **7** (2007).

[PR01] M. Patel and N. Ranganathan, *An Intelligent Decision-making System for Urban Traffic Control Applications*, Vehicular Technology **50** (2001), 816–829.

[SB98] R. S. Sutton and A. G. Barto (eds.), *Reinforcement Learning: An Introduction*, The Massachusets Institute of Technology Press, 1998.

[SL08] I. Szita and A. Lorincz, *Factored Value Iteration Converges*, Acta Cybernetica **18** (2008), no. 4, 615–635.

[SSP+05] M. Steingrover, R. Schouten, S. Peelen, E. Nijhuis, and B. Bakker, *Reinforcement Learning of Traffic Light Controllers Adapting to Traffic Congestion*, BNAIC: Belgian-Netherlands Conference on Artificial Intelligence (Katja Verbeeck, Karl Tuyls, Ann Now, Bernard Manderick, and Bart Kuijpers, eds.), Koninklijke Vlaamse Academie van Belie voor Wetenschappen en Kunsten, 2005, pp. 216–223.

[TA96] T. L. Thorpe and C. W. Anderson, *Traffic Light Control Using SARSA with Three State Representations*, Tech. report, Colorado State University, Computer Science Department, 1996.

[TBP+98] H. Taale, T. Bäck, M. Preu, A. E. Eiben, and J. M. De Graaf, *Optimizing Traffic Light Controllers by Means of Evolutionary Algorithms*, Proceeding of the EUFIT'98 Conference, Aachen, Germany (1998).

[WVK04] M. Wiering, J. Vreeken, and A. Koopman, *Intelligent Traffic Light Control*, Tech. report, University Utrecht, 2004.