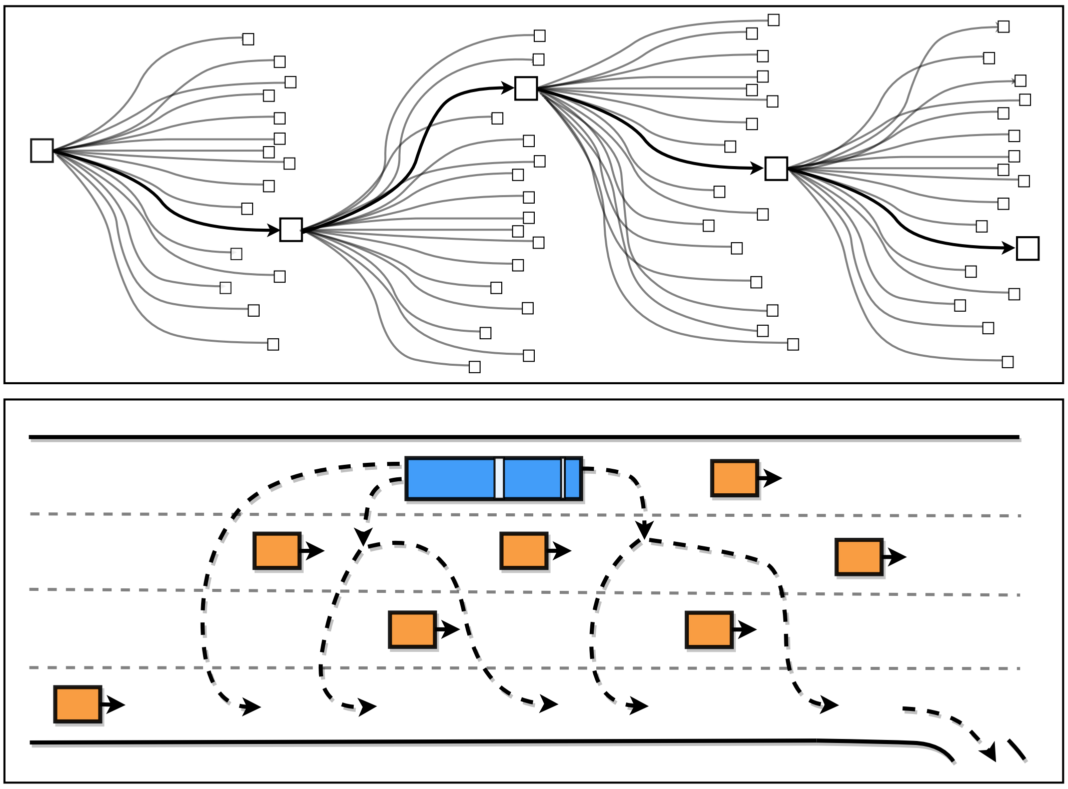
CHALMERS
UNIVERSITY OF TECHNOLOGY



Tactical Decision-Making for Highway Driving

Decision-Making in a Partially Observable Environment Using Monte Carlo Tree Search Methods

Master's thesis in engineering mathematics and computational science

ANDERS NORDMARK
OLIVER SUNDELL

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018:MVEX03

Tactical Decision-Making for Highway Driving

Decision-Making in a Partially Observable Environment Using Monte Carlo Tree Search Methods

ANDERS NORDMARK

OLIVER SUNDELL

Department of Mathematical Sciences
Division of Mathematical Optimization
Chalmers University of Technology
Gothenburg, Sweden 2018

Tactical Decision-Making for Highway Driving
Decision-Making in a Partially Observable Environment Using Monte Carlo Tree Search Methods
ANDERS NORDMARK
OLIVER SUNDELL

Supervisors:	Peter Nilsson, Volvo Trucks
	Carl-Johan Hoel, Volvo Trucks
	Peter Lindroth, Volvo Trucks
Examiner:	Michael Patriksson, Mathematical Sciences

Master's Thesis 2018:MVEX03
Department of Mathematical Sciences
Division of Mathematical Optimization
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: **Top:** A simple visualization of a tree search algorithm. **Bottom:** A decision-making problem for an automated truck during highway driving.

Gothenburg, Sweden 2018

Tactical Decision-Making for Highway Driving
Decision-Making in a Partially Observable Environment Using Monte Carlo Tree Search Methods
Anders Nordmark
Oliver Sundell
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

This thesis investigates three different Monte Carlo tree search (MCTS) algorithms for optimizing tactical decision-making during highway driving. The optimization problem was expressed in a partially observable Markov decision process (POMDP) framework, where the behaviors of the surrounding vehicles were modeled as non-observable variables. The motion of the vehicles were governed by a generative model, which used two conventional driver models; the intelligent driver model (IDM) and minimizing overall braking induced by lane changes (MOBIL). These models together contain eight parameters for each vehicle which estimate a vehicle's behaviour with respect to its longitudinal motion and lane changes. These eight non-observable parameters were inferred by a particle filter. The algorithms were tested in a simulated environment, where the objective was to change lanes to reach an exit ramp in dense highway traffic. The results show that the partially observable Monte Carlo planning (POMCP) based algorithms require more computational effort to reach the same performance as the MCTS based one, due to the inherent complexity of the history node trees. However, both methods are feasible to implement as an online tactical decision-making algorithm, where the less complex MCTS method performs best during simulations with limited resources.

Keywords: automation, decision-making, AutoFreight, highway driving, articulated vehicle, POMDP, particle filter, POMCP, online optimization, MCTS, lane changing.

Acknowledgements

This master's thesis work would not have been possible without the contributions from Volvo and the help we've got from Chalmers University of Technology. Firstly we want to thank Peter Nilsson, our main supervisor at Volvo, for giving us the opportunity to work with this project, and for constantly giving us help and support throughout the whole process. Secondly we want to thank our supervisor Carl-Johan Hoel, who has been a key part in the making of the thesis, for his dedication and guidance, and whom without we would have never got this far. We also want to thank our supervisor Peter Lindroth who has regularly helped us with the mathematical view of the problem. Lastly our examiner Michael Patriksson deserves our big gratitude for helping us keep on track, and not deviate from the academic part of the thesis.

Anders Nordmark and Oliver Sundell, Gothenburg, June 2018

Abbreviations

ACC	Adaptive Cruise Control
DPW	Double Progressive Widening
IDM	Intelligent Driver Model
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
MLMDP	Most Likely Markov Decision Process
MOBIL	Minimize Overall Braking Induced by Lane changes
OMNI	Omniscient
POMCP	Partially Observable Monte Carlo Tree Search
POMCPOW	Partially Observable Monte Carlo Tree Search with Observational Widening
POMDP	Partially Observable Markov Decision Process
PW	Progressive Widening
SAB	Static Assumed Behavior
UCB	Upper Confidence Bound
UCT	Upper Confidence Tree

Contents

Chapter 1

Introduction

This chapter introduces the concept of autonomous vehicles. It describes what is meant by tactical decision-making in autonomous vehicles, and motivates why it is a research area that is worth to investigate. It proceeds by presenting the academic background as a selection of relevant articles and literature that has been used in this thesis. Lastly, it presents the thesis' objectives, the limitations, and an outline of the thesis' structure.

1.1 Background

During recent decades the field of driver assistance and active safety systems has been steadily growing. A subset of the state-of-the-art technologies that exists in vehicles today are adaptive cruise control, blind spot detection, and automatic braking systems. This advancement is due to a lot of factors, such as road safety aspects, and economic and environmental benefits.

The concept of autonomous vehicles is not a new thing. Already in 1926 a *phantom motor car* was mentioned in the newspaper *The Milwaukee Sentinel* ([Pha18], p. 4). The 'self driving' car was controlled via radio signals from another vehicle behind to scare people into thinking it was a ghost car. Since 1926 technology has come a long way, and today it is not uncommon to encounter semi-automated vehicles on public roads, which solely rely on the information gained from its sensors.

The development of autonomous vehicles does not only apply to passenger cars, but also to commercial vehicles, such as articulated vehicles. Approximately a third of the total transportation cost for an articulated vehicle is due to the driver's salary [Bar13]. The regulations concerning the driver's work hours also greatly restrict the amount of time that the truck is able to be in service compared to an autonomous vehicle. Thus, driverless trucks would imply economic gains.

There exists many challenges in the context of an autonomous vehicle, one example being how to ensure a safe interaction with human drivers. The behaviour of human drivers can be unpredictable, and in order for the automated vehicle to drive safe it is essential to be able to predict the surrounding traffic. This thesis is a collaboration with Volvo Trucks, with the aim to develop a decision-making algorithm for autonomous trucks during highway driving. The work is also made in association with the project *AutoFreight*, where the goal is to establish an automated truck capable of transporting goods from the port of Gothenburg to an inland harbor in Borås. The level of automation of this truck will be SAE 4 [Int14], which means that the vehicle is fully automated on predetermined scenarios (for example highway driving).

1.2 Decision-making in autonomous vehicles

In an automated vehicle there is the need of decision-making for a wide range of different applications, which can be classified into different groups, illustrated by the pyramid Figure 1.1. At the top, the strategic planner takes long term decisions, such as what route to take, or what velocity to keep in order to be as fuel efficient as possible. The time between decisions in this layer is in the order of 100 seconds. The trajectory planner, at the bottom of the pyramid, is quite the opposite. It takes low level decisions, i.e., what steering angle and actuation to have in order to execute a certain maneuver. This layer has to be updated about 50 times per second in order for the maneuvers to be as safe and smooth as possible.

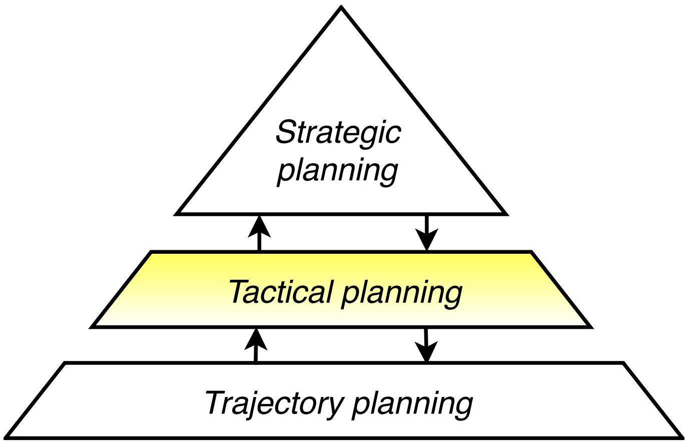


Figure 1.1: Different levels of decision-making in an automated vehicle.

A key challenge for automated vehicles is the tactical decision-making. Somewhat simplified, tactical decision-making refers to selecting a high-level maneuver (such as *maintain lane* or *lane change*) that fulfill tactical and possibly also strategical driving goals. The update frequency of this layer needs to be about once per second, and typically considers a future time horizon of about 10–20 seconds. At the tactical

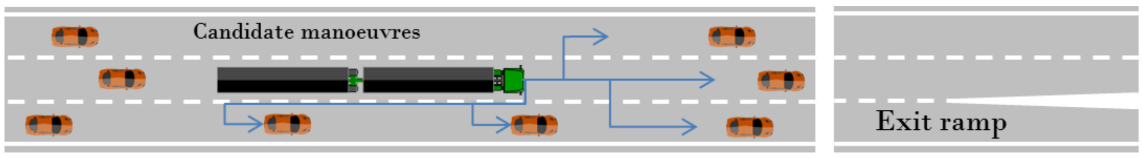


Figure 1.2: Illustration of a typical highway traffic situation for a long articulated heavy combination vehicle.

decision layer an optimal decision can be taken with respect to a variety of criteria, such as the properties of the road and traffic safety, and should fulfill the requests from the strategic planning layer. Decisions on the tactical level should also, to some extent (it doesn't need as good dynamical models as the *Trajectory planner*), be done with respect to the vehicle's dynamics and motion capabilities, which can be very complex for a heavy articulated combination vehicle [Nil17].

The goal of this thesis is to investigate algorithms that are able to take tactical decisions for a heavy articulated combination vehicle. As an example consider the traffic situation in Figure 1.2. In this scenario the truck is driving on a three-lane highway and is surrounded by other vehicles in both the current and neighboring lanes. To accomplish a lane change in this scenario the truck must either wait for a sufficiently large gap to emerge, or indicate a lane change and rely on the cooperation from the surrounding traffic.

1.3 Previous work

The articles reviewed in this section are presented in the order which they appeared during the literature study. Thus, the articles presented in the latter paragraphs are of more weight to the report.

In an article by Ardelt et al. [ACK12] they present an approach for modelling lane changes during highway driving using a deterministic state machine. The method uses a hierarchical decision process tree that validates maneuvers given their current feasibility.

Brechtel et al. [BGD11] modelled decision-making during highway driving as a *Markov decision process* (MDP), where the states are based directly upon the continuous traffic variables, such as the vehicles' positions and velocities. The same authors also show in [BGD14] by reformulating the problem as a *partially observable Markov decision process* (POMDP) that they are able to model sensor uncertainties. To render the problem feasible to solve, the continuous state problem is discretized and solved by a learning algorithm.

Ulbrich & Maurer [UM13] show how to efficiently model sensor limitations by using a signal processing network. The network projects the continuous state variables onto a discrete high-level state space. The state transition matrices are modelled to fit real data. The resulting POMDP is then solved by a branch-and-bound tree search algorithm. In another paper by the same authors ([UM15b]) they extend their work by using an uncertain mixed-integer state space combined with a sophisticated Bayesian network that provides a rigorous situation assessment. This measurement model is explained in further detail in the article [UM15a].

Silver & Veness [SV10] propose an iterative algorithm called *Partially Observable Monte Carlo Planning* (POMCP), which is an extension of the *Monte Carlo tree search* (MCTS) algorithm to partially observable problems. The authors show that the POMCP algorithm is able to better solve POMDPs defined over large discrete state spaces. A strength of the MCTS algorithms is that they only rely on a generative model of the underlying MDP. This relaxes the dependency of compact representations of the transition and observation functions. Bouton et al. [BCK17] use POMCP to solve the decision-making problem of navigating through an urban intersection environment. To formulate the POMDP the transition and observation noise are modelled as Gaussian processes. To accommodate for the continuous observation space they combine the POMCP algorithm with *Progressive Widening* (PW), that forces revisitation of already explored nodes in order to reduce the exponential growth of the search tree. The same method is used in an article by Sunberg & Kochenderfer [SHK17], where they model a lane changing-problem during highway driving. However, this article studies the importance of inferring a driver's intentions in order to make accurate traffic predictions, and as a result act less impeding on the traffic flow. To model the drivers they combine the *Intelligent Driver Model* (IDM) [THH00] and the *Minimizing Overall Braking decelerations Induced by Lane changes*-model (MOBIL) [KTH07], and estimate driver intentions with a weighted particle filter.

In an article by Sunberg & Kochenderfer [SK17] they extend the POMCP-algorithm to *partially observable Monte Carlo planning with observational widening* (POMCPOW) in order to be able to better solve problems defined on continuous observation spaces. They show that weighing the observation nodes in the POMCP tree results in a more powerful tree search. The algorithm is evaluated on three different benchmarking problems containing continuous observation spaces, and is compared to other similar algorithms, where the POMCPOW algorithm performs the best.

1.4 Thesis objectives

As previously mentioned, the goal of the *AutoFreight* project is to be able to drive autonomously from Gothenburg to Borås. This route consists mainly of highway driving, including entry and exit ramps. This thesis therefore aims at aiding the AutoFreight project in formulating and solving the tactical decision-making problem during highway driving. A request from Volvo was to solve the decision-making problem without the use of any machine learning techniques. Therefore, the methods used during the thesis is inspired by the articles [SHK17] and [SK17]. Here the problem is formulated as a POMDP, since it provides a rigorous and flexible framework able to account for sensor uncertainties and non-observable variables, such as the intent of other drivers, which they establish to be crucial to model in order to make accurate predictions. The problem is then solved by using various Monte Carlo tree search algorithms. Another property of a tactical decision-making algorithm is that it has to be computationally efficient, since a traffic situation can change rapidly. Thus it is of interest in the AutoFreight project to have an algorithm able to evaluate decisions online, i.e., take decisions in real-time. This is possible to do with an MCTS-algorithm and it is often used in applications where time is of the essence[1]. In the article [SHK17] a simulation model was used, which considers an action space containing a set of accelerations and decelerations. However, in this thesis we want the algorithm to output high level actions, which is why the action space will consist of control inputs to an adaptive cruise control (ACC).

The thesis objectives can be summarised as follows:

- Formulate tactical decision making for an articulated heavy vehicle in highway operation as a partially observable Markov decision process (POMDP).
- Solve the POMDP formulation mathematically[2].
- Acquire a policy through online optimisation.

1.5 Limitations

This work is presented, subject to the following limitations:

- Only decision-making during highway driving is considered.
- Due to the difficulty of verifying the solutions from black box techniques, the problem formulation is solved without the use of machine learning.
- The surrounding vehicles were not using, or reacting to turn indicator lights.
- During simulation, road curvature and road incline was neglected.
- No external/commercial simulation environment has been used for traffic simulation.
- The truck has a full visibility range of 100 meters (a modern Tesla car has a visibility range at up to 250 meters [Tab17]).

[1]The MCTS-algorithm has been used to program AI's in board games such as chess and Go, and also real-time applications such as Atari computer games.

[2]In this context 'mathematically' refers to the use of comprehensible and transparent optimisation methods, i.e. avoiding the use of any black-box techniques.

- The inputs to the decision-making algorithm are classified objects with positions and velocities.
- There is no sensor noise in the positions and velocities of all other traffic participants. This is since typical sensor errors are relatively small compared to the errors generated when predicting vehicle motion.

1.6 Outline

This is a brief outline of the five remaining chapters of this thesis: (The most essential parts of the theory is being referenced).

- **Theory** shows how a decision-making problem can be divided into two parts, where the first part is the *environment*. To model the environment, two frameworks are presented: the Markov decision process (MDP), and the partially observable Markov decision process (POMDP). During the implementation, later in the thesis, only the the POMDP framework (2.2.2) will be used.

  The second part of a decision-making problem is the *agent*, which can be divided into two smaller components as well: the *interpreter*, and the *solver*. Two different interpreters are presented: a Bayesian update, and a particle filter (2.3.1). A number of possible solvers are presented, where the implementation consists of different versions of the commonly used Monte Carlo tree search (MCTS) (2.4).
- **Methodology** presents how to formulate highway driving as a decision making problem, and a POMDP. The chapter then presents the different algorithms that will be used as tactical decision-makers. Lastly the highway driving scenario is presented, which were used to compare the performance of the algorithms.
- **Results** presents how the performance of a MCTS scales with the number of searches. It then compares the results from all the algorithms from the scenario. Lastly the chapter shows an analysis of the time complexity of the different algorithms.
- **Discussion** analyses the results from the previous chapter. It also discusses the advantages and disadvantages of our implementation of the problem.
- **Conclusion and future work** connects the results to the thesis' objectives, and concludes what should be investigated in order to further develop the method.

Chapter 2

Theory

This chapter introduces the concept of decision-making and its key components—the environment and the agent. The chapter also presents a variety of methods that are related to solving decision making problems. A lot of the content aims at familiarizing the reader with decision making problems, and for readers with much knowledge about POMDPs it might be possible to skip directly to Section 2.4, where the methods used in this thesis is presented. Lastly, at the end of the chapter there is a short summary of the proposed methods and a justification of our choice of methodology.

2.1 Decision-making

Decision-making can be seen as the process of selecting a course of actions among several alternative possibilities. In decision-making the entity that executes the action is referred to as the agent. The agent may represent something physical, like a human or a robot, or it may be a nonphysical entity, like a support system in a vehicle. A decision-making problem is usually defined as finding the most beneficial action to execute based on the situation and how the agent interacts with its environment. Generally the solution to a decision-making problem can be divided into two sub-processes: the *Environment* and the *Agent*. The agent can be divided further into two parts: the *Interpreter* and the *Solver*. (See Figure 2.1). The *Environment* contains a model of the agent's surroundings, how the agent interact with it, and the rewards the agent receives for selecting different actions. In order for the agent to observe and process its surroundings the agent has an *Interpreter* that models the agent's sensors. Lastly the *Solver* constitutes the brain in the agent, and is responsible for taking decisions about what actions to execute. These three concepts will be explained in detail during the next two sections.

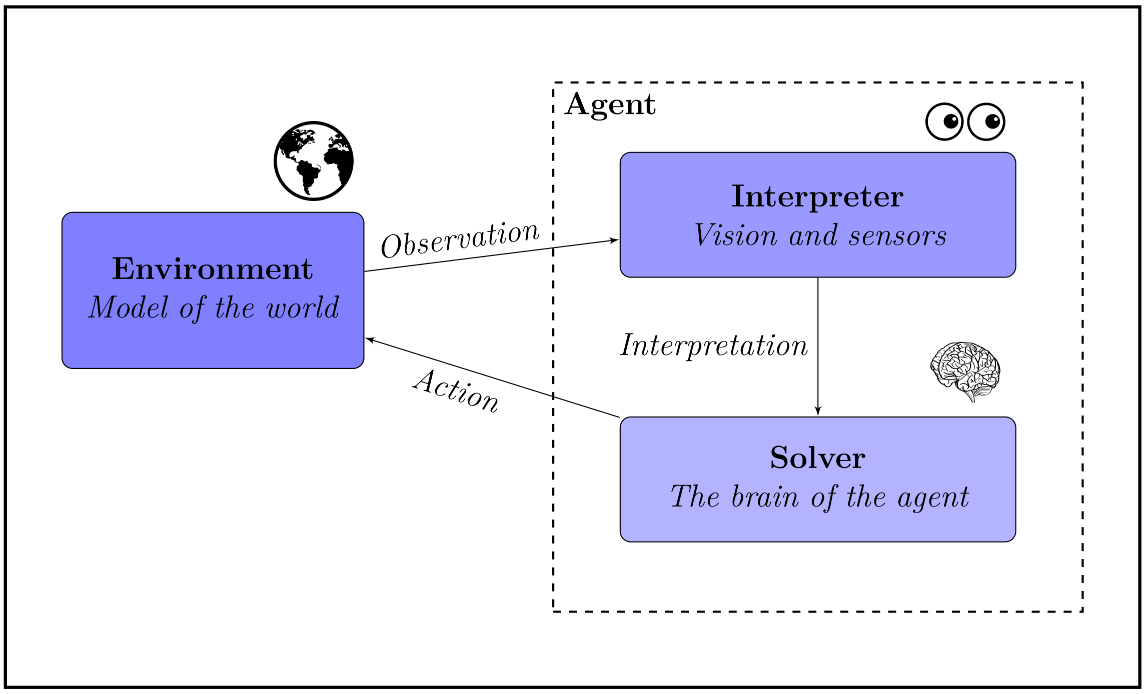
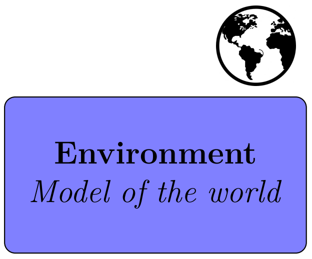


Figure 2.1: A visualization of a decision-making problem and how an agent interacts with it. The environment models the agent's surroundings, the interpreter models how the agent reasons about its observations and sensor systems, and the solver represents the agent's brain and how it evaluates the different decisions.

2.2 Frameworks for modelling the agent's environment



The problem of finding an optimal policy can be arbitrarily complex depending on the problem. As an example consider modelling chess. In chess, all the information that the agent needs in order to decide on what piece to move is given by the current state of the chess board, meaning that there is no noise or hidden variables involved. This means that the interpreter is superfluous, since the observations contain all the available information. In most real world applications, though, some information about the environment is not observable. As an example consider modelling highway driving with manually driven vehicles. There are presumably a lot of underlying non-measurable variables that are responsible for the reactions of a surrounding vehicle (such as for example the skill, the mood, and the intentions of the driver). These variables might prove to be very important to model in order to make accurate predictions. Thus, in order to model tactical decision-making, it is important to have a general mathematical framework that is able to account for non observable information. This section present the widely used *Markov decision process*- framework (MDP), used for modelling decision-making problems similar to chess, and its extension to noisy and partially observable variables called *partially observable Markov decision processes* (POMDP).

2.2.1 Markov decision processes

An MDP is a mathematical framework for modelling decision-making under uncertainty. It provides a simple and easily applicable framework able to model anything from simple discrete problems to more abstract and continuous real life problems [Koc15].

Definition

An MDP is governed by the following 5-tuple:

- $\mathcal{S}$ defines a set of states.
- $\mathcal{A}$ defines a set of actions that an agent can take.
- $\mathbf{T}(s', a, s) = P(s'|a, s)$ defines the probability to end up in state s' when taking action a in state s.
- $\mathbf{R}(s, a) \in \mathcal{R}$ defines a scalar-valued reward for taking a specific action in a specific state.
- $\gamma \in [0, 1)$ is a discount factor that takes into account that actions taken now are more important than actions taken in the future.

An MDP defines a time discrete decision-making process where an agent is located in one of the states, which are seen as elements of the state space $\mathcal{S}$: a set of all possible states. In each time step the agent can choose to perform an action from the action space $\mathcal{A}$. After selecting an action the agent's state is updated—a process governed by the state transition function $\mathbf{T}$. The state transitions are defined as conditional probabilities of the *current* state and an action, which means that a certain action a_1 taken in a specific state s_1 might not always lead to the same state s_2. The transition function in an MDP is *memoryless* and doesn't need any information about the past, only the current state and action. It means that it assumes the so called *Markov property*; the future is only dependent on the present.

The main objective when studying MDPs is to find the optimal action given that the agent is currently in state s. Such a mapping is referred to as a policy $\pi(s)$. The optimal policy $\pi^*(s)$ is defined as the policy that maximizes the expected accumulated rewards from state s. By using the elements of the previously defined tuple the reward returned by following an arbitrary policy can be expressed as:

$$R = \sum_{t=0}^{\infty} \gamma^t \mathbf{R}(s_t, \pi(s_t)), \tag{2.1}$$

which converges over an infinite time horizon since $\gamma \in [0, 1)$. However, in most applications it's better to consider a finite time horizon, T. The optimal policy for this horizon can be defined as:

$$\pi_T^* = \underset{\pi}{\operatorname{argmax}} \operatorname{E} \left[\sum_{t=0}^{T} \gamma^t \mathbf{R}(s_t, \pi(s_t)) \right], \tag{2.2}$$

where $\operatorname{E}[\cdot]$ denotes the expected value, and π_T^* is the optimal policy for time horizon T.

Grid world example

Imagine a game where an agent is placed somewhere on a discrete finite grid, like in Figure 2.2. Within this grid there is a treasure, indicated by a golden square. There are also obstacles in the form of trees and walls, which are depicted as green squares and thick black lines, respectively. The purpose of the game is for the agent to reach the treasure as *fast* as possible by traversing the grid. The agent can only travel to one of the adjacent squares and there is noise added to its movement, meaning that there is a small probability that the agent ends up in a random neighbouring square. The game ends either when the agent reaches the treasure or accidentally steps into one of the deadly red squares. Thus the agent has to plan its actions to not inflict too much danger. Let's formulate this problem as an MPD:

- The state space $\mathcal{S}$ consists of all the squares in the grid.
- The action space $\mathcal{A}$ is made up by 4 actions: *up, down, left, right*, that defines the actions the agent can select in each time step.
- The transition function $\mathbf{T}$ governs the probabilistic movement of the agent. Let's say, for example, that if the agent wants to move in a direction (e.g., *up*), there is a 1/10 probability that it will instead move in another.
- The reward function for the problem consists of 3 different objectives: avoid the red squares, find the treasure, and do so as quickly as possible. An example of a reward function could be that the agent gets $+100$ points if it finds the treasure, and is penalized by -10 points for stepping on a red square. To model the importance of reaching the treasure as quickly as possible each step penalizes the agent with -1 points.

By a quick inspection of the problem there are two possible solutions to this problem. Either the agent decides to enter next to the red squares. In this strategy the path to

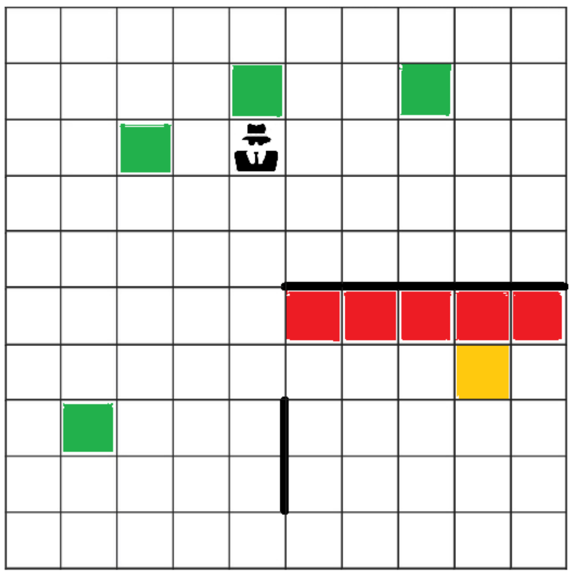


Figure 2.2: *An example of a decision-making problem which can be described by an MDP. The agent is located in a grid world where the goal is to reach the golden square. The agent moves probabilistically on the grid and has to avoid stepping onto one of the red squares. How should the agent act in order to reach the treasure?*

the treasure chest is the shortest, but at the same time it is also risky—the transitions are probabilistic, and thus the agent might unwillingly stumble off its path and step onto a red square. The other strategy is to simply walk around the wall, which is a longer path, but one that almost guarantees finding the chest. Which strategy is the best one, meaning what policy π maximizes $R_T(\pi)$ according to (2.2), depends on the parameters of the transition- and reward functions. Methods for solving these kinds of problems will be presented later in the *Solver* section 2.3.2.

2.2.2 Partially observable Markov decision processes

The previously described problem was a decision-making problem where the only uncertainty involved was based on the transition function. However, in this problem the agent had *perfect observability*, i.e., complete certainty about its environment and thus also its current state. This is rarely the case when modelling real world problems, since they often contain some measurement errors or non-observable variables. A POMDP is an extension of the MDP where the agent does not have perfect knowledge about the current state [Koc15].

Definition

A POMDP is governed by a 7-tuple, which expands the MDP model with two additional elements:

- $\mathcal{Z}$ defines a set of observations z (e.g environment measurements).
- $\mathbf{O}(s', a, z) = P(z|s', a)$ defines the probability of receiving the observation z given that the system ended up in state s' after the agent performed the action a.

The observations contain information about the agent's current state. This information is interpreted by the agent to create a *distribution* over what possible states it might be in, called a *belief* state $b(s)$. Thus the POMDP incorporates a belief space $\mathcal{B}$, a continuous space that defines a set of all possible belief distributions. Thus, the belief state representation provides a framework for modelling decision-making processes under uncertainty. The optimization of POMDPs are thus computationally complex; the number of possible belief states are by definition infinite. However, there are analytical methods to solve POMDPs, but they have a complexity proportional to the size of the state space which often makes them infeasible ([SV10]).

A partially observable grid world example

As an example of a POMDP we extend the previous grid world example (see Section 2.2.1) with partial observability by modelling an agent that uses sensors to observe its environment. These sensors have a limited range, and thus the agent can only observe the neighbouring squares of the grid. For simplicity assume that the agent's sensors does not contain any noise, which means that the observation function is deterministic. This problem formulation implies that the agent is rarely sure of where it is, and to solve the problem it has to maintain and update a belief distribution over its possible states.

The problem is visualized in Figure 2.3 where the agent was initialized same as before. The small 3x3 picture in the middle shows an example of what an observation might look like, and by solely using this information it's not possible to determine where on the grid the agent is located. But, since the observation function is not probabilistic it can only be located at four different places: the four squares that are located below a green square. In this case the agent's belief distribution is zero over all states except for when the agent is located on these fours squares. This belief state is depicted in the right panel, where the grey agents indicate the belief distribution. Given no previous history the probability over these states should be equal, i.e., $p_i = \frac{1}{4}$. At some point the agent may observe something that makes it completely sure of where it is, and similarly since the transition function is noisy the agent can actually lose track of its position again. In a POMDP-representation it is thus much more difficult to find an optimal policy, since a good solution also involves taking steps just to improve the belief distribution.

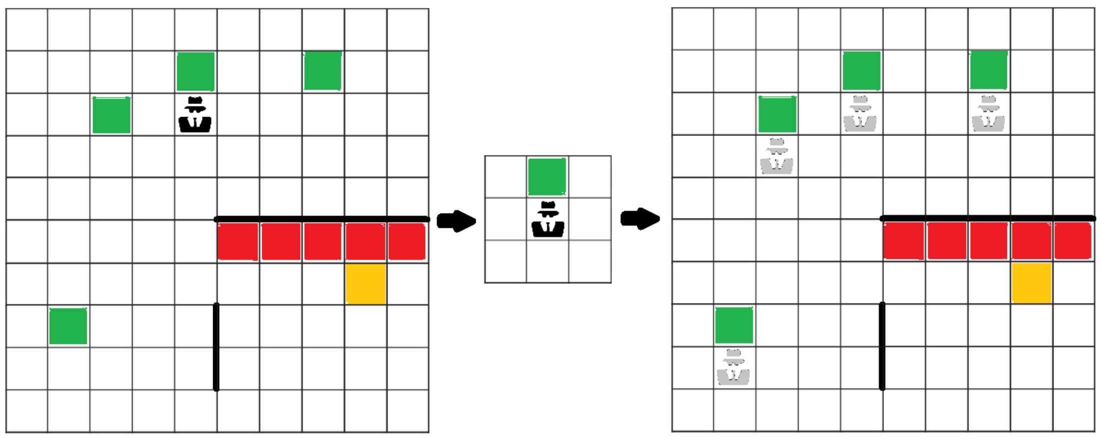
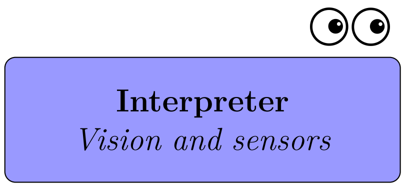


Figure 2.3: *An example of a partially observable decision-making problem which can be described by an POMDP. The agent is located in a grid world where the goal is to reach the golden square. The agent moves probabilistically on the grid and has to avoid stepping onto one of the red squares. The square in the middle represents the agents observation given from its sensors, and by interpreting this observation the agent can create a distribution of possible states seen in the right picture. To update its belief distribution the agent can use this information together with its prior knowledge. How should the agent act in order to reach the treasure?*

2.3 The agent

This section presents the two key components that compose an agent: the *Interpreter* and the *Solver*.

2.3.1 Interpretation and belief states



In order to maintain a realistic depiction of the underlying state, the agent in a POMDP must rely on its observations, and by using this information construct a belief distribution over the state space. When a new observation is gained the agent may combine this knowledge with the *prior* belief distribution and create a *posterior* distribution, a process referred to as *filtering*. Filtering has the nice property that the more observations the agent has received the more accurate the posterior distribution becomes. In this section two filtering methods are considered; the analytical Bayesian belief update ([RPPCd08]) and the stochastic particle filtering method ([SHK17]).

Bayesian belief update

The Bayesian rule states the following:

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{P(B)} = \frac{P(B|A_i)P(A_i)}{\sum_j P(B|A_j)P(A_j)}, \tag{2.3}$$

where P(A_i) is the prior probability of event A_i being true, P($A_i|B$) is the posterior probability of A_i being true if event B was observed, and P($B|A_i$) is the conditional probability of event B occurring given that A_i is true. This method computes the posterior probability of observing event A_i given that event B occurred.

This method can also be used to update the belief in the context of POMDPs. For a belief update this is expressed as

$$b_{t+1}(s') = \frac{\mathbf{O}(s', a_t, z_{t+1}) \sum_{s \in S} \mathbf{T}(s, a_t, s') b_t(s)}{\sum_{s'' \in S} \mathbf{O}(s'', a_t, z_{t+1}) \sum_{s \in S} \mathbf{T}(s, a_t, s'') b_t(s)}, \tag{2.4}$$

where $\mathbf{O}(s', a, z)$ is the observation function, $\mathbf{T}(s, a_t, s')$ is the transition function, and b_t(s) and b_{t+1}(s) are the prior and posterior belief distributions respectively. The sum in the numerator shifts the previous belief distribution with the transition function, and this is then multiplied by the how likely the belief is with respect to the most recent observation. To create the posterior distribution this quantity is normalized by the denominator which sums over all possible outcomes.

This posterior distribution provides the most accurate belief state. However, since the formula needs to sum over all possible states, it is a method that is highly dependent on the size of the state space. Therefore it might be computationally infeasible ([SV10]). Moreover in complex problems it might be impossible to find compact observation and transition functions that represent the dynamics of the problem.

Particle filter methods

Particle filters are Monte Carlo-sampling based algorithms used in signal processing and Bayesian statistical inference in order to solve filtering problems. There are a lot of different particle filters, but the one explained in this section is based on the one used in [SV10]. The authors present a method of solving POMDPs defined over large state spaces where an analytical filtering method is computationally infeasible. The particle filter relies on a black box simulator of the process, called a generative model. The generative model $(r, s', o') \sim G(a, s)$ simulates the dynamics of the underlying process, where s is a state, a an action, s' is one of the possible resulting states, r is the reward for selecting the action, and o' is the observation from state s'. This is one of the main benefits of the particle filter—it doesn't require any explicit observation and transition functions, but rather a way to simulate them, which is less constraining.

The method approximates the belief distribution by a finite set of particles, where each particle represents an independent state. Denote particle i at time t by $\theta_i^t \in \mathcal{S}$, the approximated belief distribution can then be expressed as

$$\hat{b}(s, h_t) = \frac{1}{M} \sum_{i=1}^{M} \delta_{s\theta_i^t}, \tag{2.5}$$

where δ_{ij} is the Kronecker delta function which is 1 whenever $s = \theta_i^t$ and 0 otherwise, M is the total number of particles and h_t is the history of actions and observations in time step t. Given an initial belief distribution $\mathcal{I}$ an approximated belief can be constructed by sampling M particles, i.e., $\theta_i^{t=0} \sim \mathcal{I}$, $1 \leq i \leq M$. The initial distribution can either be set by using prior knowledge or be set uniformly. Each time a real action $\tilde{a}$ is executed and a real observation $\tilde{o}$ is obtained the particle filter can be updated. The posterior particle distribution is created iteratively with Monte Carlo simulations by using the generative model to replicate the observed action-observation pair. This is done by randomly sampling from the particle set, which corresponds to sampling states $\hat{s}$ from the approximate belief distribution. A sampled state is then input to the generative model which performs the action $\tilde{a}$ and produces an observation $\hat{o}'$, $(r, \hat{s}', \hat{o}') \sim G(\tilde{a}, \hat{s})$. If $\hat{o}' = \tilde{o}$ the particle $\tilde{o}$ is added to the posterior belief, since if $o' = \tilde{o}$ the sampled particle described one of the possible states. This process is repeated until M particles have been added to the new belief which then represents an updated posterior belief.

Like all Monte Carlo algorithms the more samples that are produced the better the approximation. It can be shown that the particle filter has asymptotic properties, meaning that $\hat{b}(s, h_t) \to b(s, h_t)$, as $M \to \infty$. However, a common problem with particle filters is something called *particle deprivation*. This is a result from the fact that the number of states contained in the particle filter may decrease; during a single update the same particle might be selected multiple times. Thus there is a probability that as $t \to \infty$ the particle collection may collapse to a belief state containing a single particle that in turn contains an incorrect state. This problem is solved by introducing noise in the resampling procedure (similar to the mutation operator used in genetic algorithms). Each time that the algorithm selects a particle from the set it has a small probability of adding some noise to it, and thus changing the particle's state. New particles will therefore continuously be added to the particle filter, which if tuned correctly will result in a much more efficient algorithm able to find a good approximate belief states given a history h_t.

However, this resampling procedure will only work in a discrete state and observation - space because the probability of sampling the same observation twice in a continuous space is zero. This means that the procedure described above will not converge; $P(o' = \tilde{o}) = 0$. But, by slightly modifying the resampling procedure one can still use this method in continuous observation spaces. One way to do this is to keep an approximate belief distribution in the form of a *weighted* particle collection. The weights indicate the conditional probability of observing the real observation given the particles' state, which can be expressed as

$$w_i = P(o|\hat{s}_i, a), \tag{2.6}$$

where $\hat{s}_i$ is the state given by particle θ_i and w_i is its weight. The weights are best set by using the observation function [SK17], since this is explicitly defined as $O(o, s, a) = P(o|s, a)$. But as previously mentioned the observation function might not always have a compact representation. If this is the case one can perform Monte-Carlo sampling from the generative model and combine this with a relevant measure of distance between observations. Let $\|.\|_{obs}$ denote such a measure, then the weight of a particle given an observation is given by

$$w_i = \exp\left(-\frac{\|o - \hat{o}_i\|^2_{obs}}{2\sigma^2_{obs}}\right) \approx P(o|\hat{s}_i, a), \tag{2.7}$$

where $\hat{o} \sim G(\hat{s}, a)$ and $\hat{s}$ is the state given by particle θ_i and σ_{obs} is a normalization constant. These weights can be interpret as how likely the state was to have generated the real observation, which in the POMDP formalism roughly corresponds to the interpretation of a belief state. This procedure is used in the article [SHK17] to weigh the relative likelihood of drivers' intentions given their acceleration outputs.

2.3.2 Methods for solving decision-making problems

Solver
The brain of the agent

The solver's role in the agent is to use the information given by the interpreter to find the best possible policy to execute. In some cases calculations can be done in advance, i.e., offline. In a chess game for example, some decision-makers engines are able to train offline and then use the gained knowledge to find good actions online ([SHS$^+$17]). Another way for the solver to find a good policy is by doing the calculations purely online. There are lots of different ways to design a decision maker in an agent. In this section a couple of different methods and approaches is presented, which can all be found in the book [Koc15].

Dynamic Programming

A common way to find an optimal policy in MDPs and POMDPs is dynamic programming. Within this area there are two popular methods: value iteration and policy iteration. Value iteration uses a utility function (2.8) to find the expected reward for following an arbitrary policy $\pi(s)$, and to find the optimal policy it maximizes the equation via dynamic programming of the Bellman equation:

$$U_t^{\pi}(s) = R\left(s, \pi(s)\right) + \gamma \sum_{s'} T\left(s', \pi(s), s, \right) U_{t-1}^{\pi}(s'). \tag{2.8}$$

The optimal value of the utility function is gained by following the optimal policy, which for time horizon n can be formulated as

$$U_n(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s, a) U_{n-1}(s') \right). \tag{2.9}$$

Since this equation satisfies the Bellman equation (which is a necessary condition for optimality in dynamic programming) it can be solved. The problem when trying to apply the algorithm to solve an MDP is that the computational complexity is proportional to the size of the state space. Thus a problem defined over large or infinite state space would render these algorithms inapplicable, a problem which also applies to policy iteration [SV10].

Direct Policy Searches

Another approach when solving a decision-making problem is to search through the space of all possible decision strategies and utilize a performance measure to find the optimal policy. Even though the state space might be huge, the space of possible policies might be low dimensional. The main objective in direct policy search is to maximize the objective function,

$$V(\lambda) = \sum_s b(s) U^{\pi_\lambda}(s), \tag{2.10}$$

where λ is a parametrized policy, b(s) is a distribution over the initial state, and U^{π_λ} is the expected value for following policy π_λ.

There exist a lot of different methods for maximizing the objective function, such as for example gradient descent or evolutionary algorithms. Generally the performance of a direct policy search is dependent on the size of the space of decision strategies, and the amount of local optima there exists in the performance measure. And thus for the same reasons as with policy and value iteration these methods wont be applied.

Machine Learning

Machine learning covers a wide range of computer science methods where the goal is to 'learn' and find patterns in data. In recent times machine learning has had a lot of applications in decision-making problems (for example in the board game GO [SHS$^+$17]), where this generally involves training a neural network based on simulation data. In online, real time applications this is a smart approach, since the vast majority of computational effort is used offline during training.

Supervised learning is an example of machine learning where the agent is improving over time, on a specified task, by training on a set of provided data. This is a widely used method for problems such as classification problems and pattern recognition. For decision-making applications the data on which the agent is training on is based on how an *expert* would handle different situations, and the agent gets reward for being as close to the *expert's* behaviour as possible. However, there are some drawbacks with this method. For example, it's not always possible to get hold of an expert's judgment, and it might be hard to know if the training data is actually based on optimal play. This method also implies that the skill of the agent can not surpass that of the expert. Furthermore, it's also difficult to determine whether the agent only performs well on the data used for training and validation, or if the agent has developed a general understanding for the problem at hand.

Another machine learning technique is **reinforcement learning**, where the agent instead learns while interacting with the world. The only thing the agent needs during training is a reward function to measure its performance. The agent learns what actions are optimal in a given situation without having to consult an expert. This can be extremely useful if there exists limited knowledge of how to act optimally. The previously mentioned chess engine that well surpassed human capabilities uses this type of machine learning ([SHS$^+$17]).

Probably the biggest drawbacks of using these methods is that it is practically impossible to verify why the agent behaves in a certain manner. The system is basically a black box, and even if the agent performs really well in the test environment, it is hard to know its shortcomings when tested in a real environment. Thus, due to our limitation not to use any type of black-box techniques, this method will not be used.

Online methods

Most of the methods presented so far have been methods that partially need to be executed *offline*. In large and complex state spaces this might be impractical since to calculate the optimal policy for every possible state beforehand is too time demanding. In these cases it might be better to find the optimal policy directly from only the current state. Below, four *tree search* methods, often used in online applications, are introduced.

A tree search method evaluates actions by constructing a tree graph from the agent's current state. The nodes in the tree represent previously visited states, and the vertices connecting the nodes represents the action taken to get there. A **forward search** is a simple tree search algorithm that builds up the entire search tree up to a time horizon d. The algorithm searches depth-first, and chooses actions from state nodes recursively until the horizon is reached. During the search the state-action pairs gets assigned values, defined by a reward function. When the whole tree has been constructed the optimal action is the vertex from the root node with the highest value. Since the whole tree must be constructed the computational complexity is $\mathcal{O}((|\mathcal{S}| \times |\mathcal{A}|)^d)$ which is very time consuming.

The **branch and bound**- algorithm is an improvement of the forward search. The difference between the two is that Branch and Bound uses domain knowledge to find upper and lower bounds on the value function. With the use of these the algorithm is able the prune the search tree. If the upper bound for a state-action pair, for example, is lower than a previously found value, then it is pointless to continue the expansion in that direction. In order to be able to prune that part a state–action pair with a higher value must already be known, and thus the order of choosing actions is important. However, in the worst-case scenario the computational complexity is the same as for forward search, and therefore this method is heavily reliant on domain knowledge.

In order to escape the high computational complexity from forward search and branch and bound, **sparse sampling** uses a specified number of children, n, that each node can have. This means that the time complexity for the algorithm is $\mathcal{O}((n \times |\mathcal{A}|)^d)$, which is still exponential in the horizon, but does not depend on the size of the state space. Sparse sampling uses a generative model G to simulate the transition- and reward functions. One advantage of this is that it's easier to sample from such a function instead of having to find explicit probabilities of the transition between states. However, since the number of children that each action node can have is limited by n, the whole search tree is not spanned. Therefore the algorithm can't be guaranteed to find an optimal solution for horizon d. This means that we are in need of some way to direct our search more efficiently through the tree, and leads us to the last method.

The **Monte Carlo tree search** (MCTS) is a powerful search method. It is an online sampling-based method, that just as *sparse sampling* uses a generative function. It has the possibility to weigh the importance of exploration and exploitation, the algorithm is run for a predetermined number of searches, and the time complexity is not exponential in the time horizon. The method is quite extensive and includes many different variations, and thus the next chapter is completely dedicated to it.

2.4 Monte Carlo tree search methods

In 2016, the computer program *AlphaGo* became the first program ever to beat a human professional player in the game Go ([SHM+16]). Because of its huge branching-factor Go is considered to be a much more difficult game than chess for a computer AI, and was long considered as impossible for a computer to master. What makes *AlphaGo* so powerful is that it combines state of the art machine learning techniques with a robust tree search algorithm—the MCTS-algorithm. This chapter is based on information from the article [BPW+12].

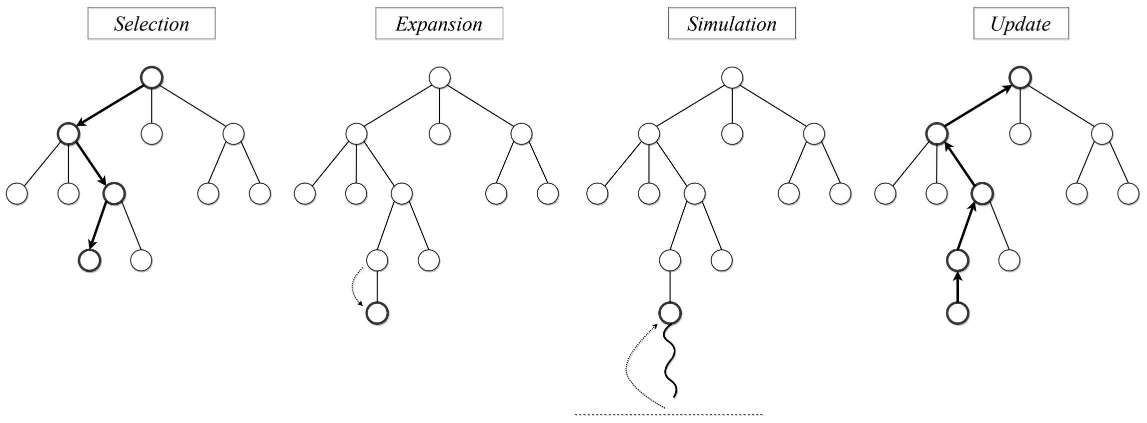


Figure 2.4: *The four sub-processes of the Monte Carlo tree search algorithm. The selection process steps through the search tree until it reaches a leaf node. The expansion step expands the tree from the leaf node found in the selection step. The simulation step then evaluates the new node by simulating a game using a rollout policy. The new results are then propagated through the branch of the tree back to the root node.*

The MCTS-algorithm uses simulations governed by the generative function, that has previously been described in the Monte Carlo belief state update section 2.3.1, to construct a search tree of possible outcomes. The generative function simulates the transition function, and is from a state and action able to output a new state and a reward. The nodes in the search tree represents previously simulated states, and the vertices represents the transition of the system when an action is chosen from a state. The root node at the top of the graph represents the current state in the decision making problem. This can be seen in Figure 2.4 where the root node is connected by three vertices to the nodes directly below. What that means is that in this particular decision problem there are only three possible actions to execute from the start state. The MCTS-algorithm consists of four sub-processes, which are performed in the following order; *selection* (select an action from a node), *expansion* (create a new node by using the transition function), *simulation* (calculate the value of the new node) and *update* (update the value of the nodes connected to the new node using back-propagation). These four steps are explained in more detail below.

The **selection** algorithm determines what action to explore given a state node in the search tree. The selection of an action can be made with respect to a couple different criteria. First the algorithm check if all possible actions have been explored, if this is not the case one of the remaining actions is randomly selected. If all actions have already been tested, the next action can be chosen by using a weighted selection. The simplest method to do this is to choose an action greedily, meaning that the algorithm always chooses the most successful child node. This strategy might sometimes work, but there is a high risk of getting stuck in a local minimum. A more sophisticated algorithm (like the upper confidence bound described below) would instead explore options by factoring in additional conditions, like the number of times an action has already been explored. The selection step is repeated recursively until it has either reached a leaf node or a state node where all actions has not yet been tried. When the selection algorithm has found such a node a new state node is created with the generative function by selecting an action. Each such step is an **expansion** of the search tree, which creates one new leaf node.

The next **simulation** step evaluates the newly expanded node. This is done by simulating into the future using a predetermined policy, a so called *rollout policy.* The actions in the rollout policy can either be chosen at random or by some prior domain knowledge. The simulations are continued either until the game is finished, or the time horizon in reached. The accumulated reward from this simulation can either be binary (win/loss), or a real number.

When a rollout is done and the accumulated reward for new node has been computed the search tree needs to be **updated**. To do this the value of this node is back propagated through the search tree, where the value of every every state encountered on the path back to the root node is effected. This means that if a good result is achieved from the new state node it will be more likely for future selection steps to return and continue to expand this branch.

Upper confidence bound algorithm

The size of the search tree is dependent on the number of state-action pairs that the underlying system consists of, and to search through the whole tree is usually computationally infeasible. To avoid this situation MCTS is often combined with the Upper Confidence Bound algorithm (UCB_1). This combination is called the Upper Confidence Tree (UCT), which provides an efficient search towards the more promising branches of the search tree.

Imagine a set of slot machines with different expected payoff. You want to maximize your total cumulative winnings, but you have no prior information about which machine that might be the best one. In each time step you get to choose a new machine, and without any insight your opening moves would naturally be to just try them all to examine which ones that seem the most promising. However, it is impossible to know if the winnings you got from a machine were because of it having a high expected payoff, or if you were just being lucky. This is an example of a problem where the UCB$_1$-algorithm provides a good solution. It balances the exploitation of promising regions with the exploration of more unknown parts. First denote playing on a machine i by action a_i (Note that in this particular example there is only one state s). Then in each time step the UCB$_1$-algorithm selects the action which maximizes the UCB-value, given by the following equation

$$UCB(s,a_i) = \tilde{Q}(s,a_i) + c\sqrt{\frac{\ln N(s)}{N(s,a_i)}}, \tag{2.11}$$

where $\tilde{Q}(s,a_i)$ is the estimated value of a state-action pair, $N(s,a_i)$ is the amount of times that action a_i has been tried from state s, $N(s) = \sum_a N(s,a_i)$ is how many times the state has been visited, and c is an exploration constant. This algorithms ensures that the first step is to sample each action once, since $UCB(s,a_i) \to \infty$ as $N(s,a_i) \to 0$. The value of c is usually determined empirically, but a rule of thumb is to set it to the same order of magnitude as the expected value of $\tilde{Q}$ ([Koc15]). The more actions are spent on action a_i, the larger the term $N(s,a_i)$ will be, which in turn decreases the exploration term for that specific action. Respectively if an action has not been tried as much, the value of $N(s,a_i)$ will be small, and thus makes the exploration term for that action larger. Therefore, as the agent chooses actions based on maximizing the UCB-value the algorithm balances exploration against exploitation. See figure 2.5 for a visualization of the UCB1 algorithm. There are alternative expansions of UCB1 that also incorporate the variance of a machine's returns to estimate $\tilde{Q}(s,a_i)$ [BPW$^+$12]. However, this will not be used during the thesis.

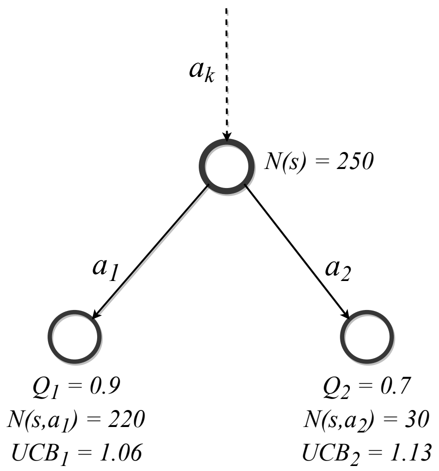


Figure 2.5: *The UCB-value of two nodes in a search tree. The most successful action so far is a_1, however, action a_2 has been sampled fewer times and might not be properly represented by its Q- value. The UCB1 algorithm uses this information and (with exploration constant, $c = 2$) would in this cases select the action a_2, even though it has a lower expected return.*

Rollout policy and heuristics

The rollout policy plays a crucial role in a MCTS algorithm. Since the purpose of the algorithm is to avoid searching through the entire state space one needs a good way to evaluate new nodes in the search tree. If there is limited knowledge about the underlying system one can simply use a rollout policy consisting of random actions. However, this will usually result in poor estimation of the nodes, and computational effort can be wasted. The better option would be to use heuristic knowledge when constructing a rollout policy - meaning that it would consist of action patterns that are more likely to show realistic behaviour.

There are also other methods to apply heuristics to MCTS. Before the graph is constructed one could for example view the tree as a large set of nodes where the number of visits in each node, N_{init}, and the value, Q_{init}, set to zero. Given some prior knowledge about the current state the nodes may instead be initialized to other values, i.e setting $\langle Q_{init}, N_{init}\rangle \neq \langle 0, 0\rangle$. An example where this could be used is during the first moves in chess. Here a MCTS solver could use a tables of good opening moves to avoid evaluating and spending computational power on known bad moves.

Progressive widening for continuous spaces

The so far considered examples have contained a finite discrete action space, but this is not necessarily the case. As an example, consider a decision-maker that's in charge of the steering in an autonomous vehicle. The possible rotation of the steering wheel may be arbitrary small, which makes the action space large. If one would construct a MCTS-algorithm of such a problem one would never be able to reach a depth larger than one action, since there for each new iteration of MCTS always is a steering angle that has not yet been tested. To solve this problem there is an algorithm called progressive widening (PW) [CD11], which evaluates the following inequality

$$k < k' N(s, a)^{\alpha}, \tag{2.12}$$

where k' and α are constants, k is the number of children of the node considered, and $N(s, a)$ is the number of times the node has been visited. If this equation holds no new node will be sampled and instead one of the existing nodes will be sampled by using the UCB1 formula. Thus, a MCTS using this algorithm will result in a progressive widening of the action space.

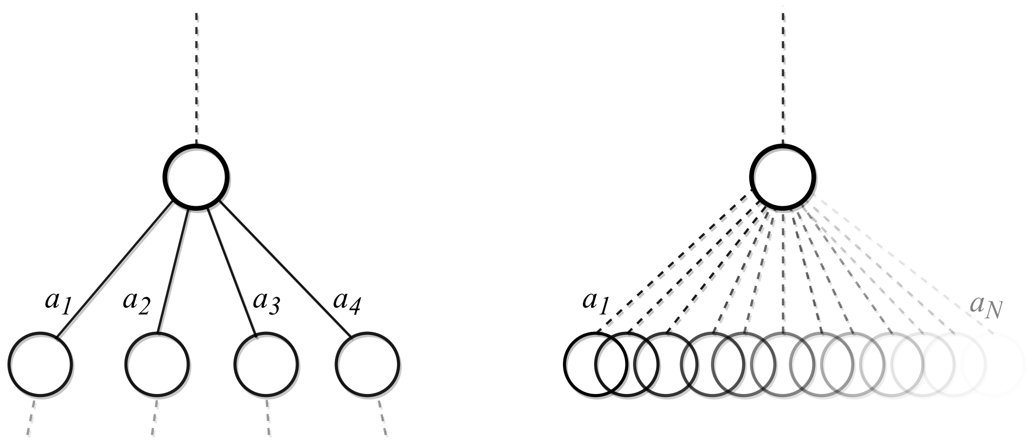


Figure 2.6: *Illustration of the progressive widening of a search tree containing an infinite action space. The right search tree will never sample the same action twice and will become very shallow. However, the search tree to the left is progressively widened and is limited to only sampling $k = 4$ child nodes. The number of child nodes for each state is one of the parameters of the progressive widening algorithm.*

As a demonstration return to the considered steering example with $k' = 4$ and $\alpha = 0$. Before any nodes have been sampled the right hand side evaluates to zero, since the number of children is zero. Thus, an action is sampled from the action space and a node is created. This process will be repeated until the MCTS sample a node for the fifth time. This time when equation 2.12 is evaluated the expression will not hold, and no new node is sampled. Instead a previously tried action will be chosen, and the algorithm continues expanding the tree from here. This example is depicted in Figure 2.6, where the left tree shows a tree search with progressive widening. With these settings one can see that each node will get a maximum of 4 child nodes. By setting $\alpha \neq 0$ (usually $\alpha \approx 0.1$) new nodes will be sampled once $N(s, a)$ is large enough. Resulting in a slow but sure expansion of the promising regions of the action space.

2.4.1 Using Monte Carlo tree search to solve POMDPs

The nature of POMDPs makes them computationally complex to solve using a regular MCTS. For each possible action the generative model will output an observation. This results in a search tree with an increased exponential growth, which is proportional to the size of the observation space. Furthermore, the states in a POMDP are represented by belief distributions. And to take a decision from a belief state, the MCTS algorithm would have to construct a tree search for all the states contained in the belief distribution. This means that the MCTS algorithm is computationally intractable if the problem is defined over a large state space.

Partially observable Monte Carlo planning

Partially observable Monte Carlo planning (POMCP) is an algorithm that expand the MCTS to handle partial observability. It solves both the problem of belief states, as well as the complication of observations in a POMDP. As an interpreter the POMCP-algorithm uses a particle filter like the one described in section 2.3.1. The belief state is thus made up by M weighted particles with weights depending on how probable they are given a recent observation [SV10].

To deal with observations during the tree search the state nodes in the POMCP-tree consist of histories, see Figure 2.7. A history is a sequence of action and observation pairs, which just as in the MCTS, has a counter of how many times that history has previously been visited. The action nodes also has a value that describes the average amount of reward that has been returned by selecting that history branch.

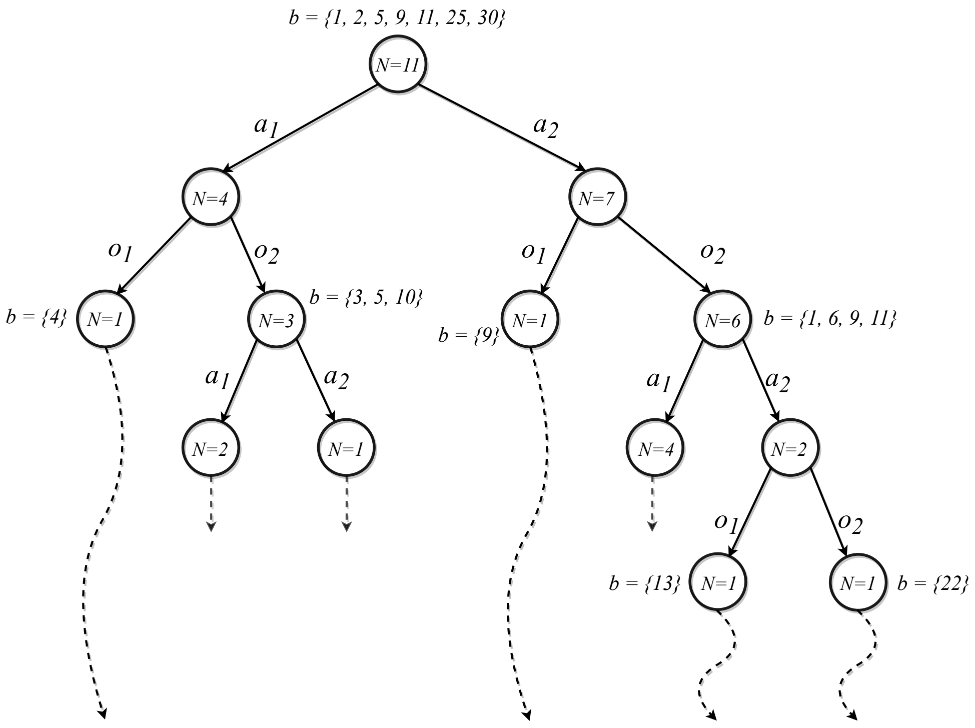


Figure 2.7: *The POMCP algorithm uses histories to reason about the action observation pairs obtained when simulating the POMDP. The belief states are represented by Monte Carlo sampling from a set of particles, which approximates the belief distribution.*

The tree search in a POMCP is similar to a MCTS. It starts by sampling a state-particle from the belief distribution. If all actions have not yet been explored from the root node of the tree, an action is chosen at random. Else an action is chosen based on their UCB- values. The state and action is passed through the generative function, which outputs a new state, a reward and an observation. From the observation, a new observation node can be created, and in this node the new state is placed. To attain a value of these action-observation nodes, a rollout from the new state is made, and finally that value is back propagated up to the root node. In the next step, the algorithm samples a new particle from the belief state, and the exact same procedure is done for this state-particle. If the same history is encountered by this particle, it is appended to that observation node. Therefore a history can contain a set of state-particles, and to proceed further down in the tree from this history, a state-particle is sampled from its collection.

Partially observable Monte Carlo planning with double progressive widening

As previously mentioned the POMCP algorithm reasons about history nodes, which contains action and observation pairs. Thus, the width of the search tree will grow exponentially with respect to the size of both these spaces. Similarly as before one can limit this growth by progressive widening, which enables solving decision processes defined over large or even infinite action- and observation spaces (in the case when both these spaces are restricted it is refereed to as *double progressive widening* (DPW)).

However, the progressive widening of an infinite observation space can be shown to have sub optimal properties. This is because of that the probability of getting the same observation twice is zero, and therefore the same history can never be created twice. The progressive widening process thus replaces the collection of state-particles in the observation nodes with a single particle. This results in an overconfidence in the search tree which might result in a vague exploration of the search space. This is illustrated in Figure 2.8 where the left search tree has been generated by the POMCP-DPW algorithm.

Partially observable Monte Carlo planning with observational widening

Monte-Carlo planning with observational widening [SK17] (POMCPOW) is an extension to POMCP-DPW that solves the single particle belief-problem. In this algorithm when the DPW-condition forces the search algorithm to pick an already created observation node, the algorithm adds the new state particle to that node's particle collection even if the generated observation doesn't match that observation node. To justify this the particle must first be weighted based on the conditional probability of the state-particle obtaining that observation. Thus this algorithm creates an approximate belief state in each observation node, where the weight of a particle is the probability of the particle observing that observation, see Figure 2.8. To compute these weights the algorithm can use the observation function. However, for some problems the observation probabilities might not be known, or impossible to formulate in a compact form. In these cases the probabilities can be approximated with some observation measure, as used in the particle filter, (see *particle filter methods* in section 2.3).

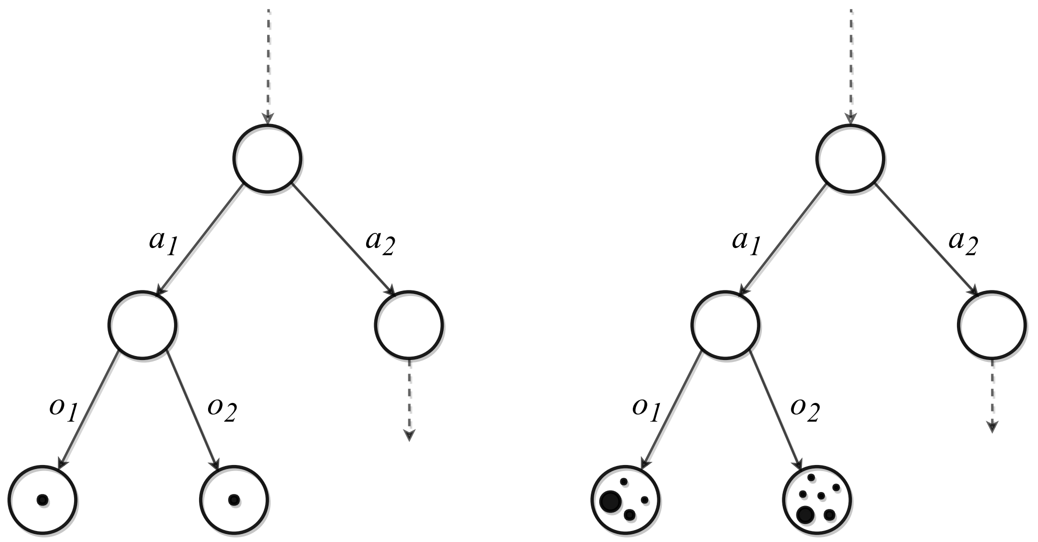


Figure 2.8: *Two search trees generated when solving a problem with infinite observation space solved by two different algorithms; POMCP-DPW to the left and POMCPOW to the right. The history nodes in the POMCP-DPW search tree contains single particles, since the probability of generating the same observation twice is zero. The POMCPOW algorithm uses a weighted particle collection in each history node, where the weights are indicated by the particles' size.*

2.5 Motivation of the methodology

This section motivates the chosen methods used to model tactical decision-making during highway driving as the three parts of a decision-making process: *the framework*, *the interpreter* and the *solver*. (A lot of inspiration will be taken from the article [SHK17]).

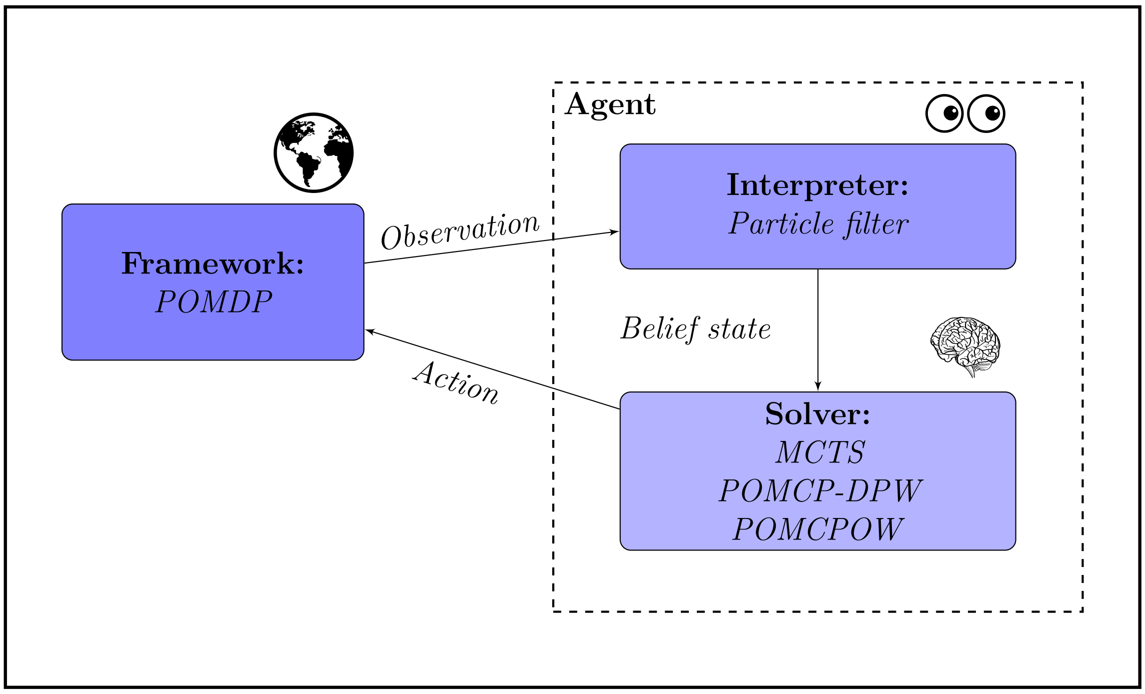


Figure 2.9: *The same structure of a decision-making problem as proposed earlier in Figure 2.1. The framework used for modelling the tactical decision-making is a POMDP. The filtering process is performed by a particle filter, and the problem is solved with the use of MCTS, POMCP-DPW, and POMCPOW.*

Framework

The problem considered in this thesis is tactical decision-making during highway driving—a problem which contains a lot of difficulties. Firstly the state space is described by continuous data, i.e., the position and velocities of the vehicles involved. Furthermore to make accurate traffic predictions the model must incorporate the intentions of other drivers, which will be modelled as a non-observable variable contained in the state space. Thus, the natural choice of mathematical framework used to model this problem would be as a POMDP with a continuous state space.

Another difficulty is that the possible accelerations and decelerations of a vehicle are more or less infinite, meaning that the POMDP would have to be modelled by an infinite action space. Since tactical decision-making refers to the selection of high-level maneuvers, the action space can be reduced to a finite set of high-level actions, such as 'Change lane to the right' or 'reduce the gap to the preceding vehicle'. The actual acceleration is instead found by using an adaptive cruise control (ACC). There is a trade-off when selecting the number of actions in the action space since more actions will provide a larger search space and thus increase the possibility to find an optimal sequence of actions. However, a larger search space also means that the probability of finding a good solution decreases.

Interpreter

The input to the tactical decision making program will be processed sensor data, containing the physical states of the other vehicles (i.e., their position and velocities). A sensor input corresponds to an observation in the POMDP framework, and since the physical data is continuous the observations space is infinite. The behaviours of the other drivers will be estimated by a particle filter, which has been shown to provide an accurate yet computationally efficient method [SHK17]. Furthermore, the implementation will not include any sensor errors, since these are typically small compared to the standard errors generated when predicting a vehicles motion, which include incorrect lane changes or accelerations in the wrong longitudinal direction (examples of prediction errors can be seen later in Chapter 4 and Figure 4.6).

Solver

The continuous state and observation space, and the non-observability of the drivers' intentions, imply that the algorithms best fit to solve this problem would (based on the theory thus far) probably be POMCP-DPW or POMCPOW. However, since the exact conditional probabilities of the observations are unknown these will be estimated by using a distance measure, similar to that of a particle filter. In addition to POMCP-DPW and POMCPOW another tree search algorithm is included. This algorithm will only consider the single best particle in the particle filter when creating the tree (i.e., the particle with the largest weight). We call it a *most likely Markov decision process* (MLMDP), since it solves the current MDP-problem, where all the other drivers' intentions are set to the most likely behaviour, with a MCTS.

Chapter 3

Methodology

In this chapter highway driving is formulated as a POMDP. Three different algorithms, able to solve the POMDP problem, are investigated: MLMDP, POMCP-DPW and POMCPOW. Two additional solvers are implemented, to provide upper and lower bounds. Lastly the highway scenario is presented, where the problem is to maneuver through dense highway traffic in order to reach an exit ramp.

3.1 Decision-making during highway driving

Autonomous driving involves various challenges, Figure 3.1 shows a simplified sketch of the different processes involved. Firstly the agent must be able to perceive its environment, which is done via multiple sensors that are mounted on the vehicle, and through sensor fusion the relevant data can be extracted and interpreted. This interpretation should include all important visible information, such as, for example, the total number of lanes, where the road markings are, and the positions and velocities of all participating vehicles (such as cars or trucks). This process is denoted in the picture by *sensor fusion*. The next step analyses this data, and determines how to best act. This is the *tactical decision-maker*-module. As we have discussed previously, not all the information needed to take a decision are visible to the sensors. Therefore it is crucial to have a robust framework that can account for this: the POMDP. The details of how this is done are shown during the next section. The decisions taken by the tactical decision-maker are ‘high level maneuvers’, such as to change the gap to the preceding vehicle, or to change lane. This information is sent to a *trajectory planner*, which calculates what actuations to execute in order to fulfill the requests from the tactical decision-maker.

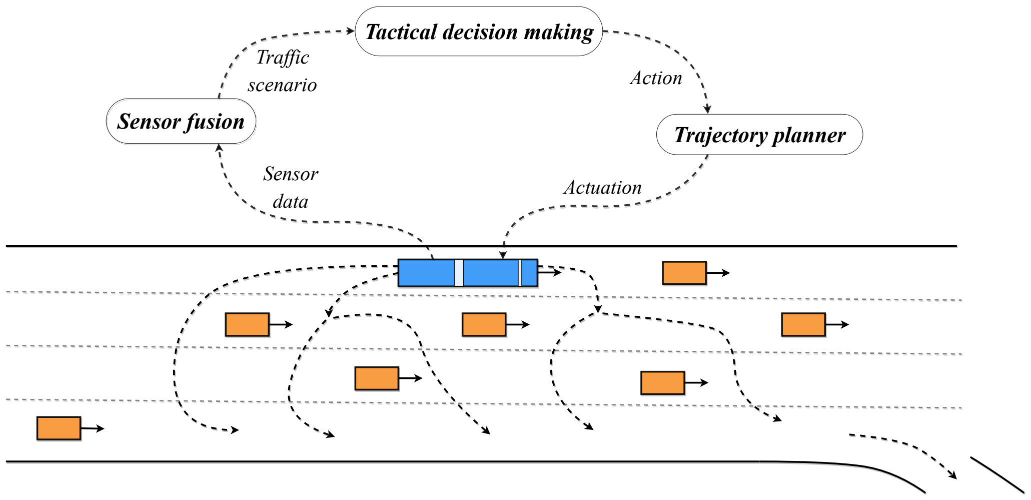


Figure 3.1: *A rough sketch of the architecture in a fully automated truck. Sensor fusion is where the sensor data is interpret, Tactical decision-making is where a high level decision about how to maneuver is taken, and in Trajectory planner the maneuver request is translated to actuations.*

3.2 Formulating highway driving as a POMDP

This section formulates the main aspects of a self-driving vehicle as the 7-tuple of a general POMDP. This includes:

- A **state space** describing the relevant physical properties of the highway scenario, and also the driver behaviours.
- **Observations** from the sensors. These will contain all information that is possible to measure with sensors, such as camera and LIDAR.
- An **action space**, which describes how the agent may act in each time frame.
- A **transition function**, which governs how a physical state transitions in time based on the agent's action.
- A **reward model**, that determines the reward received for transitioning from one state to another, due to an action.
- An **observation function**, that, due to the lack of a compact representation, will be approximated by a particle filter.
- To prioritize goals closer in the future the POMDP also includes a **discount factor** $\gamma = 0.95$; see (2.2).

The items on this list will, during the rest of this section, be explained more thorough. All numerical values of the parameters in the equations defined in this chapter can be found in table 3.1. These are either set to similar values as the ones found in the literature study, or empirically set to values that represent highway driving.

State space and behaviours

The state space represents the underlying variables governing the dynamics of highway driving. It consists of two key components: the measurable physical properties of the involved vehicles, and their intentions. Denote vehicle i's physical variables by $p_i = (x_i, \dot{x}_i, y_i, \dot{y}_i)$, where x and y denotes longitudinal and lateral position, respectively. It is assumed that this information is noise-free, and perfectly known, since the sensor noise is typically relatively small compared to prediction errors.

The second component of the state space, the behaviour of the other drivers, is something that is important to consider in order to make accurate predictions. The behaviour of vehicle i will be denoted by θ_i, which is a vector of eight scalars (this is explained during the next two pages in **Intelligent driver model** and **Modelling lane changes**). The behaviour will completely determine a vehicle's lane changes, and horizontal acceleration outputs, $\ddot{x}_i$. These parameters cannot be directly observed via the agents sensors, and are therefore a non-observable component of the state space.

Thus, the state, s, of a traffic scenario is described by all the positions and behaviours of the vehicles involved: $s = \left\{p_e \times \{p_i, \theta_i\}_i^{N_{\text{Vehicles}}}\right\}$, where p_e is the physical state of the ego-vehicle, and $N_{vehicles}$ denotes the number of involved vehicles.

Observation- and belief space

As mentioned previously, the behaviours of the surrounding vehicles are non-observable sets of variables contained in the state space. The agent will only be able to perceive the physical variables, $z = \left\{p_e \times \{p_i\}_{i=1}^{N_{\text{Vehicles}}}\right\}$, where z denotes an observation of the state space. However, in order to make accurate predictions, the agent will have to infer the underlying behaviour parameters. This is done by a particle filter, that creates a pool of possible driver 'profiles', and is explained in more detail in Section 3.2.1.

Transition function

The transition function is a set of equations that maps the current state into a new state. For the highway driving-problem, this is a function of the physical state, the agent's action, and the drivers' behaviours. To reduce the complexity of the transition function, the vehicles are modelled as point masses, and are updated by the equations of motion. Each vehicle will also be seen as fixed to the centre of its current lane. While in the lane, the driver will regulate its longitudinal and lateral velocity separately. The equations of motion for the lateral and longitudinal coordinates are given by

$$\begin{cases} x_{t+1} = x_t + \dot{x}_t \Delta t + \ddot{x}_t \frac{\Delta t^2}{2}, \\ \dot{x}_{t+1} = \dot{x}_t + \ddot{x}_t \Delta t, \\ y_{t+1} = y_t + \dot{y}_t \Delta t, \\ \dot{y}_{t+1} = \dot{y}_t + \ddot{y}_t \Delta t, \end{cases} \tag{3.1}$$

where the lateral updates neglects the instantaneous accelerations, and $\Delta t = t_{i+1} - t_i$ (i.e., the integration time step size). The transition function can be formulated as a linear system on the form

$$p_{t+1} = \begin{bmatrix} x_{t+1} \\ \dot{x}_{t+1} \\ y_{t+1} \\ \dot{y}_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ \dot{x}_t \\ y_t \\ \dot{y}_t \end{bmatrix} + \begin{bmatrix} \frac{\Delta t^2}{2} & 0 \\ \Delta t & 0 \\ 0 & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} \ddot{x}_t(a_t;\theta) \\ \ddot{y}_t(a_t;\theta) \end{bmatrix} = \mathbf{T}(p, \theta, a_t), \tag{3.2}$$

where $(\ddot{x}_t, \ddot{y}_t)$ are the acceleration at time step t. For the ego vehicle these accelerations will be given by an ACC, which is presented in the next section. For all other vehicles, the longitudinal accelerations is modelled by the *intelligent driver model* (IDM), and the lateral accelerations, corresponding to lane changes, are modelled using *minimize overall braking induced by lane changes* (MOBIL). These two models are functions of the physical state, as well as a set of eight parameters. These parameters are individual, and constant for each vehicle, and is what fully defines a vehicle's behaviour. This is explained in more detail below.

Intelligent driver model

The intelligent driver model is a microscopic model, that models longitudinal acceleration, meaning that it is a model based on the interaction between individual drivers ([THH00]). The model is given by

$$\ddot{x}_{\mathrm{IDM}} = a \left[1 - \left(\frac{\dot{x}}{\dot{x}^*} \right)^{\delta} - \left(\frac{g^*}{g} \right)^2 \right], \tag{3.3}$$

where $\ddot{x}_{\mathrm{IDM}}$ is the acceleration of the considered vehicle, $\delta > 0$ is a parameter modelling the importance of maintaining the desired speed $\dot{x}^*$, a is the vehicle's maximum acceleration, and g is the current gap to the preceding vehicle. The desired gap, g^*, is given by

$$g^* = g_0 + T\dot{x} + \frac{\dot{x}\Delta\dot{x}}{2\sqrt{ab}}, \tag{3.4}$$

where T is the desired time gap (the time it would take to reach the preceding cars' current position), $b > 0$ is the desired deceleration, and g_0 is the minimum spacing to the preceding vehicle. The relative velocity, $\Delta\dot{x}$, is computed with respect to the preceding vehicle, i.e., $\Delta\dot{x} = \dot{x}_{\text{Preceding}} - \dot{x}_{\text{Ego}}$.

The acceleration output, $\ddot{x}_{\text{IDM}}$, is not bounded from below as $g \to 0$, meaning that it may output an arbitrarily large deceleration. Therefore an artificial bound is added, which can be done as follows:

$$\ddot{x}_{\text{IDM}} \leftarrow \max\left(\ddot{x}_{\text{IDM}}, -b_{\text{Max}}\right), \tag{3.5}$$

where b_{Max} is the physical braking limit of the vehicle, and $\max(\cdot)$ returns the largest argument. The IDM-model determines a vehicle's longitudinal acceleration solely based on these five independent parameters: $(a, b, T, g_0, \dot{x}^*)$, as well as the relative velocity of the preceding vehicle. These five parameters are therefore part of the behaviour of a vehicle. The three remaining parameters comes from the lateral control model found below.

Modelling lane changes

Minimize overall braking induced by lane changes (MOBIL) is a model, which computes whether a lane change maneuver is beneficial ([KTH07]). It takes into account not only the ego vehicle, but also the surrounding drivers; see Figure 3.2. The model is based on evaluating the following statements:

$$\begin{cases} \Delta\ddot{x}_e + p\left(\Delta\ddot{x}_o + \Delta\ddot{x}_n\right) > a_{\text{Thresh}}, \\ \max\left(\tilde{\ddot{x}}_e, \tilde{\ddot{x}}_n\right) > -b_{\text{Safe}}, \end{cases} \tag{3.6}$$

where the indices e, o, n denote: *ego* vehicle, the *old follower* (the follower of the ego vehicle before the lane change maneuver), and the *new follower* (the follower of the ego vehicle after the lane change maneuver) of the ego vehicle, respectively. The acceleration differences $\Delta\ddot{x}_\cdot$ are defined in relation to whether a lane change is performed or not, i.e.:

$$\Delta\ddot{x}_e = \tilde{\ddot{x}}_e - \ddot{x}_e, \tag{3.7}$$

where $\tilde{\ddot{x}}_e$ is the longitudinal acceleration of the ego vehicle after a lane change. The model involves three parameters: the politeness factor p, the safety braking limit b_{safe}, and the beneficial threshold a_{thresh}. These three parameters are, in addition to the five IDM-parameters above, included in the behaviour of a vehicle. If the conditions in (3.6) are satisfied it means that a lane changing maneuver is beneficial

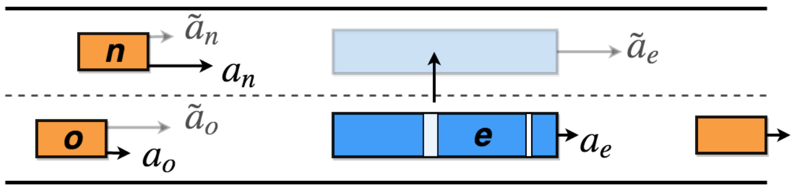


Figure 3.2: *A depiction of the different acceleration components used in the MOBIL model to evaluate whether a lane change is beneficial or not. $a_.$ is the current acceleration of the vehicles, and $\tilde{a}_.$ is the acceleration if the truck does a lane change. The indices, e, o, and n denotes the ego vehicle, the old follower of the ego vehicle, and the new follower of the ego vehicle, respectively*

for the ego vehicle, and in the case when $p > 0$ it is usually beneficial for the other involved drivers as well.

Reward model

The reward model governs the agent's choice of action, since an optimal policy is defined as the sequence of actions that returns the largest amount of reward possible. The reward model is constructed as a mapping from the state space to the real numbers, i.e., $R(s) : S \rightarrow [0, 1]$, and will be constructed to reflect some of the basic goals that a driver of an articulated vehicle might strive to fulfill. Since the reward model should factor in several different criteria, it can be seen as a combination of several different goals.

The importance of reaching and maintaining a desired lane can be modelled as following:

$$R_{\text{lane}}(y_e) = 1 - \frac{|y_e - y^*|}{\sigma_y}, \tag{3.8}$$

where y^* is the lateral coordinates of the centre of the desired lane, y_e is the lateral coordinate of the ego vehicle, and σ_y is a normalization constant—it is set so that if the ego vehicle is in the lane furthest away from the desired lane the returned reward is zero, and thus $R_{\text{lane}}(y_e) \in [0, 1]$. The reward models don't have to be normalized, but it simplifies the tuning later on.

When considering tactical decision-making in an autonomous vehicle it is important to factor in that the other drivers have their own intentions and objectives too. An agent only acting upon self-indulgent objectives could result in a dangerous non-cooperative behaviour, especially for a heavy articulated vehicle. Therefore the next reward function is designed to value actions that act less impeding on the other drivers and the traffic flow. To model this one can use a measure that is similar to the one used to evaluate good lane changes in the MOBIL model:

$$a_{\text{Induced}} = \Delta\ddot{x}_{\text{Rear}} = \ddot{x}_{\text{Rear}} - \ddot{\tilde{x}}_{\text{Rear}}, \tag{3.9}$$

where $\ddot{x}_{\text{Rear}}$ is the current acceleration of the vehicle located behind the ego, and $\ddot{\tilde{x}}_{\text{Rear}}$ would be the acceleration of the same vehicle given that the ego wasn't located in front of it—giving a measure of how much the ego blocks the rear vehicle. When there is no vehicle behind the ego driver this difference is set to zero. A reward function can be constructed from this quantity as follows:

$$R_{flow} = \begin{cases} 1, & \text{if} \quad a_{\text{Induced}} \geq 0, \\ 1 - \left|\frac{a_{\text{Induced}}}{a_{\text{Threshold}}}\right|, & \text{if} \quad a_{\text{Threshold}} < a_{\text{Induced}} < 0, \\ 0, & \text{otherwise}, \end{cases} \tag{3.10}$$

where $a_{\text{Threshold}} > 0$ is a constant that defines a threshold from which further deceleration doesn't produce more penalty.

The total reward is constructed by combining the rewards from the agent's different objectives, as well as a small penalty for being in a lane change:

$$R_{\text{Tot}} = \frac{R_{\text{Lane}} + \lambda R_{Flow} + C_{LC}\mathbf{1}(\dot{y}_{Ego} = 0)}{1 + \lambda + C_{LC}}, \tag{3.11}$$

where C_{LC} is a constant corresponding to the penalty for being in a lane change, and λ is a parameter that models the importance of the competing objectives. The denominator is set this way to ensure that the reward is normalized, i.e., $R_{\text{Tot}} \in [0, 1]$. When considering a scenario where lane changes are important, for example to quickly get across a wide highway, it might be beneficial to set the constant $C_{LC} = 0$.

To get a better understanding of this reward model, consider the example in Figure 3.3. This figure shows a scenario where the goal is to get to the rightmost lane, and there is a vehicle located behind the ego vehicle in the neighbouring lane. There are two different policies that would fulfill this task, denoted by π_1 and π_2. The policy π_1 entails a longer time to reach the lane, but it does so without disturbing any traffic. The policy π_2 is more aggressive and would result in a faster lane change at the cost of inducing braking in the other vehicle. Depending on the parameter λ in the reward model (3.11) the agent can be set to seek either solution, i.e., when $\lambda \ll 1 : R(\pi_2) > R(\pi_1)$ and when $\lambda \gg 1 : R(\pi_1) > R(\pi_2)$.

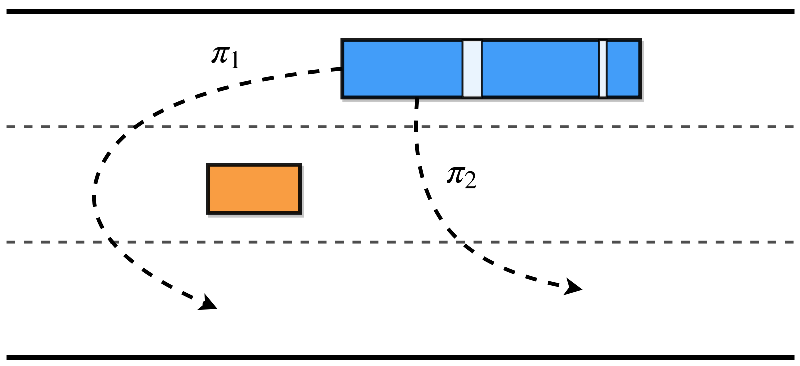


Figure 3.3: *A scenario, where there are two different ways to get to the rightmost lane, denoted by π_1 and π_2. Which one that is optimal depends on the parameter λ in the reward model* (3.11). *Thus this parameter can be used to make the agent prioritise different goals depending on the scenario.*

Action space

There are many possible ways of constructing an action space, but one of the main goals is to span a wide range of possible behaviours by using as few actions as possible. The most direct approach would be an action space containing a wide range of different longitudinal and lateral accelerations. But, since the *Trajectory planning* module from Figure 3.1 has a more detailed description of the physical model of an articulated vehicle than the *Tactical decision-making* module, it is rather this module that should take decisions about low-level actuations. Therefore, the action space in the *Tactical decision-making* module will contain this set of high-level maneuvers:

$$\mathcal{A} = [\text{Accelerate}, \text{Maintain speed}, \text{Decelerate}, \text{LC-left}, \text{LC-right}]\,, \tag{3.12}$$

which only contains five different actions. The fact that this action space is small means that the growth of the search tree will be kept minimal—the exponential growth of the search tree is proportional to the size of the action space; see Section 2.4.

Longitudinal action space

To translate the longitudinal components of the agent's high-level action space into low-level actuations, an adaptive cruise control (ACC) was implemented. The ACC allows the agent to regulate its time gap to the preceding vehicle, and when there are no vehicles in front of the agent, it instead regulates the agent's velocity. The ACC used in this thesis is an extension of IDM, with a few modifications. These were made because the steady state velocity of the IDM model, i.e., $\ddot{x}_{ACC} = 0$, does not occur when a vehicle is in a position where $g = g^*$ and $\dot{x} = \dot{x}^*$. Rather, a vehicle governed by IDM will never reach its desired gap if it has a preceding vehicle, since the velocity term in (3.3) will always contribute to a deceleration. This can be changed by changing the 1 in the equation to a 2. Then the steady state will be when $g = g^*$ and $\dot{x} = \dot{x}^*$. The resulting ACC equation is thus the following:

$$\ddot{x}_{ACC} = a\left[2 - \left(\frac{\dot{x}}{\dot{x}^*}\right)^2 - \left(\frac{t_g\dot{x} + \frac{\dot{x}\Delta\dot{x}}{2\sqrt{ab}}}{g}\right)^2\right], \tag{3.13}$$

where the parameters have the same interpretation as in the IDM (3.3), and the jam distance g_0 has been removed, since this parameter only is relevant in a traffic jam situation, which will not be included in the simulations.

As with the IDM model, the acceleration outputs from the ACC must be bounded from above and below, which is done as:

$$\begin{cases} \ddot{x}_{\text{IDM}} \leftarrow \max\left(\ddot{x}_{\text{IDM}}, -b_{\text{Max}}\right), \\ \ddot{x}_{\text{IDM}} \leftarrow \min\left(\ddot{x}_{\text{IDM}}, a_{\text{Max}}\right), \end{cases} \tag{3.14}$$

where $b_{\text{Max}} > 0$ and $a_{\text{Max}} > 0$ are parameters modelling the vehicles maximum deceleration and acceleration, respectively.

The motion of the truck is governed by (3.13), and will be regulated by changing the desired speed parameter, $\dot{x}^*$ and the desired time gap, t_g. To do this while only having three longitudinal actions, introduce a discrete state variable, $ACC_{\text{state}} \in \{1, 2, \ldots, N_{\text{ACC}}\}$. The action space is then constructed around two vectors containing relative velocities, $\mathbf{v^{ACC}}$, and time gaps $\mathbf{t_g^{ACC}}$. Using this, the desired velocity and time gap in the ACC is given by:

$$\begin{cases} \dot{x}^* & = \dot{x}_{\text{Ego}} + \mathbf{v^{ACC}}\left(ACC_{\text{state}}\right), \\ t_g & = \mathbf{t_g^{ACC}}\left(ACC_{\text{state}}\right), \end{cases} \tag{3.15}$$

where $\dot{x}_{\text{Ego}}$ is the current speed of the ego vehicle. The agent controls the ego vehicle by increasing or decreasing the ACC_{state} by selecting the different actions from the action space, the ACC state is updated as follows:

$$\begin{cases} \text{Increase speed}: & ACC_{\text{state}} \leftarrow ACC_{\text{state}} + 1, \\ \text{Maintain speed}: & ACC_{\text{state}} \leftarrow ACC_{\text{state}}, \\ \text{Decrease speed}: & ACC_{\text{state}} \leftarrow ACC_{\text{state}} - 1. \end{cases} \tag{3.16}$$

The ACC should only model the ego vehicle's desired speed given that the time gap to the preceding vehicle is larger that the desired time gap. To accomplish this, the time gap t_g in equation 3.13 is continuously updated as follows:

$$t_g \leftarrow \max\left(t_g, \frac{g}{\dot{x}}\right), \tag{3.17}$$

meaning that the ACC will set the time gap to $g/\dot{x}$ when this quantity is larger than the desired time gap. By inserting this quantity into (3.13) one gets that the acceleration will only depend on the relative velocities between the ego and the preceding vehicle.

One of the benefits of using this type of action space is that the ACC is collision free, meaning that the agent would not be able to produce a longitudinal collision by for example selecting the 'Accelerate' action repeatedly. However, this method will only work if there is an actual vehicle in front of the agent. This problem is solved by adding a 'ghost vehicle'—a fictional vehicle placed in front of the ego vehicle at a distance corresponding to the maximum range of the sensors. The ghost vehicle is set to always keep the same speed as the ego vehicle.

Lateral action space

The lateral action space should control the lane changing maneuvers. In this study, a simplified model was used, where the truck was given a constant lateral velocity $\dot{y}_e = \pm\dot{y}_{\mathrm{LC}}$ in the desired direction, and when the ego reached a neighbouring lane it stopped. This is a simplification of a real lane changing maneuvers for a truck. But, a more accurate lane change model would be computationally costly, and in the context of a MCTS it is not sure that it in the end would result in a better prediction model. Instead, the simplest way to account for the capabilities of a truck is to set 'realistic' values of $\dot{y}_{\mathrm{LC}}$, that results in 5–10 seconds long lane changes.

Pruning of the action space

The action space is pruned so that the agent may not choose a lane change maneuver if it will undoubtedly produce a crash, see Figure 3.4. The lane change action is removed if the ego vehicle arrives in a lane where the gap, g, or the time gap, t_g, is too small, i.e., $t_g < t_g^{\mathrm{prune}}$ or $g < g^{\mathrm{prune}}$, for some parameters $t_g^{\mathrm{prune}} > 0, g^{\mathrm{prune}} > 0$. The pruning also reduces the time complexity of the searching algorithm, since it decreases the size of the search tree. For more discussion about this, see Section 5.1.4.

3.2.1 Particle filter belief updates

The task of the particle filter is to find the driver behaviour that could have caused the observed acceleration. The filter keeps a set of M particles, $\{s_i\}_{i=1}^{M}$, where a particle $\tilde{s} = \left\{p_e \times \{p_i, \tilde{\theta}_i\}_{i=1}^{N}\right\}$ consists of a physical state, and eight behavioural

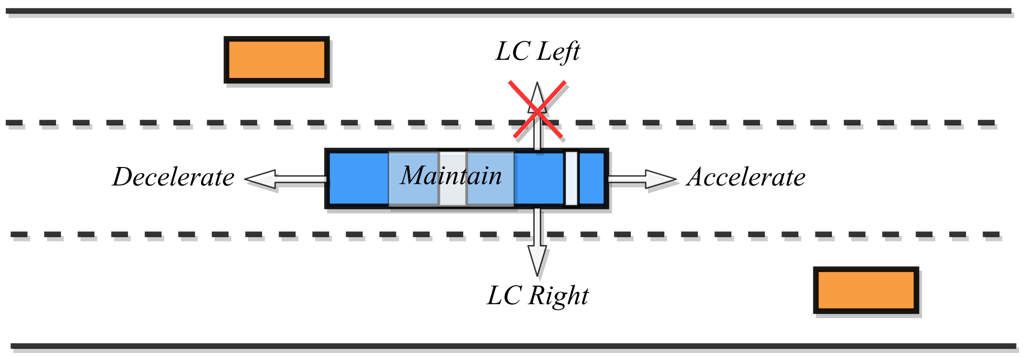


Figure 3.4: *A depiction of the five actions in the action space. The option to change lane to the left is in this instant pruned, since there is a vehicle currently inhabiting this neighbouring lane. The action to accelerate means to increase the value of the ACC state, and respectively to decelerate means to decrease it.*

parameters, $\theta = (a, b, t_g, g_0, \dot{x}^*, a_{\text{Thresh}}, b_{\text{Safe}}, p)$. Each particle has a coupled weight, w, where each weight is calculated to give a measure of how probable the behaviour was to have generated the most recently observed acceleration. The particle filter belief update is done as in Section 2.3.1, and consists of first sampling a particle with probability proportional to its weight, and then perform the following update:

$$w := P_{\text{filter}}(\theta, z) = \exp\left(-\frac{(a_z - a_\theta)^2}{2\sigma_{\text{accel}}^2}\right), \tag{3.18}$$

where a_z is the observed acceleration, a_θ is the acceleration output from the IDM model when inserting the behaviour parameters in the particle, and σ_{accel} is a normalisation constant. Thus the weight estimates how well the particle was able to predict the longitudinal motion of the driver. To filter the lateral motion the weight is modified by:

$$\begin{cases} w \leftarrow w\eta, & \text{if} \quad y_{\hat{\theta}} \neq y_z, \\ w \leftarrow w, & \text{if} \quad y_{\hat{\theta}} = y_z, \end{cases} \tag{3.19}$$

where $y_{\hat{\theta}}$ is the lateral coordinate in the particle, and y_z is the observed lateral coordinate, and lastly $\eta \in [0, 1)$ is a penalty parameter. Thus the particle filter also accounts for incorrect lane changes, i.e., it predicts the lateral motion of the vehicles.

This process is repeated M-times, resulting in M new particles weighted by the most recent observation. By normalising the weights, i.e., $w_i \leftarrow \frac{w_i}{\sum_{j=1}^{M} w_j}$, this set can be interpreted as a discrete distribution over the particles, which produces an approximate belief distribution

$$\hat{b}(s) = \left\{p_e \times \{p_i\}_{i=1}^{N_{\text{Vehicles}}} \times \{w_j, \theta_j\}\right\}_{j=1}^{M} \in \mathcal{B}, \tag{3.20}$$

where $\hat{b}(s)$ denotes the approximated belief state, and $\mathcal{B}$ is the belief space.

3.2.2 Tactical decision-making using MCTS algorithms

Three different tree search algorithms was used to solve the tactical decision-making POMDP. The first algorithm is called MLMDP, which solves a problem when only the most likely particle is considered. The resulting MDP is solved with a standard MCTS. The other two algorithms, POMCP-DPW and POMCPOW, solves the POMDP problem using the whole belief state (how these algorithms work in more detail can be found in Section 2.4). All algorithms used UCB_1, and the same rollout policy. These algorithms were compared to a baseline and upper bound, namely the static assumed behaviour- (SAB) and omniscient (OMNI) algorithms (as done in [SHK17]). The algorithms are described below.

Rollout policy, generative model, and MCTS-hyperparameters

The *simulation* step in the MCTS algorithm evaluates a newly explored state by following a predetermined rollout policy. The objective of the scenario considered in this thesis is to maneuver across a highway. Thus a good rollout policy should evaluate states based on their potential to change lanes. To accomplish this the rollout was set to maintain the current speed, and as soon as possible change lane towards the desired lane. Thus a good state corresponds to a state where the ego can reach close to the target lane.

To generate a new state the selected action from the MCTS is input to the transition function, which generates a new physical state, which also is the agents observation of the new state. A reward is then returned by evaluating the new physical state with the reward function. The generative model can be seen as a combination of these steps, i.e., $z, r, s' \sim G(s, a)$.

The exploration constant, c, in the UCB1 formula (2.11) was set to weight exploration against exploitation. Generally it is set to $c = \sqrt{2}$ given that the reward function has an output in the unit interval, i.e., $R \in [0, 1]$, where it guarantees asymptotic convergence in each branch [BPW+12]. However, we found that in our application this setting resulted in too much exploration and thus given a limited number of searches it was not optimal. By experimenting with this value we found that $c = 0.1$ performed well.

The POMCP algorithms had the DPW node limit $k = 3$, meaning that there were three observation nodes for each action node. A large value of k would result in a shallow tree search, while a smaller value might result in an overconfidence solver. The growth was set to $\alpha = 0.1$, as in [SHK17]. This parameter is not as important when running a small amount of searches.

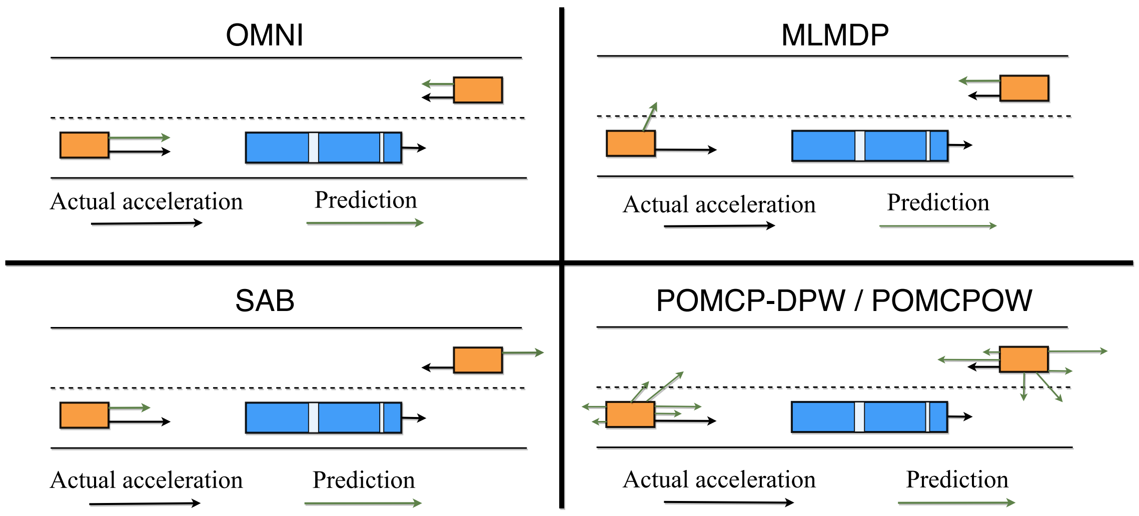


Figure 3.5: *A visualisation of the five algorithms, where the arrows indicate their predicted accelerations. The top-left panel shows the OMNI solver, which uses the exact accelerations to predict the traffic. This should provide an upper bound for the other algorithms in the simulations. In the lower left pane is the SAB algorithm, which always uses the same behaviour to predict the vehicles, regardless of the previously observed accelerations. In the top-right panel is the prediction of the MLMDP algorithm, which uses a particle filter to create a pool of behaviours. It only uses the most likely behaviour when doing the predictions. The POMCP and POMCPOW algorithms in the lower-right panel considers multiple particles at once.*

Baseline and upper bound

The static assumed behaviour formulation (SAB) is defined by solving the MDP that is derived by assuming that all vehicles drive according to a predetermined behaviour. To get a realistic baseline the chosen behaviour is set to the expected value of the behaviours used in the simulations, why all parameters θ^i in the driver models are set to $\theta^i = \theta^i_{min} + (\theta^i_{max} - \theta^i_{min})/2$. To get an *upper bound*, one instead uses the correct underlying behaviours of the vehicles. The resulting agent is thus completely aware of the behaviours of the other driver, and therefore make the most accurate predictions possible. This is something that is only possible to do in a controlled simulation environment. In the left two panels of Figure 3.5 the upper and lower bounds are visualised. The OMNI-algorithm knows exactly how much both cars will accelerate or decelerate, while the SAB-algorithm thinks that they will both behave equally.

Monte Carlo tree search with the most likely particle

The most likely MDP (MLMDP) is an algorithm that can use the POMDP formulation of highway driving, and solve it as an MDP using MCTS. To do this it performs particle filter belief state updates between each iteration, and solves the

MDP-problem that results from only considering the most likely behaviour (i.e., the behaviour with the highest weight). This algorithm can be expected to perform quite well since the complexity of the MCTS-algorithm is lower than for the other two algorithms below (this is mostly due to the lack of observations during the tree search). However, it is not able to account for multiple possible behaviours, which means that it can become overconfident. This can be seen in the upper right pane in Figure 3.5, where the arrows represents what the MLMDP-algorithm thinks is the most probable acceleration of both vehicles.

POMCP with a DPW particle collection

In each iteration, the POMCP-DPW algorithm samples states from the particle filter and performs a tree search. The tree search is combined with DPW to compensate for the large observation space. However, there are two different methods regarding how to arrange the particles in the history nodes with DPW. The regular POMCP-DPW algorithm would replace the current particle with the particle contained in the history node, and then continue the tree search with that particle. This means that as soon as DPW starts to reject new observations the problem becomes an MDP (see Figure 2.8). However, the article [SHK17] solves this problem by keeping a set of unweighted particles in each history node. In this case the algorithm proceeds the tree search simulation by drawing a particle from this set uniformly.

At initialisation the belief state in the root node is approximated by a set of particles and their corresponding weights, $\{w_i, \tilde{s}_i\}_{i=1}^{M}$, where each particle consists of a physical state and a set of behaviour parameters. Denote the particle corresponding to the behaviour $\tilde{\theta}$ by $\tilde{s} = \left\{p_e \times \{p_i, \tilde{\theta}_i\}_i^N\right\}$. The algorithm evaluates an action by sampling particles and computing the generative function $r, \tilde{s}', \tilde{z} \sim G(\tilde{s}, a)$, where r is a reward, $\tilde{s}'$ is a new state, and $\tilde{z}$ is an observation. The new state is a combination of the observation $\tilde{z}$, and the behaviours of the vehicles, i.e., $\tilde{s}' = \left\{\tilde{z}_e \times \{\tilde{z}_i, \tilde{\theta}_i\}_i^N\right\}$. The result of this simulation is saved as a history: $h = (a, \tilde{z}, \tilde{\theta})$. This process is repeated until the action a is progressive widened by the DPW algorithm. Let $\hat{s}$ denote the particle for which this happens. In this case the algorithm doesn't need to compute the resulting state and observation via the generative function, rather it appends the behaviour of the particle, $\hat{\theta}$, to the history node that the DPW-algorithm chooses, i.e., $h = \left(a, \tilde{z}, \{\tilde{\theta}, \hat{\theta}\}\right)$. This results in a pool of behaviours which all have reached this node. The algorithm proceeds by selecting one of the behaviours uniformly and combines it with the observation that was created by the first particle, i.e., it selects one of the following states: $\left\{\tilde{z}_e \times \{\tilde{z}_i, \tilde{\theta}_i\}_i^N\right\}$ or $\left\{\tilde{z}_e \times \{\tilde{z}_i, \hat{\theta}_i\}_i^N\right\}$.

The history nodes could also have included the physical states (i.e., the observation) of every particle, and not only the behaviours. But, since actions are pruned based on the physical states, i.e., T_{min}, g_{min}, different physical states could mean different actions spaces for the particles. This could result in poorly evaluated states. The lower right panel of Figure 3.5 visualise the POMCP-DPW algorithm, where multiple accelerations are taken into account when predicting the drivers.

POMCPOW

The POMCPOW algorithm is quite similar to the previously described POMCP-DPW algorithm. The difference is that the behaviours in the history nodes are weighted. The weighing should be done with respect to the probability of the observation function (i.e., $w = P(z|\theta)$), which can be interpreted as how likely the behaviour θ is to generate the observation z. However it becomes quite difficult to formulate an exact analytical observation function in the context of this problem. This is mainly due to the IDM and MOBIL models not having a unique solution, i.e., there are an infinite number of behaviours that generate the same acceleration and lane changing decisions. However, the method can still be applied by approximating the observation function (similarly as in the particle filter). When a particle gets appended to a history node the weights are computed as

$$w = \exp\left(-\frac{(a_{\tilde{z}} - a_\theta)^2}{2\sigma_{\text{accel}}^2}\right) \approx P\left(z, \theta\right), \tag{3.21}$$

where θ is the particle that is appended to the history node, and $\tilde{z}$ is the observation that created the observation node. Just as for POMCP-DPW it is shown in the right lower pane in Figure 3.5 that the algorithm takes a number of behaviours into account.

3.3 Tactical decision-making in an exit lane scenario

The algorithms are benchmarked as tactical decision-makers in a scenario where the ego vehicle has to maneuver across a densely populated highway to reach an exit; see the two left panels of Figure 3.6. The surrounding vehicles were randomly initialized (with random positions, velocities, and behaviours), and the algorithms were evaluated on 60 different variations of this scenario. All variations consisted of ten vehicles, and four lanes. The scenarios lasted for 75 seconds in total, and a decision was taken between each simulation time step, $\Delta t = 0.5$ seconds, meaning that the agent took 150 decisions in each scenario.

The *time horizon* during the tree search was $25s$, and in order to reach further down the tree a *dynamic time horizon* was used, meaning that the time between decisions during the predictions was not constant—rather it increased further down in the tree. The positions of the vehicles were updated in each time step by the transition function (3.2). As a reward function (3.11) was used for $\lambda \in \{0.001, 0.01, 0.1, 1, 10, 100\}$. A small λ-value means that the agent should mainly focus on finding a path to the objective lane, a large value means that the agent should try to minimize its negative interactions with the traffic, and $\lambda = 1$ provides the most difficult task of solving both these tasks at the same time.

Figure 3.6 shows a traffic scenario, and some interesting aspects of the state space. The lower left panel shows the true state of the traffic, while the panel above it displays the agents belief state, which is based on the most recent observation. Since the physical properties of the vehicles are assumed to be perfectly known the only difference between these states are the behaviours, which are indicated by different colors. The behaviours are vectors containing eight parameters: five IDM parameters and three MOBIL parameters, which were set to be correlated with a factor $\rho = 0.75$. This correlation means that a vehicle that maintains a high velocity is more likely to also be 'aggressive', meaning that it will keep a low time gap and possibly accelerate faster. The colour of the vehicles indicate the desired velocity parameter of their behaviour. A dark red colour indicates that the desired velocity is high, and since the behaviours are correlated the vehicle will most likely behave aggressively. Similarly a green colour represents a vehicle with a low desired velocity, and it can be expected to act passively.

The right panels of Figure 3.6 show some properties of the ACC and the particle filter during a simulation of 25 seconds. The upper right panel shows the acceleration output from the ego vehicle, and below it one can see the current ACC state. The parameters used in the ACC were set so that the ego would be slower than the surrounding traffic. By comparing these figures one can see how the acceleration is proportional to the ACC_{state}.

The lower right panel shows the root mean squared error of the behaviour parameters, E_{Filter}, which is computed as

$$E_{\text{Filter}} = \frac{1}{N_{\text{Vehicles}} N_{\text{Behaviour}}} \left[\sum_{i=1}^{N_{\text{Vehicles}}} \sum_{j=1}^{N_{\text{Behaviour}}} \left(\theta_j^i - \hat{\theta}_j^i \right)^2 \right]^{1/2} \tag{3.22}$$

where θ_j^i is the j:th behavioural parameter for vehicle i, $\hat{\theta}_j^i$ is the particle in the particle filter with the highest weight, $N_{vehicles}$ is the number of vehicles, and $N_{behaviour}$ is the number of behavioural parameters. This panel shows how the initial error is quite small, and seems to gradually improve as the agent receives more observations. The reason why the error is quite low in the beginning is because of that the behaviours are assumed to be correlated, and the believed behaviours of the vehicles are initialised as fully correlated vectors (i.e., $\rho = 1$). This means that all the parameters have the same value, and thus in the first time step the error is expected to be quite low; $E_{\text{Filter}} \approx 0.25$ from the figure.

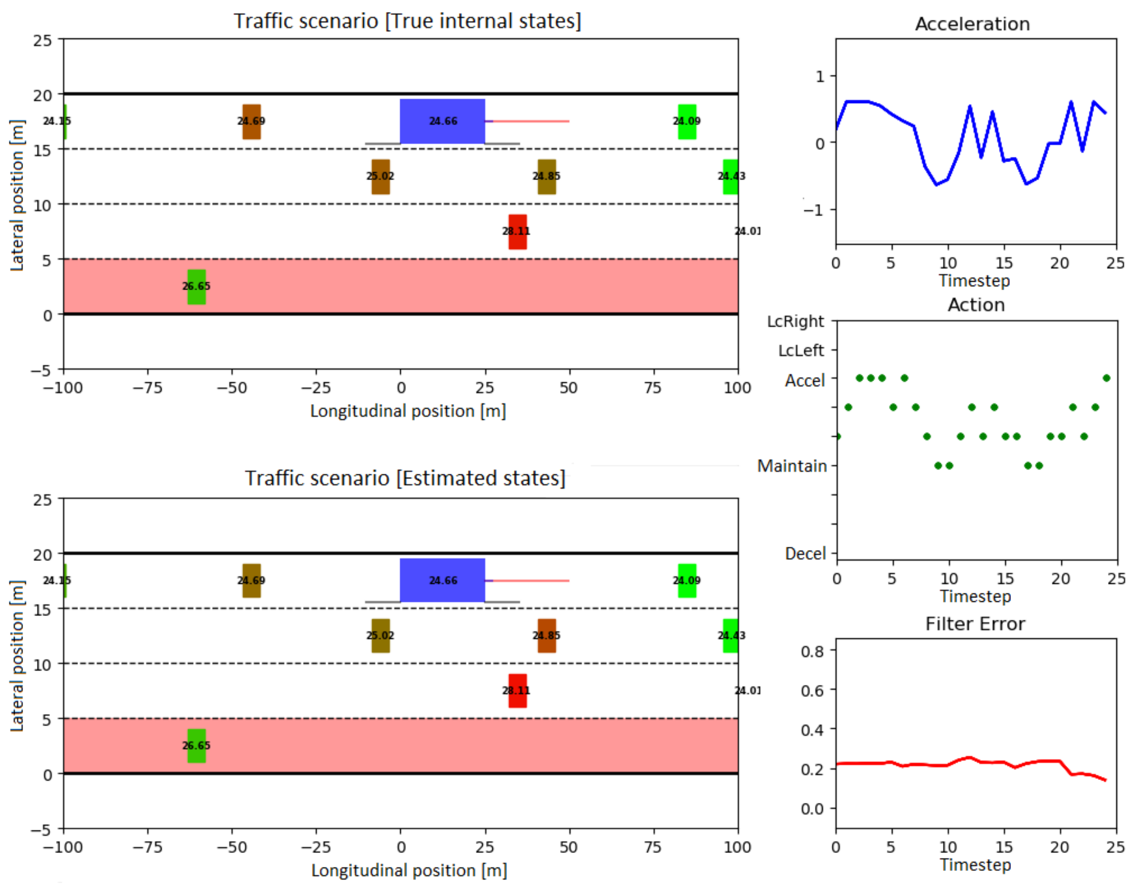


Figure 3.6: *A visualization of the different aspects of the state-space during one of the scenarios that was used to test the algorithms. The left figures show the ego vehicle as a the big blue square in the top lane. There are four lanes where the red lane indicates the target lane. The squares represent different vehicles, where their colour indicates their aggressiveness (which correspond to their behaviour), and the number on top of each vehicle indicates their current velocity. The difference between the two left pictures is that the top one shows the real state and the bottom one shows the agents current belief, which explains why some of the vehicles have different colour. The right panels in the order from top to bottom show the simulation history of: the acceleration output from the ACC, the state of the ACC, and the filter error, i.e., the accuracy of the agent's belief. The red line in front of the ego vehicle symbolises the safety time gap, meaning that if a vehicle enters this line then the ego won't regulate its speed; rather it starts to regulate the time gap to the preceding vehicle. The black lines around the ego vehicle indicate the distances in which the lane change option is pruned.*

Parameters used in the implementation of the highway driving scenario

Simulation parameters	*Symbol*	*Value*
Time step size $[s]$	Δt	0.5
Sensor range $[m]$		100
Number of lanes	N_{Lanes}	4
Number of vehicles	N_{Vehicles}	10
Minimum time gap prune $[s]$	t_g^{prune}	3
Minimum gap prune $[m]$	g^{prune}	10
Dynamic time steps $[s]$		[0.5, 1, 1.5, 2, 2.5, 2.5, 2.5, 2.5, 5, 5]
Lane reward parameter	σ_y	20
Flow reward parameter	a_{Thresh}	2
Lane change penalty	C_{LC}	0
Tree search		
Exploration constant	c	0.1
Discount factor	γ	0.95
DPW constant factor	k	3
DPW exponential factor	α	0.1
Truck capabilities		
Maximum acceleration $[m/s^2]$	a_{Max}	0.6
Maximum deceleration $[m/s^2]$	b_{Max}	7
Maximum speed $[m/s]$	v_{max}	30
Minimum speed $[m/s]$	v_{min}	15
Lane change time $[s]$		5
ACC relative speed values	$\mathbf{v}^{ACC}$	$[-10, -5, -1, 0, 1, 5, 10]$
ACC time gap values	$\mathbf{t}_g^{ACC}$	[3.5, 3, 2.5, 2, 1.5, 1, 0.5]
Particle filter		
Number of particles i filter	M	200
Correlation factor	ρ	0.75
Observation standard deviation	σ_{accel}	0.1
Resample probability	p$_r$	0.12
Resample standard deviation	σ_r	0.1
Incorrect lane change factor	η	0.2
IDM and MOBIL		*Passive/Aggressive*
Max acceleration $[m/s^2]$	a	0.8/2
Desired Speed $[m/s]$	$\dot{x}^*$	24/32
Desired Time Gap $[s]$	t_g	2/1
Desired Jam Distance $[m]$	g_0	4/0
Desired Deceleration $[m/s^2]$	b	1/3
Politeness	p	1/0.1
Safe brake $[m/s^2]$	b_{Safe}	1/3
Acceleration threshold	a_{Thresh}	3/1
Maximum deceleration $[m/s^2]$	b_{Max}	7

Table 3.1: Parameter values used during the simulations.

Chapter 4

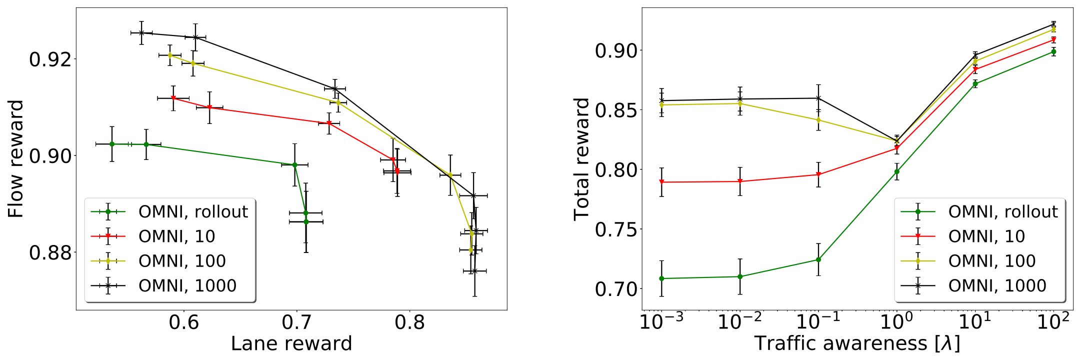
Results

This chapter presents the results from the simulations, which first evaluates how the different tree search-algorithms scale with the number of searches. Then three different tree searching algorithms are compared to the upper and lower bounds. Lastly, the algorithms' relative time complexity is presented.

4.1 How the number of searches influences the MCTS algorithm

It is not obvious that the simulation environment requires planning, and that more searches results in a higher cumulative reward. Therefore we test how well these types of algorithms scale with the number of tree queries, i.e., the number of simulations the algorithm use to construct the tree in order to find a solution. The results can be seen in Figure 4.1, where the different lines correspond to the OMNI algorithm being run for an increasing number of searches. The different amounts of searches are: 1000, 100, 10, and *rollout*, where rollout means that there is no planning involved: the agent rollout each of the allowed actions, and chooses the one with the highest value. The left panel shows the results from the individual reward functions where the points on the lines correspond to a specific value of λ used in the reward function (3.11). The flow reward, denoted by R_{flow} in section 3.2, plotted on the y-axis has been normalized by dividing the total reward accumulated by the agent by λ. The reward has also been normalized by the simulation length, and thus a reward of 1 corresponds to a perfect score with respect to that objective. The figure show the results for the values $\lambda \in \{0.001, 0.01, 0.1, 1, 10, 100\}$, denoted by the agent's *Traffic awareness*.

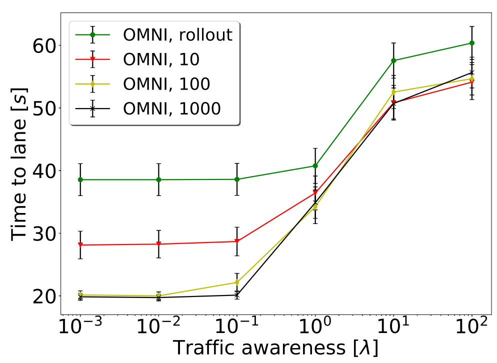
In the left panel the resulting curves produces a Pareto front, indicating that it is difficult to optimize both objectives at the same time. The top-left points on the curves correspond to $\lambda = 100$, and during these simulation the reward function mainly prioritizes the traffic flow. As λ is gradually set to smaller values a continuous convex curve is produced. As the search depth increases the Pareto front approaches the top-right corner, meaning that the more time the agent is given to plan, the better policies it is able to find. In the right panel one can see the total reward plotted against the λ-values. This graph shows the same trend, that when the searches increases the algorithm gets a higher reward.



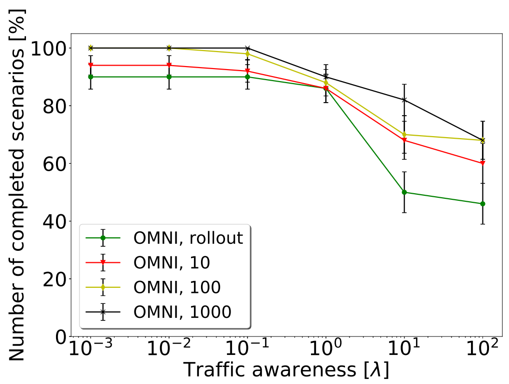
(a) *The flow and lane rewards.* **(b)** *Total reward against λ.*

Figure 4.1: *The accumulated rewards during highway simulation with the OMNI algorithm. The algorithm is run with four different values of number of searches: 1000, 100, 10, and rollout, and are evaluated on different values of lambda $\lambda \in \{0.001, 0.01, 0.1, 1, 10, 100\}$. Panel* **(a)** *shows the competing objectives plotted against each other, where the flow reward is divided by λ. The result is a convex Pareto curve. Panel* **(b)** *shows how the total reward scales with respect to the number of searches.*

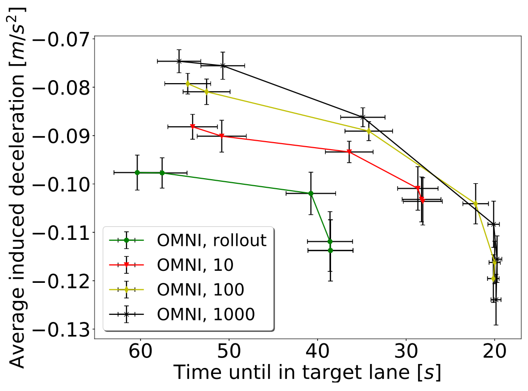
Figure 4.2 presents a different interpretation of the previous results. In panel **(a)** the average time it took for the ego to reach the target lane as a function of λ is shown. For small λ-values more searches results in a shorter time to reach the target lane. For large values of λ the average time to reach the lane increases for all search depths. When $\lambda = 100$ the truck is fastest to reach the lane when only searching 10 times, however, in these simulations the main objective is not to reach the target lane, but rather to not disturb the traffic. Panel **(b)** shows the success rate of traversing the highway and reaching the target lane during the 75 seconds that each simulation lasted. The success rate can be seen to increase with more searches for all values of λ. Panel **(c)** presents the physical results in a similar fashion as previously with the reward functions. It displays the competing objectives plotted against each other, i.e., the average time it took for the ego to reach the target lane as a function of the average induced braking that occurred while doing so. The results are quite similar to the ones in Figure 4.1 where the agent is able to prioritize either goal, producing a Pareto front. Lastly panel **(d)** shows how the induced braking scales with λ. The resulting curves are s-shaped, and as the searches increases the steeper the slope of the s-curve gets. The curvature means that the algorithm is better able to adapt and solve different objectives, which is modelled by the λ-value.



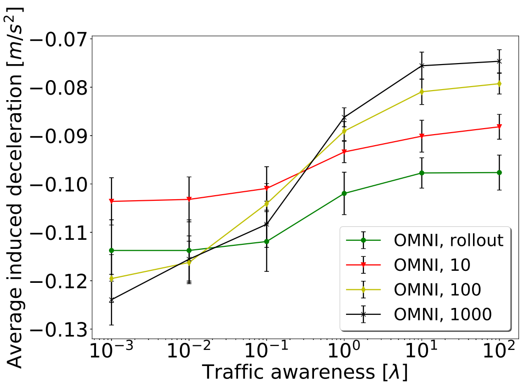
(a) *Time to target lane against* λ.



(b) *Number of completed scenarios against* λ.



(c) *Induced deceleration against time to target lane.*



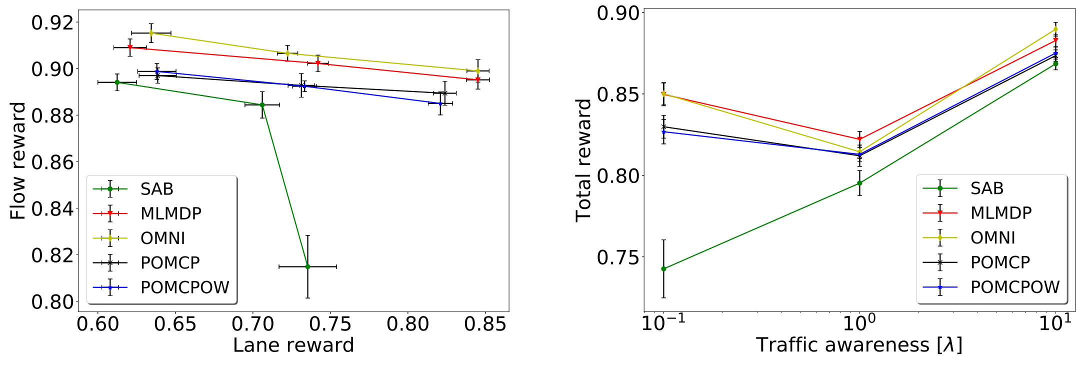
(d) *Induced deceleration against* λ.

Figure 4.2: *The results from Figure 4.1 presented in some relevant physical variables. The different panels show how the search depth affects the algorithms possibility to optimize different objectives. For each line the nodes correspond to different values of* λ. *These figures show the rewards for the values* $\lambda \in \{0.001, 0.01, 0.1, 1, 10, 100\}$.

4.2 Comparison of the algorithms' performance during simulations

Figure 4.3 shows how the different algorithms perform when run for 1000 searches per decision. The results are presented in a similar way as for the search depth, where the result from each solver is represented by a line, and each point on the line corresponds to a value of $\lambda \in \{0.1, 1, 10\}$. The left panel shows the average amount of returned reward from each objective. This panel shows that the MLMDP solver outperform the POMCP and POMCPOW algorithms, since the nodes of its Pareto curve is closer to the upper-right corner. The POMCP and POMCPOW perform almost equal and manage to outperform the SAB, which provides a lower bound.

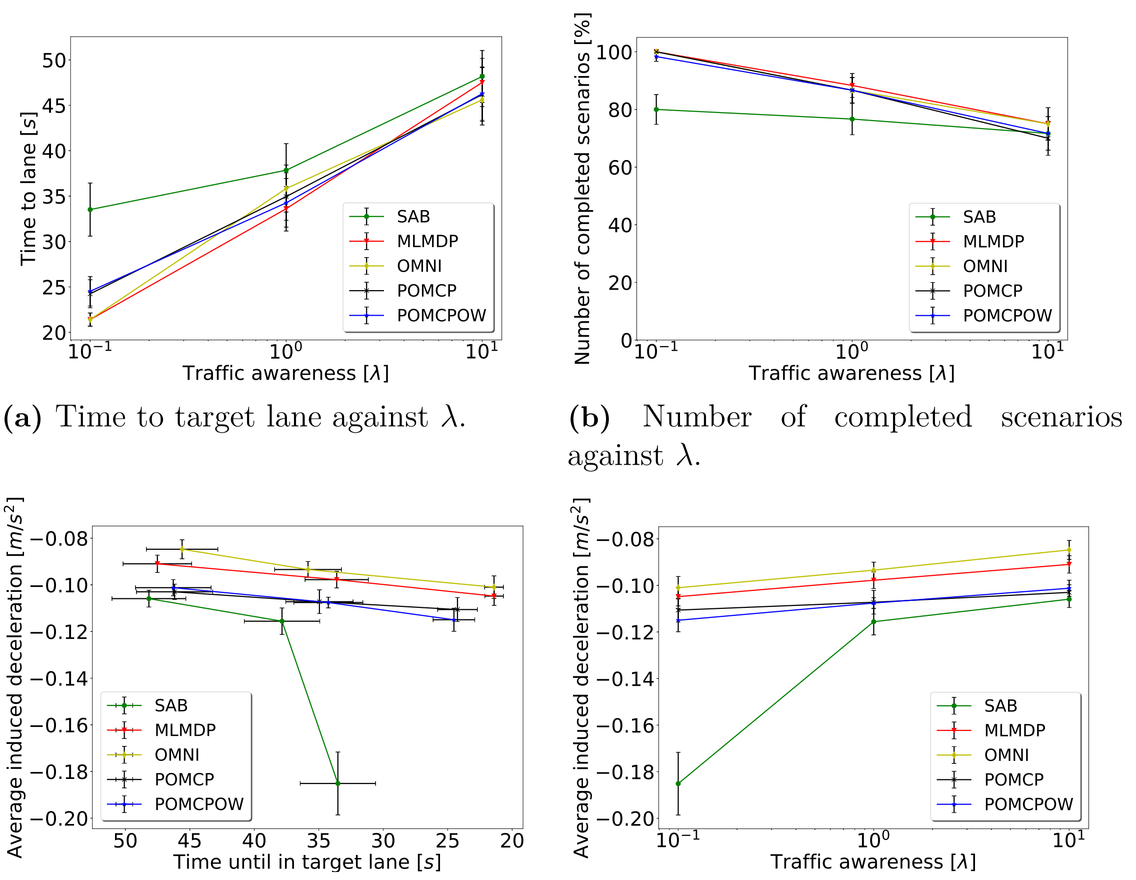
The right panel shows the total reward, i.e., the total reward from the objective function, for each algorithms plotted against the λ-values. Here it can be seen that the MLMDP algorithm is performing well. For $\lambda = 1$ it even outperforms the OMNI-solver, which is supposed to work as an upper bound. This may seem unreasonable, and will be discussed in Section 5.1.2.



(a) *The flow and lane rewards.* **(b)** *Total reward against λ.*

Figure 4.3: *Panel* **(a)** *shows how the different algorithms perform when run for 1000 searches during the exit lane scenario. The simulations are run for three different values of $\lambda \in \{0.1, 1, 10\}$. The result is plotted with respect to the two objectives denoted by the flow- and lane reward. Panel* **(b)** *shows the total reward from these simulations plotted against λ.*

In Figure 4.4 the different algorithms are compared. Panel **(a)** shows the average time it took for the ego to reach the target lane as a function of λ. For small λ-values the OMNI- and MLMDP-solvers reach the lane fastest. When $\lambda = 1$ the MLMDP-solver is generally faster than all the other algorithms, and when $\lambda = 10$ the OMNI-solver is fastest. Panel **(b)** shows the success rate of reaching the target lane. All algorithms perform well, except SAB when the value of λ is small. This panel also show how the success rate decreases as λ is set to larger values. In Panel **(c)** the physical results are presented in a similar fashion as for the reward functions. The average time it took for the ego to reach the target lane is plotted as a function of the average induced braking that occurred while doing so. The results are very similar to those in Figure 4.3. Lastly, panel **(d)** shows how the induced braking scales with λ. This figure shows that the OMNI-solver is inducing the least amount of braking on the other participating vehicles. Furthermore, the MLMDP outperforms both the POMCP and POMCPOW with respect to this objective.



(a) Time to target lane against λ.

(b) Number of completed scenarios against λ.

(c) Induced deceleration against time to target lane.

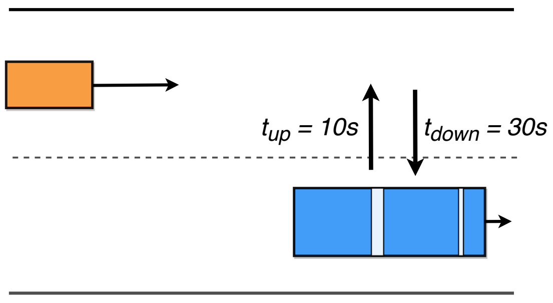
(d) Induced deceleration against λ.

Figure 4.4: *The results from the algorithm comparison, presented in physical variables. The different panels show how the different algorithms perform when optimizing different objectives. For each line, the nodes correspond to different values of* λ. *These figures show the rewards for the values* $\lambda \in \{0.1, 1, 10\}$.

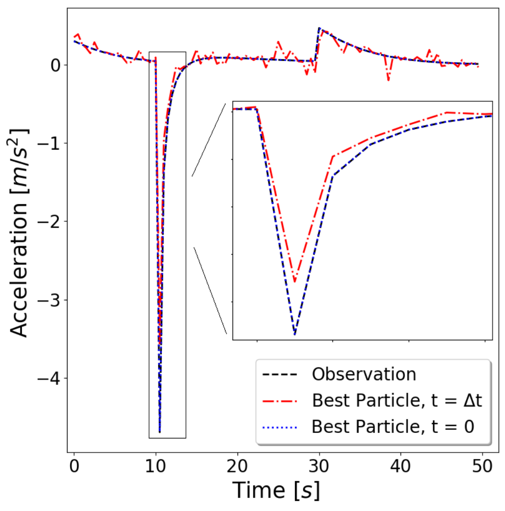
The error bars in Figure 4.1, 4.2, 4.3, and 4.4 show the standard deviations of the mean, i.e., $e = \sigma_{\text{residuals}}/\sqrt{N_{\text{samples}}}$, where N_{samples} is the number of evaluated simulations, and $\sigma_{\text{residuals}}$ is the standard deviation of residuals of the samples. The residuals were computed by partitioning the total variance into explained and residual variance, i.e., $SS_{\text{total}} = SS_{\text{residuals}} + SS_{\text{explained}}$. The explained variance, $SS_{\text{explained}}$, is the variance of the rewards which is resulting from switching between different simulation cases.

4.3 Predictions using a particle filter

During the simulations a particle filter was used to estimate the behaviours of the surrounding vehicles, i.e., their specific parameters used in the IDM- and MOBIL model. To examine how well the particle filter performed, a test scenario was set up, depicted in panel **(a)** in Figure 4.5. A faster car is approaching the truck from behind in the left lane. 10 seconds into the simulation the truck suddenly performs a lane change to the left, and is thus forcing the car to brake. When the car has been braking and adapting to the situation for about 20 seconds, the truck changes lane back to the right lane. In panel **(b)** the acceleration profile of the car is plotted together with the acceleration of the particle in the particle filter with the highest weight. The red dotted line in the graph is a single step-prediction made by the same particle. It predicts the acceleration of the car for the next time step, i.e. 0.5 seconds into the future.



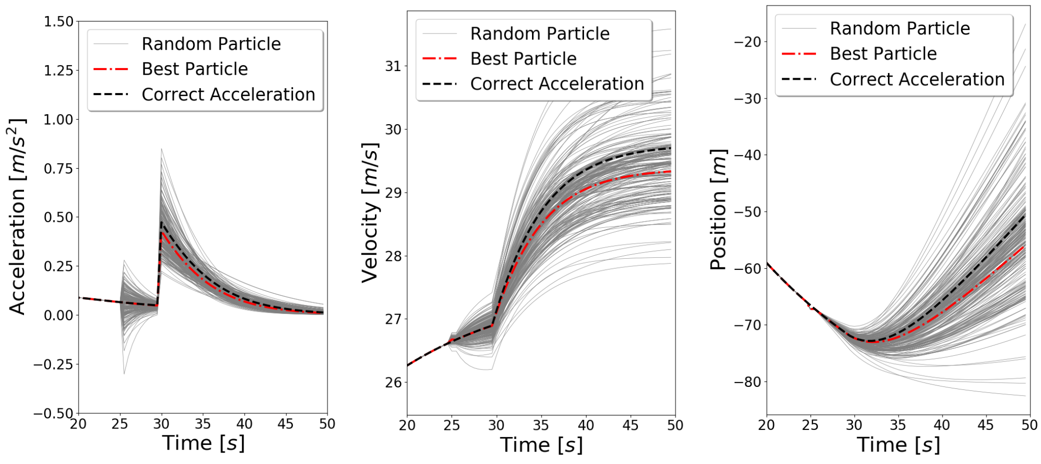
(a) Traffic scenario



(b) Acceleration of car

Figure 4.5: **(a)** *The scenario used to test the particle filter. A faster vehicle is located behind the ego in the neighbouring lane. 10 seconds into the simulation, the ego vehicle is instructed to change lane in front of the vehicle, and then at 30 seconds, change back again.* **(b)** *The acceleration profile of the car. The black curve represents the real acceleration, the blue curve represents the best acceleration produced by the particle filter in the current time step. The red curve shows how the particle with the highest weight in the particle filter predicts the acceleration for a single time step forward.*

Figure 4.6 shows how well the particle filter is able to predict the vehicle's motion. The acceleration profile is considered just before the truck makes the second lane change (30 seconds into the simulation). The real acceleration of the car, and the particle with the largest weight are plotted with thick black and red lines, respectively. These are plotted together with the predictions of the rest of the particles from the particle filter. In the middle panel, and the right panel in the same figure, there are also predictions of the associated velocity and position of the car.



(a) Acceleration **(b)** Velocity **(c)** Position

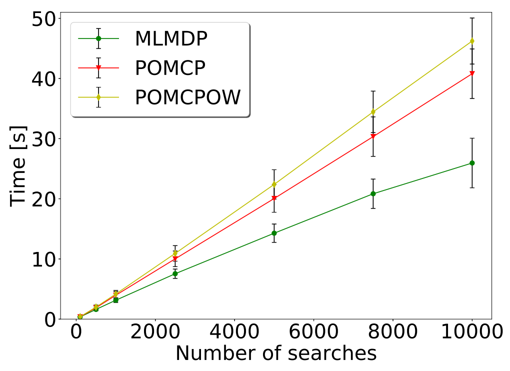
Figure 4.6: *The acceleration, velocity and position of the car from the scenario in Figure 4.5. The black dotted lines represents the real data, the red represents the prediction of the particle with the highest weight at time t=25. These are compared to the whole set of particles contained in the particle filter, plotted in grey.*

4.4 Time complexity of the MCTS algorithms

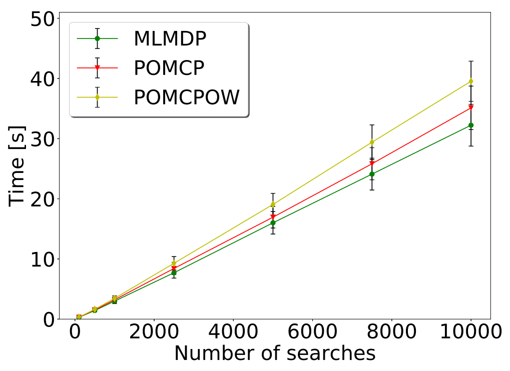
In the upper panels of Figure 4.7 it is shown how the time complexity for the different algorithms scales with the total number of tree searches. The time on the y-axis is the time spent on the tree search algorithm, which includes creating the search tree, and to compute the new belief update for the latest receives observation. One important thing to have in mind when studying these graphs is that the time is computed by using *our* implementation of the algorithms, which does not fully represent the real time complexity of the algorithms (since little effort has been made to optimize the run times). The graphs rather work as a comparison for the relative times between the algorithms.

The lower panels in Figure 4.7 show the depths of the search trees that the algorithms were able to create, which is based on the number of searches spent in the tree. The depth is defined as the maximum number of decisions taken for one of the nodes in the tree. The results show a clear logarithmic trend in the required number of searches to reach a certain depth. Along with the data, a fitted a logarithmic trend is shown by using the least squares error method. The POMCP solvers are very similar, and reach about the same depth. However, the MLMDP algorithm shows a much higher logarithmic growth.

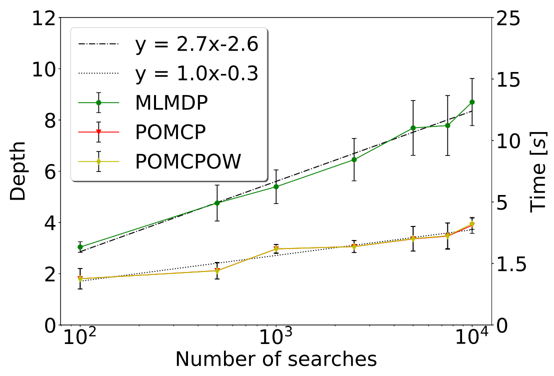
The two left panes in Figure 4.7 where computed when using a dynamic time horizon (this is briefly explained in section 3.3). The predictive time horizon was set to 25 seconds, and thus the maximum depth of the search tree was 10 nodes deep. The two right panels on the other hand used a static time step at 0.5 seconds. With the same time horizon of 25 seconds, the maximum depth of that tree was instead 50 nodes deep. The implications of this are discussed further in section 5.1.4.



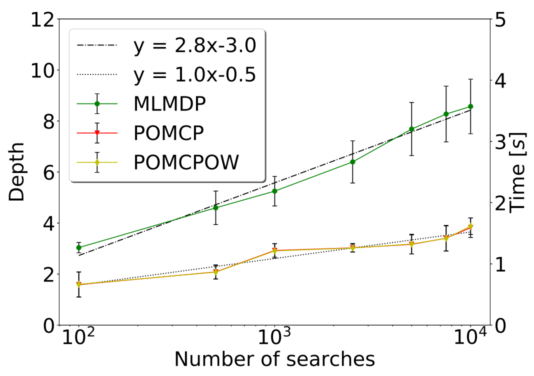
(a) Time complexity for the dynamic time horizon.



(b) Time complexity for the linear time horizon.



(c) Search tree maximum depth for a dynamic time horizon.



(d) Search tree maximum depth for a linear time horizon.

Figure 4.7: *The upper panels show the average run time complexity of our implementation of the different algorithms, and the lower shows the average depth that the algorithms reached in their respective search tree. The error bars in the figure is the standard deviations of the mean, i.e.,* $e = \sigma_{sample}/\sqrt{N_{samples}}$*, where* σ_{sample} *is the standard deviation of the samples, and* $N_{samples}$ *is the number of simulations. The panels to the left used a dynamic time horizon, the panels to the right used a linear time horizon. The graphs show how the POMCP solvers are more time intensive for larger searches, while also not reaching as deep down in the search tree as the MLMDP.*

Chapter 5

Discussion

This chapter analyses the results of the simulations that were presented in the previous chapter, and discusses our implementation of the problem.

5.1 Analysis of the results

During the next subsections the results from Chapter 4 is discussed. The discussion is performed in the same order as the results were presented, i.e., how the number of searches influences the algorithms, comparison of the algorithms, analysis of the particle filter, and the suitability of the algorithms as online optimization methods.

5.1.1 Effects on the performance of a MCTS depending on the number of searches

By varying the number of searches in the Monte Carlo tree search the agent was able to find much better solutions, i.e., solutions which returned a higher reward. However, an interesting property of Figure 4.1 in Chapter 4 is that the difference in reward between the different number of searches is larger for small λ-values, compared to when λ is large. This difference indicates that the problem of reaching the desired lane as fast as possible contains a lot of long term planning, compared to the problem of avoiding to induce deceleration in the surrounding traffic. Another interesting property is that there is a notable decrease in total reward when $\lambda = 1$, i.e., when the objectives have equal weight. This means that it is very difficult to solve both tasks simultaneously, which is known from the fact that the curves show a Pareto front. An either very large, or small, value of λ means that the majority of the reward is based on only solving one of the tasks. Thus we expect to see a higher reward for these simulations. It is also interesting that in this particular case the algorithm performs equal when run for 1000 and 100 searches (this is the point on the Pareto front in the left panel where 100 searches looks to have performed better than the 1000 searches). This may be an indication of that there is a need of more searches in the tree in order to find good policies in such a difficult scenario.

When the number of searches is large, panel **(a)** in Figure 4.2 shows that the ego vehicle manages to arrive faster to the target lane, except for when λ is small. An interesting fact is that when λ is really large, searching 10 or 100 times reaches the lane faster than when searching 1000 times. But this does not necessarily indicate that they performed better, since when λ is large the main objective is not to get to the lane, but rather to avoid impeding the traffic.

The results in panel **(d)** in Figure 4.2 are promising, and one can see that when the algorithm samples 1000 tree queries, it was able to better prioritize the different objectives. This is determined by the steepness of the s-curve. The results from all search depths are s-shaped, but as the searches increases, the steeper the curve gets.

5.1.2 Performance comparison of the algorithms

In Figure 4.3 the simulation results are compared when running all algorithms for 1000 searches per decision. In the figure it can be seen that the MLMDP algorithm performs better than both POMCP and POMCPOW. The POMCP solvers construct a graph that contains an observation node layer, and as can be seen in Figure 4.7 both POMCP-DPW and POMCPOW require more searches to reach the same search depth compared to the MLMDP solver. As we previously discussed above, since there are many ways to get to the target lane, the search depth plays a crucial role in order to find a good policy. Due to the observational layer in the POMCP algorithms, they use searches on exploring each action multiple times, and as a result cannot find as many alternatives as the MLMDP. However, the observation node layer means that the POMCP solver will be able to plan while taking several different driver intentions into account, and therefore it should estimate the actions better than an MDP solver. This means that as the number of searches increases the POMCP-DPW and POMCPOW should technically reach an asymptotically higher reward than what is possible for MLMDP, since the behaviours in the MLMDP will never be completely correct. We only had the computational power to compare these algorithms by sampling 1000 searches. However, to be fair one might want to go much deeper than that in order to compare these algorithms. Other similar work samples POMCP-DPW 2500 times and find that it outperforms an MLMDP that only samples 500 searches [SHK17], however, these results can be a bit biased from the difference in search tree sizes.

The results in Figure 4.3 were put into context by plotting them with an upper and lower bound: the OMNI and SAB solvers. However, in the right panel we actually see how the MLMDP beat the OMNI solver in the case when $\lambda = 1$. This should not be possible, since the OMNI algorithm should theoretically be an upper bound for these algorithms. Why this happened can be a result of a couple of different reasons. The simplest reason would be that the sample size was not large enough, due to the computational demand of these simulations. This is also the reason why there are only three λ-values in this test instead of six values as in Figure 4.1. However, the most likely reason behind why the MLMDP-algorithm outperformed the upper bound has to do with the time horizon. During the simulations the time horizon of the algorithms' predictions were 25s. However, the total length of the simulation scenario was 75s. Thus a good policy during the first 25 seconds does not necessarily mean a good policy when considering the whole scenario. The main difference between the two algorithms is that the OMNI solver is much better to account for the vehicles' behaviours, which can be seen in panel **(d)** in Figure 4.4, where OMNI beats MLMDP in this category. Since the MLMDP solver does not exactly know how a vehicle would behave if the ego would change lane in front of it, it sometimes misjudges the situation. Since there is no abort action in the set of actions, once the truck has started a lane change, it has to complete it. This results in the ego vehicle being closer to the target lane, a decision which should return less reward on a $25s$ time scale, but is ultimately beneficial on a $75s$ time scale. If the ego vehicle reaches the target lane, it is desired that it also stays there during the rest of the simulation. Therefore the ego vehicle does not get any penalties for inducing braking actions when it has reached the lane. As a result, the objective of reaching the target lane is therefore implicitly more important compared to avoiding inducing braking actions. Hence a misjudgment of the MLMPD-algorithm can lead to a higher cumulative reward than for the OMNI-algorithm.

To support this argument, it can be seen in panel **(a)** in Figure 4.4, that when $\lambda = 1$, the OMNI solver takes longer time getting to the target lane compared to MLMDP, POMCP and POMCPOW. Additionally, in panel **(d)** it can be seen that the OMNI solver induces less braking actions compared to all other algorithms, and panel **(b)** shows the percentage of succeeded lane changes as a function of λ. Here we can also see that the MLMDP succeeds in finding the target lane more often than OMNI when $\lambda = 1$.

5.1.3 Analysis of the particle filter

Since there are infinite sets of parameters that can produce the same acceleration when using the IDM and MOBIL model, it is a difficult task to find what behaviour caused the observed acceleration. To simplify the problem of sampling behaviours a correlation factor, $\rho \in [0, 1]$, was introduced. A large correlation factor implies that a vehicle with a high velocity also has the tendency to accelerate faster and keep a smaller time gap to the preceding vehicle. The simulations were run with $\rho = 0.75$, which means that the filter was able to roughly estimate the whole behaviour of a driver, even when there was no vehicle in front of the driver, so that only two parameters are active. In this case the only measurable variables are the maximum acceleration, a, and the desired speed parameters, $\dot{x}^*$, in the IDM. This was visualized in panel **(b)** in Figure 4.5, where the particle filter predicted quite accurately the deceleration of a vehicle at $t = 10$, even though these behaviour parameters have not been observed. But, even though a particle's acceleration is close the observation, it does not necessarily mean that the particle's behaviour contains the real underlying parameters. A particle which predicted an acceleration close to the observed value, could differ from the real accelerations only 0.5 s later.

In Figure 4.6 the whole particle filter is visualized. An interesting result here is that there are actually other particles in the collection that perform better than the particle with the highest weight, i.e., the particle with the most accurate prediction at that time. However, this way of filtering, where the filter finds particles that fit best compared to the current acceleration, may be sub-optimal. It can be better to instead find a particle that is able to predict the five, or ten, most recent observations. We did not use such a particle filter in this thesis and such methods are also left for future works.

5.1.4 MCTS algorithm as an online optimization method

This section will evaluate the results connected to whether any of these algorithms are possible to run online as a tactical decision making-algorithm in a heavy articulated vehicle.

Time complexity and search depth

The upper two panels of Figure 4.7 show how the time complexity of the different algorithms scale with the number of searches. The left panel shows the complexity for a dynamic time horizon that was 10 time steps deep with a total simulation length of $25s$. This plot shows quite an interesting trend, as for a small number of searches the difference in run times between the algorithms was small. This is because the only computational difference between the algorithms is that the POMCP solvers uses an additional observation layer which add a couple of computations, and furthermore the POMCPOW algorithm also has to weigh the particles.

As the number of searches increases one can see how the MLMDP algorithm start to deviate from the other algorithms. The reason behind this is not due to the complexity of the algorithms, but rather it is an implication of the dynamic time horizon. This phenomenon is due to the MCTS *simulation* step, where the algorithm selects previously calculated states until it reaches a leaf node, and then performs a rollout simulation for the rest of the time horizon. The computationally expensive part of the MCTS simulation step is this rollout simulation, which simulates the generative model, i.e., the traffic models, ego vehicle etc. Thus, the further the simulation step can traverse the tree by using previously calculated nodes, the lower the computational complexity becomes. In Figure 4.7 one can see that the MLMDP algorithm reaches a depth of $d \approx 9$ nodes, out of 10, and thus a majority of the time spent in the MCTS simulation consists of selecting previously calculated nodes. One can also see how this becomes more beneficial as the number of searches increases, since the depth of the tree increases. This also affects the other algorithms, but compared to the MLMDP algorithm they do not reach as far down the tree and the effect is not as noticeable.

The upper right panel in the same figure shows the same analysis, but, these simulations used a 50 nodes deep linear time horizon. The simulation time between each node was $\Delta t = 0.5$ meaning that the length of the horizon also was $25s$. Here it can be seen that the MLMDP does not divergence as much from the other algorithms. This is because none of the algorithms reach a significant depth—the MCTS simulation step mainly simulates the computationally expensive rollout policy.

The lower panels of Figure 4.7 show that there was a difference between the MLMDP and POMCP algorithms in terms of the depth of the search trees. This was a result of the history node tree used in the POMCP algorithms, where each action node has k child observation nodes. Hence these algorithms require more searches before they reach a significant depth in the search tree. The figures show how the logarithmic growth for the MLMDP algorithm is about 3 times as large when compared the POMCP solvers. This was expected from the DPW settings that were used ($k' = 3$, $\alpha = 0.1$). This means that the POMCP solvers have *at least* 3 observation nodes per action node, and should thus have a growth three times smaller.

The logarithmic growth of the depth plays a crucial role in how far the algorithms are able to predict, since the depth is directly proportional to how far into the future the algorithms are able to simulate. One could reason that when considering the exit lane scenario it might be beneficial for an algorithm to plan past *at least* one lane change, which with the settings in table 3.1 lasts for $5s$ if performed in the initial time step. With the dynamic time horizon, this correspond to searching at least $d = 4$ nodes deep in the search tree. In these figures it can be seen that the only algorithm able to plan that deep in the tree is the MLMDP, where the POMCP solvers barely reach a depth of 3 nodes. The required amount of searches to reach a depth of $d = 4$ can be computed by using linear regression, see the lower panels of Figure 4.7. The depth is calculated by $d = -0.5 + 1\log_{10}(N_{\text{searches}})$, which gives $N_{\text{Critical}} = 10^{d_{\text{critical}}+0.5}$. Thus one would have to sample somewhere around $N_{\text{Critical}} = 30000$ tree searches to be able to plan past an initial lane change with a POMCP solver, when using these UCB1 and DPW settings. This could be one of the main reasons why the MLMDP algorithm outperformed the POMCP algorithms—the POMCP algorithm need more samples to perform well.

Online decision-making

The focus in this project was to examine different tree search methods, and evaluate whether they are feasible to run online. All code used in the thesis was written in Python, which was chosen because it is a relatively easy high level programming language, and is thus efficient for rapid prototyping. However, Python is not a high performance language, and to further improve the run times of this program one could consider writing the code in for example C++. Furthermore, MCTS is also a parallelisable algorithm [CWvdH08]. In Figure 4.7 we see that our implementation of MLMDP is able to sample 1000 searches in $\sim$3s. The results in Figure 4.1 also showed that 1000 searches could potentially be enough in the decision-making algorithm in order to find good policies. This means that this implementation is roughly a factor 3 too slow. Thus, a more efficient implementation running on multiple threads would be feasible to run online, given a decision time of $1s$.

5.2 Implementing a highway driving scenario

The physical models for the ego vehicle, as well as the surrounding traffic were simplified. Simplified models means faster computations, and therefore there is a trade-off between precision and speed. The decisions taken in the *Tactical decision-maker* (see Figure 3.1) are supposed to work as a guide for the automated vehicle, and as long as the models used in the decision-making algorithm is made with respect to the capabilities of a truck, it should be enough. What is important is that the *Trajectory planner* (from the same figure) gets reasonable suggestions, that it can execute.

Reward model - Modelling objectives

It was showed in Chapter 4 how the value of λ in the reward function (3.11) effects the agent's decision making, and that λ can be used to prioritize different objectives. For example, to prioritize getting off a highway, λ can be set to have an inverse relationship with respect to the distance to an exit ramp. However, this reward function showed to be sub-optimal when trying to model *mandatory* lane changes. A *mandatory* lane change means that the ego must change lane in order not to crash or break any laws, i.e., in the case when two different lanes merge or there is an obstacle on the road. During the thesis we attempted to model this type of lane changes by introducing large penalties or rewards at different locations on the road, forcing the agent to seek policies that avoids, or collect these. However, these modifications of the reward function showed to have bad properties on the tree search, where it affected the UCB1 tuning, which resulted in deep and narrow branches, and often sub-optimal policies. However, we avoided investigating these cases further, and leave these questions for future work.

Driver models

The driver models that we chose to use when predicting the traffic were the IDM and MOBIL model. The reasons for this were that they are simple to implement, and that they are used frequently within traffic simulation applications. The IDM model has for example been shown to reproduce realistic collective dynamics, and in smaller simulations it shows plausible microscopic accelerations of the single drivers [THH00]. But, the prediction of human drivers has been shown to be a difficult problem, and even though the models reproduce some observed traffic properties, they are not specifically made for predicting the behaviour of single drivers. To model lane changes during highway driving, the predictions need to be in the range 10–25s, which is a long time in vehicle prediction context, and there will most likely be a discrepancy between the distribution of possible outcomes from the models and real drivers. We have used these driver models as proof of concept, but future work could be to replace these models by more accurate models, e.g., driver models derived from learning from real data of human drivers.

5.3 Implementing a Monte Carlo tree search

The implementation of the problem uses the UCT, which is a Monte Carlo tree search algorithm combined with the UCB1 algorithm. However, there are many other alternatives to handle the trade off between *exploration* and *exploitation*. In [BPW$^+$12] other methods are proposed, that expands the UCB1 bound to also incorporate the variance of the different branches. However, this is not investigated in this thesis.

Another thing that has to be specified when implementing a MCTS is what branch to select after the tree search is done. The alternatives is to either select the most popular branch (i.e., based on the number of searches on that specific branch), or the branch with the highest average reward. Both these methods have shown to be effective, and which one is the best depends on the problem at hand. We tested both alternatives, and did not find any significant difference between the resulting decisions. Therefore we decided to select actions based on the most popular branch. There are mainly two reasons behind this choice: if a branch is sampled enough so that the second most popular branch will not be able to overtake it, then we can stop the tree search and select this branch. In a lot of cases this improves the run time of the code significantly. Another reason for selecting the most visited branch is that it is more robust for a small number of searches. A branch containing a relatively high number of searches corresponds to a branch that has been a good option for a a lot of searches, and even though a bad decision is made far down in the tree, that damages all action values up to the root node, the branch will be still chosen as the best.

The method of selecting the most popular branch does also have its drawbacks. Consider a case, where after $N_{\text{Searches}}/2$ searches, one of the less visited branches suddenly finds a policy that is receiving extremely good results. Then even though the new branch starts to get more promising, the number of samples in this branch can be so low that it is impossible for it to catch up. There are methods designed to solve this problem [BPW$^+$12], where, for example, if the Q values of the two best actions are too similar, the tree search continues. But this was not useful in this implementation, since the benchmark ran all algorithms for an equal number of searches.

Rollout policy

The best possible rollout policy should resemble an optimal highway driving strategy, so that newly expanded nodes in the tree get evaluated as accurately as possible. During simulations we wanted the ego to reach the exit of the highway. Therefore the rollout policy was instructed to keep the ego vehicle's desired speed and time gap, and if there was a possibility of doing a lane change to the right, to do it. This rollout policy therefore encouraged the ego vehicle to find multiple ways of reaching the target lane, and thus promoted planning. One way to evaluate the rollout policy is to try the root node's allowed actions only once during the tree search, and let the rollout policy evaluate them. This was done during the analysis of the number of searches in Section 4.1, and can be found in Figure 4.1. Since the rollout performed quite well, where it provided a convex Pareto curve, it can be concluded that it at least resembles highway driving, and thus can work as a rollout policy for this problem.

5.4 Hyperparameters and the simulation environment

The implementation of the program involved the determination of a lot of parameters, such as the exploration constant in the UCB_1-algorithm, the DPW-parameters, particle filter size, and many more. Naturally these should be properly tuned in order to draw any rigorous conclusions from the results. However, we concluded that we did not have time to perform this analysis during the time frame of the thesis. Therefore, we set a lot of parameters according to values found in the literature, mostly from [SHK17]. The rest were determined experimentally by trial and error. For future work this is something that needs to be investigated further.

The simulations were created to represent interesting, and relevant highway scenarios, which also demanded a lot of planning in order to perform well. To accomplish this, the scenarios had to be exaggerated compared to what a vehicle might normally experience during highway driving. The scenario in Figure 3.6 for example involves 10 vehicles and four lanes, which is quite extreme compared to the expected scenarios when driving on Swedish highways, which usually contains two lanes. However, we assume that if the algorithms perform well in such extreme test cases it should also perform well in more realistic and simpler ones.

Chapter 6

Conclusion and future work

This chapter summarizes the results and discussions of the previous chapters, and answers the initially proposed thesis objectives. The thesis is ended with suggestions on how to develop the proposed method further.

6.1 Formulating highway driving as a POMDP

The partial observability of the POMDP allows a general problem formulation, which is necessary in order to model the uncertain aspects of highway driving, e.g., the drivers' intentions. During the trials of the investigated method, the MLMDP algorithm proved to be the best solver, which outperformed the more sophisticated POMCP-DPW and POMCPOW algorithms. It seemed like the added complexity of the POMCP algorithms outweighed their benefits, at least when being sampled for 1000 tree searches.

Furthermore, the mathematical model used in this thesis provides a transparent and easily tunable method. If the agent for example makes a 'bad' decision it is easy to reenact the scenario, and backtrack precisely where things started to go wrong. This is something that is quite attractive given the ethical aspects of autonomous vehicles.

6.2 Finding an online policy using Monte Carlo tree search

One of the objectives of this thesis was to find an algorithm suited for real time decision-making in an automated vehicle. These experiments show that this is possible with a MCTS-method, and the MLMDP algorithm seems to be a suitable candidate. This is because the MLMDP is capable of reaching larger search depths to correctly evaluate lane changes and other high-level maneuvers. Furthermore, with some code optimization, the proposed method can reach real time performance.

6.3 Future work

To proceed with this work, the next step would be to test the algorithms in a less controlled simulation environment. Most interesting, though, would be to see how well the algorithms performs in real traffic. Even though there would be a mismatch between the simulated predictions and reality, the agent could possibly still be able make good decisions.

The simulation environment should in the future include realistic road models, and more accurate models of a heavy articulated vehicle. In order to make fair validations of the algorithms they should also be evaluated on a larger set of scenarios, and, furthermore, they could also include some sensor measurement errors into the belief state.

The prediction model could in the future be completely disconnected from the tactical decision making module. To make accurate predictions about the traffic flow is a complex problem, and it is likely that the use of a traffic model learned from real data could predict the future better than IDM and MOBIL, when compared to real data. Since the predictions aim at describing human behaviour (something that can be considered to be a black box), a black box solution might be the best way to go.

The reward function is essentially what governs the behaviour of the agent. This means that it should be able to incorporate all the aspects about how to properly drive an autonomous truck. For example, apart from the two objectives that we chose to model in this thesis, it should also take the velocity of the ego vehicle into account, and maybe include penalties for too rapid movements, since these might be dangerous to perform with an articulated vehicle.

The idea of using a particle filter as a belief updater was taken from relevant literature. But, as discussed in Chapter 5, the implementation of this filter was not optimal, mostly due to the complexity of the IDM and MOBIL model. Future work would include to study alternative algorithms, such genetic algorithms or similar, which can perform better for this task.

Furthermore, we did not perform any extensive tuning of the hyper parameters used in the MCTS algorithms. We set them with inspiration from previous work within the area, and what we found to work best for our own simulations. The same applies to the rollout policies, as these are crucial for the MCTS algorithms performance.

The implementation of the proposed algorithms was made primarily to investigate if they were able to solve the decision-making problem. Therefore, to make the code run in real time, and thus to be used as an online method, it should be thoroughly optimized.

Bibliography

[ACK12] M. Ardelt, C. Coester, and N. Kaempchen. Highly automated driving on freeways in real traffic using a probabilistic framework. *IEEE Transactions on Intelligent Transportation Systems*, 13:1576–1585, 2012.

[Bar13] S. Barradas. The real cost of trucking – per mile operating cost of a commercial truck. `https://www.thetruckersreport.com/infographics/cost-of-trucking/`, 2013. Accessed: 2018-06-05.

[BCK17] M. Bouton, A. Cosgun, and M. J. Kochenderfer. Belief state planning for autonomously navigating urban intersections. *CoRR*, 2017.

[BGD11] S. Brechtel, T. Gindele, and R. Dillman. Probabilistic MPD-behavior planning for cars. *14th International IEEE Conference on Intelligent Transportation Systems*, 2011.

[BGD14] S. Brechtel, T. Gindele, and R. Dillmann. Probabilistic decision-making under uncertainty for autonomous driving using continuous POMDPs. *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 392–399, 2014.

[BPW$^+$12] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P.I. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE transactions on computational intelligence and AI in games*, 2012.

[CD11] A. Couëtoux and H. Doghmen. Adding double progressive widening to upper confidence trees to cope with uncertainty in planning problems. *The 9th European Workshop on Reinforcement Learning*, 2011.

[CWvdH08] G.M.J.B. Chaslot, M.H.M. Winands, and H.J. van den Herik. Parallel monte-carlo tree search. *Computers and Games. CG 2008. Lecture Notes in Computer Science*, pages 60–71, 2008.

[Int14] SAE International. Automated driving levels of driving automation are defined in new SAE international standard j3016. `https://www.smmt.co.uk/wp-content/uploads/sites/2/automated_driving.pdf`, 2014. Accessed: 2018-06-05.

[Koc15] M. J. Kochenderfer. *Decision Making Under Uncertainty.* MIT Press, 2015.

[KTH07] A. Kesting, M. Treiber, and D. Helbing. General lane-changing model mobil for car-following models. *Transportation Research Record: Journal of the Transportation Research Board*, 2007.

[Nil17] P. Nilsson. Traffic situation management for driving automation of articulated heavy road transports. *Thesis for the degree of doctor of philosophy in machine and vehicle systems*, 2017.

[Pha18] 'Phantom Auto' will tour city., Google News Archive. 8 December 1926. Retrieved 8 May 2018.

[RPPCd08] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research 32*, pages 663–704, 2008.

[SHK17] Z. Sunberg, C. Ho, and M. J. Kochenderfer. The value of inferring the internal state of traffic participants for autonomous freeway driving. *CoRR*, 2017.

[SHM+16] D. Silver, A. Huang, C. J. Maddison, Arthur Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and Hassabis D. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[SHS+17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, A. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *eprint arXiv:1712.01815*, 2017.

[SK17] Z. Sunberg and M. J. Kochenderfer. POMCPOW: An online algorithm for POMDPs with continuous state, action, and observation spaces. *CoRR*, 2017.

[SV10] D. Silver and J. Veness. Monte-carlo planning in large POMDPs. *Advances in Neural Information Processing Systems (NIPS)*, 2010.

[Tab17] V. Tabora. Tesla enhanced autopilot overview - L2 self driving HW2. `https://medium.com/self-driving-cars/tesla-enhanced-autopilot-overview-l2-self-driving-hw2-54f09fed11f1`, 2017. Accessed: 2018-05-30.

[THH00] M. Treiber, A. Hennecke, and D. Helbing. Congested traffic states in empirical observations and microscopic simulations. *Physical Review E 62*, pages 1805–1824, 2000.

[UM13] S. Ulbrich and M. Maurer. Probabilistic online POMDP decision making for lane changes in fully automated driving. *16th International IEEE Conference on Intelligent Transportation Systems*, pages 2063–2067, 2013.

[UM15a] S. Ulbrich and M. Maurer. Situation assessment in tactical lane change behavior planning for automated vehicles. *18th International IEEE Conference on Intelligent Transportation Systems*, pages 975–981, 2015.

[UM15b] S. Ulbrich and M. Maurer. Towards tactical lane change behavior planning for automated vehicles. *18th International IEEE Conference on Intelligent Transportation Systems*, pages 989–995, 2015.