VU University Amsterdam

Optimization of ideal racing line

Using discrete Markov Decision Processes

BMI paper

Floris Beltman
florisbeltman@gmail.com

Supervisor: Dennis Roubos, MSc

VU University Amsterdam
Faculty of Sciences
Business Mathematics and Informatics
De Boelelaan 1081a
1081 HV Amsterdam

August 2008

Floris Beltman
florisbeltman@gmail.com

Preface

During the Master Business Mathematics and Informatics (BMI) at the VU University Amsterdam, the student has to write a BMI paper. This paper is mainly a literature study with a (small) application. The objective is to demonstrate the student's ability to describe a problem in a clear manner for the benefit of an expert manager.

I finished my BMI paper at the end of the Master, which was a bit of a struggle. I would like to thank my supervisor, Dennis Roubos, to help me out during this process.

Floris Beltman
Amsterdam, 2008

Abstract

This paper is a literature study on optimization algorithms suitable for computing an ideal racing line for a given racetrack. The optimization algorithms described are mainly from the field of Dynamic Programming, with a focus on Markov Decision Processes. Furthermore the ideal racing line problem is modelled for solving it using algorithms within discrete Markov Decision Processes (MDPs).
The continuous properties of the ideal racing line problem make it not easy to solve the problem using discrete MDPs. The problem could be solved by applying good discretization techniques. This will eventually lead to a satisfying solution, but this can lead to an other difficulty, namely the curse of dimensionality.

Contents

Chapter 1

Introduction

Car racing can be considered as a sport in which lots of decisions have to be taken. For example, choices on the kind of tires, the fuel level, springs and wings settings are very important and determine the behavior of the car. One can start with a default setting and can drive a couple of practice laps to gather data which can then be used to adjust the settings. However, it is very difficult to say something about the ideal racing line.

Definition 1.1 *The ideal racing line is the fastest path a vehicle can take through a specific corner, series of corners, or track.*

Note that the fastest path is essentially different from the shortest path. The fastest path can be the shortest path, but in general they will be different. Taking wide(r) corners will result in a longer path, but it may allow the car to drive faster than taking thight corners.

The problem of finding the ideal racing line for a set of race tracks forms the starting point of this paper. If we want to solve this problem, we have to use an optimization algorithm. The heart of this paper is a literature study on several known optimization algorithms which can be suitable for solving this problem. These algorithms will be described in Chapter 2. The focus is on algorithms wihtin the field of *dynamic programming*, with (discrete) Markov decision processes (MDPs) as the major optimization algorithm.
The ideal racing line problem is described in some more detail in Chapter 3. Furthermore, the problem is modelled to solve it using a discrete Markov Decision Process algorithm.
The actual implementation of this algorithm, thus the computation of the ideal racing line is not in this paper. The focus is on the literature study of optimization algorithms which may be suitable for solving the problem.
The last chapter, Chapter 4, is a short discussion about the optimization algorithms and their potential in solving the ideal racing line problem. Furthermore, there are some thoughts about the possible performance of the solution using a discrete MDP.

Chapter 2

Optimization Methods

2.1 Introduction

The ideal racing line problem can be solved using known optimization problems. While driving the car along the racetrack, the driver of the racecar has to take decisions. Do I have to brake or accelerate? Pull the steeringwheel to take a corner right now or wait one second? These decisions have both immediate and long-term consequences. Pulling the steerwheel right now allows the driver to take the corner immediately, but it also effects the position where the car leaves the corner several seconds later. Both consequences impact the posibility to drive a fast laptime.
So the driver continuously has to make a decision which action to take (doing nothing is also an action). This can be describe as the *sequential decision making model*, symbolically represented in Figure 2.1. First the driver observes in which state the car is, for example the position on the track, the distance to the next corner and the current speed of the vehicle. Then he chooses an action to execute, which is in most cases a set of actions (e.g., braking and steering to the right). The action the driver chooses, produces two results: the system evolves to a new state (for instance a new position on the racetrack and/or a different speed of the car) and the decision gives a so-called reward (for instance the time and distance to the new position). The driver has to make a decision again in the new state, where the observed state can differ from the previous state and also the set of actions may be different (the driver can not accelerate when the car is at top speed for example). Also this action leads to a new state and so on until the final conditions are met (the car passes the finish line).

So each state the decision maker, which is in the race line problem the driver, chooses an action depending on the state where he is in at that moment. The quenstion is; which action or set of actions does the driver have to take? Which decisions has to be taken to drive the fastest laptime? Which *policy*[1] leads to the fastest path along the racetrack? There are several known techniques to find the optimal policy.

These decision models are often referred as *dynamic programming models* or *dynamic programs*. The expression "dynamic programming" will be used to describe an approach to solve

[1] A policy provides the decision maker with a prescription for choosing a typical action in any possible future state.

these models based on inductive computation. This chapter describes dynamic programming (Section 2.2) and some optimization methodes related to dynamic programming, which may be useful for finding the optimal policy (decisions) leading to the ideal racing line. Also an optimization method from another field is described, which may be useful for the racing line problem (Evolutionairy Computing, Section 2.6).

Remark This chapter does not contain mathematical proofs, such as the existence of optimal solutions or the converging of sums over an infinite time horizon. The equations and algorithms are taken from literature. For these proofs we will refer to the used literature.Furthermore, most of the time the proofs are not mentioned.
There are several reasons for this choice. First, the goal is to keep the paper readable, for mathematicians as well as for people with less knowledge about mathematics. Second, this chapter gives an overview of some optimizations methods. Therefore it is not necessary to include methematical proofs.

2.1.1 Applications

Besides the ideal racing line problem, there are a lot of other optimization problems one can think of where decisions have to be taken. Some fields where these optimization problems occur, which are widely studied and where dynamic programming methods are applied, are (from [9]):

- **Inventory management**: Sequential decision models have been widely applied to inventory control problems and represent one of the earliest areas of application. The scope of these applications ranges from determining reorder points for a single product to controlling a complex multiproduct multicenter supply network.
- **Communication models**: A wide range of computer, manufracturing and communications systems can be modeled by networks of interrelated queues and servers. Efficient operation of these systems leads to a wide range of dynamic optimization problems. Control actions for these systems include rejecting arrivals, routing, and varying service rates. These decisions are made frequently and must take into account the likelihood of future events to avoid congestion.
- **Maintenance and replacement problems**: Markov Decision Process models have been applied to a wide range of equipment maintenance and replacement problems. In these settings, a decision maker periodically inspects the condition of the equipment, and based

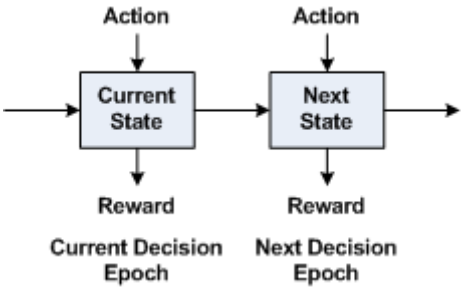


Figure 2.1. Symbolic representation of a sequential decision problem

on its age or condition decides on the extent of maintenance or replacement, if any, to carry out. Costs are associated with maintenance and operating the equipment in its current status. The objective is to balance these two cost components to minimize a measure of long-term operating costs.

- **Behavioral ecology**: Markov Decision Process models are becoming increasingly popular in behavioral ecology. They have been used in a wide range of contexts to gain insight into factors influencing animal behavior.

2.2 Dynamic Programming

The term *dynamic programming*, already mentioned in the previous section, is the name for a range of optimization methods/algorithms concerning dynamic decision problems. Dynamic refers to the notion of time; as mentioned earlier, the decisions taken (the actions) at some point in time have not only consequences for that moment, but also for future points in time. This is de crucial difference with lineair programming. Dynamic programming was developed by Richard Bellman in the fifties [1][5]. The *Bellman equation*, also known as the *Optimality equation*, refers to his contribution on this field. The principle of the Bellman equation is that it gives the optimal value of a state if the optimal value for the corresponding successor states is known. The formula can be applied recursively, to solve optimality problems.[2]

The following sections describe several solution methods which are linked to dynamic programming. The majority is about Markov Decision Processes (MDPs), for both the discrete and the continuous case. Furthermore the techniques (artificial) Neural Networks and Neuro-Dynamic Programming are described.

2.3 Markov Decision Processes

The sequential decision making model (Figure 2.1) is relevant for Markov Decision Processes (MDPs), which are an extension of Markov chains. The difference is the addition of actions (thus allowing choice) and rewards (giving motivation for that particular choice). If there was only one action, or if the action to take were fixed for each state, a Markov Decision Process would reduce to a Markov chain.

2.3.1 Definitions and Notations

First we introduce some definitions and notations, concerning MDPs. A Markov Decision Process model consists of five elements: decision epochs, states, actions, transistion probabilities, and rewards. Furthermore there are decision rules and so-called policies involved.

Decision epochs Decisions are made at points of time referred to as *decision epochs*. We denote T as the set of decision epochs and is a subset of the non-negative real line. It could either be a discrete or a continuous set. For the discrete case, time is divided in periods. For

[2] An example of the Bellman equation is given in Figure 2.2.

the continuous case, decisions can be made for instance at random points in time. Next to the difference between discrete and continuous decisions epochs, the set can also be either finite or infinite. A finite discrete decision epoch can be described as: $T = \{0, 1, 2, \ldots, N\}$ for some integer $N < \infty$.

States At each decision epoch, the system occupies a *state*, as earlier explained. The set of possible states will be denoted by S.

Actions If the decision maker observes the system in some state $s \in S$ at some point in time, he has the opportunity to choose an action. The action set he can choose from depends on the obeserved state s, and we will denote the chosen action by $a \in A_s$. Note that the action does not depend on the point in time.

Transition Probabilities As a result of choosing action $a \in A_s$ in state s at decision epoch t, the system state at the next decision epoch is determined by the probability distribution $p_t(j|s, a)$. This non-negative function denotes the probability that the system is in state $j \in S$ at time $t + 1$, when the decision maker chooses action $a \in A_s$ in state s at time t. We asume that

$$\sum_{j \in S} p_t(j|s, a) = 1. \tag{2.1}$$

Remark: The action set chosen can lead to a typical next state without any uncertainty, there is no randomness involved. Thus, for some action a the transition probability equals 1 for just one (new) state, and 0 for all other (new) states. These problems are called deterministic problems.

Rewards As a result of choosing action $a \in A_s$ in state s at decision epoch t, equally for the transition probabilities, the decision maker receives a reward, denoted by the real-valued function $r_t(s, a)$. When positive it may be regarded as income, and when negative as costs. This reward might be a lump sum received at some fixed or random point in time prior to the next epoch, accrued continuously throughout the current period or a random quantity that depends on the state.
When the reward depends on the next state j, it is denoted as $r_t(s, a, j)$. The expected value can then be evaluated by computing

$$r_t(s, a) = \sum_{j \in S} r_t(s, a, j)\, p_t(j|s, a). \tag{2.2}$$

Decision Rules and Policies In each state the decision maker has to choose an action. A *decision rule* prescribes a procedure for this action selection in each state at a specified decision epoch. It is denoted by d_s, for decision rule d at state s.
The underlying question is: which actions will lead to the optimal solution? This is the task of the optimization algorithm. This will result in a strategy which specifies the decision rule to be used at all decision epochs. This strategy is called a *policy* and is denoted by π, which is a sequence of decision rules. For a finite discrete MDP with $N < \infty$ states it can be written as $\pi = (d_1, d_2, \ldots, d_N)$.

2.3.2 Discrete Markov Decision Processes

This subsection concerns discrete Markov Decision Processes (MDPs). Since problems with a finite number of states are the most basic category of problems that dynamic programming can solve, we start this section with discrete MDPs with a finite set of states. We first focus on problems with a finite time horizon. Some definitions for these problems, additional to the definitions in the preceding subsections, are:

- a finite set of states S,
- a finite set of actions A_s for each state $s \in S$,
- a finite (discrete) time horizon T: $T = \{0, 1, 2, \ldots, N\}$, with $N < \infty$.

The goal is to find an optimal policy π. Applying a policy from a starting state s_0 produces a sequence of states $s_0, s_1, s_2, \ldots$ that is called a *trajectory*. Cumulative reward obtained over such a trajectory depends only on π and s_0. The function of the starting state s_0 that returns this reward is called the *value function* of π. It is denoted by V^π and is defined by

$$V^\pi(s_0) = \sum_{t=0}^{N-1} \mathbb{E}_{s_0,\pi} r(s_t, \pi(s_t)) + r(s_N). \tag{2.3}$$

The value function V^π is also called the expected total reward, for some policy π. The goal is to find an optimal policy. In general, this is a policy that maximizes the total reward or equivalently minimizes the total costs (cost to go). That is, we seek a policy π^* for which

$$V^{\pi^*}(s_0) = \max_\pi V^\pi(s_0), \forall s_0 \in S. \tag{2.4}$$

The value function might not be unique, since it is possible that more than one policy lead to the same total reward from a given starting state. Therefore we define the optimal value function as V^*.

Infinite-horizon The preceding examples where about problems with a finite time horizon, but not every problem can be modelled with a finite time horizon. The finite-horizon equations can be extended to solve problems with an infinite-horizon. The main problem that occurs is that the value function may not converges. It will only converges when a limit cycle with zero reward is reached.[3] This can be solved by adding a *discounting factor* for future states. The value function equation (Equation (2.3)) is then defined by

$$V^\pi(s_0) = \sum_{t=0}^{\infty} \mathbb{E}_{s_0,\pi} \gamma^t r(s_t, \pi(s_t)). \tag{2.5}$$

The discount factor $\gamma \in [0, 1)$ is a constant. The effect is to introduce a time horizon to the value function: the smaller γ, the more short-sighted V^π.

[3]Sometimes there is defined a termination state with zero reward, with only itself as possible next state. If the MDP reaches this state, it is terminated.

Policy Evaluation Markov decision problem theory and computation is based on using backward induction (dynamic programming) to recursively evaluate expected rewards. For the finite-horizon the value function (or expected total reward) can be evaluated at any state s_i $(0 \leq i < N)$ in the trajectory as $V^\pi(s_i) = \sum_{t=i}^{N-1} \mathbb{E}_{s_i,\pi} r(s_t, \pi(s_t)) + r(s_N)$ which represents the remaining reward when the system is in state s_i. This is the Markov property; the system has no "memory", it does only depend on the current state, independant of the previous states.
This property is recursively used to compute the total expected reward for a certain policy π, called *policy evaluation*. The policy evaluation algorithm computes the reward from the last state s_N back to the starting state s_0, given a certain policy.

Policy evaluation can also be used in infinite-horizon problems. The problem is that iteration over the states in time is not as straight forward as with a finite-horizon, because there could be an iteration over infinity. The discount factor gives us a solution for this problem. The set of equations to be solved is, for all states s,

$$V^\pi(s) = r(s, \pi(s)) + \gamma \sum_{j \in S} p(j|s, \pi(s)) V^\pi(j). \tag{2.6}$$

The discount factor γ causes converging of the value function, which will be used as a termination action for the iteration. The algorithm for policy evaluation for a infinite time-horizon is given below.

Algorithm 2.1 *Policy Evaluation*

1. Set $n = 0$ and set $V_0(s) = 0$ for each $s \in S$.
2. Increment n by 1.
3. For each $s \in S$, compute $V_n(s)$ by

$$V_n(s) = r(s, \pi(s)) + \gamma \sum_{j \in S} p(j|s, \pi(s)) V_{n-1}(j). \tag{2.7}$$

4. Stop if V has converged, otherwise go to step 2.

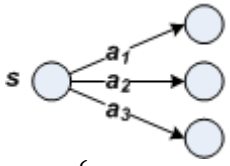
Optimal policy algorithms

Now the discrete Markov decision process is defined, the next and most important step is to find the optimal policy π. The remaining of this subsection describes algorithms used for finding optimal policies. From now on we focus on infinite-horizon problems; discounted discrete Markov decision processes.[4] The algorithms are:

- Value iteration
- Policy iteration
- Modified policy iteration

[4] For finite-horizon problems, the discount factor γ can be set to one in most situations for all following formulas and algortihms. Iterations will be terminated when the end of the time horizon is reached or a termination criterium is met.

Value iteration *Value iteration*[5] is the most widely used an best understood algorithm for solving discounted MDPs, that is finding the optimal policy. The set of possible policies can be huge. For instance, if the action does not depend on the current state, then there are $|A|^{|S|}$ policies. Instead of seeking the optimal policy directly, value iteration uses the basic idea of dynamic programming, which consists in evaluating the optimal value function V^* first, to avoid this problem. Once V^* has been computed, it is possible to obtain an optimal policy by applying Bellman's equation (Figure 2.2).



$$V^*(s) = \max_{a\in\{a_1,a_2,a_3\}} \left\{ r(s,a) + \sum_{j\in S} p(j|s,a)V^*(j) \right\}$$

Figure 2.2. Bellman's equation example. Possible actions in state s are a_1, a_2 and a_3. If the optimal value is known for the corresponding successor states, then this Bellman equation gives the optimal value of state s. Example from [4] (p. 38).

When using discounted rewards (adding discounting factor γ), the standard form of Bellman's equation becomes

$$V^*(s) = \max_{a\in A_s} \left\{ r(s,a) + \gamma \sum_{j\in S} p(j|s,a)V^*(j) \right\}. \tag{2.8}$$

Now we can iteratively apply Equation (2.8) to seek the optimal policy, that is the trajectory with the action for each state to be taken resulting in maximum reward. The algorithm that describes this iterative search is called value iteration (Algorithm 2.2). It is very similar to policy evaluation (Algorithm 2.1). The difference is that a fixed policy is taken in the policy evaluation algorithm.

Algorithm 2.2 *Value Iteration*

1. Set $n = 0$ and set $V_0(s) = 0$ for each $s \in S$.
2. Increment n by 1.
3. For each $s \in S$, compute $V_n(s)$ by

$$V_n(s) = \max_{a\in A_s} \left\{ r(s,a) + \gamma \sum_{j\in S} p(j|s,a)V_{n-1}(j) \right\}. \tag{2.9}$$

4. Stop if V has converged, otherwise go to step 2.

Convergence of this algorithm can be proved thanks to the discount factor $\gamma < 1$. It also converges when $\gamma = 1$ and all values are well defined. Value iteration can be proved to have a computational cost polynomial in $|A|$ and $|S|$. This might still be very costly for huge state or action spaces, but usually takes much less time than exploring the whole set of policies.[6]

[5] Value iteration is also known under other names. For example, successive approximation, over-relaxation, backward induction, or pre-Jacobi iteration.

[6] See [9], section 6.3.3 for adjustments on value iteration for increasing the speed of converging.

Policy iteration Another method for solving infinite-horizon Markov decision problems is *policy iteration*, or approximation in policy space. Policy iteration appears to relate directly to the structure of Markov decision problems, while value iteration may be regarded as an application of general approach for finding fixed points. While it is useful for infite-horizon problems, it is not efficient for finite-horizon problems.
Policy iteration consists in using the policy evaluation algorithm (Algorithm 2.1) defined previously to obtain succesive improved policies. The policy iteration algorithm can be described as follows.

Algorithm 2.3 *Policy Iteration*

1. Set $n = 0$ and select an arbitrary policy π_0.
2. *Policy evaluation*
   Obtain V_n for policy π_n through policy evaluation (Algorithm 2.1).
3. *Policy improvement*
   Choose a policy π_{n+1} on V_n to satisfy
   $$\pi_{n+1} = \arg\max_{a \in A_s} \left\{ r(s,a) + \gamma \sum_{j \in S} p(j|s,a) V_n(j) \right\}. \tag{2.10}$$
4. Stop if π has converged ($\pi_{n+1} = \pi_n$), otherwise increment n by 1 and go to step 2.

Modified policy iteration Finaly we will mention an algorithm for seeking an optimal policy, called *modified policy iteration*. In fact, this algorithm is a combination between the two previous described algorithms: value iteration and policy iteration.
Policy iteration is the fundament of this algorithm, but the policy evaluation step (step 2) can slow down the algorithm. When the number of states is large, solving the linear system in the policy evaluation step by direct methods is time-consuming. The modified policy iteration algorithm works around this difficulty by solving the linear system iteratively by using value iteration.

Algorithm 2.4 *Modified Policy Iteration*

1. Set $n = 0$ and set $V_0(s) = 0$ for each $s \in S$.
2. *Policy improvement*
   Choose a policy π_{n+1} on V_n as in policy iteration.
3. *Partial policy evaluation*
   A stationairy policy u_n is defined and the value function V_{n+1} is evaluated some m_n times.
4. Stop if π has converged. Otherwise return to step 2.

The sequence of non-negative integers m_n is predefined. There are two special cases of the modified policy iteration method. If $m_n = 1$ for all n, we obtain the value iteration method, while if $m_n = \infty$ we obtain the policy iteration method. For more detailed information about modified policy iteration, see [2] (p. 32) or [9] (sec. 6.5.1).

2.3.3 Continuous Markov Decision Processes

Until now we discussed Markov decision processes, where the decision maker could choose actions on a (predetermined) discrete set of time points. This is in real world problems not always the case, for instance in equipment maintenance or queueing control. In this subsection we discuss briefly Markov decision processes where the time is modelled continuously, that is, allowing action choice at random times in $[0, \infty]$.
The most general form of continuous time models are so called *semi-Markov decision processes* (SMDPs), which is a generalization MDPs. They allow (or require) choosing actions whenever the system state changes, the system evolution is modelled in continuous time and the time spent in a particular state follows an arbitrary probability distribution. A *continuous-time Markov decision process* is a special case of a SMDP, where the intertransition times are exponentially distributed.

Definitions

If we want to extend the previously defined formalism for the discrete case to the continuous case, we first have to adapt the definitions of the problem. We start immediately with a infinite set of states, so we discuss discounted semi-Markov decision processes.

States The set of possible states of the system is $S = \mathbb{R}^p$. Therefore the state of the system is defined by a vector $\vec{s}$ of p real valued variables.

Actions The set of possible actions is $A = \mathbb{R}^q$, which may depend on the current state of the system ($\vec{s}$). The decision maker can choose an action vector $\vec{a}$ of q real values.

Transition function The transition function $f : S \times A \mapsto \mathbb{R}^p$ or transition probability maps actions to derivatives of the state with respect to time. This is defined by $\dot{\vec{s}} = f(\vec{s}, \vec{a})$. This can be compared with the transition probability $p(j|s, a)$ for the discrete case. The difference is that the derivative is used in order to deal with time continuity.

Reward The reward is a function $r : S \times A \mapsto \mathbb{R}$. The goal is to maximize the reward, exactly as in the discrete case.

Policy The policy (or strategy) is a function $\pi : S \mapsto A$, which maps states to actions.

Shortness factor Because we are dealing with an infitite time horizon, we also need a discount factor; the shortness factor $s_\gamma \geq 0$. It plays a similar role as the discount factor γ for the discrete case.[7] If $s_\gamma = 0$, the problem is non-discounted. If $s_\gamma > 0$, the problem is discounted and the time horizon is given by $1/s_\gamma$.

[7] These factors are related by $\gamma = e^{-s_\gamma \delta t}$, where δt is a time step.

Value function and optimal policy

As for the discrete case, the goal for continuous MDPs is to find a policy π that maximizes the reward. Applying a policy from a starting state $\vec{s}_0$ at time t_0 produces a trajectory $\vec{s}(t)$ defined by the ordinairy differential equation

$$\dot{\vec{s}} = f(\vec{s}, \pi(\vec{s})), \forall t \geq t_0, \tag{2.11}$$
$$\vec{s}(t_0) = \vec{s}_0.$$

We have to adapt the value function for the discrete case (Equation (2.5)), to deal with continuous time. We define it by

$$V^{\pi}(\vec{s}_0) = \int_{t=t_0}^{\infty} e^{-s_\gamma(t-t_0)} r(\vec{s}(t), \pi(\vec{s}(t)))dt. \tag{2.12}$$

Again, the goal is to find an optimal policy, whatever the starting state $\vec{s}_0$. That is, we seek a policy π^* for which

$$V^{\pi^*}(\vec{s}) = \max_{\pi} V^{\pi}(\vec{s}_0), \forall \vec{s}_0 \in S. \tag{2.13}$$

Like in the discrete case, V^{π^*} does not depend on π^*. Therefore we denote the value function under the optimal policy as V^*, the optimal value function.

Now that we have defined the value function for continuous time, we can evaluate policies in order to seek for optimal policies. In the policy evaluation algorithm for discrete problems (Algorithm 2.1), the value function is recursively computed through iteration. For continuous problems we can evaluate policies by solving the value equation (2.12) at once. This is easy for very simple problems, when we are dealing with very simple policies. In general this is actually a very difficult task.
The algorithms for finding optimal policies discussed for discrete problems, value iteration, policy iteration and modified policy iteration, can also be used for continuous problems. The principles are the same, but the algorithms need some small adjustments. See for instance chapter 11 of [9] for a detailed description. As with policy evaluation, computations within the algorithms can be very difficult.

Problem Discretization

As mentioned previously, solutions for algorithms within continuous problems cannot be found in general. In order to find a method that works with all these problems, it is possible to apply some form of discretization to the continuous problem so that the algorithms for the discrete case can be applied. Discretization is described briefly below.

First a finite number of sample states and actions have to be chosen to make up the state and action sets: $S_d \subset S$ and $A_d \subset A$, representative for the infinite set. Then it is necessary to define state transitions, which is the actually descretization of the continuous state space. Note that it is not always possible to transform a continuous problem to make it fit into discrete deterministic formalism. The discretization causes a loss of information for most problems. Despite this, it is still possible to apply dynamic programming algorithms to a state discretization. The key issue is to find an equivalent to the discrete Bellman equations.

Let us consider a time step of length δt. It is possible to split the value function (2.12) into two parts:

$$V^{\pi}(\vec{s}_0) = \int_{t=t_0}^{t_0+\delta t} e^{-s_\gamma (t-t_0)} r(\vec{s}(t), \pi(\vec{s}(t)))dt + e^{-s_\gamma \delta t} V^{\pi}(\vec{s}(t_0+\delta t)). \tag{2.14}$$

For small δt, this can be approximated by

$$V^{\pi}(\vec{s}_0) \approx r(\vec{s}_0, \pi(\vec{s}_0))\delta t + e^{-s_\gamma \delta t} V^{\pi}(\vec{s}_0 + \delta\vec{s}), \tag{2.15}$$

With

$$\delta\vec{s} = f((\vec{s}_0, \pi(\vec{s}_0))\delta t.$$

Thanks to the discretization of time, it is possible to obtain a semi-continuous Bellman equation that is very similar to the discrete one (2.8):

$$V^*(\vec{s}) \approx \max_{\vec{a}\in A_d} \left(r(\vec{s},\vec{a})\delta t + e^{-s_\gamma \delta t} V^*(\vec{s}+\delta\vec{s}) \right). \tag{2.16}$$

In order to solve this equation, it should be an assignment instead of a approximation. This would allow to iteratively update $V(\vec{s})$ for states $\vec{s}$ in S_d, similarly to the discrete value iteration algorithm. One major obstacle is that $\vec{s}+\delta\vec{s}$ is not likely to be in the discretized set of states S_d. This can be solved by using some form of interpolation to estimate $V^*(\vec{s}+\delta\vec{s})$ from the values of discrete states close to $\vec{s}$ in S_d.

Now we can apply value iteration for discretized continuous (or semi-continuous) problems, according to the following algorithm.

Algorithm 2.5 *Semi-Continuous Value Iteration*

1. Set $n = 0$ and set $V_0(\vec{s}) = 0$ for each $\vec{s} \in S_d$.
2. Increment n by 1.
3. For each $\vec{s} \in S_d$, compute $V_n(s)$ by

$$V_n(\vec{s}) = \max_{\vec{a}\in A_d} \left(r(\vec{s},\vec{a})\delta t + e^{-s_\gamma \delta t} \underbrace{V_{n-1}(\vec{s}+f(\vec{s},\vec{a})\delta t)}_{\text{estimated by interpolation}} \right) \tag{2.17}$$

4. Stop if V has converged, otherwise go to step 2.

Algorithm 2.5 provides a general framework for continuous value iteration. But essential for applying this algorithm is the way of discretization. How to choose S_d, U_d, δt? How to interpolate between sample states? Many methods have been designed to make these choices so that the algorithm works efficiently. The *finite difference method* is one of the simplest. For a description of this method, see [4], page 48 and further.

2.3.4 Curse of Dimensionality

The basics of Dynamic programming and Markov decision processes are quite simple. But they suffer from a major difficulty, the so-called *curse of dimensionality*. The cost of discretizing the state space is exponential with the state dimension, which makes value iteration computationally intractable. For instance, a 4-dimensional state space with 100 samples per discretized state variable would have 100 million states. This reaches the limit of modern computers. The use of more clever discretization than a uniform grid, can reduce the number of samples. One can think of choosing large samples in areas where less accuracy is needed. Still it will keep its limitations.

2.4 Artificial Neural Networks

The (gridbased) discretization of the value function described, is a form of function approximation. Still, discretized continuous MDPs suffer from the curse of dimensionality. A technique called artificial neural networks, which is another function approximator, can help solving this problem thanks to their ability of generalization. This section describes the basics of artificial neural networks.

2.4.1 Introduction

The objective is to construct an approximate representation of the value function.[8] The first step is choosing an appropriate *approximation archicture* for the problem te be solved. This is a certain function involving a number of free parameters. The next step is to tune these parameters, in order to provide the best fit of the function to be approximated. This tuning is called *learning* or *training*. Another important aspect is the choice of a representation or encoding of the input $\vec{s}$, the state of the system.
So in order to develop an effective approximation, we need a suitable architecture and effective algorithms for tuning the parameters. Section 2.4.2 describes some well known approximation architectures and its basics. Section 2.4.3 provides some techniques for parameter tuning.

2.4.2 Approximation Architectures

The approximation problem can be framed as follows. We are interested in approximating the function $V : S \mapsto \mathbb{R}$, with V the value function and S the state space of the dynamic programming problem. We let $\vec{w}$ be a vector of parameters, often called *weights* and $V_{\vec{w}}$ the function describing the architecture under the weight vector $\vec{w}$. In general, $V_{\vec{w}}$ is easy to compute for fixed weights $\vec{w}$. We are interested in minimizing the difference between $V_{\vec{w}}(\vec{s})$ and $V(\vec{s})$, for all states $\vec{s}$. This is often called the error and says something about the efficiency of the approximation.
Approximation architectures can be broadly classified in two main categories: linear and non-linear architectures. The remainder of this subsection will describe these two categories.

[8] Of course other functions of interest can be approximated, but in this paper we are interested in the value function.

Linear architecture

Linear architectures are the most basic approximation architectures and in general they are easy to train. The general form of linear function approximators is

$$V_{\vec{w}}(\vec{s}) = \sum_i w_i \phi_i(\vec{s}), \tag{2.18}$$

where w_i is the i-th element of the weight vector and ϕ_i the corresponding easily computable function of the state vector $\vec{s}$. The function ϕ_i is called the *basis function*. A typical example of a linear approximator is the use of linear regression to interpolate or extrapolate some experimental data.
The **grid-based approximation** of the value function, described in Section 2.3.3, is a particular case of a linear function approximator. Other techniques for example are **tile coding** or **normalized gaussian networks**.[9]

Nonlinear architecture

A common nonlinear architecture is the *feedforward neural network* (FFNN) with a *single hidden layer*. They consist of a graph of nodes, called neurons, connected by weighted links. These nodes form directed acyclic graphs. The neurons receive input values and produce output values. The mapping from input to output depends on the link weights, so a feedforward neural network is a parametric approximator.
Under the FFNN architecture, the state $\vec{s}$ is encoded as a vector $\vec{y}$ with components $y_j(\vec{s})$, which is then transformed linearly through a "linear layer", involving the weights w_{ij}. This produces I scalars:

$$\sum_j w_{ij} y_j(\vec{s}), \text{ for } i = 1, \ldots, I. \tag{2.19}$$

Each of these scalars becomes the input to a differentiable function $\sigma(\cdot)$, called a *sigmoidal function*. The output scalars of the sigmoidal functions are linearly combined using the weights w_i. This results in the final output of the neural network:

$$V_{\vec{w}}(\vec{s}) = \sum_i w_i \sigma \left(\sum_{j<i} w_{ij} y_j(\vec{s}) \right). \tag{2.20}$$

This feedforward neural network is a network with one output and two layers; one linear layer (Equation (2.19)) and one hidden layer (the sigmoidal function). In general, a FFNN can have multiple hidden and linear layers, and even multiple output.

2.4.3 Network Training

When an architecture approximator is chosen, the task is to find weights in order to approximate the target function as good as possible. This optimization problem can be solved by minimizing the so-called *error function*, which measures how bad an approximation is. Typically, the

[9] For more information, see for instance [4], section 2.3.1.

quadratic error (or least squares) is used as error function. In this case we want to approximate the value function $V(\vec{s})$ by the approximator $V_{\vec{w}}(\vec{s})$, which leads to the quadratic error

$$E(\vec{w}) = \frac{1}{2} \sum_{s \in S} \left(V(\vec{s}) - V_{\vec{w}}(\vec{s}) \right)^2 . \tag{2.21}$$

We want to find weights $\vec{w}$ that minimize the error function E. This process is called *training* or *learning* in the field of artificial intelligence, or *curve-fitting* or *regression* in the field of data analysis.

Gradient Descent

Finding a value of $\vec{w}$ that minimizes the scaler error function $E(\vec{w})$ is a classical optimization problem. Many techniques have been developed to solve it, with *gradient descent* the most common one. Gradient descent consists in considering that E is the altitude of a landscape on the weight space: to find a minimum, starting from a random point, walk downwards until a minimum is reached. Gradient descent will not always converge to an absolute minimum of E, but possible to a local minimum. In most usual cases this local minmimum is good enough, provided that a reasonable initial value of $\vec{w}$ is chosen.

For more detailed information about learning and gradient descent, see for instance [2] section 3.2 or [4] section 2.2.

2.5 Neuro-Dynamic Programming

Section 2.3 described some well know algorithms for Markov Decision Processes (MDP's) used in the area of Dynamic Programming. We saw that the discrete character of the algortihms - for discrete MDPs, as well as for continuous MDPs (which are discretized) - gives computational problems while applied to an optimization problem. This is caused by the curse of dimensionality. A way to handle with this problem is the use of Artificial Neural Networks, described in the previous section. *Neuro-Dynamic Programming* (NPD) combines these two techniques (Markov Decision Processes and Artificial Neural Networks), to solve optimization problems. Bertsekas and Tsitsiklis [2] where the founders of this technique.
This section gives an overview of neuro-dynamic programming. It covers the basic ideas behind this technique and describes some algorithms used in this area briefly.

2.5.1 Basic Idea

The basic idea for *dynamic programming* is the Bellman equation theory. When using a Bellman equation, an optimal decision at the current state is that decision that maximizes the expected reward of

$$\text{current state reward} + \text{future states reward}, \tag{2.22}$$

where the future states are computed starting from the next state using an optimal policy. This is an extensive mathematical methodology, in particular for problems with large state spaces. The

key idea of neuro-dynamic programming is to use a one-step lookahead with an "approximate reward", instead of computing the future reward explicitly. The optimal decision can then be formulated as selecting the decision at the current state, that maximizes the expected reward of

$$\text{current state reward} + \text{approximate future states cost}. \tag{2.23}$$

Note the difference between Equation (2.22) and Equation (2.23). The approximation of the future reward can be done using techniques from the field of neural networks.

2.5.2 Generic Form

The approximation of the future reward in Equation (2.23) can be formulated as an approximation of the (optimal) value function $V^*(s)$. We define this approximation as $\tilde{V}^*_{\vec{w}}(s) \approx V^*(s)$, for some state s and parameter/weight vector $\vec{w}$. The standard form of the Bellman equation for discounted discrete Markov decision processes (Equation (2.8)), can then be rewritten for use within NDP as

$$V^*(s) = \max_{a \in A_s} \left\{ r(s,a) + \gamma \sum_{j \in S} p(j|s,a) \tilde{V}^*_{\vec{w}}(j) \right\}. \tag{2.24}$$

The architecture and the weight vector $\vec{w}$ for approximation $\tilde{V}_{\vec{w}}$ are defined/computed through (artificial) neural network techniques, described in Section 2.4.

Policy Approximation

From the value function we can define a policy $\pi(s)$ as

$$\pi(s) = \arg\max_{\pi} \left\{ r(s,\pi(s)) + \gamma \sum_{j \in S} p(j|s,\pi(s)) \tilde{V}^{\pi}_{\vec{w}}(j) \right\}. \tag{2.25}$$

It can be very hard to evaluate a certain policy π by direct computation. Another approach that can be used is policy approximation. Given the value function $\tilde{V}$ and Equation (2.25), we can compute the policy $\pi(s)$ for states s in a representive subset $\hat{S}$. We "generalize" the decisions $\pi(s), s \in \hat{S}$, to obtain a policy $\tilde{\pi}_{\vec{v}}(s)$, which is defined for all states, by introducing a parameter vector $\vec{v}$ and solving the least squares problem

$$\min_{\vec{v}} \sum_{s \in \hat{S}} \left\| \tilde{\pi}_{\vec{v}}(s) - \pi(s) \right\|^2. \tag{2.26}$$

Once $\vec{v}$ has been fixed, the decision $\tilde{\pi}_{\vec{v}}(s)$ is easily obtained. The vector $\vec{v}$ is obtained via neural network techniques.

The approximation architecture (see Section 2.4.2) that provides us with $\tilde{\pi}_{\vec{v}}(s)$ is often called an *action network*, to distinguish it from the architecture that provides us with $\tilde{V}_{\vec{w}}(s)$, which is often called a *critic network*.

2.5.3 Some Algorithms

Now the value function and policy evaluation (policy approximation) are defined, we can concentrate on algorithms that will result in an optimal policy π^*. We will use Equation (2.24) and asume that we have chosen a value for the weight vector $\vec{w}$ and that we have access to a *subroutine* which on input s outputs $\tilde{V}^*_{\vec{w}}(s)$.

Approximate Policy Iteration

To evaluate a certain policy π, we can use *approximate policy iteration* within neuro-dynamic programming as en equivalent of policy iteration within dynamic programming. Actually, the **general** structure of approximate policy iteration is the same as for exact policy iteration: alternate between a (approximate) policy evaluation and a policy improvement, until the final condition is met (convergence of policy). As described in Algorithm 2.3. There are two differences, however.

1. Given the current policy π, the corresponding value function V^π is not computed exactly. Instead, we compute an approximate value function $\tilde{V}^\pi_{\vec{w}}(s)$.
2. Generating the new (improved) policy can sometimes be done exactly, but there are cases in which we only obtain an approximation of a greedy policy.

These two differences can lead to some errors. In (1) we have two error factors; the approximation architecture representing V^π may not be adequately enough and tuning of the weight vector $\vec{w}$ based on simulation will introduce noise. Next to that, in (2) the approximation of the policy can be a source of error.
An intial policy is required, in order to run the approximate policy iteration algorithm.

There are a lot of variants on this general version of approximate policy iteration, which will not be discussed in this paper. Variants are approximate policy iteration based on *Monte Carlo simulation*, and *least squares approximation* or using so-called *temporal differences* (TD(λ)). Another variant is *optimistic policy iteration*, where the policy improvement step is performed before the approximate evaluation of the value function converges. For details about the specific variants of approximate policy iteration, see sections 6.2-6.4 of [2].

Approximate Value Iteration

As with policy iteration, there are also approximate algorithms for finding an optimal policy related to value iteration, Algorithm 2.2. One basic approximate value iteration algorithm within neuro-dynamic programming is *sequential backward approximation*, which is described below. We focus here on problems with an infinite horizon.

Sequential Backward Approximation The algorithm is initialized with a parameter vector $\vec{w}_0$ and a corresponding value function $\tilde{V}_{\vec{w}_0}(s)$. At a typical iteration, we have a parameter vector $\vec{w}_k$, we select a set S_k of representative states, and we compute estimates of the expected reward (value function) from the states in S_k by letting

$$\hat{V}_{k+1}(s) = \max_{a \in A_s} \left\{ r(s,a) + \gamma \sum_{j \in S} p(j|s,a) \tilde{V}_{\vec{w}_k}(j) \right\}, \; s \in S_k. \tag{2.27}$$

We then determine a new set of parameters $\vec{w}_{k+1}$ by minimizing with respect to $\vec{w}$ the quadratic reward criterion

$$\sum_{s \in S_k} v(i) \left(\hat{V}_{k+1}(s) - \tilde{V}_{\vec{w}}(j) \right)^2, \tag{2.28}$$

where $v(i)$ are some predefined positive weigths. In words; find that parameter vector $\vec{w}_{k+1}$, which minimizes the error of the approximated value function $\tilde{V}_{\vec{w}}(j)$ with the value function of Equation (2.27). This can be solved using standard algorithms in the field of neural networks[10]. We iterate this proces over $k = 1, 2, 3, \ldots$ until the value function $\tilde{V}$ converges. It is noteworthy that this iteration not always converges. There are some special cases in which approximate value iteration suffers from potential divergence. For an example, see [3] paragraph 6.5.3.

Incremental Value Iteration The value iteration algorithm described above proceeds in phase: we obtain estimates $\tilde{V}(s)$ at a number of states s and then compute a new parameter vector $\vec{w}$ by solving the least squares problem, Equation (2.28). This could also be done using a gradient descent algorithm, which is exactly the cases with *incremental value iteration*.

Other Approximate Value Iteration Algorithms As with approximate policy iteration, there are next to the two above briefly described algorithms a lot of other variants of optimal policy algorithms related to value iteration. For instance *value iteration with state aggregation*, which is a fully incremental approximate value iteration method together with a function approximator derived from state aggregation, or *value iteration with representative states*. For more variants of approximate value iteration and full descriptions of the briefly described algorithms in this section, see sections 6.5-6.9 of [2].

Other Algorithms

Besides approximate algorithms related to policy iteration and value iteration, Bertsekas and Tsitsiklis have described in sections 6.10-6.12 of [2] other methods for finding an optimal policy. For instance *Bellman error methods*, where approximation of the optimal value function $\tilde{V}_{\vec{w}}(s)$ is based on minimizing the error in Bellman's equation, or *approximate linear programming*, which concerns solving the linear programming problem with the use of approximation.

2.6 Evolutionairy Algorithms

The optimization methods in the field of dynamic programming, described in the Sections 2.3-2.5, are widely used for solving optimization problems. These methods and algorithms are very useful for optimization problems where decisions are involved, what should be obvious by now. They are also useful for optimization problems within animal behavior, as mentioned at the start of this chapter (Section 2.1.1). This brings us to a technique called *evolutionairy computing*, which may also be a suitable optimization method for solving the ideal racing line problem.

[10] If the approximation architecture of $\tilde{V}$ is linear, we are dealing with a linear least squares problem that can be solved efficiently.

This section describes in general what an evolutionairy algorithm is (Section 2.6.1) and how it works (Section 2.6.2). Furthermore, in Section 2.6.3 we give some first thoughts about the possible application of an evolutionairy algorithm to the racing line problem.

2.6.1 Introduction

Animals make decisions, for instance when to go out hunting or just wait until there are less preditors in the neighbourhood. But also which 'partner' to choose or where to find a shelter. These decisions have great impact on the survival skill of the animal. Animals who make decisions which increase their ability to survive, are more likely to have offspring. The offspring in their case, carry the genetics/chromosomes of their parents, in which their instinct in decision making is captured. Which implies that on average the offspring has a greater opportunity to survive and potentially more offspring. This is in essence the theory of natural evolution: **survival of the fittest**.
There is a computing theory within computer science called evolutionairy computing, based on the natural evolution theory. The fundamental metaphor of evolutionairy computing relates this powerful natural evolution to a particular style of problem solving - that of 'trial-and-error'.

Evolution		**Problem solving**
Environment	$\longleftrightarrow$	Problem
Individual	$\longleftrightarrow$	Candidate solution
Fitness	$\longleftrightarrow$	Quality

Table 2.1. Evolutionairy computing methaphor linking natural evolution to problem solving.

To make a link between problem solving and evolutionary theory, let us consider the following. A given *environment* is filled with a population of *individuals* that strive for survival and reproduction. The *fitness* of these individuals (determined by the environment) relates to how well they succeed in achieving their goals (survival and multiplying). In the context of stochastic trial-and-error style problem solving process, we have a collection of *candidate solutions*. Their ability to solve the *problem*, their *quality*, determines the chance that they will be kept and used as seeds for constructing further candidate solutions (see also Table 2.1).

2.6.2 Basics of Evolutionary Algorithms

The previous section described the idea behind evolutionairy algorithms. There are many different variants of evolutionairy algorithms, but the common underlying idead behind all these techniques is the same:

> Given a quality function to be maximized, we can randomly create a set of candidate solutions, i.e., elements of the function's domain, and apply the quality function as an abstract fitness measure; the higher the better. Based on this fitness, some of the better candidates are chosen to seed the next generation by applying *recombination* and/or *mutation* to them.
> Recombination is an operator applied to two or more selected candidates (the so-called parents) and results in one or more new candidates (the children). Mutation is applied to one candidate and results in one new candidation.
> Executing recombination and mutation leads to a set of new candidates (the offspring)

that compete with the old ones for a place in the next generation. *Candidate selection* (or *survivor selection*) for the new generation is based on their fitness (their quality, how well they solve the problem) and possibly their age.

The proces described above can be iterated until a candidate with sufficient quality (a solution good enough) is found or a previously set computational limit is reached. In this process there are two fundamental forces that form the basis of evolutionairy systems:

- Variation operators (recombination and mutation) create the necessary diversity and thereby facilitate novelty.
- Selection acts as a force pushing quality.

The combined application of variation and selection generally leads to improving fitness values in consecutive populations.

Many components of an evolotionairy algorithm/process are stochastic. During selection fitter individuals (better solutions) have a higher probability to be selected than less fit ones (less better solutions), but typically even the weak individuals have a chance to become a parent or survive. For recombination of individuals the choice of which pieces will be recombined is random/stochastic. Similarly for mutation; the pieces that will be mutated within a candidate solution, and the new pieces replacing them, are chosen randomly. These stochastic elements are crucial. They contribute to 'jump' out of local optima and approach a global optimum.

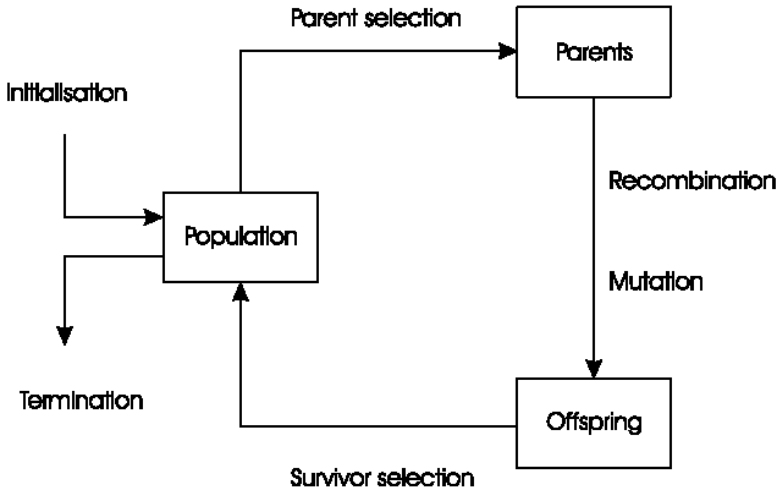


Figure 2.3. General scheme of an evolutionairy algorithm

Figure 2.3 shows the general scheme of an evolutionairy algorithm, as described above. The various dialects of this scheme, the various algorithms, are briefly described further. First the most important components of an evolutionairy algorithm, which are mentioned before (some are in the figure), are described in some more detail. Second, some specific evolutionairy algorithms are mentioned.

Components of Evolutionairy Algorithms

Representation The first step in defining an evolutionairy algorithm (EA) is to link the actual problem context (so-called *phenotype*) to the problem-solving space where evolution takes place (*genotype*). This step is called *representation*. The genotype can be a vector of real or binary numbers for example. For a lot of problems, this step is not easy.

Evaluation Function An important component is the *evaluation function*, or *fitness function*. This function represents the quality of a candidate solution; how well it solves the problem. It forms the basis for selection, and thereby facilitates improvements.

Population The population holds the representation of possible solutions, which is a set of genotypes. The population forms the unit of evolution. Individuals do not change or adapt; it is the population that does. The population size is constant and does not change during the evolutionairy search, in almost all EA applications.

Parent Selection The role of *parent selection* (or *mating selection*) is to distinguish among individuals based on their quality, in particular, to allow better individuals to become parents of the next generation. A parent is an individual, selected to undergo variation in order to create offspring. Parent selection within evolutionairy computing is typically probabilistic and uses the evaluation function to determine quality.

Variation Operators The role of *variation operators* is to create new individuals from old ones, which represent generating new solutions. They are divided into two groups, *recombination* en *mutation*.
Recombination (or *crossover*) is a so-called binary variation operator. It merges information from two[11] parent genotypes into one or two offspring genotypes. Recombination is a stochastic process. The principle behind it is simple: by mating two individuals (the parents) with different features we can produce an offspring that combines both of those features. For example; take each feature from the mother or the father randomly when dealing with a binary vector genotype, or take for each feature a random percentage of the mother and father when dealing with a real number vector genotype.
Mutation involves modifying the child or offspring. As recombination, the mutation operator is also stochastic. Modifcation (mutation) of a binary vector genotype can be flipping one bit randomly, for example. For a real number vector genotype this can be randomly scaling of a feature (number).

Survivor Selection *Survivor selection* (or *environmental selection*) is similar to parent selection; it distinguishes among individuals based on their quality. The difference is that it is used in a different stage of the evolutionary cycle (see Figure 2.3). The survivor selection mechanism is called after having created the offspring of the selection parents via variation operators. Survivor selection is used to keep a constant population size for each generation. After mating, the population has grown (the old generation/population + the created offspring). So a number of individuals has to be chosen for the next generation, equal to the size of the original population.

[11] There are also algorithm variants with recombination of more than two parents, but this has no reference with the biological idea.

This is usually based on the fitness value and done deterministic (where parent selection usually is stochastic). For instance, ranking the individuals on fitness and selecting the top segment.

Initialisation *Initialisation* is kept simple in most EA applications. The first population is seeded by randomly generated individuals, but also problem-specific heurstics can be used to generate an initial population with better fitness. It depends on the problem.

Termination Finally we mention the *termination condition* within the generic EA scheme. This can be reaching a (optimal) fitness level, if known. But also reaching a certain predefined computation time or converging of the maximum fitness value within the population.

Variants of Evolutionairy Algorithms

As mentioned earlier, there are various variants/dialects of evolutionairy algorithms. The difference between these variants are almost all differences within one or more of the above described components of the algorithm. The main difference is made at the representation component. Differences in this component, may require differences in other components. One can think of different variation operators, for example. But also the use of the population or selection mechanisms can differ.

The four major streams of evolutionairy algorithm dialects are **evolutionairy programming (EP)**, **genetic algorithms (GA)**, **evolution strategies (ES)** and **genetic programming (GP)**. The differences of these algorithms have mainly historical origin, but technically one algorithm can be preferable over others if it matches the problem better. The differences between these algorithms are not mentioned in this paper. See for instance [6] for detailed descriptions of these algorithms, as well as details about the components of evolutionairy algorithms.

2.6.3 How to Apply?

Section 2.6.2 described in general what an evolutionairy algorithm (EA) does and how it works. The idea of using evolutionairy algorithms for an optimization problem with decisions (the racing line problem) came from applications of dynamic programming in animal behavior, which forms the basic idea of evolutionairy computing. Now we have to ask ourselfs if an EA is applicable on the racing line problem or a problem involving decision making in general. The application of optimization algorithms related to dynamic programming is clear, but how to apply an EA? In this subsection we try to give some arguments and first implementation thoughts for the application of an evolutionairy algorithm on the racing line problem[12].

Representation We have to find a representation of the racing line problem context into a genotype. We can think of the racing line problem from the driver point of view, where he has to make decisions about steering for example, as described in Section 2.1. The decisions the driver will make, depend on the situation he is in. For example the current speed of the vehicle or the distance to or angle with the racetrack. One can think of a *tree structure* representation of the genotype, which is the case with genetic programming. This tree can be used as a decision tree,

[12]Fot the application of a dynamic programming related algorithm, see Chapter 3.

that outputs the decisions the driver has to make, given the situation. The situation properties will appear in the decision tree, to create dependency on the given situation.

> **Example** of (a part of) a decision tree: *if the distance in front of the car is greater than x, change the velocity by c.* Here the distance x is a situation property, and the decision (changing the velocity bij c) is the outcome of that particular tree.

One can think of multiple trees for every kind of action the driver can take (for instance changing speed and changing steering angle). This results in a set of decision trees for each individual, which represent the decisions the driver takes.

Evaluation Function Now we have defined a potential representation, we also have to think about a proper evaluation function. In fact, this is in theory very simple; the total time it takes for the car to drive one lap, according to the decision trees (the genotype) of the individual. In practice this gives a problem, because the decision making is actually continuous. This results in iteratively applying the decision trees on the problem infinity times. The problem can be solved by applying the decision trees every s seconds. In other words, take every s seconds new decisions. In this form, the evaluation function can be defined.

Other Components The representation and evaluation function are the major difficulties. The variation operators and selection methods are easier. We can use basic implementations within genetic programming. For selection methods, the evaluation function can be used. The variation operators can change the structure of the decision trees or parameters within the desicion trees (the values of x and c in the example tree above, for instance).
The initial population can be a group of mutated individuals, from an individual with decision trees that will lead to a slow (initial) laptime. Converging of the maximum fitness within the population can be used as a termination condition.

The most important factors for applying an EA are mentioned. By now, the possible application of evolutionairy algorithms on the ideal racing line problem should be clear.

Chapter 3

Ideal Racing Line Problem

The ideal racing line problem is already introduced in Chapter 2 and Section 2.1. This chapter describes the problem further into a problem formulation. We introduce the Robot Auto Racing Simulator in Section 3.2, which is used to define the vehicle dynamics used in the model (Section 3.3). Section 3.4 applies the problem formulation and the vehicle dynamics to a discrete Markov decisicion process, which results in the final model formulation for the ideal racing line problem.

3.1 Problem Formulation

The objective is to find the ideal racing line for a set of different racetracks. The ideal racing line is essentially different from the shortest path, as Definition 1.1 says. There are many different ways to take a corner, but which one is the fastest? Figure 3.1 gives an idea about two different paths the vehicle can follow, when taking a corner.

The fastest path is the shortest path at maximum speed. In general this is not possible because the racecar can not take every corner at maximum speed. Furthermore, the racecar can not go instantly from zero to maximum speed, this takes some time. These characteristics are the behavior of the vehicle. So the problem is restricted by some vehicle dynamics. These dynamics are described in detail in Section 3.3.
Apart from restrictions on the racecar, there are also other restrictions. The vehicle has to stay on the racetrack, otherwise it could cut off a corner. Also the vehicle may not turn around on the racetrack, otherwise it could make a sharp turn at the start and cross the finish line immediately. This would result in a fast (lap)time, but has nothing to do with the ideal racing line. These restrictions are formulated in Section 3.4.

The problem is continuous in time, state and action. Our goal is to solve the problem using a discrete Markov decision process. So we have to discretize the problem in order to solve it. This is also described in Section 3.4.

3.2 Robot Auto Racing Simulator

The Robot Auto Racing Simulator was originally designed and written by Mitchell E. Timin in 1995 [10]. The description he gave in his original announcement:

> The Robot Auto Racing Simulation (RARS) is a simulation of auto racing in which the cars are driven by robots. Its purpose is two-fold: to serve as a vehicle for Artificial Intelligence development and as a recreational competition among software authors. The host software, including source, is available at no charge.

In 1997 the first yearly official Formula One season was organised. People could send in their programmed cars and compete with others, trying to win races and score points. The programmed cars not only had to drive a certain track as fast as they could, but they also had to pass other cars te be able to win. The last RARS Formula One season was organised in 2003. Since 2004, a more modern car simulation program is available, called TORCS[1]. Competitions between programmers of car drivers take place at the TORCS Racing Board[2].

3.2.1 The Problem and RARS

RARS and TORCS are aiming for racing, competition. The goal is to program the most intelligent car; a car that can drive excellent laptimes on every track, with every obstacle (slow cars) and even make pitstops. This is far more than the objective of this paper. We are not interested

[1]The Open Racing Car Simulator, for more information see the website: http://torcs.sourceforge.net.
[2]For more information about the TORCS Racing Board, see the website: http://www.berniw.org/trb/.

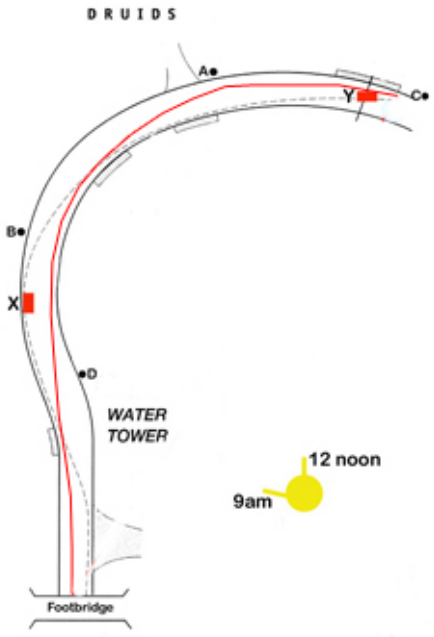


Figure 3.1. Two different ways to take the corner.

in passing other cars or taking a pitstop. We are interested in the fastest laptime on an empty track; the ideal reacing line. The only thing we are really interested in, are the vehicle dynamics used in these programs.
TORCS is a more advanced car simulation program what concerns graphics, tracks, as well as the vehicle dynamics, compared to RARS. We only need basic vehicle dynamics. Therefore we use the vehicle dynamics used in RARS as the basis of our problem and we do not look further at TORCS. These vehicle dynamics are described in Section 3.3.

3.2.2 Used Techniques in RARS

RARS has a long history of racing, and dozens of drivers have been programmed. They can be split into a group of cars that compute an optimal path first and cars that do not. The first group allows to use algorithms which are very costly (computation time). They can achieve excellent lap times on an empty track. Some optimization algorithms used in the past are for instance genetic algorithms or gradient descent. Also Coulom [4] implemented a car within this group. He used a technique from the field of neuro-dynamic programming, called temporal differences, to compute an optimal path.
Cars in the second group do not generate an optimal path. They generate control variables by simply observing their current state, without referring to a fixed trajectory. These cars are good in passing other cars, but are usually slower than those based on an optimal path when the track is empty.

Our objective, the ideal racing line, is typical a problem belonging to programmed cars in the first group mentioned. We are aiming at an optimal path. The restriction that we are dealing with, is the use of a discrete Markov decision process algorithm.

3.3 Vehicle Dynamics

The Robot Auto Racing Simulator uses a very simple two-dimensional model of car dynamics. Let $\vec{p}$ be the vector indicating the position of the car and $\vec{v}$ the velocity of the car. Then $\vec{x} = (\vec{p}, \vec{v})^t$ is the state of the system, at time t. Let $\vec{u}$ be the action (the decision) of the driver, from some set U. A simplified model of the simulation is described by the differential equations:

$$\begin{cases} \dot{\vec{p}} = \vec{v} \\ \dot{\vec{p}} = \vec{u} - k \left\| \vec{v} \right\| \vec{v} \end{cases} \tag{3.1}$$

The action $\vec{u}$ is restricted by the following constraints:

$$\{ \ \left\| \vec{u} \right\| \leq a_t \vec{u} \cdot \vec{v} \leq \frac{P}{m} \tag{3.2}$$

k, a_t, P and m are numerical constraints[3] that define some mehanical characteristics of the car:

- k = air-friction coefficient (aerodynamics of the car)
- P = maximum engine power

[3]Numerical values used in official races are $k = 2.843 \times 10^{-4}$ kg/m, $a_t = 10.30$ m/s^2 and $P/m = 123.9$ m^2/s^3.

- a_t = maximum acceleration (tires)
- m = mass of the car

In fact, the actual RARS physical model is a little more complex[4] and takes into consideration a friction model of tires on the track that makes the friction coefficient depend on slip speed. The mass of the car also varies depending on the quantity of fuel, and damage due to collisions can alter the car dynamics. These more complex characteristics are not interesting for our problem. We use the simplification proposed above.

3.4 Model Formulation Using a Discrete MDP

The major task in formulating the problem model, is discretizing the continuous variables. Otherwise we can not apply algorithms within the field of descrete Markov decision processes, as described in Section 2.3.2.

Remark By adding and adapting model variables to fit a discrete MDP, some RARS vehicle dynamics equations can not be applied correctly anymore. The basic ideas about the RARS vehicle dynamics, for example the maximum steering angle under a certain velocity or the maximum acceleration under a certain velocity and steering angle, are used. They should still hold, but the exact equations could need some adaptation. In the remaining of this chapter, we assume that the RARS vehicle dynamics hold, but we do not adapt the exact equations.

3.4.1 Discretization

The first step is discretizing the car position on the track, $\vec{p}$. This can be done by dividing the trackmap into segments, as a chess board for example. Each position on the trackmap is given by

$$\vec{p} = (m, n), \tag{3.3}$$

with $m = 1, 2, \ldots, M$ and $n = 1, 2, \ldots, N$ the position in the two dimensions of the track (and the surroundings). The whole set of positions $\vec{p}$ is denoted by P, which is the whole trackmap. The position of the car is at the center of a certain segment. The width of the segments is Δn in the n-direction, and Δm in the m-direction. We restrict each segment to be a square, so

$$\Delta n = \Delta m. \tag{3.4}$$

The width of the segments is manually set. Equation (3.4) does not implicate that $N = M$, because the whole track and surroundings can have a rectangular shape. Furthermore, we define that a car can only be in the exact middle of a segment. Figure 3.2 shows the discretization of the position on the track, with $p_1 = (m-1, n-1)$ and $p_2 = (m+2, n+1)$ as possible positions.

Remark Vector signs (for $\vec{p}$ and $\vec{v}$, for instance) are not shown in the figures in this section.

[4] For a detailed description, see http://rars.sourceforge.net/doc/co_vecto.htm.

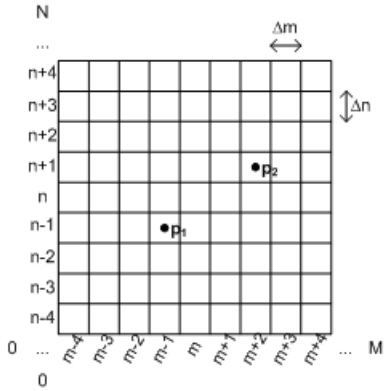


Figure 3.2. Example of discretization of the position space of a trackmap.

Now $\vec{p}$ is discretized, we have to discretize $\vec{v}$ in order to have a complete discretized state $\vec{x}$. The length of vector $\vec{v}$ is the (current) velocity, the direction of the vector can be seen as the direction the car is heading (in the "position space").
The velocity of the car (the length of $\vec{v}$) needs some discretization. This can easily be done by use a discrete set of velocities, for instance integers from zero to $v_{\max}$, with $v_{\max}$ the maximum speed, bounded by the RARS vehicle dynamics. We denote the set of possible velocities by V_v. The direction of $\vec{v}_t$ is determined by the line from $\vec{p}_{t-1}$ to $\vec{p}_t$ for the currect state at time t. Figure 3.3 shows an example of a velocity vector.

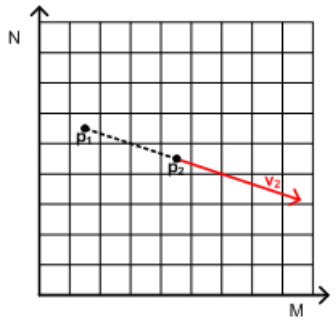


Figure 3.3. Example of a velocity vector $\vec{v}_2$ (red arrow). The direction is determined by the current position $\vec{p}_2$ and the previous position $\vec{p}_1$, the dashed line. The length of $\vec{v}_2 \in V_v$ is the velocity.

By now, the set of states $\vec{x}$ is discretized. The actionset U does also need some discretization. We let the possible actions $\vec{u}$ depend on the current state of the system $\vec{x}$, that is

$$\vec{u} \in U_{\vec{x}}. \tag{3.5}$$

The action (decision of the driver) is simply going from the current state to another state: going from state $\vec{x}_t$ to state $\vec{x}_{t+1}$. This can be seen as the new position $\vec{p}_{t+1} = (m_{t+1}, n_{t+1})$ on the track and choosing a new velocity vector $\vec{v}_{t+1}$ of the car. Without any further restrictions, the actionset $U_{\vec{x}}$ becomes very large[5]. This is undesirable. The RARS vehicle dynamics restricts the

[5]Typically this set is greater than $(M \times N \times |V_v|)$, because also possible new direction angles are involved.

actionset in some way, but not enough. Therefore we create an algorithm, that defines/computes the possible new states, thus the possible actionset. This algorithm is called the *action algorithm*, described in Section 3.4.2.

The time is handled in the action algorithm, defined in the next section. So we do not have to think about discretizing time in a set of intervals, where time is the laptime of the car and should not be confused with the time-horizon of the problem.

3.4.2 Action Algorithm

The *action algorithm* is an algorithm that tries to keep the size of the problem "small", that is to keep the possible actionset small and thereby also capture the discretizing issues. Furthermore, the algorithm tries to handle the *curse of dimensionality*. The algorithm returns a set of (discrete) actions $U_{\vec{x}}$ given the currect state $\vec{x}$. This can be seen as a function f as $f(\vec{x}) = U_{\vec{x}}$. Actually, the actionset $U_{\vec{x}}$ is just a set of possible new states $\vec{x}$. Before executing the action algorithm, the following should be defined:

- All the RARS vehicle dynamics parameters.
- The width of the trackmap segments; Δn and Δm.
- The actual track; it should be clear which segment of the trackmap is part of the track (road) and which segment is not (dirt).
- The discretized set V_v: the velocity should be discretized.
- Variables $\Delta lt_{\min}$ and $\Delta lt_{\max}$; these variables define the boundaries of the (lap)time (lt) between two following states, the so-called time view variables. Figure 3.4 shows an example of the time view variables. The major goal is to reduce the number of new possible positions, which reduces the possible new states. The variables are tools to deal with the curse of dimensionality.
- The segments (positions) of the finish line, which split a start and a finish side as in Figure 3.5; the positions $\vec{p}$ that are in the subset P_{start} or P_{finish} of the whole discretized position set P ($\{P_{\text{start}}, P_{\text{finish}}\} \subset P$).
- The current state $\vec{x_t}$, thus $\vec{p_t}$ and $\vec{v_t}$.

The action algorithm is sometimes hard to describe textually. To give some more understanding of the algorithm, some steps are graphically shown in Figure 3.6. First the action algorithm is described below. When segments (positions) are removed, the center of the segments is used as a reference point. This point is evaluated if it satisfies the restriction or not. The segment is removed based on this evaluation.

Algorithm 3.1 *Action Algorithm*

1. Define the possible new positions $\vec{p}_{t+1}$, while starting with the postition set P with all possible positions $\vec{p}(m,n)\ \forall\ m,n$ on the trackmap:
   (a) Remove the current position $\vec{p_t}$ from P; movement is compulsory.
   (b) Remove all positions that are not part of the track (dirt segments).

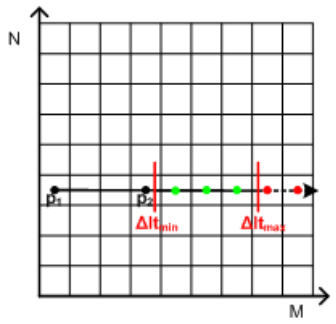


Figure 3.4. Example of the time view variables $\Delta lt_{\min}$ and $\Delta lt_{\max}$. We are in position p_t, with previous position p_{t-1} and assume that the new possible position is only bounded by the time view variables. Thus, we may not change the direction we are driving (no steering) and acceleration is equal to zero (the velocity remains constant). The possible new positions are those positions that can be reached within $\Delta lt_{\min}$ and $\Delta lt_{\max}$ seconds. This is graphically shown in the figure, where the green positions are the possible new positions. The red positions can not be reached within $\Delta lt_{\max}$ seconds with the current velocity and no acceleration. This reduces the possible new positions. Note that $\Delta lt_{\min}$ should be chosen small (for instance zero), otherwise positions near the current position can not be reached. This may result in a less optimal solution. In real problems the time view variables are acting more like boundaries with a (part of a) circle-shape, because of the possible steering angle. Furthermore, the acceleration influences the boundaries caused by the time view variables. This total boundary is Step 1e of the action algorithm (Algorithm 3.1), see also Figure 3.6.

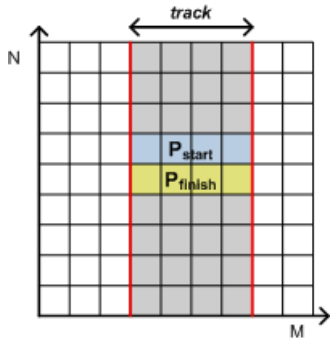


Figure 3.5. Example of a finish line with start (P_{start}, blue) and finish (P_{finish}, yellow) positions/segments. The grey segments belong to the track, the red lines are the boundaries of the track.

(c) Compute the maximum steering angle $\alpha_{\max}$ given current state (velocity), according to RARS vehicle dynamics.

(d) Remove all positions that the car can not reach, given $\alpha_{\max}$; all segments (positions) outside the 'vision' of the car, which can be seen as a flashlight in the car movement direction with an angle of two times $\alpha_{\max}$.

(e) Remove all positions that are out of range. This can be seen as a view between two distances from the current position. This view depends on the current velocity, the minimum and maximum acceleration (max. braking and max. speeding) and the possible steering angle (between $-\alpha_{\max}$ and $+\alpha_{\max}$). Next to that, it is bounded by $\Delta lt_{\min}$ and $\Delta lt_{\max}$.

(f) Remove all remaining positions that causes the car to cut off an edge of the racetrack. That is, the straight line from $\vec{p}_t$ to $\vec{p}_{t+1}$ crosses the trackline. Also positions that causes the car to pass the finish line ($\forall \vec{p} \in P_{\text{finish}}$[6] or $\forall \vec{p} \in P_{\text{start}}$[7]) are removed.

(g) The remaining positions are stored in P_{new}.

2. Define the possible new velocity vectors $\vec{v}_{t+1}$ for each new position $\vec{p}_{t+1} \in P_{\text{new}}$. In fact, only the new velocities (length of $\vec{v}_{t+1}$) have to be evaluated, because the direction is fixed by $\vec{p}_{t+1}$. This velocity is in the set V_v, furthermore it is bounded by the current velocity vector $\vec{v}_t$ and the minimum and maximum acceleration and, of course, $\vec{p}_{t+1}$ itself (in relation with $\vec{p}_t$). Define these velocities as $V_{\text{new}}^{\vec{p}_{\text{new}}}$.

3. Return $U_{\vec{x}}$. That is, all possible new states $\vec{x}_{t+1} \in X_{\text{new}}$, where $X_{\text{new}} = (\vec{p}, \vec{v}), \forall \vec{p} \in P_{\text{new}}, \forall \vec{v} \in V_{\text{new}}^{\vec{p}_{\text{new}}}$.

Computing the view in step 1e can be very nasty, because a lot of variables with dependencies (according to RARS vehicle dynamics) are involved. Furthermore, when some variables are not well defined, this view can cause the actionset $U_{\vec{x}}$ to be empty. For instance, when the time view variables $\Delta lt_{\min}$ and $\Delta lt_{\max}$ are chosen too large; all posible new positions can be only on dirt positions on the trackmap or even outside the trackmap. Or when the maximum possible acceleration does not result to be within the velocity set V_v at the next state within a time of $\Delta lt_{\max}$.
An example of executing the action algorithm is in Figure 3.6. The majority of the steps in the algorithm are graphically shown in the figure.

The action algorithm is the major part of the optimization algorithm. By defining the variables in a clever way, it is possible to reduce the set with possible new states X_{new} such that the computation time is limited. Note that setting the variables too tight (in order to reduce computation time), the search space for the ideal racing line is limited. This can cause a far from optimal solution.

3.4.3 Reward

Nou we can deal with states, times and actions, we have to define the reward. This is actually quite simple. The reward $r(\vec{x}_t, \vec{u}_t)$ is simply the time it takes to get from $\vec{x}_t$ to state $\vec{x}_{t+1}$, where $\vec{x}_{t+1}$ depends on $\vec{u}_t$.

[6] This results in not passing the finish line, thus restricts the car to drive further than one lap.
[7] This results in not passing the start line, thus restricts the car to turn around and drive direct to the finish.

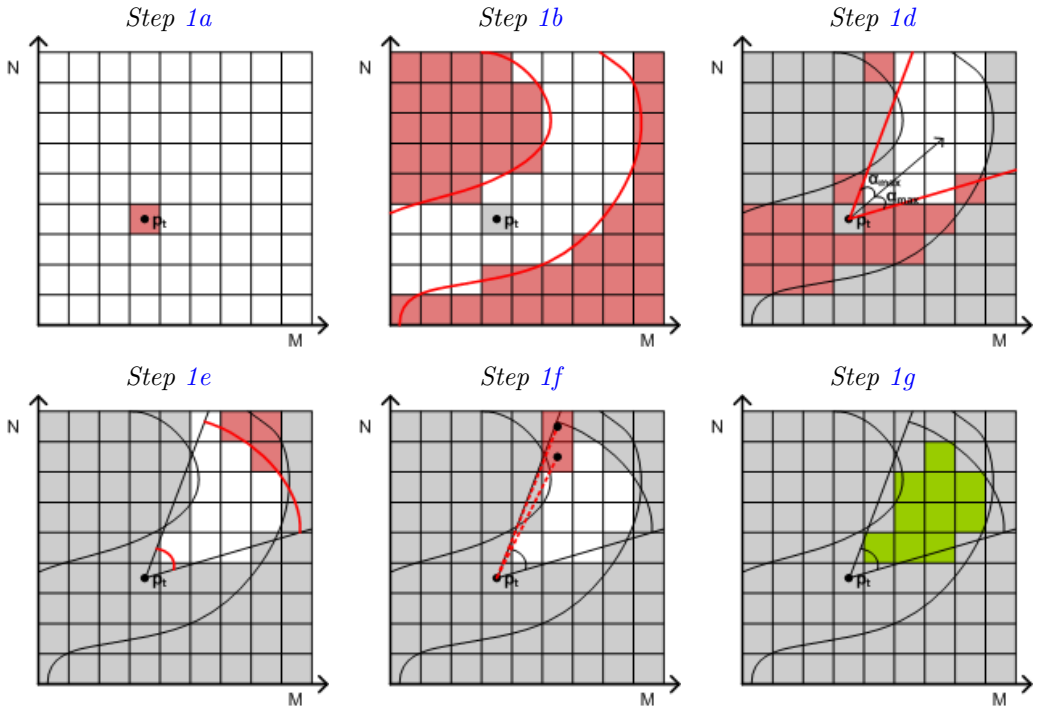


Figure 3.6. Graphical example of performing steps within the action algorithm, that is reducing segments (positions) in set P. Grey and red segments are not in P, white and green segments are in P. Furthermore, red segments are segments removed from P in that particular step (according to red restriction lines) and green segments are the final segements in $P_{\text{new}} \subset P$.

To compute the time (the reward), we assume that acceleration is linear between two states. The time is the distance between the current and the next state divided by the average velocity, that is

$$r = \frac{\sqrt{(m_{t+1} - m_t)^2 + (n_{t+1} - n_t)^2}}{\frac{1}{2}(v_{t+1} + v_t)}, \tag{3.6}$$

with $\vec{p}_t(m_t, n_t)$ the position vector at time step t in meters and v_t the velocity in $\vec{v}_t$ (the length of $\vec{v}_t$) in meters per second. Note that this reward is of quantity $\frac{\Delta m}{m/s}$, where $\Delta m = \Delta n$. To convert it to seconds, just divide it by a factor of size Δm.

3.5 Optimization

Now we have to find the ideal racing line. We can use the basics of (discrete) Markov decision processes (Section 2.3), with the use of the variables and algorithm defined in the previous section. The states, actions and rewards are defined. We are now dealing with a deterministic problem; every action $\vec{u}$ leads to one and only one new state $\vec{x}$, therefore we do not have to determine transition probabilities.
The ideal racing line is that policy π that minimizes the total reward, because the reward is the driving time. To use the algorithms and defintions in Section 2.3, we multiply the reward r as in Equation (3.6) by minus one (-1). Now we are dealing with a maximization problem; seeking the maximum reward.

3.5.1 Decision Epochs

We did not mention decision epochs for this problem, until now. The decision epochs is not fixed in time or from a known distribution for instance, but still quite simple. The driver has to make a decision (chose an action) if has reached the new position. The driver chooses an action from the actionset $U_{\vec{x}}$, which is the result of the action algorithm in Section 3.4.2. This results in a new position on the track and a new velocity when that position is reached[8]. At the moment (in time) the car reaches this new position, there is chosen a new action using the action algorithm. The decision epochs is bounded in time by the time view variables. The time between one decision and another (the time between two positions on the racetrack) can be any real number between $\Delta lt_{\min}$ and $\Delta lt_{\max}$ seconds.

3.5.2 Value function

The value function needs some adaptation for (policy) evaluation, because we are not dealing with a predefined length of the number of states for a certain trajectory. The policy 'stops' when one lap is driven. This length ('finish' in Equation (3.7)) is not infinity, because of the time view variables and the fact we stop when the car is at the finish line for the second time (when a full lap is driven). Computing the value function for a certain policy, can be seen as iteratively summate rewards until the car has reached the finish line to capture the issue of driving only

[8] Acceleration is linear over time between the current and the new position from the current to the new velocity. Therefore an acceleration unequal to zero causes the velocity to be equal the new velocity only when the new position is reached.

one full lap. The value function V for a certain policy π and a certain starting state $\vec{x_0}$ is the total reward over the policy π. The value function can be described by

$$V^{\pi}(\vec{x}_0) = \sum_{t=0}^{\text{finish}-1} \mathbb{E}_{\vec{x}_0,\pi} r(\vec{x}_t, \pi(\vec{x}_t)). \tag{3.7}$$

Note that we do not take the reward at $t = \text{finish}$. This reward is not defined, because it has no next state $\vec{x}_{t+1}$. The calculation of V^{π} is a summation until $\vec{p}_t \in P_{\text{finish}}$ (until we are at the finish line)[9]. Furthermore $\vec{x}_0 = (\vec{p}_0, \vec{v}_0)$, with $\vec{p}_0 \in P_{\text{start}}$. The 'direction' of the velocity vector $\vec{v}_0$ is the direction with a right angle with the finish line. The velocity of the car can be 0, but a certain other starting velocity is also possible. For instance the velocity of $\vec{x}_{finish}$ of a known solution to simulate a lap at full speed. $\pi(\vec{x}_t)$ results in an action $\vec{u}_t \in U_{\vec{x}_t}$. The action set $U_{\vec{x}_t}$ is determined by applying the action algorithm (Algorithm 3.1). We are now seeking a policy π^* for which

$$V^*(\vec{x}_0) = \max_{\pi} V^{\pi}(\vec{x}_0). \tag{3.8}$$

3.5.3 Policy Evaluation

We can use the policy evaluation algorithm (Algorithm 2.1), to evaluate a certain policy, with minor adjustments. The set equations that has to be solved, based on Equation (3.7), is

$$V^{\pi}(\vec{x}) = r(\vec{x}, \pi(\vec{x})) + V^{\pi}(\vec{j}), \tag{3.9}$$

where $\vec{j}$ depends on the action $\vec{u} \in U_{\vec{x}}$ based on $\pi(\vec{x})$. Another adjustment for instance is when to stop iterating, that is; when is the car at the finish line? The stop criterium can hold; stop $\vec{p}_t \in P_{\text{finish}}$ for current time t. Further adjustments are not mentioned.

3.5.4 Optimization Algorithms

To seek an optimal policy we can use the value iteration algorithm (Algorithm 2.2) or the policy iteration algorithm (Algorithm 2.1) for example. As with policy evaluation, these algorithms need some minor adjustments to fit the problem. For instance, not every state $\vec{x}$ should be evaluated when the algorithm says, but the action algorithm described in Section 3.4.2 should be used to reduce the set of states to evaluate. It may need some adjustments to handle the backward recursion.
Further (minor) adjustments one should think of during implementation. They are not mentioned in this paper. The model definitions, discretizations and the action algorithm are the basic elements to solve the ideal racing line problem. They provide a fundament to implement an optimization algorithm.

[9] Under the assumption that the time view variables Δlt_{min} and Δlt_{max} are well defined, the car always reaches a position on the finish line, because of Step 1f of the Action algorithm.

3.6 Implementation

Implementation of the model formulation and algorithms mentioned in the previous section is beyond the assignment of the BMI paper. The BMI paper is in essence a literature study. If you are interested in implementing these algorithms, you can use any programming language you prefer. The discretization causes a lot of computation time, so C++ is a language that should be suitable (because it is quite fast). Furthermore, the free software provided by RARS can be used to program the racecar. See the RARS website[10] for more information and downloads. There is also a tutorial for writing your own racecar-robot.

[10] http://rars.sourceforge.net

Chapter 4

Discussion

Chapter 2 described several optimization methods. These methods may be suitable for solving the ideal racing line problem. In fact, some racecars competing within the RARS formula one seasons used some of these methods as mentioned in Section 3.2.2. The basic ideas behind these techniques where used and adapted for the problem. The continuous property of the problem makes discrete methods as discrete Markov decision processes hard to apply. Next to that, the vehicle dynamics and the ability to pass cars for example causes the problem to be more complex. During the formula one seasons, generalization methods which did not use exact transformations of the problem properties achieved the best results. For example optimization methods such as evolutionairy algorithms or algorithms in the field of neural networks. They 'learned' how to drive under the different circumstances.

The fact that the problem is continuous causes some major difficulties in applying a discerte Markov decision process. Discretization is the most important step. The problem could be solved when proper discretization is used to work around the curse of dimensionality, such as mentioned in the previous chapter. Computation time will probably still be large, but manageable.
The restrictions made in Chapter 3 have also a negative consequence; the solution will probably be not very close to the optimal solution obtained by algorithms used within the best performing cars wihtin RARS. It is expected that solving the problem while using the formulations made in Chapter 3 will result in a solution that will give a racing line that is within the neighbourhood of racing lines used by professional racecars, but probably not as good.
Adding some features to the solution method described could improve the solution using discrete MDP. One can think of adding to the states $\vec{x}$ the free distance in front of the car or the angle of the car with the racetrack for example. But adding these characteristics also causes the problem to grow, which causes computation time to grow. So adding features should be done carefully.

The implementation of the proposed solution method is for further research.

Bibliography

[1] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[2] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[3] D.P. Bertsekas. *Neuro-Dynamic Programming: An Overview [Presentation sheet]*. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2006.

[4] R. Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.

[5] S. Dreyfus. *Richard Bellman on the birth of Dynamic Programming* in *Operations Research*, Vol. 50, No. 1, January-February 2002, pp. 48-51. Informs, 2002.

[6] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[7] G. Koole. *Lecture notes Stochastic Optimization*. Department of Mathematics, Vrije Universiteit Amsterdam, 2006.

[8] T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[9] M.L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley, 1994.

[10] M.E. Timin. *Robot Auto Racing Simulator*, 1995. Website: http://rars.sourceforge.net/.