UiO : Department of Mathematics
University of Oslo

Risk in stochastic control and reinforcement learning

Johannes Vincent Meo

Master's Thesis, Spring 2022

This master's thesis is submitted under the master's programme *Mathematics*, with programme option *Mathematics for applications*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

This thesis dives into the theory of discrete time stochastic optimal control through exploring dynamic programming and reinforcement learning. The main goal of this thesis is to closely investigate risk-sensitive control, and to look into some of the methods used in dynamic programming and reinforcement learning in order to find risk-sensitive policies. We give a comparison of the different risk-sensitive methods considered in this thesis and provide results that, under some assumptions, guarantee that we are able to find risk-sensitive policies for a class of optimal control problems.

Acknowledgements

First of all, I would like to thank my supervisors, Christian Agrell and Kristina Rognlien Dahl. Thank you Kristina for giving me an interesting project and for the guidance you provided the first months of my work on this thesis before you passed on the torch to Christian, which I would like to thank for all the help, feedback and supervision you have given me this past year. I would also like to thank Nikolai Thode Opdan for reading through this thesis and providing helpful comments. A thanks is also in need to my friends and my fellow students sitting at the eleventh floor in NHA that have all contributed to making my time at Blindern the last years a pleasant experience. A final thanks goes to my family, Filip, Margrethe and Axel, for their love and support, and to Julia for always being by my side.

Contents

Abstract **i**

Acknowledgements **ii**

Contents **iii**

List of Figures **v**

List of Tables **vii**

Listings **viii**

Symbols and notation **ix**

1 Introduction **1**
- 1.1 Structure of the thesis 2
- 1.2 Our contributions 3

2 Preliminaries **7**
- 2.1 Basic probability theory 7
- 2.2 Some real analysis 10

3 Introduction to dynamic programming **12**
- 3.1 What is a dynamic programming problem? 12
- 3.2 The dynamic programming algorithm 14
- 3.3 State augmentation and other reformulations 17
- 3.4 Deterministic systems and the shortest path problem 22
- 3.5 Stochastic systems with imperfect state information 29

4 Abstract dynamic programming **39**
- 4.1 The abstract dynamic programming model 39
- 4.2 Consequences of monotonicity and contracion assumptions . . 43
- 4.3 Finding policies 51

5 Infinite horizon dynamic programming **57**
- 5.1 Stochastic shortest path problems 57
- 5.2 Discounted problems 68

6 Reinforcement learning **74**
6.1 A short introduction to finite MDPs and reinforcement learning 75
6.2 Some reinforcement learning algorithms 77
6.3 Similarities between reinforcement learning and dynamic programming 78

7 Risk-sensitive control **86**
7.1 Motivation for risk-sensitive control 87
7.2 Model-based risk-sensitive control 87
7.3 Model-free risk-sensitive control 113
7.4 Comparison of risk-sensitive control methods 131

8 Concluding remarks **135**

Appendices **137**

A Code **138**

Bibliography **164**

List of Figures

3.1 A graph representation of a deterministic finite-state system 24
3.2 The flipped deterministic finite-state system 25
3.3 The two candidates for shortest path 27
3.4 Possible travelling routes 28
3.5 Shortest travelling routes 29

4.1 Graph of the shortest path problem 52

5.1 A simple SSP. Same coloured paths originating at the same node depict the possible transitions from the given node under a specific control. The black lines depict the transitions for control u_1, while orange shows the same for control u_2. Node C is introduced for technical reasons only. 65

6.1 The evolution of Q-values and running average of simulated costs using Q-learning, as well as theoretical values and mean of sampled values. 80
6.2 The evolution of Q-values and running average of simulated costs using SARSA, as well as theoretical values and mean of sampled values. 81
6.3 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for Q-learning. The height has been cut at $y = 250$. 82
6.4 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for SARSA. The height has been cut at $y = 250$. 83
6.5 Bar plot showing distribution of policies created with Q-learning and SARSA. Legend notation explanation: A1B1C1 denotes the policy taking action 1 in each state. 84

7.1 Illustrating the effect of β when using exponential utility function. 95
7.2 Illustrating the effect of k when using exponential utility function. 96
7.3 Value of target function for the policies considerd in Example 7.2.17. The black line shows where the target function for policy μ_1 and μ_3 intersects. Code to produce figure can be found in Listing A.11. 110

7.4 An augmented version of the SSP in Figure 5.1. Same coloured paths originating at the same node depict the possible transitions from the given node under a specific control. The black lines depict the transitions for control u_1, while orange shows the same for control u_2. 111
7.5 The evolution of Q-values and running average of simulated costs using $\hat{Q}$-learning, as well as the mean of the sampled values. 119
7.6 The evolution of Q-values and running average of simulated costs when weighting TD-rewards using $\kappa = 0.5$, as well as the mean of the sampled values 120
7.7 The evolution of Q-values and running average of simulated costs when weighting TD-rewards using $\kappa = 0.15$, as well as the mean of the sampled values. 121
7.8 The evolution of Q-values and running average of simulated costs when weighting TD-rewards using $\kappa = -0.5$, as well as the mean of the sampled values. 122
7.9 The evolution of Q-values and running average of simulated costs and risk using the error states method with a risk bound of $\omega = 0.1$ for the policy, as well as the mean of the sampled values. 123
7.10 The evolution of Q-values and running average of simulated costs and risk using the error states method with a risk bound of $\omega = 0.2$ for the policy, as well as the mean of the sampled values. 124
7.11 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for $\hat{Q}$-learning. The height has been cut at $y = 250$. 125
7.12 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies when weighting TD-rewards. The height has been cut at $y = 250$. 126
7.13 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies when weighting TD-rewards. The height has been cut at $y = 250$. 127
7.14 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies when weighting TD-rewards. The height has been cut at $y = 250$. 128
7.15 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for error states method. The height has been cut at $y = 250$. 129
7.16 Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for error states method. The height has been cut at $y = 250$. 130
7.17 Bar plot showing distribution of policies created with the different risk-sensitive RL methods introduced in this chapter with the parameter values we have considered in this example, as well as distribution of ξ values used to generate final policy for error states method. Legend notation explanation: A1B1C1 denotes the policy taking action 1 in each state. 131

List of Tables

5.1 Optimal value function and Q-values for SSP depicted in Figure 5.1 68

6.1 Q-values estimated by Q-learning and SARSA for SSP depicted in Figure 5.1. 83

7.1 Optimal value under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4. The $J^*(s)$ values for error state is the actual expected cost function values of the policy found to be optimal by the method. 112
7.2 Optimal policy under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4. 112
7.3 Expected discounted cost for the optimal policies found under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4. 112
7.4 Optimal policy under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4. 124

Listings

A.1 ssp_ex; Implementation of SSP considered in Example 5.1.9 . . 138
A.2 ex_SSP.json; JSON-file defining SSP implemented by Listing A.1 141
A.3 ssp_DP.py; Implementation of value iteration and policy iteration used to solve SSP problem in Example 5.1.9 142
A.4 ssp_rl_algs.py; Implementation of RL algorithms used to solve SSP problem in Example 6.3.1 143
A.5 ex_SSP_es.json; JSON-file defining SSP with error state implemented by Listing A.1 148
A.6 ssp_minimax.py; Code to run minimax to solve SSP problem in Example 7.2.18 150
A.7 ssp_exp_utility.py; Code to run VI and PI using exponential utility function to solve SSP problem in Example 7.2.18 151
A.8 ssp_error_states.py; Code to run VI and PI using error states notion of risk to solve SSP problem in Example 7.2.18 152
A.9 ssp_rs_rl.py; Implementation of risk-sensitive RL algorithms used to solve SSP problem in Example 7.3.5 154
A.10 exp_utility.py; Generating plots for Example 7.2.4 162
A.11 suboptimality_error_states.py; Generating plots for Example 7.2.17 163

Symbols and notation

Symbol	Meaning
ω	Scenario space.
$\mathcal{F}$	σ-algebra.
P	Probability measure.
X	Random variable.
$E[\cdot]$	Expected value.
α	Discount factor.
γ	Learning rate.
X	State space (it should be clear from the context whether X is a random variable or a state space).
U	Control/Action space.
$U(x)$	Set of controls feasible at state x.
D	Disturbance space.
Z	Observation space.
V	Observation disturbance space.
$\mathcal{R}$	Reward space.
Φ	Set of error states.
f	Function determining dynamics of control problem.
g	Cost/reward-function.
$p_{ij}(u)$	Probability of transitioning from state i to state j under control u.
π	A policy.
π^*	An optimal policy.
μ	A stationary policy.
$\mathcal{M}$	Set of stationary policies.
J_π	The cost/reward function of a policy π.
$J_{\pi,N}$	N-stage cost function of policy π.
J^*	The optimal cost/reward function.
G	Sum of future discounted rewards.
Q_π	Q-function of policy π.
$\mathcal{R}(X)$	Set of real-valued functions with a state space X as its domain.
$\mathcal{B}(X)$	Space of bounded real-valued functions with a state space X as its domain.
$H(x, u, J)$	Mapping giving cost/reward-to-go of applying control u in state x when J is the cost-to-go function.
T_μ	Bellman operator for the stationary policy μ.
T	Bellman operator.

CHAPTER 1

Introduction

The theory of stochastic optimal control is concerned with finding the optimal way to control a system in order to optimize some given objective. A rule that, given any state of the system, tells the controller which action to take is called a *policy*. Many different kinds of tasks and problems that are seemingly unrelated can be formulated as a stochastic optimal control problem. Some examples are a driver driving home, a portfolio manager trying to maximize the return for her clients, a robot trying to figure out how to walk or someone trying to beat Stockfish in chess. What all of these examples have in common is that there is some controller, either the driver, the portfolio manager, the robot or the chess player that needs to figure out what action to take in order to reach their goal.

Today, there exists several methods used to solve stochastic optimal control problems. One such method is dynamic programming that was developed in the 1950s by the mathematician Richard Bellman. The theory of dynamic programming centres around finding what is called the Bellman equation for the problem at hand and the solution of the equation. Dynamic programming is still considered one of the optimal ways to solve optimal control problems. However, in order to use the dynamic programming methods a lot of information about the underlying *dynamics* of the system needs to be known, or at least in some way estimated.

Another approach that copes with the strict assumptions of dynamic programming is reinforcement learning, which is considered to be one of the three main branches of machine learning. Reinforcement learning, as the name suggests, relies on *learning* in order to find policies. The learning in the reinforcement learning context often comes from trial-and-error which can be done in e.g., a simulation or in a real-world environment.

Traditionally, the goal in stochastic optimal control is to find the policy that most often results in the controller achieving the optimal value for his objective, e.g., a chess player would like to find the strategy that results in him winning the game. However, sometimes the optimal policy can come with unwanted side-effects. Take a portfolio manager, in order to maximize the possible return she should invest only in risky-assets as they, on average, result in the highest return. But, she would then be prone to a non-zero probability of the whole portfolio losing all of its value. Risk-sensitive optimal control tries to handle this problem by finding methods that allow us to discover policies that has a less risky behaviour, while still having acceptable performance.

In this thesis we consider stochastic optimal control by introducing the theory of dynamic programming and reinforcement learning. Towards the end

of the thesis we take a look at risk-sensitive optimal control and introduce different methods that approach the problem of finding risk-averse policies differently. We prove the existence of optimal policies for some of these methods by using abstract dynamic programming theory. Our introduction of theory is throughout this thesis complemented with several examples in order to give the reader a better understanding of the different topics.

1.1 Structure of the thesis

This thesis is structured in the following way:

In Chapter 2 we present some preliminary theory related to probability theory and real analysis. This theory is needed as a theoretical backbone in order to prove the results in the chapters that follow. The parts of this chapter that is most important for the rest of the thesis is the definition of a contraction, given in Definition 2.2.7, and Banach's fixed point theorem, which is given in Theorem 2.2.8.

Then, in Chapter 3 we give an introduction to dynamic programming for the case where we have a finite time horizon. The most important result from this chapter is Theorem 3.2.1, which gives conditions for when the dynamic programming algorithm manages to find an optimal policy. The remainder of the chapter looks at how different kinds of problems can be reformulated into the regular dynamic programming problem, and thus solved using the dynamic programming algorithm. The theory of this chapter is complemented with many self-made examples solving concrete problems.

We then proceed with Chapter 4, where we zoom out a bit from the concrete setting of Chapter 3 and take a look at the theory of abstract dynamic programming. Here, we introduce a more general framework that is more rigorous that what we presented in the preceding chapter. We introduce two important properties, namely monotonicity and contraction. These properties are sufficient for a model in order to guarantee some nice properties for the problem described by the model. We also present some general solution methods for dynamic programming problems that depends on the theory introduced in the chapter in order to guarantee convergence.

In the subsequent chapter, Chapter 5, we consider dynamic programming problems where we have an infinite time horizon. By using the abstract dynamic programming theory presented in Chapter 4 we are able to, under some given assumptions, provide new proofs for some results that guarantee convergence for the solution methods in the context of infinite horizon dynamic programming. We also include a numerical example in this chapter, illustrating that the solution methods in fact work.

The following chapter, Chapter 6, gives a brief introduction to reinforcement learning and Markov decision processes. We present the two classical tabular methods, Q-learning and SARSA. We test their performance with a numerical example where we try to use these methods to find the optimal policy for the same problem that we solved in the numerical example in Chapter 5. This chapter also contains a section where we try to explain that dynamic programming and reinforcement learning really just are two different names for the same thing, which is optimal control for sequential decision problems.

Chapter 7 is concerned with risk-sensitive optimal control, and is the highlight of this thesis. Here we introduce several methods that all try to solve the same problem: finding policies that are not blind to the underlying risk of the optimization problem we are trying to solve. Three of the methods we present are model-free, while the three other methods rely on a model of the underlying dynamics in order to find a solution. For the three model-based methods we prove that, given some assumptions, the general solution methods introduced in Chapter 4 also converge to an optimal solution for these methods.

This thesis ends with the concluding remarks in Chapter 8. Here we give a short summary of the work done in this thesis, as well as setting out some ideas for possible further work based on the theory covered in this thesis.

For several of the examples given in this thesis we have done computations and simulations numerically using Python. The code used is listed in Appendix A.

1.2 Our contributions

Here we give a brief overview of some of the contributions we have made in this thesis. The parts of the thesis that is listed below is marked with an † in the text. The theory and the methods presented in this thesis that is based on previous work by other authors are clearly marked and referenced, either at the beginning of the relevant chapter or section, or where the result or method is presented. The writings in this thesis is influenced by our understanding of the theory, and we take full responsibility for any mistakes that may be in this work.

Our main contributions are the following:

Example 2.1.8: We created this example with the intention of using the probability measures and the stochastic variable X in, among others, Example 3.2.2.

Example 3.2.2: This example is inspired by exercise 1.2 in [Ber17]. We have changed the cost function g, reduced N to 3 and use another distribution for w_k, which depend on the probability measures found in Example 2.1.8. We have also a more restrictive control space $U(x)$.

Example 3.3.1: The example considered here is inspired by exercise 3.3 in [Ber17]. We introduce forecasting in the problem considered in Example 3.2.2.

Example 3.4.1: Here we represent a deterministic version of the DP problem from Example 3.2.2 as a graph.

Example 3.4.2: In this example we demonstrate how to use the forward DP algorithm in order to solve the deterministic version of the DP problem considered in Example 3.2.2, which is depicted in the graph created in Example 3.4.1.

Example 3.4.3: We show how to convert a shortest path problem to a DP problem. The example is inspired by exercise 2.3 in [Ber17], but we have used a different graph.

Example 3.5.1: Here we consider a version of Example 3.3.1 where we have imperfect state information. The idea behind this example is inspired by exercise 4.10 in [Ber17], and the sufficient statistic used in the example is influenced by the proposed solution of exercise 4.10, which can be found in [Ber].

Example 4.1.5: This example illustrates the connections between the abstract DP theory introduced in Chapter 4 and what we considered in Chapter 3 by reformulating the DP problem from Example 3.2.2 using the abstract formulation.

Proof of Proposition 4.2.1 (v)**:** We write out a detailed proof of Proposition 4.2.1 (v), where we use the same techniques as the proof of Proposition 4.2.1 (iv) given in [Ber18].

Lemma 4.2.3: This lemma is proving a result that is necessary for the proof of Proposition 4.2.2, but which is kind of intuitive. This result is therefore claimed to hold in the proof of Proposition 4.2.2 presented in [Ber18], but as we are aiming to write out a more detailed proof of the result we introduce Lemma 4.2.3 and its proof.

Lemma 4.2.4: A lemma proving a result necessary for the proof of Proposition 4.2.2. The same result is claimed to hold in [Ber18] without a proof, but as we would like to provide some more details, we state it as a separate lemma and provide a proof.

Proof of Proposition 4.2.2: The proof presented in this thesis follows the proof from [Ber18] and a section looking at the optimality over nonstationary policies in the same book, but we have written out several details compared with the original proof.

Example 4.2.5: This example is taken from example 2.1.1 in [Ber18], but we have written it out in more detail.

Example 4.2.6: Here we illustrate the importance of the contraction property with an original example.

Example 4.3.2: Example showing how to find a lookahead policy on a graph similar to the one considered in Example 3.4.3.

Proof of Proposition 4.3.3: Here we present a more detailed proof than the original given in [Ber18].

Example 4.3.4: Here we use the value iteration algorithm to find the optimal policy for the shortest path problem associated with the graph used in Example 4.3.2.

Example 4.3.7: Find the optimal policy for the shortest path problem connected with the graph used in Example 4.3.4, but now using policy iteration.

Proposition 5.1.4 Prove that the model for an infinite horizon stochastic shortest path problem that we consider in Section 5.1 is monotone.

Proof of Proposition 5.1.5: We provide a new proof for Proposition 5.1.5, which is proven in [Ber19]. Our new proof utilises the abstract DP theory introduced in Chapter 4.

Proof of Proposition 5.1.6 We give an alternative proof of Proposition 5.1.6 by using the abstract DP theory. The result and its original proof is given in [Ber19].

Proof of Proposition 5.1.7: Here a proof of Proposition 5.1.7 that relies on the abstract DP theory is given. The last part of the proof, concerning convergence of the VI algorithm, follows the proof of the same result from [Ber19].

Proof of Proposition 5.1.8: An original proof of Proposition 5.1.8 that uses the abstract DP theory is given. The result itself is taken from [Ber19] where another, more direct, proof is provided.

Example 5.1.9: In this example we implement VI and PI, and use the algorithms to find the optimal value function and optimal policy for a SSP. We then confirm that the values found computationally are correct.

Proof of Proposition 5.2.1: Here we write out the proof of Proposition 5.2.1 in more detail than what is done in the original proof given in [Ber19].

Proposition 5.2.2: This result ensures that the infinite horizon discounted cost model indeed is monotone.

Example 6.3.1: This example tests the performance of SARSA and Q-learning on the SSP considered in Example 6.3.1.

Example 7.2.1: In this example we apply minimax to the St.Petersburg Paradox.

Proposition 7.2.2: Result showing that the minimax target function satisfy the monotonicity property.

Proposition 7.2.3: This result tells us that the minimax target function is a contraction.

Equation (7.8): In the preceding equations we showed the relationship between the exponential utility function, and the sum of the expected cost and the variance of the cost, by writing out the Taylor polynomial.

Example 7.2.4: This example show how the utilization of an exponential utility function change the behaviour of a controller for the St. Petersburg Paradox.

Proposition 7.2.5: Proposition showing that the target function for the exponential utility function is monotone.

Proposition 7.2.6: This result shows that the exponential utility function model we use is a contraction.

Corollary 7.2.7: Corollary giving guarantees for when we are able to find unique optimal risk-averse or risk-seeking policies.

Definition 7.2.12: Defining an operator used to finding the value function of a policy for the error states target function.

Proposition 7.2.13: Result guaranteeing that the operator defined in Definition 7.2.12 has a fixed point given some assumptions.

Proposition 7.2.14: This Proposition shows that the operator defined in Definition 7.2.12 has a unique fixed point for a broad range of problems.

Proposition 7.2.15: This result tells us that the mapping used in the operator defined in Definition 7.2.12 is monotone.

Example 7.2.16: In this example we use the error states method in order to find risk-sensitive policies for the St. Petersburg Paradox.

Example 7.2.17: This example illustrates how the error states method not necessarily always finds the true optimal policy.

Example 7.2.18: Here we use several model-based methods numerically to find risk-sensitive optimal policies for the SSP considered in Example 5.1.9.

Example 7.3.5: In this example we implement and test numerically several model-free methods by trying to estimate optimal risk-sensitive policies for the SSP considered in Example 5.1.9.

CHAPTER 2

Preliminaries

2.1 Basic probability theory

In this section we introduce some basic probability theory. This theory is necessary for understanding dynamic programming and stochastic control, which is introduced in later chapters. For a more comprehensive book on the topic, readers are encouraged to read [Wal12] or [Øks03]. For more on measure theory, and measures in general, [Lin17] has a good introduction to the topic, while [Fol99] is a great book on the same subject.

A probability space is a triple $(\Omega, \mathcal{F}, P)$. We can think of Ω as a set that consists of every possible event in our probability space, and is often called the scenario space. Ω can in general be finite, countable or uncountable. $\mathcal{F}$ is a σ-algebra, which is a family of subsets of Ω, while P is a probability measure. We now take a look at the formal definition of the components of a probability space.

Definition 2.1.1 (σ-algebra)**.** Let Ω be a set. A σ-algebra, $\mathcal{F}$, is a family of subsets of Ω that satisfies

i) $\emptyset \in \mathcal{F}$.

ii) $A \in \mathcal{F} \implies A^c \in \mathcal{F}$.

iii) $A_1, A_2, \ldots \in \mathcal{F} \implies \bigcup_{i\in\mathbb{N}} A_i \in \mathcal{F}$.

We say that a set A is *measurable* if $A \in \mathcal{F}$. For the remaining of this chapter, we let Ω denote a set, and $\mathcal{F}$ be a σ-algebra consisting of subsets of Ω. A tuple $(\Omega, \mathcal{F})$ is then called a *measurable space*. The only thing that remains for the measurable space to become a probability space is a probability measure defined on the σ-algebra.

Definition 2.1.2 (Probability measure)**.** Let $(\Omega, \mathcal{F})$ be a measurable space. A function $P\colon \mathcal{F} \to [0, 1]$ is called a probability measure on the measure space if it satisfies that

i) $P(A) \geq 0$ for all $A \in \mathcal{F}$.

ii) $P(\Omega) = 1$.

iii) If $A_1, A_2, \ldots \in \mathcal{F}$ are pairwise disjoint, i.e., $A_i \cap A_j = \emptyset$ when $i \neq j$, then

$$P\left(\bigcup_{i\in\mathbb{N}} A_i\right) = \sum_{i\in\mathbb{N}} P(A_i).$$

Let P be a probability measure on the measurable space $(\Omega, \mathcal{F})$. We can then go on to define a (real valued) random variable on the probability space $(\Omega, \mathcal{F}, P)$.

Definition 2.1.3 (Random variable)**.** Let $(\Omega, \mathcal{F}, P)$ be a probability space, and $(E, \mathcal{E})$ a measurable space. A function $X\colon \Omega \to E$ is called a random variable if

$$X^{-1}(A) = \{\omega \in \Omega \colon X(\omega) \in A\} \in \mathcal{F} \text{ for all } A \in \mathcal{E}.$$

If we let $E = \mathbb{R}$, and let $\mathcal{E}$ be the Borel σ-algebra on $\mathbb{R}$ (the σ-algebra generated by all open sets in $\mathbb{R}$), then we call X a *real* random variable.

In other words, a (real valued) random variable is a function on the probability space that has $\mathbb{R}$ as its codomain, where the preimage of any interval $(-\infty, x]$ is measurable, and thus in $\mathcal{F}$. We now look at how we define the expectation of a random variable with respect to a measure P.

Definition 2.1.4 (Expectation)**.** Let $(\Omega, \mathcal{F}, P)$ be a probability space and let X be a random variable on this space. The expectation $E[X]$ of X is then

$$E[X] = \int_\Omega X dP,$$

which is the Lebesgue integral (see e.g. [Lin17]) over Ω with regard to the probability measure P.

The last bit of theory we need to consider is the definition of a distribution. The following definition is inspired by [Øks03].

Definition 2.1.5 (Distribution)**.** Let $(\Omega, \mathcal{F}, P)$ be a probability space and let $X\colon \Omega \to \mathbb{R}$ be a (real valued) random variable on this probability space. The random variable then induces a probability measure denoted μ_X on its codomain $\mathbb{R}$ called the distribution of X. The distribution of X is defined as

$$\mu_X(A) = P(X \in A) = P(\{\omega \in \Omega \colon X(\omega) \in A\}), \text{ for all } A \subset \mathbb{R}.$$

That is, the distribution μ_X is a way to measure the probability of subsets $A \subset \mathbb{R}$, even though our original measure P is only defined on sets $\tilde{A} \in \mathcal{F}$, where $\tilde{A} \subset \Omega$.

Definition 2.1.6 (Conditional probability)**.** Let $(\Omega, \mathcal{F}, P)$ be a probability space, and assume that $A, B \in \mathcal{F}$ with $P(B) > 0$. The conditional probability of A given B is then defined as

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}.$$

The following well known theorem is taken from [Wal12].

Theorem 2.1.7 (Bayes theorem, [Wal12]). *Let Ω be a scenario space and $A_1, \ldots, A_n \subset \Omega$ be pairwise disjoint and measurable sets with $\bigcup_{i=1}^{n} A_i = \Omega$ and $P(A_i) > 0$, $i = 1, \ldots, n$. We then, for any $B \subset \Omega$ with $P(B) > 0$, have*

$$P(A_k|B) = \frac{P(B|A_k)P(A_k)}{\sum_{i=0}^{n} P(B|A_i)P(A_i)},\ k = 1, \ldots, n.$$

Example 2.1.8 (Probability theory). †

Consider the following example: We have a scenario space $\Omega = \{\omega_1, \omega_2, \omega_3\}$. A σ-algebra $\mathcal{F}$ on Ω is then

$$\mathcal{F} = \{\Omega, \{\omega_1, \omega_2\}, \{\omega_1, \omega_3\}, \{\omega_2, \omega_3\}, \{\omega_1\}, \{\omega_2\}, \{\omega_3\}, \emptyset\},$$

which is called the discreet σ-algebra, since it contains every possible subset of the scenario space Ω. Another σ-algebra that contains less information is e.g. the trivial σ-algebra $\mathcal{G} = \{\Omega, \emptyset\}$. $\mathcal{G}$ is in fact the smallest possible σ-algebra of subsets of Ω. We have that $(\Omega, \mathcal{F})$ is a measurable space. We can then go on to define some probability measures (Definition 2.1.2) on this measurable space. Consider the examples below:

- $P(\{\omega_1\}) = \frac{1}{3}$, $P(\{\omega_2\}) = \frac{1}{3}$, $P(\{\omega_3\}) = \frac{1}{3}$,
- $Q(\{\omega_1\}) = 0$, $Q(\{\omega_2\}) = 1$, $Q(\{\omega_3\}) = 0$,
- $R(\{\omega_1\}) = \frac{1}{2}$, $R(\{\omega_2\}) = \frac{1}{2}$, $R(\{\omega_3\}) = 0$,

We see that P, Q and R all satisfy the conditions for being a probability measure (Definition 2.1.2). Thus, $(\Omega, \mathcal{F}, P), (\Omega, \mathcal{F}, Q)$ and $(\Omega, \mathcal{F}, R)$ are valid probability spaces. We can then define a random variable (Definition 2.1.3) X on these spaces. Let

$$X(\omega_1) = -1,\ X(\omega_2) = 0,\ X(\omega_3) = 1.$$

Notice that X is a well defined random variable on all of our probability spaces. However, let us now calculate the expected value of X on the different spaces. Let $E_\mu[X]$ denote the expectation of a random variable X with regard to a probability measure μ. We then have that

$$E_P[X] = \int_\Omega X dP = \sum_{i=1}^{3} X(\omega_i)P(\omega_i) = -1\frac{1}{3} + 0\frac{1}{3} + 1\frac{1}{3} = 0,$$

$$E_Q[X] = \int_\Omega X dQ = \sum_{i=1}^{3} Q(\omega_i)P(\omega_i) = -1 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 = 0,$$

$$E_R[X] = \int_\Omega X dR = \sum_{i=1}^{3} R(\omega_i)P(\omega_i) = -1\frac{1}{2} + 0\frac{1}{2} + 1 \cdot 0 = -\frac{1}{2}.$$

Note that even though we calculated the expected value of the same random variable, we did not get the same result for the different probability measures.

2.2 Some real analysis

This chapter gives a brief introduction to some concepts from real analysis that are relevant for the topics of this thesis. The introduction in this chapter is based on the book [Lin17]. We start by defining a metric space and then proceed with defining the notion of a normed space. We begin by defining a metric.

Definition 2.2.1 (Metric [Lin17])**.** A metric on a non-empty set V is a function $d : V \times V \to \mathbb{R}$ such that:

(i) For all $\mathbf{u}, \mathbf{v} \in V$ with $\mathbf{u} \neq \mathbf{v}$, we have $d(\mathbf{u}, \mathbf{v}) > 0$.

(ii) If $\mathbf{u} = \mathbf{v}$ we have $d(\mathbf{u}, \mathbf{v}) = 0$.

(iii) We have $d(\mathbf{u}, \mathbf{v}) = d(\mathbf{v}, \mathbf{u})$ for all $\mathbf{u}, \mathbf{v} \in V$.

(iv) For arbitrary $\mathbf{u}, \mathbf{v}, \mathbf{w} \in V$ the inequality $d(\mathbf{u}, \mathbf{v}) \leq d(\mathbf{u}, \mathbf{w}) + d(\mathbf{w}, \mathbf{v})$ holds true.

We are then ready to define a metric space.

Definition 2.2.2 (Metric space [Lin17])**.** Let X be a non-empty set. A metric space is then a tuple (X, d) where d is a metric on X.

Then we proceed with defining a normed space.

Definition 2.2.3 (A norm and normed space [Lin17])**.** Let V be a vector space over $\mathbb{R}$. A function $\|\cdot\| : V \to \mathbb{R}$ is then called a norm on V if

(i) $\|\mathbf{u}\| \geq 0$ with equality if and only if $\mathbf{u} = 0$.

(ii) $\|\alpha\mathbf{u}\| = |\alpha|\|\mathbf{u}\|$ for all $\alpha \in \mathbb{R}$ and all $\mathbf{u} \in V$.

(iii) $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$ for all $\mathbf{u}, \mathbf{v} \in V$.

We then call the pair $(V, \|\cdot\|)$ a normed space.

The next proposition ensures that a normed space always has a metric induced by its norm.

Proposition 2.2.4 (Norm implies metric [Lin17])**.** *If* $(V, \|\cdot\|)$ *is a real normed space, then the function*

$$d(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\| \tag{2.1}$$

is a metric on V*.*

We then want to introduce the well known Banach's Fixed Point Theorem, but we then need the idea of complete spaces. To introducing this concept we need the following definition.

Definition 2.2.5 (Cauchy sequence [Lin17])**.** Let (X, d) be a metric space and let $\{x_n\}$ be a sequence in this metric space. The sequence is then called a Cauchy sequence if for each $\epsilon > 0$ there is an $N \in \mathbb{N}$ such that $d(x_n, x_m) < \epsilon$ for all $n, m \geq \mathbb{N}$.

Then we define the notion of complete metric spaces.

Definition 2.2.6 (Complete metric space [Lin17]). We say that a metric space (X, d) is complete if all Cauchy sequences converge.

The last definition we need before introducing Banach's Fixed Point Theorem is the one of contractions. We will later in Chapter 4 see that contractions are essential for the theory in this thesis.

Definition 2.2.7 (Contraction [Lin17]). Let (X, d) be a metric space. We then say that a function $f : X \to X$ is a contraction if there exist some contraction factor $\alpha \in (0, 1)$ such that

$$d(f(x), f(y)) \leq \alpha d(x, y) \text{ for all } x, y \in X.$$

We are then ready to introduce Banach's Fixed Point Theorem.

Theorem 2.2.8 (Banach's Fixed Point Theorem [Lin17]). *Let (X, d) be a metric space and $f : X \to X$ a contraction on this metric space. Then there exist some unique point $a \in X$, called the unique fixed point of f, such that for all $x_0 \in X$, we have that the sequence*

$$x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots, x_n = f(x_{n-1}), \ldots$$

converges to a.

We then go on to look at the notions of bounded and compactness.

Definition 2.2.9 (Compact space [Lin17]). Let $(V, \|\cdot\|)$ be a normed space. A subset K of the normed space $(V, \|\cdot\|)$ is then called a compact set if every sequence $\{\mathbf{u}_n\}$ in K has a subsequence $\{\mathbf{u}_{n_k}\}$ converging to a point $\mathbf{u}$ in K. The space $(V, \|\cdot\|)$ is itself compact if V is a compact set.

Then we proceed to define boundedness for a subset of a normed space under the metric induced by the norm.

Definition 2.2.10 (Bounded [Lin17]). Let A be a subset of a normed space $(V, \|\cdot\|)$. We then say that A is bounded if there exists a number $M \in \mathbb{R}$ such that $d(\mathbf{u}, \mathbf{v}) \leq M$ for all $\mathbf{u}, \mathbf{v} \in A$, where the metric d is the one induced by the norm of the normed space (2.1).

The next proposition ensures that a compact set K in a normed space always is closed and bounded. We leave out the proof, but see [Lin17] for a proof of the same result but for general metric spaces.

Proposition 2.2.11 (Compact set in normed space [Lin17]). *Let K be a compact subset of a normed space $(V, \|\cdot\|)$. Then K is closed and bounded.*

CHAPTER 3

Introduction to dynamic programming

The following introduction to dynamic programming is based on [Ber17].

3.1 What is a dynamic programming problem?

In dynamic programming (DP), we look at problems where decisions are made in stages, and where each action influences our cost. It is this cost that we would like to optimize. At each stage of the process, we have an idea of what outcome our choice of action will lead to, but not necessarily with certainty. We will also at each stage need to consider the long-time effect of our action, as we would like to minimize the total cost over all the decisions we make. Our basic model consists of an underlying discrete-time dynamic system, and an additive cost function. In this chapter we only consider problems where the time horizon is finite.

A dynamic programming system has the following form

$$x_{k+1} = f_k(x_k, u_k, w_k), \qquad k = 0, 1, \dots, N-1, \tag{3.1}$$

where k indexes the discrete time, x_k is what we call the state of the system, u_k is the control variable to be chosen at time k, w_k is a random parameter, which we can think of as noise, N is our time horizon, while f_k is the function that describes the evolution of our state variable. We let $x_k \in X_k, u_k \in U_k, w_k \in D_k$ and call X_k the state space, U_k the control space, while D_k is called the disturbance space. Note that all of these spaces in general depends on the discrete time k. The distribution of the random parameter value w_k depend on the state x_k and the control u_k, i.e. $w_k \sim P_k(\cdot|x_k, u_k)$. However, the random noise w_k in our model is independent of the previous disturbances $w_0, \dots, w_{k-1}$. The fact that w_k is independent of previous disturbances and that $x_{k+1} = f_k(x_k, u_k, w_k)$, implies that x_{k+1} only depends on the state x_k, the control u_k and the noise w_k. This property is called the *Markov property*. We can also introduce constraints on the control that depend on the current state e.g., the maximal capital we can invest (a control u_k) depends on the current state of our capital (the state x_k). We denote the constrained subspace of the control space that is available in state x_k by $U(x_k) \subset U_k$.

As mentioned above, we are also talking about a cost in dynamic programming problems. This cost is additive, so if the cost at time k is given

by $g_k(x_k, u_k, w_k)$, and if we add a terminal cost $g_N(x_N)$, which is deterministic given x_N, at the end of the process we end up with a total cost of

$$g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k). \tag{3.2}$$

Note again that the cost function depends on a random parameter w_k, therefore we have that the cost in general is a random variable (Definition 2.1.3) as well. Thus, we try to minimize the expected cost and not the cost directly. The expected cost is given by

$$E\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k)\right]. \tag{3.3}$$

We have two notions for minimizing this cost: Closed-loop and open-loop minimization. In open-loop minimization we choose all our controls $u_0, \ldots, u_{N-1}$ immediately at time 0, while in closed-loop minimization we wait until time k before we select our control u_k. Thus, we are able to take the current, and passed states, $(x_k, x_{k-1}, \ldots, x_0)$ into consideration when making a decision about control u_k.

In closed-loop optimization we are not interested in finding the specific numerical values, but a strategy for finding these values that depend on the current state of the system and the time k. In other words, we want to find functions $\mu_k \colon X_k \to U_k$ such that for any state x_k we have that $\mu_k(x_k)$ is the value we should choose as our control u_k. We call a sequence $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ a policy. Then, for any policy we have that the expected cost is

$$J_\pi(x_0) = E\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right], \tag{3.4}$$

where the expectation is over the random variables x_k and w_k. For a given initial state x_0 we want to minimize the value $J_\pi(x_0)$ over all policies π that are valid for our problem. For a policy to be valid we need $u_k = \mu_k(x_k) \in U_k(x_k)$. A policy that satisfies this criterion is called *admissible*.

We call a policy π^* that satisfies

$$J_{\pi^*}(x_0) = \min_{\pi \in \Pi} J_\pi(x_0)$$

an *optimal policy*, denoted π^*. Here Π denotes the set of all admissible policies π. Another point of interest is the *optimal cost function* $J^*(x_0)$ that assigns each initial state x_0 the optimal cost, i.e.

$$J^*(x_0) = \min_{\pi \in \Pi} J_\pi(x_0).$$

Discrete-state problems

Sometimes, the natural domain for our state variable x_n is the integers. If this is the case and we are looking at the system Equation (3.1) where the probability distributions $P_k(w_k|x_k, u_k)$ are known, then we can look at the transition probabilities

$$p_{i\,j}(u, k) = P_k(\{w | f_k(i, u, w) = j\} | x_k = i, u_k = u)$$

for each state. Thus we have two different ways to describe our problem. The transition probability $p_{i\,j}(u,k)$ tell us what the probability for the next state being j is, given that the current state is i and that the control we are choosing is u at time k.

Concerning specification of spaces

We will in this chapter not specify what kind of spaces X_k, U_k and D_k are, since we for a rigorous treatment would need an underlying probability space $(\Omega, \mathcal{F}, \mathbb{P})$ for each policy, and the cost function needs to be a random variable (Definition 2.1.3) on that space. In order for this to hold we may need measurability assumptions on f_k, g_k and μ_k as well as additional structure on X_k, U_k and D_k. A particular example taken from [Ber17] where the above difficulties are resolved is when the spaces D_k are countable and where we for every admissible policy have that the expected value of the cost is finite. In this situation, we can write the cost function as

$$J_\pi(x_0) = \underset{x_1,\dots,x_N}{E}\left[g_N(x_N) + \sum_{k=0}^{N-1} \tilde{g}_k(x_k, \mu_k(x_k))\right], \tag{3.5}$$

with

$$\tilde{g}_k(x_k, \mu_k(x_k)) = \underset{w_k}{E}\left[g_k(x_k, \mu_k(x_k), w_k) | x_k, \mu_k(x_k)\right],$$

where the last expectation is with respect to $P_k(\cdot|x_k, \mu_k(x_k))$, which is a probability measure (Definition 2.1.2) on the countable set D_k. We can then choose the Cartesian product $\tilde{X}_0 \times \tilde{X}_1 \times \cdots \times \tilde{X}_N$ as our basic probability space (Section 2.1), where $\tilde{X}_0 = \{x_0\}$ and

$$\tilde{X}_k = \{x_k \in X_k | x_k = f_{k-1}(x_{k-1}, \mu_{k-1}(x_{k-1}), w_{k-1}), x_{k-1} \in \tilde{X}_{k-1}, w_{k-1} \in D_{k-1}\}.$$

Thus, $\tilde{X}_k$ is the set of all states x_i that are reachable by time $k \in \{0, \dots, N-1\}$ with the policy $\{\mu_0, \dots, \mu_{N-1}\}$. Then, since $\tilde{X}_0$ is finite and D_k is countable for each k, we have that $\tilde{X}_k$ is countable for each k as well. We then have a probability distribution on the Cartesian product of the spaces $\tilde{X}_k$ induced by the dynamics of the system, $x_{k+1} = f_k(x_k, \mu_k(x_k), w_k)$, the first state x_0, the probability distributions $P_k(\cdot|x_K, \mu_k(x_k))$ and the policy $\{\mu_0, \dots, \mu_{N-1}\}$. This Cartesian product is, as mentioned above, countable and the expected value in Equation (3.5) is with respect to this distribution.

3.2 The dynamic programming algorithm

A important principle of dynamic programming is the *principle of optimality*. The principle states that if $\pi^* = \{\mu_0^*, \mu_1^*, \dots, \mu_{N-1}^*\}$ is an optimal policy, and if we by following the policy have a positive probability of state x_i occuring at time i, then we have that the policy $\hat{\pi}_i^* = \{\mu_i^*, \mu_{i+1}^*, \dots, \mu_{N-1}^*\}$ is optimal for minimizing the 'cost-to-go' from time i, which means that the policy $\hat{\pi}_i^*$ minimizes

$$E\left[g_N(x_N) + \sum_{k=i}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right].$$

This is quite intuitive, since if the policy is optimal from the initial state x_0 it should choose the optimal action at each time step, including our time i and all time steps after i. The principle of optimality implies that we can solve a dynamic programming problem backwards, since we can start by finding the optimal policy for the *tail subproblem of length 1.* That is to find the policy that minimizes

$$E\left[g_N(x_N) + g_{N-1}(x_{N-1}, \mu_{N-1}(x_{N-1}), w_{N-1})\right].$$

We can then go on to solve the *tail subproblem of length 2* where we include the $N-2$ term and so on. This way of solving a dynamic programming problem is called *The dynamic programming algorithm*, and the following theorem from [Ber17] guarantees that the algorithm finds the optimal policy π^*.

Theorem 3.2.1 ([Ber17, p. 25])**.** *For every initial state $x_0 \in X_0$, the optimal cost $J^*(x_0)$ of the basic problem is equal to $J_0(x_0)$, given by the last step of the following algorithm, which proceeds backward in time from period $N-1$ to period 0:*

$$J_N(x_N) = g_N(x_N), \tag{3.6}$$

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} E_{w_k}\left[g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))\right], \quad k = 0, 1, \ldots, N-1, \tag{3.7}$$

where the expectation is taken with respect to the probability distribution of w_k, which depends on $x_k \in X_k$ and $u_k \in U(x_k) \subset U_k$. Furthermore, if $u_k^* = \mu_k^*(x_k)$ minimizes the right-hand side of Equation (3.7) *for each $x_k \in X_k$ and $k \in \{0, \ldots, N-1\}$, the policy $\pi^* = \{\mu_0^*, \ldots, \mu_{N-1}^*\}$ is optimal.*

The function $J_k(x_k)$ from the theorem is called the *'cost-to-go' function* and the value gives us the *'cost-to-go'* from state x_k at time k.

Example 3.2.2 (Solving a DP problem, continuation of Example 2.1.8)**.** †
This example is inspired by exercise 1.2 in [Ber17]. Let

$$x_{k+1} = x_k + u_k + w_k,$$

with $x_0 = 0$, and put

$$g_N(x_N) = 0$$
$$g_k(x_k, u_k) = u_k^2 - x_k.$$

We let $k = 0, 1, 2$, and thus $N = 3$. Assume also that

$$U(x_k) = \{u \in \mathbb{N} \colon 0 \leq u + x_k \leq 3\}.$$

In addition, let $W = \{-1, 0, 1\}$, where the disturbance $w_k \in W$ is given by the random variable X described in Example 2.1.8. We let the probability distribution P_k of Ω depend on the sum of x_k and u_k, such that

$$P_k(\cdot|x_k, u_k) = \begin{cases} Q, & \text{if } x_k + u_k = 0 \text{ or } x_k + u_k = 3, \\ P, & \text{otherwise,} \end{cases}$$

where the probability measures (Definition 2.1.2) P, Q, and the corresponding probability spaces, are as described in Example 2.1.8. We also have the constraint that $x_3 = 3$. Note that our constraints imply that $X_k = \{0, 1, 2, 3\}$ for $k = 1, 2$, while $X_0 = \{0\}$ and $X_3 = \{3\}$. Let us now try to find the optimal cost $J_0(0)$ by using the algorithm described above in Theorem 3.2.1. Since $g_3(x_3) = 0$, and we have the constraint $x_3 = 3$, we see that

$$J_3(x_3) = J_3(3) = 0.$$

We then go on to calculate $J_2(x_2)$. Keep in mind that we need $x_3 = x_2 + u_2 = 3$. Thus our choice of control u_2 is forced for each fixed x_2.

$$\begin{aligned}
J_2(0) &= \min_{u_2 \in U(0)} E_{P_2}[u_2^2 - 0 + J_3(0 + u_2 + w_2)] = 3^2 - 0 + J_3(0 + 3 + 0) = 9, \\
J_2(1) &= \min_{u_2 \in U(1)} E_{P_2}[u_2^2 - 1 + J_3(1 + u_2 + w_2)] = 2^2 - 1 + J_3(1 + 2 + 0) = 3, \\
J_2(2) &= \min_{u_2 \in U(2)} E_{P_2}[u_2^2 - 2 + J_3(2 + u_2 + w_2)] = 1^2 - 2 + J_3(2 + 1 + 0) = -1, \\
J_2(3) &= \min_{u_2 \in U(3)} E_{P_2}[u_2^2 - 3 + J_3(3 + u_2 + w_2)] = 0^2 - 3 + J_3(3 + 0 + 0) = -3.
\end{aligned}$$

Thus,

$$\mu_2(0) = 3,\ \mu_2(1) = 2,\ \mu_2(2) = 1,\ \mu_2(3) = 0,$$

as $\mu_2(x)$ was the argument minimizing $J_2(x)$ in the above calculation for each $x \in \{0, 1, 2, 3\}$. We then calculate $J_1(x_1)$.

$$\begin{aligned}
J_1(0) &= \min_{u_1 \in U(0)} E_{P_1}[u_1^2 - 0 + J_2(0 + u_1 + w_1)] \\
&= \min_{u_1 \in \{0,1,2,3\}} E_{P_1}[u_1^2 - 0 + J_2(0 + u_1 + w_1)] \\
&= \min\{E_Q[0^2 + J_2(0 + w_1)], E_P[1^2 + J_2(1 + w_1)], \\
&\qquad E_P[2^2 + J_2(2 + w_1)], E_Q[3^2 + J_2(3 + w_1)]\} \\
&= \min\Big\{J_2(0), 1 + \frac{1}{3}(J_2(0) + J_2(1) + J_2(2)), \\
&\qquad 4 + \frac{1}{3}(J_2(1) + J_2(2) + J_2(3)), 9 + J_2(3)\Big\} \\
&= \min\Big\{9, \frac{14}{3}, \frac{11}{3}, 6\Big\} = \frac{11}{3}, \\
J_1(1) &= \min_{u_1 \in U(1)} E_{P_1}[u_1^2 - 1 + J_2(1 + u_1 + w_1)] \\
&= \min_{u_1 \in \{0,1,2\}} E_{P_1}[u_1^2 - 1 + J_2(1 + u_1 + w_1)] \\
&= \min\{E_P[0^2 - 1 + J_2(1 + w_1)], E_P[1^2 - 1 + J_2(2 + w_1)], \\
&\qquad E_Q[2^2 - 1 + J_2(3 + w_1)]\} \\
&= \min\Big\{-1 + \frac{1}{3}(J_2(0) + J_2(1) + J_2(2)), \\
&\qquad 1 - 1 + \frac{1}{3}(J_2(1) + J_2(2) + J_2(3)), 4 - 1 + J_2(3)\Big\}
\end{aligned}$$

$$
\begin{aligned}
&= \min\left\{\frac{8}{3}, -\frac{1}{3}, 0\right\} = -\frac{1}{3},\\
J_1(2) &= \min_{u_1 \in U(2)} E_{P_1}[u_1^2 - 2 + J_2(2 + u_1 + w_1)]\\
&= \min_{u_1 \in \{0,1\}} E_{P_1}[u_1^2 - 2 + J_2(2 + u_1 + w_1)]\\
&= \min\{E_P[0^2 - 2 + J_2(2 + w_1)], E_Q[1^2 - 2 + J_2(3 + w_1)]\}\\
&= \min\left\{-2 + \frac{1}{3}(J_2(1) + J_2(2) + J_2(3)), 1 - 2 + J_2(3)\right\}\\
&= \min\left\{-\frac{7}{3}, -4\right\} = -4,\\
J_1(3) &= \min_{u_1 \in U(3)} E_{P_1}[u_1^2 - 3 + J_2(3 + u_1 + w_1)]\\
&= \min_{u_1 \in \{0\}} E_{P_1}[u_1^2 - 3 + J_2(3 + u_1 + w_1)]\\
&= E_Q[0^2 - 3 + J_2(3 + w_1)] = -3 + J_2(3) = -6,
\end{aligned}
$$

giving us

$$
\mu_1(0) = 2,\ \mu_1(1) = 1,\ \mu_1(2) = 1,\ \mu_1(3) = 0.
$$

Because of the initial condition $x_0 = 0$, the final calculation becomes

$$
\begin{aligned}
J_0(0) &= \min_{u_0 \in U(0)} E_{P_0}[u_0^2 - 0 + J_1(0 + u_0 + w_0)]\\
&= \min_{u_0 \in \{0,1,2,3\}} E_{P_0}[u_0^2 - 0 + J_1(0 + u_0 + w_0)]\\
&= \min\{E_Q[0^2 + J_1(0 + w_0)], E_P[1^2 + J_1(1 + w_0)],\\
&\qquad E_P[2^2 + J_1(2 + w_0)], E_Q[3^2 + J_1(3 + w_0)]\}\\
&= \min\left\{J_1(0), 1 + \frac{1}{3}(J_1(0) + J_1(1) + J_1(2)),\right.\\
&\qquad \left. 4 + \frac{1}{3}(J_1(1) + J_1(2) + J_1(3)), 9 + J_1(3)\right\}\\
&= \min\left\{\frac{11}{3}, \frac{7}{9}, \frac{5}{9}, 3\right\} = \frac{5}{9}.
\end{aligned}
$$

Thus, $\mu_0(0) = 2$. Then, from Theorem 3.2.1 we have that the optimal value J_0^*, as defined in Section 3.1, is $J_0^*(0) = J_0(0) = \frac{5}{9}$.

3.3 State augmentation and other reformulations

A problem that can arise in dynamic programming is that some of the assumptions of the basic problem are not met, e.g. that the cost function is nonadditive. We can then try to reformulate the problem into the form of the basic problem described in Section 3.1. This is called *state augmentation*. We often perform state augmentation by expanding the state space at each time k such that all information known at time k is included. With inspiration from [Ber17], we will look at techniques for reformulating problems with time delays, correlated disturbances and forecasts.

Time delays

Time delay is when the state x_{n+1} does not only depend on the state x_n and control u_n, but also on earlier states and control, e.g. x_{n-1} and u_{n-1}. In other words, the Markov-property of the dynamic programming problem is no longer satisfied since the state x_{n+1} no longer only depend on the previous state and control. We can formulate this mathematicaly as

$$x_{n+1} = f(x_n, u_n, x_{n-1}, u_{n-1}, w_n).$$

We can then, by introducing two new state variables $y_k = x_{k-1}$ and $s_k = u_{k-1}$, augment the state by letting $\tilde{x}_k \coloneqq (x_k, y_k, s_k)$, and

$$\tilde{x}_{k+1} = \tilde{f}_k(\tilde{x}_k, u_k, w_k) \coloneqq (f(x_n, u_n, x_{n-1}, u_{n-1}, w_n), x_k, u_k). \tag{3.8}$$

Therefore, by having Equation (3.8) as the system equation, we can transform the problem into the format of the basic problem by also expressing the cost-function with the expanded state $\tilde{x}_k$ and by letting the μ_k^* in the policy depend on $\tilde{x}_k$ as well.

Correlated disturbances

We can have problems where the random disturbances w_k are correlated over time. A good example of this kind of correlation, taken from [Ber17], is when the noise can be modelled as $w_k = C_k y_{k+1}$, where $y_{k+1} = A_k y_k + \xi_k$ for each $k = 0, \dots, N-1$. Here, both A_k and C_k are known matrices, and the ξ_k's are random vectors that are independent. We can then, by adding y_k as an additional state variable, write the system equation as

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} f_k(x_k, u_k, C_k(A_k y_k + \xi_k)) \\ A_k y_k + \xi_k \end{pmatrix}. \tag{3.9}$$

Note that we would need to be able to observe y_k if we were to have perfect state information. This is for example the case if C_k is the identity matrix and w_{k-1} is known before the control u_k is chosen. Unfortunately, this is not very realistic.

Forecasts

Now we look at the case where at time $k \in \{0, \dots, N-1\}$ we obtain some information about the disturbance w_k such that we can choose our control u_k with that information in mind. This information could be the exact value of w_k or knowledge about the probability distribution (see Definition 2.1.5) of w_k. State augmentation can be difficult when dealing with forecasts, so we will only take a look at a simple example taken from [Ber17], and then expand Example 3.2.2 with a similar forecast structure.

Assume that $\{Q_1, \dots Q_m\}$ are probability distributions that w_k can have. So if the forecast y_k is i, then w_k has probability distribution Q_i. We are also given the a priori probabilities p_i for the forecast being equal to i, i.e., the probability of a forecast being i is p_i. We represent the forecast as

$$y_k = \xi_{k-1},$$

where ξ_{k-1} is a random variable (see Definition 2.1.3) that can take the values $1, \ldots, m$ (corresponding to the m possible probability distributions for w_k) with probabilities $p_1, \ldots, p_m$, respectively. Hence, if $\xi_{k-1} = i$ we know at time k that w_k has probability distribution Q_i. We can then define the new state $\tilde{x}_k = (x_k, y_k)$ given by

$$\tilde{x}_k = (x_k, y_k) = (f_{k-1}(x_{k-1}, u_{k-1}, w_{k-1}), \xi_{k-1}).$$

The random noise in this case is $\tilde{w}_k = (w_k, \xi_k)$, and its probability distribution is given by the distributions Q_i and the probabilities p_i. These values depend directly on the value of y_k in $\tilde{x}_k$. We now take a look at an example where we again consider the problem from Example 3.2.2, but now with forecasting.

Example 3.3.1 (DP with forecast, continuation of Example 3.2.2)**.** †
This example is partly inspired by exercise 3.3 in [Ber17]. We consider again the problem described in Example 3.2.2, but we now introduce a forecast $y_k = \xi_{k-1}$, $k = 0, 1, 2$ giving us information about the probability distribution of Ω, and thus also the distribution of w_k. Here, ξ_{k-1} is a random variable that is either 1 or 2, both with probability $\frac{1}{2}$. We also let $\tilde{x}_k = (x_k, y_k)$ and redefine the probability distribution P_k of Ω to depend on our forecast y_k in the following way:

$$P_k(\cdot|\tilde{x}_k, u_k) = P_k(\cdot|x_k, y_k, u_k) = \begin{cases} Q, & \text{if } x_k + u_k = 0 \text{ or } x_k + u_k = 3, \\ P, & \text{if } 0 < x_k + u_k < 3 \text{ and } y_k = 1, \\ R, & \text{otherwise,} \end{cases}$$

where the probability measures (Definition 2.1.2) P, Q, R are as described in Example 2.1.8. We can then use the reformulation introduced above to rewrite our system as

$$\tilde{x}_{k+1} = (x_{k+1}, y_{k+1}) = (x_k + u_k + w_k, \xi_k).$$

Then, we have that the stochastic disturbance of the system is given by $\tilde{w}_{k+1} = (w_{k+1}, \xi_{k+1})$. Note that $\tilde{w}_{k+1}$ depends on the value of y_{k+1}, as the probability distribution of Ω, which influences the probability distribution of w_{k+1}, is dependent on y_{k+1}. From Example 3.2.2 we have the constraints $x_0 = 0$ and $x_3 = 3$. We can then try to find $J_0(0,1)$ and $J_0(0,2)$ by use of the dynamic programming algorithm (Theorem 3.2.1). As in Example 3.2.2, we have that

$$J_3(\tilde{x}_3) = J_3(x_3, y_3) = J_3(3, y_3) = 0,$$

since $g_3(x_3) = 0$ and because the constraints of the problem demands that $x_3 = 3$. We then continue with finding the values of $J_2(\tilde{x}_2)$. Since we need $x_3 = 3$, our choice of control u_2 is predetermined to be such that $x_2 + u_2 = 3$. Thus, $P_3(\cdot|\tilde{x}_2, u_2) = Q$ for each possible $\tilde{x}_2$. Therefore, our calculations end up being the same as in Example 3.2.2. We do not bother to repeat the calculations here, so we just reiterate the findings. Recall that

$$J_2(0,1) = J_2(0,2) = 9,\ J_2(1,1) = J_2(1,2) = 3,$$
$$J_2(2,1) = J_2(2,2) = -1, J_2(3,1) = J_2(3,2) = -3,$$

and

$$\mu_2(0,1) = \mu_2(0,2) = 3,\ \mu_2(1,1) = \mu_2(1,2) = 2,$$
$$\mu_2(2,1) = \mu_2(2,2) = 1,\ \mu_2(3,1) = \mu_2(3,2) = 0.$$

We can now move on to calculate the values for $J_1(\tilde{x}_1)$. Note first that for all states $\tilde{x}_1 = (x_1, 1)$, $x_1 = 0, 1, 2, 3$, we get the same values as we calculated for $J_1(x_1)$ in Example 3.2.2. This is the case as we are taking the expectation with respect to the same probability measure as in Example 3.2.2 when $y_1 = 1$, and because $J_2(\tilde{x}_2) = J_2(x_2, y_2)$ is equal to the value $J_2(x_2)$ from Example 3.2.2 for each $x_2 = 0, 1, 2, 3$, and for each $y_2 = 1, 2$. The last property also let us simplify the calculations, since we do not need to take the expectation with respect to the value of y_2. We then calculate $J_1(\tilde{x}_1)$ for states where $y_1 = 2$.

$$\begin{aligned}
J_1(0,2) &= \min_{u_1 \in U(0)} E_{P_1}[u_1^2 - 0 + J_2(0 + u_1 + w_1)] \\
&= \min_{u_1 \in \{0,1,2,3\}} E_{P_1}[u_1^2 - 0 + J_2(0 + u_1 + w_1)] \\
&= \min\{E_Q[0^2 + J_2(0 + w_1)], E_R[1^2 + J_2(1 + w_1)], \\
&\qquad E_R[2^2 + J_2(2 + w_1)], E_Q[3^2 + J_2(3 + w_1)]\} \\
&= \min\left\{J_2(0), 1 + \frac{1}{2}(J_2(0) + J_2(1)), 4 + \frac{1}{2}(J_2(1) + J_2(2)), 9 + J_2(3)\right\} \\
&= \min\{9, 7, 5, 6\} = 5, \\
J_1(1,2) &= \min_{u_1 \in U(1)} E_{P_1}[u_1^2 - 1 + J_2(1 + u_1 + w_1)] \\
&= \min_{u_1 \in \{0,1,2\}} E_{P_1}[u_1^2 - 1 + J_2(1 + u_1 + w_1)] \\
&= \min\{E_R[0^2 - 1 + J_2(1 + w_1)], E_R[1^2 - 1 + J_2(2 + w_1)], \\
&\qquad E_Q[2^2 - 1 + J_2(3 + w_1)]\} \\
&= \min\left\{-1 + \frac{1}{2}(J_2(0) + J_2(1)), 1 - 1 + \frac{1}{2}(J_2(1) + J_2(2)), 4 - 1 + J_2(3)\right\} \\
&= \min\{5, 1, 0\} = 0, \\
J_1(2,2) &= \min_{u_1 \in U(2)} E_{P_1}[u_1^2 - 2 + J_2(2 + u_1 + w_1)] \\
&= \min_{u_1 \in \{0,1\}} E_{P_1}[u_1^2 - 2 + J_2(2 + u_1 + w_1)] \\
&= \min\left\{E_R[0^2 - 2 + J_2(2 + w_1)], E_Q[1^2 - 2 + J_2(3 + w_1)]\right\} \\
&= \min\{-2 + \frac{1}{2}(J_2(1) + J_2(2)), 1 - 2 + J_2(3)\} = \min\{-1, -4\} = -4, \\
J_1(3,2) &= \min_{u_1 \in U(3)} E_{P_1}[u_1^2 - 3 + J_2(3 + u_1 + w_1)] \\
&= \min_{u_1 \in \{0\}} E_{P_1}[u_1^2 - 3 + J_2(3 + u_1 + w_1)] \\
&= E_Q[0^2 - 3 + J_2(3 + w_1)] = -3 + J_2(3) = -6,
\end{aligned}$$

giving us

$$\mu_1(0,2) = 2,\ \mu_1(1,2) = 2,\ \mu_1(2,2) = 1,\ \mu_1(3,2) = 2.$$

We also have from Example 3.2.2 that

$$J_1(0,1) = \frac{11}{3},\ J_1(1,1) = -\frac{1}{3},\ J_1(2,1) = -4,\ J_1(3,1) = -6,$$

and

$$\mu_1(0,1) = 2,\ \mu_1(1,1) = 1, \mu_1(2,1) = 1, \mu_1(3,1) = 2.$$

We now let $E_{\xi_0}[\cdot]$ be the expectation (see Definition 2.1.4) with respect to the distribution (see Definition 2.1.5) of the random variable ξ_0. Since we have the constraint $x_0 = 0$, the final round of calculations yields

$$\begin{aligned}
J_0(0,1) &= \min_{u_0 \in U(0)} E_{P_0}[E_{\xi_0}[u_0^2 - 0 + J_1(0 + u_0 + w_0, \xi_0)]] \\
&= \min_{u_0 \in \{0,1,2,3\}} E_{P_0}[E_{\xi_0}[u_0^2 - 0 + J_1(0 + u_0 + w_0, \xi_0)]] \\
&= \min\{E_Q[E_{\xi_0}[0^2 + J_1(0 + w_0, \xi_0)]], E_P[E_{\xi_0}[1^2 + J_1(1 + w_0, \xi_0)]], \\
&\qquad E_P[E_{\xi_0}[2^2 + J_1(2 + w_0, \xi_0)]], E_Q[E_{\xi_0}[3^2 + J_1(3 + w_0, \xi_0)]]\} \\
&= \min\left\{E_Q\left[0^2 + \frac{1}{2}(J_1(0 + w_0, 1) + J_1(0 + w_0, 2))\right],\right. \\
&\qquad E_P\left[1^2 + \frac{1}{2}(J_1(1 + w_0, 1) + J_1(1 + w_0, 2))\right], \\
&\qquad E_P\left[2^2 + \frac{1}{2}(J_1(2 + w_0, 1) + J_1(2 + w_0, 2))\right], \\
&\qquad \left.E_Q\left[3^2 + \frac{1}{2}(J_1(3 + w_0, 1) + J_1(3 + w_0, 2))\right]\right\} \\
&= \min\left\{\frac{1}{2}(J_1(0,1) + J_1(0,2)),\right. \\
&\qquad 1 + \frac{1}{6}(J_1(0,1) + J_1(0,2) + J_1(1,1) + J_1(1,2) + J_1(2,1) + J_1(2,2)) \\
&\qquad 4 + \frac{1}{6}(J_1(1,1) + J_1(1,2) + J_1(2,1) + J_1(2,2) + J_1(3,1) + J_1(3,2)) \\
&\qquad \left.9 + \frac{1}{2}(J_1(3,1) + J_1(3,2))\right\} \\
&= \min\left\{\frac{13}{3}, \frac{19}{18}, \frac{11}{18}, 3\right\} = \frac{11}{18} \\
J_0(0,2) &= \min_{u_0 \in U(0)} E_{P_0}[E_{\xi_0}[u_0^2 - 0 + J_1(0 + u_0 + w_0, \xi_0)]] \\
&= \min_{u_0 \in \{0,1,2,3\}} E_{P_0}[E_{\xi_0}[u_0^2 - 0 + J_1(0 + u_0 + w_0, \xi_0)]] \\
&= \min\{E_Q[E_{\xi_0}[0^2 + J_1(0 + w_0, \xi_0)]], E_R[E_{\xi_0}[1^2 + J_1(1 + w_0, \xi_0)]], \\
&\qquad E_R[E_{\xi_0}[2^2 + J_1(2 + w_0, \xi_0)]], E_Q[E_{\xi_0}[3^2 + J_1(3 + w_0, \xi_0)]]\} \\
&= \min\left\{E_Q\left[0^2 + \frac{1}{2}(J_1(0 + w_0, 1) + J_1(0 + w_0, 2))\right],\right. \\
&\qquad E_R\left[1^2 + \frac{1}{2}(J_1(1 + w_0, 1) + J_1(1 + w_0, 2))\right], \\
&\qquad E_R\left[2^2 + \frac{1}{2}(J_1(2 + w_0, 1) + J_1(2 + w_0, 2))\right],
\end{aligned}$$

$$
\begin{aligned}
&E_Q\left[3^2 + \frac{1}{2}(J_1(3+w_0,1) + J_1(3+w_0,2))\right]\Bigg\} \\
&= \min\Bigg\{\frac{1}{2}(J_1(0,1)+J_1(0,2)), \\
&\quad 1 + \frac{1}{4}(J_1(0,1)+J_1(0,2)+J_1(1,1)+J_1(1,2)) \\
&\quad 4 + \frac{1}{4}(J_1(1,1)+J_1(1,2)+J_1(2,1)+J_1(2,2)) \\
&\quad 9 + \frac{1}{2}(J_1(3,1)+J_1(3,2))\Bigg\} \\
&= \min\left\{\frac{13}{3}, \frac{47}{12}, \frac{23}{12}, 3\right\} = \frac{23}{12}.
\end{aligned}
$$

We thus have

$$\mu_0(0,1) = 2, \mu_0(0,2) = 2.$$

We conclude that the expected optimal value is

$$E_{\xi_{-1}}[J_0(0,\xi_{-1})] = \frac{1}{2}(J_0(0,1)+J_0(0,2)) = \frac{1}{2}\left(\frac{11}{18}+\frac{23}{12}\right) = \frac{91}{72}.$$

3.4 Deterministic systems and the shortest path problem

The following section is also based on [Ber17]. We will in this section take a look at deterministic problems with a discrete character, which is problems where w_k only has one possible value. Then, given some state x_k and a policy $\pi = \{\mu_0, \dots \mu_{N-1}\}$, we can find the control u_k as usual by

$$u_k = \mu_k(x_k).$$

We can then determine the next state x_{k+1} by

$$x_{k+1} = f_k(x_k, \mu_k(x_k)).$$

We are therefore able to optimize the cost over the controls $\{u_0, \dots, u_{N-1}\}$ instead of the admissible policies $\pi = \{\mu_0, \dots \mu_{N-1}\}$ that we described in Section 3.1.

Finite-state systems and shortest paths

We now consider the case where the state space at each time step is finite, i.e., X_k is finite for each $k = 1, \dots, N-1$. It is then possible, when at a state $x_k \in X_k$, to relate a control $u_k \in U(x_k) \subset U_k$ with the transition to the state $x_{k+1} = f_k(x_k, u_k)$ where the cost is given by $g(x_k, u_k)$. Therefore, we can represent this kind of problem as a graph with N layers, in addition to a starting node s and a terminal node t. Here, the nodes at layer k represent the possible states at time k. The arcs from nodes at a layer k to nodes at the layer $k+1$ then represents the transition from x_k to x_{k+1}, i.e., the control u_k. If we think of the cost $g(x_k, u_k)$ as the length of the arc representing u_k,

we see that our DP problem can be interpreted as finding the shortest path from s to t. A path is a collection of arcs $(j_1, j_2), (j_2, j_3), \ldots, (j_{N-1}, j_N)$. Here (j_{k-1}, j_k) represent choosing the control u_{k-1} when $x_{k-1} = j_{k-1}$ such that $x_k = f_{k-1}(x_{k-1}, u_{k-1}) = j_k$. The shortest path from s to t is then the sequence $(s, j_1), \ldots, (j_N, t)$ such that the sum

$$\sum_{k=1}^{N-1} g(j_k, u_k)$$

is as small as possible. Here u_k represent the arc that take us from j_k to j_{k+1} when (j_k, j_{k+1}) is in our path.

We now introduce the following notation,

$$a_{ij}^k := \text{ Cost of going from state } i \in X_k \text{ to state } j \in X_{k+1} \text{ at time } k,$$

$$a_{it}^N := g_N(i) = \text{ Terminal cost for state } i \in X_N.$$

We also set $a_{ij}^k := \infty$ when there is no control u_k that take us from state i to state j at time k. The DP algorithm, introduced in Section 3.2, for a finite-state system can then be written as

$$J_N(i) = a_{it}^N, \; i \in X_N, \tag{3.10}$$

$$J_k(i) = \min_{j \in X_{k+1}} (a_{ij}^k + J_{k+1}(j)), \; i \in X_k, \; k = 0, \ldots, N-1. \tag{3.11}$$

We can then interpret the optimal cost, $J_0(s)$, as the length of the shortest path connecting s and t.

Example 3.4.1 (Graph representation, continuation of Example 3.2.2)**.** †
Let us again consider the dynamic programming problem from Example 3.2.2, but now a deterministic version. That is, we let

$$P_k(\cdot | x_k, u_k) = Q$$

induce the probability distribution (Definition 2.1.5) of Ω. Recall that

$$Q(\{w_1\}) = Q(\{w_3\}) = 0, Q(\{w_2\}) = 1.$$

Then,

$$w_k = 0, \; k = 0, 1, 2.$$

We will now try to construct a graph that represents this system. Since we have that $x_0 = 0$, we let the start node s and the node representing $x_0 = 0$ be the same node. Accordingly, since $x_3 = 3$ and $g_3(3) = 0$, we let the termination node t and the node representing $x_3 = 3$ be the same node. We also need nodes representing every possible state for $k = 1, 2$, which we call stage 1 and stage 2 respectively in our graph. Then, as described above in Section 3.4, we let the numbers on (or lengths of) the arcs between the nodes be equal to

$$a_{x_k x_{k+1}}^k = g(x_k, u_k) = u_k^2 - x_k,$$

when the node where the arc originates has the value $x_k \in X_k$ and the end node of the arc has the value $x_k + u_k = x_{k+1} \in X_{k+1}$. Here, we only draw the arcs that have a finite value, but we could have drawn the graph as a complete graph where the missing arcs had an infinite cost (length). The resulting graph can be seen in Figure 3.1.

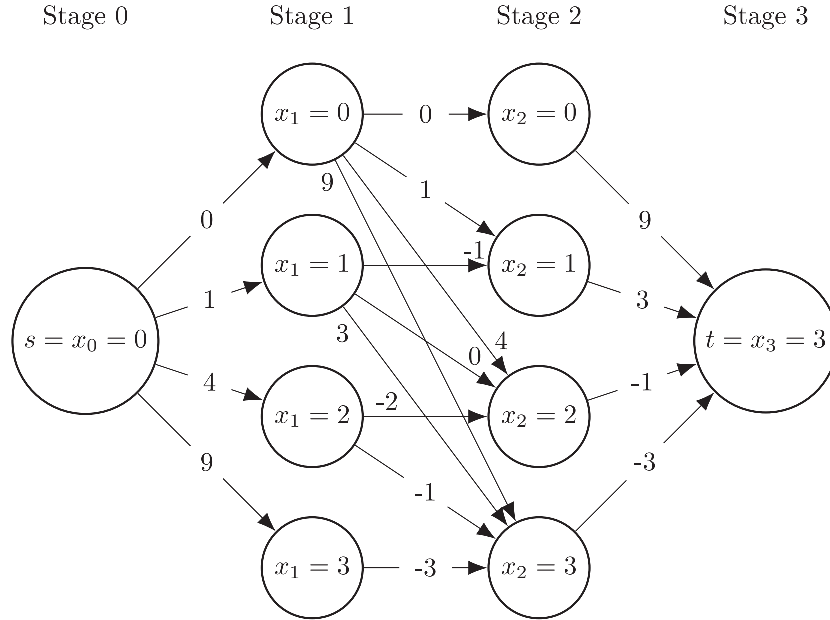

Figure 3.1: A graph representation of a deterministic finite-state system

A forward DP algorithm for shortest path problems

A disadvantage with the regular DP algorithm from Section 3.2 is that it starts with $J_N(i)$ and then calculates backwards. Thus, if we are solving a problem in real-time, the algorithm is not applicable, since we do not know what the final state looks like. However, for deterministic finite-state problems we can find a forward DP algorithm. If we again look at the DP problem as a shortest path problem, we can just flip each edge and let the length be the same as for the original edge. The optimal path will now be the same as for the original problem, just reversed. Then, the forward DP algorithm becomes

$$\tilde{J}_N(j) = a^0_{sj},\ j \in X_1, \tag{3.12}$$

$$\tilde{J}_k(j) = \min_{i \in X_{N-k}} (a^{N-k}_{ij} + \tilde{J}_{k+1}(i)),\ j \in X_{N-k+1},\ k = 1, \dots, N-1. \tag{3.13}$$

The optimal cost, or shortest path, becomes

$$\tilde{J}_0(t) = \min_{i \in X_N} (a^N_{it} + \tilde{J}_1(i)),$$

and we have that

$$J_0(s) = \tilde{J}_0(t).$$

Just as we viewed Equation (3.7) as the 'cost-to-go' function, we can view $\tilde{J}_k(j)$ as the 'cost-to-arrive' function. That is, the optimal cost of starting from s and arriving at state j.

Example 3.4.2 (Forward DP, continuation of Example 3.4.1). †
In this example, we demonstrate the forward dynamic programming algorithm on the deterministic finite-state system from Example 3.4.1. Let us start by

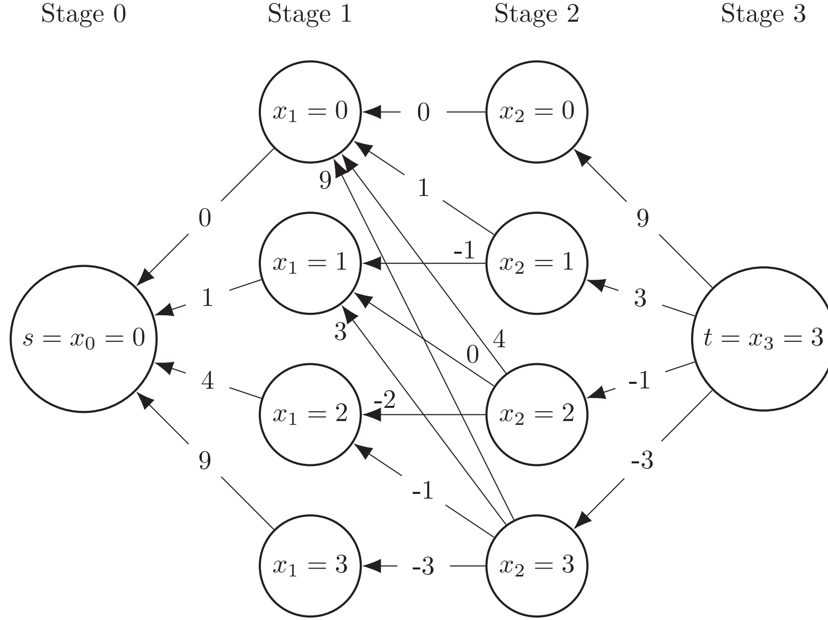

Figure 3.2: The flipped deterministic finite-state system

drawing the flipped version of the graph from Figure 3.1 that represents our new system that we will apply the forward DP algorithm to. The flipped graph can be seen in Figure 3.2. Keep in mind that $a_{ij}^k = g(i, j-i)$, as $u_k = j - i$, and that if an arc (i_k, j_k) is not drawn in the graph, we have that $a_{ij}^k = \infty$. Note also that we can find the costs a_{ij}^k from the graph in Figure 3.2. Then, since we have the initial value $x_0 = 0$ we can start the forward DP algorithm with

$$\begin{aligned}
\tilde{J}_3(0) &= a_{00}^0 = g(0,0) = 0^2 - 0 = 0, \\
\tilde{J}_3(1) &= a_{01}^0 = g(0,1) = 1^2 - 0 = 1, \\
\tilde{J}_3(2) &= a_{02}^0 = g(0,2) = 2^2 - 0 = 4, \\
\tilde{J}_3(3) &= a_{03}^0 = g(0,3) = 3^2 - 0 = 9.
\end{aligned}$$

Thus,

$$\mu_3(0) = 0, \mu_3(1) = 0, \mu_3(2) = 0, \mu_3(3) = 0.$$

Note that $\mu_3(x_1)$ in this setting is not the control $u_0 \in U(x_1)$ per se, but the next state of the reversed path. This makes sense for the forward dynamic programming algorithm since we minimize over states in X_{N-k} and not controls in $U_k(x_k)$. Thus, since each path needs to end at $s = x_0 = 0$, we have that $\mu_3(x_1) = 0$. Then, at the next stage we get

$$\begin{aligned}
\tilde{J}_2(0) &= \min_{x_1 \in X_1} (a_{x_1 0}^1 + \tilde{J}_3(x_1)) = \min_{x_1 \in \{0,1,2,3\}} (a_{x_1 0}^1 + \tilde{J}_3(x_1)) \\
&= \min\{0 + \tilde{J}_3(0), \infty + \tilde{J}_3(1), \infty + \tilde{J}_3(2), \infty + \tilde{J}_3(3)\} \\
&= \min\{0, \infty, \infty, \infty\} = 0, \\
\tilde{J}_2(1) &= \min_{x_1 \in X_1} (a_{x_1 1}^1 + \tilde{J}_3(x_1)) = \min_{x_1 \in \{0,1,2,3\}} (a_{x_1 0}^1 + \tilde{J}_3(x_1))
\end{aligned}$$

$$
\begin{aligned}
&= \min\{1 + \tilde{J}_3(0), -1 + \tilde{J}_3(1), \infty + \tilde{J}_3(2), \infty + \tilde{J}_3(3)\} \\
&= \min\{1, 0, \infty, \infty\} = 0, \\
\tilde{J}_2(2) &= \min_{x_1 \in X_1} (a^1_{x_1 2} + \tilde{J}_3(x_1)) = \min_{x_1 \in \{0,1,2,3\}} (a^1_{x_1 0} + \tilde{J}_3(x_1)) \\
&= \min\{4 + \tilde{J}_3(0), 0 + \tilde{J}_3(1), -2 + \tilde{J}_3(2), \infty + \tilde{J}_3(3)\} \\
&= \min\{4, 1, 2, \infty\} = 1, \\
\tilde{J}_2(3) &= \min_{x_1 \in X_1} (a^1_{x_1 3} + \tilde{J}_3(x_1)) = \min_{x_1 \in \{0,1,2,3\}} (a^1_{x_1 0} + \tilde{J}_3(x_1)) \\
&= \min\{9 + \tilde{J}_3(0), 3 + \tilde{J}_3(1), -1 + \tilde{J}_3(2), -3 + \tilde{J}_3(3)\} \\
&= \min\{9, 4, 3, 6\} = 3.
\end{aligned}
$$

This gives

$$\mu_2(0) = 0, \mu_2(1) = 1, \mu_2(2) = 1, \mu_2(3) = 2.$$

We then continue with calculating $\tilde{J}_1(x_3)$, and since we have the constraint $x_3 = 3$, we get

$$
\begin{aligned}
\tilde{J}_1(3) &= \min_{x_2 \in X_2} (a^2_{x_2 3} + \tilde{J}_2(x_2)) = \min_{x_2 \in \{0,1,2,3\}} (a^2_{x_2 3} + \tilde{J}_2(x_2)) \\
&= \min\{9 + \tilde{J}_2(0), 3 + \tilde{J}_2(1), -1 + \tilde{J}_2(2), -3 + \tilde{J}_2(3)\} \\
&= \min\{9, 3, 0, 0\} = 0.
\end{aligned}
$$

Therefore,

$$\mu_1(3) = 2,\ 3.$$

Now, since we have that the terminal node t and the node representing $x_3 = 3$ is the same node, we have that the optimal value

$$\tilde{J}_0(t) = \tilde{J}_1(3) = 0.$$

Recall that we introduced paths in Section 3.4, and note that we here have two candidates for the shortest path, namely

$$\{(x_3 = 3, x_2 = 2), (x_2 = 2, x_1 = 1), (x_1 = 1, x_0 = 0)\},$$

and

$$\{(x_3 = 3, x_2 = 3), (x_2 = 3, x_1 = 2), (x_1 = 2, x_0 = 0)\}.$$

Both paths are drawn in the graph in Figure 3.3, where the first path is drawn in blue and the other in orange.

Converting a Shortest Path Problem to a Deterministic Finite-State Problem

In this subsection, we show that we can convert any shortest path problem into a deterministic finite-state problem. In the previous Section 3.4, we showed that we can convert any deterministic finite-state problem into a shortest path problem. Therefore, we have that the two types of problems are equivalent.

Assume that $\{1, 2, \dots, N, t\}$ are nodes in a graph. We denote the cost of traversing the graph from node i to node j by a_{ij}, and if there is no arc from i to j, we set $a_{ij} = \infty$. The node t is called the destination node, and our

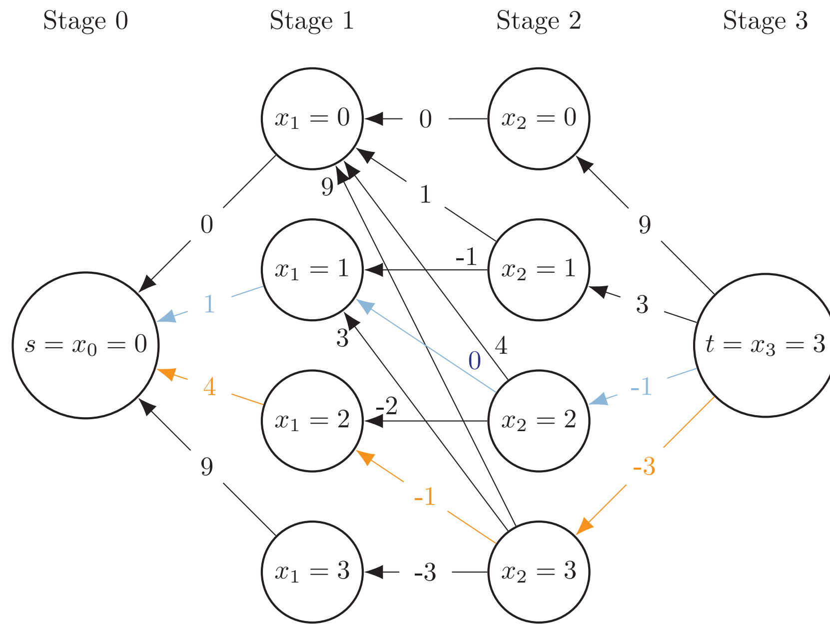

Figure 3.3: The two candidates for shortest path

goal is to discover the shortest path to the destination node from each node i. The only assumption we need is that there are no cycles with negative cost, as a negative cycle would allow us to decrease the cost as much as we would like. We then have that the optimal path can not exceed N moves. Thus, we require every solution to take exactly N moves, but we allow the paths to have stationary moves. That is, we can move from node i to itself at a cost of 0, i.e., $a_{ii} = 0$ for all $i \in \{1, 2, \ldots, N\}$. Then, we introduce

$$J_k(i) \coloneqq \text{ optimal cost when in } N-k \text{ steps moving to } t \text{ from } i,\ k \in \{0, 1, \ldots, N-1\},$$

and we let $i \in \{1, 2, \ldots, N\}$. Note that $J_0(i)$ is the cost of traversing the shortest path between i and t. The corresponding DP algorithm (Section 3.2) then takes the form

$$J_{N-1}(i) = a_{it}, \tag{3.14}$$

$$J_k(i) = \min_{j=1,\ldots,N} (a_{ij} + J_{k+1}(j)),\ k \in \{0, 1, \ldots, N-2\}. \tag{3.15}$$

We see that this is a deterministic finite-state problem, which we introduced in Section 3.4. Hence, we have shown that we can convert a general shortest path problem to a deterministic finite-state problem. Thus, as mentioned at the beginning of this subsection, we can conclude that the two types of problems are equivalent. Let us now look at an example where we take a shortest path problem and solve it using the dynamic programming algorithm described above 3.14.

Example 3.4.3 (Converting a shortest path problem to a DP-problem)**.** †

This example is inspired by exercise 2.3 in [Ber17]. Let us consider a travel planning problem. We are supposed to travel from city a to city b and would

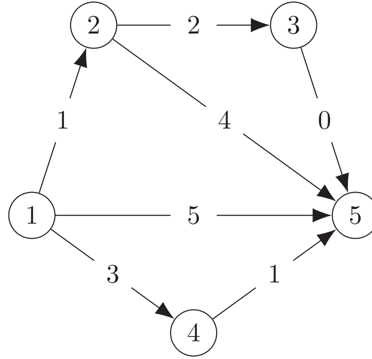

Figure 3.4: Possible travelling routes

like to minimize the cost of our trip. Since the cheapest path is not necessarily a direct route between a and b, we also consider some stopover options. We represent the possible options as a graph, where each city is represented as a node. We let node 1 and 5 represent city a and b, respectively. The cost of going between two cities is represented as the length of the arc connecting the nodes that represent the cities. If we consider an arc between the nodes i, and j we use a_{ij} to denote the length of the arc. If an arc is not drawn on the graph we did not manage to find a route between the cities, and we let the cost equal ∞. Thus, if i and j are two nodes that are not connected by an edge we let $a_{ij} = \infty$. In our example we have the nodes $\{1, 2, 3, 4, 5\}$, where node 5 is the terminal node, called t. The graph is shown in Figure 3.4.

We now want to apply the DP-algorithm described in Equation (3.14) and Equation (3.15) to find the shortest path from each node $i \in \{1, 2, 3, 4\}$ to node t. As $N = 4$, we start by finding the $J_3(i)$ values, which denote the optimal cost from node i to the terminal node t when only allowed one move, which gives

$$J_3(1) = 5,\ J_3(2) = 4,\ J_3(3) = 0,\ J_3(4) = 1.$$

Note also that

$$\mu_3(i) = 5 \text{ for } i \in \{1, 2, 3, 4\},$$

where $\mu_k(i)$ gives the next node along the optimal path from i to t that reaches t in $N - k$ moves. Thus $J_k(i)$ gives the length of the shortest path from i to t that reaches the terminal node in $N - k$ moves. We now go on to calculate $J_2(i)$ for the different choices of i.

$$J_2(1) = \min\{0 + J_3(1),\ 1 + J_3(2),\ \infty + J_3(3),\ 3 + J_3(4)\} = \min\{5, 5, \infty, 4\} = 4,$$
$$J_2(2) = \min\{\infty + J_3(1),\ 0 + J_3(2),\ 2 + J_3(3),\ \infty + J_3(4)\} = \min\{\infty, 4, 2, \infty\} = 2,$$
$$J_2(3) = \min\{\infty + J_3(1),\ \infty + J_3(2),\ 0 + J_3(3),\ \infty + J_3(4)\} = \min\{\infty, \infty, 0, \infty\} = 0,$$
$$J_2(4) = \min\{\infty + J_3(1),\ \infty + J_3(2),\ \infty + J_3(3),\ 0 + J_3(4)\} = \min\{\infty, \infty, \infty, 1\} = 1.$$

This gives

$$\mu_2(1) = 4,\ \mu_2(2) = 3,\ \mu_2(3) = 3,\ \mu_2(4) = 4.$$

We then continue with $k = 1$.

$$J_1(1) = \min\{0 + J_2(1),\ 1 + J_2(2),\ \infty + J_2(3),\ 3 + J_2(4)\} = \min\{4, 3, \infty, 4\} = 3,$$

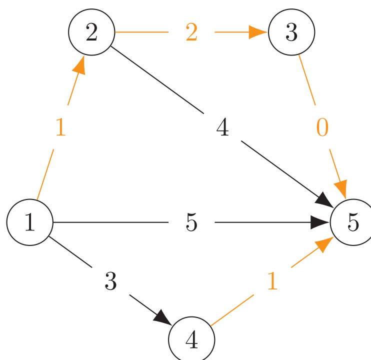

Figure 3.5: Shortest travelling routes

$$J_1(2) = \min\{\infty + J_2(1),\ 0 + J_2(2),\ 2 + J_2(3),\ \infty + J_2(4)\} = \min\{\infty, 2, 2, \infty\} = 2,$$
$$J_1(3) = \min\{\infty + J_2(1),\ \infty + J_2(2),\ 0 + J_2(3),\ \infty + J_2(4)\} = \min\{\infty, \infty, 0, \infty\} = 0,$$
$$J_1(4) = \min\{\infty + J_2(1),\ \infty + J_2(2),\ \infty + J_2(3),\ 0 + J_2(4)\} = \min\{\infty, \infty, \infty, 1\} = 1.$$

Thus,

$$\mu_1(1) = 2,\ \mu_1(2) = 2 \vee 3,\ \mu_1(3) = 3,\ \mu_1(4) = 4.$$

Then we proceed with calculating the shortest paths for each node i, namely $J_0(i)$.

$$J_0(1) = \min\{0 + J_1(1),\ 1 + J_1(2),\ \infty + J_1(3),\ 3 + J_1(4)\} = \min\{3, 3, \infty, 4\} = 3,$$
$$J_0(2) = \min\{\infty + J_1(1),\ 0 + J_1(2),\ 2 + J_1(3),\ \infty + J_1(4)\} = \min\{\infty, 2, 2, \infty\} = 2,$$
$$J_0(3) = \min\{\infty + J_1(1),\ \infty + J_1(2),\ 0 + J_1(3),\ \infty + J_1(4)\} = \min\{\infty, \infty, 0, \infty\} = 0,$$
$$J_0(4) = \min\{\infty + J_1(1),\ \infty + J_1(2),\ \infty + J_1(3),\ 0 + J_1(4)\} = \min\{\infty, \infty, \infty, 1\} = 1.$$

We then end up with

$$\mu_0(1) = 1 \vee 2,\ \mu_0(2) = 2 \vee 3,\ \mu_0(3) = 3,\ \mu_0(4) = 4.$$

Hence, the most affordable route from a to b costs $J_0(1) = 3$ units. The corresponding shortest path is drawn in the graph in Figure 3.5. We also note that this was a quite small problem with only 5 nodes, but if the number of nodes N were to increase, computing the shortest route by hand become very tedious. Therefore, we often solve problems like this computationally.

3.5 Stochastic systems with imperfect state information

This section is as the former based on [Ber17]. In the previous sections, we assumed that the controller has perfect information about state x_k at time k. Thus, we have assumed that we always know the exact value of x_k. However, this is not necessarily the case. In this section, we model a situation where the controller does not have access to the underlying state, but only observes an observation z_k. These observations, $z_0, \ldots, z_{N-1}$, depend on the current state x_k, a random observation disturbance v_k and the control u_{k-1}. We write

$$z_0 = h_0(x_0, v_0),\ z_k = h_k(x_k, u_{k-1}, v_k),\ k = 1, 2, \ldots, N-1.$$

We let the observations $z_k \in Z_k$, and the random observation disturbances $v_k \in V_k$. We also let v_k be distributed according to a given probability distribution (see Definition 2.1.5) that depends on the state at time k, i.e., x_k, as well as all previous data. In more rigorous terms, we have

$$P_{v_k}(\cdot|x_k, \ldots x_0, u_{k-1}, \ldots, u_0, w_{k-1}, \ldots, w_0, v_{k-1}, \ldots, v_0).$$

As before, we have that the probability distribution of w_k at time k, $P_{w_k}(\cdot|x_k, u_k)$, only depends on the current state and control, that is x_k, u_k. We also have that the control $u_k \in \overline{U}_k \subset U_k$ belongs to a known non-empty subset $\overline{U}_k$ of the control space U_k, but in contrast to the basic dynamic programming problem introduced in Section 3.1 we do not assume that $\overline{U}_k$ depends on x_k. Another assumption is that the initial state x_0 is distributed according to a given probability distribution P_{x_0}.

Since our observation z_k depends on all previous data through the random disturbance v_k, the controller should take into account all known information at time k to make a decision about the control u_k. That is, the functions μ_k that have the subsets $\overline{U}_k$ of the control space U_k as its co-domain should have the space $Z_0 \times \cdots \times Z_k \times U_0 \times \cdots \times U_{k-1}$ as its domain. We call vectors that live in these spaces for *information vectors*. We let

$$I_0 = z_0,\ I_1 = (z_0, z_1, u_0),\ \ldots, I_{N-1} = (z_0, \ldots, z_{N-1}, u_0, \ldots, u_{N-2}).$$

Then, for a policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ to be admissible (see Section 3.1) we would need μ_k to map the information I_k known to the controller at time k into the relevant subset $\overline{U}_k$ of the control space for each possible information vector I_k and each k. In other words, we would need $\mu_k(I_k) \in \overline{U}_k$ to hold for all $I_k, k = 0, \ldots, N-1$. The goal is then, similarly to the basic problem case described in Section 3.1, to find such an admissible policy that minimizes the cost described by the cost function

$$J_\pi = \underset{\substack{x_0, w_k, v_k,\\ k=0,\ldots,N-1}}{E} \left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(I_k), w_k) \right]$$

where the dynamics of the system is given by

$$x_{k+1} = f_k(x_k, \mu_k(I_k), w_k), k = 0, 1, \ldots, N-1,$$

along with the equations

$$z_0 = h_0(x_0, v_0),$$
$$z_k = h_k(x_k, \mu_{k-1}(I_{k-1}), v_k), k = 1, 2, \ldots, N-1.$$

To be able to solve this, we need to reformulate the problem stated above, similarly to what we did in Section 3.3. That means that we need to reformulate the problem into a perfect state information version. To do this, we need to find a new system where the state at time k contains all the information that could be useful for the controller when choosing the control u_k. An intuitive first choice is the information vector I_k. We can thus look at the system

$$I_0 = z_0,\ I_k = (I_{k-1}, z_k, u_{k-1}),\ k = 0, \ldots, N-2. \tag{3.16}$$

Then, note that we can look at u_{k-1} as the control and z_k as a random disturbance that has the probability distribution $P(z_k|I_{k-1}, u_{k-1}, z_{k-1}, \ldots, z_0)$. This would usually not satisfy the requirement of the basic problem (Section 3.1). That is because the random disturbance z_k only should depend on the current state I_{k-1} and control u_{k-1}, but since $z_{k-1}, \ldots, z_0 \in I_{k-1}$, we have that

$$P(z_k|I_{k-1}, u_{k-1}, z_{k-1}, \ldots, z_0) = P(z_k|I_{k-1}, u_{k-1}).$$

Hence, the disturbance z_k of the new system in Equation (3.16) depends only on the current state I_{k-1} and the control u_{k-1}. Therefore, the requirement of the basic problem holds. We can then write

$$E[g_k(x_k, u_k, w_k)] = E\left[\underset{x_k, w_k}{E}[g_k(x_k, u_k, w_k)|I_k, u_k]\right],$$

and use this to find a new expression for the cost at time k that only depends on the state I_k and control u_k. We get

$$\tilde{g}_k(I_k, u_k) = \underset{x_k, w_k}{E}[g_k(x_k, u_k, w_k)|I_k, u_k]. \tag{3.17}$$

We can then write one possible DP algorithm for problems with imperfect state information by using Equation (3.16) and Equation (3.17). We have

$$\begin{aligned} J_{N-1}(I_{N-1}) = \min_{u_{N-1} \in U_{N-1}} \Bigg[\underset{x_k, w_k}{E} [g_N(f_{N-1}(x_{N-1}, u_{N-1}, w_{N-1})) \\ + g_{N-1}(x_{N-1}, u_{N-1}, w_{N-1})|I_{N-1}, u_{N-1}]\Bigg], \end{aligned} \tag{3.18}$$

$$J_k(I_k) = \min_{u_k \in U_k} \left[\underset{z_{k+1}}{E}[\tilde{g}_k(I_k, u_k) + J_{k+1}(I_k, z_{k+1}, u_k)|I_k, u_k]\right]. \tag{3.19}$$

We can then for each possible information vector I_{N-1} find the control $u_{N-1} \in U_{N-1}$ that minimizes Equation (3.18) and set $\mu^*_{N-1}(I_{N-1}) = u_{N-1} \in \overline{U}_{N-1}$. Then, we can calculate $J_{N-1}(I_{N-1})$, and use Equation (3.19) to obtain μ^*_{N-2} by finding the controls $u_{N-2} \in \overline{U}_{N-2}$ that are minimizing Equation (3.19) for all possible values of I_{N-2}. We can then continue in this fashion until we find the value of $J_0(x_0) = J_0(z_0)$. The optimal policy then becomes $\pi^* = \{\mu^*_0, \ldots, \mu^*_{N-1}\}$ and the optimal cost, J^*, is given by

$$J^* = \underset{z_0}{E}[J_0(z_0)].$$

Sufficient statistic

A challenge with DP problems with imperfect state information is that the dimension of the state space where the information vector I_k resides is expanding, as we at each stage add a measurement. A sufficient statistic tries to overcome this challenge by introducing some quantity $X_k(I_k)$ of smaller dimension than I_k that still contain all the information from I_k that is important for the controller in order to solve the problem at hand. In other words, we want some mapping X_k that maps the information vector I_k to some quantity such that we can rewrite Equations (3.18) and (3.19) into

$$\min_{u_k \in U_k} H_k(X_k(I_k), u_k), \tag{3.20}$$

for some function H_k. We would then be able to find a policy $\bar{\mu}_k$ such that

$$\mu^*(I_k) = \bar{\mu}_k(X_k(I_k)),$$

where $\mu^*(I_k)$ is the optimal policy found by solving the DP algorithm given by Equations (3.18) and (3.19). The functions that are reasonable to use as sufficient statistics are of course problem dependent, and a trivial choice is $X_k(I_k) = I_k$, but that does not help us much. In the following example we illustrate the practicality of using a sufficient statistic by having the conditional probability distribution of the state x_k, given the information vector I_k, denoted $P_{x_k|I_k}$, as our sufficient statistic. In [Ber17] it is shown that this in fact is a sufficient statistic and that we are able to find a problem specific function ϕ_k such that we can recursively update our sufficient statistic by

$$P_{x_{k+1}|I_{k+1}} = \phi_k(P_{x_k|I_k}, u_k, z_{k+1}),$$

which is also illustrated in the following example.

Example 3.5.1 (Imperfect state information, continuation of Example 3.3.1)**.** †
This example is partly inspired by exercise 4.10 in [Ber17] and the sufficient statistic used is inspired by the proposed solution from [Ber]. We now consider a version of the problem discussed in Example 3.3.1 where we introduce an element of uncertainty. Instead of the value y_k being forecasted at each stage, as in Example 3.3.1, we now let $y_{k+1} = y_k, k = 0, 1, 2$ where y_0 is chosen at random. We let $P(y_0 = 1) = P(y_0 = 2) = \frac{1}{2}$. The element of uncertainty comes from y_0 which is unknown to the controller, while the probability p of $y_0 = 1$ is known. Recall that the underlying scenario space Ω (see Section 2.1) has the probability distribution

$$P_k(\cdot|\tilde{x}_k, u_k) = P_k(\cdot|x_k, y_k, u_k) = \begin{cases} Q, & \text{if } x_k + u_k = 0 \text{ or } x_k + u_k = 3, \\ P, & \text{if } 0 < x_k + u_k < 3 \text{ and } y_k = 1, \\ R, & \text{otherwise,} \end{cases}$$

where the probability measures P, Q, and R are as defined in Example 2.1.8. Thus, by not knowing the value of y_0, and thereby not knowing the value of any y_k, $k = 0, 1, 2, 3$, we do not know whether our disturbance w_k, $k = 0, 1, 2$ has the probability distribution P or R when $0 < x_k + u_k < 3$. A way to tackle this problem is to introduce a sufficient statistic (Section 3.5) and to find a way of quantifying our belief about y_k at time k based on the observed disturbances, $w_{k-1}, \cdots, w_0$. Before introducing a sufficient statistic, let us take a look at our system. The underlying system is

$$\tilde{x}_{k+1} = (x_{k+1}, y_{k+1}) = (x_k + u_k + w_k, y_k),\ k = 0, 1, 2,$$

while the controller only observes

$$z_k = x_k.$$

We then introduce the sufficient statistic, (x_k, p_k), where p_k is our belief that the value of y_k is 1 given the previously observed disturbances. That is,

$$p_k = P(y_k = 1|w_{k-1}, \cdots, w_0).$$

We can then, by use of Bayes' theorem (Theorem 2.1.7), update our belief by

$$\begin{aligned} p_{k+1} &= \phi_k(p_k, x_k, u_k, w_k) \\ &= \frac{p_k P(w_k|y_k = 1, x_k, u_k)}{p_k P(w_k|y_k = 1, x_k, u_k) + (1 - p_k) P(w_k|y_k = 2, x_k, u_k)}, k = 1, 2, \end{aligned}$$

where $p_0 = p$. We can then use the dynamic programming algorithm for problems with imperfect state information (see Section 3.5) to find the expected optimal value. We first note that since $g_3(x_3) = 0$, and because we have the constraint $x_3 = 3$, we get from Equation (3.18) that

$$J_2(x_2, p_2) = \min_{u_2 \in U(x_2)} \{u_2^2 - x_2\}, \tag{3.21}$$

which is exactly the same value calculated in Example 3.2.2. Thus, we have that

$$J_2(0, \cdot) = 9, J_2(1, \cdot) = 3, J_2(2, \cdot) = -1, J_2(3, \cdot) = -3.$$

The value of p_2 has no impact on $J_2(x_2, p_2)$, since the constraint $x_3 = 3$ forces us to choose u_2 such that $x_2 + u_2 = 3$. Hence

$$E[J_3(3, \phi_2(p_2, x_2, u_2, w_2))|y_2 = 1] = E[J_3(3, \phi_2(p_2, x_2, u_2, w_2))|y_2 = 2] = 0.$$

However, this is not the case for $k = 0, 1$. Then, we have that

$$\begin{aligned} J_1(x_1, p_1) = \min_{u_1 \in U(x_1)} &\{u_1^2 - x_1 + p_1 E[J_2(x_1 + u_1 + w_1, \phi_1(p_1, x_1, u_1, w_1))|y_1 = 1] \\ &+ (1 - p_1) E[J_2(x_1 + u_1 + w_1, \phi_1(p_1, x_1, u_1, w_1))|y_1 = 2]\} \\ J_0(x_0, p_0) = \min_{u_0 \in U(x_0)} &\{u_0^2 - x_0 + p_0 E[J_1(x_0 + u_0 + w_0, \phi_0(p_0, x_0, u_0, w_0))|y_0 = 1] \\ &+ (1 - p_0) E[J_1(x_0 + u_0 + w_0, \phi_0(p_0, x_0, u_0, w_0))|y_0 = 2]\}. \end{aligned}$$

To be able to calculate the values for $J_1(x_1, p_1)$, we first need to find each possible value for p_1. Since $x_0 = 0$, we have that $U(x_0) = \{0, 1, 2, 3\}$. We also know that $p_0 = p = \frac{1}{2}$ and that $w_0 \in \{-1, 0, 1\}$. Thus,

$$\begin{aligned} p_1 &= \phi_0(p_0, x_0, u_0, w_0) = \phi_0\left(\frac{1}{2}, 0, u_0, w_0\right) \\ &= \frac{\frac{1}{2} P(w_0|y_0 = 1, 0, u_0)}{\frac{1}{2} P(w_0|y_k = 1, 0, u_0) + (1 - \frac{1}{2}) P(w_0|y_0 = 2, 0, u_0)}, \end{aligned}$$

with $u_0 \in \{0, 1, 2, 3\}$, $w_0 \in \{-1, 0, 1\}$. We also note from the definition of P_k

that if $u_0 = 0, 3$ we have $w_0 = 0$. Some simple calculations then yield

$$\begin{aligned}
\phi_0\left(\frac{1}{2},0,0,0\right) &= \phi_0\left(\frac{1}{2},0,3,0\right) \\
&= \frac{\frac{1}{2}P(0|y_0=1,x_0=0,u_0=3)}{\frac{1}{2}P(0|y_k=1,x_0=0,u_0=3)+(1-\frac{1}{2})P(0|y_0=2,x_0=0,u_0=3)} \\
&= \frac{\frac{1}{2}}{\frac{1}{2}+\frac{1}{2}} = \frac{1}{2}, \\
\phi_0\left(\frac{1}{2},0,1,-1\right) &= \phi_0\left(\frac{1}{2},0,1,0\right) = \phi_0\left(\frac{1}{2},0,2,-1\right) = \phi_0\left(\frac{1}{2},0,2,0\right) \\
&= \frac{\frac{1}{2}P(0|y_0=1,x_0=0,u_0=2)}{\frac{1}{2}P(0|y_k=1,x_0=0,u_0=2)+(1-\frac{1}{2})P(0|y_0=2,x_0=0,u_0=2)} \\
&= \frac{\frac{1}{2}\cdot\frac{1}{3}}{\frac{1}{2}\cdot\frac{1}{3}+\frac{1}{2}\cdot\frac{1}{2}} = \frac{1}{4}, \\
\phi_0\left(\frac{1}{2},0,1,1\right) &= \phi_0\left(\frac{1}{2},0,2,1\right) \\
&= \frac{\frac{1}{2}P(1|y_0=1,x_0=0,u_0=2)}{\frac{1}{2}P(1|y_k=1,x_0=0,u_0=2)+(1-\frac{1}{2})P(1|y_0=2,x_0=0,u_0=2)} \\
&= \frac{\frac{1}{2}\cdot 1}{\frac{1}{2}\cdot 1+\frac{1}{2}\cdot 0} = 1.
\end{aligned}$$

We can then go on to find values for $J_1(x_1, p_1)$. We start by finding the following expressions

$$\begin{aligned}
J_1(0,p_1) = &\min_{u_1\in U(0)}\{u_1^2 - 0 + p_1 E_{P_1}[J_2(0+u_1+w_1,\phi_1(p_1,0,u_1,w_1))|y_1=1] \\
&\qquad + (1-p_1)E_{P_1}[J_2(0+u_1+w_1,\phi_1(p_1,0,u_1,w_1))|y_1=2]\} \\
= &\min_{u_1\in\{0,1,2,3\}}\{u_1^2 + p_1 E_{P_1}[J_2(u_1+w_1)|y_1=1] \\
&\qquad + (1-p_1)E_{P_1}[J_2(u_1+w_1)|y_1=2]\} \\
= &\min\{p_1 E_{P_1}[J_2(w_1)|y_1=1] + (1-p_1)E_{P_1}[J_2(w_1)|y_1=2], \\
&\quad 1 + p_1 E_{P_1}[J_2(1+w_1)|y_1=1] + (1-p_1)E_{P_1}[J_2(1+w_1)|y_1=2], \\
&\quad 4 + p_1 E_{P_1}[J_2(2+w_1)|y_1=1] + (1-p_1)E_{P_1}[J_2(2+w_1)|y_1=2], \\
&\quad 9 + p_1 E_{P_1}[J_2(3+w_1)|y_1=1] + (1-p_1)E_{P_1}[J_2(3+w_1)|y_1=2]\} \\
= &\min\{p_1 J_2(0) + (1-p_1)J_2(0), \\
&\quad 1 + p_1\frac{1}{3}(J_2(0)+J_2(1)+J_2(2)) + (1-p_1)\frac{1}{2}(J_2(0)+J_2(1)), \\
&\quad 4 + p_1\frac{1}{3}(J_2(1)+J_2(2)+J_2(3)) + (1-p_1)\frac{1}{2}(J_2(1)+J_2(2)), \\
&\quad 9 + p_1 J_2(3) + (1-p_1)J_2(3)\}
\end{aligned}$$

$$
\begin{aligned}
&= \min\left\{9, 1 + p_1\frac{11}{3} + (1-p_1)6, 4 - p_1\frac{1}{3} + (1-p_1), 6\right\},\\
J_1(1,p_1) &= \min_{u_1\in U(1)}\{u_1^2 - 1 + p_1 E_{P_1}[J_2(1+u_1+w_1, \phi_1(p_1,1,u_1,w_1))|y_1 = 1]\\
&\qquad + (1-p_1)E_{P_1}[J_2(1+u_1+w_1, \phi_1(p_1,1,u_1,w_1))|y_1 = 2]\}\\
&= \min_{u_1\in\{0,1,2\}}\{u_1^2 - 1 + p_1 E_{P_1}[J_2(1+u_1+w_1)|y_1 = 1]\\
&\qquad + (1-p_1)E_{P_1}[J_2(1+u_1+w_1)|y_1 = 2]\}\\
&= \min\Bigg\{ -1 + p_1\frac{1}{3}(J_2(0)+J_2(1)+J_2(2)) + (1-p_1)\frac{1}{2}(J_2(0)+J_2(1)),\\
&\qquad p_1\frac{1}{3}(J_2(1)+J_2(2)+J_2(3)) + (1-p_1)\frac{1}{2}(J_2(1)+J_2(2)),\\
&\qquad 3 + p_1 J_2(3) + (1-p_1)J_2(3)\Bigg\}\\
&= \min\left\{-1 + p_1\frac{11}{3} + (1-p_1)6, -p_1\frac{1}{3} + (1-p_1), 0\right\},\\
J_1(2,p_1) &= \min_{u_1\in U(2)}\{u_1^2 - 2 + p_1 E_{P_1}[J_2(2+u_1+w_1, \phi_1(p_1,2,u_1,w_1))|y_1 = 1]\\
&\qquad + (1-p_1)E_{P_1}[J_2(2+u_1+w_1, \phi_1(p_1,2,u_1,w_1))|y_1 = 2]\}\\
&= \min_{u_1\in\{0,1\}}\{u_1^2 - 2 + p_1 E_{P_1}[J_2(2+u_1+w_1)|y_1 = 1]\\
&\qquad + (1-p_1)E_{P_1}[J_2(2+u_1+w_1)|y_1 = 2]\}\\
&= \min\Bigg\{ -2 + p_1\frac{1}{3}(J_2(1)+J_2(2)+J_2(3)) + (1-p_1)\frac{1}{2}(J_2(1)+J_2(2)),\\
&\qquad -1 + p_1 J_2(3) + (1-p_1)J_2(3)\Bigg\}\\
&= \min\left\{-2 - p_1\frac{1}{3} + (1-p_1), -4\right\},\\
J_1(3,p_1) &= \min_{u_1\in U(3)}\{u_1^2 - 3 + p_1 E_{P_1}[J_2(3+u_1+w_1, \phi_1(p_1,3,u_1,w_1))|y_1 = 1]\\
&\qquad + (1-p_1)E_{P_1}[J_2(3+u_1+w_1, \phi_1(p_1,3,u_1,w_1))|y_1 = 2]\}\\
&= \min_{u_1\in\{0\}}\{u_1^2 - 3 + p_1 E_{P_1}[J_2(3+u_1+w_1)|y_1 = 1]\\
&\qquad + (1-p_1)E_{P_1}[J_2(3+u_1+w_1)|y_1 = 2]\}\\
&= -3 + p_1 J_2(3) + (1-p_1)J_2(3) = -6,
\end{aligned}
$$

where we have used the fact that

$$J_2(x_2,p_2) = J_2(x_2) \text{ for all } x_2 \in \{0,1,2,3\}, p_2 \in (0,1).$$

We can then insert the possible values for p_1 (that is $\frac{1}{4}, \frac{1}{2}$, or 1) in the expressions for $J_1(x_1, p_1)$ derived above. Calculations then give

$$
\begin{aligned}
J_1\left(0, \frac{1}{4}\right) &= \min\left\{9, 1 + \frac{1}{4}\cdot\frac{11}{3} + \left(1 - \frac{1}{4}\right)6, 4 - \frac{1}{4}\cdot\frac{1}{3} + \left(1-\frac{1}{4}\right), 6\right\} \\
&= \min\left\{9, \frac{77}{12}, \frac{14}{3}, 6\right\} = \frac{14}{3}, \\
J_1\left(0, \frac{1}{2}\right) &= \min\left\{9, 1 + \frac{1}{2}\cdot\frac{11}{3} + \left(1 - \frac{1}{2}\right)6, 4 - \frac{1}{2}\cdot\frac{1}{3} + \left(1-\frac{1}{2}\right), 6\right\} \\
&= \min\left\{9, \frac{35}{6}, \frac{13}{3}, 6\right\} = \frac{13}{3}, \\
J_1\left(1, \frac{1}{4}\right) &= \min\left\{-1 + \frac{1}{4}\cdot\frac{11}{3} + \left(1 - \frac{1}{4}\right)6, -\frac{1}{4}\cdot\frac{1}{3} + \left(1-\frac{1}{4}\right), 0\right\} \\
&= \min\left\{\frac{53}{12}, \frac{2}{3}, 0\right\} = 0, \\
J_1\left(2, \frac{1}{4}\right) &= \min\left\{-2 - \frac{1}{4}\cdot\frac{1}{3} + \left(1 - \frac{1}{4}\right), -4\right\} = \min\left\{-\frac{4}{3}, -4\right\} = -4, \\
J_1(2,1) &= \min\left\{-2 - 1\cdot\frac{1}{3} + (1-1), -4\right\} = \min\left\{-\frac{7}{3}, -4\right\} = -4, \\
J_1\left(3, \frac{1}{2}\right) &= J_1(3,1) = -6,
\end{aligned}
$$

which means that

$$
\begin{aligned}
\mu_1\left(0, \frac{1}{4}\right) &= \mu_1\left(0, \frac{1}{2}\right) = \mu_1\left(1, \frac{1}{4}\right) = 2, \\
\mu_1\left(2, \frac{1}{4}\right) &= \mu_1(2,1) = 1, \\
\mu_1\left(3, \frac{1}{2}\right) &= \mu_1(3,1) = 0.
\end{aligned}
$$

We can then go on to calculate $J_0\left(0, \frac{1}{2}\right)$. We have

$$
\begin{aligned}
J_0\left(0, \frac{1}{2}\right) = \min_{u_1 \in U(0)} \Bigg\{ & u_0^2 - 0 + \frac{1}{2} E_{P_0}\left[J_1\left(0 + u_0 + w_0, \phi_0\left(\frac{1}{2}, 0, u_0, w_0\right)\right) \Big| y_0 = 1\right] \\
& + \left(1 - \frac{1}{2}\right) E_{P_0}\left[J_1\left(0 + u_0 + w_0, \phi_0\left(\frac{1}{2}, 0, u_0, w_0\right)\right) \Big| y_0 = 2\right] \Bigg\} \\
= \min_{u_0 \in \{0,1,2,3\}} \Bigg\{ & u_0^2 + \frac{1}{2} E_{P_0}\left[J_1\left(u_0 + w_0, \phi_0\left(\frac{1}{2}, 0, u_0, w_0\right)\right) \Big| y_0 = 1\right] \\
& + \frac{1}{2} E_{P_0}\left[J_1\left(u_0 + w_0, \phi_0\left(\frac{1}{2}, 0, u_0, w_0\right)\right) \Big| y_0 = 2\right] \Bigg\} \\
= \min \Bigg\{ & \frac{1}{2} J_1\left(0, \phi_0\left(\frac{1}{2}, 0, 0, 0\right)\right) + \frac{1}{2} J_1\left(0, \phi_0\left(\frac{1}{2}, 0, 0, 0\right)\right),
\end{aligned}
$$

$$
\begin{aligned}
&1+\frac{1}{2}\cdot\frac{1}{3}\Bigg(J_2\left(0,\phi_0\left(\frac{1}{2},0,1,-1\right)\right)+J_2\left(1,\phi_0\left(\frac{1}{2},0,1,0\right)\right)\\
&\qquad+J_2\left(2,\phi_0\left(\frac{1}{2},0,1,1\right)\right)\Bigg)\\
&\quad+\frac{1}{2}\cdot\frac{1}{2}\left(J_1\left(0,\phi_0\left(\frac{1}{2},0,1,-1\right)\right)+J_1\left(1,\phi_0\left(\frac{1}{2},0,1,0\right)\right)\right),\\
&4+\frac{1}{2}\cdot\frac{1}{3}\Bigg(J_1\left(1,\phi_0\left(\frac{1}{2},0,2,-1\right)\right)+J_1\left(2,\phi_0\left(\frac{1}{2},0,2,0\right)\right)\\
&\qquad+J_1\left(3,\phi_0\left(\frac{1}{2},0,2,1\right)\right)\Bigg)\\
&\quad+\frac{1}{2}\cdot\frac{1}{2}\left(J_1\left(1,\phi_0\left(\frac{1}{2},0,2,-1\right),\right)+J_1\left(2,\phi_0\left(\frac{1}{2},0,2,0\right)\right)\right),\\
&9+\frac{1}{2}J_1\left(3,\phi_0\left(\frac{1}{2},0,3,0\right)\right)+\frac{1}{2}J_1\left(3,\phi_0\left(\frac{1}{2},0,3,0\right)\right)\Bigg\}\\
=\min&\left\{\frac{1}{2}J_1\left(0,\frac{1}{2}\right)+\frac{1}{2}J_1\left(0,\frac{1}{2}\right),\right.\\
&1+\frac{1}{2}\cdot\frac{1}{3}\left(J_2\left(0,\frac{1}{4}\right)+J_2\left(1,\frac{1}{4}\right)+J_2\left(2,1\right)\right)\\
&\quad+\frac{1}{2}\cdot\frac{1}{2}\left(J_1\left(0,\frac{1}{4}\right)+J_1\left(1,\frac{1}{4}\right)\right),\\
&4+\frac{1}{2}\cdot\frac{1}{3}\left(J_1\left(1,\frac{1}{4}\right)+J_1\left(2,\frac{1}{4}\right)+J_1\left(3,1\right)\right)\\
&\quad+\frac{1}{2}\cdot\frac{1}{2}\left(J_1\left(1,\frac{1}{4},\right)+J_1\left(2,\frac{1}{4}\right)\right),\\
&\left.9+\frac{1}{2}J_1\left(3,\frac{1}{2}\right)+\frac{1}{2}J_1\left(3,\frac{1}{2}\right)\right\}\\
=\min&\left\{\frac{13}{3},1+\frac{1}{2}\cdot\frac{1}{3}\left(\frac{14}{3}+0-4\right)+\frac{1}{2}\cdot\frac{1}{2}\left(\frac{14}{3}+0\right),\right.\\
&\left.4+\frac{1}{2}\cdot\frac{1}{3}\left(0-4-6\right)+\frac{1}{2}\cdot\frac{1}{2}\left(0+-4\right),9-6\right\}\\
=\min&\left\{\frac{13}{3},\frac{41}{18},\frac{4}{3},3\right\}=\frac{4}{3}.
\end{aligned}
$$

Thus,

$$\mu_0\left(0,\frac{1}{2}\right)=2.$$

This implies that the expected optimal value is

$$J^*\left(0,\frac{1}{2}\right)=J_0\left(0,\frac{1}{2}\right)=\frac{4}{3}.$$

In this chapter we introduced the theory of dynamic programming by considering the finite horizon DP problem. We have also looked at how different types of problems can be rewritten into the standard DP problem, which then can be solved by the DP algorithm. In the next chapter we continue our study of dynamic programming by taking a look at the abstract setting.

CHAPTER 4

Abstract dynamic programming

In this chapter we are going to take a look at abstract dynamic programming, which is a more rigorous look at what we did in Chapter 3. The presentation here is inspired by the book [Ber18].

4.1 The abstract dynamic programming model

We again let X be the set of states, while U is the set of controls, where $U(x) \subset U$ denotes the subset of possible controls in state x. The set of all functions $\mu : X \to U$ where $\mu(x) \in U(x)$, for all $x \in X$ is denoted by $\mathcal{M}$. A non-stationary policy can then be written $\pi = \{\mu_0, \mu_1, \ldots\}$, where $\mu_k \in \mathcal{M}$ for all $k \in \mathbb{N}$. The definition of a stationary policy is given in the following definition.

Definition 4.1.1 (Stationary policy)**.** A stationary policy π is on the form

$$\pi = \{\mu, \mu, \ldots\} \text{ for a } \mu \in \mathcal{M}.$$

We will often, by abuse of notation, write $\mu \in \mathcal{M}$ to denote the stationary policy $\pi = \{\mu, \mu, \ldots\}$. Note that this means that one can look at $\mathcal{M}$ as the set of all feasible stationary policies, and we will therefore refer to $\mathcal{M}$ as the *set of stationary policies*. We denote the set of all policies π by Π. We also introduce the set $\mathcal{R}(X)$ which consist of all real-valued functions $J : X \to \mathbb{R}$. Observe that every cost function J resides in the set $\mathcal{R}(X)$, so we can intuitively think of this set as containing every possible cost function for our problem. We also have some given mapping $H : X \times U \times \mathcal{R}(X) \to \mathbb{R}$. We will throughout this thesis look at many examples of concrete choices for the mapping H, but in general this mapping describes the dynamics of the system at hand. Given some state $x \in X$, a control $u \in U(x)$ and a cost function J, can we in many cases interpret $H(x, u, J)$ as the associated cost of applying control u in state x when assuming that the cost-to-go at each state $x' \in X$ is given by $J(x')$.

The next important ingredients in this abstract framework is the *DP mappings*, which is called *Bellman operators* in the reinforcement learning literature. For each stationary policy $\mu \in \mathcal{M}$ we define the operator $T_\mu : \mathcal{R}(X) \to \mathcal{R}(X)$ by

$$(T_\mu J)(x) = H(x, \mu(x), J), \text{ for all } x \in X \text{ and } J \in \mathcal{R}(X). \tag{4.1}$$

In addition, we have the operator $T\colon \mathcal{R}(X) \to \mathcal{R}(X)$ which is defined as

$$(TJ)(x) = \inf_{u \in U(x)} H(x, u, J), \text{ for all } x \in X \text{ and } J \in \mathcal{R}(X). \tag{4.2}$$

Then, for a given *initial cost function* $\bar{J} \in \mathcal{R}(X)$ and a policy $\pi = \{\mu_0, \mu_1, \dots\}$ we let the *N-stage cost function* of the policy π be defined as

$$J_{\pi,N}(x) = (T_{\mu_0} T_{\mu_1} \cdots T_{\mu_{N-1}} \bar{J})(x),\ x \in X,$$

while the *infinite horizon cost function* of the policy is defined as

$$J_\pi(x) = \limsup_{N \to \infty} J_{\pi,N}(x) = \limsup_{N \to \infty} (T_{\mu_0} T_{\mu_1} \cdots T_{\mu_{N-1}} \bar{J})(x),\ x \in X.$$

The objective in dynamic programming is, as we saw in Chapter 3, often to minimize the total cost. In the abstract setting, this amounts to finding a policy π that minimizes the cost function J_π over the set Π of all policies. That is, we would like to find some policy π^*, called an *optimal policy*, that satisfies

$$J_{\pi^*}(x) = J^*(x) = \inf_{\pi \in \Pi} J_\pi(x) \text{ for all } x \in X,$$

where J^* is the *optimal cost function*. For the problems we look at in this thesis we have that the optimal cost function J^* satisfies

$$J^*(x) = \inf_{u \in U(x)} H(x, u, J^*) \text{ for all } x \in X, \tag{4.3}$$

which is equivalent to J^* being a fixed point of T. We call Equation (4.3) Bellman's equation. We will later in this chapter see that it is a monotonicity and a contraction assumption that guarantee that there indeed exists a fixed point J^* of Equation (4.3) and that this fixed point satisfies

$$J^*(x) = J_{\pi^*}(x) = \inf_{\pi \in \mathcal{M}} J_\pi(x) \text{ for all } x \in X.$$

Therefore, if we are able to prove that our model satisfies the monotonicity and contraction properties, then we would be able to find the optimal cost function by calculating the fixed point of T. We will now take a look at these properties, and start by defining monotonicity.

Definition 4.1.2 (Monotonicity [Ber18])**.** We say that a mapping H has the monotonicity property if for $J, J' \in \mathcal{R}$ with $J \le J'$ pointwise, i.e. $J(x) \le J'(x)$ for all $x \in X$, we have

$$H(x, u, J) \le H(x, u, J') \text{ for all } x \in X \text{ and } u \in U(x).$$

The notion of a contraction was introduced in Definition 2.2.7, but we will now introduce a convenient normed space (Definition 2.2.3) of functions that we will assume our operators to operate on.

Definition 4.1.3 (Space of real-valued functions J [Ber18])**.** Let $v : X \to \mathbb{R}$, with $v(x) > 0$ for all $x \in X$, be a function. We then define $\mathcal{B}(X)$ as the space of functions $J : X \to \mathbb{R}$ where $\frac{J(x)}{v(x)} < M$ for all $x \in X$ for some $M \in \mathbb{R}$. We also equip $\mathcal{B}(X)$ with the following weighted sup-norm:

$$\|J\|_v = \sup_{x \in X} \frac{|J(x)|}{v(x)}. \tag{4.4}$$

If the choice of v is not important we just write $\|\cdot\|$.

We see that $\mathcal{B}(X)$ is a normed space of functions J where the fraction $\frac{J(x)}{v(x)}$ is bounded (Definition 2.2.10). The definition of $\mathcal{B}(X)$ imply that the space is a compact normed space, and a proof of this can be found in the appendix of [Ber18]. We will for the rest of this chapter assume that for arbitrary $J \in \mathcal{B}(X)$ and $\mu \in \mathcal{M}$ we have that $T_\mu J, TJ \in \mathcal{B}(X)$, and that the operator T_μ is a contraction for each $\mu \in \mathcal{M}$, i.e. for some $\alpha \in (0,1)$ the following holds true:

$$d(T_\mu J, T_\mu J') = \|T_\mu J - T_\mu J'\| \leq \alpha \|J - J'\| = \alpha d(J, J'), \tag{4.5}$$

for all $J, J' \in \mathcal{B}(X)$ and $\mu \in \mathcal{M}$, where $d(\cdot,\cdot)$ is the metric induced by the norm (4.4), as described in Proposition 2.2.4.

The following proposition ensures that the assumption of T_μ being a contraction for each $\mu \in \mathcal{M}$ implies that operator T also is a contraction.

Proposition 4.1.4 (T_μ contraction imply T contraction [Ber18])**.** *Assume that for any $J \in \mathcal{B}(X)$ and $\mu \in \mathcal{M}$ we have that $TJ, T_\mu J \in \mathcal{B}(X)$, and that T_μ is a contraction, then the operator T is also a contraction.*

Proof. This proof is based on a sketched proof of the same result given in [Ber18]. We have for all $\mu \in \mathcal{M}$ that

$$\|T_\mu J - T_\mu J'\| \leq \alpha \|J - J'\| \text{ for all } J, J' \in \mathcal{B}(X).$$

Thus,

$$\frac{T_\mu J(x) - T_\mu J'(x)}{v(x)} \leq \alpha \|J - J'\| \text{ for all } x \in X,\ J, J' \in \mathcal{B}(X).$$

This tells us that

$$T_\mu J(x) - T_\mu J'(x) \leq \alpha \|J - J'\| v(x),$$

which imply that $T_\mu J(x) \leq T_\mu J'(x) + \alpha \|J - J'\| v(x)$. Then taking the infimum on both sides over $\mathcal{M}$ gives

$$TJ(x) \leq TJ'(x) + \alpha \|J - J'\| v(x).$$

Then

$$\frac{TJ(x) - TJ'(x)}{v(x)} \leq \alpha \|J - J'\| \text{ for all } x \in X,\ J, J' \in \mathcal{B}(X). \tag{4.6}$$

Now, by repeating the same arguments as above, but with

$$\frac{T_\mu J'(x) - T_\mu J(x)}{v(x)} \leq \alpha \|J - J'\| \text{ for all } x \in X,\ J, J' \in \mathcal{B}(X),$$

we find that

$$\frac{TJ'(x) - TJ(x)}{v(x)} \leq \alpha \|J - J'\| \text{ for all } x \in X,\ J, J' \in \mathcal{B}(X). \tag{4.7}$$

Then by taking the supremum on the left-hand side of Equations (4.6) and (4.7) gives $\|TJ - TJ'\| \leq \alpha \|J - J'\|$. ∎

To better see how the abstract formulation of the dynamic programming problem given above relates to the model we introduced in Section 3.1 we take a look at an example.

Example 4.1.5 (Abstract DP formulation, continuation of Example 3.2.2)**.** †
Let us go back to the DP problem we looked at in Example 3.2.2 and try to formulate the problem in the abstract DP context introduced in this chapter. Recall that we have $x_{k+1} = f(x_k, u_k, w_k) = x_k + u_k + w_k$, with $k \in \{0, 1, 2\}$ and $N = 3$. We also have $g_N(x_N) = 0$ and $g_k(x_k, u_k) = u_k^2 - x_k$. The set of controls are also limited with $U(x_k) = \{u : 0 \leq u + x_k \leq 3, u \geq 0\}$, and we restrict the initial and final state by $x_0 = 0, x_3 = 3$. In addition, we have that the disturbances w_k can take the values $W = \{-1, 0, 1\}$, with probability $P_k(\cdot|x_k, u_k)$.

Let us first define the mapping H for this specific problem. We have

$$\begin{aligned} H(x_k, u_k, J_k) &= E_{P_k}[g_k(x_k, u_k) + \alpha J_k(f(x_k, u_k, w_k))] \\ &= E_{P_k}[g_k(x_k, u_k) + \alpha J_k(x_{k+1})], \end{aligned}$$

where in the last expression the expectation is taken with regards to the value of x_{k+1} as this is the only thing dependent on the random disturbance w_k. Note also that we did not discount the cost function in the problem formulation given in Section 3.1, so we let $\alpha = 1$. We can then simplify the expression for H by writing

$$H(x_k, u_k, J_k) = g_k(x_k, u_k) + E_{P_k}[J_k(x_{k+1})].$$

We can then go on to define the DP mappings.

$$(T_\mu J_k)(x_k) = H(x_k, \mu_k(x_k), J_k) = g_k(x_k, \mu_k(x_k)) + E_{P_k}[J_k(x_{k+1})],$$

$$\begin{aligned} (TJ_k)(x_k) &= \inf_{u_k \in U(x_k)} H(x_k, u_k, J_k) \\ &= \inf_{u_k \in U(x_k)} (g_k(x_k, u_k) + E_{P_k}[J_k(x_k + u_k + w_k)]). \end{aligned}$$

Then, for a given policy $\pi = \{\mu_0, \mu_1, \mu_2\}$, we have

$$J_\pi(x) = (T_{\mu_0} T_{\mu_1} T_{\mu_2} \bar{J})(x),$$

where $\bar{J} = g_3(x_3)$. Let us now try to find an exact expression for $J_\pi(x)$. We start with $k = 2$.

$$\begin{aligned} (T_{\mu_2} \bar{J})(x_2) &= g_2(x_2, \mu_2(x_2)) + E_{P_2}[\bar{J}(x_3)] \\ &= g_2(x_2, 3 - x_2) + E_{P_2}[\bar{J}(3)] = g_2(x_2, 3 - x_2), \end{aligned}$$

where we have used the restriction $x_3 = 3$, and that $\bar{J} = g_3(x_3) = g_3(3) = 0$. Then

$$\begin{aligned} (T_{\mu_1}(T_{\mu_2} \bar{J}))(x_1) &= g_1(x_1, \mu_1(x_1)) + E_{P_1}[(T_{\mu_2} \bar{J})(f(x_1, \mu_1(x_1), w_1))] \\ &= g_1(x_1, \mu_1(x_1)) \\ &\quad + E_{P_1}[g_2(f(x_1, \mu_1(x_1), w_1), 3 - f(x_1, \mu_1(x_1), w_1))]. \end{aligned}$$

We then continue with the last step.

$$
\begin{aligned}
J_\pi(x_0) &= (T_{\mu_0}(T_{\mu_1}T_{\mu_2}\bar{J}))(0) \\
&= g_0(0,\mu_0(0)) + E_{P_0}[(T_{\mu_1}(T_{\mu_2}\bar{J}))(f(0,\mu_0(0),w_0))] \\
&= g_0(0,\mu_0(0)) \\
&\quad + E_{P_0}[g_1(f(0,\mu_0(0),w_0),\mu_1(f(0,\mu_0(0),w_0))) \\
&\qquad\quad + E_{P_1}[g_2(f(f(0,\mu_0(0),w_0),\mu_1(f(0,\mu_0(0),w_0)),w_1), \\
&\qquad\qquad\qquad 3 - f(f(0,\mu_0(0),w_0),\mu_1(f(0,\mu_0(0),w_0)),w_1))]].
\end{aligned}
$$

Then we have that the optimal policy π^* is the one that minimizes J_{π^*}, and this is the policy that we found in Example 3.2.2.

4.2 Consequences of monotonicity and contracion assumptions

We continue with looking at some consequences of assuming that the operators T_μ (4.1) and T (4.2) have the monotonicity (Definition 4.1.2) and contraction (Definition 2.2.7) property, where the later is with respect to the norm on the function space $\mathcal{B}(X)$ as defined in Definition 4.1.3. The first proposition covers properties that are present even when we only assume that the contraction property hold.

Proposition 4.2.1 (Consequences of contraction assumption [Ber18])**.** *Assume that for any $J \in \mathcal{B}(X)$ and $\mu \in \mathcal{M}$ we have that $TJ, T_\mu J \in \mathcal{B}(X)$ and that T_μ is a contraction mapping with contraction factor α. Then*

(i) *The operators T and T_μ, for all $\mu \in \mathcal{M}$ have unique* fixed points *in the space $\mathcal{B}(X)$ which we denote J^* and J_μ, for all $\mu \in \mathcal{M}$ respectively. That is, there exist some $J^* \in \mathcal{B}(X)$ such that $J^* = TJ^*$.*

(ii) *For arbitrary $J \in \mathcal{B}(X)$ and $\mu \in \mathcal{M}$ we have*

$$\lim_{k\to\infty} \|J^* - T^k J\| = 0, \qquad \lim_{k\to\infty} \|J_\mu - T_\mu^k J\| = 0.$$

(iii) *We have that $T_\mu J^* = TJ^*$ if and only if $J_\mu = J^*$.*

(iv) *Let $J \in \mathcal{B}(X)$ be arbitrary, then*

$$\|J^* - J\| \le \frac{1}{1-\alpha}\|TJ - J\|, \qquad \|J^* - TJ\| \le \frac{\alpha}{1-\alpha}\|TJ - J\|.$$

(v) *Let $J \in \mathcal{B}(X)$ and $\mu \in \mathcal{M}$, then*

$$\|J_\mu - J\| \le \frac{1}{1-\alpha}\|T_\mu J - J\|, \qquad \|J_\mu - T_\mu J\| \le \frac{\alpha}{1-\alpha}\|T_\mu J - J\|.$$

Proof. †
Parts $(i)-(iv)$ are proven in [Ber18], so we will only write out the proof of point (v) here. The proof of (v) is similar to the one for (iv), but we will write

it out in more detail compared with the proof of part (iv) in [Ber18]. Assume that $J \in \mathcal{B}(X)$ and that $\mu \in \mathcal{M}$. We also assume that T_μ is a contraction, which in turn implies that T is a contraction from Proposition 4.1.4. We start by showing that the inequality

$$\|J_\mu - J\| \leq \frac{1}{1-\alpha}\|T_\mu J - J\|$$

holds. From part (ii) of the proposition we have that

$$\begin{aligned}
\|J_\mu - J\| &= \lim_{k\to\infty} \|T_\mu^k J - J\| \\
&= \lim_{k\to\infty} \|T_\mu^k J - T_\mu^{k-1} J + T_\mu^{k-1} J - \cdots - T_\mu J + T_\mu J - J\| \\
&\leq \lim_{k\to\infty} \|T_\mu^k J - T_\mu^{k-1} J\| + \|T_\mu^{k-1} J - T_\mu^{k-2} J\| + \cdots + \|T_\mu J - J\| \\
&\leq \lim_{k\to\infty} \alpha^{k-1}\|T_\mu J - J\| + \alpha^{k-2}\|T_\mu J - J\| + \cdots + \|T_\mu J - J\| \\
&= \lim_{k\to\infty} \sum_{i=0}^{k-1} \alpha^i \|T_\mu J - J\| = \frac{1}{1-\alpha}\|T_\mu J - J\|,
\end{aligned}$$

where the last inequality holds due to the contraction property of T_μ. To prove the second inequality, note that we have

$$\|J_\mu - T_\mu J\| \leq \frac{1}{1-\alpha}\|T_\mu(T_\mu J) - T_\mu J\| \leq \frac{\alpha}{1-\alpha}\|(T_\mu J) - J\|,$$

by using the first inequality with $T_\mu J$. ■

We then proceed with a result that guarantees that the fixed point of T, which we denote by J^*, is equal to the cost function found by minimizing J_μ over the set of stationary policies $\mathcal{M}$ when we have both monotonicity and contraction. In fact, we will see that J^* equals the minimum of J_π over all policies $\pi \in \Pi$, i.e. the optimal cost function.

Proposition 4.2.2 (Optimality of stationary policies [Ber18])**.** *If the operators T_μ, $\mu \in \mathcal{M}$ and T has the monotonicity property and are contractions on $\mathcal{B}(X)$, then*

$$J^*(x) = \inf_{\pi\in\Pi} J_\pi(x) = \inf_{\mu\in\mathcal{M}} J_\mu(x) \text{ for all } x \in X,$$

where J^* is the fixed point of T. We also have that for each $\epsilon > 0$ there exists some stationary policy $\mu_\epsilon \in \mathcal{M}$ that satisfies

$$J^* \leq J_{\mu_\epsilon} \leq J^* + \epsilon \tag{4.8}$$

pointwise.

Our proof of Proposition 4.2.2 follows the one given in [Ber18] as well as a section in the same book showing the optimality over nonstationary policies. We choose to include the proof here anyway as it is of upmost importance for the theory covered in this work. The proof we present also include somewhat more details than the original from [Ber18]. However, before we present the proof of Proposition 4.2.2 we need the following lemmas.

Lemma 4.2.3. †
The following inequality holds:

$$J^* \leq T_\mu^k J^*.$$

Proof of Lemma 4.2.3. We prove the lemma using induction on $k \in \mathbb{N}$. For $k = 1$ it is clear from the definition of T and T_μ (Equations (4.1) and (4.2)) that

$$J^* = TJ^* \leq T_\mu J^*,$$

as J^* is the fixed point of T. We now assume that the statement holds for $k = n$. Then we have that

$$J^* \leq T_\mu^n J^*,$$

and by then applying T_μ on both sides of the equation we see that

$$T_\mu J^* \leq T_\mu^{n+1} J^*.$$

Then, by the using the same argument as for the case $k = 1$ we see that

$$J^* \leq T_\mu J^* \leq T_\mu^{n+1} J^*.$$

We can then conclude that the lemma holds. ■

Lemma 4.2.4. †
Assume that the mapping H is monotone. Then, for any policy $\pi = \{\mu_0, \mu_1, \ldots, \mu_k, \ldots\} \in \Pi$ and any initial cost function $\bar{J} \in \mathcal{B}(X)$ we have that

$$\limsup_{k\to\infty} (T_{\mu_0} T_{\mu_1} \cdots T_{\mu_k} \bar{J})(i) \geq \lim_{k\to\infty} (T^{k+1} \bar{J})(i) \text{ for all } i \in X.$$

Proof. We prove the claim with help of induction by showing that the inequality holds for each $k \in \mathbb{N}$. Let $\pi = \{\mu_0, \mu_1, \ldots, \mu_k, \ldots\} \in \Pi$ be an arbitrary policy. For $k = 0$ and any cost function $\bar{J} \in \mathcal{B}(X)$ we have that

$$(T_{\mu_0} \bar{J})(i) = H(i, \mu_0(i), \bar{J}) \geq \min_{u \in U(i)} H(i, u, \bar{J}) = (T\bar{J})(i) \text{ for all } i \in X.$$

We then assume that for any $\bar{J} \in \mathcal{B}(X)$

$$(T_{\mu_0} T_{\mu_1} \cdots T_{\mu_n} \bar{J})(i) \geq (T^{n+1} \bar{J})(i) \text{ for all } i \in X$$

holds. We then have for $k = n + 1$ and any $\bar{J} \in \mathcal{B}(X)$ that

$$\begin{aligned}(T_{\mu_0} T_{\mu_1} \cdots T_{\mu_n} T_{\mu_{n+1}} \bar{J})(i) &= ((T_{\mu_0} T_{\mu_1} \cdots T_{\mu_n})(T_{\mu_{n+1}} \bar{J}))(i) \\ &\geq (T^{n+1}(T_{\mu_{n+1}} \bar{J}))(i) \\ &\geq (T^{n+1}(T\bar{J}))(i) = (T^{n+2} \bar{J})(i),\end{aligned}$$

where the first inequality follows from the assumption made above, while the second follows from the monotonicity property of H and the fact that $T_{\mu_{n+1}} \bar{J} \geq T\bar{J}$ pointwise. We thus have that the inequality holds for each $k \in \mathbb{N}$, therefore it also holds at the limit and the statement is proved. ■

We are then ready to present the proof of Proposition 4.2.2.

Proof of Proposition 4.2.2. †
This proof is based on the proof of Proposition 2.1.2 and a section on optimality over nonstationary policies in [Ber18], but with some more details. We begin by showing that the right inequality of Equation (4.8) holds. We have from Proposition 4.2.1(v) that

$$\|J_\mu - J\|_v \leq \frac{1}{1-\alpha}\|T_\mu J - J\|_v. \tag{4.9}$$

Then, by inserting J^* in place of J in Equation (4.9) we have that

$$\|J_\mu - J^*\|_v \leq \frac{1}{1-\alpha}\|T_\mu J^* - J^*\|_v. \tag{4.10}$$

Note that we then for each $\epsilon > 0$ are able to find some policy $\mu_\epsilon \in \mathcal{M}$ that satisfy

$$\|J_{\mu_\epsilon} - J^*\|_v \leq \frac{1}{1-\alpha}\|T_{\mu_\epsilon} J^* - J^*\|_v \leq \epsilon,$$

as

$$\begin{aligned}\|T_{\mu_\epsilon} J^* - J^*\|_v &= \|T_{\mu_\epsilon} J^* - TJ^*\|_v \\ &= \sup_{x\in X} \frac{|H(x,\mu_\epsilon(x),J^*) - \inf_{u\in U(x)} H(x,u,J^*)|}{v(x)},\end{aligned} \tag{4.11}$$

which imply that we are able to find some $\mu_\epsilon \in \mathcal{M}$ with $\mu_\epsilon(x) \in U(x)$ such that $\sup_{x\in X}|H(x,\mu_\epsilon(x),J^*) - \inf_{u\in U(x)} H(x,u,J^*)|$ is sufficiently small. Observe that sufficiently small in this context is $(1-\alpha)\epsilon v(x)$ as we from combining Equations (4.10) and (4.11) see that

$$\begin{aligned}\|J_{\mu_\epsilon} - J^*\|_v &\leq \frac{1}{1-\alpha}\|T_{\mu_\epsilon} J^* - J^*\|_v \\ &= \frac{1}{1-\alpha}\sup_{x\in X} \frac{|H(x,\mu_\epsilon(x),J^*) - \inf_{u\in U(x)} H(x,u,J^*)|}{v(x)} \\ &\leq \frac{1}{1-\alpha}\frac{(1-\alpha)\epsilon v(x)}{v(x)} \\ &= \epsilon.\end{aligned}$$

In other words, we find μ_ϵ by for every $x \in X$ setting

$$\mu_\epsilon(x) = \arg\min_{u\in U(x)} H(x,u,J^*)$$

while allowing an error of $(1-\alpha)\epsilon v(x)$. Thus we have that

$$J_{\mu_\epsilon}(x) \leq J^*(x) + \epsilon \text{ for all } x \in X.$$

This implies that

$$\inf_{\mu\in\mathcal{M}} J_\mu(x) \leq J^*(x).$$

We then proceed with showing that the opposite inequity holds. Note that from the definitions of T and T_μ (Equations (4.1) and (4.2)) we have that $TJ^* \leq T_\mu J^*$. Then, as J^* is the cost function that satisfies $J^* = TJ^*$, we have that

$$J^* = TJ^* \leq T_\mu J^*. \tag{4.12}$$

We can then apply T_μ on both sides of Equation (4.12) $k-1$ times. Then

$$T_\mu^{k-1} J^* \leq T_\mu^k J^*,$$

and from Lemma 4.2.3 we have that

$$J^* \leq T_\mu^k J^*.$$

This implies that we have

$$J^* \leq \lim_{k\to\infty} T_\mu^k J^* = J_\mu,$$

showing that the other side of the inequality,

$$J^*(x) \leq \inf_{\mu\in\mathcal{M}} J_\mu(x),$$

holds. This also implies that the left inequality in Equation (4.8) holds. As we then have shown that both Equation (4.8) and

$$J^*(x) = \inf_{\mu\in\mathcal{M}} J_\mu(x),$$

holds, it only reminds to show that we have

$$J(x)^* = \inf_{\pi\in\Pi} J_\pi(x). \tag{4.13}$$

As $\mathcal{M} \subset \Pi$ we immediately have that

$$J^*(x) = \inf_{\mu\in\mathcal{M}} J_\mu(x) \geq \inf_{\pi\in\Pi} J_\pi(x).$$

To get the reverse inequality recall that

$$J_\pi(i) = \limsup_{k\to\infty} (T_{\mu_0} T_{\mu_1} \cdots T_{\mu_k})(i),$$

for some policy $\pi = \{\mu_0, \mu_1, \ldots, \mu_k, \ldots\} \in \Pi$. Then, from Lemma 4.2.4 we have that

$$J_\pi(x) = \limsup_{k\to\infty} (T_{\mu_0} T_{\mu_1} \cdots T_{\mu_k} J) \geq \lim_{k\to\infty} (T^{k+1} J)(x) = J^*(x),$$

as we from Proposition 4.2.1(ii) have that $\lim_{k\to\infty}(T^{k+1}J)(x) = J^*(x)$. Thus we see that Equation (4.13) holds and the result is proven. ■

Implications of results

The results we have introduced in Sections 4.1 and 4.2 have some important implications that we are going to use a lot in the coming chapters. So far we have shown that when we have some mapping $H : X \times U \times \mathcal{R}(X) \to \mathbb{R}$ that is monotone, and that is such that the induced operator T_μ is a contraction for each $\mu \in \mathcal{M}$, then we have that:

- The operator T is a contraction as well (by Proposition 4.1.4),

- For each $\mu \in \mathcal{M}$ we have that T_μ has a unique fixed point J_μ (by Proposition 4.2.1),

- The operator T has a unique fixed point that we denote by J^* (by Proposition 4.2.1),

- The identity $J^* = \inf_{\mu \in \Pi} J_\pi = \inf_{\mu \in \mathcal{M}} J_\mu$ holds (by Proposition 4.2.2).

The strength in the theory presented in Chapter 4 is that if we have some optimal control problem, and are able to find an expression for Bellman's equation, then we can easily define the mapping H. If we then are able to show that the mapping H is monotone, and that the induced operator T_μ is a contraction for each stationary policy $\mu \in \mathcal{M}$, then that implies that T is a contraction as well. This again tell us that the operator T has a fixed point, which we denote J^*. We then know that $J^* = \inf_{\mu \in \Pi} J_\pi = \inf_{\mu \in \mathcal{M}} J_\mu$ holds, which imply two things of importance. First, the optimal cost function is indeed equal to the fixed point of T. Second, the optimal policy can be found among the stationary policies, which means that the subspace of stationary policies $\mathcal{M}$ that we need to consider is a lot smaller than the space Π containing all policies.

Let us now take a quick look at the intuition behind defining the mapping H, and the operators T and T_μ. The mapping H tell us something about how the dynamic system evolve, as it given some state $x \in X$, control $u \in U(x)$ and some estimate of the cost function J give us the estimated cost-to-go when we start in state x, use the control u and the cost-to-go values from the possible successor states $x' \in X'_{x,u} = \{x \in X \mid f(x,u,w) = x \text{ for some } w \in W(x,u)\}$ are determined by $J(x')$ for each possible successor state $x' \in X'_{x,u}$. We then have that the operator T_μ, given some cost function J, finds the estimated cost-to-go in each state $x \in X$ when we start the control process in state x, use the control $u \in U(x)$ determined by the policy μ, i.e. $u = \mu(x)$, and where J is the estimated cost-to-go in each possible successor state $x' \in X'$. In other words, the function $(T_\mu J)(x)$ gives the cost-to-go from state x, under control $\mu(x)$ where J is the estimated cost-to-go from the possible successor states x'.

Let us now take a quick look at some examples showing why the monotonicity and contraction property is important. The first example is taken from [Ber18], but we have added some details and made some remarks the author does not make in [Ber18].

Example 4.2.5 (Importance of Monotonicity property). †
This example is taken from [Ber18], but we write it out in more detail. Assume that $X = \{x_1, x_2\}$, and that $U = \{u_1, u_2\}$. We define the mapping H by

$$H(x_1,u,J) = \begin{cases} -\alpha J(x_2) & \text{if } u = u_1, \\ -1 + \alpha J(x_1) & \text{if } u = u_2, \end{cases} \quad H(x_2,u,J) = \begin{cases} 0 & \text{if } u = u_1, \\ B & \text{if } u = u_2, \end{cases}$$

where B is some positive number. Note first that the mapping H is not monotone. Let J and J' be two cost functions with

$$J(x_1) = 0, J(x_2) = 0, J'(x_1) = 1 \text{ and } J'(x_2) = 1,$$

so we have $J \leq J'$ pointwise. We then see that

$$H(x_1, u_1, J) = -\alpha J(x_2) = -\alpha \cdot 0 = 0,$$

while

$$H(x_1, u_1, J') = -\alpha J(x_2) = -\alpha \cdot 1 = -\alpha.$$

Thus we see that $J \leq J'$ pointwise, while $H(x_1, u_1, J) \geq H(x_1, u_1, J')$, showing that H indeed is not monotone.

It is then possible to find that the fixed point J^* of the operator T induced by the mapping H is given by

$$J^*(x_1) = -\frac{1}{1-\alpha},\ J^*(x_2) = 0,$$

and indeed we see that

$$\begin{aligned}
(TJ^*)(x_1) &= \min_{u \in \{u_1, u_2\}} H(x_1, u, J^*) \\
&= \min\{-\alpha J^*(x_2), -1 + \alpha J^*(x_1)\} \\
&= \min\left\{0, -\frac{1-\alpha}{1-\alpha} - \frac{\alpha}{1-\alpha}\right\} \\
&= \min\left\{0, \frac{1}{1-\alpha}\right\} \\
&= -\frac{1}{1-\alpha},
\end{aligned}$$

and

$$\begin{aligned}
(TJ^*)(x_2) &= \min_{u \in \{u_1, u_2\}} H(x_2, u, J^*) \\
&= \min\{0, B\} \\
&= 0.
\end{aligned}$$

In addition, the, in this case unique, optimal policy (in the sense that $J_{\mu^*} = J^*$) is given by

$$\mu^*(x) = \begin{cases} u_2 & \text{if } x = x_1, \\ u_1 & \text{if } x = x_2. \end{cases}$$

If we now define another policy

$$\mu(x) = \begin{cases} u_1 & \text{if } x = x_1, \\ u_2 & \text{if } x = x_2. \end{cases}$$

we can find that

$$J_\mu(x_1) = -\alpha B,\ J_\mu(x_2) = B.$$

Thus we see that $J_\mu(x_1) < J^*(x_1)$ for sufficiently large B, which means that

$$J^*(x_1) = \inf_{\mu \in \Pi} J_\mu(x_1)$$

does not hold.

The next example we are looking at considers the case where the operators T_μ and T are not contractions.

Example 4.2.6 (Importance of Contraction property). †
In this example we let $X = \{x\}$, and $U = \{u\}$. We then let the mapping H be defined by

$$H(x, u, J) = 1 + \alpha J(x),$$

where $\alpha \in (0, 1]$. We also let $v(x) = 1$ for all $x \in X$. In order for the mapping T to be a contraction we need for each pair $J, J' \in \mathcal{R}(X)$ to have

$$\|TJ - TJ'\| \leq \rho \|J - J'\|,$$

with $\rho \in (0, 1)$. Note that

$$\begin{aligned}(TJ)(x) - (TJ')(x) &= 1 + \alpha J(x) - (1 + \alpha J'(x)) \\ &= 1 - 1 + \alpha J(x) - \alpha J'(x) \\ &= \alpha(J(x) - J'(x)) \\ &= \alpha v(x) \frac{J(x) - J'(x)}{v(x)},\end{aligned}$$

implying that

$$\frac{(TJ)(x) - (TJ')(x)}{v(x)} = \alpha \frac{J(x) - J'(x)}{v(x)}.$$

Then, by taking the norm on both sides we see that

$$\|TJ - TJ'\| = \alpha \|J - J'\| \tag{4.14}$$

which implies that T is a contraction whenever $\alpha < 1$. Lets now assume that $\alpha = \frac{1}{2}$, and let $\pi = \{\mu_0, \mu_1, \ldots\}$ where $\mu_k(x) = u$ for all $k \in \mathbb{N}$. We then have that

$$\begin{aligned}J_\pi(x) &= \limsup_{k\to\infty} (T_{\mu_0} T_{\mu_1} \cdots T_{\mu_{k-1}} \bar{J})(x) \\ &= \limsup_{k\to\infty} \sum_{n=0}^{k-1} \alpha^n \\ &= \limsup_{k\to\infty} \sum_{n=0}^{k-1} \frac{1}{2^n} \\ &= 2,\end{aligned}$$

where $\bar{J}(x) = 0$. It is clear that the operator T is equal to the operator T_{μ_k} for all $k \in \mathbb{N}$ as we for the problem considered here only have one state and one action, therefore $J_\pi(x) = J^*(x) = 2$. Let us now set $\alpha = 1$, then we see from Equation (4.14) that T is not a contraction. The lack of the contraction property results in

$$\begin{aligned}J_\pi(x) &= \limsup_{k\to\infty} (T_{\mu_0} T_{\mu_1} \cdots T_{\mu_{k-1}} \bar{J})(x) \\ &= \limsup_{k\to\infty} \sum_{n=0}^{k-1} \alpha^n\end{aligned}$$

$$= \limsup_{k \to \infty} \sum_{n=0}^{k-1} 1^n$$
$$= \infty,$$

and thus we see that the sum diverges. We are therefore not able to determine the optimal cost for this problem.

The two examples above illustrate why the two assumptions we introduce in Section 4.1 are necessary. Let us now continue by looking at general methods for finding the optimal cost function J^* and optimal policies μ^*.

4.3 Finding policies

In this section we are going to take a look at different methods for finding optimal and suboptimal policies. We start with lookahead policies before proceeding with value iteration and then ending with policy iteration.

Lookahead policies

Assume that we have a function $\tilde{J} \in \mathcal{B}(X)$ that is our current best approximation of J^*. We can then obtain a *one-step lookahead* policy $\bar{\mu}$ by letting

$$\bar{\mu}(x) \in \arg\min_{u \in U(x)} H(x, u, \tilde{J}). \tag{4.15}$$

That is, we let the one-step lookahead policy be a policy that minimizes H when we assume that the cost-to-go function is our approximation $\tilde{J}$. Note that this is the same as having $\bar{\mu}$ be a policy such that $T_{\bar{\mu}}\tilde{J} = T\tilde{J}$, note also that $\bar{\mu}$ may not be unique. We then state a proposition from [Ber18] that assures some bounds on the cost function $J_{\bar{\mu}}$.

Proposition 4.3.1 (One-step lookahead error bounds [Ber18])**.** *Let the operator T_μ be a contraction for each $\mu \in \mathcal{M}$. We also let $\bar{\mu}$ be a one-step lookahead policy, that is, $T_{\bar{\mu}}\tilde{J} = T\tilde{J}$. We then have that the following bounds hold:*

$$\|J_{\bar{\mu}} - T\tilde{J}\| \leq \frac{\alpha}{1-\alpha}\|T\tilde{J} - \tilde{J}\|,$$

$$\|J_{\bar{\mu}} - J^*\| \leq \frac{2\alpha}{1-\alpha}\|\tilde{J} - J^*\|,$$

and

$$\|J_{\bar{\mu}} - J^*\| \leq \frac{2}{1-\alpha}\|T\tilde{J} - \tilde{J}\|.$$

The one-step lookahead can be generalized to *multistep lookahead* policies, but we will not go into the details of that here. Let us proceed with an example showing one-step lookahead policies in practise.

Example 4.3.2 (Finding a lookahead policy, continuation of Example 3.4.3)**.** †

Let us consider a shortest path problem similar to the one we considered in Example 3.4.3, but with some changes in the weights. The graph with the new costs is shown in Figure 4.1.

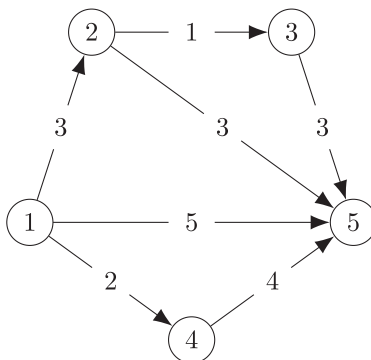

Figure 4.1: Graph of the shortest path problem

Let us begin with the zero-function as our approximation $\tilde{J}$ of J^*. That is, $\tilde{J}(x) = 0$ for all $x \in X = \{1, 2, 3, 4, 5\}$. Note also that here we have $H(x, u, J) = a_{xu} + J(u)$, where a_{xu} is the cost associated with moving from state x to state u. We can then find the one-step lookahead policy. For $x = 1$ we have:

$$\bar{\mu}(1) \in \arg\min_{u \in U(1)} H(1, u, \tilde{J}) = \arg\min_{u \in \{2,4,5\}} a_{1u} + \tilde{J}(u) = \arg\min_{u \in \{2,4,5\}} a_{1u} = 4,$$

thus $\bar{\mu}(1) = 4$. For the other states we have the following:

$$\bar{\mu}(2) = 3,\ \bar{\mu}(3) = 5,\ \bar{\mu}(4) = 5,\ \bar{\mu}(5) = 5.$$

Note that we always have $\bar{\mu}(5) = 5$, as this is the only possibility. We then say that the state is *absorbing*, as we can never escape the state if we at some point ends up there. It is also straight forward to verify that $T_{\bar{\mu}}\tilde{J} = T\tilde{J}$.

Value iteration

We then proceed to take a look at the value iteration (VI) algorithm for finding J^*. If T is assumed to be a contraction we have that T has a fixed point J^*, which is the optimal cost function. Thus for an arbitrary chosen $J_0 \in \mathcal{B}(X)$ we have that the sequence

$$J_0,\ TJ_0,\ T^2J_0, \ldots,$$

converges towards J^*. We can thus use the value iteration method to find an approximation $\tilde{J}$ of J^*, and then find the one-step lookahead policy $\bar{\mu}$ as defined by (4.15). The next proposition tell us that when the set of stationary policies $\mathcal{M}$ is finite we can in fact use the procedure just described to obtain an optimal policy.

Proposition 4.3.3 (Optimal policy using VI [Ber18])**.** *Assume that the operators T_μ is a contraction for all $\mu \in \mathcal{M}$. Let also $J_0 \in \mathcal{B}(X)$. Then if the set $\mathcal{M}$ is finite we have that there exists some $n \in \mathbb{N}$ such that every possible one-step lookahead policy μ^* found by using T^nJ_0 as our approximation of J^*, that is μ^* is such that $T_{\mu^*}T^nJ_0 = T^{n+1}J_0$, have the property that*

$$J_{\mu^*} = J^*.$$

Proof. †
This proof is taken from [Ber18], but we have added some details. As $\mathcal{M}$ is assumed to be finite, we have that the subset

$$\tilde{\mathcal{M}} = \{\mu \mid \mu \in \mathcal{M},\ J_\mu \neq J^*\} \subset \mathcal{M}$$

is finite. Thus

$$\inf_{\mu \in \tilde{\mathcal{M}}} \|J_\mu - J^*\| > 0.$$

Then, from Proposition 4.3.1 we see that

$$\|J_{\bar{\mu}} - J^*\| \leq \frac{2\alpha}{1-\alpha}\|\tilde{J} - J^*\|,$$

for all one-step lookahead policies $\bar{\mu}$ found by using $\tilde{J}$ as an approximation to J^*, i.e. for all $\bar{\mu} \in \mathcal{M}$ such that $T_{\bar{\mu}}\tilde{J} = T\tilde{J}$. Then there exist some $\beta > 0$ such that whenever $\|\tilde{J} - J^*\| \leq \beta$ and $T_{\bar{\mu}}\tilde{J} = T\tilde{J}$ we have that

$$\|J_{\bar{\mu}} - J^*\| \leq \frac{2\alpha}{1-\alpha}\beta < \inf_{\mu \in \tilde{\mathcal{M}}} \|J_\mu - J^*\|,$$

which then imply that $\bar{\mu} \notin \tilde{\mathcal{M}}$. Then, as T_μ is a contraction for all $\mu \in \mathcal{M}$ we have from Proposition 4.1.4 that T also is a contraction, hence

$$\|T^n J_0 - J^*\| \leq \alpha^n \|J_0 - J^*\|. \tag{4.16}$$

It then follows that for sufficiently large $n \in \mathbb{N}$ we have that $\|T^n J_0 - J^*\| \leq \beta$, and thus, from the arguments above, we have that the one-step lookahead policy μ^* created by having $T^n J_0$ as an approximation of J^*, i.e. $T_{\mu^*}T^n J_0 = T^{n+1}J_0$, satisfy

$$J_{\mu^*} = J^*,$$

as we needed to show. ■

The inequality (4.16) in the proof above follows from the fact that T is a contraction and that $TJ^* = J^*$, as we then have that

$$\|TJ_0 - J^*\| = \|TJ_0 - TJ^*\| \leq \alpha\|J_0 - J^*\|,$$

thus

$$\begin{aligned}\|T(TJ_0) - J^*\| &= \|T(TJ_0) - TJ^*\| \leq \alpha\|TJ_0 - J^*\| \\ &= \alpha\|TJ_0 - TJ^*\| \leq \alpha^2\|J_0 - J^*\|.\end{aligned}$$

By repeating the argument, we see that the inequality

$$\|T^n J_0 - J^*\| \leq \alpha^n \|J_0 - J^*\|$$

holds true.

Let us now look at an example where we use value iteration to solve a deterministic dynamic programming problem.

Example 4.3.4 (Value iteration, continuation of Example 4.3.2)**.** †

Let us take a look at the same dynamic programming problem that we considered in Example 4.3.2, but let us now try to find the optimal policy using value iteration. Let us again start with J_0 being the zero-function. We then start by finding TJ_0. For $x = 1$ we have that

$$(TJ_0)(1) = \min_{u \in U(1)} H(1, u, J_0) = \min_{u \in \{2,4,5\}} a_{1u} + J_0(u) = \min\{3, 2, 5\} = 2.$$

We also see that for the other states we get

$$(TJ_0)(2) = 1,\ (TJ_0)(3) = 3,\ (TJ_0)(4) = 4,\ (TJ_0)(5) = 0.$$

We then proceed with finding $T^2 J_0$, again we look carefully at the case when $x = 1$. We see that

$$\begin{aligned}(T^2 J_0)(1) &= \min_{u \in U(1)} H(1, u, TJ_0) = \min_{u \in \{2,4,5\}} a_{1u} + TJ_0(u) \\ &= \min\{3 + (TJ_0)(2), 2 + (TJ_0)(4), 5 + (TJ_0)(5)\} \\ &= \min\{3 + 1, 2 + 4, 5 + 0\} = 4.\end{aligned}$$

For the remaining states we have

$$\begin{aligned}(T^2 J_0)(2) &= \min\{1 + 3, 3 + 0\} = \min\{4, 3\} = 3, \\ (T^2 J_0)(3) &= \min\{3 + 0\} = 3 = (TJ_0)(3) \\ (T^2 J_0)(4) &= \min\{4 + 0\} = 4 = (TJ_0)(4), \\ (T^2 J_0)(5) &= \min\{0\} = 0 = (TJ_0)(5).\end{aligned}$$

We then continue with calculate the values of $T^3 J_0$. We look at the case when $x = 1$, giving

$$\begin{aligned}(T^3 J_0)(1) &= \min_{u \in U(1)} H(1, u, T^2 J_0) = \min_{u \in \{2,4,5\}} a_{1u} + T^2 J_0(u) \\ &= \min\{3 + (T^2 J_0)(2), 2 + (T^2 J_0)(4), 5 + (T^2 J_0)(5)\} \\ &= \min\{3 + 3, 2 + 4, 5 + 0\} = 5.\end{aligned}$$

For $x = 2$ we have

$$(T^3 J_0)(2) = \min\{1 + 3, 3 + 0\} = \min\{4, 3\} = 3 = (T^2 J_0)(2).$$

We also note that $(T^3 J_0)(x) = (T^2 J_0)(x) = (TJ_0)(x)$ for $x = 3, 4, 5$. In the next step we only check if the value for $x = 1$ has changed, as the value for the other x-values has not changed between the two former steps. We then find

$$\begin{aligned}(T^4 J_0)(1) &= \min_{u \in U(1)} H(1, u, T^3 J_0) = \min_{u \in \{2,4,5\}} a_{1u} + T^3 J_0(u) \\ &= \min\{3 + (T^3 J_0)(2), 2 + (T^3 J_0)(4), 5 + (T^3 J_0)(5)\} \\ &= \min\{3 + 3, 2 + 4, 5 + 0\} = 5 = (T^3 J_0)(1).\end{aligned}$$

Thus, we now have $T(T^3 J_0) = T^3 J_0$, so we see that $T^3 J_0$ is the optimal cost function, i.e. $J^* = T^3 J_0$. We can then find an optimal policy by finding a one-step lookahead policy found by solving

$$\bar{\mu}(x) \in \arg\min_{u \in U(x)} H(x, u, T^3 J_0) \text{ for all } x \in X.$$

Note first that

$$T^3J_0(1) = 5,\ T^3J_0(2) = 3,\ T^3J_0(3) = 3,\ T^3J_0(4) = 4,\ T^3J_0(5) = 0.$$

We have that

$$\begin{aligned}
\bar{\mu}(1) &\in \arg\min_{u\in U(1)} H(1,u,T^3J_0) = \arg\min_{u\in\{2,4,5\}} a_{1u} + T^3J_0(u) = 5,\\
\bar{\mu}(2) &\in \arg\min_{u\in U(2)} H(2,u,T^3J_0) = \arg\min_{u\in\{3,5\}} a_{2u} + T^3J_0(u) = 5,\\
\bar{\mu}(3) &\in \arg\min_{u\in U(3)} H(3,u,T^3J_0) = \arg\min_{u\in\{5\}} a_{3u} + T^3J_0(u) = 5,\\
\bar{\mu}(4) &\in \arg\min_{u\in U(4)} H(4,u,T^3J_0) = \arg\min_{u\in\{5\}} a_{4u} + T^3J_0(u) = 5,\\
\bar{\mu}(5) &\in \arg\min_{u\in U(5)} H(5,u,T^3J_0) = \arg\min_{u\in\{5\}} a_{5u} + T^3J_0(u) = 5.
\end{aligned}$$

Thus we have that $\mu^*(x) = \bar{\mu}(x) = 5$ for all $x \in X = \{1,2,3,4,5\}$.

Policy iteration

We end this chapter by looking at the policy iteration (PI) algorithm. At each iteration $k \in \mathbb{N}$ of the algorithm we start with a current policy, denoted μ^{k-1}, which is an estimate of an optimal policy μ^*. In each iteration we start with a *policy evaluation* step where we find the cost function $J_{\mu^{k-1}}$ which is the unique solution of the equation

$$J_{\mu^{k-1}} = T_{\mu^{k-1}} J_{\mu^{k-1}}.$$

We then move on to the *policy improvement* step, which is the last step of the iteration, where we obtain an improved policy μ^k which is a one-step lookahead policy found by using the cost function $J_{\mu_{k-1}}$ as our approximation of J^*. That is, μ^k satisfies

$$T_{\mu^k} J_{\mu^{k-1}} = T J_{\mu^{k-1}}.$$

The next proposition ensures that policy iteration converges under the assumption of T and T_μ, $\mu \in \mathcal{M}$ satisfying the monotonicity property and the last being a contraction.

Proposition 4.3.5 (Convergence of PI [Ber18])**.** *Assume that the operators T and T_μ, $\mu \in \mathcal{M}$ satisfy the monotonicity property and that T_μ, $\mu \in \mathcal{M}$ is a contraction. Then, let $\{\mu^k\}$ be a sequence of policies generated by policy iteration. Then, for any $k \in \mathbb{N}$ we have that $J_{\mu^{k+1}} \leq J_{\mu^k}$, where $J_{\mu^{k+1}} = J_{\mu^k}$ if and only if $J_{\mu^k} = J^*$. We also have that*

$$\lim_{k\to\infty} \|J_{\mu^k} - J^*\| = 0.$$

Furthermore, if the set of policies is finite we have $J_{\mu^k} = J^*$ for some k.

For a proof of Proposition 4.3.5 we refer the curious reader to [Ber18]. Note that the previous Proposition 4.3.5 only guarantees that $J_{\mu_k} = J^*$ for some $k \in \mathbb{N}$ if the set of policies is finite. However, under some strengthened assumptions we are actually able to guarantee that $J_{\bar{\mu}} = J^*$ where $\bar{\mu}$ is any limit point of the sequence $\{\mu^k\}$ of policies obtained with the policy iteration algorithm even when $\mathcal{M}$ is not finite.

Proposition 4.3.6 (Limit point PI [Ber18]). *Assume that the operators T and T_μ, $\mu \in \mathcal{M}$ satisfy the monotonicity property and that T_μ, $\mu \in \mathcal{M}$ is a contraction. Assume also the following*

(i) *The number of states is finite, i.e. $X = \{1, \dots, n\}$ for some $n \in \mathbb{N}$.*

(ii) *Each subset $U(x) \subseteq U$, $x \in X$ is a compact subset of $\mathbb{R}^n$.*

(iii) *For each $x \in X$ we have that the function $H(x, \cdot, \cdot)$ is continuous over $U(x) \times \mathbb{R}^n$.*

If we then let $\{\mu^k\}$ be a sequence of policies generated by policy iteration we have that $J_{\bar{\mu}} = J^*$ for each limit point $\bar{\mu}$ of the sequence $\{\mu^k\}$.

The proof of Proposition 4.3.6 can be found in [Ber18]. Let us now take a look at an example using policy iteration.

Example 4.3.7 (Policy iteration, continuation of Example 4.3.4). †

Let us again take a look at the shortest path problem of trying to find the shortest path to node 5 from every other node in the graph depicted in Figure 4.1, which we looked at in Example 4.3.4. We will here demonstrate how we can use policy iteration to find the optimal paths. We start with the initial policy $\mu^0(x) = \min_{u \in U(x)} u$. Thus we see that

$$\mu^0(1) = 2,\ \mu^0(2) = 3,\ \mu^0(3) = 5,\ \mu^0(4) = 5,\ \mu^0(5) = 5.$$

We then start with the policy evaluation, that is finding the cost function J_{μ^0}. From the graph in Figure 4.1 it is easy to see that the cost function from following the policy μ^0 is

$$J_{\mu^0}(1) = 7,\ J_{\mu^0}(2) = 4,\ J_{\mu^0}(3) = 3,\ J_{\mu^0}(4) = 4,\ J_{\mu^0}(5) = 0.$$

We then proceed to the policy improvement step, where we are supposed to find the improved policy μ^1 such that $T_{\mu^1} J_{\mu^0} = T J_{\mu^0}$. We do this by finding a one-step lookahead policy. We have that

$$\begin{aligned}
\mu^1(1) &\in \arg\min_{u \in U(1)} H(1, u, J_{\mu^0}) = \arg\min_{u \in \{2,4,5\}} a_{1u} + J_{\mu^0}(u) = 5,\\
\mu^1(2) &\in \arg\min_{u \in U(2)} H(2, u, J_{\mu^0}) = \arg\min_{u \in \{3,5\}} a_{2u} + J_{\mu^0}(u) = 5,\\
\mu^1(3) &\in \arg\min_{u \in U(3)} H(3, u, J_{\mu^0}) = \arg\min_{u \in \{5\}} a_{3u} + J_{\mu^0}(u) = 5,\\
\mu^1(4) &\in \arg\min_{u \in U(4)} H(4, u, J_{\mu^0}) = \arg\min_{u \in \{5\}} a_{4u} + J_{\mu^0}(u) = 5,\\
\mu^1(5) &\in \arg\min_{u \in U(5)} H(5, u, J_{\mu^0}) = \arg\min_{u \in \{5\}} a_{5u} + J_{\mu^0}(u) = 5.
\end{aligned}$$

Thus we have that $\mu^1(x) = 5$ for all $x \in X = \{1, 2, 3, 4, 5\}$. We know from Example 4.3.4 that this is the optimal policy, so we see that policy iteration found the optimal policy after one iteration.

CHAPTER 5

Infinite horizon dynamic programming

This chapter is based on Chapter 4 in [Ber19]. We will here take the abstract setting from Chapter 4 where we allowed infinite horizons and make it more concrete by looking at some examples, just like we did earlier for the finite horizon case in Chapter 3. The examples we will look at in this chapter is discounted problems and stochastic shortest path (SSP) problems. Previously in Section 3.4 we looked at deterministic shortest path problems where we were given a policy μ and knew exactly how many steps we needed to reach the end state. For the stochastic version the best we can do for some policies (or all, depending on the problem) is to assign a probability to the event of the policy reaching the end state after $N \in \mathbb{N}$ steps. The transition from the finite horizon setting of Chapter 3 to the infinite horizon setting means that we no longer try to minimize

$$J_\pi(x_0) = E\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right],$$

over the set Π of all policies for some $N \in \mathbb{N}$, but instead look at what happens when we allow $N \to \infty$, i.e. we try to minimize the cost given by

$$J_\pi(x_0) = \lim_{N\to\infty} E\left[\sum_{k=0}^{N-1} \alpha^k g_k(x_k, \mu_k(x_k), w_k)\right],$$

over Π, where $\alpha \in (0, 1]$ is the discount factor, $x_k \in X$ is the state at stage k, $w_k \in D$ is the disturbance at stage k, and $\mu_k(x_k) \in U(x_k)$ is the control chosen at stage k dictated by the function $\mu_k : X \to U$. To make the analysis easier we assume that the state space X, control space U and disturbance space D are all finite. We also introduce assumptions that makes sure that the additional cost after stage k, for some $k \in \mathbb{N}$, is zero when the discount factor α is 1. We begin this chapter by looking at the SSP problems, which is the class of problems where the discount factor has the value $\alpha = 1$.

5.1 Stochastic shortest path problems

As we assume that the number of states is a finite number $n \in \mathbb{N}$ we can label them with the first n integers, giving $X = \{1, 2, \ldots, n, t\}$, where t is the

terminal node. We can also rewrite the cost at stage k when at state $i \in X$ and choosing control $u \in U(i)$ from

$$\underset{w \in D_k}{E}[g(i,u,w)] = \sum_{w \in D_k} P(w \mid x_k = i, u_k = u)\, g(i,u,w),$$

where $P(w \mid x_k = i,\, u_k = u)$ is the probability of the random noise being w when at state i and picking control u, to

$$\sum_{j \in X} p_{ij}(u)\, g(i,u,j),$$

where $p_{ij}(u)$ is the probability of ending up at state j when at state i and choosing control u also known as the transition probability from i to j under control u, in other words

$$p_{ij}(u) = P(\{w \mid f(i,u,w) = j\} \mid x_k = i,\, u_k = u). \tag{5.1}$$

Note that we have now already made some further simplifications if we compare with the similar transition probabilities introduced in Section 3.1. We here assume that the transition probabilities are stationary, i.e. that the probability of going from a state i to another state j when choosing control u does not change over time. This means that the function f, which given the current state i, an control u and a disturbance w, giving us the next state j, is constant over time. We also assume that transitioning from a state i to some state j under the control u always give the cost $g(i,u,j)$. With these simplifying assumptions, and the transition probabilities as defined by Equation (5.1) we can write the Bellman equation (4.3) for this problem, which is taken from [Ber19]. For all states $i \in X$ we have

$$J^*(i) = \min_{u \in U(i)} \left[p_{it}(u) g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + J^*(j)) \right]. \tag{5.2}$$

We also state the corresponding Bellman equation for a stationary policy μ, which for each state $i \in X$ has the form

$$J_\mu(i) = p_{it}(\mu(i)) g(i,\mu(i),t) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + J_\mu(j)). \tag{5.3}$$

The intuition behind the formulation of (5.2) is that the first part of the equation says something about the expected cost if we were to end up at the terminal node after picking the next control, the first part of the sum says something about the expected transition cost of going to state j, while the second part consisting of $p_{ij}(u)J^*(j)$ gives us the expected future cost of ending up in state j. The intuition is the same for (5.3), as the only difference is that we choose the control u by use of the policy μ in Equation (5.3). In order to assure that an optimal cost function J^*, and policy cost functions J_μ exists, and that they are the unique solutions of Equation (5.2) and Equation (5.3) respectively, we need the following assumption taken from [Ber19], which we will adopt for the rest of this section.

Assumption 5.1.1 (Non-zero probability of termination at stage m [Ber19]). We assume that there exist some $m \in \mathbb{N}$ such that for all policies π the following holds true:

$$\rho_\pi = \max_{i=1,\dots,n} P(x_m \neq t \mid x_0 = i, \pi) < 1.$$

That is, there exist some $m \in \mathbb{N}$ such that for each policy π, and each possible initial state i, there is a non-zero probability of reaching the terminal state within m stages.

The consequences of Assumption 5.1.1 is quite important, as it ensures that each policy terminates with probability 1. To see this, note that for each set of m stages our agent has a non-zero probability less than or equal to ρ for not being at the terminal node. Thus, if we are at stage km we have that the probability of not having reached a terminal node is the same as the joint probability of not having reached the terminal node for the k preceding sets of m moves given that we condition on not having reached a terminal node in the preceding set of m stages. In other words,

$$\begin{aligned} P(x_{km} \neq t \mid x_0 = i, \pi) &= \prod_{i=1}^{k} P(x_{im} \neq t \mid x_{(i-1)m} \neq t, x_0 = i, \pi) \\ &= \prod_{i=1}^{k} \rho_\pi \leq \prod_{i=1}^{k} \rho = \rho^k, \end{aligned} \tag{5.4}$$

where we have used Assumption 5.1.1 and let

$$\rho = \max_\pi \rho_\pi.$$

We can then conclude that the limit giving the total cost exists and that it is finite. Let us now take a look at the Bellman operators. In order to be aligned with the theory presented in Chapter 4 we introduce the mapping $H : X \times U \times \mathcal{R}(X) \to \mathbb{R}$, where $\mathcal{R}(X)$ is the set of all functions $J : X \to \mathbb{R}$. The function H is defined by

$$H(x, u, J) = p_{it}(u) g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + J(j)). \tag{5.5}$$

Then we define the Bellman operators as in Equations (4.1) and (4.2), i.e.

$$(T_\mu J)(i) = H(i, \mu(i), J),$$

and

$$(TJ)(i) = \min_{u \in U(i)} H(i, u, J),$$

for all $x \in X = \{1, 2, \dots, n, t\}$ and $J \in \mathcal{R}(X)$. We now want to prove results showing that the value iteration algorithm converge to the optimal cost function, and that the optimal cost function J^* indeed is the unique fixed point of Bellman's equation, i.e. we want to show that the optimal cost function satisfies $TJ^* = J^*$ and $J^* = \inf_{\pi \in \Pi} J_\pi$, where Π is the set of all policies. In order to prove these results using the theory in Chapter 4 we need to show that the operator T_μ is a contraction for each $\mu \in \mathcal{M}$ and that the mapping H is monotone. We start with proving the contraction property. In order to ensure the validity of the proof we need the following assumption.

Assumption 5.1.2. Assume that a positive bounded function v_μ exists for each $\mu \in \mathcal{M}$ such that

$$v_\mu(i) = 1 + \sum_{j=1}^{n} p_{ij}(\mu(i)) v_\mu(j) \text{ for each } i \in X,$$

holds.

The assumption may seem a bit arbitrary, but if we think of the value $v_\mu(i)$ as the expected number of steps we need under policy μ to reach a terminal node when starting in state i, we see that the formulation make sense, as the value would be 1 (the next step we are taking) plus the expected future number of steps after the next transition. Recall that we have shown above that the probability of not having reached a terminal state after km steps, where m is the value from Assumption 5.1.1, approaches zero as k increases. Thus the number of steps is finite for every policy, regardless of starting state i. Therefore the value $v_\mu(i)$ should be finite for each $i \in X$, according to the interpolation of the values of v_μ. We now continue with the first proposition.

Proposition 5.1.3 (Contraction property of T_μ [Ber19])**.** *For each stationary policy $\mu \in \mathcal{M}$ there exist some positive scalar $\rho < 1$ such that*

$$\|T_\mu J - T_\mu J'\|_v \leq \rho \|J - J'\|_v \text{ for all } J, J' \in \mathcal{B}(X),$$

where $\mathcal{B}(X)$ is as defined in Definition 4.1.3 and the norm $\|\cdot\|_v$ is given by Equation (4.4) *for some positive function $v : X \to \mathbb{R}$.*

Proof. This proof closely follows the proof of Proposition 4.2.5 in [Ber19]. Let $\mu \in \mathcal{M}$ be an arbitrary stationary policy and let v_μ be the function we have assumed to exist from Assumption 5.1.2. That is,

$$v_\mu(i) = 1 + \sum_{j=1}^{n} p_{ij}(\mu(i)) v_\mu(j) \text{ for all } i \in X.$$

Then we have that

$$\sum_{j=1}^{n} p_{ij}(\mu(i)) v_\mu(j) = v_\mu(i) - 1 \leq \rho v_\mu(i), \tag{5.6}$$

where

$$\rho = \max_{i=1,\ldots,n} \frac{v_\mu(i) - 1}{v_\mu(i)},$$

and as $v_\mu(i) \geq 1$ for all $i \in \{1, \cdots, n\}$, we have $\rho < 1$. We then have that

$$\begin{aligned}
(T_\mu J)(i) &= (T_\mu J')(i) + \sum_{j=1}^{n} p_{ij}(\mu(i))(J(j) - J'(j)) \\
&= (T_\mu J')(i) + \sum_{j=1}^{n} p_{ij}(\mu(i)) v_\mu(j) \frac{J(j) - J'(j)}{v_\mu(j)} \\
&\leq (T_\mu J')(i) + \sum_{j=1}^{n} p_{ij}(\mu(i)) v_\mu(j) \|J - J'\|
\end{aligned}$$

$$\leq (T_\mu J')(i) + \rho\, v_\mu(i)\|J - J'\|,$$

where we in the last inequality have used that Equation (5.6) holds. Hence,

$$\frac{(T_\mu J)(i) - (T_\mu J')(i)}{v_\mu(i)} \leq \rho\,\|J - J'\|.$$

By maximizing both sides of the inequality above with regards to $i \in X$, and noting that we can change the roles of J and J' in the above equation and end up with the same bound, we see that

$$\|T_\mu J - T_\mu J'\| = \max_{i\in X} \frac{(T_\mu J)(i) - (T_\mu J')(i)}{v_\mu(i)} \leq \rho\|J - J'\|.$$

Thus, we have that T_μ is a contraction. ■

We then proceed with showing that H as defined in Equation (5.5) is monotone.

Proposition 5.1.4 (Monotonicity property of H)**.** †
The function H, as defined in Equation (5.5)*, is monotone.*

Proof. We need to show that for all $J, J' \in \mathcal{R}(X)$, with $J \leq J'$ pointwise, we have

$$H(i, u, J) \leq H(i, u, J') \text{ for all } i \in X \text{ and } u \in U(i).$$

Thus, let us assume that $J \leq J'$ pointwise for some $J, J' \in \mathcal{R}(X)$. Then we have that

$$\begin{aligned} H(x, u, J) - H(x, u, J') &= p_{it}(u)g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + J(j)) \\ &\quad - \left(p_{it}(u)g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + J'(j)) \right) \\ &= \sum_{j=1}^{n} p_{ij}(u)(J(j) - J'(j)) \leq 0, \end{aligned}$$

where the last inequality follows from the fact that $J - J' \leq 0$ for all $i \in X$, and that $p_{ij} \geq 0$ for all $i, j \in X$. Thus, for all $J, J' \in \mathcal{R}(X)$, with $J \leq J'$ pointwise, we have that

$$H(x, u, J) \leq H(x, u, J') \text{ for all } i \in X \text{ and } u \in U(i),$$

as we needed to show. ■

With Propositions 5.1.3 and 5.1.4 in hand we are able to prove multiple of the results concerning the infinite horizon SSP problem that are provided in chapter 4.2 in [Ber19]. We are also able to present proofs building on the abstract dynamic programming theory presented in Chapter 4 in contrast to the more direct proofs presented in [Ber19]. The first result guarantees that the optimal cost function J^* of the infinite horizon SSP problem as presented in this section solves the Bellman equation as stated in Equation (5.2), and that it is the unique solution of the equation.

Proposition 5.1.5 (J^* is the unique solution of Bellman's equation [Ber19])**.** *The optimal cost function J^* for the stochastic shortest path problem with the assumptions presented in this chapter is the unique solution of the Bellman equation, i.e.*

$$J^*(i) = \min_{u \in U(i)} \left[p_{it}(u) g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + J^*(j)) \right].$$

Proof. †
As we in Proposition 5.1.3 have shown that T_μ is a contraction for each stationary policy μ we have from Proposition 4.1.4 that T is a contraction as well. We then know from Proposition 4.2.1 that there exists some unique fixed point $\bar{J}$ for the operator T, i.e.

$$\begin{aligned} \bar{J}(i) = (T\bar{J})(i) &= \min_{u \in U(i)} H(i,u,\bar{J}) \\ &= \min_{u \in U(i)} \left[p_{it}(u) g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \bar{J}(j)) \right]. \end{aligned}$$

Thus, we have that $\bar{J}$ is the unique solution of Bellman's equation. Then, as a consequence of the monotonicity of H proven in Proposition 5.1.4 we have from Proposition 4.2.2 that the unique fixed point $\bar{J}$ satisfies

$$\bar{J}(i) = J^*(i) = \inf_{\pi \in \Pi} J_\pi(i) = \inf_{\mu \in \mathcal{M}} J_\mu(i) \text{ for all } i \in X.$$

Thus the optimal cost function J^* equals the unique solution of Bellman's equation $\bar{J}$, and the proposition is proven. ■

In [Ber19] the author uses the inequality in Equation (5.4) to prove that the value iteration algorithm, as previously discussed in Section 4.3, converges to the optimal cost function J^* for the kind of SSP problems described in this section. We now take a quick look at how value iteration looks for this class of problems, and present an alternative proof using the theory from Chapter 4. Keep in mind that we have impost a stricter set of assumptions by adding Assumption 5.1.2. We have also restricted the initial cost function J_0 to be in the set $\mathcal{B}(X)$.

Proposition 5.1.6 (Convergence of VI [Ber19])**.** *For each state $i \in X$ we have that $J^*(i) < \infty$, and given an initial cost function $J_0 \in \mathcal{B}(X)$, we have that the sequence $\{J_k\}$ of cost functions generated by the value iteration algorithm converges to the optimal cost function J^*. The algorithm progresses by*

$$J_{k+1}(i) = \min_{u \in U(i)} \left[p_{it}(u) g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + J_k(j)) \right]. \tag{5.7}$$

Proof. †
As mentioned above we have as a consequence of Assumption 5.1.1 that each possible policy terminates at some point. Therefore, the optimal cost must be finite as we only allow finite costs for each stage. Furthermore, we

have from Propositions 5.1.3 and 5.1.4 that the operator T_μ is monotone and a contraction for each $\mu \in \mathcal{M}$. As the number of states and controls are finite, we have that the number of stationary policies also is finite, which makes the set $\mathcal{M}$ finite. Then, as $J_0 \in \mathcal{B}(X)$, we have from Proposition 4.3.3 that there exist some $N \in \mathbb{N}$ such that the one-step lookahead policy μ^* found by using $T^n J_0$ as our approximation of J^* actually have the optimal cost function J^* as its value function, i.e. $J_{\mu^*} = J^*$. Then, by setting $J_k = T^k J_0$ we see that the iteration formula becomes

$$\begin{aligned} J_{k+1}(i) &= (T^{k+1}J_0)(i) = (T(T^kJ_0))(i) \\ &= (TJ_k)(i) = \min_{u \in U(i)} \left[p_{it}(u)g(i,u,t) + \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + J_k(j)) \right]. \end{aligned}$$

Thus we see that the value iteration algorithm is given as the proposition states and that it actually converges towards J^*. ■

The same result can also be shown for all stationary policies, as the next proposition shows.

Proposition 5.1.7 (VI and Bellman's equation for stationary policies [Ber19]). *For each stationary policy μ we have that the cost function J_μ satisfies*

$$J_\mu(i) = p_{it}(\mu(i))g(i,\mu(i),t) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + J_\mu(j)), \tag{5.8}$$

for each state $i \in X$, and we have that J_μ is the unique solution of the system of equations defined by (5.8). *We also have that for any initial cost function J_0 the sequence $\{J_k\}$ of cost functions generated by the value iteration algorithm converges to the cost function J_μ, where the next function in the sequence is generated by*

$$J_{k+1}(i) = p_{it}(\mu(i))g(i,\mu(i),t) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + J_k(j)).$$

Proof. †
We start by proving that J_μ is a fixed point for Equation (5.8) for each stationary policy $\mu \in \mathcal{M}(X)$. From Proposition 5.1.3 we have that T_μ is a contraction mapping. Then, from Proposition 4.2.1 we have that J_μ is the unique fixed point of T_μ in the space $\mathcal{B}(X)$. Thus

$$J_\mu(i) = (T_\mu J_\mu)(i) = p_{it}(\mu(i))g(i,\mu(i),t) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + J_\mu(j)).$$

The last part of this proof follows the proof of Proposition 4.2.3 in [Ber19]. To see that the value iteration algorithm converges towards J_μ, just consider the modified optimal cost problem where we let $U(i) = \{\mu(i)\}$ for each state i. Then the optimal cost function J^* is equal to the cost function J_μ of the policy μ, as the only available control at each state is the control dictated by the stationary policy $\mu \in \mathcal{M}$. We know from Proposition 5.1.6 that the value

iteration algorithm converges to the optimal cost function J^*, which for the modified problem is equal to J_μ. Note also that by rewriting Equation (5.7) by letting $u = \mu(i)$ we get that the value iteration for the modified problem takes the form

$$J_{k+1}(i) = p_{it}(\mu(i))g(i, \mu(i), t) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i, \mu(i), j) + J_k(j)),$$

just as stated in the proposition. By the arguments above we have that the algorithm converges toward the cost function of the policy J_μ. ∎

We then look at the final result for this section which guarantees the convergence of policy iteration (Section 4.3).

Proposition 5.1.8 (Convergence of exact PI [Ber19])**.** *We have that the policy iteration algorithm produces an improving sequence of policies $\{\mu^k\}$ which converges to an optimal policy for the SSP problem as described in this section, i.e. we have that $J_{\mu^{k-1}} \leq J_{\mu^k}$ pointwise for each $k \in \mathbb{N}$ and that $J_{\mu^n} = J^*$ pointwise for all $n \in \mathbb{N}$ with $n \geq N$ for some large enough $N \in \mathbb{N}$. The* policy evaluation *step of the kth iteration consists of finding the solution J_{μ^k} of the following set of equations*

$$\begin{aligned} J_{\mu^{k-1}}(i) &= p_{it}(\mu^{k-1}(i))g(i, \mu^{k-1}(i), t) \\ &\quad + \sum_{j=1}^{n} p_{ij}(\mu^{k-1}(i))(g(i, \mu^{k-1}(i), j) + J_{\mu^{k-1}}(j)) \text{ for all } i \in X, \end{aligned}$$

while the policy improvement *step consists of finding a policy μ^k that for each state $i \in X$ satisfy*

$$\mu^k(i) \in \arg\min_{u \in U(i)} \left[p_{it}(u)g(i, u, t) + \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + J_{\mu^{k-1}}(j)) \right].$$

Proof. †
We have from Proposition 4.3.5 that the policy iteration algorithm generates a sequence $\{\mu^k\}$ of policies where for each $k \in \mathbb{N}$ we have that $J_{\mu^{k-1}} \leq J_{\mu^k}$, given that the monotonicity and contraction property is met. The proposition also states that $J_{\mu^k} = J^*$ for some $k \in \mathbb{N}$ if the set of policies is finite. Note that the set of policies is in fact finite for the SSP problem as described in this section as the number of states and controls are finite. Then, as we have from Propositions 5.1.3 and 5.1.4 that the monotonicity and contraction properties are met, and as the set of policies are finite, we see that the conditions in Proposition 4.3.5 are met. Thus, the policy iteration algorithm generates a sequence $\{\mu^k\}$ of improving policies and for some $n \in \mathbb{N}$ we have that $J_{\mu^n} = J^*$, i.e. the policy μ^n (and every subsequent policy generated by the policy iteration algorithm) is an optimal policy. ∎

We end this section with a numerical example where we find the optimal cost function and an optimal policy for a simple stochastic shortest path problem with use of both value iteration and policy iteration.

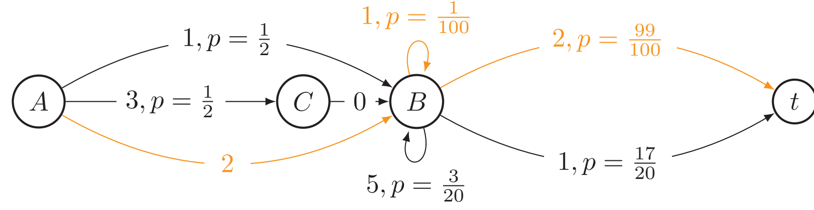

Figure 5.1: A simple SSP. Same coloured paths originating at the same node depict the possible transitions from the given node under a specific control. The black lines depict the transitions for control u_1, while orange shows the same for control u_2. Node C is introduced for technical reasons only.

Example 5.1.9 (Numerical solution of SSP problem)**.** †
In this example we consider the stochastic shortest path-problem of the graph depicted in Figure 5.1. We will first implement and use value iteration to find the optimal value function of the SSP, and then we implement policy iteration and use the implemented algorithm to find an optimal policy for the SSP along with the cost function of the final policy generated by the Policy iteration algorithm. We first note that Assumption 5.1.1 and Assumption 5.1.2 holds for this SSP. The fact that the first of the two assumptions holds can be seen from Figure 5.1 as it is fairly straight forward to confirm that any policy must terminate eventual due to the structure of the graph. It is also possible to consider each possible stationary policy to see that Assumption 5.1.2 holds, but we will not go through the details of those calculation here, as the assumption is a technical detail we needed for the proof of Proposition 5.1.3. However, the fact that these assumptions hold guarantees that both of these algorithms converge as these are the assumptions of Propositions 5.1.6 and 5.1.8. The code used to implement the graph can be found in Listing A.1, and the implementation of the value iteration and policy iteration algorithms can be found in Listing A.3.

The solution generated by our implementation of the value iteration algorithm is

$$\bar{J}(A) = \frac{97}{25}, \bar{J}(B) = \frac{47}{25}, \bar{J}(C) = \frac{47}{25},$$

when we only include the first two decimal places (the error tolerance of our implementation is 10^{-16}). Let us now check that $\bar{J}$ on this form indeed is a fixed point of the Bellman equation as stated in Equation (5.2). Note that control 1 is the control for which the possible transitions are depicted in black in Figure 5.1, while control 2 is the one coloured in orange. We have that

$$\begin{aligned}
\bar{J}(A) &= \min_{u\in\{1,2\}} \left[p_{it}(u)g(A,u,t) + \sum_{j\in\{A,B,C\}} p_{Aj}(u)(g(A,u,j) + \bar{J}(j)) \right] \\
&= \min\left\{ \frac{1}{2}\left(1 + \frac{47}{25}\right) + \frac{1}{2}\left(3 + \frac{47}{25}\right), 2 + \frac{47}{25} \right\} \\
&= \min\left\{ \frac{36}{25} + \frac{61}{25}, \frac{50}{25} + \frac{47}{25} \right\}
\end{aligned}$$

$$
\begin{aligned}
&= \min\left\{\frac{97}{25}, \frac{97}{25}\right\} \\
&= \frac{97}{25}, \\
\bar{J}(B) &= \min\left\{\frac{17}{20} + \frac{3}{20}\left(5 + \frac{47}{25}\right), 2\frac{99}{100} + \frac{1}{100}\left(1 + \frac{47}{25}\right)\right\} \\
&= \min\left\{\frac{17}{20} + \frac{3}{20}\frac{172}{25}, 2\frac{99}{100} + \frac{1}{100}\frac{72}{25}\right\} \\
&= \min\left\{\frac{941}{500}, \frac{2511}{1250}\right\} = \frac{941}{500} \\
&= \frac{47}{25} + \frac{1}{500} \approx \frac{47}{25}, \\
\bar{J}(C) &= \bar{J}(B) \approx \frac{47}{25},
\end{aligned}
$$

and as we can see, $\bar{J}$ is indeed a fixed point for the Bellman equation when we exclude errors smaller than two decimal places $\left(\frac{1}{500}\right)$, implying that the cost function is the optimal cost function, i.e $J^* = \bar{J}$. Let us now proceed to verify that the final policy generated by the policy iteration algorithm actually is an optimal policy. The output from the implementation yields

$$
\bar{\mu}(A) = 2, \, \bar{\mu}(B) = 1, \bar{\mu}(C) = 1.
$$

Observe that as the average cost of applying control 1 in state A is 2, which is the same as the cost of applying action 2, we actually have that a policy is optimal regardless of its value in state A. Our implementation also claim that the cost function of the policy $\bar{\mu}$ as defined above (when only including two decimal places) is

$$
J_{\bar{\mu}}(A) = \frac{97}{25}, \, J_{\bar{\mu}}(B) = \frac{47}{25}, J_{\bar{\mu}}(C) = \frac{47}{25},
$$

which we recognizance as the optimal cost function for this SSP-problem as we showed that it indeed is a fixed point for Equation (5.2). We now verify that the policy given by the algorithm indeed is optimal by checking that the cost function of $\bar{\mu}$ is a fixed point for the Bellman equation for policies, which is stated in Equation (5.8). We find that

$$
\begin{aligned}
J_{\bar{\mu}}(A) &= p_{At}(\bar{\mu}(A))g(A, \bar{\mu}(A), t) + \sum_{j \in \{A,B,C\}} p_{Aj}(\bar{\mu}(A))(g(A, \bar{\mu}(A), j) + \bar{J}_{\bar{\mu}}(j)) \\
&= 2 + J_{\bar{\mu}}(B) = 2 + \frac{47}{25} = \frac{92}{25}, \\
J_{\bar{\mu}}(B) &= p_{Bt}(\bar{\mu}(B))g(B, \bar{\mu}(B), t) + p_{BB}(\bar{\mu}(B))(g(B, \bar{\mu}(B), j) + \bar{J}_{\bar{\mu}}(B)) \\
&= \frac{17}{20} + \frac{3}{20}(5 + J_{\bar{\mu}}(B)) \\
&= \frac{17}{20} + \frac{3}{20}\left(5 + \frac{47}{25}\right) \\
&= \frac{17}{20} + \frac{3}{20}\frac{172}{25} = \frac{941}{500} \\
&= \frac{47}{25} + \frac{1}{500} \approx \frac{47}{25},
\end{aligned}
$$

$$J_{\bar{\mu}}(C) = J_{\bar{\mu}}(B) \approx \frac{47}{25},$$

which show that $J_{\bar{\mu}}$, as calculated by the PI algorithm, indeed is the cost function of the policy $\bar{\mu}$ as it is the fixed point for Equation (5.8). This also proves that $\bar{\mu}$ is an optimal policy as the cost function $J_{\bar{\mu}}$ is equal to the optimal cost function J^*. We can then ask ourselves, if we in a given state i choose a control u and then follow the optimal policy from then on out, what is the expected cost? We can denote this value $Q(i,u)$, which is called the Q-value of the state-control pair (i,u). This concept is crucial for some of the methods we will look at in Chapter 6. The Q-value of a given state-control pair (i,u) can be found, given that we know the optimal value function J^* of the SSP, by

$$Q(i,u) = p_{it}(u)g(i,u,t) + \sum_{j\in\{A,B,C\}} p_{ij}(u)(g(i,u,j) + J^*(j)).$$

Let us then find the Q-values for the SSP we consider in this example. We have that

$$\begin{aligned}
Q(A,1) &= p_{At}(1)g(A,1,t) + \sum_{j\in\{A,B,C\}} p_{Aj}(1)(g(A,1,j) + J^*(j)) \\
&= p_{AB}(1)(g(A,1,B) + J^*(B)) + p_{AC}(1)(g(A,1,C) + J^*(C)) \\
&= \frac{1}{2}\left(1 + \frac{47}{25}\right) + \frac{1}{2}\left(3 + \frac{47}{25}\right) = \frac{36}{25} + \frac{61}{25} = \frac{97}{25}, \\
Q(A,2) &= p_{At}(2)g(A,2,t) + \sum_{j\in\{A,B,C\}} p_{Aj}(2)(g(A,2,j) + J^*(j)) \\
&= p_{AB}(2)(g(A,2,B) + J^*(B)) \\
&= \left(2 + \frac{47}{25}\right) = \frac{50}{25} + \frac{47}{25} = \frac{97}{25}, \\
Q(B,1) &= p_{Bt}(1)g(B,1,t) + \sum_{j\in\{A,B,C\}} p_{Bj}(1)(g(B,1,j) + J^*(j)) \\
&= p_{Bt}(1)g(B,1,t) + p_{BB}(1)(g(B,1,A) + J^*(B)) \\
&= \frac{17}{20} + \frac{3}{20}\left(5 + \frac{47}{25}\right) = \frac{17}{20} + \frac{3}{20}\left(\frac{172}{25}\right) \\
&= \frac{425}{500} + \frac{516}{500} = \frac{941}{500} = \frac{47}{25} + \frac{1}{500} \approx \frac{47}{25}, \\
Q(B,2) &= p_{Bt}(2)g(B,2,t) + \sum_{j\in\{A,B,C\}} p_{Bj}(2)(g(B,2,j) + J^*(j)) \\
&= p_{Bt}(2)g(B,2,t) + p_{BB}(2)(g(B,2,B) + J^*(B)) \\
&= 2\frac{99}{100} + \frac{1}{100}\left(1 + \frac{47}{25}\right) = \frac{198}{100} + \frac{1}{100}\frac{72}{25} \\
&= \frac{4950}{2500} + \frac{72}{2500} = \frac{2511}{1250} = \frac{2500}{1250} + \frac{11}{1250} \approx 2, \\
Q(C,1) &= p_{Ct}(1)g(C,1,t) + \sum_{j\in\{A,B,C\}} p_{Cj}(1)(g(C,1,j) + J^*(j)) \\
&= p_{CB}(1)(g(C,1,B) + J^*(B))
\end{aligned}$$

State s	$J^*(s)$	$Q(s,1)$	$Q(s,2)$
A	$\frac{97}{25} = 3.88$	$\frac{97}{25} = 3.88$	$\frac{97}{25} = 3.88$
B	$\frac{47}{25} = 1.88$	$\frac{47}{25} = 1.88$	2
C	$\frac{47}{25} = 1.88$	$\frac{47}{25} = 1.88$	

Table 5.1: Optimal value function and Q-values for SSP depicted in Figure 5.1

$$= J^*(B) \approx \frac{47}{25}.$$

We now summarize the findings from this example in Table 5.1. Note that for each state i we have that the optimal value function $J^*(i)$ coincides with the Q-value of the state-control pair (i, u) for which the control u satisfy

$$u = \arg\min_{u \in U(i)} Q(i, u),$$

i.e. the optimal value function $J^*(i)$ is equal to the minimum of the Q-values $Q(i, \cdot)$. This is exactly what we would expected, as we have the following relation

$$\begin{aligned} \min_{u \in U(i)} Q(i, u) &= \min_{u \in U(i)} \left[p_{it}(u) g(i, u, t) + \sum_{j \in \{A,B,C\}} p_{ij}(u)(g(i, u, j) + J^*(j)) \right] \\ &= J^*(i). \end{aligned}$$

5.2 Discounted problems

We now continue with looking at the second type of infinite horizon problems that we are looking at in this chapter, and that is the discounted problems. The discounted problems are characterized by having a discount factor α with a value below 1, that is $\alpha \in (0, 1)$. The effect of the discount factor is that the costs encountered at future stages influence the expected cost less and less the further into the future the cost is incurred. We are in this chapter going to prove the same results as we did for the SSP problem in the former section. In [Ber19] the author argues that the results proven for the SSP case is valid for the discounted problems as well by rewriting the discounted problem into a SSP problem. We will on the other hand prove the results by again using the strength of the abstract dynamic programming theory from Chapter 4. We therefore start by introducing the Bellman equations and the DP operators for the discounted problem before we prove that the latter have the necessary properties to be able to use the theory from Chapter 4. As written in the introduction to the chapter we here also assume that the state space X and control space U is finite. We can thus also here represent the states by the first n integers when we have n states. We therefore let $X = \{1, 2, \ldots, n\}$, as we in this case does not have a terminal state t. By using the same transition probabilities as defined in Equation (5.1) we can write the Bellman equation

for the discounted problem. We have from [Ber19] that for each state $i \in X$ we have

$$J^*(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)).$$

Then, for a given stationary policy we have

$$J_\mu(i) = \sum_{j=1}^{n} p_{ij}(\mu(x))(g(i,\mu(x),j) + \alpha J_\mu(j)).$$

Thus, we let $H\colon X \times U \times \mathcal{R}(X) \to \mathbb{R}$ take the form

$$H(x,u,J) = \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J(j)). \tag{5.9}$$

It then follows that the DP operators are given by

$$(TJ)(i) = \min_{u \in U(i)} H(i,u,J) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J(j)),$$

and

$$(T_\mu J)(i) = H(i,\mu(i),J) = \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + \alpha J(j)). \tag{5.10}$$

We can then move on to show that T_μ is a contraction, which then by Proposition 4.1.4 implies that T is as well.

Proposition 5.2.1 (Contraction propoerty of T_μ [Ber19])**.** *For each stationary policy $\mu \in \mathcal{M}$ there exist some positive scalar $\rho < 1$ such that*

$$\|T_\mu J - T_\mu J'\|_v \leq \rho \|J - J'\|_v \text{ for all } J, J' \in \mathcal{B}(X),$$

where $\mathcal{B}(X)$ is as defined in Definition 4.1.3, the norm $\|\cdot\|_v$ is given by Equation (4.4) *for some positive function $v\colon X \to \mathbb{R}$ and T_μ is as defined in Equation* (5.10)*.*

Proof. †

This proof is based on the proof of Proposition 4.2.5 in [Ber19]. We let $\mu \in \mathcal{M}$ be an arbitrary stationary policy, and we let $v_\mu(i) = 1$ for all $i \in X$.

Then we have that

$$\begin{aligned}
(T_\mu J)(i) - (T_\mu J')(i) &= H(i,\mu(i),J) - H(i,\mu(i),J') \\
&= \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + \alpha J(j)) \\
&\quad - \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + \alpha J'(j)) \\
&= \sum_{j=1}^{n} p_{ij}(\mu(i))\alpha(J(j) - J'(j)) \\
&= \alpha \sum_{j=1}^{n} p_{ij}(\mu(i))v(j)\frac{(J(j) - J'(j))}{v(j)} \\
&\le \alpha \sum_{j=1}^{n} p_{ij}(\mu(i))v(j)\|J - J'\| \\
&= \alpha v(i)\|J - J'\| \sum_{j=1}^{n} p_{ij}(\mu(i)) = \alpha v(i)\|J - J'\|,
\end{aligned}$$

where the transition to the last line follows from the fact that when we fix $i \in X$ we have that $v(i) = v(j)$ for each $j \in X$. The last equality follows from the fact that when we fix $i \in X$ we have $\sum_{j=1}^{n} p_{ij}(\mu(i)) = 1$. The inequality above implies that

$$\frac{(T_\mu J)(i) - (T_\mu J')(i)}{v(i)} \le \alpha\|J - J'\|.$$

By then switching the roles of J and J' we get the inequality

$$\frac{(T_\mu J')(i) - (T_\mu J)(i)}{v(i)} \le \alpha\|J - J'\|.$$

Then, by maximizing both sides of the inequalities we get that

$$\|T_\mu J - T_\mu J'\| \le \alpha\|J - J'\|,$$

and as $\alpha \in (0,1)$ and $\mu \in \mathcal{M}$ was chosen arbitrarily we have that T_μ is a contraction for each stationary policy μ, as we were supposed to show. Note also that the modulus ρ from the proposition is in fact equal to the discount factor α when we let v_μ be the constant function with value one. ■

We then proceed with showing that the function H is monotone.

Proposition 5.2.2 (Monotonicity property of H)**.** †
The function H as defined in Equation (5.9) *is monotone.*

Proof. We need to show that for all $J, J' \in \mathcal{R}(X)$, with $J \le J'$ pointwise, we have

$$H(i,u,J) \le H(i,u,J') \text{ for all } i \in X \text{ and } u \in U(i).$$

Thus, lets assume that $J \leq J'$ pointwise for some $J, J' \in \mathcal{R}(X)$. Then we have that

$$\begin{aligned} H(x,u,J) - H(x,u,J') &= \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J(j)) \\ &\quad - \left(\sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J'(j)) \right) \\ &= \sum_{j=1}^{n} p_{ij}(u)\alpha(J(j) - J'(j)) \leq 0, \end{aligned}$$

where the last inequality follows from the fact that $J - J' \leq 0$ for all $i \in X$, and that $p_{ij} \geq 0$ for all $i, j \in X$. Thus, for all $J, J' \in \mathcal{R}(X)$ with $J \leq J'$ pointwise we have that

$$H(x,u,J) \leq H(x,u,J') \text{ for all } i \in X \text{ and } u \in U(i),$$

as we needed to show. ■

We then proceed with stating results that are analogous to the once from Section 5.1. As these results only rely on the monotonicity and contraction property to hold we leave out the proofs as they are similar to the proofs of their equivalent result for the SSP case. We start with the result that correspond to Proposition 5.1.5. It guarantees that J^* is the unique solution of the Bellman equation in the discounted case.

Proposition 5.2.3 (J^* is the unique solution of Bellman's equation [Ber19])**.** *The optimal cost function J^* for the discounted problem with finite state and control space is the unique solution of the Bellman equation, i.e.*

$$J^*(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)).$$

The next proposition concerns the convergence of the value iteration algorithm for the discounted problem.

Proposition 5.2.4 (Convergence of VI [Ber19])**.** *For each state $i \in X$ we have that $J^*(i) < \infty$, and given an initial cost function $J_0 \in \mathcal{B}(X)$ we have that the sequence $\{J_k\}$ of cost functions generated by the value iteration algorithm converges to the optimal cost function J^*. The algorithm progresses by*

$$J_{k+1}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J_k(j)). \tag{5.11}$$

We then proceed with the result for individual stationary policies.

Proposition 5.2.5 (VI and Bellman's equation for stationary policies [Ber19])**.** *For each stationary policy μ we have that the cost function J_μ satisfies*

$$J_\mu(i) = \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + \alpha J_\mu(j)), \tag{5.12}$$

for each state $i \in X$*, and we have that* J_μ *is the unique solution of the system of equations defined by* (5.12)*. We also have that for any initial cost function* J_0 *the sequence* $\{J_k\}$ *of cost functions generated by the value iteration algorithm converges to the cost function* J_μ*, where the next function in the sequence is generated by*

$$J_{k+1}(i) = \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i, \mu(i), j) + \alpha J_k(j)).$$

The last proposition of this chapter say that policy iteration also generates an optimal policy for the discounted problem in this setting.

Proposition 5.2.6 (Convergence of exact PI [Ber19])**.** *We have that the policy iteration algorithm produces an improving sequence of policies* $\{\mu^k\}$ *which converges to an optimal policy for the discounted problem, i.e. we have that* $J_{\mu^{k-1}} \leq J_{\mu^k}$ *pointwise for each* $k \in \mathbb{N}$ *and that* $J_{\mu^k} = J^*$ *pointwise for all* $k \in \mathbb{N}$ *with* $k \geq N$ *for some large enough* $N \in \mathbb{N}$*. The* policy evaluation *step of the* k*th iteration consists of finding the solution* J_{μ^k} *of the following set of equations*

$$J_{\mu^{k-1}}(i) = \sum_{j=1}^{n} p_{ij}(\mu^{k-1}(i))(g(i, \mu^{k-1}(i), j) + \alpha J_{\mu^{k-1}}(j)) \text{ for all } i \in X,$$

while the policy improvement *step consists of finding a policy* μ^k *that for each state* $i \in X$ *satisfy*

$$\mu^k(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + \alpha J_{\mu^{k-1}}(j)).$$

We have in this chapter shown that the value iteration and policy iteration algorithms converge for two different types of infinite horizon dynamic programming problems. For the case where we discount the future costs we see that the sum of accumulated costs converge with assumptions that are less strict than what we need in the case where $\alpha = 1$. This makes intuitively sense as we know that for some constant C, which we can think of as $\max_{i,j \in X, u \in U(i)} g(i, u, j)$, and a positive factor $\alpha < 1$, we have that

$$\sum_{n=0}^{\infty} \alpha^n C,$$

converges. However, for the case where $\alpha = 1$ we risk allowing policies that have an associated cost function that for some states is unbounded, which gives us problems with convergence. This is analogous with the case we considered in Example 4.2.6. We therefore need assumptions that guarantee that each policy reach the terminal state t with probability $p = 1$ in order to show that value iteration and policy iteration converges.

Another assumption that is important in the dynamic programming setting, but which often does not hold for real-world problems, is that we have perfect knowledge about the underlying system. That is, for each state i and control u we know the probability $p_{ij}(u)$ of transitioning to some state j. We also know which costs we can receive from the given transition and the distribution of these

costs. When this information is not know we need to rely on approximations and simulations, which is what approximate dynamic programming, or reinforcement learning, is concerned with. Reinforcement learning is therefore the topic of the next chapter.

CHAPTER 6

Reinforcement learning

This short introduction to reinforcement learning (RL) is based on [SB18]. The goal of reinforcement learning is to 'teach' a learning agent a rule that given a state of the learning agent in a given environment tells the agent which action to choose next. The goal is then to maximize the accumulation of rewards that are given to the learner after each action depending on the state at that time and the sequence of actions taken beforehand. A concrete example of a learning agent could be a dog trying to learn which command corresponds to what trick, where the enviorment is the owner giving commands and rewards, and the rewards are treats given to the dog by the owner when the dog is performing the correct trick for the given command flawlessly. The goal of the dog is then of course to be able to performe the correct trick flawlessly as fast as possible so that it is rewarded more often. In addition to the agent and the enviroment, we have a policy, a reward signal, a value function, and sometimes a model of the environment as part of the reinforcment learning system.

The policy dictates how the agent should act given the current situation. In other words, it is a mapping from states of the agent to actions the agent can perform in these states. The policy could also be stochastic, giving a probability to each possible action. At each time step of the reinforcement learning procedure the agent receives a reward from the environment. The goal of the agent is to maximize this reward over time, and thus the agent will try to change the policy such that the action chosen by the policy yields a high expected reward. The reward signal may also be a stochastic function of the state of the environment and the actions taken by the agent. The value function tells us something about the expected accumulation of rewards in the future when starting from a given state in the environment, called the value of the state. Thus the reward signal tells us something about the immediate consequence of an action, while the value function says something about what we can expect in the long-run. The estimating of values is a more complicated task then finding the reward for the next action, and the estimating of the values is central to reinforcement learning since we often would like to choose actions that maximizes the value. A model of the environment is a model that could help us predict the reward from a given action in a given state before they are experienced thus allowing us to make a plan on which actions to take before actually observing anything. In fact, if a model of the environment is given, we have that dynamic programming and reinforcement learning becomes equivalent. Thus, we can look at dynamic programming as the special case of reinforcement

learning where we are able to obtain some model of the environment. However, in contrast to dynamic programming, a model of the environment is not needed in reinforcement learning. We will in this chapter give a short introduction to RL, and some RL methods, before looking at the similarities and differences between reinforcement learning and dynamic programming.

6.1 A short introduction to finite MDPs and reinforcement learning

Finite Markov decision processes

Let us now take a look at Markov decision processes (MDPs), which is the kind of processes that are underlying the problems we are trying to solve using reinforcement learning, and is the same kind of processes underlying dynamic programming. That is because an MDP is a way of modelling a sequential decision problem, which is exactly what we try to solve both with reinforcement learning and dynamic programming.

In an MDP we have discreet time steps, just like the model we covered in Section 3.1. Unlike the form of problems we looked at in Chapter 3, here we let the time horizon possible be infinite, just as the case we considered in Chapter 5. At each time step $t \in \mathbb{N}$ an agent is in a given state $x_t \in X$, take an action $u_t \in U(x_t) \subset U$, and then receive a reward $r_{t+1} \in \mathcal{R} \subset \mathbb{R}$, before proceeding to the next state $x_{t+1} \in X$. The MDP is said to be finite when the cardinality of X and U, which are the set of possible states and actions respectively, are both finite. Another property of an MDP is the Markov property, which says that at time t the state x_t and reward r_t of the process only depend on the previous state x_{t-1} and action u_{t-1}. Because of the Markov property, the *dynamics* of the MDP, that is how the process behaves, is completely determined by the function $p : X \times \mathcal{R} \times X \times U \to [0, 1]$, which is defined as

$$p(x', r|x, u) = P(x_t = x', r_t = r|x_{t-1} = x, u_{t-1} = u),$$

for all $x', x \in X, r \in \mathcal{R}, a \in U$. In other words, the probability of ending up in state x', and receiving reward r, when taking action u in state x is $p(x', r|x, u)$, and from the Markov property we have that p completely describes the process.

Reinforcement learning

As in dynamic programming, the goal in reinforcement learning is most often to optimize some value. However, in contrast to dynamic programming where we want to minimize cost, in reinforcement learning we want to maximize the cumulative sum of future discounted rewards obtained by the agent, denoted G_t. That is, we want to maximize

$$G_t = \sum_{k=t+1}^{\infty} \alpha^{k-t-1} r_k,$$

where $\alpha \in [0,1)$. Note also from the definition of G_t that we have

$$\begin{aligned} G_t &= \sum_{k=t+1}^{\infty} \alpha^{k-t-1} r_k \\ &= r_{t+1} + \sum_{k=t+2}^{\infty} \alpha^{k-t-1} r_k \\ &= r_{t+1} + \alpha \sum_{k=t+2}^{\infty} \alpha^{k-t-2} r_k \\ &= r_{t+1} + \alpha G_{t+1}. \end{aligned}$$

A *policy* π in the reinforcement learning context is, given a state x, a probability distribution over actions u. Then $\pi(\cdot|x)$ denotes the probability density of the actions u when in state x. For a finite MDP, where U and X are finite, we have that π is a mapping $\pi : U \times X \to [0,1]$, and thus $\pi(u|x)$ denotes the probability of choosing action u when in state x. We then continue to look at the *state-value function* of a given policy π, denoted $J_\pi(x)$, where $x \in X$. The value function is defined as

$$J_\pi(x) = E_\pi[G_t|x_t = x] = E_\pi \left[\sum_{k=0}^{\infty} \alpha^k r_{t+k+1} \middle| x_t = x \right], \text{ for all } x \in X,$$

which is the expectation of the discounted sum of all future rewards when we go from state x and follow policy π. Given the definition of $J_\pi(x)$ we can show the following relation

$$\begin{aligned} J_\pi(x) &= E_\pi[G_t|x_t = x] \\ &= E_\pi[r_{t+1} + \alpha G_{t+1}|x_t = x] \\ &= \sum_{u \in U(x)} \pi(u|x) \sum_{x' \in X} \sum_{r \in \mathcal{R}} p(x', r|x, u)(r + \alpha E_\pi[G_{t+1}|x_{t+1} = x']) \\ &= \sum_{u \in U(x)} \pi(u|x) \sum_{x' \in X, r \in \mathcal{R}} p(x', r|x, u)(r + \alpha J_\pi(x')), \text{ for all } x \in X, \end{aligned}$$

which we can see is very similar to Bellman equation for policies.

We also have the *action-value function* $Q_\pi(x,u)$ which is defined by

$$Q_\pi(x,u) = E_\pi[G_t|x_t = x, u_t = u] = E_\pi \left[\sum_{k=0}^{\infty} \alpha^k r_{t+k+1} \middle| x_t = x, u_t = u \right].$$

The goal in reinforcement learning is to find the best policies, denoted π^*, but what denotes a good policy? Like in the dynamic programming case, we say that a policy π' is better than another policy π if the expected sum of discounted rewards are greater for π' than for π. This is equivalent to $J_{\pi'}(x) \geq J_\pi(x)$ for all $x \in X$, and if that is the case, we write $\pi' \geq \pi$. Thus, an optimal policy, denoted π^*, must satisfy

$$J_{\pi^*}(x) = \max_\pi J_\pi(x), \text{ for all } x \in X,$$

and

$$Q_{\pi^*}(x,u) = \max_{\pi} Q_\pi(x,u), \text{ for all } x \in X \text{ and } u \in U(x).$$

We can also rewrite the last equation in terms of v_{π_*} by

$$Q_{\pi^*}(x,u) = E[r_{t+1} + \alpha J_{\pi^*}(x_{t+1}) | x_t = x, u_t = u].$$

Now that we have an idea of what an optimal policy is we can proceed to take a look at some algorithms for estimating such optimal policies. Observe that many of the terms and equations are similar to what we are used with from the dynamic programming case.

Remark 6.1.1 (Notation in optimal control)*.* Note that we in the previous chapters have denoted the *cost* function for a given policy by J_π, while we in this chapter use the same notation in order to denote the *value* function of a policy. In principle, the idea of cost- and value functions are the same, J_π is some function that we want to optimize pointwise. When we are treating the signals received by the agent after each transition as *costs* we often want to minimize J_π, while we would like to maximize the same values if we refer to the signals as *rewards*. We have therefore opted to denote both the value and cost function by J_π in order to make it clear that in both cases the goal is to optimize the pointwise value of the function J_π, and that depending on the formulation of the problem, we either maximize or minimize. Observe also that any minimization problem, where we treat the signals received by the controller as costs, can be turned into an maximization problem where we interpret the signals as reward by changing the sign of the received signals. A last thing to note is that the notation used for reinforcement learning in this thesis is borrowed from dynamic programming in order to make the connection between DP and RL as clear as possible. In the RL literature the states are often denoted by s, actions by a and the value function of a policy is denoted by v_π. It is also common to use γ instead of α as the discount factor, while α is used to denote the learning rate in different RL methods. It is also useful to note the difference in language used. In reinforcement learning we often talk about agent, actions, rewards and value functions, while we in dynamic programming use controller, controls, costs and cost functions to refer to the same thing (if we disregard the fact that reward $= -$cost). A longer list of concepts in optimal control that has different names in RL and DP can be found in [Ber19]

6.2 Some reinforcement learning algorithms

In this section we take a look at the Q-learning and SARSA algorithms which are two reinforcement learning algorithms that are common in the literature. The introductions given here is inspired by the ones given in [SB18]. Note that policy iteration and value iteration that we covered in their abstract form in Section 4.3 also are viable methods for finding the optimal policy and the value function for an MDP, but we will not restate these methods here. Q-learning and SARSA are both temporal differences methods. This class of methods are able to learn from observations gained from simulations, and does not need a mathematical model of the dynamics of the environment in order to learn, in comparison to policy iteration and value iteration where we need to know the underlying model of the MDP in order to define the Bellman operator T.

Q-learning

Q-learning is an *off-policy* temporal differences control algorithm that approximates the optimal Q-function by use of sampling, often from simulations. That the method is an off-policy method means that the Q-values approximated by the method is not dependent on the policy used to pick actions during simulations. We start by initializing $Q(x, u)$ for all $x \in X, u \in U(x)$ to an arbitrary value, except for the terminal state for which $Q(terminal, \cdot) = 0$. We then train the algorithm on multiple samples, often called episodes in the RL literature. For each episode we start in some state $x_0 \in X$ and take some action $u_0 \in U(x_0)$ decided from some policy π based on the current Q-value estimates, e.g. the ϵ-greedy policy which with a small probability ϵ chooses a random action, and otherwise take the action for which the state-action pair has the maximum Q-value estimate. We then observe the reward r_0 and the next state x_1. We then repeat the same procedure by finding $u_1 \in U(x_1)$ using policy π and then observing r_1 and x_2. This process is repeated until we reach the terminal state of the episode. For each transition we update the Q-value with the following equation

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \gamma[r_{t+1} + \alpha \max_{u \in U(x_{t+1})} Q(x_{t+1}, u) - Q(x_t, u_t)], \tag{6.1}$$

where γ is the learning rate of the agent, and α is the discount rate.

SARSA

SARSA is in contrast to Q-learning an *on-policy* temporal difference algorithm. The two algorithms function in almost the same way, with the exception of a small change in the updating function of the Q-values. The minimization in (6.1) is substituted by $Q(x_{t+1}, u_{t+1})$, i.e. we use the Q-value of the state-action pair that consist of the next state x_{t+1} and the action our current policy π maps the state x_{t+1} to, which is u_{t+1}. That is why SARSA is called *on-policy*, as the updated Q-value depends on the policy π that we use to generate the state-action pairs. The Q-value updates thus take the form

$$Q(x_t, u_t) \leftarrow Q(x_t, u_t) + \gamma[r_{t+1} + \alpha Q(x_{t+1}, \pi(x_{t+1})) - Q(x_t, u_t)].$$

6.3 Similarities between reinforcement learning and dynamic programming

We have now shortly introduced reinforcement learning with emphasis on Markov Decision Processes as the underlying model. With this model assumed to be underlying we see that there is much resemblance between the theory of dynamic programming that we have covered in Chapters 3 to 5 and the reinforcement learning theory we have covered so far in this chapter. We are considering problems where we traverse a path of states, where the next state on the path is determined by a chosen control, also called action, and some random value w drawn from some distribution. We also pay costs, or receive rewards, whenever we transition from one state to the next. This is in essence equivalent to traversing an MDP. The problems we are trying to solve is then centred around finding an optimal policy that when applied to our control

problem results, on average, in the cumulated sum of costs/rewards being equal to the optimal expected cumulative sum of costs/rewards. That is, in DP we are often concerned with solving problems on the form

$$\min_{\pi\in\Pi} \underset{w\in W}{E}\left[\sum_{n\in\mathbb{N}} g(x_n, \pi(x_n), w_n)\right],$$

while we in the reinforcement learning context look at problems often formulated as

$$\max_{\pi\in\Pi} \underset{w\in W}{E}\left[\sum_{n\in\mathbb{N}} r(x_n, \pi(x_n), w_n)\right].$$

But in essence these problems are equivalent. In fact, the function we are optimizing can also be subject to change. We will therefore in the next section sometimes refer to a *target function*, which is just a way of referring to the expression we are optimizing. That is, in the usual DP case the target function is $E_{(w_0,w_1,\cdots)}\left[\sum_{n\in\mathbb{N}} g(x_n, \pi(x_n), w_n)\right]$.

Another similarity between RL and DP is that we are able to use value iteration and policy iteration in order to solve both kind of problems. But it is here the differences between DP and RL are starting to show. Methods such as Q-learning and SARSA are used in RL, but not in DP. That is because in DP we focus on methods that in the RL literature are called model-based methods, i.e. methods where the mathematical model of the environment is required. The up side of focusing on model-based methods is that we often are able to find exact solutions, and can therefore be certain that our solution is sound. The same can not be said for all RL methods, as many of them are dependent on approximations, simulations or simplifications. Therefore RL is sometimes called approximate dynamic programming in the literature, with the classic theory then being called exact dynamic programming. The short story is that we can in some sense consider dynamic programming as being the part of the reinforcement learning theory that considers problems were we are able to find exact solutions using model-based methods. We can also consider reinforcement learning as being an extension of dynamic programming with approximate methods, hence justifying the label approximate dynamic programming. If we for example are not able to obtain a model of the underling system governing the optimal control problem we are trying to solve we would need to use the model-free RL methods, as we would not be able to apply the model-based DP modes. Another situation where RL proves useful is when we have large state- and/or control spaces, as the complexity of value iteration and policy iteration can be huge when the cardinality of the state- and control space increases. As many RL methods only depend on simulations, which in general are less computationally demanding than VI and PI, we could use them in order to find a good policy for a problem where value iteration and policy iteration are not feasible solution methods.

We have so far in this thesis only covered methods that converge when the state and control space is finite. There exists several methods in order to deal with problems where we have infinite state- and/or control spaces, but we will not cover them in detail in this work. However, we will here give a short descriptions of some of the methods available, and the idea behind them. For the curious reader we referee to [Ber19]. If the state space is infinite there

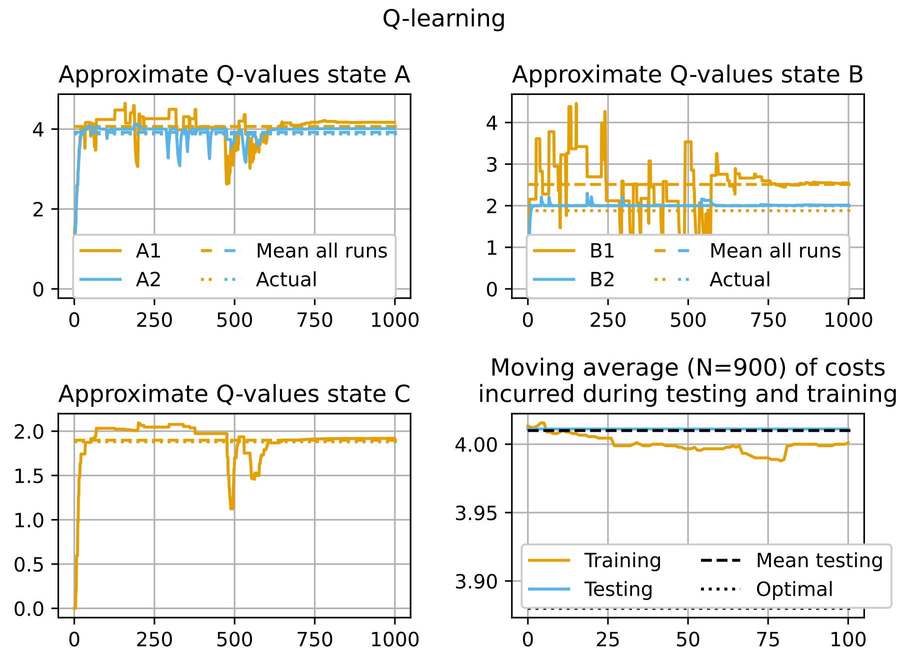

Figure 6.1: The evolution of Q-values and running average of simulated costs using Q-learning, as well as theoretical values and mean of sampled values.

exists several RL/approximate dynamic programming methods in order to approximate the values in the state space. This is often done by use of e.g regression or neural networks. The idea is that we assume some relationship between the cost at two states. If we for example assume a linear relationship between the cost at the different states, then we can find the cost at some states, and use regression in order to interpolate the cost at every other state. If we on the other hand have that the control space is e.g continuous we could approach the problem using policy parametrization. We then assume some parametrization of the policy, and try to find the optimal feature vector by use of for example gradient descent. These examples also explain some of the need for RL and approximate DP methods.

Let us now consider an example where we try to solve a problem that we earlier solved using exact methods, but now using Q-learning and SARSA, that in contrast to the DP methods, depend on simulations. That is, let us consider the problem that we in Example 5.1.9 solved using exact dynamic programming as we assumed that we had perfect information about the underlying model, an assumption we remove in the following example.

Example 6.3.1 (Continuation of Example 5.1.9). †

We can now take a second look at the SSP-problem depicted in Figure 5.1 that we considered in Example 5.1.9. In the previous example we assumed that we had perfect information. Let us now remove this assumption and try to solve the same problem using Q-learning and SARSA. The concrete problem we will try to solve is to find the policy that on average gives us the optimal cost

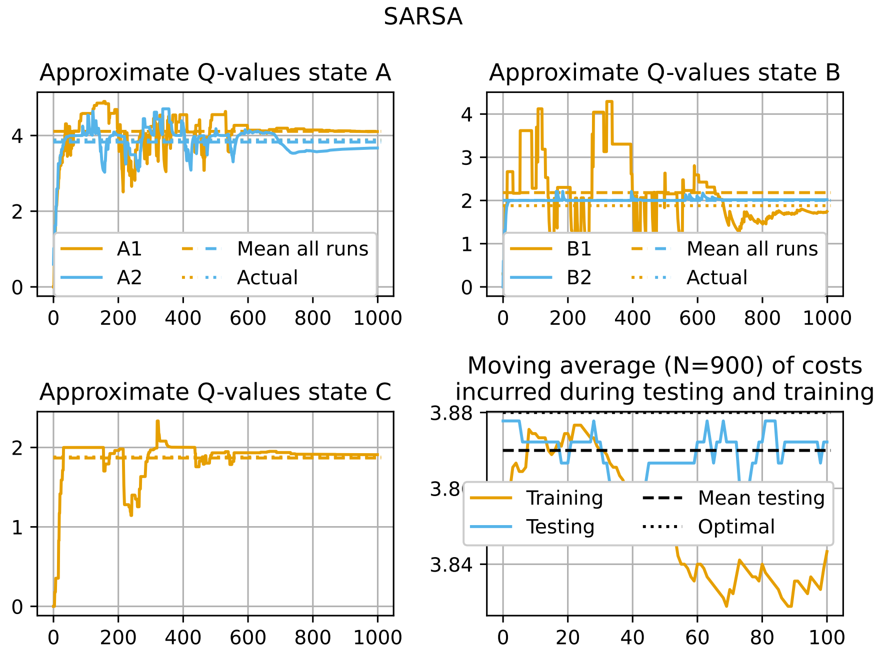

Figure 6.2: The evolution of Q-values and running average of simulated costs using SARSA, as well as theoretical values and mean of sampled values.

when traversing the stochastic shortest path shown in Figure 5.1 from state A to the terminal state t. We have implemented the two methods using the ϵ-greedy policy with initial parameter values of $\gamma = 0.3, \epsilon = 0.15,$ and $\alpha = 1$. After half of the training runs we let γ decrease by setting $\gamma = \frac{99}{100}\gamma$ between each run. The code for running the Q-learning and SARSA examples can be found in Listing A.4, while the code for implementing the graph itself can again be found in Listing A.1.

In order to approximate the Q-values and the optimal policy for the SSP-problem we first train the algorithm by using Q-learning as described above, except for the fact that the maximization in Equation (6.1) is switched out with a minimization as we in our problem formulation want to minimize cost and not maximize reward. We run 1000 rounds of training and record the Q-values of each state-action pair at the end of each round as well as the accumulated cost for the given round. In addition, for every 200 rounds of training we test the current policy. This is done by setting $\epsilon = 0$ such that we always choose the action with the highest Q-value in each state. We also lock the Q-values for the duration of the test runs such that the behaviour dictated by the policy does not change, as the Q-values stay fixed during the test runs. We run the current policy for 1000 rounds during each test and collect the accumulated cost for each round. Then, in order to approximate the expected cost of the policy generated by the Q-learning algorithm we run an additional 1000 rounds of testing with the final policy, i.e. the policy that in each state chooses the action with the minimal Q-value, where the Q-values are the once generated

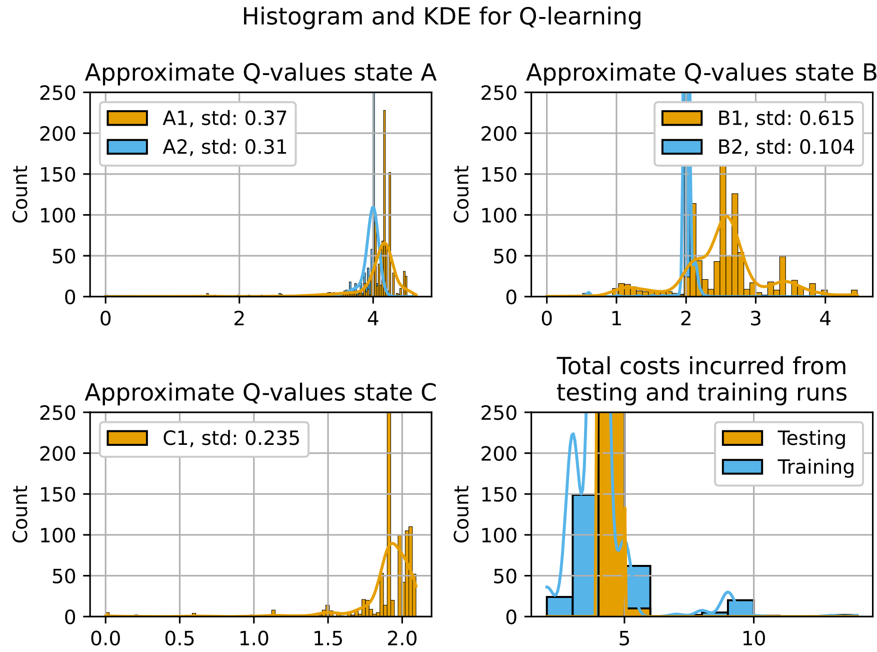

Figure 6.3: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for Q-learning. The height has been cut at $y = 250$.

by the algorithm after the 1000 rounds of training. During the final testing we again track the accumulated cost for each round.

We then plot in Figure 6.1 the evolution of the Q-values for the different state-action pairs after each of the training runs as well as the running average of the last 900 approximations of the expected cost for both the training runs and the testing runs of the final policy generated by the algorithm. This gives us a clear picture of how the approximation of the optimal policy evolves over the training period and from the plot depicting the running averages of the sampled total costs we also get an estimate of the expected cost of the final policy. We do exactly the same for SARSA which produces the plot shown in Figure 6.2. From the plots we see that only SARSA attain the optimal policy after 1000 rounds of training as action 1 yields the smallest approximated Q-value for SARSA when at state B and C while action 2 does the same for state A, which is exactly the same as the optimal policy we found in Example 5.1.9. However, we note that during some periods of the training we actually have an approximation for the optimal policy that in fact is not the optimal policy. These periods are the once where we have $Q(B, 2) < Q(B, 1)$. For Q-learning we see that it does not correctly approximate the Q-values. This fact showcases the possible instability of these approximate dynamic programming methods.

The actual Q-value estimates for both Q-learning and SARSA after the training rounds is shown in Table 6.1. By looking at Table 5.1, with the results from Example 5.1.9, we see that the Q-values for some of the states match

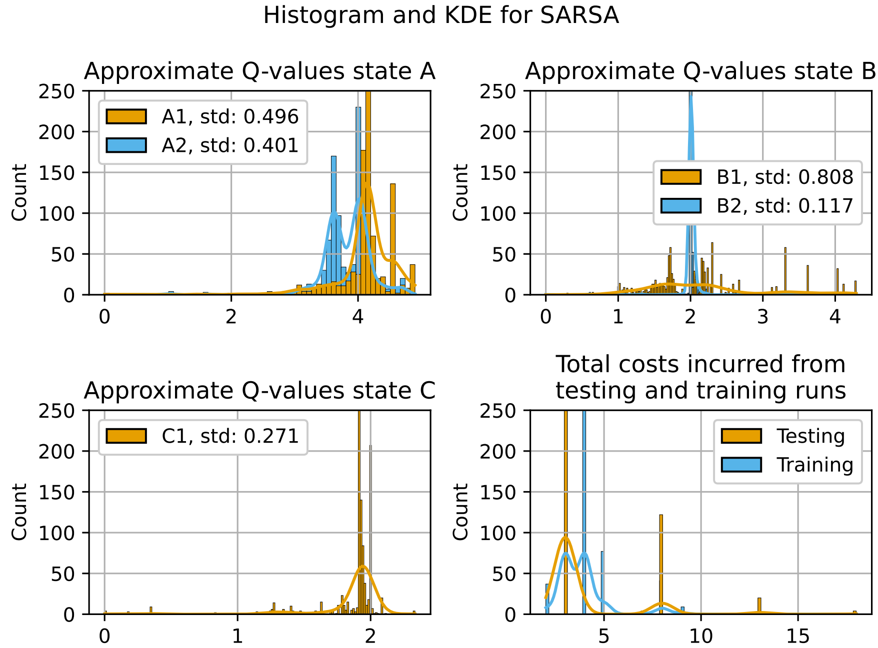

Figure 6.4: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for SARSA. The height has been cut at $y = 250$.

Method and value	$s = A$	$s = B$	$s = C$
Q-learning $Q(s, 1)$	4.17	2.54	1.92
Q-learning $Q(s, 2)$	4.01	2.01	
SARSA $Q(s, 1)$	4.11	1.74	1.91
SARSA $Q(s, 2)$	3.67	2.02	
Actual $Q(s, 1)$	3.88	1.88	1.88
Actual $Q(s, 2)$	3.88	2	

Table 6.1: Q-values estimated by Q-learning and SARSA for SSP depicted in Figure 5.1.

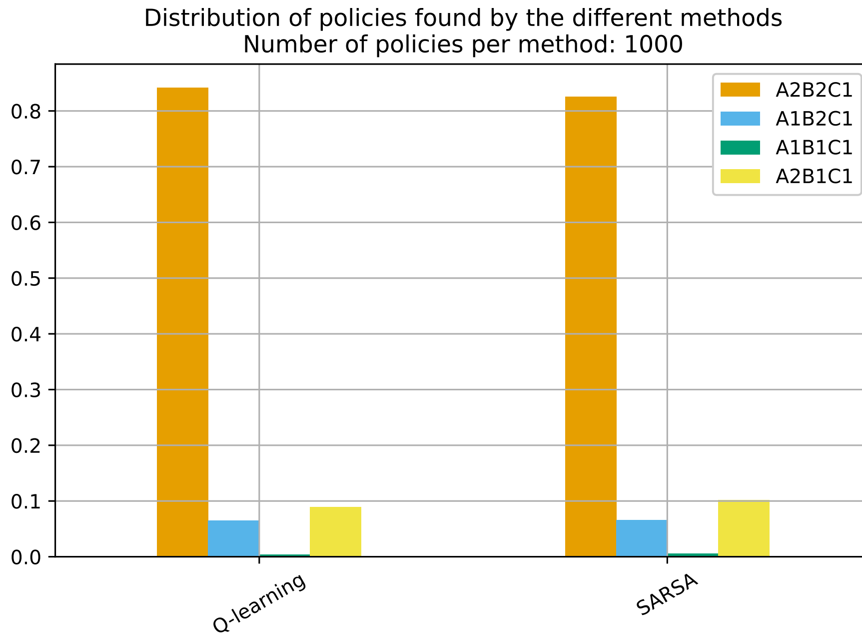

Figure 6.5: Bar plot showing distribution of policies created with Q-learning and SARSA. Legend notation explanation: A1B1C1 denotes the policy taking action 1 in each state.

quite well, while the methods are less accurate for others. We can see the same by looking at the deviation from the final Q-value approximations and the actual Q-values in Figures 6.1 and 6.2. It can also be interesting to look at the variability of the Q-value estimates and the distribution of the accumulated costs that the final policy incurred during testing as well as the costs we sampled during training. We therefore plot a histogram for the Q-values and the observed costs for Q-learning in Figure 6.3 and for SARSA in Figure 6.4. We also plot the kernel density estimate for the values. We note that for both methods the Q-value $Q(B, 1)$ had the largest standard deviation. This is probably due to the fact that we under action 1 in state B have a $\frac{3}{20}$ chance of receiving a cost of 5 which is significantly higher than any other transition cost in the SSP. Thus, the Q-value approximation of $Q(B, 1)$ will increase by a relative high amount each time this cost is observed during training, hence increasing the variability of the approximation. We can also see from the histogram of the incurred costs that the policy learned by the Q-learning method more or less only accumulated a cost of 4 or 5 units during a run, while the optimal policy correctly approximated by SARSA most often receive a cost of 3. However, the policy learned by SARSA have a significant peak around a cost of 8 units and another smaller peak around 13. Thus the optimal policy have a non-zero chance of yielding relatively high costs when traversing the SSP.

In this example we only look in detail on a single run for both Q-learning and SARSA. As the RL methods we look at here rely on simulations in order to generate a policy the results of a single run are prone to randomness and

we are not guaranteed that running the same algorithm again would result in the same policy being generated. In order to get an idea about how the methods perform on average we generate another 1000 policies with both of the methods and collect the generated policy after each round of training. We then plot a bar plot in Figure 6.5 that depicts the distribution of the policies generated by each method. We see that the two methods usually generate the policy that takes action 2 in state A and B, and action 1 in state C, which was the policy generated by Q-learning in the run we considered earlier. It is also interesting to note that the distribution of policies are quite similar for both methods, and that the actual optimal policy for this problem, which is A2B1C1, only is generated about once in ten runs. Keep in mind that these results are dependent on our choice of parameter values for the learning rate γ, discount rate α and the exploration rate ϵ. We will however not look into the effect of these parameters in this work.

In the next chapter we take a look risk-sensitive optimal control by considering some methods that try to reduce the variation of the cost incurred, typically by generating policies that are avoiding state-control pairs like $(B, 1)$ that has a probability of generating relatively large costs.

CHAPTER 7

Risk-sensitive control

In the usual form of dynamic programming and reinforcement learning we are trying to optimize the *expected* value of a given target function. This is however not always desirable. In some cases it would be better to have a less desirable expected value for our target function, if we were able to, for example, make the probability of extreme losses (almost) non-existing. An example could be deciding on a strategy to save money. We could either invest our savings in stocks and financial derivatives, or we could simply put the money into a savings account. We would expect our savings to grow slower in a savings account, but we would on the other hand avoid the price volatility of the financial market. This means that by putting our money into a savings account we have a lower expected gain, but we trade that for a lower variability and thus a more predictable return. In other words, we have a savings solution with lower *risk*. The idea behind risk sensitive control is to include the minimization of risk as an optimization criterion together with the classical notion of optimizing the expected value. Thus, an optimal policy in the context of risk sensitive control is not necessarily a policy with an expected value equal to the optimal expected value, but would need to have a trade-off between risk and the expected cost/reward. We could therefore in risk sensitive control allow for an expected value worse than the optimal if the risk associated with the policy is lower that what we would have with a policy considered optimal in the regular DP and RL context. This is analogues to how a risk-sensitive person that is risk-avoiding would accept the lower expected gain associated with putting his money in a savings account as that is less risky than the financial market, while a risk-neutral person would not hesitate with entering the financial market with her savings as the expected gain is higher. There exist many different methods to include risk in the optimization criterion and we will in this chapter discuss a few of them. Some methods depend on a model-based framework while others use reinforcement learning techniques, and a few are based on ideas that are compatible with both model-based and model-free frameworks. We start this chapter with a section devoted to give some motivation for risk sensitive control before we move on to introducing some of the model-based risk-sensitive control methods. We then continue with a section on model-free risk-sensitive control methods, and then we end the chapter with a discussion and comparison of the methods presented in this chapter.

7.1 Motivation for risk-sensitive control

In this section we look at an example taken from [Heg94] that illustrates the motivation for risk sensitive control.

Example 7.1.1 (St. Petersburg Paradox, motivation for risk-sensitive control). The example considered here is taken from [Heg94]. Consider a situation where someone offers us to play a game. The rules of the game is that we put in a bet of k units, and then a fair coin is tossed over and over again until it hit the ground with heads pointing upwards. We then receive 2^n units, where n is the number of tosses we needed for a heads to appear. That is, if we get heads on the second toss, we win $2^2 = 4$ units. The question is then, how much would we be willing to pay to play this game, or would we not be willing to play at all? If we use the expected payoff to decide whether to play or not, we see that not playing will result in an expected gain of 0. While for playing the game we expect the payoff X to be

$$E[X] = \sum_{n=1}^{\infty} \frac{1}{2^n} 2^n - k = \sum_{n=1}^{\infty} 1^n - k = \infty.$$

Note that $\frac{1}{2^n}$ is the probability that the nth toss is the first showing heads. We can then conclude that we would, following the *maximize expected value* criterion, always choose to play the game. It even turns out that we would be willing to put in any finite stake of k units that is asked of us to play the game, since our expected gain is infinite. This is however not sensible, as most people would not be willing to put in any arbitrary amount of units.

We will in the following chapters look at different methods that try to take risk into account in order to find policies that we would find more sensible than those found by only considering the optimal expected value criterion. In the context of Example 7.1.1 we saw that the optimal policy found when only considering the optimal expected value would always play, no matter the initial stake. Depending on your tolerance for risk the optimal policy, when considering both the expected value and the risk, would be something like

$$\pi^*(k) = \begin{cases} \text{take bet} & k \leq c, \\ \text{do not take bet} & k > c, \end{cases}$$

where k is the initial stake in order to play the game, while c is some finite number that indicate what we would be willing to bet to enter the game and is thus dependent on the level of risk that we tolerate. In the next section we start by looking at some model-based approaches before continuing on with some model-free methods.

7.2 Model-based risk-sensitive control

This section set out to introduce some classical ways of handling risk in a dynamic programming setting with perfect information. We will look at minimax, the utilization of a exponential utility function, and the introduction of so called error states, as ways to find optimal policies in a risk-sensitive sense.

Minimax

This subsection is based on [Heg94], [Ber17] and [Ber18]. In some situations even small probabilities of extreme outcomes as a result of following a particular policy is considered to be too severe to be acceptable. In such cases we could be tempted to always consider the worst possible outcome of choosing a given control as the outcome we would need to be willing to accept in order to pick that given control. The controller should then optimize over the worst-case outcomes for each action. This is the idea behind minimax where we minimize over the maximal possible cost of the available controls in each state. When we deal with rewards instead of costs, the analogue is called maximin and is when we are trying to find a policy that is maximizing the minimal possible reward gained by applying the policy. The problem of finding the optimal policy for the minimax problem can in the finite-horizon case be rigorously stated as

$$\min_{\pi\in\Pi} \max_{\substack{w_k\in D_k(x_k,\mu_k(x_k)),\\ k=0,1,\dots,N-1}} \left(g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k,\mu_k(x_k),w_k) \right). \tag{7.1}$$

In [Ber17] the author shows that the following DP algorithm, which is the minimax analogue of the algorithm presented in Section 3.2, finds the optimal cost function of a finite-horizon minimax problem, and is thus the cost function J_π of each policy π satisfying Equation (7.1). The algorithm takes the form

$$J_N(x_N) = g_N(x_N), \tag{7.2a}$$

$$J_k(x_k) = \min_{u_k\in U_k(x_k)} \max_{w_k\in D_k(x_k,u_k)} (g_k(x_k,u_k,w_k)+J_{k+1}(f_k(x_k,u_k,w_k))). \tag{7.2b}$$

Let us now apply this method on the problem we considered in Example 7.1.1

Example 7.2.1 (Maximin/Minimax, continuation of Example 7.1.1)**.** †
We can now apply the maximin algorithm, by switching min and max in Equation (7.2), in order to find a more sensible policy for the problem considered in Example 7.1.1. We have only one state denoted by x, and the process lasts for one single period, so we have $N = 1$. In addition, we have just two possible controls which is to play or to not play, denoted by u_p and u_n, respectively. If we choose to not play our reward is 0, implying that $g(x, u_n, \cdot) = 0$, while if we choose to play and pay a cost $k \in [0, \infty)$ we get the price 2^n with probability $p = \frac{1}{2^n}$ where $n \in \mathbb{N} \setminus \{0\}$. Thus we receive the reward $2^n - k$ with probability $p = \frac{1}{2^n}$, i.e. the probability of the random noise $w \in W(x, u_p)$ having a value satisfying $g(x, u_p, w) = 2^n - k$ is $\frac{1}{2^n}$, written compactly as

$$P(\{w \in W(x,u_p)\colon g(x,u_p,w) = 2^n - k\}) = \frac{1}{2^n} \text{ for each } n \in \mathbb{N}\setminus\{0\},$$

where P is the probability measure of the relevant probability space. Note that we have no deterministic final cost, and therefore $g_N(x) = 0$. This gives

$$\begin{aligned} J^*(x) &= \max_{u\in\{u_p,u_n\}} \min_{w\in W(x,u)} g(x,u,w) \\ &= \max\left\{ \min_{n\in\mathbb{N}\setminus\{0\}} 2^n - k,\ g(x,u_n,\cdot) \right\} \end{aligned}$$

$$= \max\{2 - k,\, 0\}$$
$$= \begin{cases} 0 & kn > 2, \\ 2 - k & k \leq 2. \end{cases}$$

Note that the reward of 0 is associated with the control u_n, while the reward $2 - k$ is achieved by applying the control u_p. Therefore, we have that

$$\pi^*(x) = \begin{cases} u_n & k > 2 \\ u_p & k \leq 2. \end{cases}$$

Thus we see that when using the maximin criterion we should opt not to play if it cost more than 2 units, while we should play if the stake is 2 units or less. This makes sense as we are always guaranteed a price of at least 2 units, thus assuming that the odds are always *not* in our favour the optimal maximin policy will only choose to play if $k \leq 2$. As expected the maximin optimal policy is more sensible than the optimal policy found by considering the optimal expected value criterion as we now do not allow for arbitrarily large bets, but only smaller bets that is of a size that guarantees that we at least get our money back, and when $k = 2$, we double our initial investment with probability $p = \frac{1}{2}$, while for $k < 2$ we are guaranteed a profit. Sadly, we would probably never find anyone willing to let us play the game for a stake $k < 2$.

For the infinite-horizon case we can turn to the theory of Chapter 4. In order to use the abstract dynamic programming notation from Chapter 4 we define

$$H(x, u, J) = \sup_{w \in W(x,u)} \left(g(x, u, w) + \alpha J(f(x, u, w)) \right), \tag{7.3}$$

with $\alpha \in (0, 1)$. We see that Equation (7.3) is more or less the same expression we are minimizing in Equation (7.2b), with the exception of H being stationary and discounted, as well as the change from max to sup. The mapping H gives rise to our usual operators T_μ and T, defined by

$$(T_\mu J)(x) = H(x, \mu(x), J), \tag{7.4}$$

for each stationary policy $\mu \in \mathcal{M}$ and

$$(TJ)(x) = \min_{u \in U(x)} H(x, u, J).$$

We would then like to show that the optimal cost function J^* satisfies the Bellman equation $J^* = TJ^*$ and that we are able to apply the methods presented in Section 4.3 in order to find an optimal policy. We know from Chapter 4 that this is the case if the operator T_μ is a contraction for each $\mu \in \mathcal{M}$ and if the mapping H satisfies the monotonicity property. We start with showing that H in fact is monotone.

Proposition 7.2.2 (Minimax mapping H is monotone)**.** †

The mapping $H : X \times U \times \mathcal{R}(X) \to \mathbb{R}$, as defined in Equation (7.3), is monotone, i.e. for all functions $J, J' \in \mathcal{R}(X)$, with $J(x) \leq J'(x)$ for all $x \in X$, we have that

$$H(x, u, J) \leq H(x, u, J') \text{ for all } x \in X \text{ and } u \in U. \tag{7.5}$$

Proof. Assume that $J, J' \in \mathcal{R}(X)$ with $J \leq J'$ pointwise. We then need to prove that Equation (7.5) holds, which is equivalent to showing that

$$H(x,u,J) - H(x,u,J') \leq 0.$$

We see that

$$\begin{aligned} H(x,u,J) - H(x,u,J') &= \sup_{w \in W(x,u)} \left(g(x,u,w) + \alpha J(f(x,u,w))\right) \\ &\quad - \sup_{w \in W(x,u)} \left(g(x,u,w) + \alpha J'(f(x,u,w))\right) \\ &\leq \sup_{w \in W(x,u)} \left(g(x,u,w) + \alpha J(f(x,u,w)) \right. \\ &\qquad \left. - g(x,u,w) - \alpha J'(f(x,u,w))\right) \\ &= \sup_{w \in W(x,u)} \left(g(x,u,w) - g(x,u,w) \right. \\ &\qquad \left. + \alpha(J(f(x,u,w)) - J'(f(x,u,w)))\right) \\ &= \sup_{w \in W(x,u)} \alpha(J(f(x,u,w)) - J'(f(x,u,w))) \\ &\leq \alpha C \\ &\leq 0, \end{aligned}$$

where

$$C = \sup_{w \in W(x,u)} (J(f(x,u,w)) - J'(f(x,u,w))) \leq \sup_{x \in X} (J(x) - J'(x)) \leq 0,$$

as we have assumed that $J(x) \leq J'(x)$ for all $x \in X$. Then, as we have showed that $H(x,u,J) - H(x,u,J') \leq 0$, we have that H is monotone. ■

We can then move on to show that the operator T_μ, as defined in Equation (7.4), is a contraction for each stationary policy $\mu \in \mathcal{M}$.

Proposition 7.2.3 (Minimax operator T_μ is a contraction)**.** †
The operator T_μ as defined in Equation (7.4) *is a contraction on the space $\mathcal{B}(X)$ for each stationary policy $\mu \in \mathcal{M}$, that is, for some arbitrary cost functions $J, J' \in \mathcal{B}(X)$, and some positive number $\rho < 1$, we have that*

$$\|T_\mu J - T_\mu J'\|_v \leq \rho \|J - J'\|_v,$$

where the norm $\|\cdot\|_v$ is as defined in Definition 4.1.3 and where $v : X \to \mathbb{R}$ is some positive function.

Proof. Let $J, J' \in \mathcal{R}(X)$ be some arbitrary cost functions and let $v(x) = 1$ for each $x \in X$. We then see that

$$\begin{aligned} (T_\mu J)(x) - (T_\mu J')(x) &= H(x,\mu(x),J) - H(x,\mu(x),J') \\ &= \sup_{w \in W(x,u)} \left(g(x,\mu(x),w) + \alpha J(f(x,\mu(x),w))\right) \\ &\quad - \sup_{w \in W(x,u)} \left(g(x,\mu(x),w) + \alpha J'(f(x,\mu(x),w))\right) \\ &\leq \sup_{w \in W(x,u)} \alpha(J(f(x,u,w)) - J'(f(x,u,w))) \end{aligned}$$

$$
\begin{aligned}
&= \sup_{w \in W(x,u)} \alpha v(f(x,u,w)) \frac{(J(f(x,u,w)) - J'(f(x,u,w)))}{v(f(x,u,w))} \\
&\leq \sup_{w \in W(x,u)} \alpha v(f(x,u,w)) \|J - J'\| \\
&= \alpha v(x) \|J - J'\|,
\end{aligned}
$$

where we in the last equality have used that $v(x) = v(f(x,u,w)) = 1$. The above calculation implies that

$$
\frac{(T_\mu J)(x) - (T_\mu J')(x)}{v(x)} \leq \alpha \|J - J'\|.
$$

We can then switch the roles of J and J' to show that the inequality

$$
\frac{(T_\mu J')(x) - (T_\mu J)(x)}{v(x)} \leq \alpha \|J - J'\|,
$$

holds. If we now maximize both sides of the inequities we see that

$$
\|T_\mu J - T_\mu J'\| \leq \alpha \|J - J'\|.
$$

Then, as $\alpha \in (0,1)$, and as the choice of $\mu \in \mathcal{M}$ is not important, we see that T_μ indeed is a contraction for each policy $\mu \in \mathcal{M}$ with modulus $\rho = \alpha$. ■

We have then showed that the operator T_μ is a contraction for each stationary policy $\mu \in \mathcal{M}$ and that the mapping H is monotone. We then know from Chapter 4 that the optimal cost function J^* satisfies

$$
J^* = \inf_{\pi \in \Pi} J_\pi = \inf_{\mu \in \mathcal{M}} J_\mu = TJ^*,
$$

that is, J^* is a fixed point of Bellman's equation $J^* = TJ^*$ and an optimal policy is possible to find among the stationary policies $\mu \in \mathcal{M}$. We then also have that we are able to use the methods presented in Section 4.3 in order to find both an optimal policy and the optimal cost function J^* for the discounted minimax problem.

As we see from Equation (7.1) the actual probability distributions of the random values w_k are actually not needed in order to solve the minimax problem. We only need to have knowledge about which values the random noise can take when we in a given state x pick the control u, along with the cost function g and state transition function f. Therefore, in a situation where we in some real-world application are not able to find sufficiently accurate estimates of the probability distributions for the random quantities w_k, but are only able to find bounds on the values of the random noise, we could use minimax in order to find a safe policy. We would then avoid the risk inherited from depending on bad estimates of the probability distributions and would end up with a policy where we are able to get some idea of the worst-case outcome of applying the policy in question.

Exponential utility function

This subsection is based on the paper [MN02] as well as [HM72] and [Ber18]. Even though the minimax/maximin method introduced in Section 7.2 is a step

on the way towards dealing with the problem of risk-neutrality when using the optimal expected value criterion, as illustrated in Example 7.1.1, it has some drawbacks. In Example 7.2.1 we saw that using maximin instead of the regular optimization criterion gave a more sensible policy compared to the one found in Example 7.1.1. However, a player that allow for some risk could for instance be willing to put up, lets say, a stake of $k = 3$. This sounds reasonable as the chance of suffering a loss is $p = \frac{1}{2}$ while the probability of a gain is the same. The exponential utility function approach to risk-sensitivity allows for this kind of slight risk-averse behaviour. It includes a parameter β that represents how risk-averse or -seeking the agent is and allows for behaviour that does not always assumes the worst-case outcome and that still is not risk-neutral.

The idea behind using an exponential utility function is to transform the cost function such that we can control the variation of our optimal policy π^*. We do this in the finite-horizon case by changing the goal from finding a policy $\pi^* = \{\mu_0, \cdots, \mu_{N-1}\} \in \Pi$ that satisfies

$$J_{\pi^*}(x_0) = \min_{\pi\in\Pi} E\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right],$$

into finding the policy that satisfies

$$J_{\pi^*}(x_0) = \min_{n\pi\in\Pi} \frac{1}{\beta} \log E\left[e^{\beta\left(g_N(x_N)+\sum_{k=0}^{N-1} g_k(x_k,\mu_k(x_k),w_k)\right)}\right]. \tag{7.6}$$

The value β in (7.6) is a risk aversion coefficient. To see why it is preferable to optimize the function in Equation (7.6) when we would like to control the variability of our cost, we can write out the Taylor expansion of the expression. We start by writing out the second order Taylor expansion of $f(x) = e^{\beta x}$ around the point $a = E[G]$, where

$$G \coloneqq g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k).$$

We have that (†)

$$\begin{aligned} T_2 f_{E[G]}(x) &= f(E[G]) + f'(E[G])(x - E[G]) + \frac{f''(E[G])}{2}(x - E[G])^2 \\ &\qquad + \frac{f'''(c)}{6}(x - E[G])^3 \\ &= e^{\beta E[G]} + \beta e^{\beta E[G]}(x - E[G]) + \frac{\beta^2 e^{\beta E[G]}}{2}(x - E[G])^2 \\ &\qquad + \frac{\beta^3 e^{\beta c}}{6}(x - E[G])^3, \end{aligned}$$

where $c \in (E[G], x)$. Then by taking the expectation of $e^{\beta G}$ we get

$$\begin{aligned} E\left[e^{\beta G}\right] &= E\left[T_2 f_{E[G]}\right] \\ &= E\Bigg[e^{\beta E[G]} + \beta e^{\beta E[G]}(G - E[G]) + \frac{\beta^2 e^{\beta E[G]}}{2}(G - E[G])^2 \\ &\qquad + \frac{\beta^3 e^{\beta c}}{6}(G - E[G])^3\Bigg] \end{aligned}$$

$$
\begin{aligned}
&= E\left[e^{\beta E[G]}\right] + E\left[\beta e^{\beta E[G]}(G - E[G])\right] + E\left[\frac{\beta^2 e^{\beta E[G]}}{2}(G - E[G])^2\right] \\
&\qquad + E\left[\frac{\beta^3 e^{\beta c}}{6}(G - E[G])^3\right] \\
&= e^{\beta E[G]} + \beta e^{\beta E[G]}(E[G] - E[E[G]]) + \beta^2 \frac{e^{\beta E[G]}}{2} E\left[(G - E[G])^2\right] \\
&\qquad + \beta^3 \frac{e^{\beta c}}{6} E\left[(G - E[G])^3\right] \\
&= e^{\beta E[G]} + \beta e^{\beta E[G]}(E[G] - E[G]) + \beta^2 \frac{e^{\beta E[G]}}{2} \mathrm{Var(G)} \\
&\qquad + \beta^3 \frac{e^{\beta c}}{6} E\left[(G - E[G])^3\right] \\
&= e^{\beta E[G]} + \beta^2 \frac{e^{\beta E[G]}}{2} \mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6} E\left[(G - E[G])^3\right].
\end{aligned}
$$

We can then take the logarithm of expression above, giving us

$$
\begin{aligned}
\log\left(E\left[e^{\beta G}\right]\right) &= \log\left(e^{\beta E[G]} + \beta^2 \frac{e^{\beta E[G]}}{2} \mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6} E\left[(G - E[G])^3\right]\right) \\
&= \log\left(e^{\beta E[G]}\left(1 + \frac{\beta^2}{2}\mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6e^{\beta E[G]}} E\left[(G - E[G])^3\right]\right)\right) \\
&= \log\left(e^{\beta E[G]}\right) \\
&\qquad + \log\left(1 + \frac{\beta^2}{2}\mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6e^{\beta E[G]}} E\left[(G - E[G])^3\right]\right) \\
&= \beta E[G] + \log\left(1 + \frac{\beta^2}{2}\mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6e^{\beta E[G]}} E\left[(G - E[G])^3\right]\right).
\end{aligned}
\tag{7.7}
$$

The next step is to find the first order Taylor expansion of $g(x) = \log(x)$ around the point $a = 1$, which is given by

$$
T_1 g_1(x) = \log(1) + g'(1)(x - 1) + \frac{g''(d)}{2}(x - 1)^2 = x - 1 - \frac{(x - 1)^2}{2d^2},
$$

where $d \in (1, x)$. Thus, by using the Taylor approximation of $\log(x)$ in Equation (7.7), we see that

$$
\begin{aligned}
\log\left(E\left[e^{\beta G}\right]\right) &= \beta E[G] + \Bigg(1 + \frac{\beta^2}{2}\mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6e^{\beta E[G]}} E\left[(G - E[G])^3\right] - 1 \\
&\qquad - \frac{\left(1 + \frac{\beta^2}{2}\mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6e^{\beta E[G]}} E\left[(G - E[G])^3\right] - 1\right)^2}{d^2}\Bigg) \\
&= \beta E[G] + \frac{\beta^2}{2}\mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6e^{\beta E[G]}} E\left[(G - E[G])^3\right] \\
&\qquad - \frac{\left(\frac{\beta^2}{2}\mathrm{Var}(G) + \beta^3 \frac{e^{\beta c}}{6e^{\beta E[G]}} E\left[(G - E[G])^3\right]\right)^2}{d^2}.
\end{aligned}
$$

This results in

$$\begin{aligned}\frac{1}{\beta}\log E\left[e^{\beta G}\right] &= E[G] + \frac{\beta}{2}\text{Var}(G) + \beta^2\frac{e^{\beta c}}{6e^{\beta E[G]}}E\left[(G-E[G])^3\right] \\ &\qquad - \frac{\left(\frac{\beta^2}{2}\text{Var}(G) + \beta^3\frac{e^{\beta c}}{6e^{\beta E[G]}}E\left[(G-E[G])^3\right]\right)^2}{\beta d^2} \\ &= E[G] + \frac{\beta}{2}\text{Var}(G) + \mathcal{O}(\beta^2). \end{aligned} \tag{7.8}$$

Then it is clear that when we consider a minimization of cost problem we have that with increased values of β we punish variation of the cost more strongly. We also see that when $\beta \to 0$, we have that optimizing the function in Equation (7.6) becomes equivalent with optimizing the risk-neutral expected cost. Note also that by having $\beta < 0$ we get the opposite situation. That is, we then prefer policies with a higher variation in the cost, i.e, we then seek more risk taking policies. In short, we have that

- $\beta \to 0$: Risk-neutral,

- $\beta > 0$: Risk-averse,

- $\beta < 0$: Risk-seeking.

Keep in mind that for a maximizaiton problem it is the other way around. That means that then $\beta > 0$ promotes risk-seeking policies, while $\beta < 0$ would give more risk-averse policies.

Let us now take a look at an example where we revisit the St. Petersburg paradox in order to illustrate the benefits of the exponential utility function for avoiding policies with a higher risk.

Example 7.2.4 (Exponential utility function, continuation of Example 7.1.1)**.** † Again, consider the example with the coin tossing game. We now try to use an exponential utility function to decide on whether to play or not. The target function that our policy should maximize is

$$\frac{1}{\beta}\log\sum_{n=1}^{\infty}\frac{1}{2^n}e^{\beta(2^n-k)}.$$

If we now fix $\beta := -1$, we can try to find for which value of k we would be as willing to put in a bet of k units as not to play the game. That is, the value of k for which

$$-\log\sum_{n=1}^{\infty}\frac{1}{2^n}e^{-(2^n-k)} = 0. \tag{7.9}$$

We find that for $k = 2.62709$, equation (7.9) is satisfied. Thus, if we tried to optimize the target function (7.6), we should follow the policy that play the game if we only need to put in a stake lower than 2.62709 units, while we would opt to not play if $k > 2.62709$. This was the case when $\beta = -1$, but how does the value of k for which we would be willing to play change with β? We try to illustrate this in Figure 7.1 (see Listing A.10 for plot generating code), where

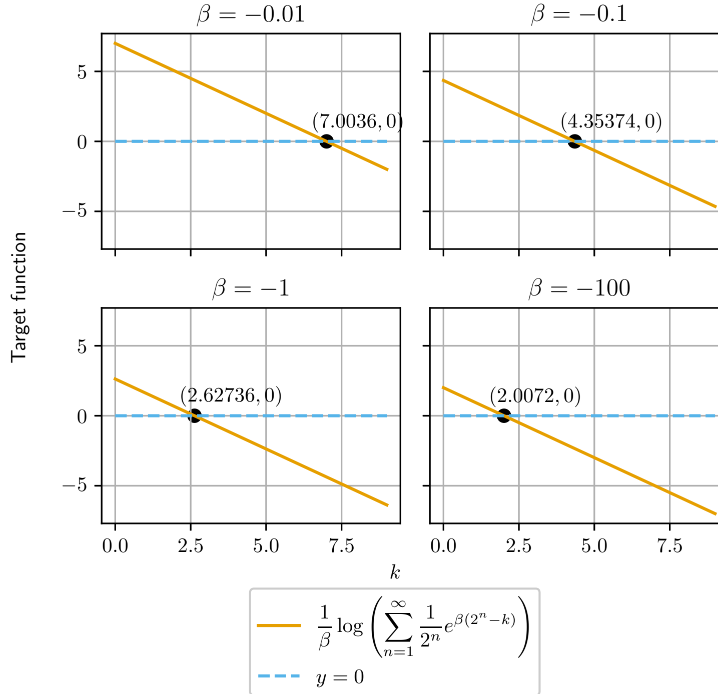

Figure 7.1: Illustrating the effect of β when using exponential utility function.

we visualise how the target function value changes with k for some different values of β. We have also drawn the line $y = 0$ and marked the intersection between the two curves, which marks the highest stake value k for which we would be willing to play for the different values of β. We see from Figure 7.1 that for $\beta = -0.01$ we are willing to play if $k < 7$, while for $\beta = -100$ we are only willing if $k \leq 2$. Since we are guaranteed a payoff of at least 2 units, it seems reasonable that the equilibrium point, that is the value of k for which we are equally willing to do both actions, approaches 2 as we seek more and more risk-averse policies.

It would also be interesting to figure out how the target value varies for different values of β for fixed values of k. We illustrate this in Figure 7.2. From the figure we see that when $\beta \to 0$ the target value also increases rapidly. This is expected, as we know that the limit for the target value is infinite when $\beta \to 0$. We also note that for $k = 1, 2$ the target value is above zero for each β. This is expected, as we noted earlier that we are guaranteed to always receive 2 units. We see from the two other plots that we need a negative β that is quite close to zero for preferring to take the bet when we have $k \geq 4$, which again is reasonable, as the probability for receiving 4 units or more is $\frac{1}{2}$.

For the infinite-horizon case we again turn to the notation introduced in

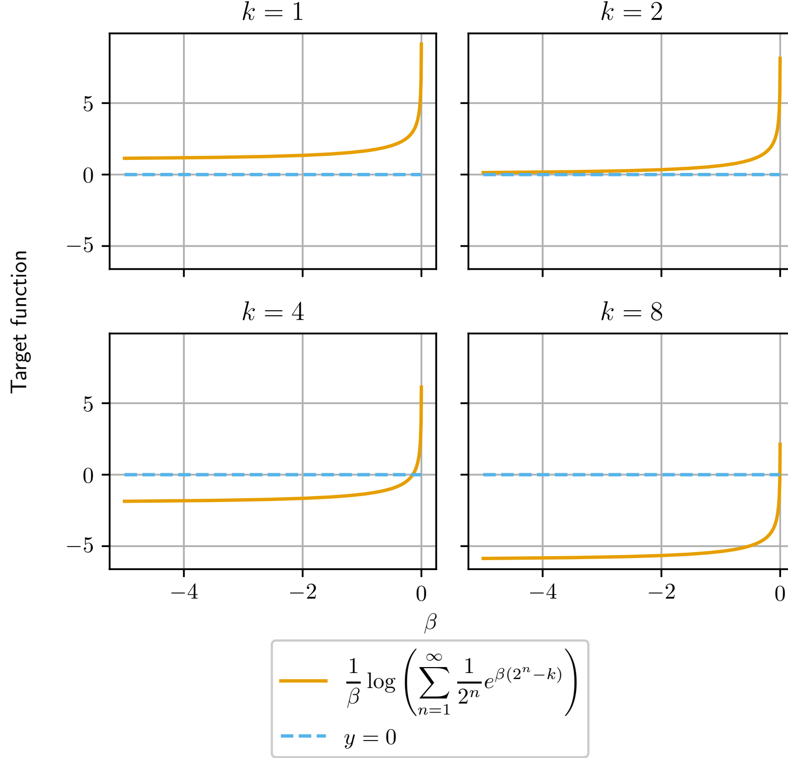

Figure 7.2: Illustrating the effect of k when using exponential utility function.

Chapter 4. The mapping H takes the form

$$H(x,u,J) = E\left[e^{\beta g(x,u,w)}J(f(x,u,w))\right]. \tag{7.10}$$

It is shown in [Ber18] that the multiplicative form of the mapping H implies that

$$\begin{aligned}
J_\pi(x_0) &= \limsup_{N\to\infty}(T_{\mu_0}T_{\mu_1}\cdots T_{\mu_{N-1}}J)(x_0)\\
&= \limsup_{N\to\infty} E\left[e^{\beta g(x_0,\mu_0(x_0),w_0)}\cdots e^{\beta g(x_{N-1},\mu_{N-1}(x_{N-1}),w_{N-1})} \mid x_0\right]\\
&= \limsup_{N\to\infty} E\left[e^{\beta(g(x_0,\mu_0(x_0),w_0)+\cdots+g(x_{N-1},\mu_{N-1}(x_{N-1}),w_{N-1}))} \mid x_0\right],
\end{aligned}$$

where $x_{n+1} = f(x_n,\mu_n(x_n),w_n)$ and the expectation is taken with respect to $(w_0,\ldots,w_{N-1})$.

We again want to prove that we are able to write the optimal cost function J^* as the solution of Bellman's equation $J^* = TJ^*$. In order to show that the mapping H, as defined in Equation (7.10), is monotone, and that the operators T and T_μ induced by the mapping H are contractions. As usual the operators

are defined by

$$(T_\mu J)(x) = H(x, \mu(x), J), \tag{7.11}$$

for some stationary policy $\mu \in \mathcal{M}$, and

$$(TJ)(x) = \min_{u \in U(x)} H(x, u, J).$$

Note that the usual definition of the cost function of a policy π would, as shown above, give us

$$J_\pi(x_0) = \limsup_{N \to \infty} E\left[e^{\beta(g(x_0,\mu_0(x_0),w_0)+\cdots+g(x_{N-1},\mu_{N-1}(x_{N-1}),w_{N-1}))} \mid x_0\right],$$

which is not equal to the expression we are minimizing in Equation (7.6). Therefore, we redefine the cost function of a policy when using the exponential utility function. We let

$$J_\pi(x_0) = \frac{1}{\beta} \log\left(\limsup_{N \to \infty} E\left[e^{\beta(g(x_0,\mu_0(x_0),w_0)+\cdots+g(x_{N-1},\mu_{N-1}(x_{N-1}),w_{N-1}))} \mid x_0\right]\right),$$

as our cost then is equal to what we would have got from minimizing according to the expression in Equation (7.6).

As usual we begin with showing that the mapping H, as defined in Equation (7.10), is monotone.

Proposition 7.2.5 (Exponential utility function H is monotone)**.** †
The mapping H as defined in Equation (7.10) *is monotone. In other words, given two cost functions $J, J' \in \mathcal{R}(X)$ with $J(x) \le J'(x)$ for all $x \in X$ we have that*

$$H(x, u, J) \le H(x, u, J') \text{ for all } x \in X \text{ and } u \in U.$$

Proof. Assume that $J, J' \in \mathcal{R}(X)$ is such that $J \le J'$ pointwise. We then have that

$$\begin{aligned}
H(x,u,J) - H(x,u,J') &= E\left[e^{\beta g(x,u,w)} J(f(x,u,w))\right] - E\left[e^{\beta g(x,u,w)} J'(f(x,u,w))\right] \\
&= E\left[e^{\beta g(x,u,w)} J(f(x,u,w)) - e^{\beta g(x,u,w)} J'(f(x,u,w))\right] \\
&= E\left[e^{\beta g(x,u,w)} (J(f(x,u,w)) - J'(f(x,u,w)))\right] \\
&\le E\left[e^{\beta g(x,u,w)} C\right] \\
&= CE\left[e^{\beta g(x,u,w)}\right] \\
&\le 0,
\end{aligned}$$

where $C = \sup_{x \in X}(J(x) - J'(x)) \le 0$. The above inequality implies that

$$H(x, u, J) \le H(x, u, J'),$$

and thus we see that H indeed is monotone. ∎

We then proceed by presenting a result that show that the operator T_μ, as defined in Equation (7.11), is a contraction for each stationary policy $\mu \in \mathcal{M}$.

Proposition 7.2.6 (Exponential utility function T_μ is contraction)**.** †
The operator T_μ, as defined in Equation (7.11), is a contraction for each stationary policy $\mu \in \mathcal{M}$ whenever the choice of β is such that $E_w\left[e^{\beta g(x,u,w)}\right] < 1$ for all choices of $x \in X$ and $u \in U(x)$. That is, there exist some $\rho \in (0,1)$ such that

$$\|T_\mu J - T_\mu J'\| \leq \rho \|J - J'\|,$$

where $J, J' \in \mathcal{B}(X)$ are two arbitrary functions.

Proof. We assume that $J, J' \in \mathcal{B}(X)$ are two arbitrary functions and that $E_w\left[e^{\beta g(x,u,w)}\right] < 1$ for all $x \in X$ and $u \in U(x)$. Then we let $v(x) = 1$ for all $x \in X$ and define

$$\rho = \sup_{(x,u)\in X\times U} E_w\left[e^{\beta g(x,u,w)}\right] < 1.$$

Then we have that

$$\begin{aligned}
(T_\mu J)(x) - (T_\mu J')(x) &= E\left[e^{\beta g(x,u,w)} J(f(x,u,w))\right] - E\left[e^{\beta g(x,u,w)} J'(f(x,u,w))\right] \\
&= E\left[e^{\beta g(x,u,w)} J(f(x,u,w)) - e^{\beta g(x,u,w)} J'(f(x,u,w))\right] \\
&= E\left[e^{\beta g(x,u,w)} (J(f(x,u,w)) - J'(f(x,u,w)))\right] \\
&= E\left[e^{\beta g(x,u,w)} v(f(x,u,w)) \frac{(J(f(x,u,w)) - J'(f(x,u,w)))}{v(f(x,u,w))}\right] \\
&\leq E\left[e^{\beta g(x,u,w)} v(f(x,u,w)) \|J - J'\|\right] \\
&= E\left[e^{\beta g(x,u,w)}\right] v(x) \|J - J'\| \\
&\leq \rho v(x) \|J - J'\|,
\end{aligned}$$

where we have used that $v(x) = v(y) = 1$ for any pair $x, y \in X$. The calculations above imply that

$$\frac{(T_\mu J)(x) - (T_\mu J')(x)}{v(x)} \leq \rho \|J - J'\|.$$

Observe that by letting J and J' switch places in the calculation above we are able to prove that

$$\frac{(T_\mu J')(x) - (T_\mu J)(x)}{v(x)} \leq \rho \|J - J'\|.$$

Then, by maximizing both sides of the inequalities above we see that

$$\|T_\mu J - T_\mu J'\| \leq \rho \|J - J'\|,$$

as we set out to prove. ■

Note that Proposition 7.2.6 demands very strict assumptions on the value of $e^{\beta g(x,u,w)}$, as we require

$$\sup_{(x,u)\in X\times U} E\left[e^{\beta g(x,u,w)}\right] < 1, \tag{7.12}$$

in order to be able to prove that the operator T_μ is a contraction for each $\mu \in \mathcal{M}$. Nevertheless, it could be that the same result holds with other underlying assumptions. However, it is comforting that the statement holds whenever we are searching for risk-averse policies for a maximization problem, as we always have $e^{\beta g(x,u,w)} < 1$ whenever we have $\beta < 0$ and positive costs, which we assume in general. Nonetheless, the results we have shown in this subsection imply that under our assumptions we have that the fixed point J^* of Bellman's equation equals the cost function of an optimal policy π^* for the risk-sensitive control problem where we utilise an exponential utility function. We have also seen that we are able to find the optimal cost function and a corresponding optimal policy by using the methods covered in Section 4.3. We summarize the findings in the preceding corollary.

Corollary 7.2.7 (Guarantees of risk-sensitive policies)**.** †
We are able to find a unique optimal risk-averse policies for any optimal control problem involving maximization by applying an exponential utility function as described in this section. Conversely, the same applies for risk-seeking policies when we are concerned with a minimization problem.

Error states

This subsection is based on [GW11]. In the previous parts of this section we looked at how minimax and the introduction of an exponential utility function can be used to tackle risk-sensitive model-based control when we define risk as the possibility of extreme costs occurring during the control process. As we have seen minimax handles this problem by always assuming worst-case behaviour and preferring the policy that minimize the worst-case cost. On the other hand, the introduction of the risk-sensitivity parameter β in the exponential utility function case allows the controller to adjust how much the variance of the cost under a given policy should count towards our preference of the policy in addition to the expected cost of the policy in question. However, there has been proposed other ways to define risk in a risk-sensitive control context. We will now take a look at one of these, proposed in the paper [GW11]. In the paper the authors use the notion of error states in order to define risk. We will in this subsection apply the notation we use in connection with SSP problems, that is, we denote a state by i, the successor state as j, a control as u and the probability of being transferred from a state i to the successor state j under the control u is denoted by $p_{ij}(u)$ which is associated with a cost $g(i, u, j)$. Let us now take a closer look at their definition of risk. As usual we let X be the set of states and U the set of controls. For the time being we assume that both X and U are finite, and are therefore able to write $X = \{1, \ldots, n\}$. We then define the set of *error states* to be a subset

$$\Phi \subseteq X.$$

Each error state $\hat{i} \in \Phi$ is a terminal state as these represent e.g. a system failure and thus mark an end of the control process. We denote the remaining terminal states Γ, with $\Gamma \cap \Phi = \emptyset$. The definition of risk for a given state $i_0 \in X$ under a policy π as defined in [GW11] can then be introduced.

Definition 7.2.8 (Risk, [GW11])**.** Let π be a policy and $i_0 \in X$ an arbitrary state. Then the risk is defined to be

$$\rho_\pi(i_0) = P(\exists k \, i_k \in \Phi | \pi), \tag{7.13}$$

where $i_0, i_1, \ldots,$ is the states encountered during the control process when starting in state i_0 and selecting controls according to the policy π.

We see that their notion of risk for a given state i_0 under a policy π is the probability of ending up in any error state $\hat{i} \in \Phi$ when we start our control process at state i_0 and choose our controls following the policy π. Then, as expected, we have from the definition that $\rho_\pi(\hat{i}) = 1$ for each $\hat{i} \in \Phi$. In addition, we see that Definition 7.2.8 also imply that $\rho_\pi(i) = 0$ for each $i \in \Gamma$, as we have defined every state $i \in \Gamma$ to be a terminal state and as $\Phi \cap \Gamma = \emptyset$.

The authors then go on to define an additional cost function, that we denote by $\bar{g}$, and they augment the MDP by adding a new absorbing state η. The controller is transferred, with cost $g = 0$, to the new absorbing state η after reaching one of the states in the set $\Phi \cup \Gamma$. Thus, the states in $\Phi \cup \Gamma$ are no longer terminal states in the MDP with the augmented state space. The additional cost function $\bar{g}$ is given by

$$\bar{g}(i, u, j) = \begin{cases} 1 & i \in \Phi \text{ and } j = \eta, \\ 0 & \text{otherwise.} \end{cases} \tag{7.14}$$

The idea behind the cost function $\bar{g}$ is that a sequence of costs $\bar{g}$ incurred by traversing the MDP under a policy π starting from some initial state i contains the value 1 exactly once if the process reaches the terminal state η from a state $\hat{i} \in \Phi$, while it contains zeros only otherwise. It is therefore possible to express the risk $\rho_\pi(i)$ as the expected value of the costs $\bar{g}$ as stated by the following proposition.

Proposition 7.2.9 (Risk as expected value [GW11])**.** *The identity*

$$\rho_\pi(i_0) = E\left[\sum_{k=0}^{\infty} \bar{g}(i_k, \pi(i_k), w_k)\right] \tag{7.15}$$

holds.

The fact that Equation (7.15) holds is proven in [GW11]. In risk-sensitive control we often want our optimization criterion to in some way include both the expected cost of the policy as well as the risk connected with the policy in question. In [GW11] they consider the maximization counterpart of the minimization problem

$$\min_\pi J_\pi \tag{7.16}$$

subject to

$$\text{for all } i \in X' \colon \rho_\pi(i) \leq \omega, \tag{7.17}$$

where X' is a set that contains the states we are interested in, e.g., a given initial state i_0 or for example $X \setminus (\Phi \cup \Gamma \cup \{\eta\})$. Note that ω specifies what level of risk, as defined by Definition 7.2.8, we allow for the states in X' and therefore we let $\omega \in [0, 1]$. In this context we call a policy *feasible* if it satisfies

Equation (7.17). The authors also introduce another value function $J_{\pi,\xi}$ defined by

$$J_{\pi,\xi}(i) = \xi J_\pi(i) + \rho_\pi(i), \tag{7.18}$$

with $\xi \geq 0$. The policy attaining the minimum for all states $i \in X'$ given a value of ξ in Equation (7.18) is denoted by π^*_ξ. We see that ξ determines the weight of the value of the policy, and that whenever $\xi = 0$, we have $J_{\pi,0}(x) = \rho_\pi(x)$. Thus, a minimization of $J_{\pi,0}$ will imply a minimization of the risk $\rho_\pi(x)$. The parameter ξ is adapted by starting with $\xi = 0$ and then successively increasing the parameter slightly by some value ϵ. By starting with $\xi = 0$ we are able to check whether the constrained problem even is feasible, as the policy attaining the optimal value of $J_{\pi,0}$ is the one with minimal risk. Thus, if $\rho_{\pi^*_0}(i) \leq \omega$ for all $i \in X$ it implies that the problem indeed is feasible. Then, as we are successively increasing ξ we are more strongly weighting the value of the policy, and thus promoting policies that takes more risk given that the ξ weighted decrease in the cost is higher than the increase in the risk. If we let $0, \xi_1, \xi_2, \ldots$ be the weights considered, we need for each policy $\pi^*_{\xi_k}$ to check that the risk constraint Equation (7.17) is satisfied, i.e. we need for each k to check that

$$\rho_{\pi^*_{\xi_k}}(i) \leq \omega \text{ for each } i \in X'.$$

If we at some point reach a value ξ_k that generates a policy $\pi^*_{\xi_k}$ that do violate the constraint we stop the process and say that $\pi^*_{\xi_{k-1}}$ is our optimal policy.

We are now interested in finding a way to calculate the optimal policy for the control problem stated in Equation (7.16) subject to the constraint in Equation (7.17) in a model-based fashion. We would therefore like to find an expression for Bellman's equation for the cost given in Equation (7.18), i.e. would like to find an operator T^ξ_μ that has $J_{\mu,\xi} = \xi J_\mu + \rho_\mu$ as its fixed point. Note that we now write $J_{\mu,\xi}$ instead of $J_{\pi,\xi}$ as we from Proposition 4.2.1 only are guaranteed that there is a fixed point for the operator if the policy is stationary, given that the operator is a contraction. We would therefore like T^*_μ to be a contraction. With that in mind, let us first consider the following operators.

Definition 7.2.10 ($T_\mu, \overline{T}_\mu$)**.** We define the operators T_μ and $\overline{T}_\mu$ by

$$(T_\mu J)(i) = H(i, \mu(i), J), \tag{7.19}$$

and

$$(\overline{T}_\mu J)(i) = \overline{H}(i, \mu(i), J), \tag{7.20}$$

respectively, where $\mu : X \to U$ is some policy function. The mappings $H : X \times U \times \mathcal{R}(X) \to \mathbb{R}$ and $\overline{H} : X \times U \times \mathcal{R}(X) \to \mathbb{R}$ are defined by

$$H(i, u, J) = p_{i\eta}(u) g(i, u, \eta) + \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + J(j)),$$

and

$$\overline{H}(i, u, J) = p_{i\eta}(u) \bar{g}(i, u, \eta) + \sum_{j=1}^{n} p_{ij}(u)(\bar{g}(i, u, j) + J(j)).$$

Note that both the operators T_μ and $\overline{T}_\mu$ have a fixed point, with the fixed points being J_μ and ρ_μ, respectively. In order to see this, observe that the mappings H and $\overline{H}$ are written on the same form as the mapping in Equation (5.5) where we in H and $\overline{H}$ use the cost functions g and $\bar{g}$, respectively. Then it follows from Proposition 5.1.7 that the fixed point of the operator T_μ indeed is the cost function J_μ of the stationary policy μ as g is the cost function of the MDP. For $\overline{T}_\mu$, observe that the solution $\overline{J}_\mu$ of Bellman's equation $\overline{J}_\mu = \overline{T}_\mu \overline{J}_\mu$ is the expected cost of the cost function $\bar{g}$, which is

$$\overline{J}_\mu(i_0) = E\left[\sum_{k=0}^{\infty} \bar{g}(i_k, \pi(i_k), w_k)\right],$$

as we know from Chapter 5 that the fixed point $J_\mu(i)$ of $J = T_\mu J$ satisfies

$$J_\mu(i) = (T_\mu J_\mu)(i) = E\left[\sum_{k=0}^{\infty} g(i_k, \pi(i_k), w_k)\right],$$

and then, as $\overline{T}_\mu$ is on the same form as T_μ, we must have that the solution $\overline{J}_\mu$ of $\overline{J}_\mu = \overline{T}_\mu \overline{J}_\mu$ satisfies

$$\overline{J}_\mu(i) = (\overline{T}_\mu \overline{J}_\mu)(i) = E\left[\sum_{k=0}^{\infty} \bar{g}(i_k, \pi(i_k), w_k)\right].$$

Then it also follows from Proposition 5.1.7 that $\overline{T}_\mu$ has a fixed point $\overline{J}_\mu$, as $\overline{H}$ is on the same form as Equation (5.5). We then have from the argument above, and the fact that Equation (7.15) holds that the fixed point $\overline{J}_\mu$ of $\overline{T}_\mu$ indeed is equal to ρ_μ. As we now have applied Proposition 5.1.7 we need to keep in mind that the proposition only holds true under some assumption. In Chapter 5 we used Assumption 5.1.1 and Assumption 5.1.2 in order to apply the theory from Chapter 4 to prove the results in Section 5.1. However, in the literature Chapter 5 is based on, which is [Ber19], a result similar to our Proposition 5.1.7 is proved using a more direct proof that is not based on the abstract DP theory we considered in Chapter 4 and that only assumes Assumption 5.1.1. In this chapter we will use the assumptions from [Ber19], as these are less restrictive. Thus, in order to guarantee that both T_μ and $\overline{T}_\mu$ have fixed points, with the fixed points being J_μ and ρ_μ respectively, we need Assumption 5.1.1, which we restate here for completion.

Assumption 7.2.11 (Non-zero probability of termination at stage m [Ber19])**.** Assume that there is some number $m \in \mathbb{N}$ such that for all policies $\pi \in \Pi$ we have that

$$\max_{i=1,\dots,n} P(i_m \neq \eta \mid i_0 = i, \pi) < 1.$$

A consequence of Assumption 7.2.11 is that we do not allow MDPs that have policies that do not terminate with probability one, i.e. we do not allow there to be some subset $C \subseteq X \setminus (\Phi \cup \Gamma \cup \{\eta\})$ that satisfy

$$\min_{\pi \in \Pi} P(\exists k\, i_k \in X \setminus C | i_0 \in C, \pi) = 0.$$

The natural approach to find the operator we are looking for is to look at the sum of the two operators we defined in Definition 7.2.10. This motivates the following definition.

Definition 7.2.12 (T_μ^ξ)**.** †
We define the operator T_μ^ξ by

$$(T_\mu^\xi J)(i) = H_\xi(i, \mu(i), J),$$

where $\mu : X \to U$ is some stationary policy. The mapping $H_\xi : X \times U \times \mathcal{R}(X)$ is defined by

$$\begin{aligned} H_\xi(i, u, J) = p_{i\eta}(u)(\xi g(i, u, \eta) + \bar{g}(i, u, \eta)) \\ + \sum_{j=1}^{n} p_{ij}(u)(\xi g(i, u, j) + \bar{g}(i, u, j) + J(j)). \end{aligned}$$

We will now use the remainder of this chapter to show that given a weight ξ, we can write Bellman's equation for the control problem considered in this subsection as $J_{\mu,\xi} = T_\mu^\xi J_\mu^\xi$, and we will prove that the value iteration and policy iteration methods we looked at in Section 4.3 are also applicable for the error state formulation of risk-sensitive model-based control. We start with the following proposition.

Proposition 7.2.13. †
Assume that J_μ and ρ_μ are fixed points for T_μ and $\overline{T}_\mu$, respectively. Then $J_{\mu,\xi}$ is a fixed point for T_μ^ξ, with

$$J_{\mu,\xi} = \xi J_\mu + \rho_\mu.$$

Proof. Assume that J_μ and ρ_μ are fixed points for T_μ and $\overline{T}_\mu$, respectively. Let also $\xi \in \mathbb{R}$ with $\xi > 0$. We then have that

$$J_\mu(i) = p_{i\eta}(\mu(i))g(i, \mu(i), \eta) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i, \mu(i), j) + J_\mu(j)),$$

and

$$\rho_\mu(i) = p_{i\eta}(\mu(i))\bar{g}(i, \mu(i), \eta) + \sum_{j=1}^{n} p_{ij}(\mu(i))(\bar{g}(i, \mu(i), j) + \rho_\mu(j)).$$

Moreover, by letting $J_{\mu,\xi}(i) = \xi J_\mu(i) + \rho_\mu(i)$ for each $i \in X$, we see that

$$\begin{aligned} J_{\mu,\xi}(i) &= \xi J_\mu(i) + \rho_\mu(i) \\ &= \xi(p_{i\eta}(\mu(i))g(i, \mu(i), \eta) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i, \mu(i), j) + J_\mu(j))) \\ &\quad + p_{i\eta}(\mu(i))\bar{g}(i, \mu(i), \eta) + \sum_{j=1}^{n} p_{ij}(\mu(i))(\bar{g}(i, \mu(i), j) + \rho_\mu(j)) \\ &= p_{i\eta}(\mu(i))(\xi g(i, \mu(i), \eta) + \bar{g}(i, \mu(i), \eta)) \\ &\quad + \sum_{j=1}^{n} p_{ij}(\mu(i))(\xi g(i, \mu(i), j) + \xi J_\mu(j) + \bar{g}(i, \mu(i), j) + \rho_\mu(j)) \\ &= p_{i\eta}(\mu(i))(\xi g(i, \mu(i), \eta) + \bar{g}(i, \mu(i), \eta)) \end{aligned}$$

$$
\begin{aligned}
&+\sum_{j=1}^{n} p_{ij}(\mu(i))(\xi g(i,\mu(i),j)+\bar{g}(i,\mu(i),j)+\xi J_\mu(j)+\rho_\mu(j))\\
&= p_{i\eta}(\mu(i))(\xi g(i,\mu(i),\eta)+\bar{g}(i,\mu(i),\eta))\\
&\quad+\sum_{j=1}^{n} p_{ij}(\mu(i))(\xi g(i,\mu(i),j)+\bar{g}(i,\mu(i),j)+J_\mu^\xi(j))\\
&= H_\xi(i,\mu(i),J_\mu^\xi)=(T_\mu^\xi J_\mu^\xi)(i).
\end{aligned}
$$

Thus, we see that T_μ^ξ indeed has a fixed point $J_{\mu,\xi}$, and that it is possible to write the fixed point as $J_{\mu,\xi}(i)=\xi J_\mu(i)+\rho_\mu(i)$. ■

In order to prove that the operator T_μ^ξ has a fixed point it suffices, according to Proposition 7.2.13, to prove that the operators T_μ and $\overline{T}_\mu$ has fixed points. We know from the discussion above that both operators indeed has fixed points. We are thus able to prove the following result.

Proposition 7.2.14 (T_μ^ξ has fixed point)**.** †
Assume that the state space X and control space U is finite, and that Assumption 7.2.11 holds. Then the operator T_μ^ξ defined by

$$
\begin{aligned}
(T_\mu^\xi J)(i) &= p_{i\eta}(\mu(i))(\xi g(i,\mu(i),\eta)+\bar{g}(i,\mu(i),\eta))\\
&\quad+\sum_{j=1}^{n} p_{ij}(\mu(i))(\xi g(i,\mu(i),j)+\bar{g}(i,\mu(i),j)+J(j)),
\end{aligned}
$$

has a fixed point $J_{\mu,\xi}$ that we are able to decompose into

$$J_{\mu,\xi}=\xi J_\mu+\rho_\mu$$

where J_μ is the unique fixed point of T_μ and ρ_μ is the unique fixed point of $\overline{T}_\mu$. In addition, the value $\rho_\mu(i)$ denotes the probability of entering an error state $j\in\Phi$ when choosing controls according to the stationary policy μ and stating in state i_0, i.e. $\rho_\mu(i_0)=P(\exists k\, i_k\in\Phi\mid\mu)$.

Proof. We start by proving that both T_μ and $\overline{T}_\mu$ have fixed points, and that these fixed points are J_μ and ρ_μ respectively. As the mappings H and $\overline{H}$, as defined in Definition 7.2.10, are on the same form as the mapping in Equation (5.5). As we assume that Assumption 7.2.11 hold, we are able to use Proposition 4.2.3 from [Ber19], which provide the same result as Proposition 5.1.7 but without the constraints of Assumption 5.1.2. That is, Proposition 4.2.3 from [Ber19] grantees that the operators T_μ and $\overline{T}_\mu$ have fixed points, with the fixed point $J_\mu$of T_μ being the cost function of the policy as g is the cost function determining the cost associated with traversing the MDP. From the discussion above we have that the fixed point $\overline{J}_\mu$ of $\overline{T}_\mu$ is given by

$$\overline{J}_\mu(x_0)=E\left[\sum_{k=0}^{\infty}\bar{g}(x_k,\mu(x_k),w_k)\right],$$

which by Proposition 7.2.9 is equal to $\rho_\mu(i_0)=P(\exists k\, i_k\in\Phi\mid\mu)$. As the operators T_μ and $\overline{T}_\mu$ have J_μ and ρ_μ, respectively, as fixed point we know from Proposition 7.2.13 that the operator T_μ^ξ has a fixed point $J_{\mu,\xi}$ that we are able to write as $J_{\mu,\xi}=J_\mu+\rho_\mu$, just as we set out to prove. ■

We have now found a formulation of Bellman's equation for the risk sensitive control problem considered in this subsection. We would also like to show that we are able to utilize the methods presented in Section 4.3 in order to find the unique solution of the set of equations generated by the Bellman equation

$$J_{\mu,\xi} = T^{\xi}_{\mu} J^{\xi}_{\mu}.$$

We know from Propositions 4.3.3 and 4.3.5 that it is sufficient to show that the operator T^{ξ}_{μ} is a contraction for each stationary policy μ and that the mapping H_{ξ} satisfies the monotonicity property for all $\xi > 0$. In order to guarantee convergence we also need the set of stationary policies $\mathcal{M}$ to be finite, but as both the state space X and control space U are assumed to be finite we see that the set of stationary policies $\mathcal{M}$ is finite as well. As Proposition 4.2.5 in [Ber19] show that an operator T'_{μ} on the form

$$(T'_{\mu}J)(i) = p_{it}(\mu(i))g(i,\mu(i),t) + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + J(j))$$

is a contraction for each $\mu \in \mathcal{M}$ under Assumption 7.2.11 with finite state space, and as we are able to write T^{ξ}_{μ} as

$$\begin{aligned}(T^{\xi}_{\mu}J)(i) &= p_{i\eta}(\mu(i))(g(i,\mu(i),\eta) + \bar{g}(i,\mu(i),\eta)) \\ &\quad + \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + \bar{g}(i,\mu(i),j) + J(j)) \\ &= p_{i\eta}(\mu(i))h(i,\mu(i),\eta) + \sum_{j=1}^{n} p_{ij}(\mu(i))(h(i,\mu(i),j) + J(j)),\end{aligned}$$

where $h(i,u,j) = g(i,u,j) + \bar{g}(i,u,j)$, we see that T^{ξ}_{μ} also is a contraction as it can be written on the same form as T'_{μ}. Therefore it only remains to show that H_{ξ} is monotone for each $\xi > 0$.

Proposition 7.2.15 (H_{ξ} is monotone)**.** †
For each $\xi \in \mathbb{R}$ with $\xi > 0$ we have that the mapping $H_{\xi} : X \times U \times \mathcal{R}(X)$ defined by

$$H_{\xi}(x,u,J) = \sum_{j=1}^{n} p_{ij}(u)(\xi g(i,u,j) + \bar{g}(i,u,j) + J(j)),$$

satisfies the monotonicity property. That is, for some $J, J' \in \mathcal{R}(X)$ with $J \leq J'$ pointwise we have

$$H_{\xi}(x,u,J) \leq H_{\xi}(x,u,J') \text{ for all } x \in X \text{ and } u \in U(x).$$

Proof. To prove the monotonicity property of H_{ξ} it is sufficient to prove that

the difference between $H_\xi(x, u, J')$ and $H_\xi(x, u, J)$ is non-negative. We see that

$$\begin{aligned} H_\xi(x,u,J') - H_\xi(x,u,J) &= \sum_{j=1}^{n} p_{ij}(u)(\xi g(i,u,j) + \bar{g}(i,u,j) + J'(j)) \\ &\quad - \sum_{j=1}^{n} p_{ij}(u)(\xi g(i,u,j) + \bar{g}(i,u,j) + J(j)) \\ &= \sum_{j=1}^{n} p_{ij}(u)(\xi g(i,u,j) + \bar{g}(i,u,j) + J'(j) \\ &\qquad - \xi g(i,u,j) - \bar{g}(i,u,j) - J(j)) \\ &= \sum_{j=1}^{n} p_{ij}(u)(J'(j) - J(j)) \\ &\geq \sum_{j=1}^{n} p_{ij}(u) C \\ &= C \sum_{j=1}^{n} p_{ij}(u) \\ &\geq 0, \end{aligned}$$

where $C = \min_{j \in X}(J'(j) - J(j)) \geq 0$, since we assume that $J(j) \leq J'(j)$ for all $j \in X$. Thus we see that the difference between $H_\xi(x, u, J')$ and $H_\xi(x, u, J)$ in fact is non-negative, proving that H_ξ is monotone. ■

We have in this subsection showed that under the assumption of having a finite state space X and a finite control space U, as well as assuming that Assumption 7.2.11 holds, we are able to find the policy that tries to attain the minimum in

$$\min_{\pi \in \Pi} J_\pi, \tag{7.21}$$

subject to

$$\text{for all } i \in X' : \rho_\pi(i) \leq \omega, \tag{7.22}$$

for some subset $X' \subset X \setminus \Phi$ and a risk threshold $0 \leq \omega \leq 1$. We also have from Proposition 4.2.2 that we are able to restrict the policy space we are minimizing over from the set of all policies Π to the set of all stationary policies $\mathcal{M}$, as we have proved above that the operator T_ξ is a contraction and that the mapping H_ξ is monotone for each value of $\xi \in \mathbb{R}$ with $\xi > 0$. That is, we are able to write Equation (7.21) as

$$\min_{\mu \in \mathcal{M}} J_\mu.$$

As written in the beginning of the subsection, the optimization process should start with finding the policy that minimizes the risk, as this policy would need to satisfy Equation (7.22) for the problem to be feasible. We can then set the parameter ξ to some value $\epsilon > 0$, giving $\xi_0 = 0$ and $\xi_1 = \epsilon$. The thought behind iteratively increasing the value ξ is to more strongly punish high-cost policies and in that way favour more risk-seeking low-cost policies. Then, after finding the policy $\pi^*_{\xi_1}$ that solves

$$\min_{\mu \in \mathcal{M}} J_{\mu,\xi_1} = \min_{\mu \in \mathcal{M}} (\xi_1 J_\mu + \rho_\mu), \tag{7.23}$$

we need to check whether it satisfies the constraint in Equation (7.22). If that is the case, we again increase the parameter ξ by ϵ, giving $\xi_2 = 2\epsilon$, and repeat the process until we find some value ξ_k such that $\pi^*_{\xi_k}$ does not satisfy Equation (7.22). We then stop the process and have $\pi^*_{\xi_{k-1}}$ as our optimal policy. Furthermore, we also proved in this subsection that we are able to find the policy that is optimal for each value of ξ by using the methods from Section 4.3. That is, we are able to use value iteration and policy iteration in order to find the policy π^*_ξ that satisfies

$$J_{\pi^*_\xi} = J^*_\xi = T^\xi J^*_\xi,$$

where J^*_ξ is the optimal cost function, i.e. the cost function attaining the minimum in Equation (7.23), for the given value of ξ, and where

$$(T^\xi J)(i) = \min_{u \in U(i)} H_\xi(i, u, J).$$

In addition, we showed with Proposition 7.2.14 that the cost function satisfying Equation (7.23) for each value of ξ is possible to write as $\xi J_\pi + \rho_\pi$, which as we have seen in this subsection has a nice interpretation from Proposition 7.2.14. Let us now take a look at how we can utilize this model-based method on the St. Petersburg problem

Example 7.2.16 (Error states, continuation of Example 7.1.1)**.** †
We will now use the error state formulation of risk in order to find another take on what a sensible risk-sensitive policy is for the St. Petersburg paradox that we introduced in Example 7.1.1. In order to use the error state formulation of risk we need to introduce a new state space. We let $X = \{x_0, t, e, \eta\}$ and $X' = \{x_0\}$, where x_0 is the initial state, t is a non-error state, e is an error state, while η is an artificial terminal state introduced for technical reasons as described earlier in this subsection. The controller is automaticity transferred to η at no cost whenever she enters the state t or the state e. However, if the agent is transferred to η from the error state e, it attains the 'risk' cost $\bar{g}(e, \cdot, \eta) = 1$. As in Example 7.2.1 we have that $U(x_0) = \{u_p, u_n\}$, where choosing u_p means that we pay some amount $k \in [0, \infty)$ in order to take the bet, while u_n means that we do not play. We let

$$f_0(x_0, u_n, \cdot) = t.$$

For the control u_p we let

$$f_0(x_0, u_p, w) = t$$

whenever

$$w \in \{w \in W(x, u_p) \mid g(x_0, u_p, w) - k \geq 0\}.$$

However, if

$$w \in \{w \in W(x, u_p) \mid g(x_0, u_p, w) - k < 0\}, \tag{7.24}$$

we have that

$$f_0(x_0, u_p, w) = e.$$

In other words, we transfer to the error state e if we end up with a price that is smaller than the bet we put in. Let μ_p be the policy with $\mu_p(x_0) = u_p$ and μ_n the policy where $\mu_n(x_0) = u_n$. Then, as we know that

$$P\left(\{w \in W(x, u_p) \mid g(x, u_p, w) = 2^n - k\}\right) = \frac{1}{2^n} \text{ for each } n \in \mathbb{N} \setminus \{0\},$$

we see that

$$\begin{aligned}\rho_{\mu_p}(x_0) &= P(\exists i\, x_i \in \{e\} | u_0 = u_p) \\ &= P(\{w \in W(x_0, u_p) \mid g(x_0, u_p, w) < k\}) \\ &= \sum_{n=1}^{\lceil \log_2(k) \rceil - 1} \frac{1}{2^n},\end{aligned}$$

given some value $k \in [0, \infty)$ for the initial stake, and where the transition from the first to the second line comes from the fact that the random values w that lies in the set defined in Equation (7.24) are the same values that we need in order to have $x_1 = e$. In order to explain why $\lceil \log_2(k) \rceil - 1$ is the limit of the sum, consider some bet $k \in \mathbb{R}$ with $k > 0$. It is then clear that there exist a unique $m \in \mathbb{N}$ such that $2^{m-1} < k \leq 2^m$. We would then need the coin to show heads on the mth throw, or later, in order to not be at a loss. What is then the probability of that *not* happening, i.e. what is the probability of heads showing on one of the $m - 1$ first throws? We have that

$$P(\text{heads showing on first } m-1 \text{ throws}) = \sum_{n=1}^{m-1} \frac{1}{2^n}.$$

Thus, given some stake k we see that $m = \lceil \log_2(k) \rceil$, and as we need to sum up to $m - 1$ we conclude that the upper limit of the sum must be $\lceil \log_2(k) \rceil - 1$.

We have now seen that the policy μ_p is feasible, in the sense that $\rho_{\mu_p}(x_0) \leq \omega$, whenever

$$\sum_{n=1}^{\lceil \log_2(k) \rceil - 1} \frac{1}{2^n} \leq \omega.$$

Clearly, we also have that $\rho_{\mu_n}(x_0) = 0$, as $f_0(x_0, u_n, \cdot) = t$, which imply that μ_n is feasible for all values of k and all choices of $\omega \in [0, 1]$. Note that we only need to consider the case where the weight $\xi = 0$, as $J_{\mu_p} = \infty$ and $J_{\mu_n} = 0$, so for any $\xi > 0$ we would have $J_{\mu_p,\xi} > J_{\mu_n,\xi}$. We will therefore have μ_p as our optimal policy whenever the value of k and the controllers choice of ω is such that the policy μ_p is feasible, otherwise our optimal policy will be μ_n. Therefore it suffices to check whether the policy μ_p is feasible in order to find the optimal policy for any value k and risk level ω.

Let us now set $\omega = \frac{1}{2}$ and find out for which values of k we would be willing to bet and play the game. We see that for $k = 4$ we have

$$\begin{aligned}\rho_{\mu_p}(x_0) &= P(\{w \in W(x_0, u_p) \mid g(x_0, u_p, w) < 4\}) \\ &= \sum_{n=1}^{\lceil \log_2(4) \rceil - 1} \frac{1}{2^n} = \sum_{n=1}^{2-1} \frac{1}{2^n} \\ &= \sum_{n=1}^{1} \frac{1}{2^n} = \frac{1}{2},\end{aligned}$$

while for $k > 4$ we have that

$$\rho_{\mu_p}(x_0) = P(\{w \in W(x_0, u_p) \mid g(x_0, u_p, w) < 4 + \epsilon\})$$

$$= \sum_{n=1}^{\lceil \log_2(4+\epsilon) \rceil - 1} \frac{1}{2^n} \geq \sum_{n=1}^{3-1} \frac{1}{2^n}$$
$$= \sum_{n=1}^{2} \frac{1}{2^n} = \frac{3}{4} > \frac{1}{2},$$

where $\epsilon > 0$. Thus, when our accepted level of risk is $\omega = \frac{1}{2}$ we have that μ_p is feasible whenever $k \leq 4$. That gives in this instance

$$\mu^*(x_0) = \begin{cases} u_p & \text{if } k \leq 4, \\ u_n & \text{otherwise.} \end{cases}$$

Observe that by setting the level of risk equal to $\omega = \sum_{i=1}^{n} \frac{1}{2^n}$ for some $n \in \mathbb{N} \setminus \{0\}$ we will be willing to bet up to 2^{n+1} units.

Another point of interest is to look at how risk-averse/risk-seeking we would need to be in order to accept a stake of k units, i.e. given some value k, what would our level ω need to be in order for us to be willing to participate in the game. By the observation above we see that for some stake k we would be willing to take the bet whenever $\omega \geq \sum_{i=1}^{\lceil \log_2(k) \rceil - 1} \frac{1}{2^i}$. Let us for example assume that $k = 5$. We then have that

$$\omega \geq \sum_{i=1}^{\lceil \log_2(5) \rceil - 1} \frac{1}{2^i} = \sum_{i=1}^{3-1} \frac{1}{2^i} = \sum_{i=1}^{2} \frac{1}{2^i} = \frac{3}{4}.$$

We see that even with a stake of $k = 5$ we would need to at least be willing to sustain a loss in 3 out of 4 games in order to be willing to play.

Before we end this subsection a word of caution is needed. Even though we have showed that for each value of ξ there exists a fixed point J_ξ^* of the equation

$$J_\xi^* = T^\xi J_\xi^*,$$

we need to keep in mind that the policy π_ξ^* satisfying $J_{\pi_\xi^*} = J_\xi^*$ only achieves the minimum for the cost function

$$\min_{\pi \in \Pi} \xi J_\pi(i) + \rho_\pi(x).$$

This means that the final policy π_ξ^* generated by the algorithm described in this subsection may be suboptimal, as illustrated by the following example.

Example 7.2.17 (Suboptimal policy error states method). †
Assume that we are considering some optimal control problem where we in a state $x \in X$ has three different choices for the control, i.e. $U(x) = \{u_1, u_2, u_3\}$. Let us also assume that we have three different policies, μ_1, μ_2 and μ_3 with $\mu_i(x) = u_i$. The controllers level of risk-sensitivity yields $\omega = 0.2$. Assume then that the cost functions of the policies satisfy

$$J_{\mu_1}(x) = 2, J_{\mu_2}(x) = 1, J_{\mu_3}(x) = 0,$$

and that the risk of the policies are

$$\rho_{\mu_1}(x) = 0, \rho_{\mu_2}(x) = 0.2, \rho_{\mu_3}(x) = 0.21.$$

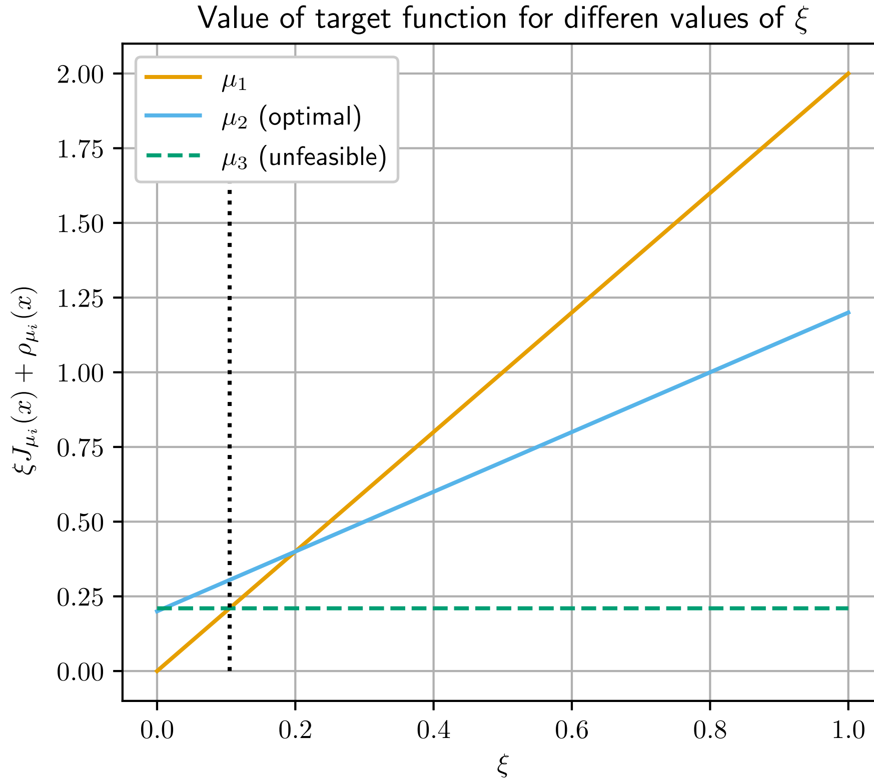

Figure 7.3: Value of target function for the policies considerd in Example 7.2.17. The black line shows where the target function for policy μ_1 and μ_3 intersects. Code to produce figure can be found in Listing A.11.

Note that the policy μ_3 is unfeasible as $\rho_{\mu_3}(x) > \omega$. It is clear that the optimal policy for this problem according to the problem formulation given in Equations (7.16) and (7.17) is μ_2. However, when we gradually increase ξ from 0 we see that when e.g. $\xi = 0.11$ we get

$$J_{\mu_3,0.11} = 0.11 \cdot J_{\mu_3}(x) + \rho_{\mu_3} = 0.21,$$

$$J_{\mu_2,0.11} = 0.11 \cdot J_{\mu_2}(x) + \rho_{\mu_2} = 0.31,$$

and

$$J_{\mu_1,0.11} = 0.11 \cdot J_{\mu_1}(x) + \rho_{\mu_1} = 0.22.$$

Thus $\mu^*_{0.11} = \mu_3$, and as $\rho_{\mu_3}(x) = 0.21 > \omega$, the method terminates and return the policy that was optimal for the previous choice of ξ, which we from Figure 7.3 can see that is μ_1. We can therefore conclude that the error states method presented in this chapter does not always produce the optimal policy.

In the next subsection we will apply the three model-based risk-sensitive approaches we have looked at in this section in order to find risk-sensitive policies for the SSP problem we considered in Examples 6.3.1 and 5.1.9.

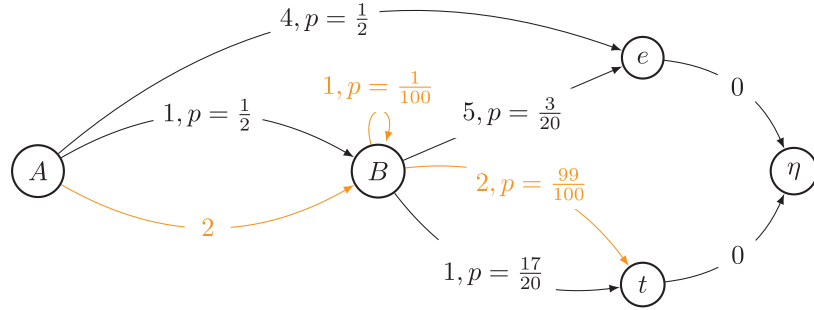

Figure 7.4: An augmented version of the SSP in Figure 5.1. Same coloured paths originating at the same node depict the possible transitions from the given node under a specific control. The black lines depict the transitions for control u_1, while orange shows the same for control u_2.

Numerical example

We will in this subsection take a look at how the risk-sensitive methods covered in this section performs on the SSP problem we looked at in Examples 6.3.1 and 5.1.9 in order to see what kind of behaviour they promote.

Example 7.2.18 (Continuation of Examples 6.3.1 and 5.1.9)**.** †
We are now going to see which policies are favoured by the risk-sensitive model-based methods we have looked at in this section. We implement the value iteration and policy iteration algorithms using the optimization criteria we have looked at in this chapter and run them on the SSP problem we considered earlier in Examples 6.3.1 and 5.1.9. The code used for this example resides in several files. The structure of the SSP, with among other things, a state class that have methods that, given some cost function, finds the best action and optimal cost from the state under the different optimization criteria can be found in Listing A.1. The code to run the minimax experiments can be found in Listing A.6, the code for the experiments using the exponential utility function are in Listing A.7, while to see the code for testing the error state implementation consult Listing A.8. For the error state notion we had to define an augmented SSP in order to introduce an error state. The new SSP is depicted in Figure 7.4, and the json file describing it can be found in Listing A.5. Note that we define the error state as being the situation where the controller applies control 1 in state A and ends up incurring a transition cost of 4 (the value 4 was chosen as it is the sum of the original cost of 3 and the best-case transition cost between state B and t, which is 1), or when the agent choose action 1 in state B and receives the transition cost of 5.

For the exponential utility function method we ran the algorithm using two different values for β, namely $\beta = -0.75$ and $\beta = -1$, as we by trial-and-error found that these parameter choices resulted in two different policies. We also tried to run the algorithm for positive values of β, but the method did then not produce a sensible result. This may be caused by the fact that the assumption of Proposition 7.2.6 does not holds for this problem whenever $\beta > 0$. For the

Method and parameter values	$J^*(A)$	$J^*(B)$	$J^*(C)$	$\rho_{\mu_\xi^*}(A)$	$\rho_{\mu_\xi^*}(B)$
Minimax, $\alpha = 0.9$	11.00	10.00	9.00		
Exp. utility, $\beta = -0.075$	-0.17	5.30	-2.22		
Exp. utility, $\beta = -1$	0.34	5.24	-1.66		
Error states, $\omega = 0.1, \xi = 0.25$	4.01	2.01		0	0
Error states, $\omega = 0.2, \xi = 1$	3.60	1.60		0.15	0.15

Table 7.1: Optimal value under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4. The $J^*(s)$ values for error state is the actual expected cost function values of the policy found to be optimal by the method.

Method and parameter values	$\mu^*(A)$	$\mu^*(B)$	$\mu^*(C)$
Minimax, $\alpha = 0.9$	2	2	1
Exp. utility, $\beta = -0.075$	1	1	1
Exp. utility, $\beta = -1$	2	1	1
Error states, $\omega = 0.1, \xi = 0.25$	2	2	
Error states, $\omega = 0.2, \xi = 1$	2	1	

Table 7.2: Optimal policy under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4.

Method and parameter values	$J_{\mu^*}(A)$	$J_{\mu^*}(B)$	$J_{\mu^*}(C)$
Minimax, $\alpha = 0.9$	3.81	2.01	1.81
Exp. utility, $\beta = -0.075$	3.88	1.88	1.88
Exp. utility, $\beta = -1$	3.88	1.88	1.88
Error states, $\omega = 0.1, \xi = 0.25$	4.01	2.01	
Error states, $\omega = 0.2, \xi = 1$	3.60	1.60	

Table 7.3: Expected discounted cost for the optimal policies found under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4.

error state method we used two different ω thresholds for allowed level of risk, specifically $\omega = 0.1$ and $\omega = 0.2$. We also increase the ξ parameter by 0.25 for each round.

The result from all three methods with the different parameter values can be found in table Table 7.1. We quickly see that the values shown in Table 7.1 does not tell us much as the transformations used in the different methods makes the interpretation harder, except for the error states method, as we report the actual cost function and the risk function for that method. Let us therefore include two other tables. In Table 7.2 we present the optimal policy found by policy iteration for the different optimization criteria, while we in Table 7.3

present the expected costs for the policies found to be optimal by the different methods, i.e. the cost function of the policies shown in Table 7.2.

We can see from Table 7.2 that only two of the combinations of risk-sensitive methods and parameter choices resulted in the optimal policy μ^* having $\mu^*(B) = 2$, which we from Figures 5.1 and 7.4 can see that is the 'safer' option if we would like to avoid high-cost transitions at the price of a lower expected value, or if we would like to avoid the error states at all costs in the error state formulation of the problem. It is no surprise that minimax promotes an optimal policy that at all states chooses action 2, as these have the lowest worst-case cost. It is also not surprising that the error state formulation with $\omega = 0.1$ yields the same result, as applying control 1 in state B gives a probability $p = \frac{3}{20} = 0.15$ of transitioning to an error-state, and as $\omega = 0.1$, we only allow a probability of $p \leq 0.1$ to end up in an error-state from any one state. It is therefore expected that when we increase the parameter value of ω to 0.2 the preferred policy changes, as we then allow our policy to apply the more risky action 1 in state B, given that the decrease in the ξ-weighted expected cost is high enough. We see from Table 7.2 that indeed the ξ-weighted increase is large enough for the method to change its preferred policy, as $\mu^*(B) = 1$ when using the error-state method with $\omega = 0.2$. Another thing that was quite interesting is how the exponential utility method with a value of β close to 0 yields an optimal policy that has $\mu^*(A) = 1$, while we by decreasing the parameter value to $\beta = -1$ have that the optimal policy becomes the one that has $\mu^*(A) = 2$. This is surprising as the exponential utility function method should promote policies that has a greater variability when the value of β decreases, but as we with $\mu^*(A) = 1$ have a 50/50 chance of either incurring a cost of 1 or 3, while the other possible control always gives a cost of 2, we see that the variability actually decreases as the negative value of β increases in this case. Interestingly, by experimenting with further decreasing the value of β we noticed that the method consistently returned the same optimal policy as we received with a parameter value of $\beta = -1$.

When looking at the results in Table 7.3, it is interesting to see that both risk-sensitive policies generated by policy iteration with the exponential utility function optimization criterion have the same expected cost as the policy that is optimal in the risk-neutral sense, which we found in Example 5.1.9. In fact, as we briefly mentioned in Example 5.1.9, both policies found by the exponential utility function method are considered optimal in the risk-neutral sense, as is evident from the fact that the cost-functions of the policies are identical. For minimax we see that it yields the expected policy, which is the policy that in all states picks the control that has the lowest maximum cost. In the case of the error-state formulation we see that when increasing ω from 0.1 to 0.2 we have that the optimal policy satisfies $\mu^*(B) = 1$, as the associated decrease in cost with switching from applying control 2 in the same state outweighs the increase in risk.

7.3 Model-free risk-sensitive control

In the previous section we looked at model-based methods for dealing with risk-sensitive control problems, both by controlling the variation of performance of our policy and by introducing error states. We will now take a look at

model-free methods dealing with the same problem. We start with looking at a method that is a model-free version of Minimax that we introduced in Section 7.2. It was introduced in [Heg94] and is named $\hat{Q}$-learning, which essentially is a pessimistic version of Q-learning, and hence we will referee to the method as pessimistic Q-learning.

Pessimistic Q-learning

This subsection is based on [Heg94]. The pessimistic Q-learning method does, as minimax, assume that the worst-case outcome is the one that we are going to observe, and are thus trying to find a policy that optimize over the worst-case outcome. Let now

$$\hat{g}(i,u,j) = \sup_{w \in W(i,u)} \{g(i,u,w) | f(i,u,w) = j\}.$$

Then,

$$J_k(i) = \min_{u \in U(i)} \left(\max_{j \in \{x \in X | p_{ix} > 0\}} (\hat{g}(i,u,j) + \alpha J_{k-1}(j)) \right), \tag{7.25}$$

is value iteration corresponding to using the mapping in Equation (7.3) when we write the supremum with regards to the possible successor states instead of the events $w \in W(i,u)$. The Q-function we then try to estimate with pessimistic Q-learning is

$$\hat{Q}^*(i,u) = \max_{j \in \{x \in X | p_{ix} > 0\}} (\hat{g}(i,u,j) + \alpha J^*(j)), \tag{7.26}$$

where J^* is the cost function found as the limit $\lim_{k \to \infty} J_k$, where J_k is as defined in Equation (7.25). The fact that $J^* = \lim_{k \to \infty} J_k$ when $J_0 \in \mathbb{R}^{|X|}$, where $|X|$ is the number of states in the state space X, is taken from Theorem 2 in [Heg94]. The pessimistic Q-learning tries to approximate $\hat{Q}$ by use of the following update formula:

$$Q(i,u) = \max \left\{ Q(i,u), r + \alpha \min_{u' \in U(j)} Q(j,u') \right\}, \tag{7.27}$$

where the initial Q-values needs to be less than or equal to the value of Q^*, i.e. $Q_0(i,u) \leq Q^*(i,u)$. Thus with no prior knowledge of Q^* we could set each initial value to zero, that is $Q_0(i,u) = 0$ for all $u \in U(i), i \in X$. The author claims in [Heg94] that the values $Q(i,u)$ found by running $\hat{Q}$-learning (Equation (7.27)) converges to the optimal values $\hat{Q}^*(i,u)$ (Equation (7.26)) with probability one as the number of iterations goes to infinity, given that each pair (i,u) for all $i \in X$ and $u \in U(i)$ is visited infinity often.

Weighting TD-rewards

Even though introducing an exponential utility function is a good way to introduce a risk sensitive target function, it has one major drawback when it comes to trying to find reinforcement learning methods approximating the cost given by Equation (7.6). Since we are applying the utility function to the final cost, or gain, we are actually not able to make use of this knowledge during training of our algorithm. To tackle this problem we can take a look at

the proposed solution from [MN02]. They propose a new framework for risk-sensitive control where they instead of applying a transformation on the final cost itself, they transform the so called temporal differences during learning. By applying the transformation during learning we are able to train the algorithms into being risk-sensitive using algorithms like Q-learning and SARSA. The transformation introduced in [MN02] is

$$\chi^\kappa(x) = \begin{cases} (1-\kappa)x \text{ if } x > 0, \\ (1+\kappa)x \text{ otherwise,} \end{cases}$$

where $\kappa \in (-1, 1)$ is a parameter that decides on the risk-sensitivity of the algorithm. The risk-sensitive value function J_κ^μ is then defined implicitly as the solution of the equations

$$0 = \sum_{j \in X} p_{ij}(\mu(i))\chi^\kappa(g(i, \mu(i), j) + \alpha J_\kappa^\mu(j) - J_\kappa^\mu(i)), \tag{7.28}$$

where μ is some fixed stationary policy. The following theorem ensures that the risk-sensitive value function is well defined. For a proof of the result consult [MN02].

Theorem 7.3.1 (Well-defined value function [MN02]). *For each $\kappa \in (-1, 1)$ there is a unique solution J_κ^μ of the defining system of equations* (7.28)*. Hence, the value function in the risk-sensitive sense is well defined. The following properties apply for each $i \in S$*

$$J_0^\mu(i_0) = E\left[\sum_{t=0}^{\infty} \alpha^t g(i_t, \mu(i_t), i_{t+1})\right], \tag{7.29}$$

$$\lim_{\kappa \to 1} J_\kappa^\mu(i_0) = \inf_{\substack{i_0, i_1, \ldots \\ p(i_0, i_1, \ldots) > 0}} \left(\sum_{t=0}^{\infty} \alpha^t g(i_t, \mu(i_t), i_{t+1})\right), \tag{7.30}$$

$$\lim_{\kappa \to -1} J_\kappa^\mu(i_0) = \sup_{\substack{i_0, i_1, \ldots \\ p(i_0, i_1, \ldots) > 0}} \left(\sum_{t=0}^{\infty} \alpha^t g(i_t, \mu(i_t), i_{t+1})\right). \tag{7.31}$$

Note that when $\kappa = 0$, we have that $\chi^0(x) = x$, and thus (7.28) reduces to the normal risk-neutral case, which is what we see in (7.29). On the other hand, when $\kappa > 0$ we overweight negative temporal differences, and the policy thus becomes more risk-seeking, given that we are trying to minimize costs. However, if we try to maximize rewards, having $\kappa > 0$ makes the resulting policy more risk-averse. Similarly, if $\kappa < 0$ we overweight positive temporal differences, and are thus less optimistic and more risk-averse if we try to minimize costs, and again the interpretation is opposite if we are maximizing rewards. We see from Equations (7.30) and (7.31) that in the limits $\kappa \to 1$ and $\kappa \to -1$ we approach a very optimistic value function where we always assume that the least costly outcome is the one that is going to happen and the minimax value function, respectively. The natural way to then define a optimal policy π^* in the risk-sensitive sense, given that we are considering a maximization problem, is that the policy π^* satisfies the equation

$$J_\kappa^{\pi^*}(i) \geq J_\kappa^\pi \text{ for all } \pi \in \Pi, i \in X. \tag{7.32}$$

Observe that [MN02] is concerned with the maximization formulation of the optimization problems we are looking at. In order to present the results from the paper as they are stated, we therefore assume for the rest of this subsection that we are looking at a maximization problem, but note that the results still hold for the minimization problem by exchanging max with min and arg max with arg min at the appropriate places.

In [MN02] the authors prove that there exists such an optimal policy that is stationary through the following theorem (Theorem 4 in [MN02]).

Theorem 7.3.2 (Optimal policy [MN02])**.** *For each $\kappa \in (-1, 1)$ there is a unique optimal value function*

$$J^*_\kappa = \max_{\pi \in \Pi} J^\pi_\kappa,$$

which satisfies the optimality equation

$$0 = \max_{u \in U(i)} \sum_{j \in X} p_{ij}(u) \chi^\kappa (g(i,u,j) + \alpha J^*_\kappa(j) - J^*_\kappa(i)).$$

Furthermore, a policy π^* is optimal if and only if

$$\pi^*(i) = \arg\max_{u \in U(i)} \sum_{j \in X} p_{ij}(u) \chi^\kappa (g(i,u,j) + \alpha J^*_\kappa(i) - J^*_\kappa(i)).$$

We then, as in [MN02] introduce the following system of equations, for which the solution is the risk-sensitive Q-functions $Q^\pi_\kappa(i,u)$.

$$0 = \sum_{j \in X} p_{ij}(u) \chi^k \left(g(i,u,j) + \alpha J^\pi_\kappa(j) - Q^*_k(i,u)\right) \text{ for all } i \in X, u \in U(i). \tag{7.33}$$

The following theorem (which is a combination of Theorem 5 and Theorem 6 from [MN02]) makes sure that the above defined Q-function is well defined, and also introduce the optimality equations for the Q-function.

Theorem 7.3.3 (Q-function [MN02])**.** *For each $\kappa \in (-1, 1)$ there is a unique solution Q^π_κ of the defining system of equations* (7.33)*. Thus, the Q-function Q^π_κ is well defined.*

The optimal Q-function Q^*_κ is the unique solution of the optimality equation

$$0 = \sum_{j \in X} p_{ij}(u) \chi^\kappa \left(g(i,u,j) + \alpha \max_{v \in U(j)} Q^*_\kappa(j,v) - Q^*_\kappa(i,u) \right) \textit{ for all } i \in X, u \in U(i)$$

Furthermore, a policy π^* is optimal if and only if

$$\pi^*(i) = \arg\max_{u \in U(i)} Q^*_\kappa(i,u).$$

The authors then go on to introduce the following risk-sensitive Q-learning algorithm:

$$\begin{aligned} \hat{Q}_t(i,u) = \hat{Q}_{t-1}(i,u) + \gamma_{t-1}(i,u) \chi^\kappa (g(i_{t-1}, u_{t-1}, i_t) \quad (7.34)\\ + \alpha \max_{v \in U(i_t)} \hat{Q}_{t-1}(i_t, v) - \hat{Q}_{t-1}(i_{t-1}, u_{t-1})), \end{aligned}$$

$$\gamma_{t-1}(i_{t-1}, u_{t-1}) = \gamma_{t-1} > 0, \tag{7.35}$$

$$\gamma_{t-1}(i,u) = 0 \text{ if } (i,u) \neq (i_{t-1}, u_{t-1}). \tag{7.36}$$

The following theorem from [MN02] proves convergence of the algorithm to the optimal Q-function Q^*_κ under given conditions.

Theorem 7.3.4 (Convergence of Q-learning algorithm [MN02])**.** *Let $\kappa \in (-1,1)$. Consider the risk-sensitive Q-learning algorithm, as described by Equations* (7.34) *to* (7.36)*. If the learning rates $\gamma_i(i,u)$ are nonnegative and satisfy*

$$\sum_{t=0}^{\infty} \gamma_i(i,u) = \infty, \sum_{t=0}^{\infty} (\gamma_i(i,u))^2 < \infty \text{ for all } u \in U(i), i \in X,$$

then $\hat{Q}_t(i,u)$ converges to Q^*_κ for all i and u with probability 1.

We thus see that under certain conditions our Q-function will converge towards the optimal Q-function Q^*_κ.

Error states

This part of the section is based on [GW11]. The last model-free risk-sensitive method we will consider in this section is related to the notion of error states that we introduced along with the last model-base method of Section 7.2. The method we will look at in this subsection is the one originally proposed in the paper [GW11]. Recall that we in this framework consider a subset $\Phi \subset X$ consisting of states we call error states. The error states are all terminal states, and we allow a set $\Gamma \subset X$, with $\Gamma \cap \Phi = \emptyset$, of non-error terminal states, i.e. the terminal states of the MDP are the states in the set $\Gamma \cup \Phi$. We also augmented the MDP and introduced a new terminal state η by automatically transferring an agent from the states in $\Gamma \cup \Phi$ to the terminal state η at no cost. This imply that the states in $\Gamma \cup \Phi$ are no longer absorbing. The notion of risk introduced in [GW11] is given by

$$\rho_\pi(x) = P(\exists k \, i_k \in \Phi \mid \pi),$$

which is the probability of ending up in an error state when starting in state x and following the policy π. As we wrote in Section 7.2, the authors the proceed by defining a new cost function $\bar{g}(i,u,j)$ that we defined in Equation (7.14), but we repeat it here for convenience. We have that

$$\bar{g}(i,u,j) = \begin{cases} 1 & i \in \Phi \text{ and } j = \eta, \\ 0 & \text{otherwise.} \end{cases}$$

The model-free algorithm presented in [GW11] estimates the optimal policy by use of an algorithm similar to Q-learning (Section 6.2). The authors therefore introduce a function representing the state-action risk, it is defined by

$$\begin{aligned} \bar{Q}_\pi(x,u) &= E\left[\bar{g}(x,u,f(x,u,w)) + \alpha \rho_\pi(f(x,u,w))\right] \\ &= \sum_{x' \in X} p_{xx'}(u)(\bar{g}(x,u,x') + \alpha \rho_\pi(x'). \end{aligned}$$

As in Section 7.2, the problem we want to solve is

$$\min_{\pi \in \Pi} J_\pi$$

subject to

$$\text{for all } x \in X' : \rho_\pi(x) \leq \omega.$$

With this in mind, the authors introduce a third state-action function given by

$$Q_\pi^\xi(x,u) = \xi Q_\pi(x,u) + \bar{Q}_\pi(x,u), \tag{7.37}$$

where $Q_\pi(x,u)$ is the usual Q-function approximated by Q-learning and ξ again is some non-negative value. Note that the state-action function $Q_\pi^\xi(x,u)$ presented in [GW11] finds the difference between the two terms instead of the sum, as they consider a maximization problem whereas we consider the corresponding minimization problem. Observe also that the expression in Equation (7.37) is the state-action function corresponding to the cost function given in Equation (7.18) that we considered in the model-based case. The learning algorithm introduced in [GW11] approximates the usual state-action values $Q(x,u)$ and the new state-action function $\bar{Q}(x,u)$ by use of Q-learning while Q_π^ξ is approximated by taking the weighted sum of the two previously mentioned estimates. More rigorously the learning algorithm can for a given value of ξ be written as

$$Q_{t+1}(x,u) = Q_t(x,u) + \gamma_t(g + \alpha Q_t(x',u^*) - Q_t(x,u)) \tag{7.38}$$
$$\bar{Q}_{t+1}(x,u) = \bar{Q}_t(x,u) + \gamma_t(\bar{g} + \bar{\alpha}\bar{Q}_t(x',u^*) - \bar{Q}_t(x,u)) \tag{7.39}$$
$$Q_{t+1}^\xi(x,u) = \xi Q_{t+1}(x,u) + \bar{Q}_{t+1}(x,u), \tag{7.40}$$

where α and $\bar{\alpha}$ are the discount rate for the Q-approximation of the cost function and the risk, respectively, and where u^* is the greedy action at the state x'. We say in this context that an action u is preferable to u' if

$$Q_{t+1}^\xi(x,u) < Q_{t+1}^\xi(x,u'),$$

and in the case where

$$Q_t^\xi(x,u) = Q_t^\xi(x,u')$$

we prefer the action satisfying

$$\arg\min_{v \in \{u,u'\}} Q_t(x,v).$$

The authors in [GW11] claim that we are guaranteed convergence for the algorithm whenever $\alpha = \bar{\alpha}$, given that there exists at least one *proper* policy, and that the cost function of any non-proper policy is infinite. A proper policy is a policy that reaches the absorbing state, which here is η, with probability 1. Note that assuming that there only exist proper policies is equivalent to assuming Assumption 7.2.11.

In order to solve the optimization problem we are interested in the authors proposes to iteratively increase the value ξ and in each iteration estimate the policy π_ξ^* that optimize Equation (7.40) for the given value of ξ. We can then check whether the policy π_ξ^* is too risky by checking if some state $x \in X'$ has an associated risk greater than ω, where $X' \subseteq X \setminus (\Gamma \cup \Phi \cup \{\eta\})$ is the states we are interested in. When we at some point find a policy that is too risky we pick the policy found during the previous iteration as our optimal policy. By letting $\xi = 0$ in the first iteration we are able to find out if the problem even is

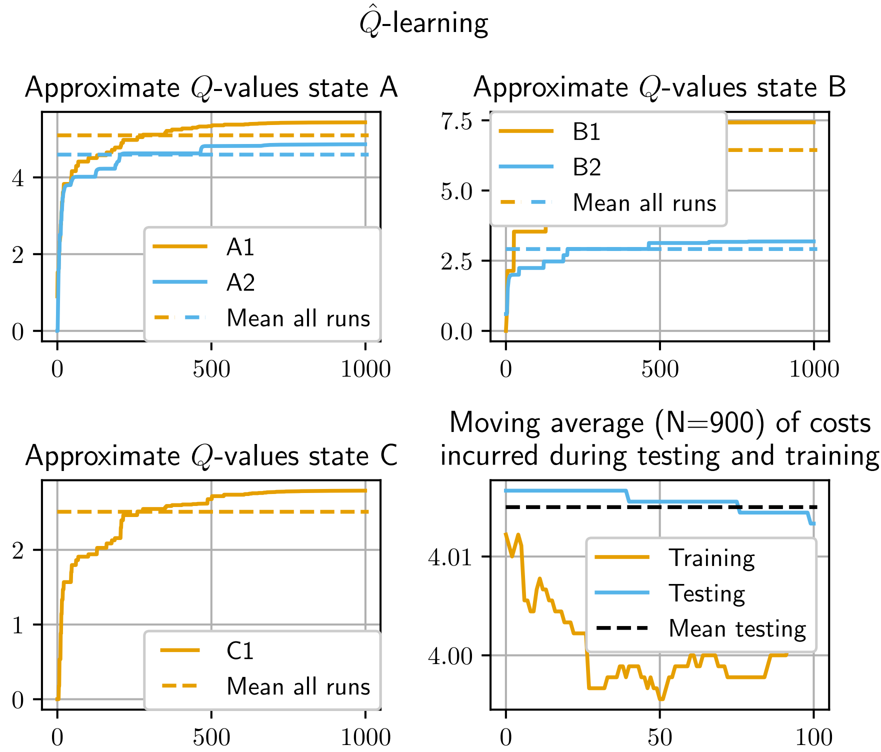

Figure 7.5: The evolution of Q-values and running average of simulated costs using $\hat{Q}$-learning, as well as the mean of the sampled values.

feasible, as we then are trying to minimize the risk, thus π_0^* is the minimal-risk policy. Therefore, $\rho_{\pi_0^*}(x) \geq \omega$ for some $x \in X'$ implies that the problem is not feasible, as we then would not be able to find a policy with a lower risk for any state $x \in X'$ than what we have for π_0^*. Let us now see how the method performs in practice by considering the SSP-problem we have used in previous numerical examples.

Numerical example

We now want to look into how the model-free risk-sensitive methods covered in this chapter perform on the SSP problem we so far have seen in Examples 6.3.1, 5.1.9 and 7.2.18. It will also be interesting to see if the policies found by the model-free methods coincide with the once we found with the model-based methods in Example 7.2.18.

Example 7.3.5 (Continuation of Examples 6.3.1, 5.1.9 and 7.2.18)**.** †
In this example we are going to apply the three different model-free risk-sensitive methods that we have presented in this section to the SSP problem that we previously have looked at in Examples 6.3.1, 5.1.9 and 7.2.18. Again, the regular SSP depicted in Figure 5.1 is described in Listing A.2, while the augmented version with an error state which is show in Figure 7.4 can be

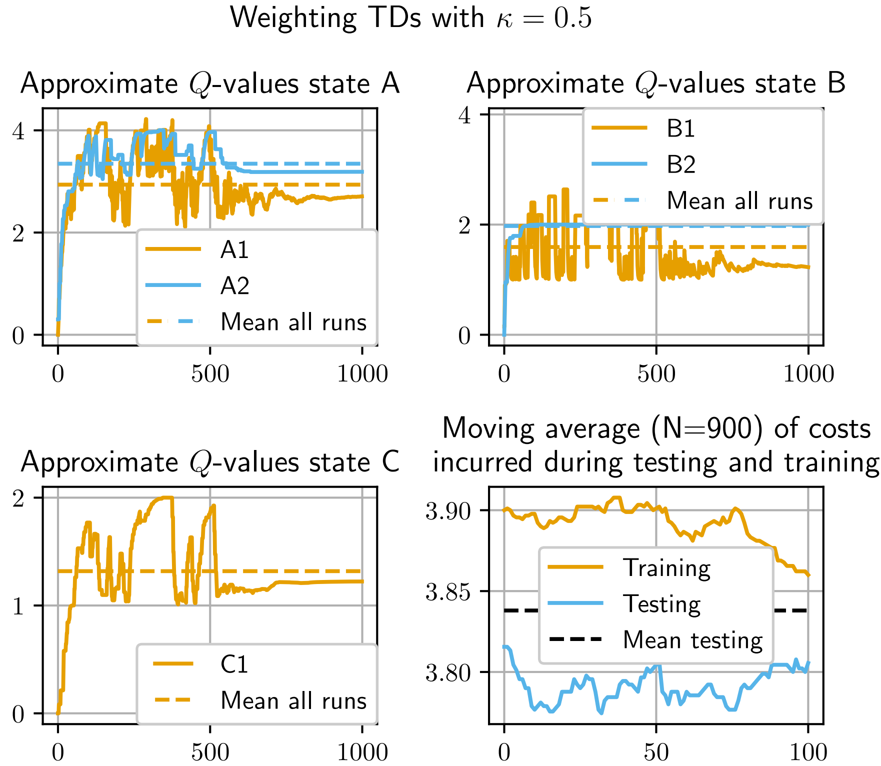

Figure 7.6: The evolution of Q-values and running average of simulated costs when weighting TD-rewards using $\kappa = 0.5$, as well as the mean of the sampled values

found in Listing A.5. The code used to simulate the states, and thus facilitate the simulations, can be found in Listing A.1 and the code implementing the model-free methods used in this example can be found in Listing A.9. We ran the different methods with learning rate $\gamma = 0.3$ and with $\epsilon = 0.15$. Note that $\hat{Q}$-learning tries to estimate the same policy that minimax promotes, which is a policy that is optimal under the assumption that the worst-case outcome of applying a control is the outcome we are going to have each time we use the given control in a given state. Therefore we set the discount rate $\alpha = 0.9$ for $\hat{Q}$-learning as that is the same value we used for minimax in Example 7.2.18, while we have $\alpha = 1$ for the two other methods.

For the method-specific parameters we used multiple values. For the method where we weight the TDs we let the risk-sensitivity parameter κ take the values $\kappa = 0.5, \kappa = 0.15$, and $\kappa = -0.5$. As we are considering a minimization problem we expect that the parameter value of $\kappa = 0.5$ should give us a risk-seeking policy, and the policy found using $\kappa = 0.15$ should give a risk-seeking policy that is somewhat more pessimistic than the one found using $\kappa = 0.5$, while the last policy based on the Q-value estimates found with the parameter value $\kappa = -0.5$ should be risk-adverse. For the method based on the error states formulation of risk the model-specific parameter is ω, which specifies the accepted level of risk

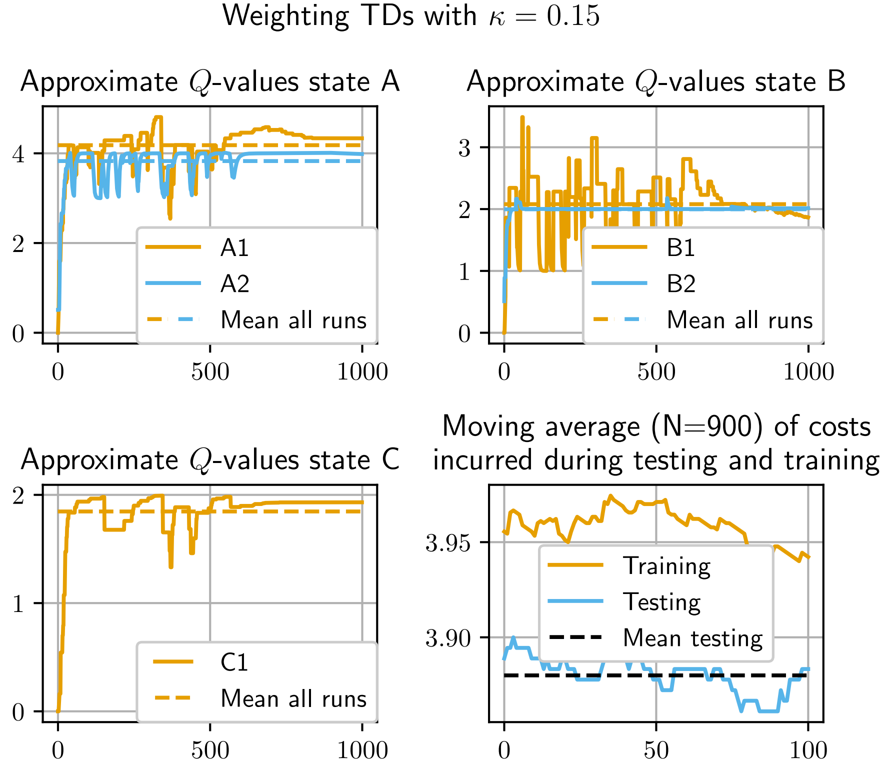

Figure 7.7: The evolution of Q-values and running average of simulated costs when weighting TD-rewards using $\kappa = 0.15$, as well as the mean of the sampled values.

for the policy we are looking to find in the states we are interested in. We ran the error states method using the values $\omega = 0.1$ and $\omega = 0.2$. The parameter ξ, which is used to weight the cost of the policy, is varied from $\xi = 0$ to $\xi = 1$. For each value of ξ we run the same set of simulations as we do for any other method and estimate the optimal policy given that the objective is to solve

$$\min_{\pi \in \{\pi' \in \Pi | \rho_{\pi'}(x) < \omega \ \forall x \in X'\}} (\xi J_\pi(x) + \rho_\pi(x))$$

for each $x \in X'$. If the policy found is less risky than the threshold ω we increase the value of ξ by 0.1 and do the same exercise again. We repeat this process until either ξ reach the value 1 and the next policy found still has an associated risk within the bound set by ω, or we for some value of ξ end up with a too risky policy, at which point we return the policy found for the previous value of ξ. For the first round, where the value of ξ is 0, we initialize the Q-values with value 0 for each state-action pair, while we for each subsequent round initialize the Q-tables with the Q-values found in the previous round.

Just like we did in Example 6.3.1, we plot the evolution of the Q-values observed during the simulations for each state along with the mean of the values. We also include the moving average of the incurred costs for both training and

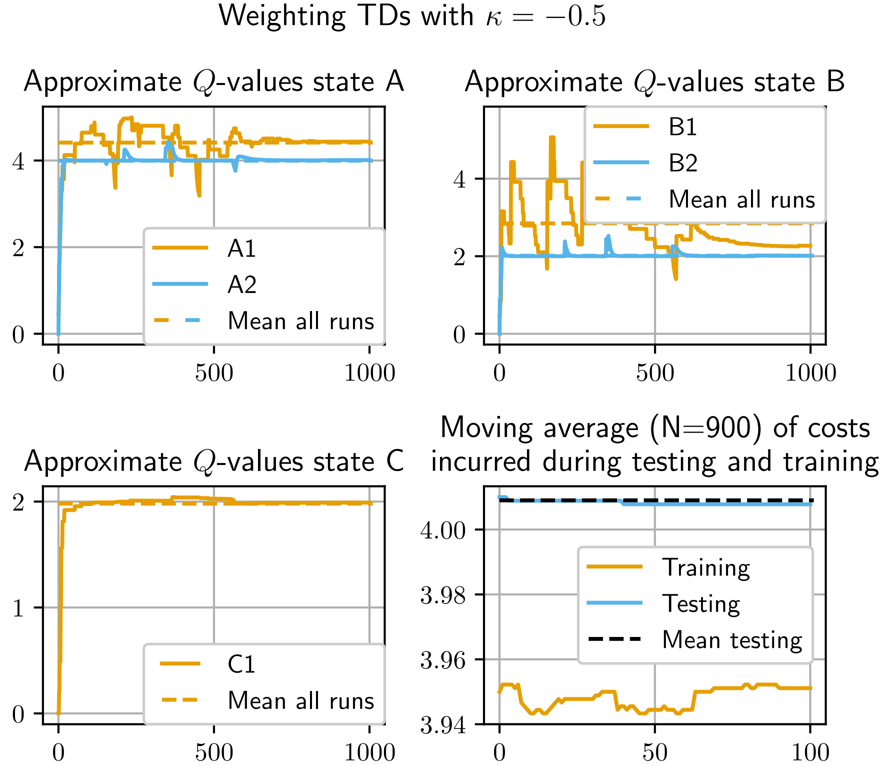

Figure 7.8: The evolution of Q-values and running average of simulated costs when weighting TD-rewards using $\kappa = -0.5$, as well as the mean of the sampled values.

testing. For the error state experiments we included the moving average of the empirically observed risk for training and testing as well, including the mean of the testing rounds. Note that the empirically observed risk of a policy is the mean of the $\bar{g}$ costs received by the agent during a number of rounds, and that this estimate is more accurate as the number of runs increase. This is because the $\bar{g}$ cost of 1 is received only when the controller exits the error state, which terminates the run, and as we would like to know the probability of ending up in the error state, we have that the mean over multiple rounds gives us exactly such an estimate. We can draw the same conclusion from Equation (7.15) in Proposition 7.2.9 as the mean of observed cumulative sums of the $\bar{g}$ costs is the maximum likelihood estimator of the expected value

$$E\left[\sum_{k=0}^{\infty} \bar{g}(i_k, \pi(i_k), w_k)\right].$$

However, keep in mind that as the policy changes during training we can not really draw any conclusion about the risk of any single policy from looking at the mean of the cumulative $\bar{g}$ costs observed during training (unless the Q-value estimates for every state has the same ordering over the training period, but

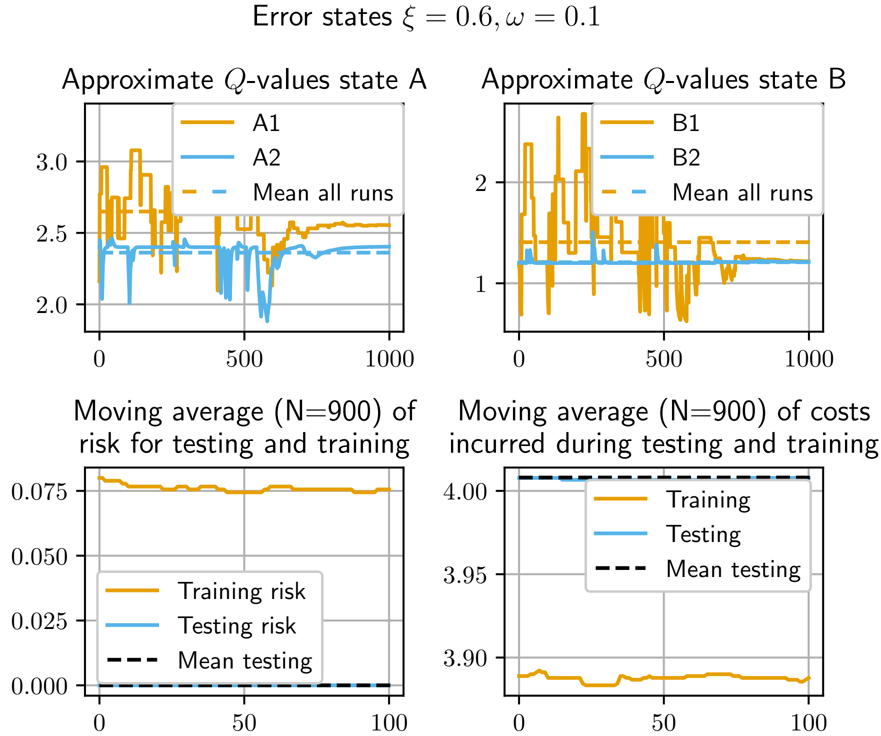

Figure 7.9: The evolution of Q-values and running average of simulated costs and risk using the error states method with a risk bound of $\omega = 0.1$ for the policy, as well as the mean of the sampled values.

keep in mind that we then would estimate the risk of the ϵ-greedy policy, and not the final policy found by the method), which is why we do not include this mean. The running average of the observed risk for the training runs is included in order to get some idea of how the riskiness of the current (ϵ-greedy) policy evolved over time.

We have plotted the results from running the $\hat{Q}$-learning in Figure 7.5. The results from the method where we weight the temporal differences with $\kappa = 0.5$ can be found in Figure 7.6, for the results with $\kappa = 0.15$ see Figure 7.7, while the data from the run with $\kappa = -0.5$ can be found in Figure 7.8. For the error states method we have plotted the results from the run with $\omega = 0.1$ in Figure 7.9 and the results we obtained by setting $\omega = 0.2$ can be seen in Figure 7.10.

We can make multiple interesting observations from the plots mentioned above. We can for instance find the policies found by the methods by looking at which action has the minimum Q-value for each state. In Table 7.4 we list the policies found by the methods. First, note that we from Figure 7.5 and Table 7.4 can see that the policy found by $\hat{Q}$-learning indeed is the same policy that we found by using minimax in Example 7.2.18 which is listed in Table 7.2. This shows that $\hat{Q}$-learning is able to approximate correctly the minimax algorithm in this instance, which is exactly what it is meant to do. For the method

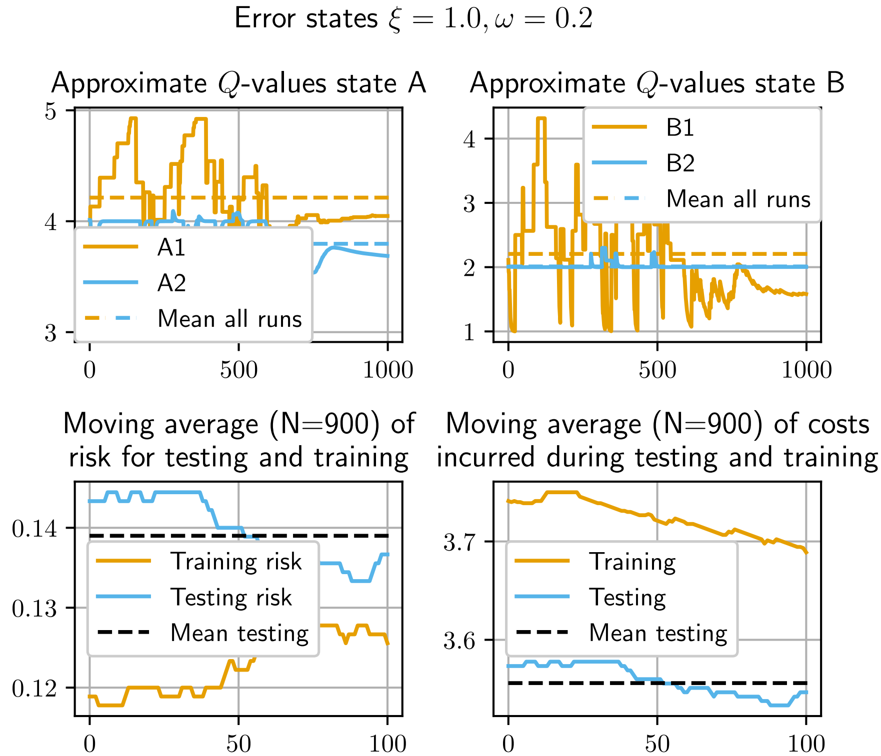

Figure 7.10: The evolution of Q-values and running average of simulated costs and risk using the error states method with a risk bound of $\omega = 0.2$ for the policy, as well as the mean of the sampled values.

Method and parameter values	$\mu^*(A)$	$\mu^*(B)$	$\mu^*(C)$	Empirical testing risk
$\hat{Q}$-learning, $\alpha = 0.9$	2	2	1	
Weighting TDs, $\kappa = 0.5$	1	1	1	
Weighting TDs, $\kappa = 0.15$	2	1	1	
Weighting TDs, $\kappa = -0.5$	2	2	1	
Error states, $\omega = 0.1, \xi = 1$	2	2		0
Error states, $\omega = 0.2, \xi = 1$	2	1		0.153

Table 7.4: Optimal policy under different risk-sensitive optimality conditions for the SSPs depicted in Figures 5.1 and 7.4.

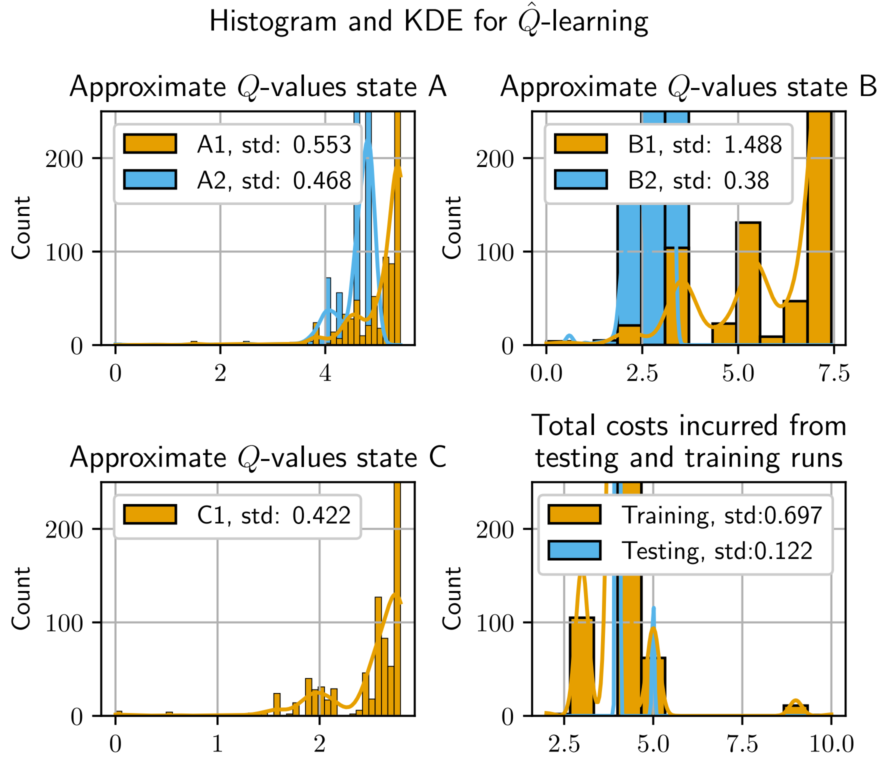

Figure 7.11: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for $\hat{Q}$-learning. The height has been cut at $y = 250$.

that weight the temporal differences we see that we get three different policies based on our choice of the risk-sensitivity parameter. As expected, the run with $\kappa = -0.5$ ends up with the most risk-averse policy, and does in fact arrive at the minimax policy. We then see that by increasing the risk-sensitivity parameter by setting it to $\kappa = 0.15$ we get a somewhat more risk-seeking policy compared with the minimax policy. This policy is actually one of the optimal policies when we only consider the expected cost of the policy. Observe that the policy we then get by increasing the risk-sensitivity parameter further to $\kappa = 0.5$ is the other optimal policy for the optimal expected cost problem. It is interesting to note that the only difference between the two policies that is optimal for the expected cost case is the action chosen in state A, as one of them has $\mu_1^*(A) = 1$, while the other has $\mu_2^*(A) = 2$. For the expected cost case there is no difference between the two options, as the expected cost under both actions are equal. However, for the risk-sensitive case where we consider variation of the cost to be a negative attribute, as it promotes uncertainty around the performance of the policy, the difference actually matters. We see that the weighted TDs method with the highest risk-sensitivity parameter indeed promoted the more risk-seeking policy, which is to have $\mu^*(A) = 1$, as this leads to the agent receiving a non-deterministic cost for the transition

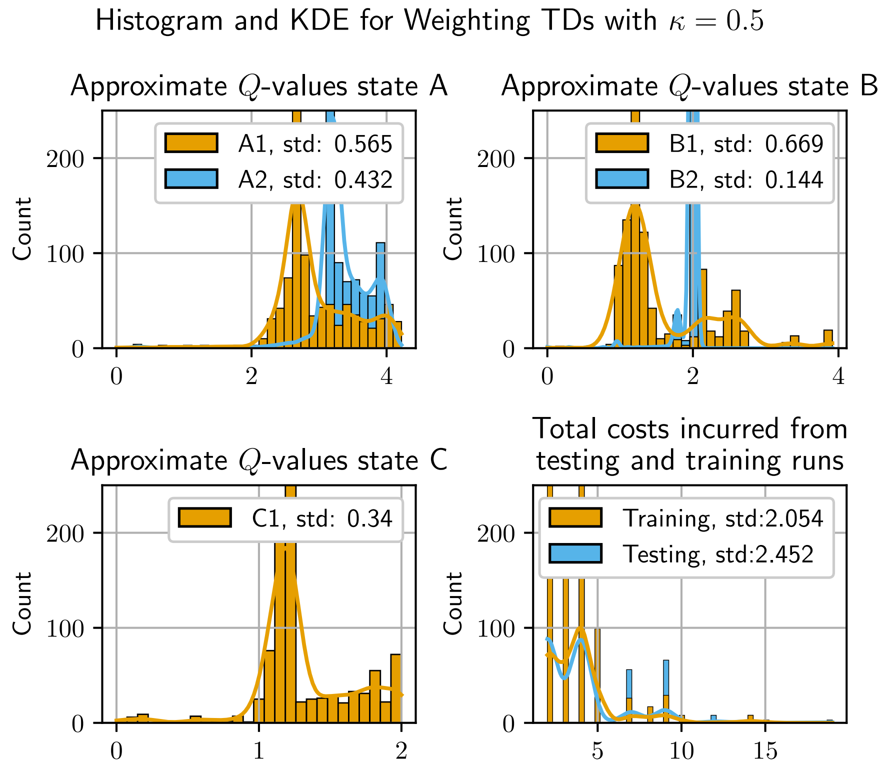

Figure 7.12: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies when weighting TD-rewards. The height has been cut at $y = 250$.

between state A and B. The other action, which is chosen by the same method when ran with a parameter value of $\kappa = 0.15$ in this instance, results in the same transition, but with a deterministic cost. This means that for the simulations we ran the weighting TDs method behaved as expected with the policy getting gradually more risky as we increase the risk-sensitivity parameter κ. For the last method dependent on the error states formulation of risk we see that when running with the parameter value $\omega = 0.1$ we actually got the minimax policy as well, which has a risk of zero, while we with the parameter value $\omega = 0.2$ ended up with the somewhat more riskier policy which is optimal in the case where we only want to minimize the expected cost. Note also that we in the first case only were able to reach $\xi = 0.6$, while we in the other case were able to reach $\xi = 1$, which means that we for the experiment with $\omega = 0.1$ encountered a too risky policy about half-way through the set of ξ values we used. Observe that we in Example 7.2.18 found that for the model-based error state method with $\omega = 0.1$ we could not increase the ξ parameter from 0.25 to 0.5 without having an optimal policy that was to risky. We can therefore conclude that the model-free error states method in this instance was not able to find the correct optimal policy for the runs with $\xi = 0.5$ and $\xi = 0.6$ when $\omega = 0.1$ as the method first stopped when it found a too risky policy after the run with

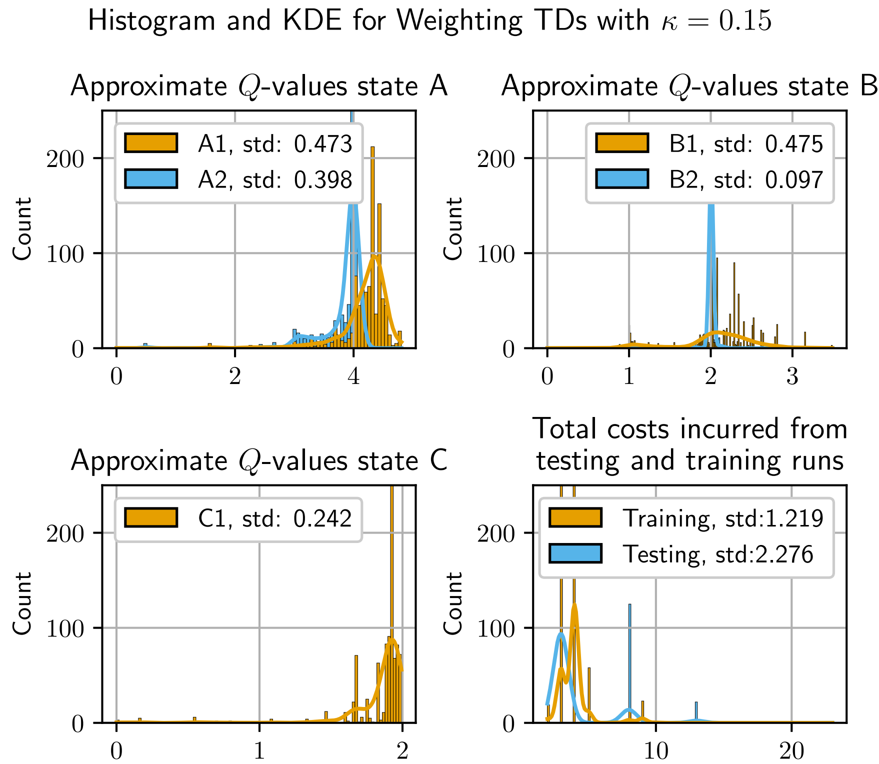

Figure 7.13: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies when weighting TD-rewards. The height has been cut at $y = 250$.

$\xi = 0.7$. The method therefore returned the policy it found with $\xi = 0.6$. Note however that we for $\omega = 0.2$ were able to estimate the optimal policy correctly.

In addition to plotting the results we also, as in Example 6.3.1, plotted histograms, along with the kernel density estimate, showing the distribution of the data underlying the plots mentioned earlier. We do this is in order to get an idea of both how the distribution of the cost incurred by the agent is during the training rounds of the final policy and in order to see how much variation we had in the approximation of the Q-values during the training. The histogram regarding the $\hat{Q}$-learning is plotted in Figure 7.11. The histograms for the method weighting temporal differences is plotted in the figures Figures 7.12 to 7.14 with the parameter values being $\kappa = 0.5, \kappa = 0.15$ and $\kappa = -0.5$, respectively. For the method relying on the error states notion of risk we have the histogram in Figure 7.15 for the run with the parameter value $\omega = 0.1$ and $\xi = 0.6$, while we in Figure 7.16 have plotted the data generated by the run with $\omega = 0.2$ and $\xi = 1$.

From the histograms shown in Figures 7.11, 7.14 and 7.15 we can see that the most risk-averse methods, which is $\hat{Q}$-learning, the weighting of TDs with $\kappa = -0.5$ and the error states method with $\omega = 0.1$, all had a standard deviation close to 0.1 for the costs incurred during testing. The fact that the standard

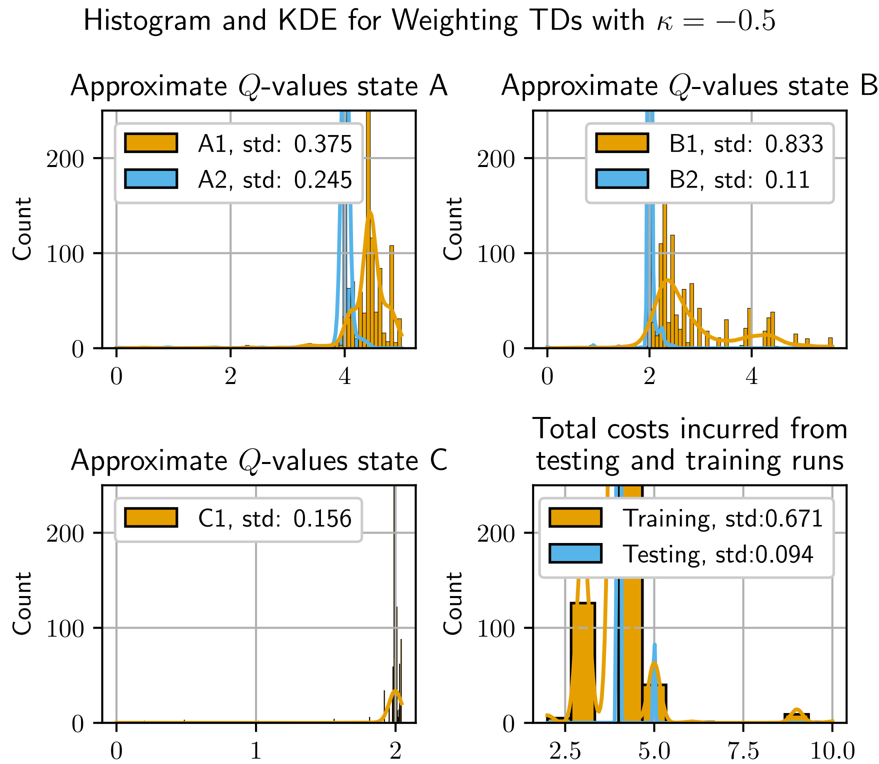

Figure 7.14: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies when weighting TD-rewards. The height has been cut at $y = 250$.

deviation was so similar for these methods was no surprise as they all ended up with finding the minimax policy to be the optimal policy. The interesting thing is that a standard deviation of 0.1 is significantly lower than what we get with the policies found by the other methods. It is also interesting to see that even though the weighting TDs method with $\kappa = 0.15$ and the error states method with $\omega = 0.2$ takes the same actions in state A and B the former has a standard deviation for the costs incurred during testing that is about 60% greater than that of the latter method. This is probably due to the fact that the two underlying SSPs used by the methods behave differently when the policy satisfies $\mu(A) = 2$ and $\mu(B) = 1$. For example, in the case where the controller receive the cost 5 in state B after applying control 1 we see from Figure 7.4 that the run ends immediately if we use the augmented SSP, while we in the other case would need to apply control 1 in state B again, risking receiving the high cost of 5 units again.

Another interesting observation is that the standard deviation of the cost received during testing in this instance for the two policies found by using the weighting of TDs method with the parameter values $\kappa = 0.5$ and $\kappa = 0.15$ does only differ slightly. As we only compare the results of one batch of testing we really can not say much about how this will be on average, but it indicates that

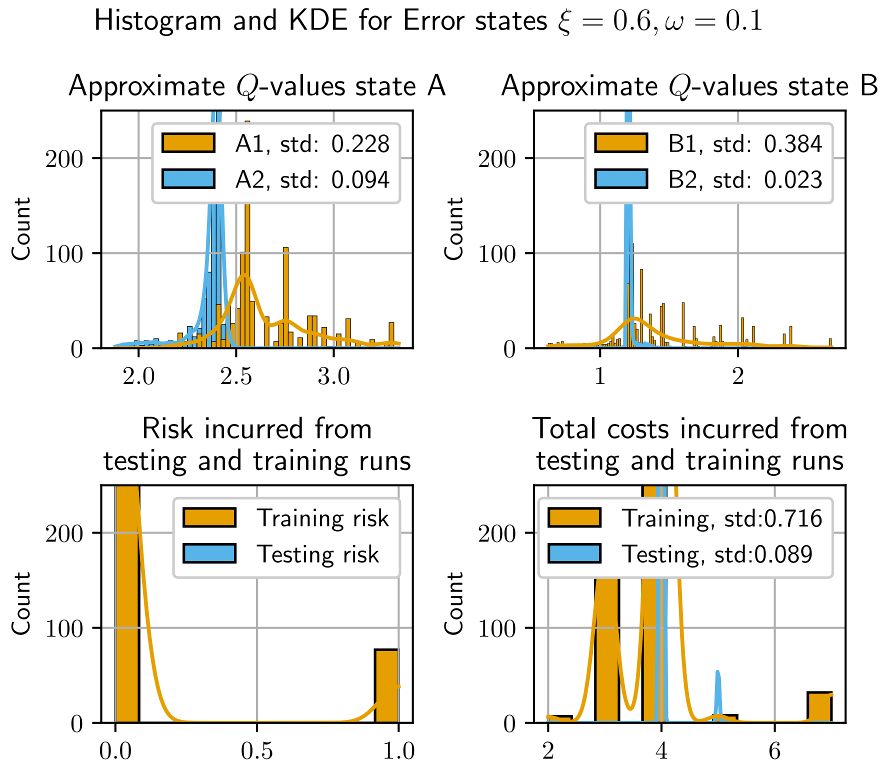

Figure 7.15: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for error states method. The height has been cut at $y = 250$.

choosing action 1 instead of action 2 in state A in the non-augmented SSP does not have a great impact on the standard deviation of the observed costs.

As we in this example only take a detailed look at one run for each method and choice of parameter value, we do not really get a good idea about how robust our results are. We therefore, just as we did in Example 6.3.1, plot a bar plot showing the distribution of policies generated by each method and parameter choice when we generated a number of 1000 policies for each method and parameter value. For the error states methods we also provide a bar plot showing the distribution of ξ values used to generate the policies returned by the method for each of the choices of ω. The plot is shown in Figure 7.17. The first interesting thing to note is that three of the methods are really stable, as they generated the same policy for each of the 1000 runs of training. Surprisingly, these three methods are $\hat{Q}$-learning, weighting TDs with $\kappa = -0.5$ and the error states method with $\omega = 0.1$, which are the three most risk-averse methods. The fact that the error states method with $\omega = 0.1$ only generates the policy A2B2 is actually not very surprising as that is the only policy with a risk bellow 0.1. We also see that the weighting TDs method with $\kappa = 0.5$ generates the same policy about nine out of ten times and that the policy it creates most often in fact is the most optimistic. It is also interesting that the method only creates

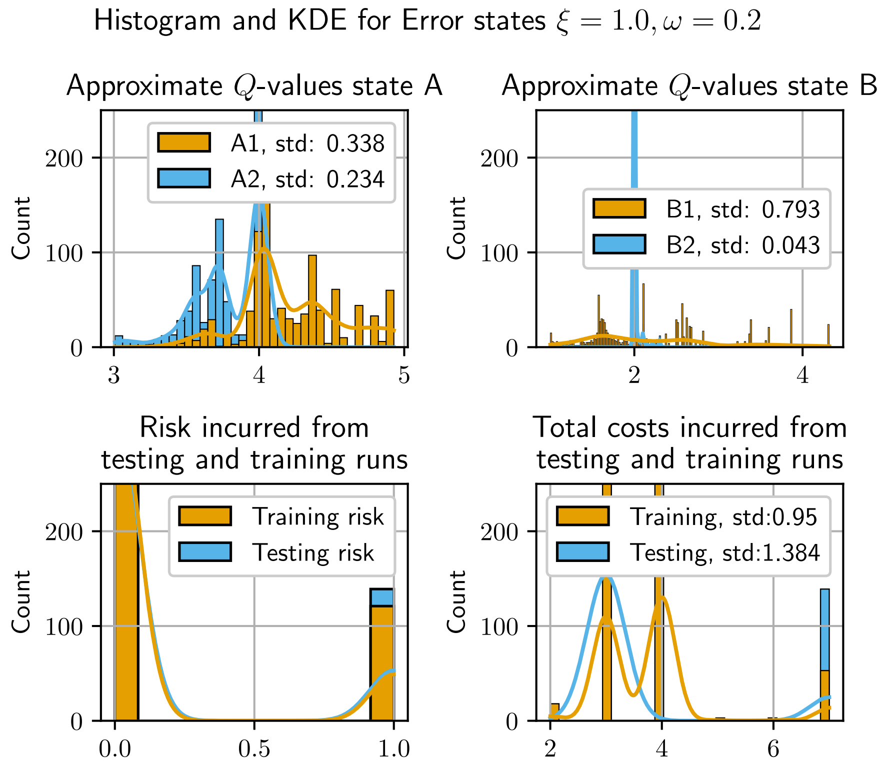

Figure 7.16: Histogram of Q-values attained during training as well as costs incurred during training and testing of the final and intermediate policies for error states method. The height has been cut at $y = 250$.

one other policy, which is the policy that is optimal in the risk-neutral case. For the weighting of TDs with $\kappa = 0.15$ we see that it actually generate all of the different possible policies, and that each of them is generated with about the same frequency. It is also interesting to note that even though the method should be risk-seeking, as we have $\kappa > 0$, we only generate the most optimistic policy about 15% of the time, while the most risk-averse policy is generated about 30% of the time. For the error states method with $\omega = 0.2$ we see that the actual optimal policy only is found about once in four runs, while the safer, more risk-averse policy, is generated in about three out of four runs. Note also that the error states method with $\omega = 0.2$ almost always reach $\xi = 1$, which indicates that we maybe should see what happens if we allow ξ to take greater values that 1. In addition, observe that for $\omega = 0.1$ the story is different as the ξ value reached varies quite a lot.

We can conclude that the methods we have looked at here are varying in their level of consistency, but it looks like the more risk-averse methods are the most stable, with the most risk-seeking algorithm also being quite robust. Let us now end this chapter with a brief comparison of the risk-sensitive methods presented in this chapter.

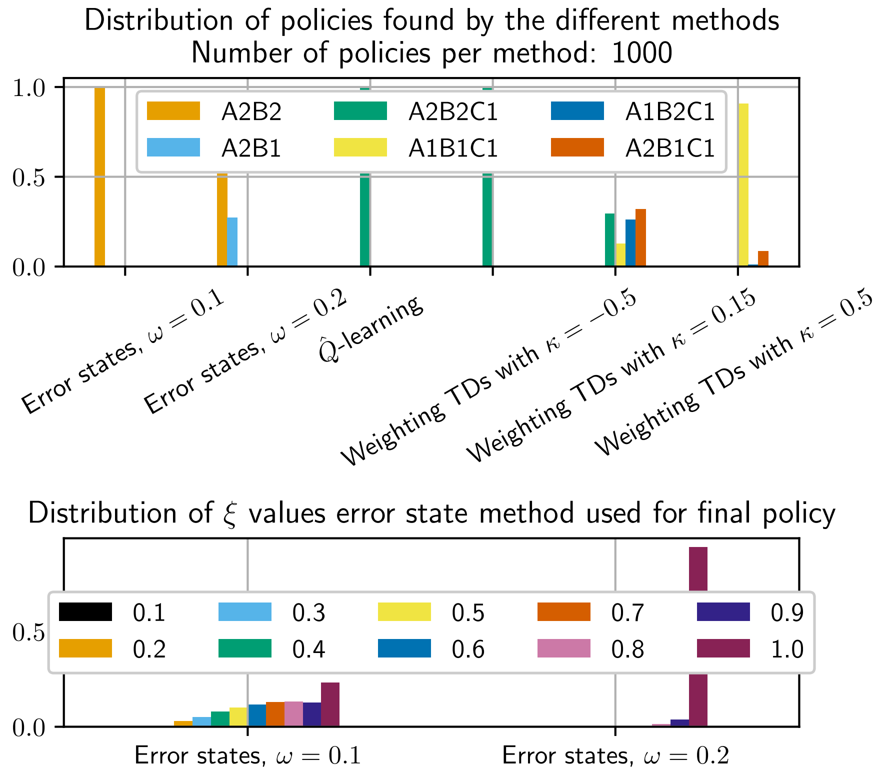

Figure 7.17: Bar plot showing distribution of policies created with the different risk-sensitive RL methods introduced in this chapter with the parameter values we have considered in this example, as well as distribution of ξ values used to generate final policy for error states method. Legend notation explanation: A1B1C1 denotes the policy taking action 1 in each state.

7.4 Comparison of risk-sensitive control methods

We have in this chapter considered different methods for risk-sensitive control. We have introduced three different model-based methods, and three other model-free algorithms. We can argue that each of the model-based methods we have presented have a counterpart among the model-free algorithms we considered. For example, the cost function and Q-values of a policy under the minimax-criterion we looked at in Section 7.2 is what we try to approximate by using $\hat{Q}$-learning. We also have that the two methods based on the error states representation of risk are quite similar, as the model-based method is designed around using the information available in a model-based framework to find exactly the cost functions we are trying to estimate using the model-free algorithm introduced in [GW11] which we presented in Section 7.3. For the two last methods, which is using the exponential utility and weighting TDs, the similarity is that they both have a risk-sensitivity parameter that allows the controller to choose how risk-seeking or risk-averse we would like our policy to be. However, the interpretation of the parameters are different and the

range of values the parameters can take is different as $\kappa \in (0, 1)$ while $\beta \in \mathbb{R}$, conditioned on the choice of β being such that Equation (7.12) holds. We have also seen from Table 7.2 in Example 7.2.18 and Table 7.4 in Example 7.3.5 that even though the two methods create the same policies they do not agree on the ordering of the policies regarding how risk-averse the policies are. From the tables we can see that for the parameter choices of $\beta = -0.075$ and $\kappa = 0.5$ both methods produce the same policy, and with the parameter values being $\beta = -1$ and $\kappa = 0.15$ the two methods produced a second policy, and again the two methods end up with the same policy. Note however that as we have decreased the value of β the exponential utility function would interpret the second policy as being more risk-seeking, while when we decrease the value of κ from 0.5 to 0.15 we would expect a more risk-averse policy. We therefore conclude that the two methods does not necessarily have the same interpretation of how risk-averse a policy is and the ordering of policies when ordering them according to their risk-sensitivity.

A nice thing about the method weighting TDs is that the κ parameter has a nice interpretation. For negative values it tells us something about where we lie on a scale from 0 to -1, where 0 is being risk-neutral while -1 is being really risk-averse by trying to obtain the minimax policy, given that we are considering a minimization of cost problem. On the other hand, for positive values of κ we can view it as being able to tell where we want to be on the scale between risk-neutral and risk-seeking, with $\kappa = 0$ being risk-neutral, and $\kappa = 1$ being very risk-seeking in the sense that we try to approximate the Q-values of a method where we minimize over the set of policies assuming the best-case outcome of each action. In short, the κ parameter tells us something about where we are on a scale between trying to estimate a very optimistic policy and the minimax policy with the risk-neutral case being midway between the two. As we have seen before in Section 7.2 we also have a nice interpretation of the β parameter as we have,

$$\frac{1}{\beta} \log E\left[e^{\beta G}\right] = E[G] + \frac{\beta}{2}\operatorname{Var}(G) + \mathcal{O}(\beta^2),$$

which tells us that β says something about how heavily we should weight the variance of the cost. The final method-specific parameter that we have not discussed yet is the ω parameter used in the error states method. The interpretation of the ω parameter is quite straight forward, as it say something about the maximum probability for transitioning into an error state from any of the states $x \in X'$ that we allow for the policies we consider. We then look for the policy that on average achieve the lowest cost among the policies that fulfils this risk-criteria.

To compare the behaviour of the model-based methods presented in Section 7.2 we can take a look at how they handled the St. Petersburg paradox. The minimax policy was only willing to bet if the stake was 2 or less, which is probably a stake no sensible person would be willing to give a gambler, as it would never give a profit. Thus, in practice, the minimax policy would not allow for any sort of gamble in this case (or any case, really). For the exponential utility function we see that we would need a value of β pretty close to 0 in order to be willing to bet much more than 2 units. As we can see from Figure 7.1 a gambler with $\beta = -0.1$ would be willing to bet around 4 units, while a less risk-averse player with $\beta = -0.01$ would be willing to bet around 7 units.

From the error states method we saw that when allowing to lose money with probability $p = 0.5$ (that is, $\omega = 0.5$) we would still only be willing to bet 4 units. In other words, having a parameter value of $\beta = -0.1$ in the exponential utility case roughly corresponds to having $\omega = 0.5$ for the error state method in this case. At the same time, we see that the willingness to bet 7 units with a parameter value of $\beta = -0.01$ would mean that the gambler would need to be willing to sustain a loss with probability 0.75. We are able to see this from the discussion in Example 7.2.16 as we found that for a given stake k we would be willing to take the bet if

$$\omega \geq \sum_{i=1}^{\lceil \log_2(k) \rceil - 1} \frac{1}{2^i}.$$

Thus, by letting $k = 7$ we have that

$$\omega \geq \sum_{i=1}^{\lceil \log_2(7) \rceil - 1} \frac{1}{2^i} = \sum_{i=1}^{3-1} \frac{1}{2^i} = \sum_{i=1}^{2} \frac{1}{2^i} = \frac{3}{4}.$$

We can therefore see that by tuning the risk-sensitivity parameters we are able to get quite similar behaviour for the policies found by using the exponential utility function and the error states method. However, it is much easier to interpret the value of the ω parameter compared with the β parameter. On the other hand, the β parameter allows for more flexibility. For example, with $\beta = -0.01$ we were willing to bet up to around 7 units, while the only way to be willing to take the same bet using the error states method would be to have $\omega = \frac{3}{4}$, but we would then be just as willing to bet 8 units. Keep in mind that this is the case for the St. Petersburg Paradox when we are defining the error state as being whenever we lose money, so this need not be the situation in general, but it indicates that there is some trade-off here between the two methods and their interpretability and flexibility.

In this chapter we have considered two numerical examples, namely Examples 7.3.5 and 7.2.18. From Tables 7.2 and 7.4 we see that combined the model-free methods from Example 7.3.5 actually managed to recreate every method we found by using the model-based methods in Example 7.2.18. The two model-free runs of the error states method managed to find the same policy as their model-based counterpart. As mentioned before, the $\hat{Q}$-learning method managed to end up with the minimax policy, which is exactly what the algorithm is designed to do. For the model-free method weighting the temporal differences we see that by varying the κ parameter we were able to estimate both methods found by using the exponential utility function in Example 7.2.18 as well as the minimax policy. However, as we saw in Example 7.3.5 not all of the methods generated the same policy consistently, so when applying these methods on real-world problems we should run each of them several times and compare the different policies generated by looking at their empirically found cost and risk.

The six methods we have looked at in this chapter are quite different. Two of them, $\hat{Q}$-learning and minimax, try to find the optimal policy when we assume that the worst-case outcome is the result of each transition. This means that these methods are nice to use e.g., when we are uncertain about the data that we have and therefore would like to plan for the worst-case outcome. Being uncertain about the data can in this instance mean that we do not know if the

sampled simulations we have to train our RL algorithm is representative, or that we do not entirely trust the estimated probabilities of our model-based problem. These methods are also strong candidates when it is really important to not allow for small probabilities of severe outcomes. In addition, we have the exponential utility method and the weighting of the TDs method that both rely on a tuning parameter that controls the risk-sensitivity of the methods. Thus these methods are practical if we would like to control *how* risk-sensitive the resulting policy should be. For the model-free weighting TDs method the κ parameter allows us to lie somewhere on the scale between being really optimistic and minimax, while the β parameter used in the exponential utility function controls how much, and with what sign, the variability of the cost should count towards the value we are looking to optimize. The last two methods we have looked at are concerned with a definition of risk that is dependent on there being defined at least one error state, as it sees a policy as more risky if it has a higher probability of ending up in one of these error states compared with some other policy. Thus this method is preferable if it is easy to define these error states, and if it is important to control the probability of the controller ending up in one of these states. We therefore see that the choice of which method to use is heavily dependent on the problem at hand, and also on whether we have access to a model of the underlying system.

In order to extend the work done in this chapter we could have included additional risk-sensitive methods, such as any of those that we have not looked at that is covered in the paper [GFF15], which is a review paper on risk-sensitive reinforcement learning. If we had the time we could also have looked at the methods presented in the papers [TDM12], [Ach+17] and [Cho+17]. In the three papers the authors, among other things, find policy gradient types of algorithms that estimates the optimal policies. The fact that the algorithms are policy gradient type of algorithms means that the algorithms rely on there being some parametrized family of policies for which we want to do the optimization over. The way these papers introduce the notion of risk-sensitivity is also quite different from the methods presented in this chapter, as e.g. [Ach+17] consider constrained Markov decision processes, while [Cho+17] uses the notion of risk-measures by utilising CVaR. Looking into, and comparing these methods are something we will leave for further work.

CHAPTER 8

Concluding remarks

We have in this work taken a stroll through a rather small part of the jungle of optimal control theory. We started by introducing optimal control through finite horizon dynamic programming in Chapter 3. In the chapter that followed, Chapter 4, we took a step back and looked at the theory of abstract dynamic programming and some general methods for finding solutions of dynamic programming problems in the abstract DP framework. Then, in Chapter 5 we applied the abstract dynamic programming theory from the previous chapter in order to give some new proofs for the convergence of the value iteration and policy iteration methods when used to solve infinite horizon dynamic programming problems. We then introduced reinforcement learning and Markov decision processes in Chapter 6. In addition, we explained how dynamic programming and reinforcement learning in essence are two sides of the same coin when we are doing optimal control on Markov decision processes, as these processes also are underlying the dynamic programming problems we had covered in the previous chapters. We then ended our walk with Chapter 7 which is the highlight of this thesis. In that chapter, we considered the important topic of risk-sensitive control. We introduced three different model-based methods and three different model-free algorithms, looked at their performance on a numerical example and ended the chapter with a comparison of the different methods.

As we have covered many different topics in this thesis there are multiple natural directions for future work. One such example is to look further into the theory of abstract dynamic programming. The theory covered in this thesis only consider contractive models, but there has been done work related to semi-contractive and non-contractive models as well. It could be interesting to look at the introduction of risk-sensitivity in such models, and to see if doing so could extend our understanding of risk-sensitive reinforcement learning.

Another topic of interest could be risk-sensitive reinforcement learning in the case where we would need approximations in value space and/or approximations in policy space. That is, where we have a state space and/or control space that is too large for the classical tabular reinforcement learning methods that we have considered in this thesis. There is already many research papers on this topic, but it is also a field with a lot of open questions that would be interesting to work on. We could also consider other notions of risk, such as risk-measures and the introduction of constraints that the feasible policies must satisfy.

In this work we have only considered the theory of discreet time stochastic optimal control, but there exist a lot of theory on continuos-time stochastic

optimal control as well. Finding ways to introduce risk-sensitivity for the continuos-time case could also prove to be an interesting exercise. We could for instance try to introduce error states in the continuos-time optimal control framework. A continuation of that kind of work could then be to check whether reinforcement learning algorithms manage to correctly estimate the optimal cost function for risk-sensitive continuos-time optimal control problems even though RL methods depend on discretisation in order to approximate continuous-time solutions.

Appendices

APPENDIX A

Code

Listing A.1: ssp_ex; Implementation of SSP considered in Example 5.1.9

```
import json
import numpy as np
rng = np.random.default_rng(5)

class State:
    """ Class to hold informaiton about the different states.
    This class features a method that supply the minimum expected
    cost of the state when given a cost function, and the expected
    cost of following a given policy.
    """

    def __init__(self, name: str, action_probs: dict,
                 discount_rate: float=1):
        self.name = name
        self.discount_rate = discount_rate
        self._set_actions(action_probs)

    def _set_actions(self, action_probs: dict):
        self.actions = action_probs.keys()
        self.action_probs = action_probs

    # --------------------------- Regular DP methods ---------------------------
    def get_policy_exp_cost(self, policy: dict, cost_func: dict):
        action = policy[self.name]
        exp_cost = 0
        for i in range(len(self.action_probs[action]['states'])):
            exp_cost += self.action_probs[action]['probs'][i]\
                *(self.action_probs[action]['costs'][i]
                  + self.discount_rate\
                  *cost_func[self.action_probs[action]['states'][i]])
        return exp_cost

    def get_min_exp_cost(self, cost_func: dict):
        """ Calculate the expected cost given cost function.
        """
        min_cost = np.inf
        min_act = list(self.actions)[0]
        for action in self.actions:
            exp_cost = 0
            for i in range(len(self.action_probs[action]['states'])):
                exp_cost += self.action_probs[action]['probs'][i]\
                    *(self.action_probs[action]['costs'][i]
                      + self.discount_rate\
                      *cost_func[self.action_probs[action]['states'][i]])
```

```
            if exp_cost < min_cost:
                min_cost = exp_cost
                min_act = action
        return min_cost, min_act

    # ---------------------------- Minimax methods ---------------------------
    def get_policy_minimax_cost(self, policy: dict, cost_func: dict,
                                maximin = False,):
        action = policy[self.name]
        worst_cost = 0
        for i in range(len(self.action_probs[action]['states'])):
            tmp_cost = self.action_probs[action]['costs'][i] \
                + self.discount_rate \
                *cost_func[self.action_probs[action]['states'][i]]
            if (not maximin) and (tmp_cost > worst_cost):
                worst_cost = tmp_cost
            elif maximin and (tmp_cost < worst_cost):
                worst_cost = tmp_cost
        return worst_cost

    def get_opt_minimax_cost(self, cost_func: dict, maximin = False,):
        """ Calculate the minimax cost given cost function.
        """
        worst_costs = []
        actions = list(self.actions)
        for action in actions:
            worst_cost = 0
            for i in range(len(self.action_probs[action]['states'])):
                tmp_cost = self.action_probs[action]['costs'][i] \
                    + self.discount_rate \
                    *cost_func[self.action_probs[action]['states'][i]]
                if (not maximin) and (tmp_cost > worst_cost):
                    worst_cost = tmp_cost
                elif maximin and (tmp_cost < worst_cost):
                    worst_cost = tmp_cost
            worst_costs.append(worst_cost)
        worst = max(worst_costs) if maximin else min(worst_costs)
        return worst, actions[worst_costs.index(worst)]

    # ----------------------- Exponential utility methods ----------------------
    def get_policy_exp_util_cost(self, policy: dict, cost_func: dict,
                                 rs_factor: float=-0.1):
        action = policy[self.name]
        exp_cost = 0
        for i in range(len(self.action_probs[action]['states'])):
            exp_cost += self.action_probs[action]['probs'][i]\
                *(np.exp(rs_factor*self.action_probs[action]['costs'][i])
                  *self.discount_rate\
                  *cost_func[self.action_probs[action]['states'][i]])
        return 1/rs_factor*np.log(exp_cost) if exp_cost != 0 else exp_cost

    def get_min_exp_util_cost(self, cost_func: dict,
                              rs_factor: float=-0.1):
        """ Calculate the exponential utility cost given cost function.
        """
        min_cost = np.inf
        min_act = list(self.actions)[0]
        for action in self.actions:
            exp_cost = 0
            for i in range(len(self.action_probs[action]['states'])):
                exp_cost += self.action_probs[action]['probs'][i]\
                    *(np.exp(rs_factor*self.action_probs[action]['costs'][i])
```

```
                    *self.discount_rate\
                    *cost_func[self.action_probs[action]['states'][i]])
            if exp_cost < min_cost:
                min_cost = exp_cost
                min_act = action
        if min_cost != 0:
            return 1/rs_factor*np.log(min_cost), min_act
        else:
            return min_cost, min_act

    # --------------------------- Error state methods --------------------------
    def get_policy_es_cost(self, policy: dict, cost_func: dict, risk_func:dict,
                           xi: float=0):
        action = policy[self.name]
        exp_cost = 0
        risk_comp = 0
        for i in range(len(self.action_probs[action]['states'])):
            exp_cost += self.action_probs[action]['probs'][i]\
                *(xi*self.action_probs[action]['costs'][i]
                  + self.action_probs[action]['risk_cost'][i]
                  + self.discount_rate\
                  *cost_func[self.action_probs[action]['states'][i]])
            risk_comp += self.action_probs[action]['probs'][i]\
                *(self.action_probs[action]['risk_cost'][i]
                  + self.discount_rate\
                  *risk_func[self.action_probs[action]['states'][i]])
        return exp_cost, risk_comp

    def get_min_es_cost(self, cost_func: dict, risk_func: dict, xi: float=0):
        """ Calculate the expected cost given cost function.
        """
        min_cost = np.inf
        min_risk = np.inf
        min_act = list(self.actions)[0]
        for action in self.actions:
            exp_cost = 0
            risk_comp = 0
            for i in range(len(self.action_probs[action]['states'])):
                exp_cost += self.action_probs[action]['probs'][i]\
                    *(xi*self.action_probs[action]['costs'][i]
                      + self.action_probs[action]['risk_cost'][i]
                      + self.discount_rate\
                      *cost_func[self.action_probs[action]['states'][i]])
                risk_comp += self.action_probs[action]['probs'][i]\
                    *(self.action_probs[action]['risk_cost'][i]
                      + self.discount_rate\
                      *risk_func[self.action_probs[action]['states'][i]])
            if exp_cost < min_cost:
                min_cost = exp_cost
                min_risk = risk_comp
                min_act = action
        return min_cost, min_act, min_risk

    # --------------------------- RL sampling method ---------------------------
    def sim(self, action: str):
        """ Simulate the transition from this state when the inputed
        action is chosen.
        """
        r = rng.random()
        p = 0
        for i in range(len(self.action_probs[action]['states'])):
            p += float(self.action_probs[action]['probs'][i])
```

```
            if r < p:
                return self.action_probs[action]['states'][i]
        return self.action_probs[action]['states'][i]

def max_cost_funcs_diff(cf: dict, other: dict):
    """ Funciton that calculates the sup-norm of two cost functions
    defined as dicts.
    """
    assert cf.keys() == other.keys()
    max_diff = 0
    for state in cf.keys():
        s_diff = np.abs(cf[state]-other[state])
        max_diff = s_diff if s_diff > max_diff else max_diff
    return max_diff

def get_rng():
    return rng

def get_environment(discount_rate: float=1, path: str='ex_SSP.json'):
    # load problem information
    with open(path) as fp:
        state_actions_probs = json.load(fp)
    # create states and cost func
    env = {}
    for state, action_probs in state_actions_probs.items():
        env[state] = State(state, action_probs, discount_rate)
    return env
```

Listing A.2: ex_SSP.json; JSON-file defining SSP implemented by Listing A.1

```
{
    "A": {
        "1": {
            "states": [
                "B",
                "C"
            ],
            "probs": [
                0.5,
                0.5
            ],
            "costs": [
                1,
                3
            ]
        },
        "2": {
            "states": [
                "B"
            ],
            "probs": [
                1
            ],
            "costs": [
                2
            ]
        }
    },
    "B": {
        "1": {
            "states": [
                "B",
```

```
                "t"
            ],
            "probs": [
                0.15,
                0.85
            ],
            "costs": [
                5,
                1
            ]
        },
        "2": {
            "states": [
    "B",
                "t"
            ],
            "probs": [
    0.01,
                0.99
            ],
            "costs": [
    1,
                2
            ]
        }
    },
    "C": {
  "1": {
      "states": [
    "B"
      ],
      "probs": [
    1
      ],
      "costs": [
    0
      ]
  }
    },
    "t": {
        "terminate": {
            "states": [
                "t"
            ],
            "probs": [
                1
            ],
            "costs": [
                0
            ]
        }
    }
}
```


Listing A.3: ssp_DP.py; Implementation of value iteration and policy iteration used to solve SSP problem in Example 5.1.9

```
from ssp_ex import max_cost_funcs_diff, get_environment

print('\n{} Regular DP {}'.format(34*'#', 34*'#'))
# declear variables and get states
eps = 1E-16
```

```
states = get_environment()

# ------------------------------ Value Iteration ------------------------------
print('{} Value Iteration {}'.format(31*'-', 32*'-'))
# create initial cost func
prv_cost_func = dict.fromkeys(states.keys(), 1E9)
prv_cost_func['t'] = 0

# solve the Bellman equation using value iteration
flag = True
while flag:
    cost_func = {}
    for state in states.values():
        cost_func[state.name] = state.get_min_exp_cost(prv_cost_func)[0]
    if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
        flag = False
    prv_cost_func = cost_func
print(cost_func)

# ------------------------------ Policy Iteration -----------------------------
print('{} Policy Iteration {}'.format(31*'-', 31*'-'))
# create states and cost func
policy = {}
for state in states.keys():
    policy[state] = list(states[state].actions)[0]
prv_cost_func = dict.fromkeys(states.keys(), 1E9)
prv_cost_func['t'] = 0

p_flag = True
while p_flag:
    # Policy evaluation: Find value of current policy by use of value iteration
    flag = True
    while flag:
        cost_func = {}
        for state in states.values():
            cost_func[state.name] = state.get_policy_exp_cost(policy,
                                                              prv_cost_func)
        if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
            flag = False
        prv_cost_func = cost_func
    # Policy improvement:
    p_flag = False
    for state in states.values():
        prv = policy[state.name]
        policy[state.name] = state.get_min_exp_cost(cost_func)[1]
        if policy[state.name] != prv:
            p_flag = True
print(cost_func)
print(policy)
```

Listing A.4: ssp_rl_algs.py; Implementation of RL algorithms used to solve SSP problem in Example 6.3.1

```
from ssp_ex import max_cost_funcs_diff, get_rng, get_environment
import re
import os, sys
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerTuple
```

```
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from plot_palette import ito_cmap

def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w

def run_sarsa(states: dict, Q_table: dict, rng, alpha: float=0.1,
                   gamma: float=1, epsilon: float=0, training_flag: bool=False):
    cum_cost = 0
    curr_state = states['A']
    update_flag = False
    while True:
        rand = rng.random()
        if rand < epsilon and training_flag:
            action = rng.choice(list(Q_table[curr_state.name].keys()))
        else:
            pos_actions = []
            min_val = np.inf
            for (k, v) in Q_table[curr_state.name].items():
                if v < min_val:
                    min_val = v
                    pos_actions = [k]
                elif v == min_val:
                    pos_actions.append(k)
            action = rng.choice(pos_actions)
        nxt_state = states[curr_state.sim(action)]
        state_action_data = curr_state.action_probs[action]
        idx = state_action_data['states'].index(nxt_state.name)
        cum_cost += state_action_data['costs'][idx]
        # Update Q-value and cum_sum from second action
        if training_flag and update_flag:
            state_action_data = prv_state.action_probs[prv_action]
            idx = state_action_data['states'].index(curr_state.name)
            Q_table[prv_state.name][prv_action] += alpha \
                *(state_action_data['costs'][idx] \
                  + gamma*Q_table[curr_state.name][action] \
                  - Q_table[prv_state.name][prv_action])
        prv_state = curr_state
        prv_action = action
        curr_state = nxt_state
        update_flag = True
        if curr_state.name == 't':
            if training_flag:
                state_action_data = prv_state.action_probs[prv_action]
                idx = state_action_data['states'].index(curr_state.name)
                Q_table[prv_state.name][prv_action] += alpha \
                    *(state_action_data['costs'][idx] \
                      + gamma*Q_table[curr_state.name]['terminate'] \
                      - Q_table[prv_state.name][prv_action])
            return cum_cost, Q_table

def run_q_learning(states: dict, Q_table: dict, rng, alpha: float=0.1,
                   gamma: float=1, epsilon: float=0, training_flag: bool=False):
    cum_cost = 0
    curr_state = states['A']
    while True:
        rand = rng.random()
        if rand < epsilon and training_flag:
            action = rng.choice(list(Q_table[curr_state.name].keys()))
        else:
            pos_actions = []
            min_val = np.inf
```

```
            for (k, v) in Q_table[curr_state.name].items():
                if v < min_val:
                    min_val = v
                    pos_actions = [k]
                elif v == min_val:
                    pos_actions.append(k)
            action = rng.choice(pos_actions)
        nxt_state = states[curr_state.sim(action)]
        # Update Q-value and cum_sum
        state_action_data = curr_state.action_probs[action]
        idx = state_action_data['states'].index(nxt_state.name)
        cum_cost += state_action_data['costs'][idx]
        if training_flag:
            Q_table[curr_state.name][action] += alpha \
                *(state_action_data['costs'][idx] \
                  + gamma*min(Q_table[nxt_state.name].values()) \
                  - Q_table[curr_state.name][action])
        curr_state = nxt_state
        if curr_state.name == 't':
            return cum_cost, Q_table

def run_experiments(states: dict, N: int, rng, run_func, name: str,
                    theoretical_q: dict, alpha: float=0.1,
                    gamma: float=1, epsilon: float=0.1, n_test: int=5,
                    ret_q: bool=False, test: bool=True, plot: bool=True):
    q_data = []
    cum_costs = []
    costs = {}
    Q_table = {}
    for n in states.keys():
        Q_table[n] = dict.fromkeys(states[n].actions, 0)

    training_flag = True
    for i in range(2*N):
        if i == N:
            if not test:
                break
            epsilon = 0
            training_flag = False
            costs['Training'] = cum_costs.copy()
            cum_costs = []
        if i >= 0.5*N:
            alpha *= 0.99
        cost, Q_table = run_func(states, Q_table, rng, alpha, gamma,
                                 epsilon, training_flag)
        cum_costs.append(cost)
        if training_flag:
            q_data.append([Q_table[s][a] for s in states.keys()
                           for a in states[s].actions])
        if i%(round(N/n_test)) == 0 and i != 0 and training_flag:
            curr_policy_cost = []
            for j in range(N):
                cost, Q_table = run_func(states, Q_table, rng)
                curr_policy_cost.append(cost)
            costs['{} iterations'.format(i)] = curr_policy_cost

    costs['Testing'] = cum_costs
    costs = pd.DataFrame(costs)
    if plot:
        print(Q_table)
        plot_data(states, N, name, costs, q_data, theoretical_q)
    if ret_q:
```

```python
        return Q_table

def plot_data(states: dict, N: int, name: str, costs: dict, q_data,
              theoretical_q: dict):
    path_name = '_'.join(re.split(' |-', name)).lower().replace('$', '')
    fig, axs = plt.subplots(2, 2)
    n = int(9*N/10)
    idx = 0
    for i, ax in enumerate(axs.flatten()):
        if i < len(states.keys())-1:
            p = []; m = []; t = []
            s = list(sorted(states.keys()))[i]
            for j, a in enumerate(states[s].actions):
                string = '{}{}'.format(s, a)
                data = [q_data[i][idx] for i in range(len(q_data))]
                p1, = ax.plot(data, c=ito_cmap.colors[j+1])
                p2 = ax.axhline(y=np.mean(data), xmin=0.045, xmax=0.965,
                                linestyle='--', c=ito_cmap.colors[j+1])
                p3 = ax.axhline(y=theoretical_q[string], xmin=0.045, xmax=0.965,
                                color=ito_cmap.colors[j+1], linestyle=':')
                p.append((p1, string)); m.append(p2); t.append(p3)
                idx += 1
            ax.set_title('Approximate Q-values state {}'.format(s))
            if len(states[s].actions) == 2:
                   ax.legend([p[0][0], p[1][0], (m[0], m[1]), (t[0], t[1])],
                             [p[0][1], p[1][1], 'Mean all runs', 'Actual'],
                             numpoints=1, ncol=2, loc='lower center', borderaxespad=0.,
                             handler_map={tuple: HandlerTuple(ndivide=None)})
        else:
            costs[['Training', 'Testing']].apply(lambda x:
                                                 moving_average(x, n)) \
                                          .plot(kind='line', ax=ax,
                                                color=ito_cmap.colors[1:3])
            ax.axhline(y=np.mean(costs['Testing']), xmin=0.045, xmax=0.965,
                       color='k', label='Mean testing', linestyle='--')
            ax.axhline(y=97/25, xmin=0.045, xmax=0.965, color='k',
                       linestyle=':', label='Optimal')
            ax.set_title(('Moving average (N={}) of costs\nincurred '
                          'during testing and training').format(n))
            ax.legend(ncol=2)
        ax.grid()
    fig.suptitle('{}'.format(name))
    fig.tight_layout()
    plt.savefig('../../figures/ssp_{}_{}runs.eps'.format(path_name, N))
    # Plot hists
    fig, axs = plt.subplots(2,2)
    idx = 0
    for i, ax in enumerate(axs.flatten()):
        if i < len(states.keys())-1:
            plot_data = {}
            s = list(sorted(states.keys()))[i]
            for j, a in enumerate(states[s].actions):
                d = [q_data[i][idx] for i in range(len(q_data))]
                key = '{}{}, std: {}'.format(s, a, round(np.std(d, ddof=1), 3))
                plot_data[key] = d
                idx += 1
            ax.grid()
            ax.set_title('Approximate Q-values state {}'.format(s))
            sns.histplot(data=pd.DataFrame(plot_data), ax=ax, kde=True,
                         palette=ito_cmap.colors[1:len(states[s].actions)+1])
            ax.set_ylim(0, 250)
        else:
```

```python
            ax.grid()
            ax.set_title('Total costs incurred from\ntesting and training runs')
            sns.histplot(data=costs[['Testing', 'Training']], ax=ax, kde=True,
                         palette=ito_cmap.colors[1:3])#len(costs.columns)+1])
            ax.set_ylim(0, 250)
    fig.suptitle('Histogram and KDE for {}'.format(name))
    fig.tight_layout()
    plt.savefig('../../figures/ssp_{}_hists_{}runs.eps'.format(path_name, N))

def find_policy_distr(allPol: dict, states: dict, runs: int, N: int, rng,
                      run_func, name: str, alpha: float=0.1, gamma: float=1,
                      epsilon: float=0.1):
    policies = {}
    for _ in range(runs):
        Q_table = run_experiments(states, N, rng, run_func, name, None, alpha,
                                  gamma, epsilon, ret_q=True, test=False,
                                  plot=False)
        policy = ''
        for state in Q_table.keys():
            if state != 't':
                opt_Q = np.inf
                opt_A = None
                for action in Q_table[state].keys():
                    if (Q_table[state][action] < opt_Q):
                        opt_Q = Q_table[state][action]
                        opt_A = action
                policy += '{}{}'.format(state, opt_A)
        if policy not in policies:
            policies[policy] = 1
        else:
            policies[policy] += 1
    allPol[name] = policies
    return allPol

def plot_policies_distr(policies: dict, runs: int):
    fig, ax = plt.subplots()
    df = pd.DataFrame(policies)
    df /= runs
    df.T.plot.bar(ax=ax, color=ito_cmap.colors[1:len(df.index)+1])
    ax.tick_params(axis='x', labelrotation=30)
    fig.canvas.draw()
    ax.set_title(('Distribution of policies found by the different methods'
                  '\nNumber of policies per method: {}').format(runs))
    ax.grid()
    fig.tight_layout()
    plt.savefig('../../figures/ssp_rl_policies.eps')

if __name__ == '__main__':
    rng = get_rng()
    N = int(1E3) # Number of training and test runs
    alpha = 0.3
    gamma = 1
    epsilon = 0.15
    states = get_environment()
    theoretical_q = {'A1':97/25, 'A2':97/25, 'B1':47/25,
                     'B2':2, 'C1':47/25}
    run_experiments(states, N, rng, run_q_learning, 'Q-learning',
                    theoretical_q, alpha, gamma, epsilon)
    run_experiments(states, N, rng, run_sarsa, 'SARSA',
                    theoretical_q, alpha, gamma, epsilon)
    # Find policy distribution
    allPol = {}
```

```
runs = N
find_policy_distr(allPol, states, runs, N, rng, run_q_learning,
                  'Q-learning', alpha, gamma, epsilon)
find_policy_distr(allPol, states, runs, N, rng, run_q_learning,
                  'SARSA', alpha, gamma, epsilon)
plot_policies_distr(allPol, runs)
```

Listing A.5: ex_SSP_es.json; JSON-file defining SSP with error state implemented by Listing A.1


```
{
    "A": {
        "1": {
            "states": [
                "B",
                "e"
            ],
            "probs": [
                0.5,
                0.5
            ],
            "costs": [
                1,
                4
            ],
      "risk_cost": [
        0,
    0
      ]
        },
        "2": {
            "states": [
                "B"
            ],
            "probs": [
                1
            ],
            "costs": [
                2
            ],
      "risk_cost": [
        0
      ]
        }
    },
    "B": {
        "1": {
            "states": [
                "e",
                "t"
            ],
            "probs": [
                0.15,
                0.85
            ],
            "costs": [
                5,
                1
            ],
      "risk_cost": [
        0,
    0
```

```
  ]
    },
    "2": {
        "states": [
"B",
            "t"
        ],
        "probs": [
            0.01,
0.99
        ],
        "costs": [
1,
            2
        ],
  "risk_cost": [
    0,
0
  ]
    }
},
"t": {
    "terminate": {
        "states": [
            "eta"
        ],
        "probs": [
            1
        ],
        "costs": [
            0
        ],
  "risk_cost": [
    0
  ]
    }
},
"e": {
    "terminate": {
        "states": [
            "eta"
        ],
        "probs": [
            1
        ],
        "costs": [
            0
        ],
  "risk_cost": [
    1
  ]
    }
},
"eta": {
    "terminate": {
        "states": [
            "eta"
        ],
        "probs": [
            1
        ],
        "costs": [
```

```
                0
            ],
        "risk_cost": [
          0
        ]
          }
      }
}
```

Listing A.6: ssp_minimax.py; Code to run minimax to solve SSP problem in Example 7.2.18

```python
from ssp_ex import max_cost_funcs_diff, get_environment

print('\n{} Minimax {}'.format(35*'#', 36*'#'))
# declear variables and get states
eps = 1E-16
states = get_environment(0.90) # discount factor of 0.9

# ------------------------------ Value Iteration ------------------------------
print('{} Value Iteration {}'.format(31*'-', 32*'-'))
# create initial cost function
prv_cost_func = dict.fromkeys(states.keys(), 1E9)
prv_cost_func['t'] = 0

# solve the Bellman equation using value iteration
flag = True
while flag:
    cost_func = {}
    for state in states.values():
        cost_func[state.name] = state.get_opt_minimax_cost(prv_cost_func)[0]
    if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
        flag = False
    prv_cost_func = cost_func
print(cost_func)

# ------------------------------ Policy Iteration ------------------------------
print('{} Policy Iteration {}'.format(31*'-', 31*'-'))
# create initial policy and cost function
policy = {}
for state in states.keys():
    policy[state] = list(states[state].actions)[0]
prv_cost_func = dict.fromkeys(states.keys(), 1E9)
prv_cost_func['t'] = 0
prv_act_cost = dict.fromkeys(states.keys(), 1E9)
prv_act_cost['t'] = 0

p_flag = True
while p_flag:
    # Policy evaluation: Find value of current policy by use of value iteration
    flag = True
    flag_act = True
    while flag or flag_act:
        cost_func = {}
        act_cost = {}
        for state in states.values():
            cost_func[state.name] = state.get_policy_minimax_cost(policy,
                                                                   prv_cost_func)
            res_act = state.get_policy_exp_cost(policy, prv_act_cost)
            act_cost[state.name] = res_act
        if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
            flag = False
```

```
        if max_cost_funcs_diff(act_cost, prv_act_cost) < eps:
            flag_act = False
        prv_cost_func = cost_func
        prv_act_cost = act_cost
    # Policy improvement:
    p_flag = False
    for state in states.values():
        prv = policy[state.name]
        policy[state.name] = state.get_opt_minimax_cost(cost_func)[1]
        if policy[state.name] != prv:
            p_flag = True
print(cost_func)
print(policy)
print(act_cost)
```

Listing A.7: ssp_exp_utility.py; Code to run VI and PI using exponential utility function to solve SSP problem in Example 7.2.18

```
from ssp_ex import max_cost_funcs_diff, get_environment

print('\n{} Exponetial utility {}'.format(30*'#', 30*'#'))
# declear variables and get states
eps = 1E-16
rs_f = -1
states = get_environment() # discount factor of 0.9

# ------------------------------ Value Iteration ------------------------------
print('{} Value Iteration {}'.format(31*'-', 32*'-'))
# create initial cost function
prv_cost_func = dict.fromkeys(states.keys(), 1)
prv_cost_func['t'] = 0

# solve the Bellman equation using value iteration
flag = True
while flag:
    cost_func = {}
    for state in states.values():
        cost_func[state.name] = state.get_min_exp_util_cost(prv_cost_func,
                                                             rs_f)[0]
    if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
        flag = False
    prv_cost_func = cost_func
print(cost_func)

# ------------------------------ Policy Iteration -----------------------------
print('{} Policy Iteration {}'.format(31*'-', 31*'-'))
# create initial policy and cost function
policy = {}
for state in states.keys():
    policy[state] = list(states[state].actions)[0]
prv_cost_func = dict.fromkeys(states.keys(), 1)
prv_cost_func['t'] = 0
prv_act_cost = dict.fromkeys(states.keys(), 1)
prv_act_cost['t'] = 0

p_flag = True
while p_flag:
    # Policy evaluation: Find value of current policy by use of value iteration
    flag = True
    while flag:
        cost_func = {}
        act_cost = {}
```

```
        for state in states.values():
            cost_func[state.name] = state.get_policy_exp_util_cost(policy,
                                                                   prv_cost_func,
                                                                   rs_f)
            res_act = state.get_policy_exp_cost(policy, prv_act_cost)
            act_cost[state.name] = res_act
        if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
            flag = False
        if max_cost_funcs_diff(act_cost, prv_act_cost) < eps:
            flag_act = False
        prv_cost_func = cost_func
        prv_act_cost = act_cost
    # Policy improvement:
    p_flag = False
    for state in states.values():
        prv = policy[state.name]
        policy[state.name] = state.get_min_exp_util_cost(cost_func,
                                                         rs_f)[1]
        if policy[state.name] != prv:
            p_flag = True
print(cost_func)
print(policy)
print(act_cost)
```

Listing A.8: ssp_error_states.py; Code to run VI and PI using error states notion of risk to solve SSP problem in Example 7.2.18

```
from ssp_ex import max_cost_funcs_diff, get_environment
import numpy as np

print('\n{} Error states {}'.format(30*'#', 30*'#'))
# declear variables and get states
eps = 1E-16
states = get_environment(path='ex_SSP_es.json')
a=0; b=1; n=5
omegas = [0.1, 0.2]

# ------------------------------ Value Iteration -------------------------------
print('{} Value Iteration {}'.format(31*'-', 32*'-'))
for omega in omegas:
    print('Omega: {}'.format(omega))
    # create initial cost function
    prv_cost_func = dict.fromkeys(states.keys(), 0)
    prv_risk_func = dict.fromkeys(states.keys(), 0)
    break_f = False
    # solve the Bellman equation using value iteration
    for i, xi in enumerate(np.linspace(a, b, n)):
        xi = round(xi, 3)
        flag = True
        while flag:
            cost_func = {}
            risk_func = {}
            for state in states.values():
                res = state.get_min_es_cost(prv_cost_func, prv_risk_func, xi)
                cost_func[state.name] = res[0]
                risk_func[state.name] = res[2]
            if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
                flag = False
            prv_cost_func = cost_func
            prv_risk_func = risk_func
        for k in cost_func:
            cost_func[k] = round(cost_func[k], 3)
```

```
            risk_func[k] = round(risk_func[k], 3)
            if k not in ['t', 'e', 'eta'] and risk_func[k] > omega:
                print(('Current cost function can only be achived by a to '
                       'risky policy! Breaking with xi={}').format(xi))
                break_f = True
                break
        print('Optimization criterion: {}'.format(cost_func))
        print('Risk function: {}'.format(risk_func))
        if break_f:
            break
    print()

# ------------------------------ Policy Iteration ------------------------------
print('{} Policy Iteration {}'.format(31*'-', 31*'-'))
for omega in omegas:
    print('Omega: {}'.format(omega))
    break_f = False
    for xi in np.linspace(a, b, n):
        xi = round(xi, 3)
        # create initial policy and cost function
        policy = {}
        for state in states.keys():
            policy[state] = list(states[state].actions)[0]
        prv_cost_func = dict.fromkeys(states.keys(), 0)
        prv_risk_func = dict.fromkeys(states.keys(), 0)
        prv_act_cost = dict.fromkeys(states.keys(), 0)
        p_flag = True
        while p_flag:
        # Policy evaluation: Find value of current policy by use of value iteration
            flag = True
            flag_act = True
            while flag or flag_act:
                cost_func = {}
                risk_func = {}
                act_cost = {}
                for state in states.values():
                    res = state.get_policy_es_cost(policy, prv_cost_func,
                                                   prv_risk_func, xi)
                    cost_func[state.name] = res[0]
                    risk_func[state.name] = res[1]
                    res_act = state.get_policy_exp_cost(policy, prv_act_cost)
                    act_cost[state.name] = res_act
                if max_cost_funcs_diff(cost_func, prv_cost_func) < eps:
                    flag = False
                if max_cost_funcs_diff(act_cost, prv_act_cost) < eps:
                    flag_act = False
                prv_cost_func = cost_func
                prv_risk_func = risk_func
                prv_act_cost = act_cost
            # Policy improvement:
            p_flag = False
            for state in states.values():
                prv = policy[state.name]
                policy[state.name] = state.get_min_es_cost(cost_func, risk_func, xi)[1]
                if policy[state.name] != prv:
                    p_flag = True
        for k in cost_func:
            cost_func[k] = round(cost_func[k], 3)
            risk_func[k] = round(risk_func[k], 3)
            if k not in ['t', 'e', 'eta'] and risk_func[k] > omega and not break_f:
                print(('Current cost function can only be achived by a to '
                       'risky policy! Breaking with xi={}').format(xi))
```

```
                break_f = True
        print('Optimization criterion: {}'.format(cost_func))
        print('Risk function: {}'.format(risk_func))
        print('Actuall cost of policy: {}'.format(act_cost))
        print('Policy: {}'.format(policy))
        if break_f:
            break
    print()
```

Listing A.9: ssp_rs_rl.py; Implementation of risk-sensitive RL algorithms used to solve SSP problem in Example 7.3.5

```
from ssp_ex import max_cost_funcs_diff, get_rng, get_environment
import re
import copy
import json
import os, sys
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerTuple
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from plot_palette import ito_cmap
matplotlib.rcParams['text.usetex'] = True

def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w

def run_q_hat_learning(states: dict, Q_table: dict, rng, alpha: float=0.1,
                       gamma: float=1, epsilon: float=0,
                       training_flag: bool=False):
    cum_cost = 0
    curr_state = states['A']
    while True:
        rand = rng.random()
        if rand < epsilon and training_flag:
            action = rng.choice(list(Q_table[curr_state.name].keys()))
        else:
            pos_actions = []
            min_val = np.inf
            for (k, v) in Q_table[curr_state.name].items():
                if v < min_val:
                    min_val = v
                    pos_actions = [k]
                elif v == min_val:
                    pos_actions.append(k)
            action = rng.choice(pos_actions)
        nxt_state = states[curr_state.sim(action)]
        # Update Q-value and cum_sum
        state_action_data = curr_state.action_probs[action]
        idx = state_action_data['states'].index(nxt_state.name)
        cum_cost += state_action_data['costs'][idx]
        if training_flag:
            a = Q_table[curr_state.name][action] + alpha \
                *(state_action_data['costs'][idx] \
                 + gamma*min(Q_table[nxt_state.name].values()) \
                 - Q_table[curr_state.name][action])
            b = Q_table[curr_state.name][action]
            Q_table[curr_state.name][action] = max(a, b)
        curr_state = nxt_state
```

```
        if curr_state.name == 't':
            return cum_cost, Q_table

def run_overweighting(states: dict, Q_table: dict, rng, alpha: float=0.1,
                      gamma: float=1, epsilon: float=0,
                      training_flag: bool=False, kappa: float=0.99):
    def chi(x: float, kappa: float):
        return (1-kappa)*x if x > 0 else (1+kappa)*x

    cum_cost = 0
    curr_state = states['A']
    while True:
        rand = rng.random()
        if rand < epsilon and training_flag:
            action = rng.choice(list(Q_table[curr_state.name].keys()))
        else:
            pos_actions = []
            min_val = np.inf
            for (k, v) in Q_table[curr_state.name].items():
                if v < min_val:
                    min_val = v
                    pos_actions = [k]
                elif v == min_val:
                    pos_actions.append(k)
            action = rng.choice(pos_actions)
        nxt_state = states[curr_state.sim(action)]
        # Update Q-value and cum_sum
        state_action_data = curr_state.action_probs[action]
        idx = state_action_data['states'].index(nxt_state.name)
        cum_cost += state_action_data['costs'][idx]
        if training_flag:
            Q_table[curr_state.name][action] += alpha \
                *chi(state_action_data['costs'][idx] \
                     + gamma*min(Q_table[nxt_state.name].values()) \
                     - Q_table[curr_state.name][action], kappa)
        curr_state = nxt_state
        if curr_state.name == 't':
            return cum_cost, Q_table

def run_error_states(states: dict, Q_table: dict, Q_bar_table: dict,
                     Q_xi_table: dict, rng, alpha: float=0.1, gamma: float=1,
                     epsilon: float=0, training_flag: bool=False, xi: float=0):
    cum_cost = 0
    cum_risk = 0
    curr_state = states['A']
    while True:
        rand = rng.random()
        if rand < epsilon and training_flag:
            action = rng.choice(list(Q_table[curr_state.name].keys()))
        else:
            pos_actions = []
            min_val = np.inf
            for (k, v) in Q_xi_table[curr_state.name].items():
                if v < min_val:
                    min_val = v
                    pos_actions = [k]
                elif v == min_val:
                    pos_actions.append(k)
            action = rng.choice(pos_actions)
        nxt_state = states[curr_state.sim(action)]
        # Update Q-value and cum_sum
        state_action_data = curr_state.action_probs[action]
```

```
        idx = state_action_data['states'].index(nxt_state.name)
        cum_cost += state_action_data['costs'][idx]
        cum_risk += state_action_data['risk_cost'][idx]
        if training_flag:
            pos_greedy_actions = []
            min_val = np.inf
            for (k, v) in Q_xi_table[nxt_state.name].items():
                if v < min_val:
                    min_val = v
                    pos_greedy_actions = [k]
                elif v == min_val:
                    pos_greedy_actions.append(k)
            greedy_action = rng.choice(pos_greedy_actions)
            cs_n = curr_state.name
            Q_table[cs_n][action] += alpha \
                *(state_action_data['costs'][idx] \
                  + gamma*Q_table[nxt_state.name][greedy_action] \
                  - Q_table[cs_n][action])
            Q_bar_table[cs_n][action] += alpha \
                *(state_action_data['risk_cost'][idx] \
                  + gamma*Q_bar_table[nxt_state.name][greedy_action] \
                  - Q_bar_table[cs_n][action])
            Q_xi_table[cs_n][action] = xi*Q_table[cs_n][action] \
                + Q_bar_table[cs_n][action]
        curr_state = nxt_state
        if curr_state.name == 'eta':
            return cum_cost, cum_risk, Q_table, Q_bar_table, Q_xi_table

def run_experiments(states: dict, N: int, rng, run_func, name: str,
                    alpha: float=0.1, gamma: float=1, epsilon: float=0.1,
                    n_test: int=5, kappa = None, ret_q: bool=False,
                    test: bool=True, plot: bool=True):
    q_data = []
    cum_costs = []
    costs = {}
    Q_table = {}
    for n in states.keys():
        Q_table[n] = dict.fromkeys(states[n].actions, 0)

    training_flag = True
    for i in range(2*N):
        if i == N:
            if not test:
                break
            epsilon = 0
            training_flag = False
            costs['Training'] = cum_costs.copy()
            cum_costs = []
        if i >= 0.5*N:
            alpha *= 0.99
        if kappa is not None:
            cost, Q_table = run_func(states, Q_table, rng, alpha, gamma,
                                     epsilon, training_flag, kappa=kappa)
        else:
            cost, Q_table = run_func(states, Q_table, rng, alpha, gamma,
                                     epsilon, training_flag)
        cum_costs.append(cost)
        if training_flag:
            q_data.append([Q_table[s][a] for s in states.keys()
                           for a in states[s].actions])
        if i%(round(N/n_test)) == 0 and i != 0 and training_flag:
            curr_policy_cost = []
```

```
            for j in range(N):
                cost, Q_table = run_func(states, Q_table, rng)
                curr_policy_cost.append(cost)
            costs['{} iterations'.format(i)] = curr_policy_cost
    costs['Testing'] = cum_costs
    costs = pd.DataFrame(costs)
    path = None
    if kappa is not None:
        path = name + ' kappa {}'.format(kappa)
        name = r'{}  $\kappa={}$'.format(name, kappa)
        if plot:
            print('Value of kappa: {}'.format(kappa))
    if plot:
        print(Q_table)
        plot_data(states, N, name, costs, q_data, path=path)
    if ret_q:
        return Q_table

def run_experiments_es(states: dict, N: int, rng, run_func, name: str,
                       omega: float, n_xi: int, alpha: float=0.1,
                       gamma: float=1, epsilon: float=0.1, n_test: int=5,
                       ret_q: bool=False, test: bool=True, plot: bool=True):
    alpha_ = alpha
    epsilon_ = epsilon
    Q_table = {}
    Q_bar_table = {}
    Q_xi_table = {}
    Q_prv = Q_xi_table
    for n in states.keys():
        Q_table[n] = dict.fromkeys(states[n].actions, 0)
        Q_bar_table[n] = dict.fromkeys(states[n].actions, 0)
        Q_xi_table[n] = dict.fromkeys(states[n].actions, 0)
    for xi in np.linspace(0, 1, n_xi):
        q_data = []
        cum_costs = []
        cum_risks = []
        costs = {}
        alpha = alpha_
        epsilon = epsilon_
        training_flag = True
        for i in range(2*N):
            if i == N:
                if not test:
                    break
                epsilon = 0
                training_flag = False
                costs['Training'] = cum_costs.copy()
                costs['Training risk'] = cum_risks.copy()
                cum_costs = []
                cum_risks = []
            if i >= 0.5*N:
                alpha *= 0.99
            cost, risk, Q_table, Q_bar_table, Q_xi_table = run_func(states,
                                                                    Q_table,
                                                                    Q_bar_table,
                                                                    Q_xi_table,
                                                                    rng,
                                                                    alpha,
                                                                    gamma,
                                                                    epsilon,
                                                                    training_flag,
                                                                    xi)
```

```
            cum_costs.append(cost)
            cum_risks.append(risk)
            if training_flag:
                q_data.append([Q_xi_table[s][a] for s in states.keys()
                               for a in states[s].actions])
            if i%(round(N/n_test)) == 0 and i != 0 and training_flag:
                curr_policy_cost = []
                curr_policy_risk = []
                for j in range(N):
                    cost, risk, Q_table, Q_bar_table, Q_xi_table=run_func(states,
                                                                        Q_table,
                                                                        Q_bar_table,
                                                                        Q_xi_table,
                                                                        rng,
                                                                        xi=xi)
                    curr_policy_cost.append(cost)
                    curr_policy_risk.append(risk)
                costs['{} iterations'.format(i)] = curr_policy_cost
                costs['{} iterations risk'.format(i)] = curr_policy_risk
        costs['Testing'] = cum_costs
        costs['Testing risk'] = cum_risks
        if np.mean(cum_risks) > omega:
            xi -= 1/(n_xi-1)
            xi = round(xi, 3)
            break
        costs_df = pd.DataFrame(costs)
        q_data_curr = q_data
        Q_prv = copy.deepcopy(Q_xi_table)
    if plot:
        print('Value of xi when breaking with omega={}: {}'.format(omega, xi))
        print(Q_xi_table)
        plot_data(states, N, r'{} $\xi = {}, \omega = {}$'.format(name, round(xi, 3), omega),
                  costs_df, q_data_curr, 3, '{}_omega_{}'.format(name, omega))
    if ret_q:
        return Q_prv, xi

def plot_data(states: dict, N: int, name: str, costs: dict, q_data,
              non_plotted_states: int = 1, path: str = None):
    if path is not None:
        path_name = '_'.join(re.split(' |-', path.replace('.', '_'))).lower()
    else:
        path_name = '_'.join(re.split(' |-', name.replace('.', '_'))).lower()
    fig, axs = plt.subplots(2, 2)
    n = int(9*N/10)
    idx = 0
    for i, ax in enumerate(axs.flatten()):
        if i < len(states.keys()) - non_plotted_states:
            p = []; m = []; t = []
            s = list(sorted(states.keys()))[i]
            for j, a in enumerate(states[s].actions):
                string = '{}{}'.format(s, a)
                data = [q_data[i][idx] for i in range(len(q_data))]
                p1, = ax.plot(data, c=ito_cmap.colors[j+1])
                p2 = ax.axhline(y=np.mean(data), xmin=0.045, xmax=0.965,
                                linestyle='--', c=ito_cmap.colors[j+1])
                p.append((p1, string)); m.append(p2);
                idx += 1
            ax.set_title('Approximate $Q$-values state {}'.format(s))
            if len(states[s].actions) == 2:
                ax.legend([p[0][0], p[1][0], (m[0], m[1])],
                          [p[0][1], p[1][1], 'Mean all runs'],
                          numpoints=1, #ncol=3, #loc='lower center',
```

```
                            borderaxespad=0.,
                            handler_map={tuple: HandlerTuple(ndivide=None)})
            else:
                ax.legend([p[0][0], (m[0])], [p[0][1], 'Mean all runs'],
                            numpoints=2, #ncol=2,
                            borderaxespad=0.,
                            handler_map={tuple: HandlerTuple(ndivide=None)})
        elif i == 2:
            costs[['Training risk',
                   'Testing risk']].apply(lambda x:
                                          moving_average(x, n)) \
                                          .plot(kind='line', ax=ax,
                                                color=ito_cmap.colors[1:3])
            ax.axhline(y=np.mean(costs['Testing risk']), xmin=0.045, xmax=0.965,
                       color='k', label='Mean testing', linestyle='--')
            ax.set_title(('Moving average (N={}) of\n '
                          'risk for testing and training').format(n))
            ax.legend()
        else:
            costs[['Training', 'Testing']].apply(lambda x:
                                                 moving_average(x, n)) \
                                          .plot(kind='line', ax=ax,
                                                color=ito_cmap.colors[1:3])
            ax.axhline(y=np.mean(costs['Testing']), xmin=0.045, xmax=0.965,
                       color='k', label='Mean testing', linestyle='--')
            ax.set_title(('Moving average (N={}) of costs\nincurred '
                          'during testing and training').format(n))
            ax.legend()#ncol=2)
        ax.grid()
        fig.suptitle(r'{}'.format(name.replace('Q-hat ', r'$\hat Q$-')))
    fig.tight_layout(rect=[0.125, 0.1, 0.875, 1])
    plt.savefig('../../figures/ssp_{}_{}runs.eps'.format(path_name, N), bbox_inches='tight')
    # Plot hists
    fig, axs = plt.subplots(2,2)
    idx = 0
    for i, ax in enumerate(axs.flatten()):
        if i < len(states.keys()) - non_plotted_states:
            plot_data = {}
            s = list(sorted(states.keys()))[i]
            for j, a in enumerate(states[s].actions):
                d = [q_data[i][idx] for i in range(len(q_data))]
                key = '{}{}, std: {}'.format(s, a, round(np.std(d, ddof=1), 3))
                plot_data[key] = d
                idx += 1
            ax.grid()
            ax.set_title('Approximate $Q$-values state {}'.format(s))
            sns.histplot(data=pd.DataFrame(plot_data), ax=ax, kde=True,
                         palette=ito_cmap.colors[1:len(states[s].actions)+1])
            ax.set_ylim(0, 250)
        elif i == 2:
            ax.grid()
            ax.set_title('Risk incurred from\ntesting and training runs')
            sns.histplot(data=costs[['Training risk', 'Testing risk']], ax=ax, kde=True,
                         palette=ito_cmap.colors[1:3])
            ax.set_ylim(0, 250)
        else:
            plot_data = {}
            for col in ['Training', 'Testing']:
                key = '{}, std:{}'.format(col, round(np.std(costs[col], ddof=1), 3))
                plot_data[key] = costs[col]
            ax.grid()
            ax.set_title('Total costs incurred from\ntesting and training runs')
```

```
            sns.histplot(data=plot_data, ax=ax, kde=True,
                         palette=ito_cmap.colors[1:3])
            ax.set_ylim(0, 250)
    fig.suptitle('Histogram and KDE for {}'.format(name.replace('Q-hat ', r'$\hat Q$-')))
    fig.tight_layout(rect=[0.125, 0.1, 0.875, 1])
    plt.savefig('../../figures/ssp_{}_hists_{}runs.eps'.format(path_name, N), bbox_inches='tight')

def find_policy_distr(allPol: dict, states: dict, runs: int, N: int, rng,
                      run_func, name: str, alpha: float=0.1, gamma: float=1,
                      epsilon: float=0.1, kappa=None):
    policies = {}
    for _ in range(runs):
        if kappa is None:
            Q_table = run_experiments(states, N, rng, run_func, name,
                                      alpha, gamma, epsilon, ret_q=True,
                                      test=False, plot=False)
        else:
            Q_table = run_experiments(states, N, rng, run_func, name,
                                      alpha, gamma, epsilon, kappa=kappa,
                                      ret_q=True, test=False, plot=False)
        policy = ''
        for state in Q_table.keys():
            if state != 't':
                opt_Q = np.inf
                opt_A = None
                for action in Q_table[state].keys():
                    if (Q_table[state][action] < opt_Q):
                        opt_Q = Q_table[state][action]
                        opt_A = action
                policy += '{}{}'.format(state, opt_A)
        if policy not in policies:
            policies[policy] = 1
        else:
            policies[policy] += 1
    allPol[name] = policies
    return allPol

def find_policy_distr_es(allPol: dict, states: dict, runs: int, N: int, rng,
                         run_func, name: str, omega: float, n_xi: int, xis: dict,
                         alpha: float=0.1, gamma: float=1, epsilon: float=0.1):
    policies = {}
    xis_loc = {}
    for _ in range(runs):
        Q_table, xi = run_experiments_es(states, N, rng, run_func, name, omega,
                                         n_xi, alpha, gamma, epsilon,
                                         ret_q=True, test=True, plot=False)
        policy = ''
        for state in Q_table.keys():
            if state not in ['t', 'e', 'eta']:
                opt_Q = np.inf
                opt_A = None
                for action in Q_table[state].keys():
                    if (Q_table[state][action] < opt_Q):
                        opt_Q = Q_table[state][action]
                        opt_A = action
                policy += '{}{}'.format(state, opt_A)
        if policy not in policies:
            policies[policy] = 1
        else:
            policies[policy] += 1
        if xi not in xis_loc:
            xis_loc[xi] = 1
```

```
        else:
            xis_loc[xi] += 1
    allPol[name] = policies
    xis[name] = xis_loc
    return allPol, xis

def plot_policies_distr(policies: dict, runs: int, xis: dict = None):
    fig, (ax1, ax2) = plt.subplots(2, 1)
    df = pd.DataFrame(policies)
    cols = list(df.columns)
    for i, col in enumerate(cols):
        cols[i] = r'{}'.format(col.replace('Q-hat ', '$\hat Q$-'))
    df.columns = cols
    df /= runs
    df.T.plot.bar(ax=ax1, color=ito_cmap.colors[1:len(df.index)+1])
    ax1.tick_params(axis='x', labelrotation=30)
    ax1.set_title(('Distribution of policies found by the different methods'
                   '\nNumber of policies per method: {}').format(runs))
    ax1.grid()
    ax1.legend(ncol=3, loc='upper center')
    df_xi = pd.DataFrame(xis)
    df_xi /= runs
    df_xi.T.plot.bar(ax=ax2, color=ito_cmap.colors[0:len(df_xi.index)])
    ax2.tick_params(axis='x', labelrotation=0)
    ax2.set_title(r'Distribution of $\xi$ values error state method used for final policy')
    ax2.grid()
    ax2.legend(ncol=5, loc='center')
    fig.tight_layout(rect=[0.125, 0.1, 0.875, 1])
    plt.savefig('../../figures/ssp_rs_rl_policies.eps', bbox_inches='tight')

if __name__ == '__main__':
    rng = get_rng()
    N = int(1E3) # Number of training and test runs
    alpha = 0.3
    gamma = 0.9
    epsilon = 0.15
    states = get_environment()
    run_experiments(states, N, rng, run_q_hat_learning, 'Q-hat learning',
                    alpha, gamma, epsilon)
    gamma = 1
    kappas = [0.5, 0.15, -0.5]
    for kappa in kappas:
        run_experiments(states, N, rng, run_overweighting,
                        'Weighting TDs with', alpha, gamma,
                        epsilon, kappa = kappa)
    states = get_environment(path='ex_SSP_es.json')
    omegas = [0.1, 0.125, 0.15, 0.175, 0.2]
    n_xi = 11
    for omega in omegas:
        run_experiments_es(states, N, rng, run_error_states, 'Error states',
                           omega, n_xi, alpha, gamma, epsilon, 5)
    # Find policy distribution
    allPol = {}
    runs = N
    states = get_environment()
    gamma = 0.9
    allPol = find_policy_distr(allPol, states, runs, N, rng, run_q_hat_learning,
                               'Q-hat learning', alpha, gamma, epsilon)
    print('Finished Q-hat')
    gamma = 1
    for k in kappas:
        allPol = find_policy_distr(allPol, states, runs, N, rng, run_overweighting,
```

```
                                   r'Weighting TDs with $\kappa = {}$'.format(k),
                                   alpha, gamma, epsilon, kappa = k)
        print('Finished weighting TDs with kappa = {}'.format(k))

    states = get_environment(path='ex_SSP_es.json')
    omegas = [0.1, 0.2]
    n_xi = 11
    xis = {}
    for o in omegas:
        allPol, xis = find_policy_distr_es(allPol, states, runs, N, rng,
                                           run_error_states,
                                           r'Error states, $\omega = {}$'.format(o),
                                           o, n_xi, xis, alpha, gamma, epsilon)
        print('Finished error states with omega = {}'.format(o))
    plot_policies_distr(allPol, runs, xis)
```

Listing A.10: exp_utility.py; Generating plots for Example 7.2.4

```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from plot_palette import ito_cmap
matplotlib.rcParams['text.usetex'] = True

k = np.linspace(0,9, 10000)
beta = np.linspace(-5, -1E-3, 1000)

def find_sum(k, beta, N=50):
    'Function approximating the sum of a target function using exp utility'
    s = 0
    for n in range(1, N):
        s += (1/(2**n))*np.exp(beta*(2**n - k))
    return 1/beta*np.log(s)

# Create subplots for multiple fixed values for k
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
ax1, ax2, ax3, ax4 = axs[0, 0], axs[0, 1], axs[1, 0], axs[1, 1]
for i, ax in zip([1, 2, 4, 8], [ax1, ax2, ax3, ax4]):
    y = [find_sum(i, b) for b in beta]
    ax.plot(beta, y, c=ito_cmap.colors[1])
    ax.hlines(0, beta[0], beta[-1], colors=ito_cmap.colors[2],
              linestyles='dashed')
    ax.set_title(r'$k = {}$'.format(i))
    ax.grid()

plt.suptitle('Target function value when playing and betting $k$ units for '
             'different values of $\\beta$')
lgd = fig.legend([r'$\displaystyle\frac{1}{\beta} \log \left'
                  r'(\sum_{n=1}^\infty\frac{1}{2^n} e^{\beta(2^n-k)}\right)$',
                  r'$y = 0$'],
                 loc='upper center', bbox_to_anchor=(0.5,0.1))
fig.text(0.5, 0.1, r'$\beta$', ha='center')
fig.text(0.1, 0.5, 'Target function', va='center', rotation='vertical')
fig.tight_layout(rect=[0.125, 0.1, 0.875, 1])
fig.savefig('../figures/exp_utility.eps', bbox_inches='tight')

# Create subplots for multiple fixed values for beta
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
ax1, ax2, ax3, ax4 = axs[0, 0], axs[0, 1], axs[1, 0], axs[1, 1]
for b, ax in zip([-0.01, -0.1, -1, -100], [ax1, ax2, ax3, ax4]):
    y = [find_sum(k, b) for k in k]
    y_abs = [abs(i) for i in y]
```

```
    min_indx = y_abs.index(min(y_abs))
    ax.plot(k, y, c=ito_cmap.colors[1])
    ax.hlines(0, k[0], k[-1], colors=ito_cmap.colors[2], linestyles='dashed')
    ax.scatter(k[min_indx], y[min_indx], color =ito_cmap.colors[0])
    ax.annotate(r'$({:g}, 0)$'.format(k[min_indx]),
                (k[min_indx]-0.5, y[min_indx]+1))
    ax.set_title(r'$\beta = {}$'.format(b))
    ax.grid()

plt.suptitle('Target function value when playing and betting different values '
             'of $k$ units for a given $\\beta$')
lgd = fig.legend([r'$\displaystyle\frac{1}{\beta} \log \left'
                  r'(\sum_{n=1}^\infty \frac{1}{2^n} e^{\beta(2^n-k)} \right )$',
                  r'$y = 0$'],
                 loc='upper center', bbox_to_anchor=(0.5,0.1))
fig.text(0.5, 0.1, r'$k$', ha='center')
fig.text(0.1, 0.5, 'Target function', va='center', rotation='vertical')
fig.tight_layout(rect=[0.125, 0.1, 0.875, 1])
fig.savefig('../figures/exp_utility_vary_k.eps', bbox_inches='tight')
```

Listing A.11: suboptimality_error_states.py; Generating plots for Example 7.2.17

```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from plot_palette import ito_cmap
matplotlib.rcParams['text.usetex'] = True

n = 100
xi = np.linspace(0, 1, n)
fig, ax = plt.subplots()
ax.plot(xi, 2*xi, label=r'$\mu_1$', c=ito_cmap.colors[1])
ax.plot(xi, xi+0.2, label=r'$\mu_2$ (optimal)', c=ito_cmap.colors[2])
ax.plot(xi, 0.21*np.ones(100), label=r'$\mu_3$ (unfeasible)',
        c=ito_cmap.colors[3], linestyle='--')
ax.vlines(0.21/2, 0, 2, 'k', linestyle=':')
ax.set_xlabel(r'$\xi$')
ax.set_ylabel(r'$\xi J_{\mu_i}(x) + \rho_{\mu_i}(x)$')
ax.legend()
ax.grid()
ax.set_title(r'Value of target function for differen values of $\xi$')
fig.tight_layout(rect=[0.125, 0.1, 0.875, 1])
fig.savefig('../figures/suboptimality_es.eps', bbox_inches='tight')
```

Bibliography

[Ach+17] Achiam, J. et al. “Constrained Policy Optimization”. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML’17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 22–31.

[Ber] Bertsekas, D. P. *Dynamic programming and optimal control. Vol. I. Selected Theoretical Problem Solutions.* Available at http://athenasc.com/DP_4thEd_theo_sol_Vol1.pdf (last updated 2.11.2017).

[Ber17] Bertsekas, D. P. *Dynamic programming and optimal control. Vol. I.* Fourth. Athena Scientific, Belmont, MA, 2017, pp. xix+555.

[Ber18] Bertsekas, D. P. *Abstract dynamic programming.* Athena Scientific Optimization and Computation Series. Second edition of [MR3204932]. Athena Scientific, Belmont, MA, 2018, pp. xiv+345.

[Ber19] Bertsekas, D. P. *Reinforcement learning and optimal control.* Athena Scientific Optimization and Computation Series. Athena Scientific, Belmont, MA, 2019, pp. xiv+373.

[Cho+17] Chow, Y. et al. “Risk-Constrained Reinforcement Learning with Percentile Risk Criteria”. In: *J. Mach. Learn. Res.* Vol. 18, no. 1 (Jan. 2017), pp. 6070–6120.

[Fol99] Folland, G. B. *Real analysis.* Second. Pure and Applied Mathematics (New York). Modern techniques and their applications, A Wiley-Interscience Publication. John Wiley & Sons, Inc., New York, 1999, pp. xvi+386.

[GFF15] García, J., Fern, and Fernández, o. “A Comprehensive Survey on Safe Reinforcement Learning”. In: *Journal of Machine Learning Research* vol. 16, no. 42 (2015), pp. 1437–1480.

[GW11] Geibel, P. and Wysotzki, F. “Risk-Sensitive Reinforcement Learning Applied to Control under Constraints”. In: *Journal of Artificial Intelligence Research - JAIR* vol. 24 (Sept. 2011).

[Heg94] Heger, M. “Consideration of risk in reinforcement learning”. In: *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 105–111.

[HM72] Howard, R. A. and Matheson, J. E. “Risk-sensitive Markov decision processes”. In: *Management science* vol. 18, no. 7 (1972), pp. 356–369.

[Lin17] Lindstrøm, T. L. *Spaces—an introduction to real analysis*. Vol. 29. Pure and Applied Undergraduate Texts. American Mathematical Society, Providence, RI, 2017, pp. xii+369.

[MN02] Mihatsch, O. and Neuneier, R. "Risk-Sensitive Reinforcement Learning". In: *Machine Learning* vol. 49, no. 2 (2002), pp. 267–290.

[Øks03] Øksendal, B. *Stochastic differential equations*. Sixth. Universitext. An introduction with applications. Springer-Verlag, Berlin, 2003, pp. xxiv+360.

[SB18] Sutton, R. S. and Barto, A. G. *Reinforcement learning: an introduction*. Second. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2018, pp. xxii+526.

[TDM12] Tamar, A., Di Castro, D., and Mannor, S. "Policy Gradients with Variance Related Risk Criteria". In: *Proceedings of the 29th International Coference on International Conference on Machine Learning*. ICML'12. Edinburgh, Scotland: Omnipress, 2012, pp. 1651–1658.

[Wal12] Walsh, J. B. *Knowing the odds*. Vol. 139. Graduate Studies in Mathematics. An introduction to probability. American Mathematical Society, Providence, RI, 2012, pp. xvi+421.