Stockholms universitet

From Markov chains to Markov decision processes

Niclas Lovsjö

Kandidatuppsats i matematisk statistik
Bachelor Thesis in Mathematical Statistics

Kandidatuppsats 2015:5
Matematisk statistik
Juni 2015

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

Mathematical Statistics
Stockholm University
Bachelor Thesis **2015:5**
http://www.math.su.se

From Markov chains to Markov decision processes

Niclas Lovsjö*

June 2015

Abstract

In this bachelor's thesis we will try to build an understanding of Markov decision processes as an extension to ordinary discrete time Markov chains in an informal setting. The intended reader is assumed to have knowledge of basic probability theory. Throughout the text we use trivial examples of applying the theory, which we believe builds good intuition. As a last section we show how a basic Reinforcement learning algorithm, namely Q-learning, can be used to find a solution of an MDP-problem.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden. E-mail: niclas.lovsjo@me.com. Supervisor: Mathias Lindholm.

Abstract

In this bachelor's thesis we will try to build an understanding of Markov decision processes as an extension to ordinary discrete time Markov chains in an informal setting. The intended reader is assumed to have knowledge of basic probability theory. Throughout the text we use trivial examples of applying the theory, which we believe builds good intuition. As a last section we show how a basic Reinforcement learning algorithm, namely Q-learning, can be used to find a solution of an MDP-problem.

Sammanfattning

Med denna kandidatuppsats ska vi försöka bygga upp en förståelse för Markov beslutsprocesser, som en utökning till grundläggande Markovkedjor i diskret tid. Den förmodade läsaren antas ha kunskap om grundläggande sannolikhetsteori. Vi kommer genom uppsatsen använda enkla exempel som hjälpmedel till att introducera teorin, vilket vi tror ger bättre förståelse för materialet. Som ett sista kapitel kommer vi titta på hur en Belöningsbaserad inlärnings-algoritm, vid namn Q-learning, kan användas för att lösa en Markov beslutsprocess.

Acknowledgements

I would like to thank my supervisor Mathias Lindholm for helping me in all aspects of writing this thesis. Mathias has helped me not only to understand the material in this thesis, but how to approach learning mathematical statistics in general.

Contents

Introduction

Consider the way you teach a dog to sit. Usually it is done by you telling it to sit, and most often this won't work the first couple of times. Then, all of a sudden, it follows your command and sits. Now you should obviously try to reinforce this good behaviour by giving it some kind of reward, say a biscuit, and slowly it will learn that each time it follows your sit-command it will receive a biscuit.

Now could we take this idea of reinforcing desired behaviour and apply it in mathematical algorithms? In fact this idea has been around since the early days of computers, and as one can imagine the complexity of tasks computers can learn has grown accordingly. In this thesis we will try to build the foundation needed to understand these algorithms from a mathematical point of view, as opposed to the computer science approach used in *most* literature, for example Sutton and Barto (1998) and Mitchell (1997)[1]. We will learn that to really understand learning algorithms, one has to have knowledge of various subjects, not only mathematical statistics.

The theory that we are interested in here is based on Markov theory. Say we have an *ordinary discrete time Markov chain*, at the basic case each time-step it will have a probability distribution of *states* to where this *stochastic process* is going to *transition*. We can think of this transition as being an *action*, albeit a single-choice action, that is forced upon the process at each time step. Now if we make this a multi-choice action instead, that is, each action has its own transition probabilities. After each action, and after the transition to the next state, we supply the process with some *reward*. The problem now comes down to finding a set of actions, called a *policy*, as to maximize the total reward of moving through some process. We call a process having these properties a *Markov Decision Process*(MDP).

An MDP-problem can be solved using optimization theory, more precisely *Dynamic Programming*(DP). DP is a vast area of optimization, and we will focus on a part which we call *backward induction algorithm*. This can briefly be described as a way of breaking down optimization problems into sub-problems, solving these and bringing them back together to solve the main problem. A central concept in DP is the Bellman-equation, which we will go through in the MDP-section below.

As computers evolved and simulation-techniques became more powerful other techniques, with roots in DP, was developed. One of these is *Reinforcement learning*(RL), which combines DP with *Monte Carlo-simulations*. That is, we simulate and optimize at the same time and change the behaviour of an *agent* while it is interacting in this process. In this thesis we will not focus as much on the family of RL-algorithms, but instead put focus on taking the reader from the more familiar ordinary Markov chains to the concept of Markov decision processes, and then introduce a classical RL-algorithm as a way of solving MDP's. This RL-algorithm is called *Q-learning*.

[1]When searching for information about this subject, one is most often directed to courses at various computer science departments, very seldom the mathematics department.

We have divided this thesis into three blocks - *Markov Chains*, *Markov Decision processes* and *Reinforcement Learning*. The style of writing we will use is to not include formal theorems with proofs, but instead use them in an informal manner and give reference to where the proofs can be found. Furthermore, the derivations which will be included are the ones we think is central to understanding that particular section, and we have chosen to include them in the running text of that reason. We will also use basic examples to explain the concepts with where we find it to be useful. The nature of both the examples and derivations are such that they can easily be skipped without disturbing the exposition. In the beginning of each block we will declare if we have followed a particular exposition of a certain reference, otherwise references are given throughout the text. In general, the references are taken from educational literature as opposed to scientific articles.

Markov chains

The objective of this section is to declare what a Markov chain(MC) is and what properties it has. Throughout this thesis we will only consider finite discrete time processes. This section partly follows the exposition of (Ross, 2010), and some ideas are taken from (Taylor, 2012).

The basics of MC

We define a *process* as something that changes over time. Let X_t denote the state of this process at time t. Furthermore we make the assumption that each X_t is a random variable and that $t \in \mathbb{Z}^+$. We limit this process to be ran inside some system, which consists of different *states* $\mathcal{S}$, i.e. $X_t \in \mathcal{S}$, and that the process *always* is in some state in this system. If the transitions between these states is stochastic, we have a *stochastic process*, and we say that the process is in state i at time t if $X_t = i$. Each such move from state i to j in one time-step has a *transition probability* p_{ij}. This probability is fixed, i.e it does not change over time. This process has the property that the probability of moving from some state i to another state j is independent of the states visited prior to i. Formally we define this as, $P(X_{t+1} = j|X_t = i, X_{t-1} = i_{t-1}, ..., X_1 = i_1, X_0 = i_0) = P(X_{t+1} = j|X_t = i) = p_{ij}$ and we say that a process which has this property is markovian.

Let p_{ij}^n denote the probability of moving from i to j in n time-steps. Whenever $p_{ij}^n > 0$ for some number of timesteps $n \geq 0$ we say that j is *accessible* from i, and if two states are accessible to each other they are said to *communicate*. The transition probability p_{ij}^n partitions the process into communicating *classes*, for those states that communicate, which implies that each state only can belong to one class. We say that a class is a *closed* class, if the process only can enter states in that class but never leave. If all states communicate, i.e. consists of only one class, the chain is said to be *irreducible*.

The regular properties of probabilities apply, i.e. we have that $p_{ij} \geq 0$, $\sum_{j=0}^{\infty} p_{ij} = 1$ and we also constrain the value of each X_t to be a positive integer. We say that a sequence of random variables $X_0, X_1, X_2...$ that has these properties is a *Markov chain* X.

Assume that the probability of re-entering state i, while starting in i, is f_i. Now if $f_i = 1$ we say that i is a *recurrent* state, and *transient* for $f_i < 1$. From the markovian property we know that each time the process enters i it will be conditionally independent of the states prior to i, including i itself. For a recurrent state this means that i will be visited infinitely many times. For this to be true there can not be any state that absorbs the process, i.e. a state who only communicates with itself. We call such a state an *absorbing* state. On the other hand, for a transient state j there will for each visit be a positive probability $1 - f_j$ that the present visit is the last, i.e. a transient state will only be visited a finite number of times. If the Markov chain is finite, this implies that all the states can't be transient. To see this, assume we have a process containing only transient states. Say that the present visit is the final time j was visited. As the number of timesteps goes to infinity, this will occur to every

transient state. Since every state is transient, it will end up being in no state - which contradicts the property of Markov chains always being in some state. That is, every Markov chain needs to have at least one recurrent state.

If the expected time until the process returns to the the recurrent state i is finite while starting in i, then we say that i is *positive recurrent*. A state in a Markov chain is said to be *periodic* if it can return to some state i, while starting in i, only under some multiple $d(i)$ of steps, where d is a positive integer satisfying $d > 1$. If instead $d = 1$ we say that the state is *aperiodic*. States that are both aperiodic and positive recurrent is said to be *ergodic*, which we will discuss the implications of further in the next section.

Example 1: Say we have a 3-state Markov chain with states $\{1, 2, 3\}$ and the transition probabilities as in Figure 1.

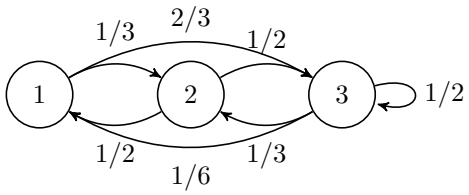

Figure 1: Example of a Markov chain

with the accompanying transition matrix,

$$P = \begin{pmatrix} 0 & 1/3 & 2/3 \\ 1/2 & 0 & 1/2 \\ 1/6 & 1/3 & 1/2 \end{pmatrix}$$

We see for example that $P(X_{t+1} = 2|X_t = 1) = 1/3$, i.e. the probability of transitioning to 2 while being in 1 is $1/3$ at some time t. Furthermore, we note that Example 1 is ergodic and irreducible.

The Chapman-Kolmogorov equations

From the last section we know that p_{ij}^n defines the probability of moving from i to j in n time-steps, more formally we have that $p_{ij}^n = P(X_n = j|X_0 = i)$ for any $n \geq 1$. Now using the first law of total probability and the Markov property

we see that,

$$\begin{aligned} p_{ij}^n &= P(X_n = j | X_0 = i) \\ &= \sum_{k \in K} P(X_n = j, X_r = k | X_0 = i) \\ &= \sum_{k \in K} P(X_n = j | X_r = k, X_0 = i) P(X_r = k | X_0 = i) \\ &= \sum_{k \in K} P(X_n = j | X_r = k) P(X_r = k | X_0 = i) \\ &= \sum_{k \in K} p_{ik}^r p_{kj}^{n-r}, \end{aligned} \tag{1}$$

where K is the set of all possible states, and r is a non-negative integer $r < n$. We call (1) the Chapman-Kolmogorov equations, and they essentially say that the probability of moving from some state i to another state j in n-steps is equal to the sum of the probabilities of all the intermediate steps k between i and j.

What makes this convenient is that we can now express the n-transition in form of matrix multiplication. If we let $P^{(n)}$ denote the matrix holding the probabilities for n-step transitions, (1) implies that $P^{(n)} = P^{(r)} P^{(n-r)}$. By induction, see (Taylor, p.10, 2012), it can be shown that,

$$P^{(n)} = P^n \tag{2}$$

i.e. raising the one-step probability matrix to the power of n gives the probabilities of moving from state i to j in n steps.

Now using the Markov chain from Figure 1, we see for example that $P(X_{t+2} = 3 | X_t = 1) = P_{1,3}^2 = 1/2$.

$$P^2 = \begin{pmatrix} 0.28 & 0.22 & 0.50 \\ 0.08 & 0.33 & 0.58 \\ 0.25 & 0.22 & 0.53 \end{pmatrix}$$

Which is confirmed by the step-by-step computations: we have the feasible transitions $\{A \to B \to C, A \to C \to C\}$ with probabilities $\frac{1}{3}\frac{1}{2} + \frac{2}{3}\frac{1}{2} = 1/2$.

Hitting times

The following subsection is inspired by (Taylor, p.13, 2012).

We define a *hitting time* as the first time t the process enters some non-empty set of states $C \subset \mathcal{S}$, under the assumption that C is a closed communicating class. Now, if we have a discrete time Markov chain X on $\mathcal{S}$, and let $C \subset \mathcal{S}$ be a closed communicating class for X, we define the *absorption time* $\tau^C \in \mathbb{N}$ of C as,

$$\tau^C = \begin{cases} \min\{t \geq 0 : X_t \in C\} & \textit{if } X_t \in C \quad \textit{for some } t \geq 0, \\ \infty & \textit{otherwise}, \end{cases}$$

and the *absorption probability* of entering C while starting at state i by

$$\begin{aligned} h_i^C &= P(\tau^C < \infty | X_0 = i) \\ &= P(\exists t < \infty : X_t \in C | X_0 = i) \\ &:= P(X_t \in C | X_0 = i). \end{aligned}$$

Now, to solve for a vector of probabilities $h^C = (h_1^C, h_2^C, ...)$ we set up a linear equation system as,

$$\begin{cases} h_i^C = 1 & \text{if} \quad i \in C \\ h_i^C = \sum_{j \in \mathcal{S}} p_{ij} h_j^C & \text{if} \quad i \notin C. \end{cases} \tag{3}$$

We have the trivial case $h_i^C = 1$ if $i \in C$. To see that (3) holds for the case $i \notin C$ note that,

$$\begin{aligned} h_i^C &= P(X_t \in C | X_0 = i) \\ &\overset{ltp}{=} \sum_{j \in \mathcal{S}} P(X_t \in C | X_1 = j, X_0 = i) P(X_1 = j | X_0 = i) \\ &= \sum_{j \in \mathcal{S}} P(X_t \in C | X_1 = j, X_0 = i) p_{ij} \\ &\overset{Mp}{=} \sum_{j \in \mathcal{S}} P(X_t \in C | X_1 = j) p_{ij} \\ &= \sum_{j \in \mathcal{S}} p_{ij} h_j^C, \end{aligned}$$

where *ltp, Mp* denote that we use the law of total probability and the Markov property. We will see in the next chapter that hitting times can be useful for evaluating policies in Markov decision processes.

Stationary distributions

Finishing this section we want to discuss the asymptotic behaviour of Markov chains. If a Markov chain has absorbing states, then under the assumption that the chain will run sufficiently long, the Markov chain will be more and more likely to have been absorbed by this state. On the contrary if all the states are non-absorbing we can say something about the distribution for a Markov chain as $t \to \infty$. In fact, for an irreducible ergodic Markov chain letting

$$\nu_j = \lim_{n \to \infty} p_{ij}^n, \quad j \geq 0,$$

ν_j is the unique non-negative solution of

$$\begin{aligned} \nu_j &= \sum_{i=0}^{\infty} \nu_i p_{ij}, \quad j \geq 0, \\ \sum_{j=0}^{\infty} \nu_j &= 1, \end{aligned} \tag{4}$$

This is a convenient way of representing Markov chains, since instead of using (2) with higher and higher powers until we find convergence, we can solve for the stationary distribution using the set of linear equations in (4).

Markov decision processes

We attempt here to make the transition from Markov chains to Markov decision processes. The material used is mostly based on (Sutton and Barto, 1998) and (Taylor, 2012).

The basics of MDP

Assume that we have an ordinary Markov chain, at each time-step t this process makes the decision to move according to some transition probability distribution as described in the last section. Now, if for each such time-step we introduce a set of actions to be taken before this transition happens, with a probability distribution for each such action. One can think of this as instead of having a process that transitions by itself, we now have a decision-making *agent* moving through the process. This agent chooses an action $a_i \in \mathcal{A}$. That is, for each state s_t we have a finite set of feasible actions $\mathcal{A}_{s_t}$. We call each such pair $\{(a_1, s_t), (a_2, s_t), ..., (a_n, s_t)\}$, the *state-action* pairs.

We call the environment of this agent its *nature*. After an action has been taken by the agent, the nature responds by moving the agent to a new state according to some probability distribution P^a_{ij}, for which the markovian property hold[2], i.e. $P^a_{ij} = P(S_{t+1} = j | S_t = i, a_t = a, ..., S_{t_0} = i_0, a_{t_0} = a_0) = P(S_{t+1} = j | S_t = i, a_t = a)$. Furthermore, the agent is rewarded with some bounded value $r^a_{ss'}$ for entering state s' from s under action a. We call a process having these properties a Markov decision process.

The rewards are setup as to make the agent solve a given problem. In some cases we want it to be defined by taking an action and then entering a state. In others we just want it to be rewarded for entering a state, independent of the action that brought it there. In the simplest case, we just reward the agent for entering one state in the process, usually the absorbing state. This gives that, the reward $r^a_{ss'}$ will in some cases be deterministic and in other cases stochastic, depending of the nature of the problem. The agent will have some starting state, and some absorbing terminal state. In some cases we need a closed subset $C \subset \mathcal{S}$ of terminal states.

Each subset of actions through the states from starting state to terminal state forms a *policy*, and is denoted π. The problem for the agent comes down to finding a path through nature that maximizes the rewards received while following this policy. If some policy is better or equally good, with respect to some given objective, than all the other feasible policys, we say that it is an optimal

[2]Notice that we differ between the n-step transition probability matrix $P^n_{ij}$from the MC-section, and P^a_{ij} the probability of transitioning to j from i given *action* a.

policy and denote it with π^*. In the case where there are multiple policies that are better than the others, *and* mutually equally good, we say they form a set $\Pi^* = \{\pi_1^*, ..., \pi_n^*\}$ of optimal policies.

Example 2: We would like to introduce a basic two-state, two-action MDP to give some more intuition to this. Assume that we have two states $\{1, 2\}$ in which we can take two actions $\{\alpha_i, \beta_i\} \in \mathcal{A}$ in each state i. That is, we have the actions $\{\alpha_1, \beta_1, \alpha_2, \beta_2\}$. Let $r_i \in R$ be a reward-vector for entering state i. We assign the following transition probabilities P_{ij}^a and rewards r_i to the above scenario,

$$\begin{aligned}
\alpha_1 &: \ P_{1,2}^{\alpha_1} = 0.8, \qquad P_{1,1}^{\alpha_1} = 0.2 \qquad\qquad r_1 = 1 \\
\beta_1 &: \ P_{1,2}^{\beta_1} = 0.5, \qquad P_{1,1}^{\beta_1} = 0.5 \qquad\qquad r_2 = 0 \\
\alpha_2 &: \ P_{2,1}^{\alpha_2} = 0.5, \qquad P_{2,2}^{\alpha_2} = 0.5 \\
\beta_2 &: \ P_{2,1}^{\beta_2} = 0.9, \qquad P_{2,2}^{\beta_2} = 0.1
\end{aligned}$$

As an example starting in state 1, the agent takes action a_1 and then moves back to 1 with probability $P_{1,1}^{\alpha_1} = 0.2$ or to state 2 with $P_{1,2}^{\alpha_1} = 0.8$, then receives $r_1 = 1$. On the other hand if it chooses b_1 it moves with probability $P_{1,1}^{\beta_1} = 0.5$ back to state 1 and with probability $P_{1,2}^{\beta_1} = 0.5$ to state 2. Since we only receive a reward of entering state 1, the task of the agent is trying to choose the actions that brings it to 1 with highest probability.

As with the Markov chain-graph from Example 1, we can illustrate this with a Markov decision process-graph, see Figure 2,

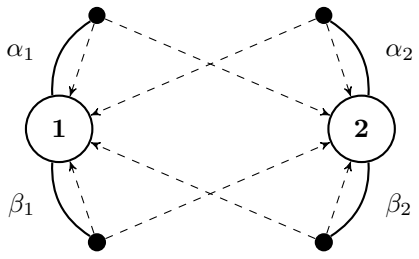

Figure 2: A Markov Decision graph. The *thick line* represents an action a and the *dotted line* the possible next states while taking a specific action.

where we have omitted the actual transition probabilities given an action for sake of clarity. This process has no absorbing state, and all states are therefore recurrent.

Now assume that we have a scenario such that the agent starts in state 1, and lives for a two-step time-period, in which it wants to maximize its rewards.

Each policy dictates what action to take given a certain state. That is, we have the following policies for Example 2: $(\{\alpha_1, \alpha_2\}, \{\alpha_1, \beta_2\}, \{\beta_1, \alpha_2\}, \{\beta_1, \beta_2\})$. Assume we decide to follow a policy, say $\pi_\alpha = \{\alpha_1, \alpha_2\}$, i.e. at state 1 we choose α_1 and at state 2 we choose α_2. Let R_t be the total reward, i.e. *return*, for some state at t. That is, here this means that R_1 is the total return starting in 1 and stopping after two transitions. The following calculations give the expected value of following π_α for this scenario, which we denote $E_{\pi_\alpha}[R_1|S_1 = 1]$. Thus for $t = 1$ we have,

$$\begin{aligned} E_{\pi_\alpha}[R_1|S_1 = 1] &= (r_1 + E_{\pi_\alpha}[R_2|S_2 = 1])P_{1,1}^{\alpha_1} \\ &\quad + (r_2 + E_{\pi_\alpha}[R_2|S_2 = 2])P_{1,2}^{\alpha_1} \\ &= (r_1 + (r_1 P_{1,1}^{\alpha_1} + r_2 P_{1,2}^{\alpha_1}))P_{1,1}^{\alpha_1} \\ &\quad + (r_2 + (r_1 P_{2,1}^{\alpha_2} + r_2 P_{2,2}^{\alpha_2}))P_{1,2}^{\alpha_1}, \end{aligned}$$

which also can be written as,

$$E_{\pi_\alpha}[R_1|S_1 = 1] = \sum_{i=1}^{2} P_{1,i}^{\alpha_1}(r_i + E_{\pi_\alpha}[R_2|S_2 = i]). \tag{5}$$

By doing this for all four policies, we see that policy $\pi = \{\beta_1, \beta_2\}$ will maximize the expected return, with the value $E_{\pi^*}[R_1|S_1 = 1] = 1.2$.

Notice that we differ between the random variable R_t, the return at time t, and the deterministic parameter r_i, the reward for entering i. Also, note that in this example we have used that $r_{1,1}^{\alpha_1} = r_{1,1}^{\beta_1} = r_{2,1}^{\alpha_2} = r_{2,1}^{\beta_2} = r_1$ which must not be the case in a more general setting, as noted earlier.

Now to find the optimal policy, we would have to compare all policies for all time-steps. We will in the next section see how this can be done iteratively. In the final section we use this example to find the optimal policy by simulations.

For this solution to be tractable we need perfect knowledge of the transition probabilities, the reward-vector and we need a time-setup that is finite. If applied to some real-life examples, it is usually not the case that we have knowledge of this. In our example we had no terminal-state but instead restricted the process to go on for two time-steps.

The value function

Now let us formalize the idea from the previous section in a more general setting. Let each state in the process have a value $V^\pi(s)$ for a given policy π. We define this value as the expected total reward R_t from being in s at time t when following policy π, i.e. $V^\pi(s_t) = r_{s_t} + \gamma r_{s_{t+1}} + \gamma^2 r_{s_{t+2}} + ... + \gamma^k r_{s_{t+k}} = \sum_{i=0}^{k} \gamma^i r_{s_{t+i}}$, where γ is a *discount factor* satisfying $0 \leq \gamma \leq 1$. This implies that using a discount factor of $\gamma = 0$ only the immediate reward from choosing a_i in s_t will be considered, and using $\gamma = 1$ all of the rewards in policy π will have equal weight for $V^\pi(s_t)$. Note that in Example 2 we used $\gamma = 1$. We say that a policy π that maximizes $V^\pi(s)$ for all states s is an optimal policy and

denote it with π^*. That is,

$$\pi^* = \max_{\pi} V^{\pi}(s). \tag{6}$$

Sometimes it can be useful to introduce a probability of taking action a in s, given that we follow π by $P^{\pi}(s,a)$. We will discuss this concept of, having a probability of *choosing* an action, later in the section *"ϵ-greedy policy"*, because it has more impact in Reinforcement learning. Note that in Example 2, we used $P^{\pi}(s,a)$ which only took values 0 and 1. We now derive a way of representing $V^{\pi}(s)$ which will make it easier for us to understand how we can find and compare these policies, and most importantly it can be linked to a known equation guaranteeing optimality. We have,

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\left\{R_t | s_t = s\right\} \\
&= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right\} \\
&= E_{\pi}\left\{R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s\right\} \\
&= \sum_a P^{\pi}(s,a) \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_{t+1} = s'\right\}\right] \\
&= \sum_a P^{\pi}(s,a) \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V^{\pi}(s')\right].
\end{aligned} \tag{7}$$

As noted earlier, to be able solve for $V^{\pi}(s)$ we need $V^{\pi}(s_i)$ for the subsequent states $s_i = s', s'', ..., s^{N-1}$. In practice, we set the value of the terminal-state s^N to be zero, but implement a reward for arriving at this terminal state. This means that the reward received for going to the terminal-state, will be incorporated in the value of the state prior to the terminal-state. Furthermore, notice the similarities between (5) and the last line of (7), which give some more intuition to the value-function.

Now this only states the value of s under the policy π. Since we are interested in finding the optimal action at each state we need to, at each time step, choose the action a that maximizes $V(s)$. That is, for every state s the maximum value is attained at,

$$V^*(s) = \max_a \left\{\sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V^*(s')\right]\right\}, \tag{8}$$

where $*$ denotes that it is the maximum value. Now, if we store each such maximizing action a, denoted a^*, in a vector, this vector will contain the optimal policy. That is,

$$a^* = \arg\max_a V(s),$$

and

$$\pi^* = \{a^*_s, a^*_{s'}, ..., a^*_{s^{N-1}}\}, \qquad \forall s \in \mathcal{S},$$

and we note that s^N is some terminal state not included in the policy per definition. This implies that following π^*,

$$V^{\pi^*}(s) = V^*(s).$$

We want to emphasize that for this to hold, the action a^* guarantees optimality at s if and only if the process follows an optimal policy in all subsequent states $s', s'', ..., s^N$, see *the principle of optimality* in (Bellman, p.4, 1954).

The Bellman equation

Equation (8) is the so called Bellman-equation[3]. The Bellman-equation can be shown to guarantee optimality, see e.g. (Ross, 1968). We will discuss this in an intuitive sense, which also will give us an understanding of how MDP's can be solved.

As noted in the last section, what is distinctive of (8) is that the choice a^* takes the value of $V^*(s')$ into account, that is for each time-step the value of $V^*(s)$ uses the optimal choice a^* in s'. $V^*(s')$ takes into account $V^*(s'')$ and so on, down to $V^*(s^N)$ for some terminal state s^N. We can use this to set up an iterative algorithm. Notice that we want to store both the optimal value, and the optimal action. We call this value- and policy-iteration, respectively.

Now if we set $s = s^N$ and let

$$V^*(s^N) = r_{s^N} \tag{9}$$

for all $s \in \mathcal{S}$. Substitute $N-1$ for N and let,

$$\begin{aligned} V^*(s^{N-1}) &= \max_{a \in \mathcal{A}} \left\{ \sum_{s^N} P^a_{s^{N-1}s^N} \left[r^a_{s^{N-1}s^N} + \gamma V^*(s^N) \right] \right\}, \\ a^*_{s^{N-1}} &= \arg\max_{a \in \mathcal{A}} \left\{ \sum_{s^N} P^a_{s^{N-1}s^N} \left[r^a_{s^{N-1}s^N} + \gamma V^*(s^N) \right] \right\}, \end{aligned} \tag{10}$$

and repeat this until $s^{N-1} = s$. We call each such procedure from s^N to s an *epoch*. This procedure iterates until each $V(s^i)$ has converged. As noted earlier, the storage vector $\{a^*_s, a^*_{s'}, a^*_{s''}, ..., a^*_{s^{N-1}}\}$ now forms an optimal policy π^* (Taylor, p.46, 2012). Note that (9) is only set for $s = s^N$ since we dont have a *value* in the terminal state s^N.

This method is known as the *backward induction-algorithm*. It belongs to the area of *Dynamical Programming*, which itself is a sub-area of optimization-theory. Different types of backward induction-algorithms similiar to this is encountered in undergraduate courses like finance mathematics - the *Rubinstein*

[3]Due to Richard E. Bellman, whom also laid the foundation of MDP's.

binomial model for pricing options or computer science - *depth first search*-algorithms. To get a better understanding of how this works, consider the following example:

Example 3:
Let us work through a classical MDP-problem using (9). Consider a value-grid like the one in figure 3, with the rewards to the right. For each step we take, we get a reward of -5, and the process ends whenever we reach $a4$ or $b4$. The possible actions a are "take a step in direction..." - *"North", "East", "South", "West"*, denoted $\{N, E, S, W\} \in \mathcal{A}$. We let the the states $\{a : c, 1 : 4\}$ span the grid, so for example the upper rightmost square is state "a4".

	1	2	3	4
a				$+100$
b				-100
c	*start*			

	1	2	3	4
a	-5	-5	-5	
b	-5		-5	
c	-5	-5	-5	-5

Figure 3: The starting value- and reward-grids, left and right respectively.

That is, the system is completely defined by,

$$\begin{aligned}
&\mathcal{S} = \{a1 : c4\},\\
&\mathcal{A} = \{N, E, S, W\},\ \textit{where } N := \textit{"go North"}, ..., W := \textit{"go West"},\\
&\gamma = 0.9\\
&r_{ss'} = \begin{cases} -5 & \textit{for} \quad s = \{a1 : c3, c4\},\\ 100 & \textit{for} \quad s = \{a4\},\\ -100 & \textit{for} \quad s = \{b4\}, \end{cases}
\end{aligned}$$

and let $P^a_{ss^a} = 0.7$, where s^a denotes the next state from s in direction a, and $P^a_{ss'} = 0.1$ for the rest of the feasible directions under action a. For example, say we stand in $c3$ and choose $a = N$, the probability of moving in direction N is 0.7 and 0.1 for the rest of the directions, i.e. $P^N_{c3,b3} = 0.7$ and $P^N_{c3,c2} = P^N_{c3,c4} = P^N_{c3,c3} = 0.1$. Whenever we end up trying to transition outside the grid we bounce back to the same position and receive the corresponding reward, that is why $P^N_{c3,c3} = 0.1$ and $P^S_{c3,c3} = 0.7$. Note that the rewards only depends on what state the process is taken to, i.e. $r^N_{ss'} = ... = r^W_{ss'} = r_{ss'}$, and that $P^\pi(s, a)$ only takes on values of 0 and 1.

Figure 4-5 in Appendix A shows the result after the first and second update, and Figure 6 after the first epoch. Figure 7 shows the optimal policy after the value-function has converged.

Now each such policy transforms an MDP into an ordinary discrete time MC as discussed earlier, and we can use the theory from the last section to analyze hitting times and stationary distributions etc. for comparing properties of different policies. Though this becomes limited very fast, since we have 4^9 different policies for this small example, it can be convenient for comparing just a few of these, and give a link back to Markov chains.

For the optimal policy of Figure 7, we have the following equation system to solve for the absorption probability of entering $C = a4$, while starting in $c1$:

$$
\begin{aligned}
h_{c1} &= 0.7h_{b1} + 0.1h_{c2} + 0.2h_{c1} \\
h_{c2} &= 0.7h_{c1} + 0.2h_{c2} + 0.1h_{c3} \\
h_{c3} &= 0.7h_{c2} + 0.1h_{b3} + 0.1h_{c3} + 0.1h_{c4} \\
h_{c4} &= 0.7h_{c3} + 0.2h_{c4} + 0.1h_{b4} \\
h_{b1} &= 0.7h_{a1} + 0.1h_{c1} + 0.2h_{b1} \\
h_{b3} &= 0.7h_{a3} + 0.1h_{b3} + 0.1h_{b4} + 0.1h_{c3} \\
h_{b4} &= 0 \\
h_{a1} &= 0.7h_{a2} + 0.1h_{b1} + 0.2h_{a2} \\
h_{a2} &= 0.7h_{a3} + 0.1h_{a1} + 0.2h_{a2} \\
h_{a3} &= 0.7h_{a4} + 0.1h_{a3} + 0.1h_{a2} + 0.1h_{b3} \\
h_{a4} &= 1
\end{aligned}
$$

However, we fail at solving this using standard linear equation system-methods. Using the method of writing this as a matrix in canonical form, inspired by (Björkström), we get the following absorption probabilities for absorption in $b4, a4$ respectively from each state,

	b4	a4
c1	0.25	0.75
b1	0.25	0.75
a1	0.25	0.75
c2	0.25	0.75
a2	0.05	0.95
c3	0.25	0.75
b3	0.16	0.84
a3	0.02	0.98
c4	0.34	0.66

which seem like a reasonable result.

Reinforcement Learning

This section follows the exposition of (Mitchell, 1997), with ideas from (Sutton and Barto, 1998).

First we are going to discuss what reinforcement learning is in a general and intuitive sense, then we will go into the theoretical parts.

Reinforcement learning is a way of combining dynamical programming with simulations. By letting an *agent* interact in an artificial *nature* it can learn a behavior by receiving *rewards* from a *teacher*. There are a few different algorithms developed for reinforcement learning, and we will keep focus on one of the most well-known: Q-learning.

Q-learning was introduced by (Watkins, 1989). Watkins introduces Q-learning by drawing parallels to how animals learn by getting rewards for desired behavior. The essence of reinforcement learning are these rewards. As an example, assume that we want to learn a game-strategy. We set up the nature of this particular game and the let the agent play. If the agent performs a win, we reward it by 1 and 0 otherwise. If a win has occured, the question of what stages has had significant impact on the outcome arises.

The basics of RL

RL is based upon MDP-theory, with the main difference being the backward induction algorithm we have used solves an MDP-problem directly using the known nature, i.e. $\mathcal{S}$, $\mathcal{A}$, P_{ij}^a and r_{ij}^a. RL on the other hand is simulated, that is, the agent *explores* the nature by actually testing an action in any given state, and observes what happens. *After* each such event it adapts to whatever nature has responded back to the agent with, in terms of this reward and the *next state*. This means that RL can learn optimal policies in which the nature is unknown, simply by interacting in it.

The following part assumes deterministic actions, for the stochastic case we will have to make minor changes. This will be done in the last part of this section.

Learning the optimal policy

So how does the agent learn the policy π^*? From the last section we know that maximizing the value-function will produce an optimal policy. Though the value-function only keeps track of the value of being in a state, it does not consider what actions to choose in each state. We define the optimal action a^* in state s as the one that maximizes the immediate reward $r_{ss'}^a$ and the value

of $V(\delta(s,a))$ for some mapping $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. That is,

$$\pi^*(s) = \arg\max_a [r^a_{ss'} + \gamma V^*(\delta(s,a))], \tag{11}$$

where γ is the mentioned discount factor. We will devote a subsection for discussing $\delta(s,a)$ later, for now we just need to know it outputs the *next state* s' *with the highest value*, and this value is precisely $V^*(\delta(s,a))$.

To be able to store the actions chosen in each state, we introduce the *Q-function*. Now we want the algorithm not only to store the *action* that maximizes (9), but to store the value for each action. Implying we need a storage for both the state and the action at each timestep. This is commonly called the Q-function,

$$Q(s,a) = r^a_{ss'} + \gamma V^*(\delta(s,a)). \tag{12}$$

Notice how (11) and (12) can be combined as

$$\pi^*(s) = \arg\max_a Q(s,a)$$

which yields that taking a^* in each state will provide an optimal policy. This means that the value of choosing action a under state s incorporates the future values of states.

This sounds promising but we need to know more how the actual updating occurs, and perhaps more important - how can we be assured that this updating of the state-action-values converges to the true ones of the nature? The maximized value function of state s is the same as choosing the action a' that maximizes the Q-function in state s', that is

$$V^*(s) = \max_{a'} Q(s',a')$$

and we can rewrite (11) as

$$Q(s,a) = r^a_{ss'} + \gamma \max_{a'} Q(s',a')$$

and we want to emphasize that s' is produced from the function $\delta(s,a)$, i.e. Q is recursive. Furthermore, the basis of this definition is the Bellman-equation, and has the optimality properties we discussed in the last section.

In practice $Q(s,a)$ is set up as lookup-table, i.e a storage of the Q-values, which updates at each time-step. One immediate complication of this is that in real-world-problems this method tend to grow too large to handle with the basic approach included in this thesis, we discuss this issue in the section *"the problem of dimensionality"*.

Say we define $\hat{Q}(s,a)$ to be the estimate of the true underlying value of $Q(s,a)$ for each s,a. In practice the updating follows,

$$\hat{Q}(s,a) \leftarrow r^a_{ss'} + \gamma \max_{a'} \hat{Q}(s',a'), \tag{13}$$

i.e the estimate is updated using an estimate. That is, (12) is a form of pseudo-*bootstraping*, see (Sutton and Barto, 1998).

The greedy policy

Now if the agent always chooses the action that maximizes Q, we will often end up at a local optima, since if a given path is better than the starting values of Q, this path will be built upon in the next epochs.

To conquer this problem we make our agent choose the present optimal action with some probability ϵ, and with probability $1 - \epsilon$ choose an *exploring* policy. That is, it chooses suboptimal actions deliberately to find unexploited policies. We call this the *exploration vs. exploitation-tradeoff*, since exploring will decrease the value at first, but *might* increase the value in the long run. As the number of updates goes to infinity, this insures that each state-action pair is visited an infinite number of times, which is one of the restrictions for $\hat{Q}(s, a)$ to converge.

On the other hand, as the number of updates grows large less exploration is needed, since more of the nature is known, and exploitation becomes more important than exploring. Using a increasing function $\epsilon = f(k)$ for updates $k = 1, 2, 3...$ handles this by increasing the probability of exploiting as the number of updates increase.

Furthermore, notice the connection between having a ϵ-greedy policy and $P^{\pi}(s, a)$ from the MDP-section. We have not used $P^{\pi}(s, a)$ in the RL-section, but these are essentially the same thing. The difference is merely in the way they are used. The ϵ is used as to inforce exploration in simulations. This can obviously not be done using the backward induction-algorithm, since we solve it using the *complete* nature, i.e. there is no point in exploring. Instead we use $P^{\pi}(s, a)$ in MDP's if the nature of the problem demands it. Assume for example that the problem is such that it has a number of actions to choose, but we must restrict some of these action to be chosen only some percentage of the time.

The delta-function

We have devoted this section to discussing the mentioned function $\delta(s, a)$, since we wanted to have some more knowledge of the ϵ-greedy function before doing so. How this function is chosen while programming an algorithm like this actually separates Q-learning from a variation of Q-learning, called SARSA.

In the basic Q-learning, $\delta(s, a)$ is constructed as to search the value of all subsequent states of s, and output the state with the highest value. That is,

$$\delta(s, a) = \arg\max_{s} V(s_i) = s',$$

for all subesequent states s_i. This means that although we can construct Q-learning as to follow an ϵ-greedy policy, the *next state optimal decision* at s' when updating for s will follow a greedy policy in Q-learning.

On the other hand, with SARSA we construct the algorithm as to actually choose an action in s following a given policy and then transition to the next state s'. Only *after* doing this will it send the value of s' back to s and update

$V(s)$. That is, we use a delayed updating rule. We can also alter the number of steps we want this to be delayed, which is then called *TD-learning* for *Temporal Difference*, i.e. we can have it simulate multiple steps from s and then send back the values observed from $s', s'', ...$

So what is the difference? Remember that each action has a probability distribution. While Q-learning only look up the values of the possible states s', SARSA actually explores, and therefore learns, the distribution of a in s' both when in state s and s'. Since the transition is simulated we can use an ϵ-greedy policy for the decision in s' while updating the value of s. Using an ϵ-greedy decision for choosing s' would not make much sense in Q-learning, since we *could* end up taking the value from some s'_i and then transition to another state s'_j, i.e. no exploration, just varying the state where we get the value of s' in s.

The problem of dimensionality

As noted earlier, the actual Q-values of each pair is stored in a lookup-table. For real-life applications this tend to grow big rapidly, to the point where it infeasible to use these tables. This is overcome by, instead of storing each value, approximations of the Q-function is used. The area of function approximation is outside the scope of this thesis, and we have deliberately avoided situations where it is needed. But since a lot of the literature includes this, we've been exposed to some of the tools used, which are usually various forms of neural networks. Essentially what this does is to cluster states which seem to have some properties in common. Treating each such cluster as one state, the state-action-space is shrunk to a hopefully manageable size. This is an obvious next subject to look into, to learn this theory further.

The stochastic case

In the last section we showed how to come up with a Q-function and how it is related to the Bellman-equation. We also assumed we had a deterministic system, i.e the nature takes the agent to the same state each time an action is chosen, the reward-function is stationary etc. The problem with implementing stochastic elements into a updating rule like (12) is that each element in the lookup-table takes a *new* value each update. Having stochastic effects on the actions taken could cause these values to diverge. To counter this, we introduce a step-size parameter θ which instead of changing the values of each $Q(s,a)$ adjusts them in the desired direction. Now arguing similarly as we did in (6) it can be shown that (Mitchell, p.381, 1997),

$$\hat{Q}(s,a) \leftarrow (1-\theta)\hat{Q}(s,a) + \theta\left[r^a_{ss'} + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)\right], \qquad (14)$$

in the stochastic case. The parameter α is usually chosen as to decrease the impact of each update, as the number of updates increases. An example is

given in (Mithchell, p.382, 1997),

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}.$$

Updating rule (14) is a tuple $\langle s, a, r, s', a' \rangle$, hence the name SARSA. (Watkins and Dayan ,1992) shows that $\hat{Q}(s,a)$ converges to $Q(s,a)$ with probability 1, given some conditions on α. Updating rule (14) is one of the most basic Reinforcement learning algorithms, and as noted by (Watkins and Dayan, 1992), "Q-learning is a primitive form of learning, but, as such, it can operate as the basis of far more sophisticated devices".

As a last example, we will go through the actual updating in SARSA. We have also spent time experimenting with implementations for more involved scenarios, none of which we are *certain* of having found an optimal policy. Of that reason, we think this thesis will benefit more from seeing this being done manually for a few time-steps in a very basic example as follows.

Example 4:
Consider Example 2 again, although this time we assume it will not live for 2 time-steps, but continue to run until we have found convergence. We will here go through one step of the updating process, and make assumptions regarding the outcome of each random variable. Let the parameters be,

$$\begin{aligned} \theta &= 0.9 \\ \gamma &= 0.9 \\ \epsilon &= 0.9. \end{aligned}$$

We use random numbers to generate the starting lookup-table Q(s,a), which becomes,

Q(s,a):		action α_i	action β_i
	state 1	0.67	0.23
	state 2	0.19	0.44

We remind ourselves of the transition probabilities and rewards,

$$\begin{aligned} \alpha_1 &: \ P^{\alpha_1}_{1,2} = 0.8, \quad P^{\alpha_1}_{1,1} = 0.2 \quad r_1 = 1 \\ \beta_1 &: \ P^{\beta_1}_{1,2} = 0.5, \quad P^{\beta_1}_{1,1} = 0.5 \quad r_2 = 0 \\ \alpha_2 &: \ P^{\alpha_2}_{2,1} = 0.5, \quad P^{\alpha_2}_{2,2} = 0.5 \\ \beta_2 &: \ P^{\beta_2}_{2,1} = 0.9, \quad P^{\beta_2}_{2,2} = 0.1 \end{aligned}$$

1. Now starting in state 1, we see that $\arg\max_a Q(1,a) = \alpha$. Meaning we choose α with probability $\epsilon = 0.9$. Assume *exploitative* action α is chosen, i.e. $Q(s,a) = Q(1,\alpha) = 0.67$.

2. We look at the transition probabilities for α_1. Assume the outcome of the transition is state 2. The reward for arriving at state 2, is $r_2 = 0$.

3. Before actually updating the first step, we need *the next state*-transition. The agent is now in state 2, where $\arg\max_a(Q, 2) = \beta$. Assume that the *exploring* action α is chosen, i.e. $Q(s', a') = Q(2, \alpha) = 0.19$.

4. Now we update using (14),

$$Q(1, \alpha) \leftarrow (1 - \theta)Q(1, \alpha) + \theta[0 + \gamma Q(2, \alpha) - Q(1, \alpha)] = -0.38.$$

The updated lookup-table becomes,

$Q_1(s, a) =$

	action α_i	action β_i
state 1	-0.38	0.23
state 2	0.19	0.44

and this keep iterating until convergence.

Discussion

Our interest in MDP's started with self-learning algorithms, neural networks and the hype of deep-learning. Pretty soon it was apparent that we needed to step backwards theory-wise, to see where RL came from, in order to understand to the more advanced algorithms. That was how we found MDP's. To understand more of the theoretical parts of MDP's we need to study optimization theory, since a lot of the applications we've encountered is in *planning*. In order to be able to understand RL better, we need more computer science or perhaps more specifically statistical machine learning, and we also need to work a lot more with programming to be able to implement these algorithms. With that being said, the theory included in this thesis is the theory we found we could handle fairly well.

Our aim with writing this thesis, has been to introduce ourselves to the subject, which we think we have succeeded with. Furthermore, we wanted to do a thesis involving statistical programming. Not because that is a skill we master, but the direct opposite.

What we could have done differently, is to realize at an early stage that this thesis should be about MDP's and the link to Markov theory, instead of spending time reading about RL and the algorithmical variations of it. Also, we should have put focus on sources that built this theory from a mathematical point of view such as Ross (1970) and Taylor (2012) early on, instead of a computer science-perspective such as Sutton and Barto (1998). Since this is a subject we had no prior exposure to, we feel that we have not had time to get both an overview and depth. The next time we will probably try to do something which we have been more exposed to, and try to make it more focused on some more narrow aspect of the material.

Reference list

Ross, S.M. (2010) *Introduction to Probability Models.* 10th edn. Academic Press

Taylor, J (2012) *Markov decision processes:Lecture notes for STP 425.* Arizona State University

Sutton, R.S. and Barto, A.G. (1998) *Reinforcement Learning: An introduction.* Cambridge: MIT Press

Ross, S.M. (1968) *Arbitrary state markovian decision processes.* Stanford University

Bellman, R.E. (1954), *The theory of dynamical programming.* Rand Corporation

Björkström, A. () *Några kommentarer till avsnitt 4.6 i Ross: Introduction to probability models (7th ed).* Stockholm University

Mitchell, T.M. (1997) *Machine Learning.* McGraw-Hill

Watkins, C.J.C.H (1989) *Learning from delayed rewards.* Kings College

Watkins, C.J.C.H and Dayan, P. (1992) *Technical note: Q-learning.* Boston: Academic Publishers

Ross, S.M. (1970) *Applied probability models with optimization applications.* New York: Dover Publications

Appendix

A: The grid-problem

	1	2	3	4
a			68.5	+100
b				−100
c	*start*			

	1	2	3	4
a			→	
b				
c				

Figure 4: The value- and policy-grid after 1 update.

	1	2	3	4
a	25.06	42.95	68.5	+100
b	12.54		33.45	−100
c	*start*			

	1	2	3	4
a	→	→	→	
b	↑		↑	
c				

Figure 5: The value- and policy-grid after 5 updates.

	1	2	3	4
a	25.06	42.95	68.5	+100
b	12.54		33.45	−100
c	3.78	11.21	21.91	0.84

	1	2	3	4
a	→	→	→	
b	↑		↑	
c	↑	→	↑	←

Figure 6: The Value- and policy-grids from the first epoch.

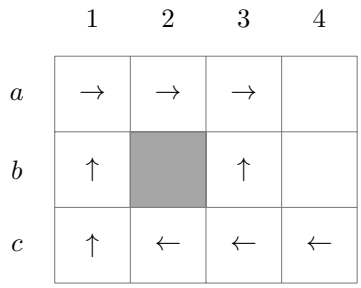

Figure 7: The optimal policy π^*.