From A to B, how to get there fast?

Travel Time Predictions Based on Markov Decision Processes

BMI Thesis

December 10th, 2009

by

Cindy de Groot

Copyright ©Cindy de Groot, Weesp 2009.

VRIJE UNIVERSITEIT

From A to B, how to get there fast?

Travel Time Predictions Based on Markov Decision Processes

BMI Thesis

by

Cindy de Groot

Abstract

Nowadays, travel time predictions are very important to plan when, where and how to travel. In this study we develop a travel-time prediction model using the theory of Markov Decision Processes (MDP). We describe two reinforcement learning methods which can be used for solving the MDP model: Dynamic Programming and Temporal-Difference learning. Both methods are applicable in real-time and still work when input data is unreliable or missing.

In contrast with recent studies the methods developed in this study react automatically on changing environments, i.e. incidents, work zones, and adverse weather conditions. Hence, the methods are valid in all possible traffic conditions. Additionally, the methods used in this study do not only give a route-travel-time estimation but also provide a direction in which to travel.

To validate the methods developed, we used a dataset containing vehicle-specific information about the A10, the ring road around Amsterdam in the Netherlands. However, the same methods could be applied to other freeway segments if appropriate speed data is available. The results of our study were quite in line with our expectations.

Preface

This thesis is part of acquiring the Masters degree in Business Mathematics and Informatics. Business Mathematics and Informatics is a multidisciplinary program, aimed at business processes optimization by applying a combination of methods based upon mathematics, computational intelligence and business management. These three disciplines will also play a central role throughout this thesis.

The subject of this study is travel-time predictions. Travel-time predictions can be a vital tool for companies for planning when, where and how to travel. However, not only the business aspect, but also the mathematical and computer science aspect of Business Mathematics and Informatics are present in this thesis. The purpose of present study is to develop a real-time travel time prediction method using Markov Decision Processes (MDPs). We will give a basic introduction to the mathematical background of these processes and we will describe some methods which can be used solve the developed MDP model. Eventually, we made a simulation program to visualize the results of the solution methods to the model described applied on a real dataset. This dataset contains vehicle specific information from the A10, the ring road around Amsterdam, and is collected by Rijkswaterstaat, a part of the Dutch Ministry of Transport, Public Works and Water Management. I would like to thank ir. Henk Taale from Rijkswaterstaat for putting this dataset at my disposal.

After reading this thesis, one should be able to formulate an answer to the following questions: "What are Markov Decision Processes?", "How can Markov Decision Processes be applied to real-life situations?" and "How can we find an optimal solution to a real-life problem using MDPs?".

Finally, I would like to thank my supervisor Dennis Roubos, MSc for his help and support and for acquainting me with the subject of Markov Decision Processes.

Cindy de Groot
Weesp, 2009

Contents

Abstract **v**

Preface **vii**

Introduction **xi**

1 Mathematical Models and Methods **1**

1.1 Supervised learning 1

1.1.1 Neural Networks 1

1.1.2 State-space neural network 3

1.1.3 Bayesian trained neural network 4

1.1.4 A committee of neural networks 5

1.2 Reinforcement learning 6

1.2.1 Markov Decision Process 6

1.2.1.1 Value functions 7

1.2.1.2 Optimal value functions 9

2 The Impact of Travel Information on Travel Behaviour **11**

2.1 Structure of travel processes 11

2.2 Effects of weather on the travel conditions 12

3 Model development and data collection **15**

3.1 Data Collection 15

3.2 Application of a MDP using Dynamic Programming 16

3.2.1 Definition of states and variables 18

3.2.2 Link travel time estimation 19

3.2.3 Transition Matrix 19

3.2.4 The Reward 20

3.2.5 The Optimal Value Functions 21

3.2.5.1 Policy Evaluation 21

3.2.5.2 Policy Improvement 22

3.2.5.3 Policy Iteration 23

3.2.5.4 Value Iteration 24

3.3 Application of a MDP using Temporal-Difference Learning . . . 26
3.3.1 The Forward View of TD(λ) . . . 27
3.3.2 The Backward View of TD(λ) . . . 28

4 Results from the motorway A10, Netherlands **31**
4.1 Markov Decision Processes . . . 31
4.1.1 Definition of States and Variables . . . 31
4.1.2 Transition Matrix . . . 32
4.2 Dynamic Programming . . . 39
4.3 Temporal-Difference Learning . . . 40

Discussion **51**

Bibliography **52**

Index **54**

Introduction

In the past, traffic signs represented the length of congestion. Based on this information a decision about the direction to take has to be made. This information, however, is not very useful. The actual travel time by congestion depends on the type of congestion, cause of congestion and the time of the day (week, month or year). One can easily imagine the difference between two kilometers stationary traffic and four kilometers slow moving traffic. It still can be much faster to take the direction with the congestion over a length of four kilometers. Additionally, the cause of a congestion and time of the day may influence the travel time. For example, most roadwork will be done during the night or in the weekends, while congestions because of open bridges or waiting for a railway crossing mostly happen during the day.

Nowadays, we display travel time prediction (in minutes) on the traffic signs. Travel time predictions are very important to plan when, where and how to travel. To be useful in practice, the models used for predicting travel times should be fast and applicable in real-time. Furthermore, the results of the predictions should be accurate and valid in all possible traffic conditions. Finally, the models have to work when input data is missing.

Because of the wide interest of travel time predictions in our society, this subject has been studied extensively in recent years. Travel time can be predicted from historical data through analysing limited traffic information from certain fields. Data acquired through analysis can then be used, applying varying techniques; e.g, Fuzzy theory, Artificial Intelligence, statistics and mathematics; to develop travel-time prediction models. Traffic-time prediction models which are most commonly used are time series approaches (Nihan, 1980 [17]; Lee et al., 1999 [19]), linear regression (Zhang, 2003 [25]), nearest neighbour techniques (Clark, 2003 [5]), neural networks (Lint et al., 2005 [13]; Hinsbergen et al., 2009 [7]) and so-called *committee* or *ensemble* approaches, in which multiple model-predictions are combined (Zheng, 2006 [23]). The last two approaches, neural networks and committees have shown a high level of accuracy for prediction of traffic conditions (Hinsbergen, 2007 [6]). Little time, however, has been devoted to Markov Decision Processes (MDPs) to develop a real-time travel time prediction model. Contrary to neural networks and committees, which are supervised learning methods, the MDP method is an example of an reinforcement learning method. Reinforcement learning tries to find the best solution to a problem facing a learning agent interacting with its environment to achieve a goal, while supervised learning learns from examples provided by a knowledgeable external supervisor.

The purpose of the present study is to develop a real-time travel time prediction method using Markov Decision Processes. Chapter 1 starts with an theoretical explanation of the mathematical models and methods underlying travel time predictions. Chapter 2 discusses the impact of travel information on travel behaviour. Chapter 3 applies the theory behind the MDPs of the first chapter to develop a model. The model development also includes a description of the collected data and also an explanation of several methods to find the solution of a MDP. Chapter 4 shows the result analysis. We finish this thesis with a discussion, including some comments about further research.

Chapter 1

Mathematical Models and Methods

Different methods are used to predict real-time travel times.

- *Supervised learning*: learning from examples provided by a knowledgeable external supervisor.
- *Reinforcement learning*: finding the best solution to a real problem facing a learning agent interacting with its environment to achieve a goal.

To help understand the principles of the different methods and how they are used for predicting real-time travel times the next sections will give an brief explanation. Section 1.1 discusses Neural Networks and committee methods as examples of supervised learning methods for making predictions. Section 1.2 explains the theory behind Markov Decision Processes: a framework for decision-making where outcomes are (partly) random and can be controlled by an agent.

1.1 Supervised learning

Supervised learning is a machine learning technique for learning a function from training data. The training data consist of pairs of input objects and desired outputs. The input typically consists of vectors. The output of the function can be a continuous value, called regression, or can predict a class label of the input object, called classification. The task of the supervised learner is to predict the value of the function for any valid input object after having seen a number of training examples (i.e., pairs of input and target output). Supervised learning is the kind of learning studied in most current research in machine learning, statistical pattern recognition and neural networks.

1.1.1 Neural Networks

Currently, the method most commonly used for travel time prediction are neural networks. In the application several of potential networks will be developed of which the best, based on the performance of the selected networks on an independent validation set, will be selected to make predictions. A neural network typically exists of three layers:

- The *input* layer takes the input signals and delivers these inputs to every neuron in the next layer.
- The *hidden* layer represents the non-linear function that specifies the states' behaviour.
- The *output* layer takes the hidden layers inputs and these are added to each output neuron. These outputs are the outputs of the neural network.

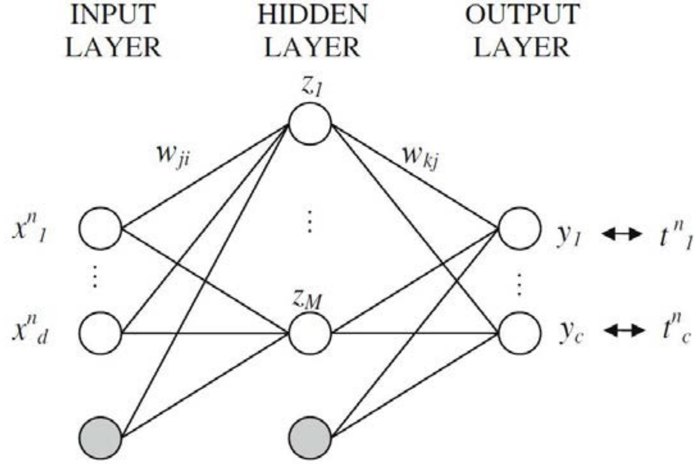

Figure 1.1. A neural network with d input elements, one hidden layer with M hidden nodes and c outputs, where the biases are represented as an extra node.

Figure 1.1 shows a graphical representation of a neural network. The input layer consists of d input elements, the hidden layer of M hidden nodes and the output layer of c outputs.

An output $y_k, k = (1, ..., c)$, can be mathematically described by the following equations:

$$y_k(x) = f_2 \left(\sum_{j=1}^{M+1} w_{kj} z_j \right) \quad \text{and} \quad z_j = f_1 \left(\sum_{i=1}^{d+1} w_{ji} x_i \right), \tag{1.1}$$

where w_{kj} and w_{ji} are called *weights* which are adjustable and whose values need to be estimated from data. The *bias weights (biases)* are represented by an extra node in a layer to the left (the grey nodes in Figure 1.1) which have a constant output of 1, so $x_{d+1} = 1$ and $z_{M+1} = 1$. The functions f_1 and f_2 are called *activation functions* and apply transformations to the weighted sum of the output of the units to the left. The weights $\vec{w}$ form a weight vector with a total of W weights (parameters). The input vector $\vec{x}^n \equiv (\vec{x_1}^n, ..., \vec{x_d}^n)$ is drawn from a data set $X \equiv \left(\vec{x}^1, ..., \vec{x}^N\right)$ of N data points. The output values of the network $\vec{y}\left(\vec{x}^n\right) \equiv \left(y_1\left(\vec{x}^n\right), ..., y_c\left(\vec{x}^n\right)\right)$ can be compared to the target values $\vec{t^n} \equiv (t_1^n, ..., t_c^n)$, drawn from a target data set $D \equiv (t_1^n, ..., t_c^n)$.

The values of the weight vector $\vec{w}$ of the network need to be learned from data, which is usually referred to as neural network training. Typically this learning mechanism is based on a maximum likelihood approach, equivalent to the minimization of an error function such as the sum of squared errors:

$$E_D = \frac{1}{2} \sum_{n=1}^{N} \left(y\left(\vec{x}^n; \vec{w}\right) - \vec{t^n} \right)^2 \tag{1.2}$$

Preferably, a regularisation term E_w is added to Equation (1.2) to avoid overfitting of the networks to the training data. A commonly used regulariser is the *partitioned weight decay* error term which has empirically been found by Krogh and Herts (1995, [2]) to improve network generalization. To build this regulariser define V groups of weights w_v, for example one for each layer and one for the biases. Then define the regulariser by:

$$\begin{aligned} E_W &= \sum_{v=1}^{V} \alpha_v E_{W,v} \\ E_{W,v} &= \frac{1}{2} \sum_{w \in w_v} w^2. \end{aligned} \tag{1.3}$$

The regularized performance (error) function then becomes:

$$E(w) = \beta E_D + \alpha E_W \tag{1.4}$$

The parameters α and β regulate to which extend the output error, (first term in Equation (1.3)) and the size of the weights (second term) contribute to the performance function. The minimum of this performance function can be found by *back-propagation* [16] or one of its many variations.

1.1.2 State-space neural network

Many recent studies have used a special neural network, called a state-space neural network (SSNN). In computer science, a state space is a description of a configuration of discrete states used as a simple model of machines. Formally, it can be defined as a tuple (N, A, I, O) where:

- N is a set of states;
- A is a set of arcs connecting the states;
- I is a nonempty subset of N that contains input states;
- O is a nonempty subset of N that contains the output states.

This special recurrent neural network is composed of a number of different layers, as can be seen from Figure 1.2. Therefore, this special type of neural network adds two extra layers to the network described above, one hidden layer and a state layer. Each neuron in the *state* layer represents one state. The output value is the value of the state.

Compared to simple neural networks SSNN have two main advantages. First, being a neural model, it has the flexibility to represent any non-linear function. Second, being a state-space model, the number of outside connections is minimal. The inputs will be the causes that drive the system operation and the outputs will be the effects observed on the system. Parallel inputs and outputs, and therefore causes and effects, are established between the neural model and the physical system. As has been proved by Hornik et al. (1989, [14]), every non-linear function can be represented by a neural network containing a single hidden layer composed of neurons whose transfer function is bounded. Thus, two neural networks can be concatenated; one of them representing the function that gives the state behaviour and the other representing the function that relates outputs to states. The mathematical form of such a model is (Zamarreno et al., 1998 [11]):

$$\begin{aligned}\hat{x}(t+1) &= W^h f_1\left(W^r \hat{x}(t) + W^i \vec{u}(t) + B^h\right)\\ \hat{y}(t) &= W^0 f_2\left(W^{h_2} \hat{x}(t) + B^{h_2}\right)\end{aligned} \tag{1.5}$$

where: $W^i, W^{h_1}, W^r, W^{h_2}, W^0$ are matrices with dimensions $h_1 \times n$, $s \times h_1$, $h_1 \times s$, $h_2 \times s$ and $m \times h_2$, respectively; B^{h_1} and B^{h_2} are two vectors with h_1 and h_2 elements respectively; and f_1 and f_2 are two functions that are applied element-wise to a vector or a matrix. Like with general neural networks, a back-propagation method is used to find the optimal set of weights for a state-space neural network.

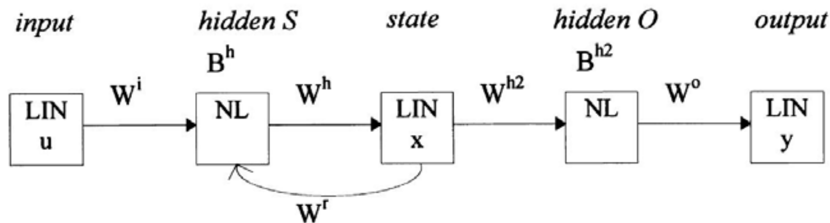

Figure 1.2. State-space neural network.

1.1.3 Bayesian trained neural network

However, instead of using maximum likelihood techniques, many researchers (Lint et al., 2005 [13], Hinsbergen et al., 2009 [7]; Zheng et al.; 2006 [23]) train neural networks from a Bayesian inference perspective (Hinsbergen et al., 2009 [7]). This has some major advantages in the application of the neural networks. First, error bars can be assigned to the predictions of a network. Second, an automatic procedure for weighing the two error parts E_D and E_W of the error function can be derived; the values of these weights can be inferred simultaneously from the training data without the need of a separate validation data set. Since all data is used for training, this method will result in better models. Third, the *evidence* measure merging from the Bayesian analysis can be used as an early stopping criterion in the training procedure. Finally, different networks can be selected and combined in a committee approach using this evidence measure.

From a Bayesian inference perspective, the parameters in a neural network should not be conceived as single values, but as a *distribution* of values representing various degrees of belief. The goal is then to find the posterior probability distribution for the weights after observing the dataset D, denoted by $p(w|D)$, which can be found using Bayes' theorem:

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)} \tag{1.6}$$

where $p(D)$ is the normalization factor, $p(D|w)$ represents a noise model on the target data and corresponds to the likelihood function, and $p(w)$ is the prior probability of the weights (Bishop, 1995 [3]). Although many forms of the prior and the likelihood function are possible, often Gaussian forms are chosen to simplify further analyses:

$$\begin{aligned} p(w) &= \frac{1}{Z_W(\alpha)} exp\left(-\sum_v \alpha_v E_{W,v}\right) \\ p(D|w) &= \frac{1}{Z_D(\beta)} exp\left(-\frac{\beta}{2}\sum_{n-1}^{N}\left(y\left(x^n;w\right)-t^n\right)^2\right) \end{aligned} \tag{1.7}$$

where $Z_W = \int exp\left(-\sum_v \alpha_v E_{w,v}\right) dw$ and $Z_D = \int exp\left(-\beta E_D\right) dD$ are normalizing constants and $\alpha = (\alpha_1, ..., \alpha_v)$ and β are called *hyper parameters* as they control the distributions of other parameters, the weights w of the network. The prior has zero mean and variances $1/\alpha_v$ for every group of weights, the likelihood function has zero mean and variance $1/\beta$. It can be seen that the exponents in Equation (1.7) take the form of the error functions E_W and E_D already introduced in Equation (1.2). Substituting Equation (1.7) in (1.6) results in an expression for the posterior:

$$p(w|D) = \frac{1}{Z_S(\alpha,\beta)} exp\left(-E(w)\right) \tag{1.8}$$

$$E(w) = \beta E_D + \sum_v \alpha_v E_{W,v} \tag{1.9}$$

where $Z_S(\alpha,\beta) = \int \left(-E(w)\right) dw$ is a normalizing constant. The maximum of the posterior $p(w|D)$, which is equivalent to minimizing Equation (1.9) can be found by back-propagation techniques again.

Although the methods described above might intuitively make sense, there are a number of serious drawbacks to the Neural Network approach. In the first place, by selecting the best network out of the candidate networks, a lot of effort involved in training networks is wasted. More seriously, the fact that one neural network model outperforms all other models on one particular validation data set does not guarantee that this neural network model indeed contains the "optimal" weights and structure, nor that this model has the best generalization capabilities. This completely depends on the statistical properties of the training and validation set (e.g., the amount of noise in the data), the complexity of the problem at hand and most importantly on the degree on which the training and validation set are representative for the true underlying process which is modeled. The network performing best on the validation set may therefore not be the one with the best performance on new data. These drawbacks can be overcome by combining all (or a representative selection of) trained neural network models in a committee.

1.1.4 A committee of neural networks

In a committee, the predictions of multiple models are combined. Bishop and Thodberg ([3], [22]) have proved that committees can lead to improved generalization. How to build a committee? First, construct many different networks with different numbers of hidden units and with different initial weight values. Second, train the network recursively with one of the back-propagation methods. For each of these models draw initial weight values for the hyper parameters, α and β, from their priors. These can be approximated by the same Bayesian inference framework that is used to approximate the posterior distributions of the weights. The posterior distribution of α and β given the data D is given by:

$$p(\alpha, \beta|D) = \frac{p(D|\alpha, \beta)p(\alpha, \beta)}{p(D)} \tag{1.10}$$

From previous research (among others, Bishop [3]) we obtained that this function is maximized when α and β have the following values:

$$\begin{aligned} \alpha_v &= \frac{\gamma_v}{2E_{W,v}} \\ \beta &= \frac{N-\gamma}{2E_D}, \end{aligned} \tag{1.11}$$

where $\gamma = \sum_v \gamma_v$ is the so-called number of well-determined parameters, which are given by:

$$\gamma_v = \sum j \left\{ \frac{\eta_j}{\eta_j + \alpha_v} (V^T I_v V)_{jj} \right\}, \tag{1.12}$$

where η_j is the jth eigenvalue of the A, V is the matrix of eigenvectors of A and I_v is a matrix with all elements zero except for the elements $I_{ii} = 1$ where i corresponds to a weight from a group v. A is the Hessian given by

$$A = \nabla\nabla E(w) - \beta\nabla\nabla E_D + \sum_v \alpha_v I_v. \tag{1.13}$$

In this summation negative eigenvalues are omitted (Thodberg, 1993 [22]). In practice, the optimal values for α and β as well as the optimal weight vector $\vec{w}$ need to be found simultaneously. A simple approach to do this is to use a standard iterative training algorithm, use the scaled conjugate gradient algorithm described by Møller [15], for example. Re-estimate values for α and β using Equations (1.11) and (1.12). When do we have to stop the iteration? As stopping criterion we can use evidence. When we consider a certain neural network i with a set of assumptions H_i, such as the number of layers and the number of hidden units, then the evidence of a network can be found using (Hinsbergen, 2009 [7])

$$p(D|H_i) = \int\int p(D|\alpha, \beta, H_i)p(\alpha, \beta|H_i)d\alpha d\beta. \tag{1.14}$$

The iteration can be stopped once the increase in evidence falls below a certain threshold value ς. Third, after all networks are trained, choose a selection of the better networks on the basis of their final evidences and construct a generalized committee given by a weighted combination of predictions of its L members using (Perrone, 1994 [18])

$$y_{GEN}(x) = \sum_{i=1}^{L} q_i y_i(x), \tag{1.15}$$

where different types of weights q_i are possible. Finally, combine the error bars of the selected networks and draw 95% confidence intervals by adding and substracting twice the standard deviation from the committee predictions. Error bars are used on graphs to indicate the error in a reported measurement.

The methods described above have proved to perform accurate results for travel time prediction in practice. However, in this investigation we will introduce another kind of models to solve the real-time prediction problem.

1.2 Reinforcement learning

Reinforcement learning is a method used to find the best solution to a real problem facing a learning agent interacting with its environment to achieve a goal. The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that *evaluates* the actions taken rather than *instructs* by giving correct actions. As a result the challenge of making a trade-off between exploration and exploitation arises. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that is has not selected before. The agent has to *exploit* what it already knows in order to obtain reward, but it also has to *explore* to make better action selections in the future. Another key factor of reinforcement learning is that is explicitly considers the whole problem, which is in contrast with many approaches that consider subproblems without addressing how they might fit into a larger picture.

Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system:

- The *policy* is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior.
- A *reward function* defines the goal in a reinforcement learning problem. Roughly speaking, it maps each perceived state (or state-action pair) of the environment to a single number, a *reward*, indicating the intrinsic desirability of that state.
- A *value function* specifies what is good in the long run. Roughly speaking, the *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.
- A *model* mimics the behaviour of the environment. For example, given a state and action, the model might predict the resulting next state and next reward.

In general, policies and reward functions may be stochastic. A reinforcement learning agent's sole objective is to maximize the total reward it receives in the long run. Maximizing the total long run reward, means in our problem space: minimizing the total predicted travel time. This is a dynamic decision problem, the future actions and rewards depend on decisions made in the past. Randomness is often involved in the evolution of the problem, hence the name stochastic dynamic programming or Markov decision theory. The latter because it emphasizes the connection with Markov and semi-Markov chains and processes.

1.2.1 Markov Decision Process

A reinforcement learning task that satisfies the Markov property is called a *Markov Decision Process*, or *MDP*. A stochastic process has the Markov property if the conditional probability distribution of future states of the process, given the present state and a constant number of past states, depends only upon the present state and not on the given states in the past, i.e., it is conditionally independent of these older states. In other words, the evolution of a Markov process from some point in time t_n does not depend on the history but only on the current state X_n. It can be seen as a memoryless property. In mathematical formulas this looks like:

$$\Pr\left\{X_{t_n} = x_n | X_{t_1} = x_1, \ldots, X_{t_{n-1}} = x_{n-1}\right\} = \Pr(X_{t_n} = x_n | X_{t_{n-1}} = x_{n-1}), \tag{1.16}$$

for all $t_1 < \ldots < t_n$.

In the case of real-time travel-time predictions the state and action spaces, denoted by $\mathcal{X}$ and $\mathcal{A}(x)$, respectively, are finite. Therefore we use finite Markov Decision Processes. A particular finite MDP is defined by its state and action sets and by the dynamics of the environment. The

policy, π, is a mapping from each state and action to the probability $\pi(x, a)$ of taking action a when in state x. Given any state and action, x and a, the probabilities of each possible next state, y, is

$$\mathcal{P}_{xy}^{a} = p(x, a, y) = Pr\{X_{t+1} = y | X_t = x, a_t = a\} \tag{1.17}$$

These quantities are called *transition probabilities*. Similarly, given any current state and action, x and a, together with any next state, y, the expected value of the *reward* is

$$\mathcal{R}_{xy}^{a} = r(x, a, y) = E\{r_{t+1} | X_t = x, a_t = a, X_{t+1} = y\} \tag{1.18}$$

Before we move on we will make the following three assumptions. Relaxing any of these is possible, but usually leads to additional constraints or complications. Moreover, in practical situations all these constraints are satisfied, but we have to check them (see Chapter 3) before we can apply this method on the real-time travel time problem. Before formulating the assumptions, it is convenient to define the notion of a path in a Markov chain.

Definition
A sequence of states $z_0, z_1, \ldots, z_{k-1}, z_k \in \mathcal{X}$ with the property that $p(z_0, z_1), \ldots, p(z_{k-1}, z_k) > 0$ is called a *path* of length k.

Assumption 1
$|\mathcal{X}| < \infty$ and $|\mathcal{A}| < \infty$, where $\mathcal{X}$ is the state space and $\mathcal{A}$ is the action space.

Assumption 2
For every policy π, there is at least one state $x \in \mathcal{X}$ (that may depend on π), such that there is a path from any state to x. If this is the case we call the chain *unichain*, state x is called recurrent.

Assumption 3
For every policy π, the greatest common divider of all paths from x to x is 1, for some recurrent state x. If this is the case we call the chain *aperiodic*.

Define the probability matrix P as follows: $P_{xy} = p(x, y)$ for some arbitrary action. Then, $\pi_t = \pi_{t-1} \times P$, and it follows immediately that $\pi_t = \pi_0 \times P^t$. For this reason we call $p^t(x, y) = P_{xy}^t$ the t-step transition probabilities. The limiting distribution is given by $\lim_{t\to\infty} \pi_t = \pi_*$, with the distribution π_* the unique solution of $\pi_* = \pi_* P$ independent of π_0.

Writing out the matrix equation $\pi_* = \pi_* P$ gives $\pi_*(x) = \sum_{y\in\mathcal{X}} \pi_*(y)p(y, x)$. The right-hand side is the probability that, starting from stationarity, the chain is in state x the next time unit. Note that there are $|\mathcal{X}|$ equations, but as this system is dependent, there is no unique solution. The solution is only unique if the equation $\sum_{x\in\mathcal{X}} \pi_*(x) = 1$ is added.

1.2.1.1 Value functions

Almost all reinforcement learning algorithms are based on estimating *value functions* - functions of states (or of state-action pairs) that estimate how good it is to perform a given action in a given state. Informally, the *value* of a state x under a policy π, denoted $\mathcal{V}^\pi(x)$, is the expected return when starting in x and following π thereafter. For a MDP two value functions can be defined:

1. *the state-value function for policy* π, $\mathcal{V}^\pi(x)$, defined as the expected return when starting in x and following π thereafter.

   $$\mathcal{V}^\pi(x) = E_\pi \sum_{t=0}^{T-1} r(X_t) = E_\pi \sum_{t=0}^{T-1} \sum_{y\in\mathcal{X}} p^t(x, y) r(y), \tag{1.19}$$

   where $E_\pi\{\cdot\}$ denotes the total expected reward given that the agent follows policy π and $x_0 = 0$.

Note that

$$
\begin{aligned}
&\sum_{x\in\mathcal{X}} \pi_*(x)\mathcal{V}_T(x) = \\
&\sum_{x\in\mathcal{X}} \pi_*(x) \sum_{t=0}^{T-1} \sum_{y\in\mathcal{X}} p^t(x,y)r(y) = \\
&\sum_{t=0}^{T-1} \sum_{y\in\mathcal{X}} \sum_{x\in\mathcal{X}} \pi_*(x)p^t(x,y)r(y) = \\
&\sum_{t=0}^{T-1} \sum_{y\in\mathcal{X}} \pi_*(y)r(y) = gT,
\end{aligned} \tag{1.20}
$$

where the average long run reward, g, is given by $g = \sum_{x\in\mathcal{X}} \pi_*(x)r(x)$.

2. *the action-value function for policy* π, $\mathcal{Q}^\pi(x,a)$, defined as the expected return starting from x, taking the action a, and thereafter following policy π.

$$
\mathcal{Q}^\pi(x,a) = E_\pi \sum_{t=0}^{T-1} r(X_t,a) = E_\pi \sum_{t=0}^{T-1} \sum_{y\in\mathcal{X}} p^t(x,a,y)r(y,a), \tag{1.21}
$$

The value functions can be estimated from experience with help of Monte Carlo methods [24], which involve averaging over random samples of actual returns. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value, $\mathcal{V}^\pi(x)$, as the number of times that state is encountered approaches infinity. If separate averages are kept for each action taken in a state, then these averages will similarly converge to the action values, $\mathcal{Q}^\pi(x,a)$. If there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain $\mathcal{V}^\pi$ and $\mathcal{Q}^\pi$ as parameterized functions and adjust the parameters to better match the observed returns.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming (a method to solve reinforcement learning problems) is that they satisfy particular recursive relationships. Let $\mathcal{V}(x) = \lim_{T\to\infty}[\mathcal{V}_T(x) - gT]$. Then, $\mathcal{V}(x)$ is the total expected difference in reward between starting in x and starting in stationarity. For any policy π and any state x, the two following consistency conditions hold between the value of x and the value of its possible successor states y:

$$
\mathcal{V}_{T+1}(x) = \mathcal{V}_T(x) + \sum_{y\in\mathcal{X}} \pi_T(y)r(y), \tag{1.22}
$$

for π_0 with $\pi_0(x)$ arbitrarily chosen. As $\pi_T \to \pi_*$

$$
\mathcal{V}_{T+1}(x) = \mathcal{V}_T(x) + g + o(1), \tag{1.23}
$$

where $o(1)$ means that this term disappears if $t \to \infty$. On the other hand, for $\mathcal{V}_{T+1}$ the following recursive formula exists:

$$
\mathcal{V}_{T+1}(x) = r(x) + \sum_{y\in\mathcal{X}} p(x,y)\mathcal{V}_T(y). \tag{1.24}
$$

Thus,

$$
\mathcal{V}_T(x) + g + o(1) = r(x) + \sum_{y\in\mathcal{X}} p(x,y)\mathcal{V}_T(y). \tag{1.25}
$$

Subtract gT from both sides, and take $T \to \infty$:

$$\mathcal{V}(x) + g = r(x) + \sum_{y \in \mathcal{X}} p(x, y)\mathcal{V}(y). \tag{1.26}$$

This equation is called the Poisson or Bellman equation. This equation averages over all the possibilities, weighing each by its probability of occurring. It states that the value of the start state must equal the value of the expected next state, plus the reward expected along the way. The value function $\mathcal{V}^{\pi^*}$ is the unique solution to its Poisson equation. Note that Equation (1.26) does not have a unique solution. There are two possible solutions to this problem: either take $\mathcal{V}(0) = 0$ for some reference state 0, or add the additional condition $\sum_{x \in \mathcal{X}} \pi_*(x)\mathcal{V}(x) = 0$. Only under the latter condition, $\mathcal{V}$ has the interpretation as the total expected difference in reward between starting in a state and starting in stationarity.

1.2.1.2 Optimal value functions

Solving a reinforcement task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we define an optimal policy in the following way. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $\mathcal{V}^{\pi}(x) \geq \mathcal{V}^{\pi'}(s)$ for all $x \in \mathcal{X}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π^*. They share the same state-value function, called the *optimal state-value function*, denoted by $\mathcal{V}^*$, and defined as

$$\mathcal{V}^{\pi^*}(x) = \max_{\pi} \mathcal{V}^{\pi}(x), \tag{1.27}$$

for all $x \in \mathcal{X}$. Optimal policies also share the same *optimal action-value function*, denoted by $\mathcal{Q}^*$, and defined as

$$\mathcal{Q}^{\pi^*}(x, a) = \max_{\pi} \mathcal{Q}^{\pi}(x, a), \tag{1.28}$$

for all $x \in \mathcal{X}$ and all $a \in \mathcal{A}(x)$. For the state-action pair (x,a) this function gives the expected return for taking action a in state x and thereafter following an optimal policy. Thus, we can write $\mathcal{Q}^*$ in terms of $\mathcal{V}^*$ as follows:

$$\mathcal{Q}^{\pi^*}(x, a) = \max_{a \in \mathcal{A}} \left\{ r(x, a) + \sum_{y \in \mathcal{X}} p(x, a, y)\mathcal{V}^{\pi^*}(y) \right\}. \tag{1.29}$$

Because $\mathcal{V}^*$ is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values, Equation (1.26). Because it is the optimal value function, however, $\mathcal{V}^*$'s consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for $\mathcal{V}^*$, or the *Bellman optimality equation*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$r(x, \pi^*(x)) + \sum_{y \in \mathcal{X}} p(x, \pi^*(x), y)\mathcal{V}^{\pi^*}(y) = \max_{a \in \mathcal{A}} \left\{ r(x, a) + \sum_{y \in \mathcal{X}} p(x, a, y)\mathcal{V}^{\pi}(y) \right\}. \tag{1.30}$$

At the same time, by the Poisson equation:

$$\mathcal{V}^{\pi^*}(x) + g^{\pi^*} = r(x, \pi^*(x)) + \sum_{y \in \mathcal{X}} p(x, \pi^*(x), y)\mathcal{V}^{\pi^*}(y). \tag{1.31}$$

Combining these two gives the Bellman optimality equation for $\mathcal{V}^*$:

$$\mathcal{V}^{\pi^*}(x) + g^{\pi^*} = \max_{a \in \mathcal{A}} \left\{ r(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{V}^{\pi^*}(y) \right\}. \tag{1.32}$$

The Bellman optimality equation for $\mathcal{Q}^*$ is

$$\mathcal{Q}^{\pi^*}(x,a) + g^{\pi^*} = \max_{a' \in \mathcal{A}} \left\{ r(x,a') + \sum_{y \in \mathcal{X}} p(x,a',y)\mathcal{Q}^{\pi}(y,a') \right\}. \tag{1.33}$$

For finite MDPs, the Bellman optimality equation (1.32) has a unique solution independent of the policy. The Bellman optimality equation is actually a system of equations, one for each state, so if there are N states, then there are N equation in N unknowns. If the dynamics of the environment are known ($r(x,a)$ and $p(x,a,y)$ for all x, y and a), then in principle one can solve this system of equations for $\mathcal{V}^*$ using any one of a variety of methods for solving systems of nonlinear equations. This can, for example, be done by using Dynamic Programming. One can solve a related set of equations for $\mathcal{Q}^*$.

Once one has solved $\mathcal{V}^*$, it is relatively easy to determine an optimal policy. For each state x, there will be one or more actions at which the maximum is attained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. You can think of this as a one-step search. If you have the optimal value function, $\mathcal{V}^*$, then the actions that appear best after a one-step search will be optimal actions. Another way of saying this is any policy that is *greedy* with respect to the optimal value function $\mathcal{V}^*$ is an optimal policy. By means of $\mathcal{V}^*$, the optimal expected long-term return is turned into a quantity that is locally and immediately available for each state. Hence, a one-step-ahead search yields the long-term optimal actions.

Having $\mathcal{Q}^*$ makes choosing optimal actions still easier. With $\mathcal{Q}^*$, the agent does not even have to do a one-step-ahead search: for any state x, it can simply find any action that maximizes $\mathcal{Q}^*(x,a)$. The action-value function effectively caches the results of all one-step-ahead searches. It provides the optimal expected long-term return as a value that is locally and immediately available for each state-action pair. Hence, at the cost of representing a function of state-action pairs, instead of just of states, the optimal action-value function, when Q^* is known, allows optimal actions to be selected without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solve the reinforcement learning problem. However, this solution is not always directly useful. It is akin to an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least three assumptions that have to be checked before we can use this method:

1. we accurately know the dynamics of the environment;
2. we have enough computational resources to complete the computation of the solution;
3. meet the Markov Property.

If the solution cannot be implemented exactly, there are many different decision-making methods which can be viewed as ways of approximately solving the Bellman optimality equation. More information about this methods and Reinforcement learning in general can be found in *Reinforcement Learning: An Introduction* written by R.S. Sutton and A.G. Barto [20].

Chapter 2

The Impact of Travel Information on Travel Behaviour

This chapter discusses several factors which affect travel and traffic behaviour on a freeway. The first section of this chapter, section 2.1, starts with an explanation of the structure of travel processes, the following section, section 2.2 describes the effects of weather on the travel conditions.

2.1 Structure of travel processes

In general when people think about structure of travel processes, probably the first thoughts will be about things to improve structure, to decrease traffic congestions. Traffic congestion occurs when a volume of traffic generates demand for space greater than the available road capacity. There are a number of specific circumstances which cause or aggravate congestion; most of them reduce the capacity of a road at a given point or over a certain stretch of freeway. Congestion can also be caused by weather, by the amount of slip roads or by people looking at a congestion on the other side of the road. In short, congestions are the result of interaction between traffic supply and demand. The demand consists of people traveling from A to B, the supply of the capacity of a road. We will define the capacity as the maximum amount of traffic on a certain part of the road per time unit. The relationship between traffic supply and demand is very complicated, there are roughly three explanations:

1. The demand of traffic is largely dependent of the traffic capacity, or the quality of supply. When the amount of traffic jams decreases, traffic demand increases.
2. The capacity of a traffic network decreases when traffic delays occur. The "drive-off capacity" of a lane, which indicates the maximum amount of traffic possible to drive off from standstill per time unit, decreases compared to the situation before the delay occurred. Moreover, the capacity of a road is not a constant parameter, but a changing outcome as a result of human behaviour under a number of factors.
3. Several complex network-effects appear, which cause delays on roads with a low traffic demand. For example, people change their route or choose another moment in time to travel.

For the description of traffic conditions in traffic networks we need models which can explain these complexities. Roughly, these models exist of three components:

- *A traffic-demand model*: What is the expected amount of traffic between point of origin and destination?
- *A route-choice model*: In which way are the available resources used by the traffic?

- *A flow model*: What are the consequences of the route-choosing on the network? Where and when do traffic jams arise?

All these models are influenced by the structure of travel processes, but as well by the effects of weather, as described in the next section.

2.2 Effects of weather on the travel conditions

In this section we focus on the effect of weather on the expected travel conditions. Because a model to describe traffic will be developed, the focus will be on the following question: How may weather influences be converted into concrete parameters? With these parameters we can discount for the effects of weather. But this is not as easy as we would like it to be, because every component named above has a potential influence:

1. First, the weather influences traffic demand. When weather is very bad, demand will increase on some roads but will decrease on others. Hanbali and Kuemmel investigated volume reductions due to winter storms, during their investigation they varied the intensity of snow fall, time of the day, day of the week, and roadway type [8]. Overall, they found that reductions ranged from 7% to 56% depending on the category of winter event. Hanbali and Kuemmel conclude that volume reductions increase with the total volume of snow, and volume decreases are smaller during the peak travel periods and on weekdays than during off-peak hours and on weekends [8]. Maze and collegues give an interesting overview of research on the effects of weather on as well traffic demand as drive-off capacity [21]. They conclude, depending on different types of traffic (e.g., work-trip, recreational trip) and the type of weather (including visibility, wind and precipitation), that demand decreases almost 5% by serious rain and they even found reductions ranging from 7% to 80% by moderate or serious snowfalls. They also found that during snowstorms, commercial vehicles made up a higher percentage of the traffic stream than their typical proportion during clear weather. This indicated that although motorists where diverting trips, commercial vehicle operators where much less likely to divert trips due to inclement conditions. In general, travel demand reductions are partially dependent on the type of trip: recreational travelers are more likely to postpone long-distance trips and commericial trips are least likely to be deferred. But the more adverse the weather, the smaller the demand of traffic.

2. Next to the impact of weather conditions on traffic demand Maze et al. [21] also investigated the impact on traffic safety. They found that during snow days, crashes increased and were highly correlated with visibility and wind speed. During low visibility conditions and high wind speeds, crash rate increased to 25 times the normal crash rate. What occurs during winter storms is, that there are fewer vehicles on the road and those vehicles that remain are much more likely to be in crash. As a result, the crash rate skyrockets. It is apparent that snowy weather greatly increases crash frequency and crash rate, while crash severity tends to be sightly lower.

3. Human choice-behaviour in terms of time to leave, route and way of travel depend on weather conditions as well. When travelers do make a trip, the weather will tempt someone in taking the car in stead of taking the bike for example. Not only that, the weather might influence the time to leave (you expect to travel longer) or the route one chooses (think of taking a less windy road). Unfortunately there is little literature available about explicit and quantitative research on the effect of weather on this kind of choice-behaviour.

4. Furthermore, weather has a great impact on driving itself (speed, tracking distance, acceleration) and therefore on road capacity. Weather changes behaviour and therefore the drive-off capacity. The effects of weather conditions on speed-flow-occupancy relationships and capacities for example, have been examined in several studies (e.g., Ibrahim and Hall, 1994 [9]; Brilon and Ponzlet, 1996 [4]). These studies mainly described local effects. However, changes in driving can have far-reaching consequences for the entire network. For example, as a result

> of heavy rainfall, the capacity on a part of the road can decrease with 15%. Which means that congestions occur more often and on unusual locations. These congestions need more time to solve and can result in problems on other areas. Therefore a reduction in capacity on a local part of the road caused by a heavy rainfall can have a great impact on the expected travel time on other parts of the network.

Not only bad weather influences travel conditions, sunny weather does as well. Take the traffic jams which occur on the roads to the beach on beautiful summer day for example.

In short, traffic flow is a function of traffic speed and traffic density, the amount of vehicles per km per lane. Freeway capacity is defined in Highway Capacity Manual 2000 as the "maximum flow rate that can be expected to be achieved repetitively at a single freeway location and at all locations with similar roadway, traffic and control conditions without breakdown" [1]. The capacity of a freeway segment (its maximum flow) is dependent on the speed of the traffic stream and the density. Under inclement conditions, drivers moderate their speed and increase the headway between vehicles; hence, weather impacts the capacity of a freeway segment. Because the weather conditions are hard to predict, best travel-time predictions are based on real-time weather conditions.

Chapter 3

Model development and data collection

In this chapter we will apply the theory of Markov Decision Processes on the travel time prediction model in order to develop an appropriate method to predict real-time travel times. Section 3.1 presents a detailed description of the data collection. Sections 3.2 and 3.3 describe the application of the developed Markov decision processes using field data for two different solution methods: Dynamic Programming and Temporal Difference learning.

3.1 Data Collection

Model development requires the collecting of appropriate data. For this research we used a dataset about the A10, the motorway around the city center of Amsterdam in the Netherlands. The total length of the A10 is 32.4 kilometers. Unfortunately, the dataset, collected by "Rijkswaterstaat", part of the Dutch Ministry of Transport, contains only vehicle specific information over a trajectory of 5605 meters. Because we would compare links with congestion to links without congestion, we made the assumption that the A10 has a length of only 5605 meters. A graphical representation of the motorway A10 divided into links can be found in Figure 3.1. The travel time data were collected on a period from 17 June 2005 until 12 July 2005.

The dataset contains 96 files of which each filename contains the hectometers pole number, the direction (to the right (clockwise) or left) and a part number. For example, we take the file: "$21260L_1.log$". The five numbers in the front contain the hectometers pole number, hence the roadside-detector station is located at hectometers pole number 21260. The vehicle specific information in this file is about vehicles that are driving to the left (anti-clockwise) and this is the first part of the data for this detector station. In total we use data from 24 different roadside-detector stations (12 to the right and 12 to the left). The data for each detector station is split over 4 .log-files, which makes 96 files in total.

The text below shows an example of the vehicle specific information logged in the roadside-detector station:

```
6/17/2005\14:09:43 Det: 3 Alarm     t12:0 t13:0 t24:0 t04:0 speed: 0 km/h
6/17/2005\14:09:43 Det: 4 Normal    t12:102 t13:-1 t24:136 t04:179 speed: 88 km/h
6/17/2005\14:09:44 Det: 4 Normal    t12:89 t13:-1 t24:181 t04:259 speed: 101 km/h
6/17/2005\14:09:53 Det: 2 Normal    t12:90 t13:-1 t24:207 t04:27 speed: 100 km/h
6/17/2005\14:09:54 Det: 4 Normal    t12:99 t13:-1 t24:211 t04:36 speed: 90 km/h
6/17/2005\14:09:57 Det: 2 Normal    t12:75 t13:-1 t24:148 t04:363 speed: 120 km/h
6/17/2005\14:09:58 Det: 4 Normal    t12:101 t13:-1 t24:219 t04:408 speed: 89 km/h
6/17/2005\14:10:00 Det: 4 Normal    t12:100 t13:-1 t24:213 t04:28 speed: 90 km/h
6/17/2005\14:10:01 Det: 4 Normal    t12:96 t13:-1 t24:204 t04:125 speed: 93 km/h
6/17/2005\14:10:02 Det: 2 Normal    t12:90 t13:-1 t24:192 t04:274 speed: 100 km/h
6/17/2005\14:10:03 Det: 1 Normal    t12:78 t13:-1 t24:169 t04:276 speed: 115 km/h
6/17/2005\14:10:04 Det: 0 Occupancy t12:-1 t13:-1 t24:211 t04:-1 speed: 0 km/h
```

Each line starts with a notification of the date(m/dd/yyyy): month/day/year; and time (hh:mm:ss): hour:minute:second. After "Det:" we find the lane number, where we start numbering at the central reservation. Hence, when we have a 3-lane motorway, lane 0 is the left one, lane 1 is the middle one and lane 2 the right one. Then we see a text which is called 'Message type', this text gives us different warnings, like "Alarm, Normal and Occupancy". Because we would like to exclude unreliable data, we only used data with the type "Normal". The message type is followed by several times (in milliseconds) between loops in the road surface and the calculated vehicle speed in kilometers per hour (km/h), respectively.

Figure 3.2 shows the timing of "Normal" vehicles. With help of this figure we can explain the several t-values. The travel time, t12, is the time a vehicle needs to drive from loop 1 to loop 2, calculated by $t_2 - t_1$. The occupancy of a loop, t13 or t24, can be calculated by $t_3 - t_1$ or $t_4 - t_2$. Because we would not use the occupancy of a loop we do not need to specify one. The calculated speed is based on the travel time, t12, over a distance of 2.5 meter as can be seen in the figure.Take for example record 2: The time between arriving at the first and second loop is given by t12, which equals 102 milliseconds, the distance covered is 2.5 meters, hence the calculated speed results in 24.51 m/s (88.2 km/hour).

3.2 Application of a MDP using Dynamic Programming

Section 1.2 developed a stochastic process called a Markov decision process. A stochastic process is a random function which varies in time and/or space. Its future values can be predicted with a certain amount of probability. This means that the process does not behave in a completely unpredictable manner but is governed by a random mechanism. Because a Markov decision process satisfies the Markov property, the value of the process at time t does only depend on the value at time t-1. For a transportation system, if the demand does not fluctuate much for a given time period, the system states would not depend on time but could be considered a random mechanism. In these cases, a stochastic process analysis can be applied. A stochastic process $X = \{X_t, t \in T\}$ is a collection of random variables. That is, for each t in the index set T, X_t is a random variable. Usually, t is interpret as time and X_t is the state of the process at time t. If the index set T is countable, the process is called a discrete-time stochastic process, while if T is a non-countable variable, it is called a continuous-time stochastic process. In other words, under the discrete time, $T \in \{0, 1, 2, \ldots\}$ the change of state occurs at the end of a time unit, while under continuous time, $T \geq 0$, the change of state occurs at any point in time. This study considers how the system states change at every time unit, so a discrete-time stochastic process is employed.

To apply a stochastic process for estimating travel time, relatively uniform demand time periods are used in the analysis. These uniform demand distributions depend on the type of day (e.g., weekday, weekend, holiday) and the time on the day (e.g., peak-hours, regular hours). The following tasks were undertaken:

1. Define system states and actions considering the type of congestion for each link, think of junctions, bridges, level crossings, et cetera.
2. Estimate travel time for each link for each type of congestion using field data.
3. Determining time periods to consider time of day and daily variation of travel time.
4. Estimate the transition matrices considering how the system changes from one interval to another for each time period, considering transitions between non-congested and each type of congested flow. Use Equation (1.17).
5. Calculate the expected value of the reward (1.18).
6. Use these rewards to obtain how good it is to perform a given action in a given state, that is find the unique solution of the optimal value functions (1.32) and (1.33).
7. Determine the optimal policy π^*.

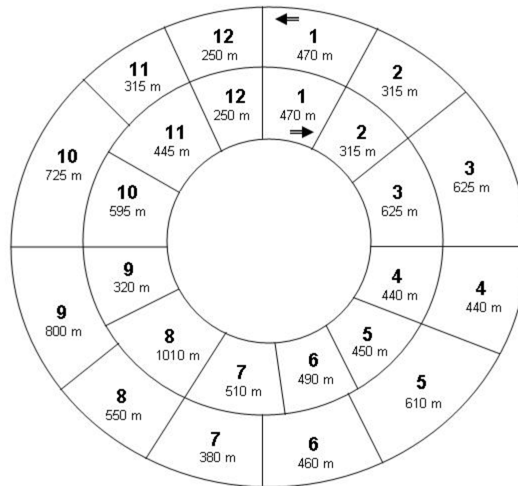

Figure 3.1. Graphical overview of the motorway A10 divided into links.

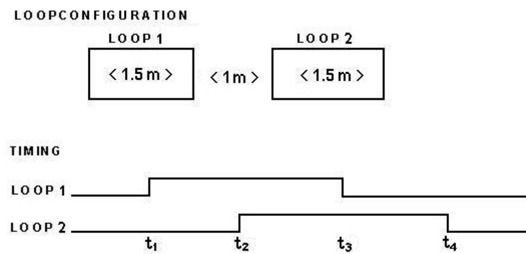

Figure 3.2. Graphical overview of loop configuration and timing.

As discussed in section 1.2, we can easily obtain optimal policies ones we have found the optimal value functions, V^* or Q^*, which satisfy the bellman optimality equations:

$$V^{\pi^*}(x) + g^{\pi^*} = \max_{a \in \mathcal{A}} \left\{ r(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y) V^{\pi}(y) \right\}. \tag{3.1}$$

or

$$Q^{\pi^*}(x,a) + g^{\pi^*} = \max_{a' \in \mathcal{A}} \left\{ r(x,a') + \sum_{y \in \mathcal{X}} p(x,a',y) Q^{\pi}(y,a') \right\}. \tag{3.2}$$

for all $x \in \mathcal{X}$ and $a \in \mathcal{A}$. As we shall see, Dynamic Programming (DP) algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.

The following sections elaborate the steps mentioned above. The solution to a Markov Decision Process can be expressed as a policy π, a function from states to actions. Note that once a Markov decision process is combined with a policy in this way, this fixes the action for each state and the resulting combination behaves like a Markov chain. Therefore, step 1 to 5 are similar to the steps taken in previous research about travel time estimation using discrete time Markov chains (J. Yeon et al., 2008 [10]).

3.2.1 Definition of states and variables

In the introduction of this section we already defined a stochastic process, $X = \{X_t, t \in T\}$. In a stochastic process, a system has to be defined as a state which is analyzed using a measurable characteristic. For the purposes of this research, the system is defined as a freeway route, and the system state variable is X_t at time t ($t = 0, 1, 2, \ldots$), which describes how the states of a given freeway route change every time unit. X_t can be described as a set of $x_i(t)$, where $x_i(t)$ is a link state variable of link i at time t. A link is defined as the segment between detectors on the freeway, and a route is composed of several links. We have chosen the congestion type intervals based on the maximum speed on each link i the state variable $x_i(t)$ can take the following values:

$$x_i(t) = \left\{ \begin{array}{ll} 0 & \text{if } \overline{s_i}(t) \text{ is higher than the maximum speed} \\ 1 & \text{if } \overline{s_i}(t) \text{ is between 75\% and 100\% of the maximum speed} \\ 2 & \text{if } \overline{s_i}(t) \text{ is between 50\% and 75\% of the maximum speed} \\ 3 & \text{if } \overline{s_i}(t) \text{ is between 25\% and 50\% of the maximum speed} \\ 4 & \text{if } \overline{s_i}(t) \text{ is lower than 25\% of the maximum speed} \end{array} \right\}, \tag{3.3}$$

where $\overline{s_i}(t)$ is the average speed of the vehicles in link i at time t.

The system state variable and link state variable can be expressed as follows:

$$X(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_k(t) \end{bmatrix}, \tag{3.4}$$

where k is the total number of links. Thus for a freeway route with 5 links, a system state variable (i.e., X_t at time t) can be [0,3,0,0,4], where links 2 and 5 are congested with congestion type 3 and 4, respectively.

3.2.2 Link travel time estimation

The route travel time equals the sum of travel times on each link. To compute this, the link travel time has to be estimated for types of traffic congestion conditions. Table 3.1 shows the travel time notation for each link. The travel time under all conditions is considered as constant, contrary to previous research (Van Lint and Van Zuylen, 2005 [12]) where the travel time under congested conditions is estimated as a function of the flow rates and consequently of demands. Difference can be declared by the state definition. We choose to define several types of congestion, based on average speed in a time period, where in previous research each link could only be congested or non-congested. Additionally, while in previous research time periods had to be determined, we do not have to account for time of day and daily variations because our link travel time estimation does not depend on demand. This is a huge advantage because demand does not only vary over day, week and month, but is as well influenced unpredictable factors like the effects of weather as described in Chapter 2.

3.2.3 Transition Matrix

This section discusses the development of the transition matrix, which considers whether each link is congested or not. The transition is a change of state and the one-step transition matrix (e.g., $n = 1$, where n is the number of steps) shows the changing rate from state i to state j as shown below:

$$P^1 = \begin{pmatrix} p_{11} & p_{12} & p_{13} & \cdots & p_{1m} \\ p_{21} & p_{22} & p_{23} & \cdots & p_{2m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{m1} & p_{m2} & p_{m3} & \cdots & p_{mm} \end{pmatrix}, \tag{3.5}$$

where a state is defined in Section 3.2.1, and p_{ij} represents the transition rate from state i to state j, which can be expressed in conditional probability as follows:

$$p_{ij} = \mathcal{P}\left\{X_{t+1} = j | X_t = i\right\} \text{ for } i, j = 1, 2, \ldots, m, \text{ and } t = 1, 2, 3, \ldots, \tag{3.6}$$

where m is the total number of possible states.

Or in other words, given any state and action, s and a, the probabilities of each possible next state, s', is (1.17):

$$\mathcal{P}^a_{xy} = Pr\left\{X_{t+1} = y | X_t = x, a_t = a\right\} = p(x, a, y) \tag{3.7}$$

For example, p_{12} in the transition matrix is computed as the total number of transitions from state 1 to state 2 divided by the total number of transitions from state 1 to all other states including state 1. The system state variable at time t, X_t, includes the link state variable $x_i(t)$, thus, each state (i.e., $i, \ldots, j, \ldots, m$) of the transition matrix can be denoted as $1, 2, \ldots, m$. Thus, the number of total possible states (m) is equal to 5^k, where k the number of links of the system and there are 5 congestion types, l. As the number of steps (t) (3.6) increases, the system becomes more stable:

Travel notation by link					
	Link information congestion type				
Link number	0	1	2	3	4
1	NT_{10}	NT_{11}	NT_{12}	NT_{13}	NT_{14}
2	NT_{20}	NT_{21}	NT_{22}	NT_{23}	NT_{24}
...	...	...	...	...	...
k	NT_{k0}	NT_{k1}	NT_{k2}	NT_{k3}	NT_{k4}

Table 3.1. k: total number of links; i: link number; NT_{il}: the travel time of link i (for $i = 1, 2, \cdots, k$) when the type of congestion on that link is l ($l = 0, \ldots, 4$).

$$\begin{aligned}\pi_{t+1} &= \pi_t P, \\ \pi_{t+1} &= \pi_0 P^t,\end{aligned} \tag{3.8}$$

if $t \to \infty$, then $\pi_{t+1} = \pi_t$.

The probabilities when to which the system converges are called *steady-state probabilities*. These probabilities equal the probabilities that a system will eventually arrive at state j whatever the initial state is. Usually the initial state becomes less and less relevant, but this is not always the case.

The solution to a Markov Decision Process can be expressed as a policy π, a function from states to actions. Note that once a Markov decision process is combined with a policy in this way, this fixes the action for each state and the resulting combination behaves like a Markov chain. The Markov chain is *unichain*, which means that all states communicate with each other, and *ergodic*, a process will finally return to the starting state within a certain time period, where there exists a unique steady-state probability for all j. The steady-state probability for state j, Π_j, is defined as follows:

$$\Pi_j = \lim_{n\to\infty} \mathcal{P}\left\{X_{(t=n)} = j | X_{(t=0)} = i\right\} = \lim_{n\to\infty} \mathcal{P}\left\{X_{(t=n)} = j\right\} \text{ for } i = 1, \dots, m. \tag{3.9}$$

these steady-state probabilities show the probabilities that the system eventually will be at each defined state, and they can be used to calculate the expected travel time of a system for the time period being analyzed.

3.2.4 The Reward

The final definition task is to estimate route travel time using the output from the previous four tasks. Since the steady-state probabilities for all j can be obtained as described above, the travel time of each link under each type of congestion can be estimated next. Then, the expected route travel time can be estimated as follows:

$$\overline{T} = \sum_{j=1}^{m} \sum_{i=1}^{k} \sum_{l=0}^{4} \Pi_j \left\{NT_{il} \times \mathbb{1}_{x_i(t)=l}\right\}, \tag{3.10}$$

where Π_j is the steady-state probability for state j, $x_i(t)$ is the state variable of link i at time t, NT_{il} is the travel time with congestion type l of link i, m is the total number of possible states and k is the total amount of links.

Multiply each steady-state probability for a given state by the travel times when the system is in that state. For instance, if the system is in a state where all the links contain no congestion, then the probability is multiplied by the state of a link times the sum of the travel times with congestion type 0 for each link. Likewise, if the system is in a state that contains one link with congestion of type j, the probability multiplied by the state of a link times the sum of the non-congested travel times for all the non-congested links (congestion type 0) is added to the travel time of congestion type j. Repeat for each steady-state probability. The result is the expected travel time for the study route and for the given time period.

To find a solution of the MDP developed, we would like to maximize the reward. In our case this is equal to minimize the total route travel time. Therefore the following reward function will be used:

$$r(x, a, y) = E\left\{r_{t+1} | X_t = x, a_t = a, X_{t+1} = y\right\} = -\overline{T} \tag{3.11}$$

Since the problem is clearly defined for our situation now, we can start doing the real work.

3.2.5 The Optimal Value Functions

3.2.5.1 Policy Evaluation

First we consider how to compute the state-value function $\mathcal{V}^\pi$ for an arbitrary policy π. This is called *policy evaluation* in DP literature. We also refer to it as the *prediction problem*. Recall from Chapter 1 that, for all $x \in \mathcal{X}$,

$$\mathcal{V}^\pi(x) + g = r(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{V}^\pi(y). \tag{3.12}$$

where the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of $\mathcal{V}^\pi$ are guaranteed as long as either $\mathcal{V}(0) = 0$ for some reference state 0 or the condition $\sum_{x \in \mathcal{X}} \pi_*(x)\mathcal{V}(x) = 0$ is added.

If the environment's dynamics are completely known, then (3.12) is a system of $|\mathcal{X}|$ simultaneous linear equation in $|\mathcal{X}|$ unknowns (the $\mathcal{V}^\pi(x), x \in \mathcal{X}$). In principle this solution is a straightforward, if tedious, computation. For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2, \ldots$, each mapping $\mathcal{X}$ to $\mathbb{R}$. The initial approximation, $\mathcal{V}_0$, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for $\mathcal{V}^\pi$ (1.26) as an update rule:

$$\mathcal{V}_{T+1}(x) = r(x) + \sum_{y \in \mathcal{X}} p(x,y)\mathcal{V}_T(y). \tag{3.13}$$

for all $x \in \mathcal{X}$. If $T \to \infty$, then is $\mathcal{V}_T = \mathcal{V}^\pi$ a fixed point for this update rule because the Bellman equation for $\mathcal{V}^\pi$ assures us of equality in this case. Indeed, the sequence $\{\mathcal{V}_T\}$ can be shown in general to converge to $\mathcal{V}^\pi$ as $T \to \infty$ under the same conditions that guarantee the existence of $\mathcal{V}^\pi$. This algorithm is called *iterative policy evaluation*. To produce each successive approximation, $\mathcal{V}_{T+1}$ from $\mathcal{V}_T$, iterative policy evaluation applies the same operation to each state x: it replaces the old value of x with a new value obtained from the old values of the successor rates of x, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation a *full backup*. Each iteration of iterative policy evaluation *backs up* the value of every state once to produce the new approximate value function $\mathcal{V}_{T+1}$.

To write a sequential computer program to implement iterative policy evaluation, as given by Equation (3.13), you would have to use two arrays, one for the old values, $\mathcal{V}_T(x)$, and one for the new values, $\mathcal{V}_{T+1}(x)$. This way, the new values can be computed one by one from the old values without the old values being changed. Of course it is easier to use one array and update the values "in place", that is, with each new backed-up value immediately overwriting the old one. Then, depending on the order in which the states are backed-up, sometimes new values are used instead of old ones on the right-hand side of Equation (3.13). This slightly different algorithm also converges to $\mathcal{V}^\pi$; in fact, it usually converges faster than the two-array version, as you might expect, since it uses new data as soon as they are available. We think of the backups as being done in a *sweep* through the state space. For the in-place algorithm, the order in which states are backed-up during the sweep has a significant influence on the rate of convergence. We usually have the in-place version in mind when we think of DP algorithms. Another implementation point concerns the termination of the algorithm. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. A typical stopping condition for iterative policy evaluation is to test the quantity $\max_{x \in \mathcal{X}} |\mathcal{V}_{T+1}(x)| - \min_{x \in \mathcal{X}} |\mathcal{V}_T(x)|$ after each sweep and stop when it is sufficiently small. Figure 3.3 gives a complete algorithm for iterative policy evaluation with this stopping condition.

3.2.5.2 Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function $\mathcal{V}^\pi$ for an arbitrary deterministic policy π. For some state x we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(x)$. We know how good it is to follow the current policy from x - that is $\mathcal{V}^\pi$ - but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting action a in state x and thereafter following the existing policy, π. The value of this way of behaving is

$$\mathcal{Q}^\pi(x,a) = r(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{V}^\pi(y). \tag{3.14}$$

The key criterion is whether this is greater than or less than $\mathcal{V}^\pi$. If it is greater - that is, if it is better to select a once in x and thereafter follow π than it would be to follow π all the time - then one would expect it to be better still to select a every time x is encountered, and that the new policy would in fact be a better one overall. That this is true is a special case of a general result called the *policy improvement theorem*.

Theorem 1. *Let π and π' be any pair of deterministic policies such that for all $x \in \mathcal{X}$,*

$$\mathcal{Q}^\pi\left(x, \pi'(x)\right) \geq \mathcal{V}^\pi(x). \tag{3.15}$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $x \in \mathcal{X}$:

$$\mathcal{V}^{\pi'}(x) \geq \mathcal{V}^\pi(x). \tag{3.16}$$

Moreover, if there is strict inequality of Equation (3.15) at any state, then there must be strict inequality of Equation (3.16) at one or more states. This result applies in particular to the two policies that we considered in the previous paragraph, an original deterministic policy, π', that is identical to π except that $\pi'(x) = a \neq \pi(x)$. Obviously, $\mathcal{Q}^\pi\left(x, \pi'(x)\right) > \mathcal{V}^\pi(x)$ holds at all states other than x. Thus, if $\mathcal{Q}^\pi(x,a) > \mathcal{V}^\pi(x)$, then the changed policy is indeed better than π.

The idea behind the proof of the policy improvement theorem is easy to understand. Starting from Equation (3.15), we keep expanding the $\mathcal{Q}^\pi$ side using Equation (3.14) and reapplying Equation (3.15) until we get $\mathcal{V}^{\pi'}$:

$$\begin{aligned}
\mathcal{V}^\pi(x) &\leq \mathcal{Q}^\pi(x, \pi'(x)) \\
&= r_t(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{V}^\pi(y) \\
&\leq r_t(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{Q}^\pi(y, \pi'(y)) \\
&= r_t(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)[r_{t+1}(x,a) + \sum_{y' \in \mathcal{X}} p(x,a,y')\mathcal{V}^\pi(y')] \\
&= r_t(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)r_{t+1}(x,a) + \sum_{y' \in \mathcal{X}} p(x,a,y')^2\mathcal{V}^\pi(y') \\
&\leq r_t(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)r_{t+1}(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)^2 r_{t+2}(x,a) + \sum_{y'' \in \mathcal{X}} p(x,a,y'')^3\mathcal{V}^\pi(y'') \\
&\vdots \\
&\leq r_t(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)r_{t+1}(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)^2 r_{t+2}(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)^3 r_{t+3} + \sum_{y \in \mathcal{X}} p(x,a,y)^4 r_{t+4} + \dots \\
&= \mathcal{V}^{\pi'}(x)
\end{aligned}$$

So far we have seen now, given a policy and its value function, we can easily evaluate a change in the policy at a single state to a particular action. It is a natural extension to consider changes at *all* states and to *all* possible actions, selecting at each state the action that appears best according to $\mathcal{Q}^\pi(x,a)$. In other words, to consider the new *greedy* policy, π', given by

$$\begin{aligned} \pi'(x,a) &= \arg\max_{a\in\mathcal{A}} \mathcal{Q}^\pi(x,a) \\ &= \arg\max_{a\in\mathcal{A}} \left\{ r(x,a) + \sum_{y\in\mathcal{X}} p(x,a,y)\mathcal{V}^{\pi^*}(y) \right\}. \end{aligned} \tag{3.17}$$

where $\arg\max_a$ denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). The greedy policy takes the action that looks best in the short term - after one step of lookahead - according to $\mathcal{V}^\pi$. By construction the greedy policy meets the conditions of the policy improvement theorem, so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves on an original policy, by making it greedy or nearly greedy with respect to the value function of the original policy, is called *policy improvement*.

Suppose the new greedy policy, π', is as good as, but not better than, the old policy π. Then $\mathcal{V}^\pi = \mathcal{V}^{\pi'}$, and from Equation (3.17) it follows that for all $x \in \mathcal{X}$:

$$\mathcal{V}^{\pi'}(x) = \max_{a\in\mathcal{A}} \left\{ r(x,a) + \sum_{y\in\mathcal{X}} p(x,a,y)\mathcal{V}^{\pi^*}(y) \right\}. \tag{3.18}$$

But this is the same as the Bellman optimality equation (3.1), and therefore, $\mathcal{V}^{\pi'}$ must be $\mathcal{V}^*$, and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

So far in this section we have considered the special case of deterministic policies. In the general case, a stochastic policy π specifies probabilities, $\pi_*(x,a)$, for taking each action, a, in each state, x. We will not go through the details, but in fact all the ideas of this section extend easily to stochastic policies. In particular, the policy improvement theorem carries through for the stochastic case as stated, under the natural definition:

$$\mathcal{Q}^\pi\left(x,\pi'(x)\right) = \sum_a \pi_*(x,a)\mathcal{Q}^\pi(x,a). \tag{3.19}$$

In addition, if there are ties in policy improvement steps such as in Equation (3.17) - that is, if there are several actions at which the maximum is achieved - then in the stochastic case we do not need to select a single action from among them. Instead, each maximizing action can be given a portion of the probability of being selected in the new greedy policy. Any apportioning scheme is allowed as long as all sub maximal actions are given zero probability.

3.2.5.3 Policy Iteration

Once a policy, π, has been improved using $\mathcal{V}^\pi$ to yield a better policy, π', we can then compute $\mathcal{V}^{\pi'}$ and improve it again to yield an even better policy, π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} \mathcal{V}^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} \mathcal{V}^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi^* \xrightarrow{E} \mathcal{V}^*, \tag{3.20}$$

where $\xrightarrow{E}$ denotes a policy *evaluation* and $\xrightarrow{I}$ denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in Figure 3.4. Note that each policy evaluation, itself an iterative computation, is started with the value function of the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

3.2.5.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to $\mathcal{V}^\pi$ occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? In fact, the policy evaluation step of policy improvement can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one backup of each state). This algorithm is called *value iteration*. It can be written as a particularly simple backup operation that combines the policy improvement and truncated policy evaluation steps:

$$\mathcal{V}_{T+1}(x) = \max_{a \in \mathcal{A}} \left\{ r(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{V}_T(y) \right\}, \tag{3.21}$$

for all $x \in \mathcal{X}$. For arbitrary $\mathcal{V}_0$, the sequence $\{\mathcal{V}_T\}$ can be shown to converge to $\mathcal{V}^*$ under the same conditions that guarantee the existence of $\mathcal{V}^*$.

Another way of understanding value iteration is by reference to the Bellman optimality equation (3.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration backup is identical to the policy evaluation backup, Equation (3.13), except that it requires the maximum to be taken over all actions.

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to $\mathcal{V}^*$. In practice, we stop once the value function changes by only a small amount in a sweep. Figure 3.5 gives a complete iteration algorithm with this kind of termination condition.

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation backups and some of which use value iteration backups. Since the *max*-operation in Equation (3.21) is the only difference between these backups, this just means that the *max*-operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for finite MDPs.

Dynamic Programming (DP) is not the only method for estimating value functions and discovering optimal policies. To obtain the same results you can use Temporal Difference (TD) learning and Monte Carlo methods as well. Unlike DP, these methods do not assume complete knowledge of the environment. TD learning will be explained in the next section. Monte Carlo methods require only *experience* - sample sequences of states, actions, and rewards from on-line or simulated interaction with an environment. Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that are required by DP methods. In surprisingly many cases it is easy to generate experience sampled

Let $|\mathcal{X}| = N$, $Z(x) \subset \{y|p(x,y) > 0\}$ and ϵ some small number (e.g., $\epsilon = 10^{-6}$).

Vector $\mathcal{V}[1,\ldots,N]$, $\mathcal{V}'[1,\ldots,N]$
Float *min, max*

Input π, the policy to be evaluated.
Initialize $\mathcal{V}(x) = 0$, for all $x \in \mathcal{X}$
Repeat
    $\mathcal{V}' \leftarrow \mathcal{V}$
    For each $x \in \mathcal{X}$
        $\mathcal{V}(x) \leftarrow r(x)$
        For each $y \in Z$
            $\mathcal{V}(x) \leftarrow \mathcal{V}(x) + p(x,y)\mathcal{V}'(y)$
    $max \leftarrow 10^{-10}$
    $min \leftarrow 10^{10}$
    For each $x \in \mathcal{X}$
        if $(\mathcal{V}(x) - \mathcal{V}'(x) < \text{min})\,\text{min} \leftarrow \mathcal{V}(x) - \mathcal{V}'(x)$
        if $(\mathcal{V}(x) - \mathcal{V}'(x) > \text{max})\,\text{max} \leftarrow \mathcal{V}(x) - \mathcal{V}'(x)$
until $(\text{max} - \text{min} < \epsilon)$
Output $\mathcal{V} \approx \mathcal{V}^{\pi}$

Figure 3.3. Complete algorithm for iterative policy evaluation.

1. Initialization
   $\mathcal{V}(x) \in \mathbb{R}$ and $\pi(x) \in \mathcal{A}(x)$ arbitrarily for all $x \in \mathcal{X}$

2. Policy Evaluation
   Repeat
       $\mathcal{V}' \leftarrow \mathcal{V}$
       For each $x \in \mathcal{X}$
           $\mathcal{V}(x) \leftarrow r(x)$
           For each $y \in Z$
               $\mathcal{V}(x) \leftarrow \mathcal{V}(x) + p(x,y)\mathcal{V}'(y)$
       $max \leftarrow 10^{-10}$
       $min \leftarrow 10^{10}$
       For each $x \in \mathcal{X}$
           if $(\mathcal{V}(x) - \mathcal{V}'(x) < \text{min})\,\text{min} \leftarrow \mathcal{V}(x) - \mathcal{V}'(x)$
           if $(\mathcal{V}(x) - \mathcal{V}'(x) > \text{max})\,\text{max} \leftarrow \mathcal{V}(x) - \mathcal{V}'(x)$
   until $(\text{max} - \text{min} < \epsilon)$

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $x \in \mathcal{X}$:
       $b \leftarrow \pi(x)$
       $\pi(x) \leftarrow \arg\max_{a \in \mathcal{A}} \left\{ r(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{V}^{\pi^*}(y) \right\}$
       If $b \neq \pi(x)$, then *policy-stable* $\leftarrow$ false
   If *policy-stable*, then stop; else go to 2

Figure 3.4. Complete algorithm for policy iteration

Initialize $\mathcal{V}$ arbitrarily, e.g., $\mathcal{V}(x) = 0$, for all $x \in \mathcal{X}$

Repeat
  $\mathcal{V}' \leftarrow \mathcal{V}$
  For each $x \in \mathcal{X}$
    $\mathcal{V}(x) \leftarrow \max_{a \in \mathcal{A}} \{r(x,a) + p(x,a,y)\mathcal{V}'(y)\}$
  $max \leftarrow 10^{-10}$
  $min \leftarrow 10^{10}$
  For each $x \in \mathcal{X}$
    if $(V(x) - V'(x) < \min) \min \leftarrow \mathcal{V}(x) - \mathcal{V}'(x)$
    if $(V(x) - V'(x) > \max) \max \leftarrow \mathcal{V}(x) - \mathcal{V}'(x)$
until $(\max - \min < \epsilon)$

Output a deterministic policy, π such that
  $\pi(x) = \arg\max_{a \in \mathcal{A}} \left\{ r(x,a) + \sum_{y \in \mathcal{X}} p(x,a,y)\mathcal{V}_T(y) \right\}$.

Figure 3.5. Complete algorithm for value iteration

according to the desired probability distributions, but infeasible to obtain the distributions in explicit form. In short, Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. Despite the difference between Monte Carlo and DP methods, the most important ideas carry over from the DP to the Monte Carlo case. Not only do they compute the same value functions, but they interact to attain optimality in essentially the same way.

3.3 Application of a MDP using Temporal-Difference Learning

Another method which can be used for estimating value functions and discovering optimal policies is *Temporal-Difference* (TD) learning. TD learning is a combination of Monte Carlo and DP ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics: given some experience following a policy π both methods update their estimate $\mathcal{V}$ or $\mathcal{V}^\pi$. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The backup of simple TD methods is based on just the one next reward, using the value of the state one step later as an approximation for the remaining rewards. At time $t+1$ they immediately form a target and make a useful update using the observed reward r_{t+1} and the estimate $\mathcal{V}(X_{t+1})$. The simplest TD method, known as TD(0) is:

$$\mathcal{V}(X_t) \leftarrow \mathcal{V}(X_t) + \alpha \left[r_{t+1} + \mathcal{V}(X_{t+1}) - \mathcal{V}(X_t) \right]. \tag{3.22}$$

where $\mathcal{V}(X_t)$ is the remaining travel time to link N and $X_t \in 1, 2, \ldots, N$. Hence, $\mathcal{V}(N) = 0$. Because the TD method bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. Figure 3.6 specifies TD(0) completely in procedural form.

TD learning have some advantages over Monte Carlo and DP methods. First, they do not require a model of the environment, of its reward and next-state probability distributions. The next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an on-line, fully incremental fashion. With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step. Surprisingly often this turns out to be a critical consideration. Some application have very long episodes, so that delaying all learning until an episode's end is too slow. Other application are continuing tasks and have no episodes at all. Finally, in practice, TD methods have usually been found to converge faster to $\mathcal{V}^\pi$ than constant α-Monte Carlo methods.

Thus, the backup of simple TD methods is based on just the one next reward, using the value of the state one step later as a approximation for the remaining rewards. One kind of intermediate method, then, would perform a backup based on a intermediate number of rewards, more than one, but less than all of them until termination. For example, a two-step backup would be based on the first two rewards and the estimated value of the state two steps later. Similarly, we could have three-step backups, four-step backups, and so on. The methods that use n-step backups are still TD methods because they still change an earlier estimate based on how it differs from a later estimate. Now the later estimate is not one step later, but n steps later. This kind of methods are called n-step TD methods.

3.3.1 The Forward View of TD(λ)

Backups can be done not just toward any n-step return, but toward any *average* of n-step returns. For example, a backup can be done toward a return that is half of a two-step return and half of a four-step return: $\overline{R_t} = \frac{1}{2}R_t^{(2)} + \frac{1}{2}R_t^{(4)}$, where $R_t^{(n)} = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_{t+n} + \mathcal{V}_t(X_{t+n})$. Any set of return can be averaged in this way, even an infinite set, as long as the weights on the component returns are positive and sum up to 1. The overall return possesses an error reduction property which can be used to construct backups with guaranteed convergence properties. The TD(λ) algorithm can be understood as one particular way of averaging n-step backups, each weighted proportional to λ^{n-1}, where $0 \leq \lambda \leq 1$. A normalization factor of $1-\lambda$ ensures that the weights sum to 1. The resulting backup is toward a return, called the λ-*return*, defined by

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}. \tag{3.23}$$

Figure 3.7 illustrates this weighting sequence. The one-step return is given the largest weight, $1-\lambda$; the two-step return is given the next largest weight, $(1-\lambda)\lambda$; the three-step return is given the weight $(1-\lambda)\lambda^2$; and so on. The weight fades by λ with each additional step. After a terminal state has been reached, all subsequent n-step returns are equal to R_t. If we want, we can separate these terms from the main sum, yielding

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t. \tag{3.24}$$

This equation makes it clearer what happens when $\lambda = 1$. In this case the summation term goes to zero, and the remaining term reduces to the conventional return, R_t. Thus, for $\lambda = 1$, backing up according to the λ-return is the same as the constant-α Monte Carlo method. On the other hand, for $\lambda = 0$, backing up according to the λ-return is the same as the one-step TD method, TD(0). We define the λ-*return algorithm* as the algorithm that performs backups using the λ-return. On each step, t, it computes an increment, $\Delta\mathcal{V}_t(X_t)$, to the value of the state occurring on that step:

$$\Delta\mathcal{V}_t(X_t) = \alpha[R_t^\lambda - \mathcal{V}_t(X_t)]. \tag{3.25}$$

Initialize $\mathcal{V}(x)$ arbitrarily, π to the policy to be evaluated

Repeat (for each episode)
    Initialize x
    Repeat (for each step of episode):
        $a \leftarrow$ action given by policy π for state x
        Take action a; observe reward, r, and next state, y
        $\mathcal{V}(x) \leftarrow \mathcal{V}(x) + \alpha[r + \mathcal{V}(y) - \mathcal{V}(x)]$
        $x \leftarrow y$
    until x is terminal

Figure 3.6. Tabular TD(0) for estimating $\mathcal{V}^\pi$.

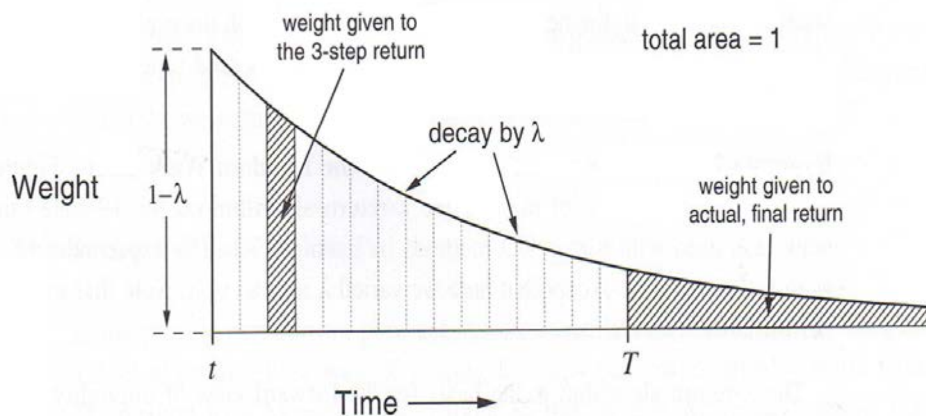

Figure 3.7. Weighting given in the λ-return to each of the n-step returns.

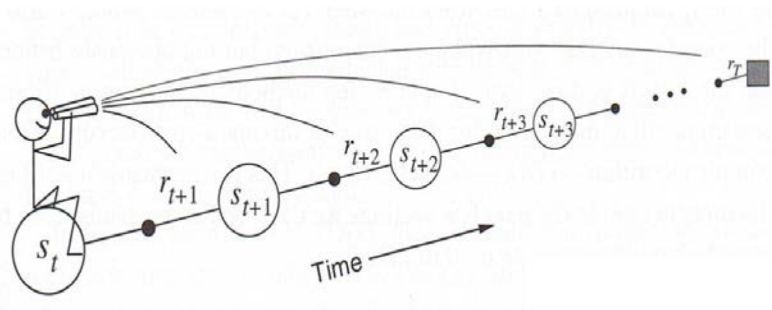

Figure 3.8. The forward or theoretical view. We decide how to update each state by looking forward to future rewards and states.

(The increments for other states are of course $\Delta \mathcal{V}_t(x) = 0$, for all $x \neq X_t$.)

The approach that we have been taking so far is what we call the theoretical, or *forward*, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them. We might imagine ourselves riding the stream of states, looking forward form each state to determine its update, as suggested by Figure 3.8. After looking forward from and updating one state, we move on to the next and never have to work with the preceding state again. Future state, on the other hand, are viewed and processed repeatedly, once from each vantage point preceding them.

3.3.2 The Backward View of TD(λ)

In the previous subsection we presented the forward or theoretical view of the tabular TD(λ) algorithm as a way of mixing backups that parametrically shifts from a TD method to a Monte Carlo method. In this subsection we instead define TD(λ) mechanistically. The mechanistic, or *backward*, view of TD(λ) is useful because it is simple conceptually and computationally. In particular, the forward view itself is not directly implementable because it is *acausal*, using at each step knowledge of what will happen many steps later. The backward view provides a causal, incremental mechanism for approximation the forward view.

In the backward view of TD(λ), there is an additional memory variable associated with each state, its *eligibility trace*. The eligibility trace for state x at time t is denoted $e_t(x) \in \mathbb{R}^+$. On each step, the eligibility traces for all states decay by λ, and the eligibility trace for the one state visited on

the step is incremented by 1:

$$e_t(x) = \left\{ \begin{array}{ll} \lambda e_{t-1}(x) & \text{if } x \neq X_t; \\ \lambda e_{t-1}(x) + 1 & \text{if } x = X_t, \end{array} \right\} \tag{3.26}$$

for all $x \in \mathcal{X}$, where λ is the parameter introduced in the previous subsection. Henceforth we refer to λ as the *trace-decay parameter*. This kind of eligibility trace is called an *accumulating* trace because it accumulates each time a state is visited, then fades away gradually when the state is not visited. At any time, eligibility traces record which states have recently been visited, where "recently" is defined in terms of λ. The traces are said to indicate the degree to which each state is *eligible* for undergoing learning changes, should a reinforcing event occur. The reinforcing events we are concerned with are the moment-by-moment one-step TD errors. For example, the TD error for state-value prediction is

$$\delta_t = r_{t+1} + \mathcal{V}_t(X_{t+1}) - \mathcal{V}_t(X_t). \tag{3.27}$$

In the backward view of TD(λ), the global TD error signal triggers proportional updates to all recently visited states, as signaled by their nonzero traces:

$$\Delta\mathcal{V}_t(x) = \alpha\delta_t e_t(x), \qquad \text{for all } x \in \mathcal{X}. \tag{3.28}$$

As always, these increments could be done on each step to form an on-line algorithm, or saved until the end of the episode to produce an off-line algorithm. In either case, Equations (3.26 - 3.28) provide the mechanistic definition of the TD(λ) algorithm. Pseudo code for on-line TD(λ) is given in Figure 3.9.

The backward view of TD(λ) is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to the state's eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states, as suggested by Figure 3.10. Where the TD error and traces come together, we get the update given by Equation (3.28).

To better understand the backward view, consider what happens at various values of λ. If $\lambda = 0$, then by Equation (3.26) all traces are zero at t except for the trace corresponding to X_t. Thus the TD(λ) update, Equation (3.28), reduces to the simple TD rule (Equation (3.22)), TD(0). In terms of Figure 3.10, TD(0) is the case in which only the one state preceding the current one is changed by the TD error. For larger values of λ, but still $\lambda < 1$, more of the preceding states are changed, but each more temporally distant state is changed less because its eligibility trace is smaller, as suggested in the figure. We say that the earlier states are given less *credit* for the TD error.

Initialize $V(x)$ arbitrarily and $e(x) = 0$, for all $x \in \mathcal{X}$

Repeat (for each episode)
    Initialize x
    Repeat (for each step of episode):
        $a \leftarrow$ action given by policy π for state x
        Take action a; observe reward, r, and next state, y
        $\delta \leftarrow r + \mathcal{V}(y) - \mathcal{V}(x)$
        $e(x) \leftarrow e(x) + 1$
        For all x:
            $\mathcal{V}(x) \leftarrow V(x) + \alpha\delta e(x)$
            $e(x) \leftarrow \lambda e(x)$
        $x \leftarrow y$
    until x is terminal

Figure 3.9. On-line tabular TD(λ).

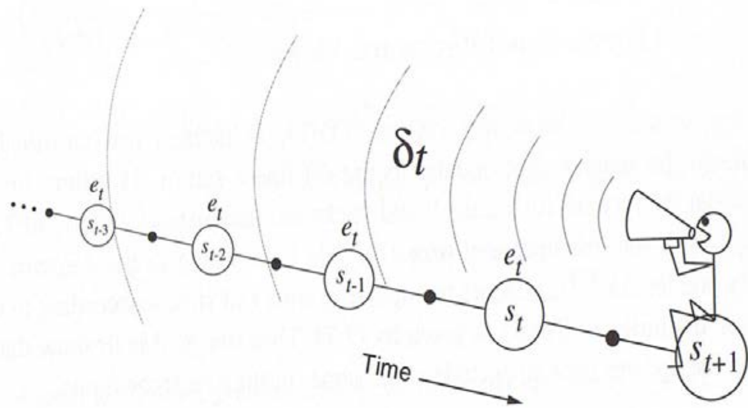

Figure 3.10. The backward or mechanistic view. Each update depends on the current TD error combined with traces of past events.

If $\lambda = 1$, then the eligibility trace do not decay at all with time. If $\lambda = 1$, the algorithm is also known as TD(1). Whereas Monte Carlo methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well. Moreover, TD(1) can be performed incrementally on-line. One disadvantage of Monte Carlo methods is that they learn nothing from an episode until it is over. On-line TD(1), on the other hand, learns in an n-step TD way from the incomplete ongoing episode, where the n steps are all the way up to the current step. If something unusually good or bad happens during an episode, control methods based on TD(1) can learn immediately and alter their behavior on that same episode.

Chapter 4

Results from the motorway A10, Netherlands

This section describes the case study which implements the above methodology for estimating the route travel time given the congestion type on each link. This case study is intended to provide a better understanding how the proposed model can estimate the expected travel time using speed data.

4.1 Markov Decision Processes

In this analysis, we have two choices: We can drive from link A to B clockwise (to the right) or anti-clockwise (to the left). Figure 4.1 and 4.2 show how the speed in a link changes over time when driving anti-clockwise, where Figure 4.3 and 4.4 show how the speed in a link changes over time when driving clockwise. From these figures several remarkable deductions can be made. First, the yellow and pink lines represent the weekend-days, on these days congestions happen very rarely. If congestion on these days happens (as you can see on July 3th in figure 4.5), this is usually caused by irregular events, like an accident or road works. Another remarkable observation in these figures are the large fluctuations during the night (midnight until about six o'clock). In this period the flow rate (the number of vehicles per hour) is relatively small, therefore the average speed calculations show big differences. Finally, we observe that the traffic jams shift evenly through the links over time. Congestion occurrences are usually registered at a specific detection location. However, this study focuses on estimating travel time on a link, and therefore, the congestion types are defined based on the average speed per minute at a specific detector station within a link. Consequently, in this particular study, congestion types are determined based on average link speeds.

4.1.1 Definition of States and Variables

Figure 4.6 shows how the system variables are defined at each time interval for a system with two types of congestion: congestion or no congestion. It is assumed that all time intervals shown in Figure 4.6 are equal (five minutes), and the red bars represent the duration of congestion at each link. There is no congestion on at links 2 and 5 during the period described, while there are two congestions on link 3 and 4, and a long congestion on link 1.

For example, the system state variable at time 17:00, $X(t = 17.00)$, can be described as $[0, 1, 1, 0, 1]$, and the system state variable at time 17:40, $X(t = 17.40)$, can be described as $[0, 0, 0, 0, 1]$. In this situation, the state number could be calculated using

$$state = \sum_{i=1}^{5} x_i(t) 2^{i-1}, \tag{4.1}$$

state	0	1	4	5	6	9	20	21	24	25	26	27	29	30	31	45	46	47	49	50
$x_{12}(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{11}(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_{10}(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_9(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_8(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_7(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_6(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_5(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_4(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$x_3(t)$	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	2
$x_2(t)$	0	0	0	1	1	1	4	4	4	0	0	0	0	1	1	4	4	4	4	0
$x_1(t)$	0	1	4	0	1	4	0	1	4	0	1	2	4	0	1	0	1	2	4	0

Table 4.1. Defined states during the period

where $x_i(t) \in 0, 1$. The maximum number of different states equals $2^5 = 32$, $x \in \{0, 1, \ldots, 31\}$.

In this particular study we did not use two types of congestion, but we separated the average speed in a link into five intervals with each interval belonging to a separate congestion type. For example, congestion type 4 means that there is a heavy congestion with a average speed on that link lower than 25% of the maximum speed, congestion type 0 means there is no congestion. Although we use more different type of congestions in this study, the state definition is analogue to the example above with two types of congestion and 12 different links. In total we have information for 24 different links, 12 clockwise and 12 anti-clockwise, but when we want to travel from A to B, we only use the information for link A to B clockwise and for link $A-1$ to $B-1$ anti-clockwise. For example, when we want to travel from the beginning of link 2, $A = 2$, until the end of link 5, $B = 5$, we add the information for links $2, 3, 4$ and 5 clockwise and for links $1, 12, 11, \ldots, 6$ anti-clockwise to our state space. Thus, the maximum number of different states equals 5^{12}, $x \in \{0, 1, \ldots, 244140625\}$. The parameters used in the example above (traveling from link 2 until link 5), will be used in the rest of this section. During the period analysed, June 17th until July 12th 2005, only 4653 different states occured. Table 4.1 shows the first 20 of them (in sorting order). The time intervals used in this particular research are assumed to be equal to one minute, hence the average speed on a link may change in a minute, consequently each minute the state can change.

4.1.2 Transition Matrix

Based on the defined states and congestion occurrences in a 1-minute time span, the one-step transition matrices are calculated as shown in Table 4.2 for a part of the transition matrix. Almost all states have a high change of going to state zero, the state that occurs most frequently in the observed data. If we have a look at the complete transition matrix, we also see that the process often remains in its present state (i.e., p_{ii} has higher probabilities than p_{ij}).

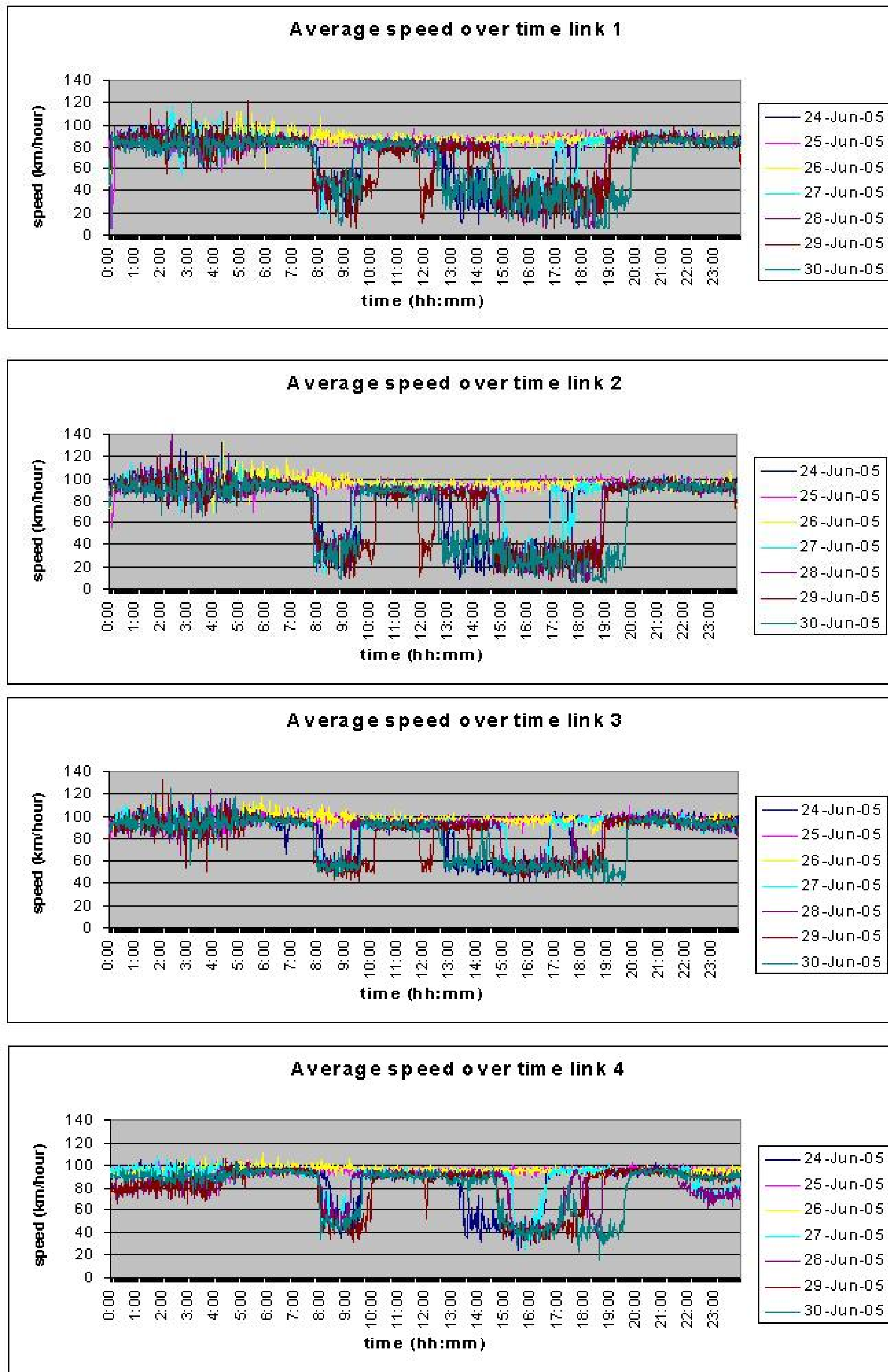

Figure 4.1. Graphical overview of average speed changing over the day for different links, driving anti-clockwise, over the period 24 until 30 June.

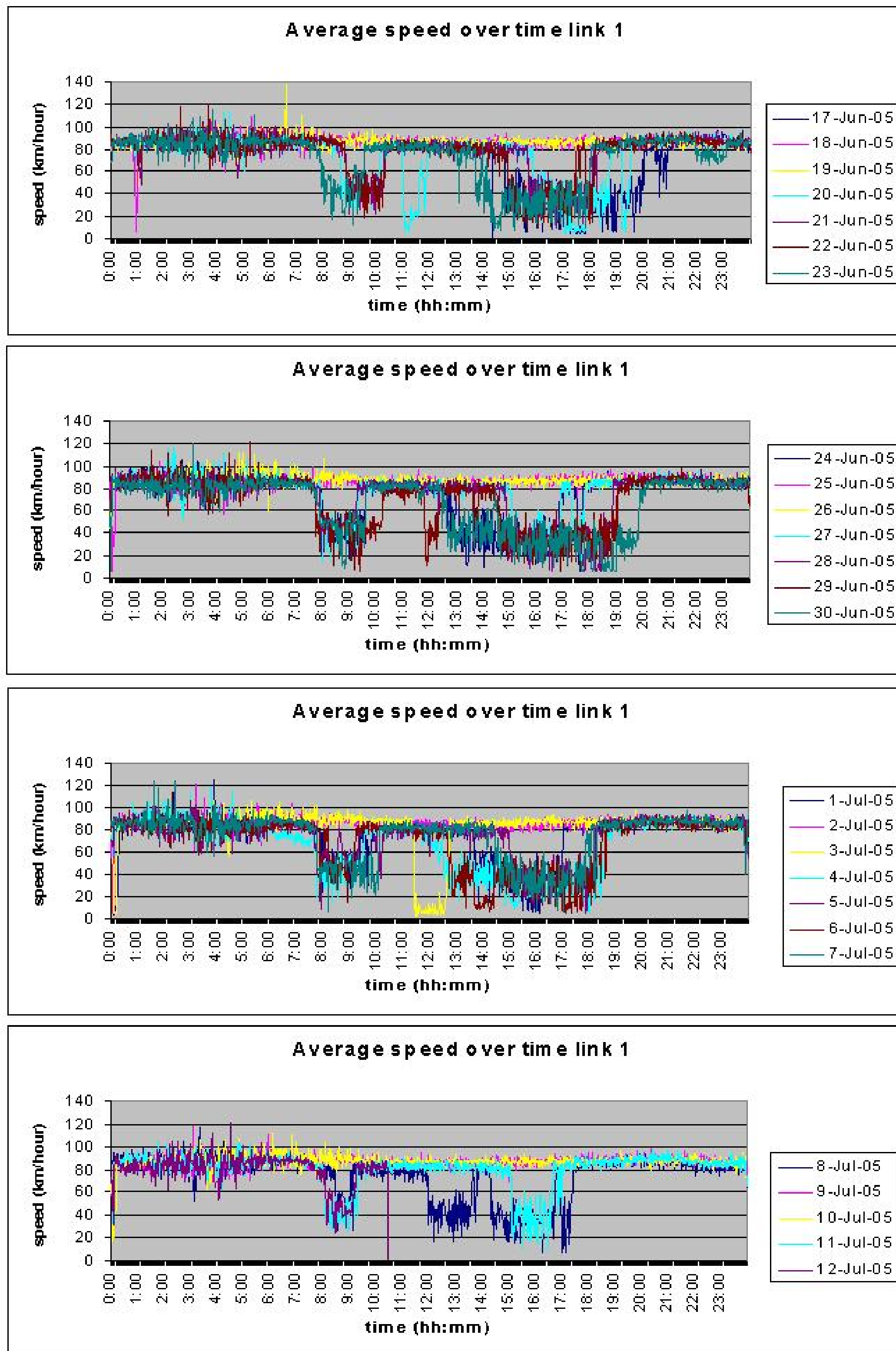

Figure 4.2. Graphical overview of average speed changing over the day for different weeks, driving anti-clockwise, over link 1.

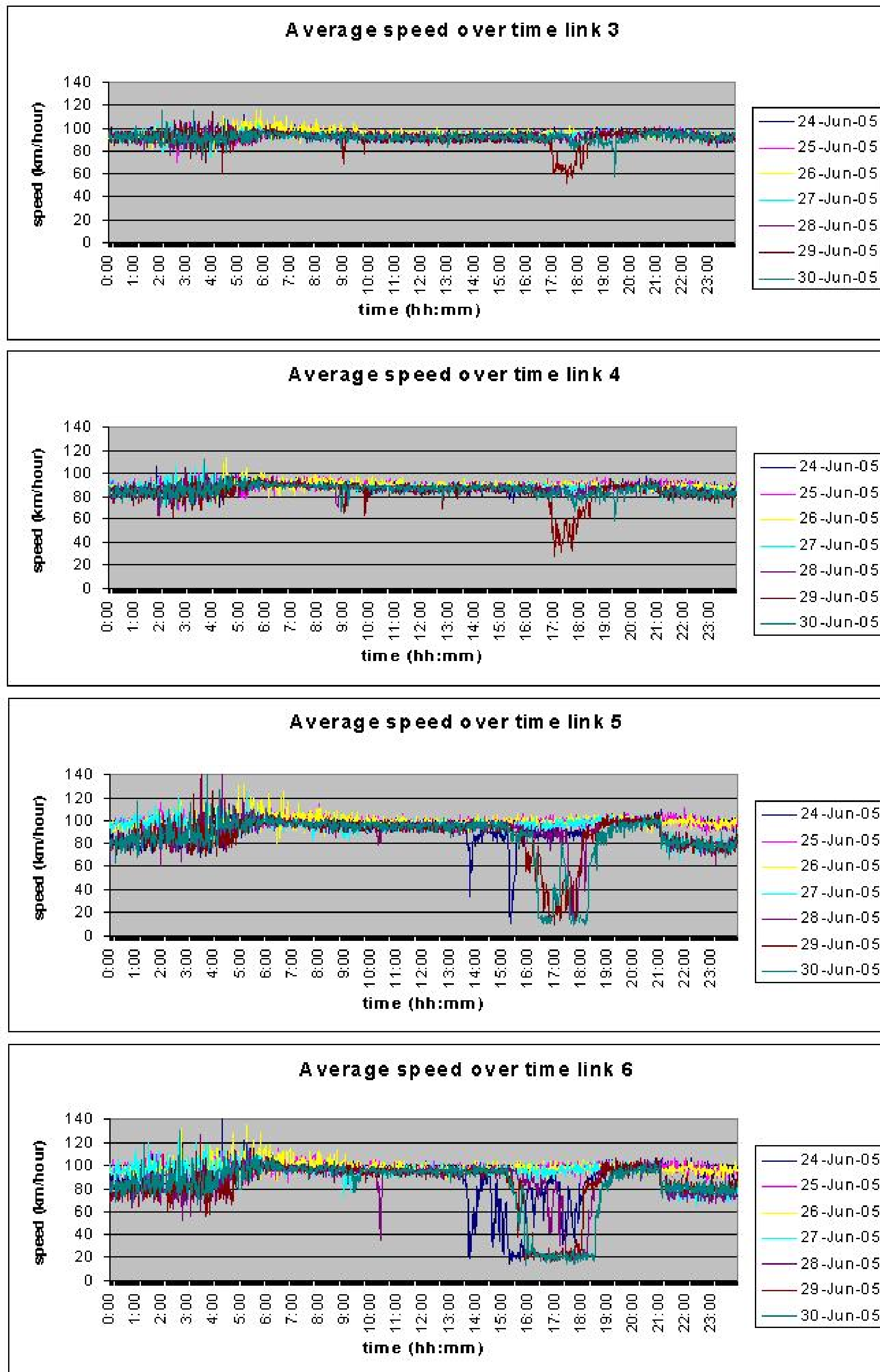

Figure 4.3. Graphical overview of average speed changing over the day for different links, driving clockwise, over the period 24 until 30 June.

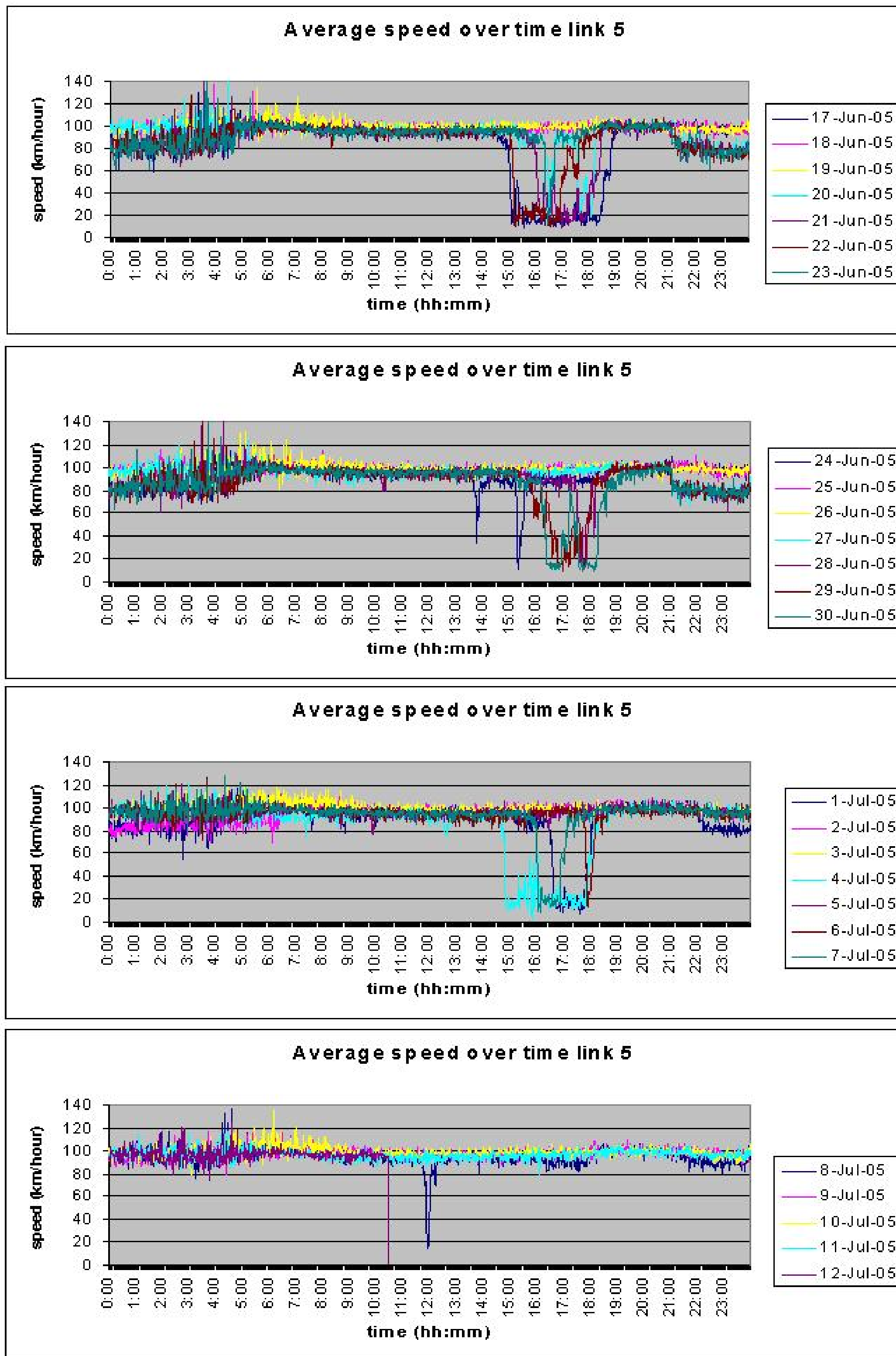

Figure 4.4. Graphical overview of average speed changing over the day for different weeks, driving clockwise, over link 5.

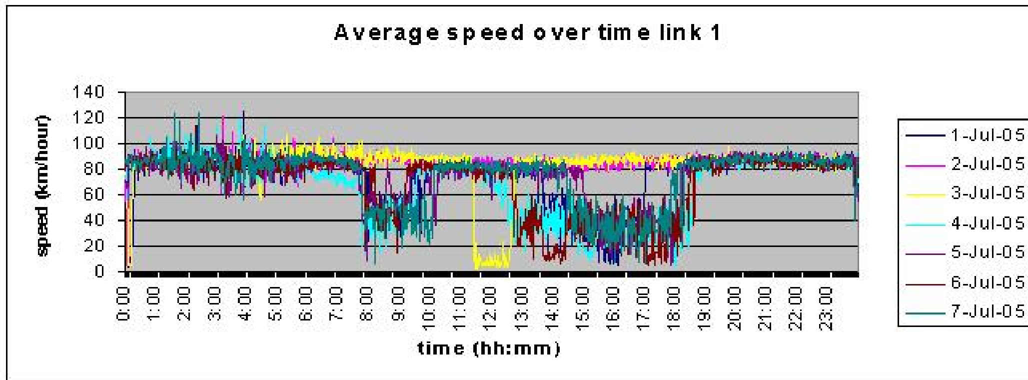

Figure 4.5. Example of a congestion on a weekend day(July 3th, 2005).

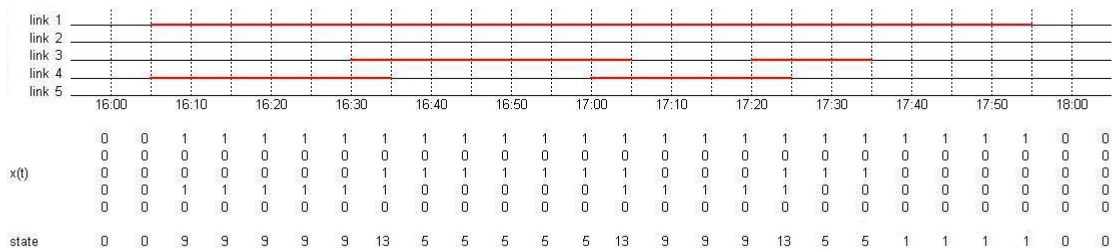

Figure 4.6. Defined system state variable with two types of congestion.

state	0	1	4	5	6	9	20	21	24	25	26	27	29	30	31	45	46	47	49	50
0	0.881	0.003	0.001	0.000	0.000	0.000	0.001	0.000	0.001	0.014	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1	0.642	0.025	0.012	0.000	0.000	0.000	0.012	0.000	0.012	0.025	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4	0.380	0.000	0.000	0.000	0.000	0.000	0.020	0.000	0.000	0.060	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
5	0.250	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.125	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6	0.167	0.167	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
9	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
20	0.300	0.050	0.050	0.000	0.000	0.000	0.000	0.025	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
21	0.333	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.167	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
24	0.520	0.000	0.020	0.000	0.000	0.000	0.020	0.000	0.000	0.020	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
25	0.314	0.004	0.005	0.000	0.003	0.000	0.000	0.000	0.003	0.252	0.000	0.000	0.001	0.000	0.000	0.001	0.000	0.000	0.001	0.006
26	0.308	0.000	0.000	0.000	0.000	0.000	0.077	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
27	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
29	0.333	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
30	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
31	0.500	0.000	0.500	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
45	0.500	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
46	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
47	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
49	0.125	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
50	0.019	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.222	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.370

Table 4.2. Part of the one-step transition matrix for the first 20 states, sorted by state number

4.2 Dynamic Programming

The estimated travel time of each link can be calculated using

$$\overline{T_i} = \frac{|x_i|}{s_{ij}}, \tag{4.2}$$

where $|x_i|$ is length of link i in meters and s_{ij} is the average speed in meters per second of link i given there is a congestion of type j on that link.

The estimated travel time or reward in a state, $r(x, from, to, d, a)$, is defined as a function of the recent state, x, the link in which we are at the moment, called $from$, our destination link, called to, the distance already covered in the present link, d, and the action, driving clockwise or anti-clockwise. Thus, when we start at the beginning of a link: $d = 0$. Because our state changes each minute, our reward is always less or equal to 60 (seconds). Equal to 60 in the case that we did not yet arrive at our destination after the minute, less if we did arrive at our destination.

$$r(x, from, to, d, a) = \left\{ \begin{array}{ll} 0 & \text{if } from = to + 1 \\ 60 & \text{if } (\alpha\overline{T_{from}} + \sum_{i=from+1}^{to} \overline{T_i}) > 60 \\ \alpha\overline{T_{from}} + \sum_{i=from+1}^{to} \overline{T_i} & \text{otherwise} \end{array} \right\}, \tag{4.3}$$

where $\alpha = \frac{|x_{from}| - d}{|x_{from}|}$ is the fraction of the first link to be covered. In the first situation the total route is already covered when we start a new minute, hence the expected future travel time equals zero. In the second situation we did not arrive at our destination and used the whole minute to travel. In the last situation we arrive at our destination within the minute. Obviously, this is the exact formula for the action: driving clockwise. For driving anti-clockwise we take the sum over $i = from - 1, from - 2, \ldots, to$ and we should take $from = to - 1$ in the first part of the equation. Another easier and less confusing option is to renumber the links, $i_{new} = 12 - i_{old} + 1$.

Next, we determine the new origin, $from'$, and the distance already covered in this link, d':

$$from' = \left\{ \begin{array}{ll} to & \text{if } r(x, from, to, d, a) \leq 60 \\ from & \text{if } \alpha\overline{T_{from}} > 60 \\ j & \text{if } r(x, from, j, d, a) > 60 \end{array} \right\}, \tag{4.4}$$

$$d' = \left\{ \begin{array}{ll} 0 & \text{if } r(x, from, to, d, a) = r(x, from, from', d, a) \\ |x_i| - (r(x, from, to, d, a) - 60)s_{ij} & \text{otherwise,} \end{array} \right\}. \tag{4.5}$$

where $((r(x, from, to, d, a) - 60)s_{ij}$ represents the meters covered after the minute ends, $|x_i|$ represents the total length of link i and s_{ij} is the average speed in meters per second of link i given there is a congestion of type j on that link.

Using the estimated travel time of each link in a time unit and transition matrices of each origin-destination pair in the previous step, expected route travel times can be estimated by the following equation:

$$\overline{T}(x, from, to, d, a) = r(x, from, to, d, a) + \sum_{Z(x)} p(x, a, y)\overline{T}(y, from', to, d', a), \tag{4.6}$$

where $Z(x) \subset \{y | p(x, a, y) > 0\}$, contains all possible states that can be reached from state x. The first part of the summation represents the expected travel time in the present minute, where the second part represents the travel time estimation for future minutes.

The complete algorithm for recursive travel time estimation is given in Figure 4.7.

We calculate the expected travel times for both actions and make our choice for a direction based on the minimum travel-time estimation:

$$a = \arg\min_{a \in \mathcal{A}} \overline{T}(x, from, to, d, a). \tag{4.7}$$

An example of the output of the program written to validate the solutions of the MDP model using Dynamic Programming can be found in Figure 4.8. One can enter the congestion types on each link by hand, even as the start and destination link.

Results of the travel-time estimation using Dynamic Programming to solve the Markov Decision Process can be found in Figures 4.9 and 4.10. We still use the example data for driving from link 2 to the end of link 5. When we input all possible state numbers in the program, in 3275 of the 4653 cases the program outputs drive clockwise, where in only 1126 cases the system advises to drive anti-clockwise. Unfortunately our state representation does not present any information about the total congestion of a state, but only about the congestion on the different links. Therefore, we choose to present our results against the sum of congestion over several links, $S_{i,j}$, which is defined by

$$S_{i,j}(x) = \sum_{k=i}^{j} x_k, \tag{4.8}$$

where $x_k \in \{0, 1, \ldots, 4\}$ represent the congestion type on link k. We have to remark that several different states sum up to the same value for $S_{i,j}(x)$ and this value does not represent any information about which link contains which type of congestion.

Figure 4.9 shows a graphical representation of all possible travel time estimations for the example traveling from link 2 to link 5. For each travel time estimation the congestion type combinations is plotted against the sum of the congestion types over all links, $S_{1,12}$, using two actions: driving clockwise or anti-clockwise. Figures 4.11 and 4.12 split this figure in two separate actions. First, we clearly see a linear increase in the estimated travel time when there is more or heavier congestion. Second, we observe that the shortest estimated travel times belong to the action: drive clockwise, which seems to be logical because then a shorter distance has to be covered. Third, a more debatable observation we can make is, the more congestion the more frequently we drive anti-clockwise. On one hand, this observation matches our expectations: we will only consider driving a longer distance if there is a lot of delay caused by congestion, when there is no congestion at all we would always choose the action which covers the shortest distance. But on the other hand, we would only choose to take the long way round if we are in a shorter time at our destination.

Figure 4.10 shows a graphical representation link 2 until 5 only, $S_{2,5}$, using the same actions: driving clockwise or anti-clockwise. Again we split this figure in two separate actions, see Figures 4.13 and 4.14. When drawing conclusions we have to keep in mind that there is a big difference in the occurrence of both actions, the action "driving clockwise" is more likely to occur. Although, there is an obvious action changing boundary, for $S_{2,5} > 9$ in most cases it is optimal to drive anti-clockwise, where for $S_{2,5} \leq 9$ is it optimal to take the action with the shortest distance: drive clockwise. This completely matches our expectations.

4.3 Temporal-Difference Learning

We started to obtain the estimated travel times of the Temporal-Difference learning by implementing the TD(0) algorithm. Figures 4.15 and 4.16 show the results for anti-clockwise and clockwise driving, respectively. The times represented in the graphs are the times from the start of link 1 to the end of link 12. When we want to calculate the travel time from link A until the end of link B we use the following equation:

$$\overline{T}(A, B, a) = \widehat{T}(A, 12, a) - \widehat{T}(B+1, 12, a) \tag{4.9}$$

If we have a closer look to the results, again, we immediately see that the estimated travel times on weekend days are significantly lower than on the weekdays. We also see that the estimated travel times on weekdays driving clockwise are higher than the estimated travel times for driving anti-clockwise. However, the most remarkable fact are the not constant decreasing lines, which do not match our expectation about the estimated travel times. If we interpret this results, there would be moments in time that the estimated travel time is shorter when we drive from link $i-1$ to link 12 than it would be when driving from i to link 12. This seems to be weird because the distance covered from link $i-1$ until link 12 includes the distance covered when driving from link i to link 12. These results can be explained by the fact that the links are not updated in regular order, it can happen that links in the end are updated after the last update of first links.

We could prevent this from happening by using a $TD(\lambda)$ algorithm. In our case we the estimated travel time on link n is always dependent on the estimated travel times of the links that will have to be covered thereafter. Thus if we update link i we should also update link $1, \ldots, n, \ldots$, and $i-1$ as well. Therefore we choose to apply the TD(1) algorithm to the A10 data. The results are summarized in Figures 4.17 and 4.18. As can been seen from these figures the problem we had with TD(0)-learning is solved, all graph are decreasing. The estimated travel times when driving clockwise are longer than when using the TD(0) algorithm.

One thing we did not describe above is our choose for $\alpha = 0.001$. This is the same as setting the travel time estimation to 0.1% of the difference with the travel time observation. The choice for α has great importance on the speed of convergence to the right travel time estimation. If we choose $\alpha = 1$, we update the link time travel time estimation with the difference between the old estimated travel time and the travel time observation. To check the effect of α we calculated the results of Monday June 20th 4.00 PM by several α, which can be found in Figure 4.19.

Let $|\mathcal{X}| = N$ and $Z(x) \subset \{y|p(x,a,y) > 0\}$.

Initialize: $\alpha = \frac{|x_{from}|-d}{|x_{from}|}$, $i = from - 1$, $r(x, from, to, d, a) = \alpha\overline{T_{from}}$ and $sum = 0$.
Repeat
$i \leftarrow i + 1$
  if(($a == 1$ (clockwise) and $from == to + 1$) or ($a == 2$ (anti-clockwise) and $from == to - 1$)
    Output $\overline{T}(x, from, to, d, a) = r(x, from, to, d, a)$
    Exit!
  $r(x, from, to, d, a) \leftarrow r(x, from, to, d, a) + \overline{T_i}$
while($r(x, from, to, d, a) < 60$)

$d' \leftarrow |x_i| - (r(x, from, to, d, a) - 60)s_{ij}$
$from' \leftarrow i$

For each $y \in Z(x)$
  $sum \leftarrow sum + p(x, a, y)\overline{T}(y, from', to, d', a)$
Output $\overline{T}(x, from, to, 0, a) = 60 + sum$

Note: In the equation for d', represents $r(x, from, to, d, a)$ the travel time to the end of the last covered link, which is probably larger than a minute. s_{ij} is the average speed in meters per second on link i given there is a congestion of type j on that link. Hence, $(r(x, from, to, d, a) - 60)s_{ij}$ represents the distance to be covered of link i in the next minute. Because we do not use the distance to cover, but the distance already covered, we take the length of link i, $|x_i|$ and subtract the distance to be covered.

Figure 4.7. Complete algorithm for recursive travel time estimation.

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
In which link do like to start your trip? 2
In which link do you want to end your trip? 5
Enter the type of congestion in link 1:0
Enter the type of congestion in link 2:0
Enter the type of congestion in link 3:0
Enter the type of congestion in link 4:0
Enter the type of congestion in link 5:0
Enter the type of congestion in link 6:0
Enter the type of congestion in link 7:0
Enter the type of congestion in link 8:0
Enter the type of congestion in link 9:0
Enter the type of congestion in link 10:0
Enter the type of congestion in link 11:0
Enter the type of congestion in link 12:0

The state number is: 0
***************************************************
Estimated travel-time when driving clockwise: 80.57605535204121 seconds.
Estimated travel-time when driving anti-clockwise: 158.26812944598413 seconds.
***************************************************
For state 0 000000000000 amount the estimated travel time when driving clockwise
: 1 minutes and 21 seconds.
***************************************************
Press any key to continue...
```

Figure 4.8. The output of the program written to test the DP-method on the MDP model developed in this study.

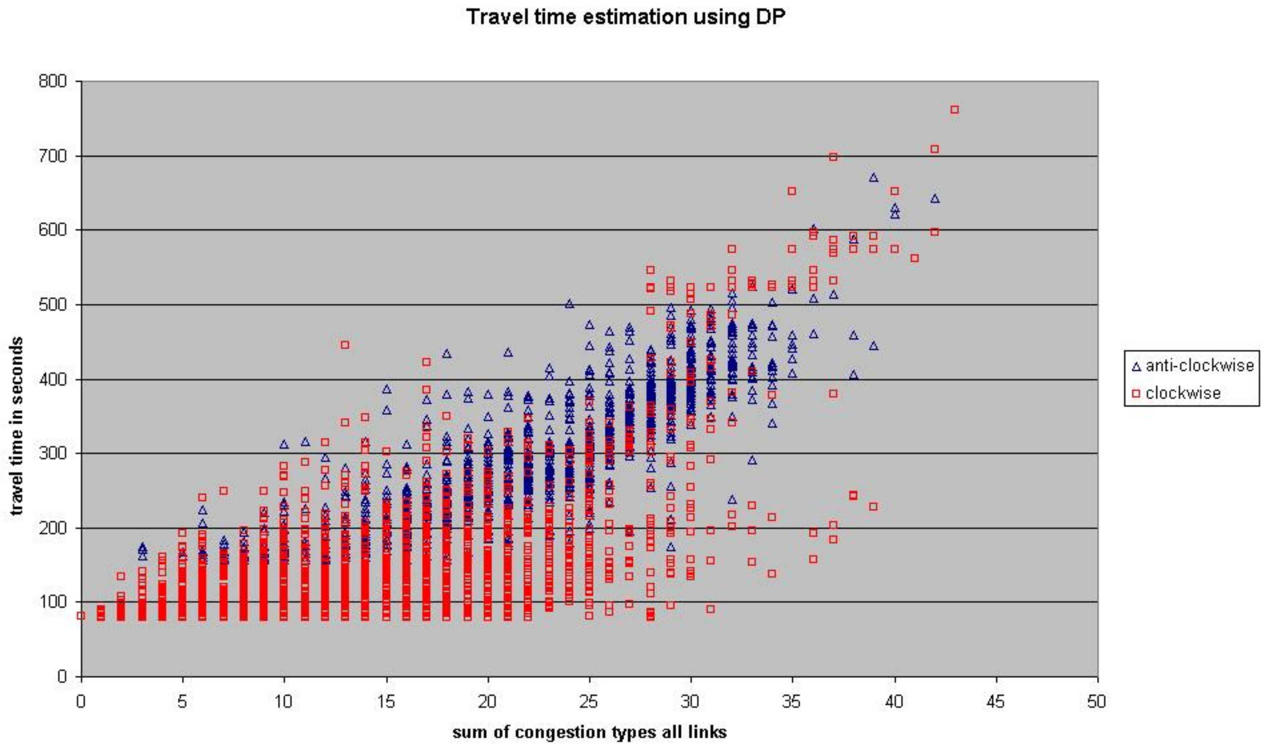

Figure 4.9. Travel time estimations by several congestion type combinations against the sum of the congestion types over all links, $S_{1,12}$, using two actions.

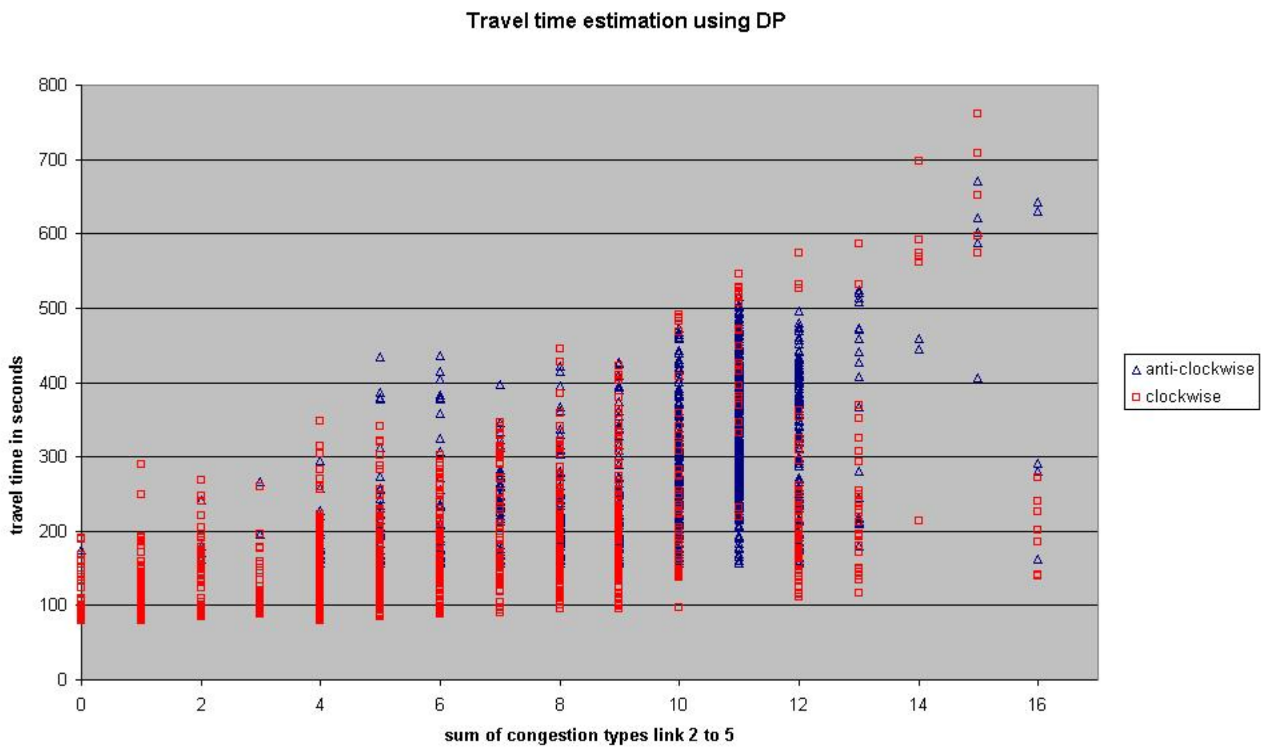

Figure 4.10. Travel time estimations by several congestion type combinations against the sum of the congestion types over link 2 until 5, $S_{2,5}$, using two actions.

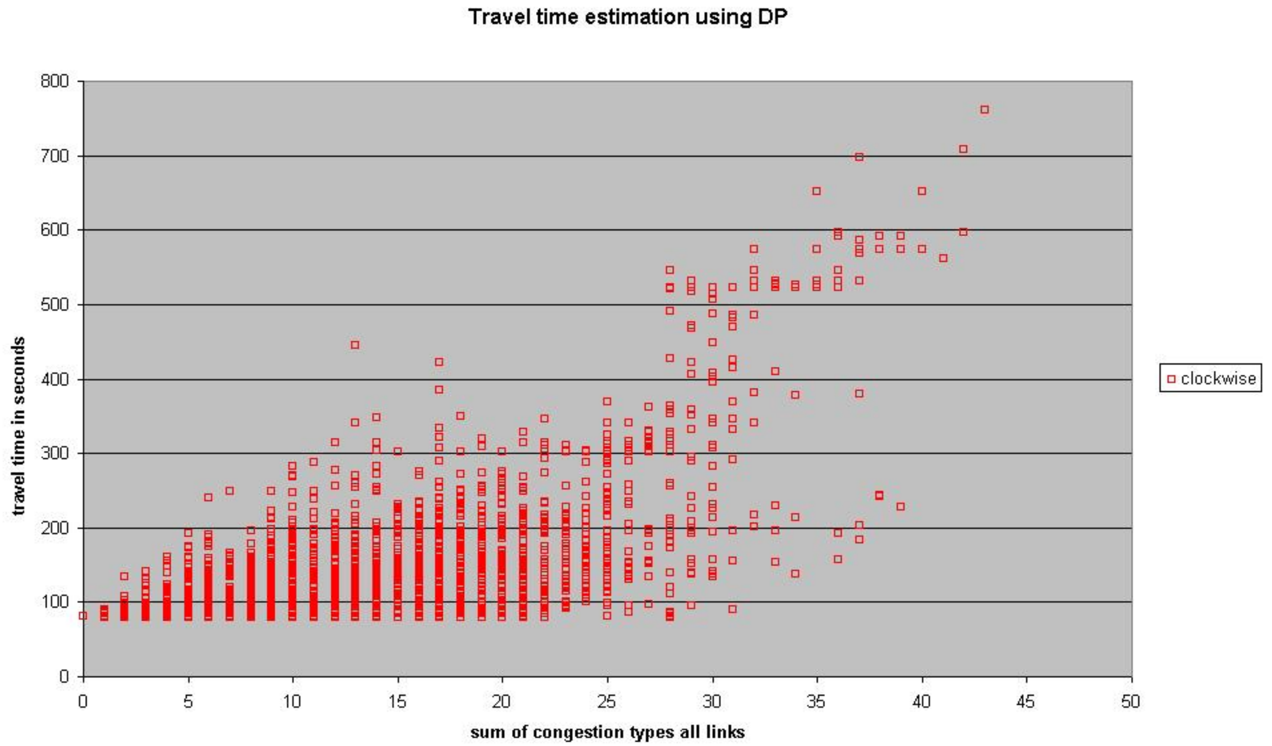

Figure 4.11. Travel time estimations by several congestion type combinations against the sum of the congestion types over all links driving clockwise, $S_{1,12}$.

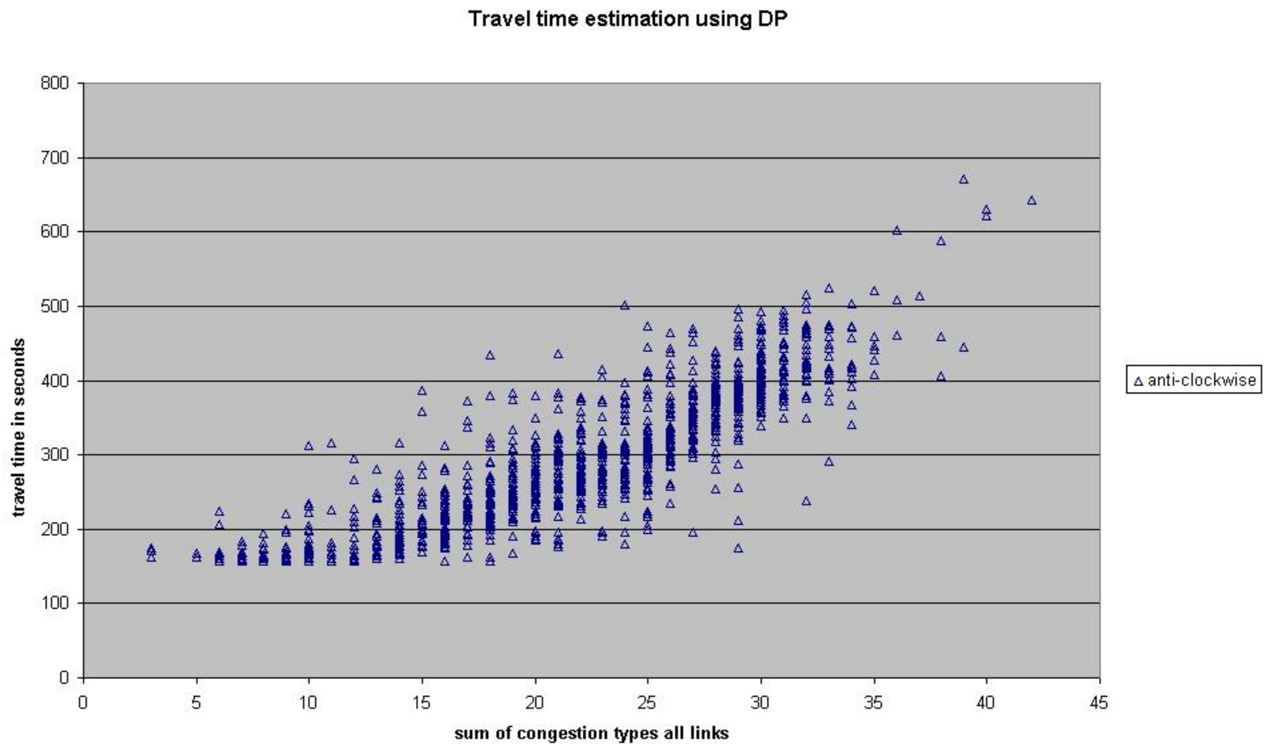

Figure 4.12. Travel time estimations by several congestion type combinations against the sum of the congestion types over all links driving anti-clockwise, $S_{1,12}$.

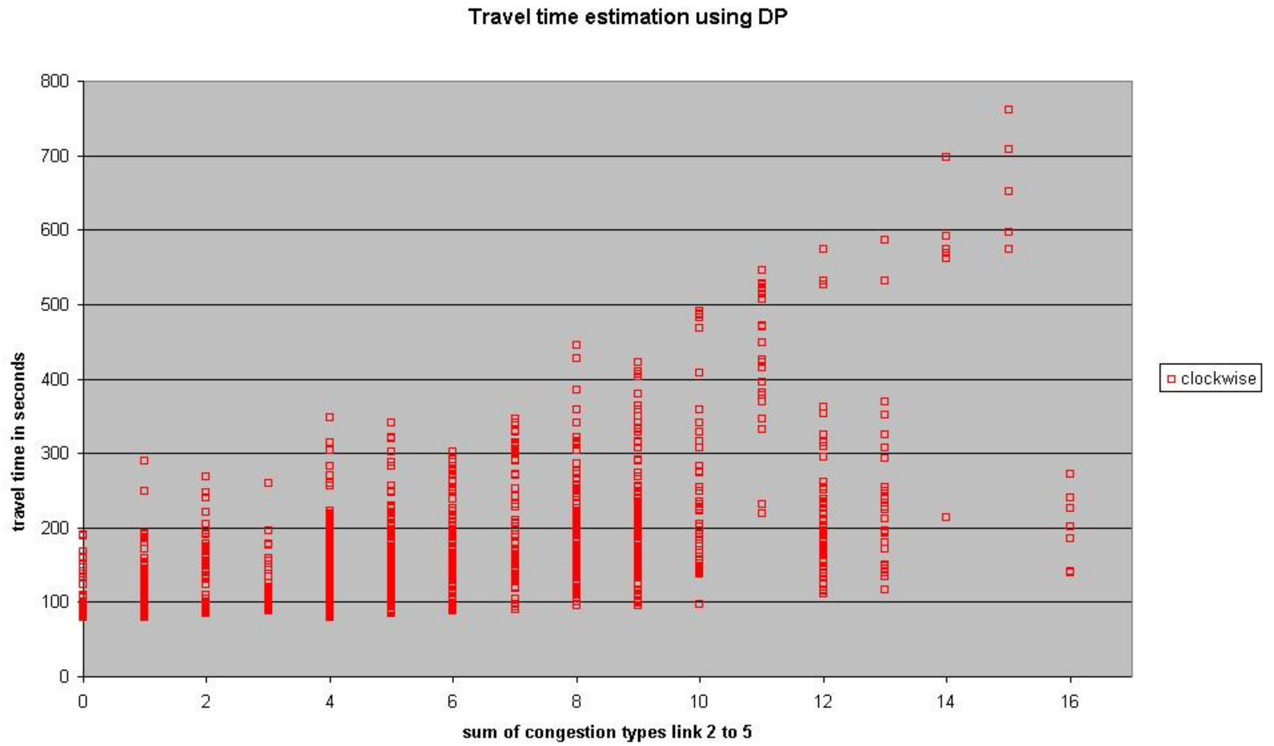

Figure 4.13. Travel time estimations by several congestion type combinations against the sum of the congestion types over link 2 to 5 driving clockwise, $S_{1,12}$.

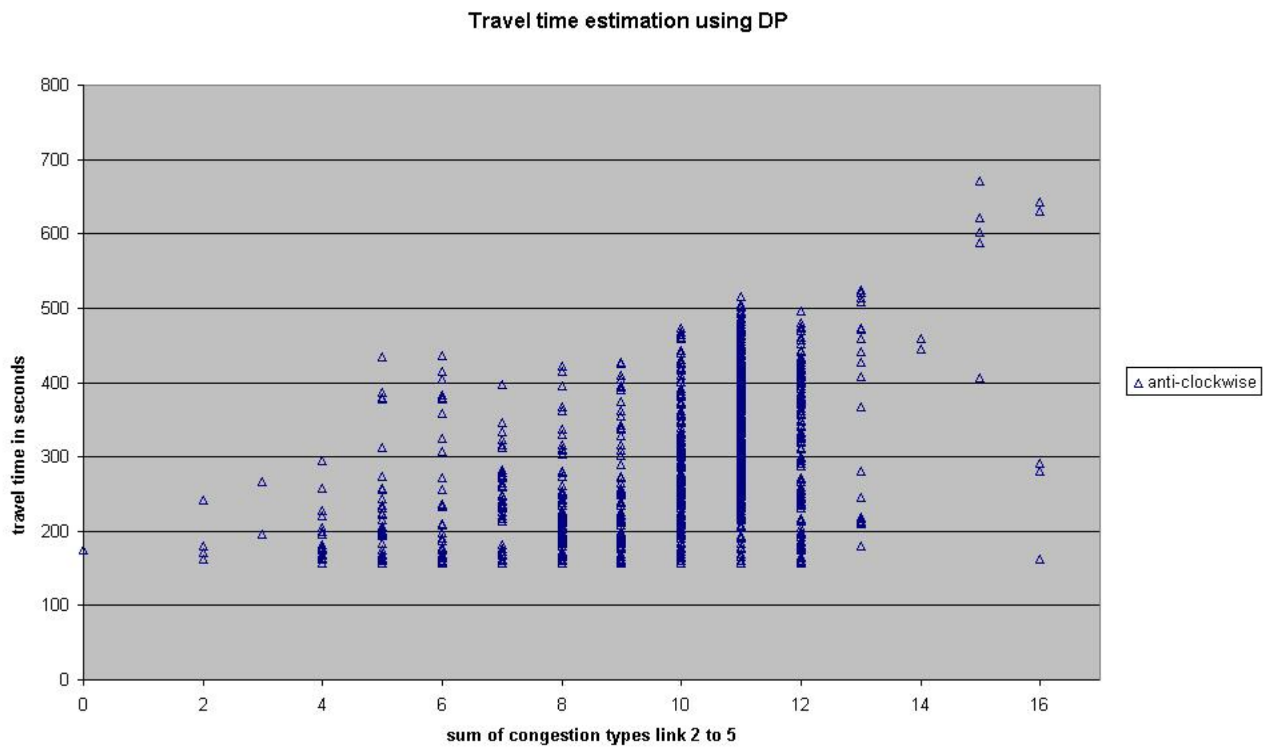

Figure 4.14. Travel time estimations by several congestion type combinations against the sum of the congestion types over link 2 until 5 driving anti-clockwise, $S_{2,5}$.

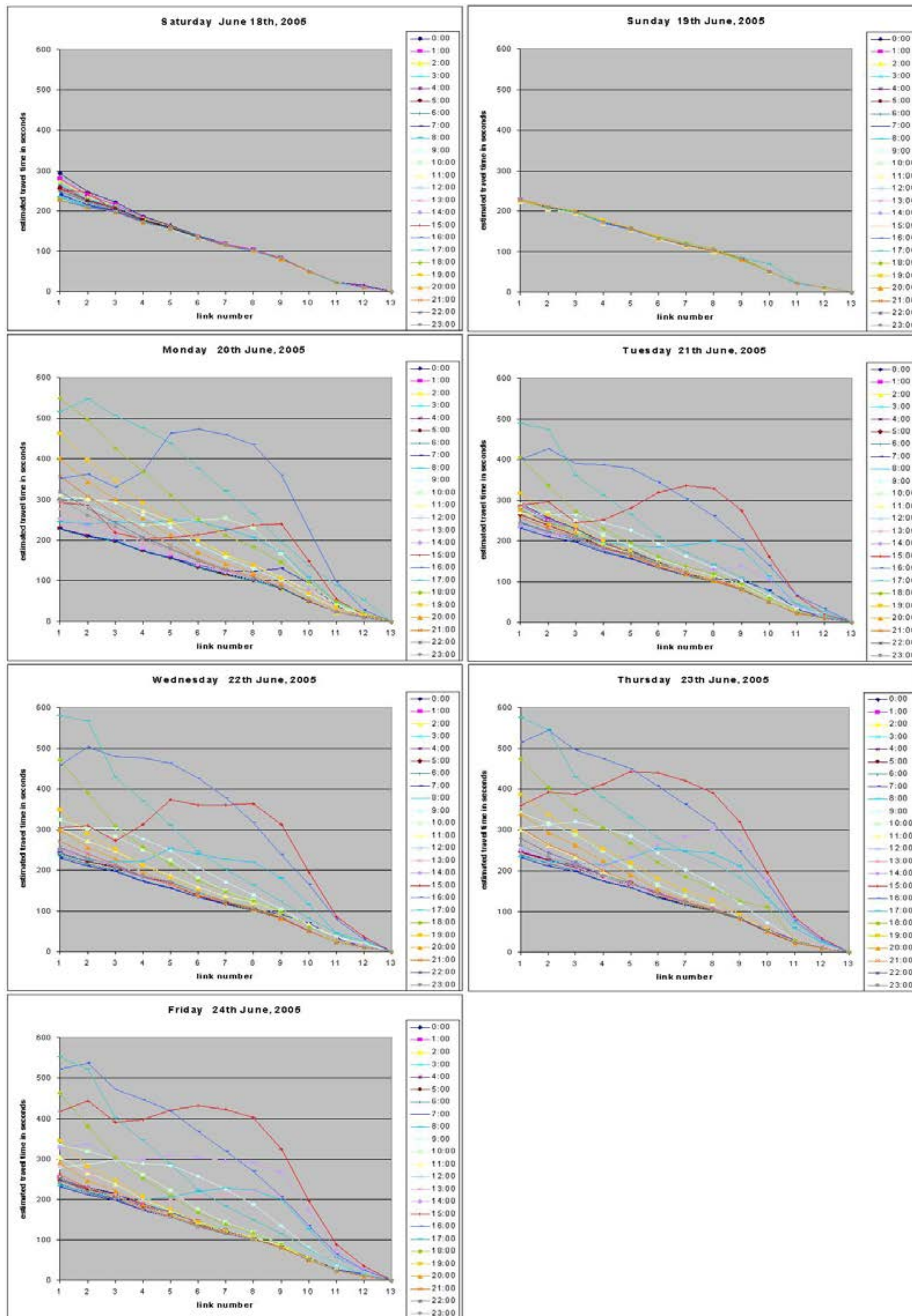

Figure 4.15. Results for TD(0) when driving anti-clockwise.

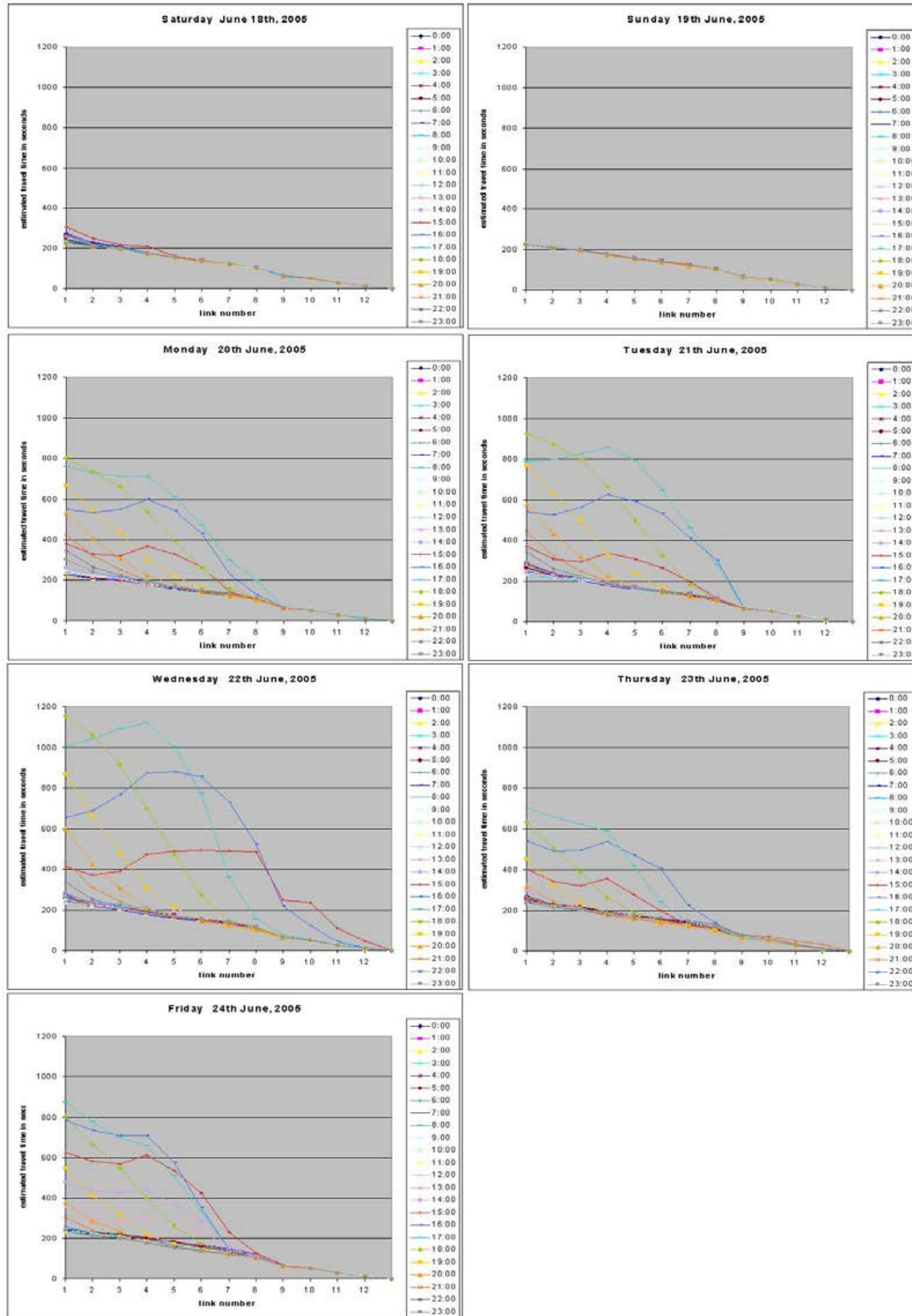

Figure 4.16. Results for TD(0) when driving clockwise.

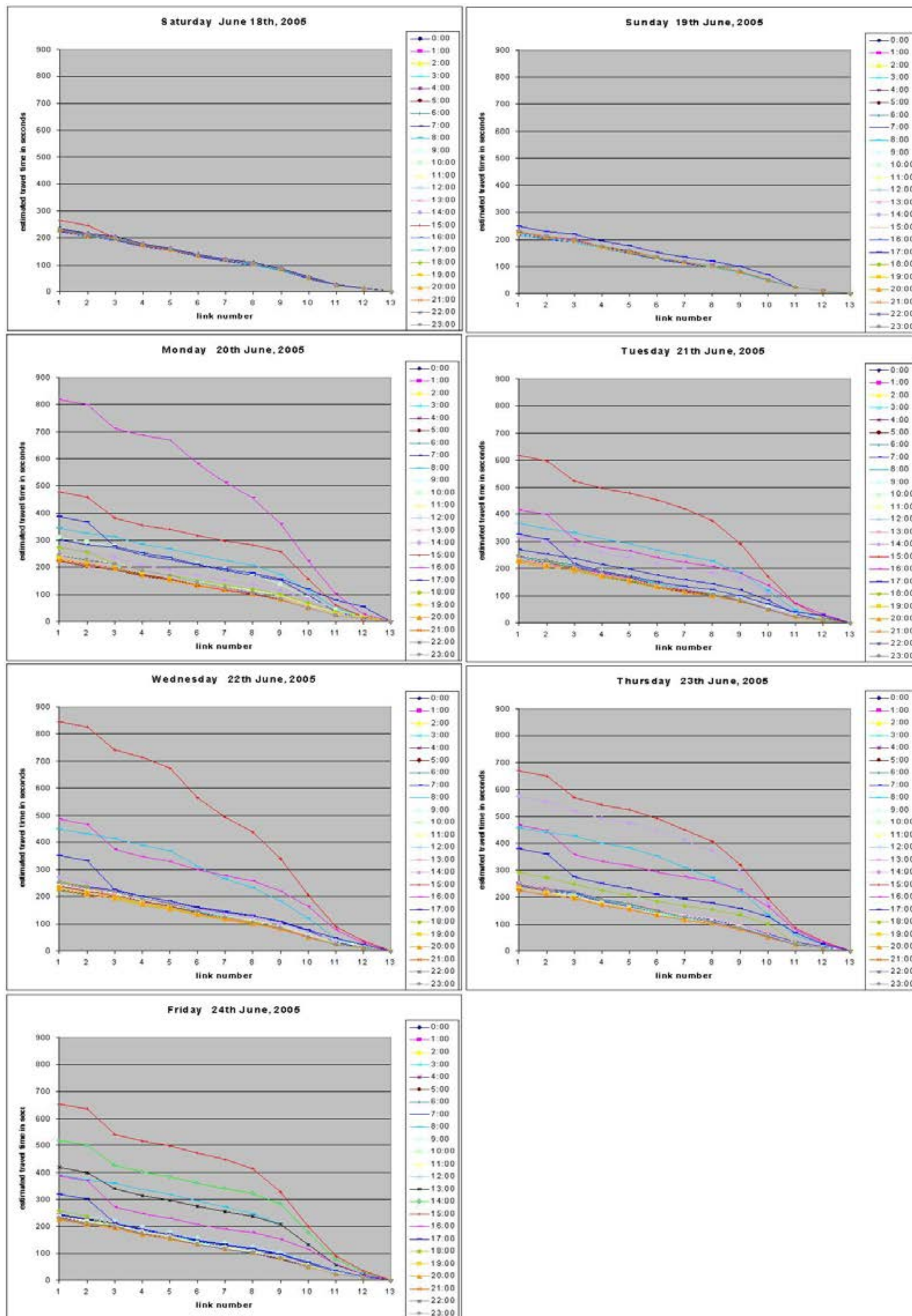

Figure 4.17. Results for TD(1) when driving anti-clockwise.

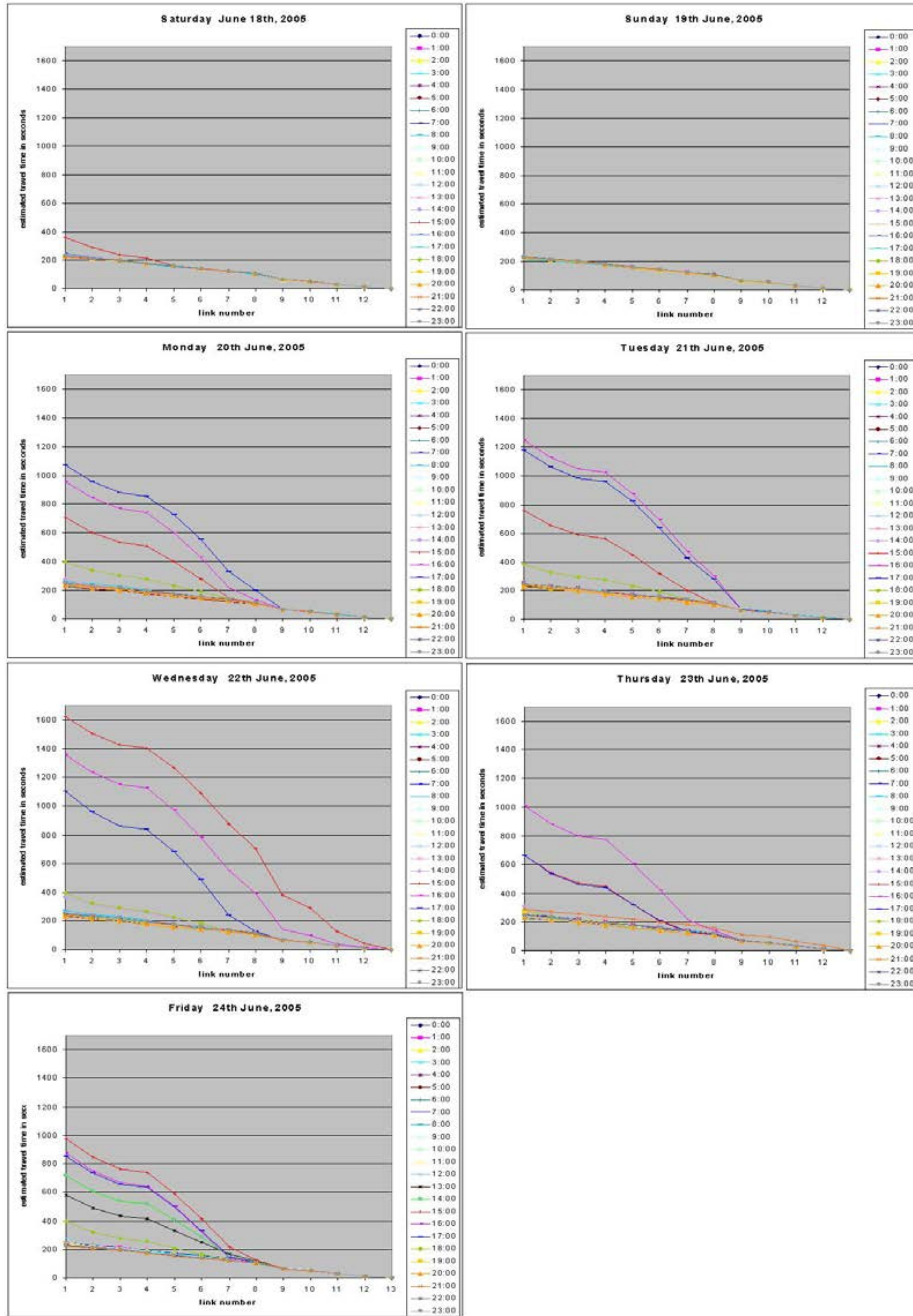

Figure 4.18. Results for TD(1) when driving clockwise.

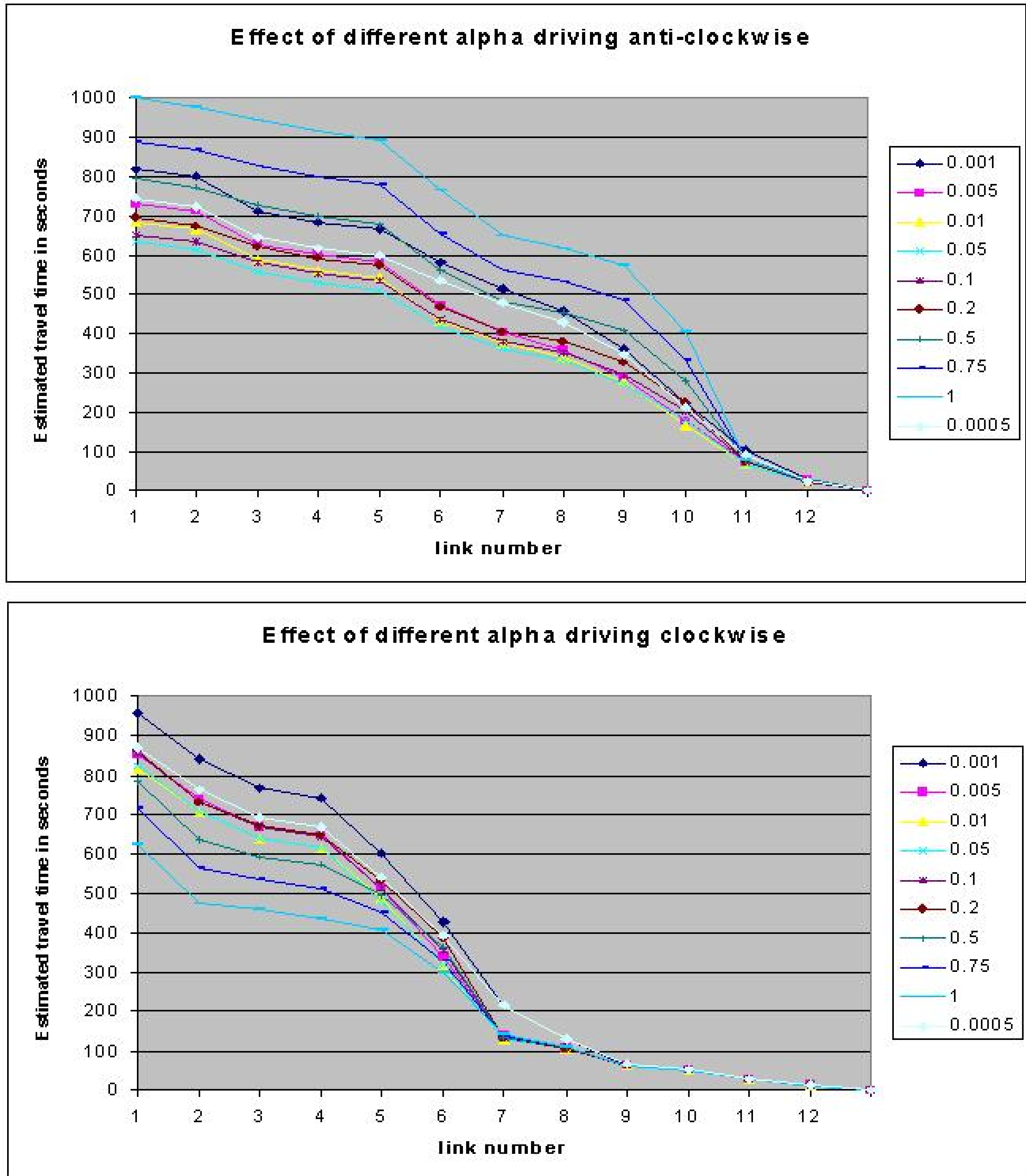

Figure 4.19. Results for different α using TD(1).

Discussion

The purpose of the present study is to develop a real-time travel time prediction method using Markov Decision Processes, in which the system state changes each minute. The model considers different congestion types, based on the average speed along a freeway link, and it subsequently estimates the expected travel time for the entire route as a function of those probabilities of congestion types. When calculating the average link speeds we ignore unreliable data points, for example stationary vehicles or impossible high speeds.

To find the optimal action policy for the developed MDP, we used two types of solution methods, first we applied Dynamic Programming (DP), then we used a Temporal-Difference (TD) algorithm. There are several differences between these methods. The first method bases the route-travel-time estimates on the average speed and consequently the congestion types on each link. When there is a large chance on (heavy) congestion in the coming period the estimated travel time will be longer than in a situation where there is no chance on congestion. For this method we need information of the past to obtain the one-step-transitions between states, which make this method work a bit slower than TD learning. TD learning updates its prediction every time a new observation is made and hence depends only on the actual travel-time observations. This method has proved to be a very fast real-time-travel-time prediction method in practice. As opposed to the differences there are also similarities. Both methods have proved to result in applicable estimates on a dataset from the A10 in the Netherlands. They can be used to decide which action to take, drive clockwise or drive anti-clockwise, and make reliable route-travel-time estimations.

In contrast with recent studies the methods developed in this study react automatically on changing environments, i.e. incidents, work zones, and adverse weather conditions. Hence, the methods are valid in all possible traffic conditions. Additionally, the methods used in this study do not only give a route-travel-time estimation but also provide a travel direction. The same methods could be applied to other freeway segments if appropriate speed data are available even if the are not ring roads. In that case one could collect data from other, non-freeway, roads. Then, the action space changes from driving clockwise or anti-clockwise to drive the freeway or taking the direct route.

The most important limitation of this study is the lack of data. The data used, contained only information about part of the A10 and unfortunately it is not possible to make a connection between the vehicles on different links. Although we may have developed a theoretical model on travel-time estimation it is not possible to measure actual travel-time with the dataset as used, hence we cannot validate our results except for our expectations. Therefore, further research should be done including a more extensive dataset. Second, future research about travel time prediction will be needed to specify the optimal boundaries for dividing the state space into congestion types. Third, in this study we used a constant α-factor when implementing a TD-algorithm, but we could have used a function, based on for example the time of day or average speed. To improve the methods described in this study, one should have a closer look on the content of this function. Finally, one can try to apply the model developed on more than two actions or on more than one action moment.

Nowadays, travel time predictions are very important to plan when, where and how to travel. In this research we developed a model which is useful in practice, the methods used for predicting travel times are fast and applicable in real-time. Furthermore, the results of the predictions meet our expectations and the methods work in all possible traffic conditions. Finally, the model still works when input data is non-reliable or missing.

Bibliography

[1] Highway capacity manual 2000. *Transportation Research Board, Washington, D.C.*, 2000.

[2] J.A. Herts A. Krogh. A simple weight decay can improve generalization. *Advances in Neural Information Processing Systems*, 4:950–957, 1995.

[3] C.M. Bishop. *Neural networks for Pattern Recognition*. Oxford University Press, Oxford, 1995.

[4] W. Brilon and M. Ponzlet. Variability of speed-flow relationships on german autobahns. *Transportation Research Board*, 1555:91–98, 1996.

[5] S. Clark. Traffic prediction using multivariate nonparametric regression. *Journal of Transportation Engineering*, 21:161–168, 2003.

[6] F.M. Sanders C.P.IJ. van Hinsbergen, J.W.C. van Lint. Short term traffic prediction models. *Proceedings of the 14th ITS World Congress, Beijing, China*, 2007.

[7] H.J. van Zuylen C.P.IJ. van Hinsbergen, J.W.C. van Lint. Bayesian committe of neural networks to predict travel times with confidence intervals. *Transportation Research Part C*, 2009.

[8] R.M. Hanbali and D.A. Kuemmel. Traffic volume reductions due to winter storm condition's. *Transportation Research Board*, 1387:159–164, 1993.

[9] A.T. Ibrahim and F.L. Hall. Effect of adverse weather conditions on speed-flow-occupancy relationships. *Transportation Research Board*, 1457:184–191, 1994.

[10] S. Lawphongpanich J. Yeon, L. Elefteriadou. Travel time estimation on a freeway using discrete time markov chains. *Transportation Research Board part B*, 42:325–338, 2008.

[11] P. Vega J.M. Zamarreo. State space neural network. properties and application. *Neural Networks*, 11:1099–1112, 1998.

[12] H.J. van Zuylen J.W.C. van Lint. Monitoring and predicting freeway travel time reliability using width and skew of the day-to-day travel time distribution. *Transportation Research Board 84th Annual Meeting, Washington DC.*, 2005.

[13] H.J. van Zuylen J.W.C. van Lint, S.P. Hoogendoorn. Accurate freeway travel time prediction with state-space neural networks under missing data. *Transportation Research Part C*, 13:347–369, 2005.

[14] H. White K. Hornik, M. Stinchcome. Multilayer feed forward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[15] M. F. Møller. *A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning*. Computer Science Department University of Aarhus, Denmark, 1993.

[16] T.M. Mitchell. *Machine Learning*. McGraw-Hill Book Co, 1997.

[17] N.L. Nihan. Use of the box and jenkins time series technique in traffic forecasting. *Transportation*, 9:125–143, 1980.

[18] M.P. Perrone. *General averaging results for convex optimization.* Connectionists Models Summer School, Hillsdale, NJ, USA, 1994.

[19] D.B. Fambro S. Lee. Application of subset autoregressive integrated moving average model for short-term freeway traffic volume forcasting. *Transportation Research Record*, 1678:179–188, 1999.

[20] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2000.

[21] et al. T.H. Maze, M. Agarwal. Whether weather matters to traffic demand, traffic safety and traffic operations and flow. *Proceedings of the 85th Annual Meeting of the Transportation Research Board, Washington D.C. USA*, 2006.

[22] H.H. Thodberg. *Ace of Bayes, Application of Neural networks with Pruning.* The Danish Meat Research Institute, Roskilde, 1993.

[23] Q. Shi W. Zheng, D. Lee. Short-tem freeway traffic flow prediction: Bayesian combined neural network approach. *Journal of Transportation Engineering*, 132:114–121, 2006.

[24] S.Richardson W.R. Gilks and D.J. Spiegelhalter. *Markov Chain Monte Carlo in Practice.* Chapman & Hall/CRC, 1995.

[25] J.A. Rice X. Zhang. Short-term travel time prediction. *Transportation Research Part C*, 11:187–210, 2003.

Index

Action-value function, 8
Aperiodic, 7

Bellman equation, 9
Bellman optimality equation, 9
Bias weights, 2

Committee, 5

Dynamic Programming (DP), 24, 39

Eligibility trace, 28

Flow model, 12

Markov Decision Process (MDP), 6, 16, 31
Markov property, 6, 16
Model, 6
Monte Carlo, 24

Neural Network, 1

Optimal policy, 9

Poisson equation, 9
Policy, 6
Policy evaluation, 21
Policy improvement, 22
Policy iteration, 24

Reinforcement learning, 6
Reward function, 6, 20
Reward-function, 7
Road capacity, 11
Route-choice model, 11

State, 18, 31
State-Space Neural Network (SSNN), 3
State-value function, 7
Steady-state probability, 20
Stochastic process, 16, 18
Supervised learning, 1

TD(λ), 27
Temporal Difference learning (TD), 24, 26, 40
Traffic congestion, 11
Traffic-demand model, 11
Transition probability, 7, 19, 32

Unichain, 7

Value function, 6, 7
Value iteration, 24