Institut für Mathematik
Naturwissenschaftliche Fakultät

Reinforcement Learning for Several Environments Theory and Applications

Dissertation
zur Erlangung des Doktorgrades
an der
Naturwissenschaftlichen Fakultät
der Leopold-Franzens-Universität
Innsbruck

vorgelegt von
Mag.rer.nat. Andreas Matt und Mag.rer.nat. Georg Regensburger

eingereicht bei
o.Univ.-Prof. Dr. Ulrich Oberst

Innsbruck, April 2004

This thesis is the result of a joint project on reinforcement learning that was initiated and carried out by Andreas Matt and Georg Regensburger. The main results were developed together. At the end of the thesis an explicit description of the contributions of each author is given.

Acknowledgements

We gratefully thank:

Our advisor Prof. Ulrich Oberst for his excellent critical comments and discussions, his perpetual support and help concerning any questions and matters and for being interested in reinforcement learning. Moreover, for his great effort and encouragement during the final phase.

Prof. Juan Miguel Santos for introducing us to reinforcement learning, for his constant support and for visiting us twice in Austria.

The Institute of Mathematics at the University of Innsbruck and its members for our mathematical education and the enormous institutional support, in particular for purchasing the mobile robot. Herwig Hauser for discussions and coffee, Franz Pauer for support and discussions on polytopes, Christoph Kollreider and Christian Kraler for encouragement and organizing our computers. Our colleagues and friends from the 7th and 8th floor.

The Institute of Computer Science at the University of Innsbruck and its members in particular Sybille Hellebrand for the integration of our project and for technical support. We enjoyed the enriching work as teaching assistants over the last years.

The University of Innsbruck for financial support. We received the scholarship "Forschungsstipendien an österreichische Graduierte" from April to September 2001 and the scholarship "Förderung für wissenschaftliche Arbeiten im Ausland" for Andreas Matt from October 2002 to January 2003. Several times we received grants from the "Studienbeiträge der Ausländer" to cover some of the travel costs and conference fees. We very much appreciate the honor of receiving the "Preis für junge Wissenschaftler" from the Faculty of Natural Sciences at the University of Innsbruck.

All people, institutions and companies who invited us to give talks and to present our research in public, in particular to fabula Europe for support in organizing several events.

André Mitterbacher and Martin Ruetz of the Institute of Applied Physics at the University of Innsbruck for technical first-aid for the mobile robot.

Sylvia Baumgartner for helping to proof-read the English of several parts of the thesis.

Andreas Matt additionally thanks:

The Departamento de Computación of the Facultad de Ciencias Exactas y Naturales at the University of Buenos Aires and its members for the kind hospitality during my visits, in particular Prof. Juan Santos for everything, Diego Bendersky for his excellent ideas, motivation and programming skills and Andrés Stoliar for his brilliant technical advice.

My coauthor and friend for his patience and accuracy. It was a big pleasure to work together from the spontaneous beginning to the hard working end. I hope that this fruitful co-operation will continue.

My grandfather Erwin Kathrein for the construction of the robot's *wooden box*. My mother and all friends who helped, encouraged and supported me in writing this work.

Georg Regensburger additionally thanks:

My coauthor and friend who asked me three and a half years ago to join him on a project for a few weeks. Evidently, the few weeks became a little longer and the result of our joint research is this thesis. I really enjoyed working together.

Prof. Otmar Scherzer and all my colleagues from his group.

My friends for their encouragement and interest.

My family for their love and support.

Dedicated to Verena.

Innsbruck, April 2004, Andreas Matt and Georg Regensburger

"A final goal of any scientific theory must be the derivation of numbers."
Richard Bellman [Bel84]

Contents

Introduction

Reinforcement learning addresses problems of sequential decision making and stochastic control and is strongly connected to dynamic programming and Markov decision processes. In the last two decades it has gained importance in the fields of machine learning and artificial intelligence where the term reinforcement learning was established. Researchers from a variety of scientific fields that reach from cognitive sciences, neurology and psychology to computer science, physics and mathematics, have developed algorithms and techniques with impressive applications as well as mathematical foundations.

Reinforcement learning is based on the simple idea of learning by trial and error while interacting with an environment. At each step the agent performs an action and receives a reward depending on the starting state, the action and the environment. The agent learns to choose actions that maximize the sum of all rewards in the long run. The resulting choice of an action for each state is called a policy. Finding optimal policies is the primary objective of reinforcement learning.

For a history of dynamic programming see the bibliographical remarks in Puterman [Put94], in particular the section on discounted Markov decision processes [Put94, pp. 263]. Refer to Sutton [SB98, pp. 16] for a historical overview of reinforcement learning and its first achievements.

Our Approach

Until now reinforcement learning has been applied to learn the optimal behavior for a single environment. The main idea of our approach is to extend reinforcement learning to learn a good policy for several environments simultaneously. We link the theory of Markov decision processes (MDP) with notions and algorithms from reinforcement learning to model this idea and to develop solution methods. We do not only focus on rigorous models and mathematical analysis, but also on applications ranging from exact computations to real world problems.

Combining theory and applications turned out to be very fruitful. Ideas

from experiments and programming influenced the theoretical development and vice versa.

The development of software forms a major part of our work. The programs and the source code can be found on the attached CD-ROM. All experiments can be reproduced following the descriptions given. This allows the reader to experience reinforcement learning in an interactive way.

Overview

The thesis is split into two parts which are similarly structured into theoretical sections, computational examples, simulation results and robot experiments.

In the first part we discuss reinforcement learning for one environment. The main results and algorithms for finite discounted MDPs are presented and extended. Stochastic policies often play an inferior role in the presentation of reinforcement learning since for one environment there always exists a deterministic optimal policy. For several environments good policies are not deterministic in general, and therefore we formulate the theory and algorithms in the first part for stochastic policies too. Furthermore, we use the notion of action-values in our presentation, which is common in the reinforcement learning literature but not in the literature on MDPs. We introduce a new geometric interpretation of policy improvement that is fundamental for the theory for several environments. Finally, model-free methods are presented and further bibliographical remarks given.

We describe the `MDP` package for the computer algebra system Maple and give computational examples. We introduce our grid world simulator `SimRobo`, describe implementation details and show results of a variety of experiments. The program `RealRobo` for the mobile robot Khepera is presented and several experiments are conducted.

In the second part we address reinforcement learning for several environments. First we introduce the notion of a state action space which allows us to apply one policy to several environments. Environments are called realizations in this context. Then we extend policy improvement for this model and introduce the notion of balanced policies. We discuss a geometric interpretation of improving policies for a possible infinite family of realizations using convex geometry. We give constructive methods to compute improving policies for finite families of realizations by means of polytopes and linear programming. This leads to policy iteration for several realizations. For the model-free case we finally introduce approximate policy iteration for several realizations.

We give an introduction to the publication [MR03b] where we discuss the idea of learning in one environment and applying the policy obtained to other environments.

The theoretical sections are followed by computational examples for two realizations with the `MDP` package. Then we describe the implementation and the functions of the simulator `SimRobo` for several realizations and discuss a series of experiments. Finally, we discuss an experiment where the robot Khepera learns two behavior simultaneously. We conclude with a discussion of the theory presented, possible applications to other fields and future work.

The appendix includes several listings with comments for the programs `SimRobo` and `RealRobo`, a description with examples of all functions in the `MDP` package and an overview of the contents of the attached CD-ROM.

Moreover details of the contributions of each author are presented. Three papers were published while working on this dissertation.

A summary of the main contributions of this thesis:

- Mathematical treatment of MDPs and reinforcement learning for stochastic policies.
- Characterization of equivalent policies and geometrical interpretation of improving policies and policy improvement using the theory of polytopes.
- State action spaces and realizations to apply one policy to several environments.
- Policy improvement and the notion of balanced policies for a family of realizations.
- Geometrical interpretation and computation of improving policies for a family of realizations.
- Policy iteration and approximate policy iteration for finite families of realizations.
- `MDP` package for exact and symbolic computations with MDPs and two realizations.
- Grid world simulator `SimRobo` for MDPs and several realizations.
- Program `RealRobo` for the mobile robot Khepera.

Part I

One Environment

Chapter 1

Reinforcement Learning

1.1 Markov Decision Processes

The term Markov decision process to identify problems of optimal control in dynamic systems was introduced by Bellman [Bel54, Bel57, Bel03]. In the following definitions we rely on the common language used in reinforcement learning. The main difference to the standard definition of Markov decision processes is the separation of the environment and the rewards.

An *environment* is given by

- A finite set S.
  The set S is interpreted as the set of all possible states. An element $s \in S$ is called a *state*.
- A family $\mathbf{A} = (A(s))_{s \in S}$ of finite sets.
  The set $A(s)$ is interpreted as the set of available actions in state s. An element $a \in A(s)$ is called an *action*.
- A family $\mathbf{P} = (P(- \mid a, s))_{s \in S, a \in A(s)}$ of probabilities $P(- \mid a, s)$ on S.
  We interpret $P(s' \mid a, s)$ as the *transition probability* that performing action a in state s leads to the *successor state* s'.

Let $E = (S, \mathbf{A}, \mathbf{P})$ be an environment. A *policy* for E is given by

- A family $\pi = (\pi(- \mid s))_{s \in S}$ of probabilities $\pi(- \mid s)$ on $A(s)$.
  We interpret $\pi(a \mid s)$ as the probability that action a is chosen in state s.

A policy is called *deterministic* if $\pi(- \mid s)$ is deterministic for all $s \in S$, that is, if for each state s a unique action $a \in A(s)$ is chosen with probability

one. We call a probability $\pi(- \mid s)$ on $A(s)$ a *policy in state* s. We identify an action $a \in A(s)$ with the deterministic policy $\pi(- \mid s)$ in s, defined by

$$\pi(\tilde{a} \mid s) = \begin{cases} \pi(\tilde{a} \mid s) = 1, & \text{if } \tilde{a} = a, \\ \pi(\tilde{a} \mid s) = 0, & \text{otherwise,} \end{cases}$$

and write

$$\pi(- \mid s) = a.$$

The set of all policies for an environment is denoted by Π.

Let π be a policy for E. Let s and $s' \in S$ be two states. We define the *transition probability* from state s to successor state s' for policy π by

$$P^{\pi}(s' \mid s) = \sum_{a \in A(s)} P(s' \mid a, s)\pi(a \mid s).$$

Then $P^{\pi}(- \mid s)$ is a probability on S, since

$$\begin{aligned} \sum_{s' \in S} P^{\pi}(s' \mid s) &= \sum_{s' \in S} \sum_{a \in A(s)} P(s' \mid a, s)\pi(a \mid s) \\ &= \sum_{a \in A(s)} \pi(a \mid s) \sum_{s' \in S} P(s' \mid a, s) = 1. \end{aligned}$$

A *finite Markov Decision Process (MDP)* is given by an environment $E = (S, \mathbf{A}, \mathbf{P})$ and

- A family $\mathbf{R} = (R(s', a, s))_{s', s \in S, a \in A(s)}$ with $R(s', a, s) \in \mathbb{R}$.
  The value $R(s', a, s)$ represents the *reward* if performing action a in state s leads to the successor state s'.

Let $(E, \mathbf{R})$ be an MDP. Let $s \in S$ and $a \in A(s)$. We define the *expected reward of action a in state s* by

$$R(a, s) = \sum_{s' \in S} R(s', a, s)P(s' \mid a, s).$$

Let π be a policy for E. Let $s \in S$. The *expected reward for policy π in state s* is defined by

$$R^{\pi}(s) = \sum_{a \in A(s)} R(a, s)\pi(a \mid s).$$

1.2 Value Functions and Action-Values

Suppose that, starting in a particular state, actions are taken following a fixed policy. Then the expected sum of rewards is called the value function. It is defined for all starting states and maps states to numerical values to classify policies as good or bad in the long run.

We first start building sequences of state-action pairs and define a probability distribution on them. Then we discuss the discounted value function and action-values.

1.2.1 Histories

Let E be an environment. Let

$$\mathbf{A}_S = \{(a, s) \mid s \in S, a \in A(s)\}$$

denote the set of allowed state-action pairs. Let $T \in \mathbb{N}_0$. The set of *histories* up to time T is defined by

$$H_T = S \times (\mathbf{A}_S)^T .$$

Then $H_0 = S$ and $H_1 = S \times \mathbf{A}_S$. Let $T \in \mathbb{N}$ and

$$h_T = (s^T, a^{T-1}, s^{T-1}, \ldots, a^0, s^0) \in H_T.$$

The vector h_T represents the trajectory starting from initial state s^0, performing a series of actions until finally getting to state s^T. We write

$$h_T = (s^T, a^{T-1}, h_{T-1}) \text{ with } h_{T-1} = (s^{T-1}, \ldots, a^0, s^0) \in H_{T-1}. \tag{1.1}$$

We define a probability on the set of histories given an initial distribution on all states and a fixed policy. Let μ be a probability on S. Let π be a policy for E. We set $\mathbb{P}_0^{\pi,\mu} = \mu$. Inductively we define a probability $\mathbb{P}_T^{\pi,\mu}$ on H_T by

$$\mathbb{P}_T^{\pi,\mu}(h_T) = P(s^T \mid a^{T-1}, s^{T-1})\pi(a^{T-1} \mid s^{T-1})\mathbb{P}_{T-1}^{\pi,\mu}(h_{T-1})$$

with h_T as in (1.1). The probability depends on the initial distribution μ and we write $\mathbb{P}_T^{\pi} = \mathbb{P}_T^{\pi,\mu}$.

By definition

$$\mathbb{P}_T^{\pi}(s^T \mid a^{T-1}, h_{T-1}) = \mathbb{P}_T^{\pi}(s^T \mid a^{T-1}, s^{T-1}) = P(s^T \mid a^{T-1}, s^{T-1})$$

using a simplified notation. This property is called the *Markov property*. It says that the conditional probability depends only on the last state and the chosen action and not on past states and actions respectively.

The Kolmogorov extension theorem ensures the existence of a unique probability $\mathbb{P}^\pi_\infty$ on the set $H_\infty = (\mathbf{A}_S)^\mathbb{N}$ of infinite histories with marginal probabilities $\mathbb{P}^\pi_T$, that is

$$\mathbb{P}^\pi_\infty \left\{ h = (\ldots, s^{T+1}, a^T, h_T) \in H_\infty \right\} = \mathbb{P}^\pi_T(h_T), \quad \text{for } T \in \mathbb{N} \text{ and } h_T \in H_T.$$

1.2.2 Returns and Value Functions

Let $(E, \mathbf{R})$ be an MDP. Let $T \in \mathbb{N}$ and

$$h_T = (s^T, a^{T-1}, s^{T-1}, \ldots, s^1, a^0, s^0) \in H_T$$

be a history up to time T. We define the *return* R_T of h_T by

$$R_T(h_T) = \sum_{t=0}^{T-1} R(s^{t+1}, a^t, s^t).$$

Let $\gamma \in \mathbb{R}$ with $0 \leq \gamma < 1$. We call γ a *discount rate*. We define the *discounted return* R^γ_T of h_T by

$$R^\gamma_T(h_T) = \sum_{t=0}^{T-1} \gamma^t R(s^{t+1}, a^t, s^t).$$

The discount rate controls the importance of future rewards. It also allows us to define the return for infinite histories $h \in H_\infty$ by

$$R^\gamma_\infty(h) = \sum_{t=0}^{\infty} \gamma^t R(s^{t+1}, a^t, s^t).$$

Let $h_T \in H_T$ such that $h = (..., h_T)$ for $T \in \mathbb{N}$. Then

$$R^\gamma_\infty(h) = \lim_{T \to \infty} R^\gamma_T(h_T). \tag{1.2}$$

By definition $R_0(h_0) = 0$ and $R^\gamma_0(h_0) = 0$ for $h_0 \in H_0 = S$.

Let π be a policy for E. Let μ be an initial distribution on S, $T \in \mathbb{N}$ and let $\mathbb{P}^\pi_T$ be the previously defined probability on the set of histories H_T. The expectation value $\mathbb{E}^\pi$ of the return R_T for $\mathbb{P}^\pi_T$ is

$$V^\pi_T(\mu) = \mathbb{E}^\pi(R_T) = \sum_{h_T \in H_T} R_T(h_T) \mathbb{P}^\pi_T(h_T).$$

Let $s \in S$. We denote by μ_s the initial distribution of the system to start in state s at time 0, that is

$$\mu_s(s') = \begin{cases} 1, & \text{if } s' = s, \\ 0, & \text{otherwise.} \end{cases}$$

The function V_T^π on S defined by

$$V_T^\pi(s) = V_T^\pi(\mu_s) = \mathbb{E}^\pi(R_T \mid s_0 = s), \quad \text{for } s \in S,$$

is called the *undiscounted value function* for policy π up to time T and $V_T^\pi(s)$ the *undiscounted value* or *utility* of state s for the policy π up to time T respectively. We have $V_0^\pi(s) = 0$ for $s \in S$.

Let γ be a discount rate. Analogously we consider the expectation value $\mathbb{E}^\pi$ of the discounted return

$$V_T^{\pi,\gamma}(\mu) = \mathbb{E}^\pi(R_T^\gamma) = \sum_{h_T \in H_T} R_T^\gamma(h_T)\mathbb{P}_T^\pi(h_T).$$

We call

$$V_T^{\pi,\gamma}(s) = V_T^{\pi,\gamma}(\mu_s) = \mathbb{E}^\pi(R_T^\gamma \mid s_0 = s) \tag{1.3}$$

discounted value or *utility* of state s for the policy π up to time T and $V_T^{\pi,\gamma}$ the *discounted value function* for policy π up to time T. Again we have $V_0^{\pi,\gamma}(s) = 0$ for $s \in S$.

1.2.3 Bellman Equation

In this subsection we derive the basic recursion for the value function for a policy. The idea is the following. Let $T \in \mathbb{N}$ and

$$h_T = (s^T, \ldots, a^1, s^1, a^0, s^0) \in H_T.$$

a history up to time T. Then its return can be split up as the immediate reward for the first state-action-state triple plus the return of the remaining trajectory

$$R_T(h_T) = R(s^1, a^0, s^0) + \sum_{t=1}^{T-1} R(s^{t+1}, a^t, s^t).$$

An elementary but lengthy computation gives the following theorem, see for example Dynkin and Yushkevich [DY79, pp. 19] or Matt [Mat00].

Theorem 1. *Let $(E, \mathbf{R})$ be an MDP. Let π be a policy for E. Let $s \in S$ and $T \in \mathbb{N}$. Then*

$$V_T^\pi(s) = R^\pi(s) + \sum_{s'} V_{T-1}^\pi(s')P^\pi(s' \mid s). \tag{1.4}$$

The theorem describes a recursive relation for the undiscounted value function for a policy up to time T given the expected reward for the policy and value function up to time $T-1$. Equation (1.4) is called the *Bellman equation* for V_T^π, Sutton and Barto [SB98, p. 70].

We use the following convention for vectors and matrices in $\mathbb{R}^S$ and $\mathbb{R}^{S\times S}$ to simplify the notation. Let

$$x = (x(s))_{s\in S} \in \mathbb{R}^S$$

be a row vector and

$$B = (B(s', s))_{s', s\in S} \in \mathbb{R}^{S\times S}$$

a matrix. The multiplication of x and B is defined as usual by

$$(xB)(s) = \sum_{s'} x(s')B(s', s), \quad \text{for } s \in S.$$

Note that we use only row vectors and multiplications of row vectors and matrices.

We consider $V_T^\pi = (V_T^\pi(s))_{s\in S}$ and $R_T^\pi = (R_T^\pi(s))_{s\in S}$ as row vectors in $\mathbb{R}^S$. We define the *transition matrix* $P^\pi \in \mathbb{R}^{S\times S}$ for policy π by

$$P^\pi(s', s) = P^\pi(s' \mid s), \quad \text{for } s', s \in S,$$

where $P^\pi(s' \mid s)$ denotes the transition probability from state s to s' of π. Since $P^\pi(- \mid s)$ is a probability on S, the matrix P^π is a *stochastic matrix*, that is, all entries are nonnegative and all columns sum up to one. This differs from the standard definition, where all rows sum up to one, because we consider row instead of column vectors. The previous theorem and induction imply the following assertion.

Theorem 2. *Let $(E, \mathbf{R})$ be an MDP. Let π be a policy for E and $T \in \mathbb{N}$. Then*

$$V_T^\pi = R^\pi + V_{T-1}^\pi P^\pi = R^\pi \sum_{t=0}^{T-1} (P^\pi)^t.$$

Let γ be a discount rate. The calculations are exactly the same for the discounted value function except that

$$R_T^\gamma(h_T) = R(s^1, a^0, s^0) + \gamma \sum_{t=1}^{T-1} \gamma^{t-1} R(s^{t+1}, a^t, s^t).$$

Theorem 1 then reads as follows.

Theorem 3. *Let $(E, \mathbf{R})$ be an MDP and $0 \leq \gamma < 1$ a discount rate. Let π be a policy for E. Let $s \in S$ and $T \in \mathbb{N}$. Then*

$$V_T^{\pi,\gamma}(s) = R^\pi(s) + \gamma \sum_{s'} V_{T-1}^{\pi,\gamma}(s') P^\pi(s' \mid s).$$

Using the vector notation and induction we obtain the following theorem.

Theorem 4. *Let $(E, \mathbf{R})$ be an MDP and $0 \leq \gamma < 1$ a discount rate. Let π be a policy for E and $T \in \mathbb{N}$. Then*

$$V_T^{\pi,\gamma} = R^\pi + \gamma V_{T-1}^{\pi,\gamma} P^\pi = R^\pi \sum_{t=0}^{T-1} (\gamma P^\pi)^t.$$

1.2.4 Discounted Value Function

In this section we discuss the discounted value function on infinite histories.

We use the *maximum norm* on $\mathbb{R}^S$, that is

$$\|x\|_\infty = \max_{s \in S} |x(s)|, \quad \text{for } x \in \mathbb{R}^S.$$

The associated *matrix norm* for the maximum norm is

$$\|B\|_\infty = \sup \left\{ \|xB\|_\infty : x \in \mathbb{R}^S, \|x\|_\infty = 1 \right\}, \quad \text{for } B \in \mathbb{R}^{S \times S}.$$

The definition implies that

$$\|xB\|_\infty \leq \|x\|_\infty \|B\|_\infty$$

for $x \in \mathbb{R}^S, B \in \mathbb{R}^{S \times S}$ and

$$\|B\|_\infty = \max_{s \in S} \sum_{s'} |B(s', s)|.$$

Let $I \in \mathbb{R}^{S \times S}$ denote the identity matrix. Obviously, a stochastic matrix P with nonnegative entries and whose columns add up to 1 has norm $\|P\|_\infty = 1$. Recall that the discount factor satisfies $0 \leq \gamma < 1$. Hence $(I - \gamma P^\pi)$ is an invertible matrix and its inverse is given by

$$(I - \gamma P^\pi)^{-1} = \sum_{k=0}^{\infty} (\gamma P^\pi)^k.$$

Therefore the limit $\lim_{T \to \infty} V_T^{\pi,\gamma}$ exists and we have

$$\lim_{T \to \infty} V_T^{\pi,\gamma} = R^\pi \sum_{t=0}^{\infty} (\gamma P^\pi)^t = R^\pi \left(I - \gamma P^\pi\right)^{-1}. \tag{1.5}$$

This allows us to define the *discounted value function* for policy π by

$$V^{\pi,\gamma} = \lim_{T\to\infty} V_T^{\pi,\gamma} = R^\pi \left(I - \gamma P^\pi\right)^{-1}. \tag{1.6}$$

We call $V^{\pi,\gamma}(s)$ the *discounted value* or *utility* of state s for policy π.

For a fixed discount rate γ we write $V^\pi = V^{\pi,\gamma}$ for the value function. We call $(E, \mathbf{R}, \gamma)$ a *(discounted) MDP*. From now on we consider only discounted MDPs and the discounted value function. The following theorem summarizes Equations (1.5) and (1.6) for the discounted value function.

Theorem 5. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π be a policy for E. Then*

$$V^\pi = R^\pi + \gamma V^\pi P^\pi \tag{1.7}$$

$$= R^\pi (I - \gamma P^\pi)^{-1} = R^\pi \sum_{t=0}^{\infty} (\gamma P^\pi)^t. \tag{1.8}$$

Equation (1.7) is called the *Bellman equation* for V^π. Equation (1.8) gives us a method to compute the value function by inverting the matrix $(I - \gamma P^\pi)$. Computing the value function for a given policy is often referred to as *policy evaluation*, see for example Puterman [Put94, pp. 143] or Sutton and Barto [SB98, p. 90]. Note that for policy evaluation only the transition matrix P^π and the expected rewards R^π for policy π are needed.

We define the affine map $T^\pi \colon \mathbb{R}^S \to \mathbb{R}^S$ for a policy π by

$$T^\pi(V) = R^\pi + \gamma V P^\pi. \tag{1.9}$$

Then T^π is a *contraction mapping* with respect to the maximum norm, since

$$\begin{aligned} \left\|T^\pi(V_1) - T^\pi(V_2)\right\|_\infty &= \gamma \left\|(V_1 - V_2)P^\pi\right\|_\infty \\ &\leq \gamma \left\|V_1 - V_2\right\|_\infty \left\|P^\pi\right\|_\infty = \gamma \left\|V_1 - V_2\right\|_\infty. \end{aligned}$$

The value function V^π is a *fixed point* of T^π by Equation (1.7). The Banach Fixed-Point Theorem for contractions implies that T^π has a unique fixed point and that

$$\lim_{n\to\infty} (T^\pi)^n(V_0) = V^\pi \quad \text{for any } V_0 \in \mathbb{R}^S.$$

The resulting iterative algorithm for solving a system of linear equations is called *Richardson's method*, see Bertsekas and Tsitsiklis [BT89, pp. 134] and Deuflhard and Hohmann [DH91, pp. 241] for a discussion.

Algorithm 1 is a *Gauss-Seidel* variant of the algorithm, where updated values are used as soon as they become available, see Bertsekas and Tsitsiklis

Input: a policy π and $\epsilon > 0$
Output: an approximation V of V^π with $\|V - V^\pi\|_\infty \leq \epsilon$
  initialize $V \in \mathbb{R}^S$ arbitrarily
  repeat
    $\Delta = 0$
    for all $s \in S$ **do**
      $v \leftarrow V(s)$
      $V(s) \leftarrow R^\pi(s) + \gamma \sum_{s' \in S} V(s') P^\pi(s', s)$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  until $\Delta \leq \epsilon$

Algorithm 1: Policy evaluation

[BT89, p. 135] and Sutton and Barto [SB98, p. 92]. For convergence results and a discussion of performance see Bertsekas and Tsitsiklis [BT89, pp. 151]. An asynchronous variant of the algorithm is discussed in [BT89, pp. 434].

We further observe that the value function is linear with respect to rewards. Let E be an environment, π a policy for E and γ a discount rate. Let $\mathbf{R}_1$ and $\mathbf{R}_2$ be two families of rewards for E and

$$\mathbf{R} = \alpha_1 \mathbf{R}_1 + \alpha_2 \mathbf{R}_2, \quad \text{with } \alpha_1, \alpha_2 \in \mathbb{R}.$$

Let R^π and V^π denote the expected reward and the value function for policy π and the MDP $(E, \mathbf{R}, \gamma)$ and analogously for $\mathbf{R}_1$ and $\mathbf{R}_2$. Then

$$R^\pi = \alpha_1 R_1^\pi + \alpha_2 R_2^\pi$$

and by Equation (1.8) we conclude

$$V^\pi = \alpha_1 V_1^\pi + \alpha_2 V_2^\pi. \tag{1.10}$$

1.2.5 Action-Values

Action-values are the expected sum of rewards, starting in a state, applying an action and then following a given policy. They became an important part of reinforcement learning with the introduction of Q-learning by Watkins [Wat89]. The notion of action-values simplifies the formulation of theorems and algorithms. It plays a fundamental role in our presentation of the theory. Whereas in reinforcement learning action-values are standard, in the literature on Markov decision processes and Dynamic Programming action-values are often not considered explicitly.

The Bellman equation (1.7) yields

$$V^\pi = R^\pi + \gamma V^\pi P^\pi \tag{1.11}$$

or, for each $s \in S$,

$$V^\pi(s) = \sum_{a \in A(s)} \left(R(a,s) + \gamma \sum_{s' \in S} V^\pi(s') P(s' \mid a, s) \right) \pi(a \mid s).$$

The equation suggests defining the *action-value* of action $a \in A(s)$ in state s for policy π

$$Q^\pi(a,s) = R(a,s) + \gamma \sum_{s' \in S} V^\pi(s') P(s' \mid a, s), \tag{1.12}$$

as the average reward if action a is chosen in state s and afterwards the policy π is followed. With this definition the Bellman equation obtains the form

$$V^\pi(s) = \sum_{a \in A(s)} Q^\pi(a,s) \pi(a \mid s). \tag{1.13}$$

Let $\tilde{\pi}(- \mid s)$ be any policy in s. Then

$$\sum_{a \in A(s)} Q^\pi(a,s) \tilde{\pi}(a \mid s) \leq \max_{a \in A(s)} Q^\pi(a,s) \sum_{a \in A(s)} \tilde{\pi}(a \mid s) = \max_{a \in A(s)} Q^\pi(a,s). \tag{1.14}$$

Equation (1.13) implies that the utility is between the minimal and maximal action-value, that is

$$\min_{a \in A(s)} Q^\pi(a,s) \leq V^\pi(s) \leq \max_{a \in A(s)} Q^\pi(a,s). \tag{1.15}$$

Like the value function the action-value function can be interpreted as the fixed point of a contraction mapping. The Bellman equation for state s' reads as follows

$$V^\pi(s') = \sum_{a' \in A(s')} Q^\pi(a', s') \pi(a' \mid s').$$

Substituting this equation in the definition of the action-value yields

$$Q^\pi(a,s) = R(a,s) + \gamma \sum_{s' \in S} \sum_{a' \in A(s')} Q^\pi(a', s') \pi(a' \mid s') P(s' \mid a, s). \tag{1.16}$$

We define the *transition matrix* $\widetilde{P}^\pi \in \mathbb{R}^{\mathbf{A}_S \times \mathbf{A}_S}$ of policy π for state-action pairs by

$$\widetilde{P}^\pi((a', s') \mid (a, s)) = \pi(a' \mid s') P(s' \mid a, s), \quad \text{for } (a', s'), (a, s) \in \mathbf{A}_S.$$

The affine map $\widetilde{T}^\pi \colon \mathbb{R}^{\mathbf{A}_S} \to \mathbb{R}^{\mathbf{A}_S}$ for a policy π defined by

$$\widetilde{T}^\pi(Q) = R^\pi + \gamma Q \tilde{P}^\pi. \tag{1.17}$$

is a contraction mapping, since $\tilde{P}^\pi$ is a stochastic matrix, and Q^π is the unique fixed point, compare Section 1.2.4.

Again the action-values are linear with respect to rewards. We use the same notation as for Equation (1.10). Then for a linear combination of rewards,

$$\mathbf{R} = \alpha_1 \mathbf{R}_1 + \alpha_2 \mathbf{R}_2, \quad \text{with } \alpha_1, \alpha_2 \in \mathbb{R},$$

and fixed discount rate γ, we see that the action-values for $(E, \mathbf{R}, \gamma)$ satisfy

$$Q^\pi = \alpha_1 Q_1^\pi + \alpha_2 Q_2^\pi, \tag{1.18}$$

using (1.12) and the linearity of the value function.

1.3 Comparing Policies

The value function reflects the performance of a policy. Thus we consider the value function as a criterion to compare policies and to characterize optimal policies. In our approach we emphasize the order on policies which turned out to be useful for the development of the theory for several environments. Additionally we gain new insights into the theory for one MDP.

1.3.1 Partially Ordered Sets

We recall the notation of partially ordered sets and some associated definitions. Let X be a set. A binary relation $\leq$ on X is a *partial order* on X if it is

- *reflexive*, $x \leq x$, for all $x \in X$,
- *transitive*, if $x \leq y$ and $y \leq z$ then $x \leq z$,
- *antisymmetric*, if $x \leq y$ and $y \leq x$ then $x = y$.

Let $(X, \leq)$ be a partially ordered set (*poset*). We use the notation $x < y$ to mean $x \leq y$ and $x \neq y$. We say that two elements x and y are *comparable* if $x \leq y$ or $y \leq x$; otherwise x and y are *incomparable*. A *total order* is a partial order in which every pair of elements is comparable. By a *greatest element* of X one means an element $y \in X$ such that $x \leq y$ for all $x \in X$. An element $m \in X$ is called a *maximal element* of X such that if $x \in X$

and $m \leq x$, then $x = m$ or equivalently if there is no element $x \in X$ greater than m. There may be many maximal elements in X, whereas if a greatest element exists, then it is unique. If a partial order is total then the notions of maximal and greatest element coincide.

We consider the *componentwise order* on $\mathbb{R}^S$, that is, for two vectors $x, y \in \mathbb{R}^S$ we define

$$x \leq y \text{ if } x(s) \leq y(s), \quad \text{for } s \in S.$$

The componentwise order is a partial order on $\mathbb{R}^S$. Note that $x < y$ means that $x(s) \leq y(s)$ for all $s \in S$ and $x(s) < y(s)$ for at least one $s \in S$.

1.3.2 Equivalent and Optimal Policies

Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π and $\tilde{\pi}$ be two policies for E and V^π and $V^{\tilde{\pi}}$ their value functions. We say that π and $\tilde{\pi}$ are *equivalent* if they have the same value function, that is $\pi \sim \tilde{\pi}$ if $V^\pi = V^{\tilde{\pi}}$. We denote the equivalence class of policy π by $[\pi]$ and define a partial order on the set of equivalence classes by $[\tilde{\pi}] \leq [\pi]$ if $V^{\tilde{\pi}} \leq V^\pi$.

The above construction can be summarized as follows. Let Π denote the set of all policies. The map

$$\Pi \to \mathbb{R}^S, \, \pi \mapsto V^\pi$$

induces the injection

$$\Pi/_\sim \to \mathbb{R}^S, \; [\pi] \mapsto V^\pi, \quad \text{where } \pi \sim \tilde{\pi} \text{ if } V^\pi = V^{\tilde{\pi}}.$$

The componentwise order on $\mathbb{R}^S$ induces a partial order on the set of equivalence classes $\Pi/_\sim$.

Note that equivalent policies have the same action-values, that is, if $\pi \sim \tilde{\pi}$ then

$$Q^\pi(a, s) = Q^{\tilde{\pi}}(a, s), \quad \text{for } s \in S, a \in A(s).$$

The inverse implication does not hold, see example in Section 2.5.2.

A policy π is called *optimal* for $(E, \mathbf{R}, \gamma)$ if $[\pi]$ is the greatest element of the set of equivalence classes of all policies. This means $V^{\tilde{\pi}} \leq V^\pi$ for all policies $\tilde{\pi}$. The existence of an optimal policy is proved constructively in Section 1.9.

Since the value function is linear with respect to rewards, the equivalence classes are invariant if we multiply the family of rewards by a nonzero real number. The order on equivalence classes and the notion of optimal policies are invariant if the rewards are multiplied by a positive number.

1.3.3 Utility of Policies

We call the average utility

$$V(\pi) = \frac{1}{|S|} \sum_{s \in S} V^{\pi}(s)$$

the *(discounted) utility* of policy π. Using the utility of a policy we can repeat the above construction of a partial order. We say that two policies π and $\tilde{\pi}$ are *equivalent on average* if their utilities are equal, that is $\pi \sim_a \tilde{\pi}$ if $V(\tilde{\pi}) = V(\pi)$. We define a partial order on the set of equivalence classes by $[\tilde{\pi}]_a \leq_a [\pi]_a$ if $V(\tilde{\pi}) \leq V(\pi)$.

Again the map

$$\Pi \to \mathbb{R},\ \pi \mapsto V(\pi)$$

induces the injection

$$\Pi/_{\sim_a} \to \mathbb{R},\ [\pi]_a \mapsto V^{\pi}, \quad \text{where } \pi \sim_a \tilde{\pi} \text{ if } V(\pi) = V(\tilde{\pi}).$$

The order on $\mathbb{R}$ induces a total order on the set of equivalence classes $\Pi/_{\sim_a}$. Obviously $[\tilde{\pi}] \leq [\pi]$ implies $[\tilde{\pi}]_a \leq_a [\pi]_a$. If $[\pi]$ is the greatest element with respect to $\leq$ then $[\pi]_a$ is the greatest (maximal) element with respect to $\leq_a$. Conversely, it is easy to see that if $[\pi]_a$ is the greatest element with respect to $\leq_a$ then it is maximal with respect to $\leq$ and hence optimal, once the existence of an optimal policy is shown. So if we want to speak of optimal policies it is sufficient to compare their utilities. For an optimal policy π the two equivalence classes $[\pi]$ and $[\pi]_a$ are equal.

1.4 Differences between Value Functions

In this section we introduce one-step differences between policies based on action-values to compare value functions. The following lemma relates action-values, value functions and transition matrices of two policies.

Lemma 6. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π and $\tilde{\pi}$ be policies for E. Then*

$$\sum_{a \in A(s)} Q^{\pi}(a, s)\tilde{\pi}(a \mid s) = \left(R^{\tilde{\pi}} + \gamma V^{\pi} P^{\tilde{\pi}}\right)(s), \quad \textit{for } s \in S.$$

Proof. Let $s \in S$. Then

$$\sum_{a \in A(s)} Q^{\pi}(a,s)\tilde{\pi}(a \mid s)$$
$$= \sum_{a \in A(s)} \left(R(a,s) + \gamma \sum_{s'} V^{\pi}(s') P(s' \mid a, s) \right) \tilde{\pi}(a \mid s)$$
$$= R^{\tilde{\pi}}(s) + \gamma \sum_{s'} V^{\pi}(s') P^{\tilde{\pi}}(s' \mid s) = \left(R^{\tilde{\pi}} + \gamma V^{\pi} P^{\tilde{\pi}} \right)(s).$$

■

The sum

$$\sum_{a \in A(s)} Q^{\pi}(a,s)\tilde{\pi}(a \mid s)$$

is the expected discounted return where the first step is chosen according to policy $\tilde{\pi}$ and policy π is followed afterwards.

Let π and $\tilde{\pi}$ be policies for E. We define the *one-step difference* between policies $\tilde{\pi}$ and π in state s by

$$D^{\tilde{\pi},\pi}(s) = \sum_{a \in A(s)} Q^{\pi}(a,s)\tilde{\pi}(a \mid s) - V^{\pi}(s), \quad \text{for } s \in S, \tag{1.19}$$

and call the vector $D^{\tilde{\pi},\pi} = \left(D^{\tilde{\pi},\pi}(s) \right)_{s \in S}$ in $\mathbb{R}^S$ the *one-step difference* between policies $\tilde{\pi}$ and π. The previous lemma gives

$$D^{\tilde{\pi},\pi} = R^{\tilde{\pi}} + \gamma V^{\pi} P^{\tilde{\pi}} - V^{\pi}. \tag{1.20}$$

Let π be a policy. The following theorem describes the difference between the value function for an arbitrary policy $\tilde{\pi}$ and the value function for π.

Theorem 7. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π and $\tilde{\pi}$ be policies for E and $D^{\tilde{\pi},\pi}$ the one-step difference between $\tilde{\pi}$ and π. Then*

$$V^{\tilde{\pi}} - V^{\pi} = D^{\tilde{\pi},\pi}(I - \gamma P^{\tilde{\pi}})^{-1}.$$

Proof. We have

$$(V^{\tilde{\pi}} - V^{\pi})(I - \gamma P^{\tilde{\pi}}) = V^{\tilde{\pi}} - V^{\pi} - \gamma V^{\tilde{\pi}} P^{\tilde{\pi}} + \gamma V^{\pi} P^{\tilde{\pi}}.$$

The Bellman equation (1.7) for $\tilde{\pi}$ gives

$$V^{\tilde{\pi}} - \gamma V^{\tilde{\pi}} P^{\tilde{\pi}} = R^{\tilde{\pi}}.$$

Therefore

$$(V^{\tilde{\pi}} - V^{\pi})(I - \gamma P^{\tilde{\pi}}) = R^{\tilde{\pi}} + \gamma V^{\pi} P^{\tilde{\pi}} - V^{\pi}.$$

and the theorem follows by Equation (1.20). ■

Reformulating this theorem to compare policies we obtain

Corollary 8. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π and $\tilde{\pi}$ be policies for E. Then*

$$V^{\tilde{\pi}} \geq V^{\pi} \text{ if and only if } D^{\tilde{\pi},\pi}(I - \gamma P^{\tilde{\pi}})^{-1} \geq 0.$$

We now consider criteria to compare value functions using the one-step difference only.

Theorem 9. *Let $(E, \mathbf{R})$ be an MDP. Let π and $\tilde{\pi}$ be policies for E. Then*

(i) $D^{\tilde{\pi},\pi} = 0$ *if and only if* $[\tilde{\pi}] = [\pi]$, *that is* $V^{\tilde{\pi}} = V^{\pi}$.

(ii) $D^{\tilde{\pi},\pi} \geq 0$ *implies* $[\tilde{\pi}] \geq [\pi]$, *that is* $V^{\tilde{\pi}} \geq V^{\pi}$.

(iii) $D^{\tilde{\pi},\pi} \leq 0$ *implies* $[\tilde{\pi}] \leq [\pi]$, *that is* $V^{\tilde{\pi}} \leq V^{\pi}$.

Proof. By theorem 7 we have

$$V^{\tilde{\pi}} - V^{\pi} = D^{\tilde{\pi},\pi}(I - \gamma P^{\tilde{\pi}})^{-1}.$$

Since $(I - \gamma P^{\tilde{\pi}})^{-1}$ is an invertible matrix assertion (i) follows. Observe furthermore that

$$(I - \gamma P^{\tilde{\pi}})^{-1} = \sum_{k=0}^{\infty} \gamma^k (P^{\tilde{\pi}})^k$$

is a nonnegative matrix. Hence $D^{\tilde{\pi},\pi} \geq 0$ implies $V^{\tilde{\pi}} - V^{\pi} \geq 0$. This proves assertions (ii) and (iii). ■

Corollary 10. *Let π and $\tilde{\pi}$ be policies for E. Then*

(i) $D^{\tilde{\pi},\pi} > 0$ *implies* $[\tilde{\pi}] > [\pi]$, *that is* $V^{\tilde{\pi}} > V^{\pi}$.

(ii) $D^{\tilde{\pi},\pi} < 0$ *implies* $[\tilde{\pi}] < [\pi]$, *that is* $V^{\tilde{\pi}} < V^{\pi}$.

1.5 Policy Improvement

We express the theorems on the one-step difference introduced in the previous section by means of action-values. Recall that the one-step difference between two policies in state s is defined by

$$D^{\tilde{\pi},\pi}(s) = \sum_{a \in A(s)} Q^{\pi}(a, s)\tilde{\pi}(a \mid s) - V^{\pi}(s).$$

A nonnegative one-step difference between two policies $\tilde{\pi}$ and π, $D^{\tilde{\pi},\pi} \geq 0$, is equivalent to

$$\sum_{a \in A(s)} Q^\pi(a,s)\tilde{\pi}(a \mid s) \geq V^\pi(s), \quad \text{for } s \in S.$$

Thus (ii) from Theorem 9 and (i) from Corollary 10 can be reformulated as follows.

Corollary 11. *Let $(E, \mathbf{R})$ be an MDP. Let π and $\tilde{\pi}$ be policies for E. Then*

$$\sum_{a \in A(s)} Q^\pi(a,s)\tilde{\pi}(a \mid s) \geq V^\pi(s) \quad \textit{for all } s \in S$$

implies $[\tilde{\pi}] \geq [\pi]$, that is $V^{\tilde{\pi}} \geq V^\pi$. If additionally

$$\sum Q^\pi(a,s)\tilde{\pi}(a \mid s) > V^\pi(s) \quad \textit{for at least one } s \in S$$

then $[\tilde{\pi}] > [\pi]$, that is $V^{\tilde{\pi}} > V^\pi$.

Sutton and Barto [SB98, pp. 95] call this result the *policy improvement* theorem.

Corollary 12. *Let $(E, \mathbf{R})$ be an MDP. Let π and $\tilde{\pi}$ be policies for E. Then $[\tilde{\pi}] = [\pi]$, that is $V^{\tilde{\pi}} = V^\pi$, if and only if*

$$\sum_{a \in A(s)} Q^\pi(a,s)\tilde{\pi}(a \mid s) = V^\pi(s) \quad \textit{for all } s \in S.$$

Note that to decide if two policies are equivalent we have to evaluate just one of them.

1.6 Improving Policies

In the previous section we consider the one-step difference between two policies. Now we look at the one-step difference between one policy and all possible policies.

Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π be a policy for E and $s \in S$. Let $\tilde{\pi}(- \mid s)$ denote a probability on $A(s)$. We define the set of *improving policies* for policy π in state s by

$$C^\pi_{\geq}(s) = \{\tilde{\pi}(- \mid s) : \sum_{a \in A(s)} Q^\pi(a,s)\tilde{\pi}(a \mid s) \geq V^\pi(s)\}.$$

Analogously we define the set of *equivalent policies* for policy π in state s by

$$C^\pi_{=}(s) = \{\tilde{\pi}(- \mid s) : \sum_{a \in A(s)} Q^\pi(a,s)\tilde{\pi}(a \mid s) = V^\pi(s)\}.$$

Observe that $\pi(- \mid s) \in C^\pi_=(s)$, since

$$\sum_{a \in A(s)} Q^\pi(a,s)\pi(a \mid s) = V^\pi(s)$$

by the Bellman equation (1.7). We define the set of *strictly improving policies* for policy π in state s by

$$C^\pi_>(s) = \{\tilde{\pi}(- \mid s) : \sum_{a \in A(s)} Q^\pi(a,s)\tilde{\pi}(a \mid s) > V^\pi(s)\},$$

Note that

$$C^\pi_>(s) = C^\pi_\geq(s) \setminus C^\pi_=(s).$$

We define the set of *strictly improving actions* for policy π in state s by

$$A^\pi_>(s) = \{a \in A(s) : Q^\pi(a,s) > V^\pi(s)\}.$$

We identify actions with deterministic policies in a state and interpret $A^\pi_>(s)$ as a subset of $C^\pi_>(s)$.

The following corollaries are reformulations of Theorem 9 and Corollary 10 with these definitions.

Corollary 13. *Let $(E, \mathbf{R})$ be an MDP. Let π and $\tilde{\pi}$ be policies for E. Then*

$$\tilde{\pi}(- \mid s) \in C^\pi_\geq(s) \quad \textit{for all } s \in S$$

implies $[\tilde{\pi}] \geq [\pi]$, that is $V^{\tilde{\pi}} \geq V^\pi$. If additionally

$$\tilde{\pi}(- \mid s) \in C^\pi_>(s) \quad \textit{for at least one } s \in S$$

then $[\tilde{\pi}] > [\pi]$, that is $V^{\tilde{\pi}} > V^\pi$.

Corollary 14. *Let $(E, \mathbf{R})$ be an MDP. Let π and $\tilde{\pi}$ be policies for E. Then $[\tilde{\pi}] = [\pi]$, that is $V^{\tilde{\pi}} = V^\pi$, if and only if*

$$\tilde{\pi}(- \mid s) \in C^\pi_=(s) \quad \textit{for all } s \in S.$$

This reformulation using sets of improving, strictly improving and equivalent policies motivates a geometric interpretation of policies, which is explained in Section 1.8.

Since the value function and action-values are equal for equivalent policies, the above sets are well defined for equivalence classes, that is, if $\pi \sim \tilde{\pi}$ then

$$C^\pi_\geq(s) = C^{\tilde{\pi}}_\geq(s), \quad \text{for } s \in S,$$

and analogously for $C^\pi_>(s)$, $C^\pi_=(s)$ and $A^\pi_>(s)$.

Recall from the Sections 1.2.4 and 1.2.5 that the value function and the action-values are linear with respect to rewards. Therefore the sets of (strictly) improving policies and actions are invariant if we multiply the family of rewards by a positive real number. The set of equivalent policies remains unchanged if the rewards are multiplied by a nonzero number.

1.7 Optimality Criteria

We characterize optimal policies by means of one-step differences and the sets of strictly improving policies and actions. Recall that a policy π is optimal if $[\pi]$ is the greatest element of the set of equivalence classes of all policies, that is, if $V^\pi \geq V^{\tilde{\pi}}$ for all policies $\tilde{\pi}$, see Section 1.3.2.

Theorem 15. *Let $(E, \mathbf{R})$ be an MDP. Let π be a policy for E. Then the following conditions are equivalent:*

(i) *π is optimal.*

(ii) *$D^{\tilde{\pi},\pi} \leq 0$ for all policies $\tilde{\pi}$.*

(iii) *$C^\pi_>(s) = \emptyset$ for all $s \in S$.*

(iv) *$A^\pi_>(s) = \emptyset$ for all $s \in S$.*

(v) *$V^\pi(s) = \max_{a \in A(s)} Q^\pi(a, s)$ for all $s \in S$.*

Proof. We first prove the equivalence of (ii), (iii), (iv) and (v) and then (ii)⇒(i) and (i)⇒(iii).

(ii)⇒(iii) Let π be a policy such that $D^{\tilde{\pi},\pi} \leq 0$ for all policies $\tilde{\pi}$. Then by definition of the one-step difference (1.19) we have

$$\sum_{a \in A(s)} Q^\pi(a, s)\tilde{\pi}(a \mid s) \leq V^\pi(s), \quad \text{for } s \in S \text{ and all policies } \tilde{\pi}.$$

Therefore $C^\pi_>(s) = \emptyset$ for all $s \in S$.

(iii)⇒(iv) Obvious, since $A^\pi_>(s) \subset C^\pi_>(s)$.

(iv)⇒(v) Suppose now that condition (v) does not hold. Then for some $s \in S$ we have

$$V^\pi(s) < \max_{a \in A(s)} Q^\pi(a, s),$$

see Equation (1.15). Thus there exists an $a \in A(s)$ with $Q^\pi(a, s) > V^\pi(s)$ and hence $a \in A^\pi_>(s)$, a contradiction to (iv).

(v)⇒(ii) Let $\tilde{\pi}$ be a policy. Then by Equation (1.14)

$$\sum_{a \in A(s)} Q^{\pi}(a,s)\tilde{\pi}(a \mid s) \leq \max_{a \in A(s)} Q^{\pi}(a,s) = V^{\pi}(s), \quad \text{for } s \in S.$$

Thus $D^{\tilde{\pi},\pi} \leq 0$.

(ii)⇒(i) The condition $D^{\tilde{\pi},\pi} \leq 0$ for all $\tilde{\pi}$ implies $V^{\tilde{\pi}} \leq V^{\pi}$ for all $\tilde{\pi}$ according to Theorem 9 and thus π is optimal.

(i)⇒(iii) Suppose that $C^{\pi}_{>}(\tilde{s}) \neq \emptyset$ for an $\tilde{s} \in S$. Let $\tilde{\pi}(- \mid \tilde{s}) \in C^{\pi}_{>}(\tilde{s})$ and define the policy $\tilde{\pi}$ by

$$\tilde{\pi}(a \mid s) = \begin{cases} \tilde{\pi}(- \mid \tilde{s}), & \text{if } s = \tilde{s}, \\ \pi(- \mid s), & \text{if } s \in S \setminus \{\tilde{s}\}. \end{cases}$$

Then $[\tilde{\pi}] > [\pi]$ by Corollary 13 and π is not optimal, a contradiction to (i).

■

The conditions (iv) and (v) of the theorem are used as termination rules for the policy improvement algorithm, see Section 1.9.

1.8 Policies and Polytopes

In the following subsections we give a geometric interpretation of policies.

1.8.1 Polyhedra and Polytopes

We recall some basic notions and results on polyhedra and polytopes. Our notation is based on Ziegler [Zie98]. Further references for this section are Borgwardt [Bor01], Chvátal [Chv83], the lecture notes [Pau02] by Pauer, the lecture notes [Sch03] and the classical book [Sch86] by Schrijver.

We denote column vectors in $\mathbb{R}^d$ by $\mathbf{x}, \mathbf{x}_0, \ldots, \mathbf{y}, \mathbf{z}$ and row vectors by $\mathbf{a}, \mathbf{a}_0, \ldots, \mathbf{b}, \mathbf{c}$. A subset $K \subset \mathbb{R}^d$ is *convex* if with any two points $\mathbf{x}, \mathbf{y} \in K$ it contains the *line segment*

$$[\mathbf{x}, \mathbf{y}] = \{\lambda \mathbf{x} + (1-\lambda)\mathbf{y} : 0 \leq \lambda \leq 1\}$$

between them. The intersection of any number of convex sets is again a convex set. So, the smallest convex set containing any subset $X \subset \mathbb{R}^d$ exists.

This set is called the *convex hull* of X and is denoted by $\text{conv}(X)$. It is given by

$$\text{conv}(X) = \{\lambda_1 \mathbf{x}_1 + \cdots + \lambda_n \mathbf{x}_n : n \geq 1, \mathbf{x}_i \in X, \lambda_i \geq 0, \sum_{i=1}^{n} \lambda_i = 1\}.$$

Let $\mathbf{a} = (a_1, \ldots, a_d)$ be a row and $\mathbf{x} = (x_1, \ldots, x_d)^T$ a column vector. We write

$$\mathbf{ax} = \sum_{i=1}^{d} a_i x_i \in \mathbb{R}$$

for the inner product of $\mathbf{a}$ and $\mathbf{x}$. Let $\mathbf{a}$ be a nonzero vector and $z \in \mathbb{R}$. Sets of the form $\left\{\mathbf{x} \in \mathbb{R}^d \colon \mathbf{ax} = z\right\}$ are called *(affine) hyperplanes* and sets of the form $\left\{\mathbf{x} \in \mathbb{R}^d \colon \mathbf{ax} \leq z\right\}$ *(affine) halfspaces*. Hyperplanes and halfspaces are obviously convex sets.

If A is a matrix and $\mathbf{x}, \mathbf{z}$ are vectors, then when using notations like

$$A\mathbf{x} = \mathbf{z} \text{ or } A\mathbf{x} \leq \mathbf{z},$$

we implicitly assume compatibility of sizes of $A, \mathbf{x}, \mathbf{z}$. Here $A\mathbf{x} \leq \mathbf{z}$ stands for the *system of inequalities*

$$\mathbf{a}_1\mathbf{x} \leq z_1, \ldots, \mathbf{a}_m\mathbf{x} \leq z_m,$$

where $\mathbf{a}_1, \ldots, \mathbf{a}_m$ are the rows of A and $\mathbf{z} = (z_1, \ldots, z_m)^T$. The system $A'\mathbf{x} \leq \mathbf{z}'$ is a *subsystem* of $A\mathbf{x} \leq \mathbf{z}$ if $A'\mathbf{x} \leq \mathbf{z}'$ arises from $A\mathbf{x} \leq \mathbf{z}$ by deleting some (or none) of the inequalities in $A\mathbf{x} \leq \mathbf{z}$. Analogously, we define subsystems of linear equations.

A subset $P \subset \mathbb{R}^d$ is a *polyhedron* if it is the intersection of finitely many halfspaces, that is

$$P = P(A, \mathbf{z}) = \left\{\mathbf{x} \in \mathbb{R}^d : A\mathbf{x} \leq \mathbf{z}\right\}$$

for a matrix A and vector $\mathbf{z}$. A convex subset of $\mathbb{R}^d$ is *bounded* if it does not contain a *halfline*

$$\left\{\mathbf{x} + t\mathbf{y} : t \geq 0, \mathbf{x}, \mathbf{y} \in \mathbb{R}^d, \mathbf{y} \neq 0\right\}.$$

A subset $P \subset \mathbb{R}^d$ is a *polytope* if it is the convex hull of finitely many vectors, that is

$$P = \text{conv}(\mathbf{x}_1, \ldots, \mathbf{x}_n), \quad \text{with } \mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d.$$

The finite basis theorem for polytopes describes the geometrically clear relation between polyhedra and polytopes. It is usually attributed to Minkowski, Steinitz and Weyl. For a proof we refer to Schrijver [Sch86, p. 89] or Ziegler [Zie98, pp. 29].

Theorem (finite basis theorem). *A subset $P \subset \mathbb{R}^d$ is a polytope if and only if it is a bounded polyhedron.*

The *(affine) dimension* of a convex subset $K \subset \mathbb{R}^d$, $\dim(K)$, is the dimension of its *affine hull*

$$\text{aff}(K) = \{\lambda_1 \mathbf{x}_1 + \cdots + \lambda_n \mathbf{x}_n : n \geq 1, \mathbf{x}_i \in K, \sum_{i=1}^{n} \lambda_i = 1\}.$$

Let $P \subset \mathbb{R}^d$ a polyhedron. A linear inequality $\mathbf{a}\mathbf{x} \leq z$ is called *valid* for P if $\mathbf{a}\mathbf{x} \leq z$ holds for all $\mathbf{x} \in P$. Any subset $F \subset P$ of the form

$$F = P \cap \left\{\mathbf{x} \in \mathbb{R}^d : \mathbf{a}\mathbf{x} = z\right\}$$

for a valid inequality $\mathbf{a}\mathbf{x} \leq z$ is called a *face* of P. The polyhedron P and the empty set are faces of P for the valid inequalities $\mathbf{0}\mathbf{x} \leq 0$ and $\mathbf{0}\mathbf{x} \leq -1$ respectively. Faces of dimensions $0, 1, \dim(P) - 2$ and $\dim(P) - 1$ are called *vertices*, *edges*, *ridges* and *facets*.

We introduce the following notation to give a different characterization of vertices which we use later on. Let $P = P(A, \mathbf{z})$ be a polyhedron and $\mathbf{x} \in P$. Then $A_{\mathbf{x}}$ is the submatrix of A consisting of the rows $\mathbf{a}_i$ of A for which $\mathbf{a}_i\mathbf{x} = z_i$. For a proof of the following theorem we refer to Chvátal [Chv83, pp. 271] or Schrijver [Sch03, p. 20].

Theorem. *Let $P = P(A, \mathbf{z})$ be a polyhedron in $\mathbb{R}^d$ and $\mathbf{x} \in P$. Then $\mathbf{x}$ is a vertex of P if and only if* $\text{rank}(A_{\mathbf{x}}) = d$.

If $\mathbf{x}$ is a vertex then we can choose d linear independent rows from $A_{\mathbf{x}}$. So we obtain the following corollary.

Corollary 16. *Let $P = P(A, \mathbf{z})$ be a polyhedron in $\mathbb{R}^d$ and $\mathbf{x} \in P$. Then $\mathbf{x}$ is a vertex of P if and only if $\mathbf{x}$ is the unique solution of $A'\mathbf{x} = \mathbf{z}'$ for a subsystem of $A\mathbf{x} = \mathbf{z}$ with an invertible $A' \in \mathbb{R}^{d \times d}$.*

Thus theoretically we can compute all vertices of a polyhedron as follows. We choose all possible sets of d linear independent rows of A. For each set of equations we compute the unique solution $\mathbf{x}'$ of the resulting subsystem $A'\mathbf{x} = \mathbf{z}'$ of $A\mathbf{x} = \mathbf{z}$. Then we test if $\mathbf{x}'$ satisfies all other inequalities, that is, if $\mathbf{x}' \in P$.

So if A has m rows then P has at most $\binom{m}{d}$ vertices. This is just a basic upper bound for the number of vertices. The best possible bound is given by McMullen's Upper Bound Theorem [McM70], which implies in particular that the number of vertices of a polytope in $\mathbb{R}^d$ with m facets is at most

$$f(m, d) = \binom{m - \left\lfloor \frac{d+1}{2} \right\rfloor}{m - d} + \binom{m - \left\lfloor \frac{d+2}{2} \right\rfloor}{m - d}. \tag{1.21}$$

Compare also Chvátal [Chv83, pp. 272]. Here $\lfloor x \rfloor$ denotes the *floor function* for $x \in \mathbb{R}$, that is, the largest integer less than or equal to x.

For a discussion of efficient algorithms for computing all vertices of a polyhedron, related questions and available software we refer to the FAQ in Polyhedral Computation by Fukuda [Fuk00].

The set of vertices of a polytope P is denoted by $\mathrm{vert}(P)$. We have the following theorem, see Ziegler [Zie98, p. 52].

Theorem. *Let $P \subset \mathbb{R}^d$ be a polytope. Then P is the convex hull of its vertices, that is*

$$P = \mathrm{conv}(\mathrm{vert}(P)).$$

The set of vertices of a polytope is the minimal set of points such that P is its convex hull.

1.8.2 Policies and Simplices

We denote by $\mathbf{e}_i$ the ith *standard (basis) vector* in $\mathbb{R}^d$, that is, $\mathbf{e}_i = (e_1, ..., e_d)^T$ with $e_i = 1$ and $e_j = 0$ for $j \neq i$. The *standard d-simplex* is the convex hull of the $d+1$ standard vectors in $\mathbb{R}^{d+1}$ and is denoted by

$$\triangle_d = \mathrm{conv}\,\{\mathbf{e}_1, \ldots, \mathbf{e}_{d+1}\} = \{\mathbf{x} \in \mathbb{R}^{d+1} : x_i \geq 0, \sum x_i = 1\}.$$

It is a d-dimensional polytope with the $d+1$ standard vectors as vertices.

Let $(E, \mathbf{R}, \gamma)$ be an MDP and $s \in S$. We consider the standard d-simplex with $d = |A(s)| - 1$ in $\mathbb{R}^{A(s)}$ and denote it by $C(s)$. If we write δ_a for the standard vectors in $\mathbb{R}^{A(s)}$, that is

$$\delta_a(\tilde{a}) = \begin{cases} 1, & \text{if } \tilde{a} = a, \\ 0, & \text{otherwise,} \end{cases}$$

then

$$C(s) = \mathrm{conv}(\delta_a : a \in A(s)) = \{x \in \mathbb{R}^{A(s)} : x(a) \geq 0, \sum_{a \in A(s)} x(a) = 1\}.$$

We identify a policy $\pi(- \mid s)$ in state s with the point $x \in C(s)$ by

$$x(a) = \pi(a \mid s), \quad \text{for } a \in A(s).$$

The vertices δ_a of $C(s)$ represent the deterministic policies in state s. For an example with three actions $A(s) = \{a_1, a_2, a_3\}$ see Figure 1.1.

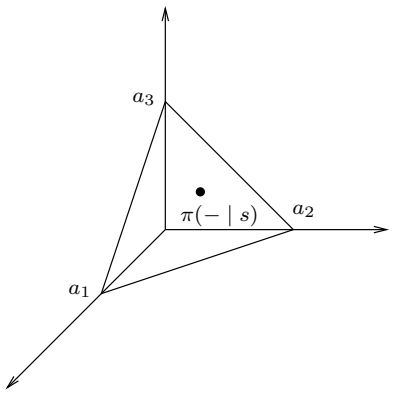

Figure 1.1: A policy in state as a point on the standard simplex

We identify a policy π for E with an element in the product of simplices

$$\pi \in \prod_{s \in S} C(s)$$

and the set of policies Π with the polytope

$$\Pi = \prod_{s \in S} C(s) \subset \prod_{s \in S} \mathbb{R}^{A(s)}.$$

It is easy to see that the vertices of the set Π are the deterministic policies and therefore

$$\Pi = \text{conv}(\pi \in \Pi : \pi \text{ deterministic}).$$

1.8.3 Improving Vertices

Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π be a policy for E and $s \in S$. Recall that the set of equivalent policies $C^{\pi}_{=}(s)$ for π in s is defined as the set of policies $\tilde{\pi}(- \mid s)$ in s satisfying

$$\sum_{a \in A(s)} Q^{\pi}(a, s)\tilde{\pi}(a \mid s) = V^{\pi}(s).$$

Geometrically, we can interpret $C^{\pi}_{=}(s)$ as the polytope given by the intersection of the hyperplane

$$H^{\pi}_{=}(s) = \{x \in \mathbb{R}^{A(s)} : \sum_{a \in A(s)} Q^{\pi}(a, s)x(a) = V^{\pi}(s)\}$$

and the standard simplex $C(s)$ in $\mathbb{R}^{A(s)}$, that is

$$C^{\pi}_{=}(s) = H^{\pi}_{=}(s) \cap C(s).$$

See Figure 1.2 for an example with three actions.

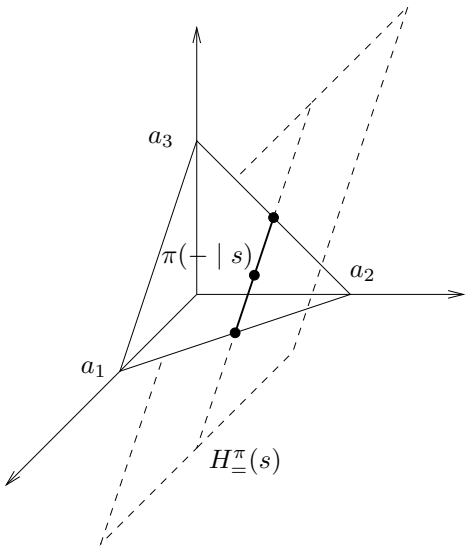

Figure 1.2: Intersection of a hyperplane and the standard simplex

The set of improving policies $C^{\pi}_{\geq}(s)$ for π in s is defined by the set of policies $\tilde{\pi}(- \mid s)$ in state s satisfying

$$\sum_{a \in A(s)} Q^{\pi}(a, s)\tilde{\pi}(a \mid s) \geq V^{\pi}(s).$$

Thus $C^{\pi}_{\geq}(s)$ is the intersection of the halfspace

$$H^{\pi}_{\geq}(s) = \{x \in \mathbb{R}^{A(s)} : \sum_{a \in A(s)} Q^{\pi}(a, s)x(a) \geq V^{\pi}(s)\}$$

and the standard simplex $C(s)$ in $\mathbb{R}^{A(s)}$, that is

$$C^{\pi}_{\geq}(s) = H^{\pi}_{\geq}(s) \cap C(s).$$

To compute the vertices of the polytopes $C^{\pi}_{=}(s)$ and $C^{\pi}_{\geq}(s)$ we use the following notation. Let $A(s) = \{a_1, \ldots, a_{d+1}\}$,

$$\mathbf{c} = (Q^{\pi}(a_1, s), \ldots, Q^{\pi}(a_{d+1}, s)) \in \mathbb{R}^{d+1} \text{ and } z = V^{\pi}(s) \in \mathbb{R}.$$

Let $\mathbf{e}_i \in \mathbb{R}^{d+1}$ denote the ith standard basis vector and $\mathbf{1} = (1, \ldots, 1) \in \mathbb{R}^{d+1}$. Then the standard $d+1$-simplex is defined by the system of (in)equalities

$$\mathbf{e}_i^T \mathbf{x} \geq 0, \quad \text{for } i = 1, \ldots, d+1, \tag{1.22}$$

$$\mathbf{1}\mathbf{x} = 1. \tag{1.23}$$

The polytopes $C^\pi_=(s)$ and $C^\pi_\geq(s)$ are defined by the additional (in)equality

$$\mathbf{c}\mathbf{x} = z, \text{ or } \mathbf{c}\mathbf{x} \geq z \tag{1.24}$$

respectively.

Theorem 17. *We have*

$$\operatorname{vert}(C^\pi_=(s)) = \{\mathbf{e}_k : c_k = z\} \cup \left\{\mathbf{v}_{ij} = \frac{z - c_j}{c_i - c_j}\mathbf{e}_i + \frac{c_i - z}{c_i - c_j}\mathbf{e}_j : c_i > z, c_j < z\right\} \tag{1.25}$$

and

$$\operatorname{vert}(C^\pi_\geq(s)) = \{\mathbf{e}_k : c_k > z\} \cup \operatorname{vert}(C^\pi_=(s)). \tag{1.26}$$

Proof. We use the characterization of vertices from Corollary 16 to compute the vertices of $C^\pi_=(s)$ and $C^\pi_\geq(s)$. So, we have to choose all subsets of $d+1$ linearly independent rows from the $d+3$ Equations (1.22), (1.23) and (1.24). Then we compute the unique solution $\mathbf{x} \in \mathbb{R}^{d+1}$ of the resulting system of linear equations and test whether the solution satisfies the remaining (in)equalities. The unique solution for all $d+1$ linear equations from (1.22) is $\mathbf{x} = 0$, which is not a point on the standard simplex. Choosing d linear equations from (1.22), omitting $\mathbf{e}_k^T\mathbf{x} = 0$, and Equation (1.23) gives the solution $\mathbf{x} = \mathbf{e}_k$. If the solution satisfies the (in)equality from Equation (1.24) then either $c_k = z$ or $c_k > z$, which gives the equivalent and improving vertices

$$\{\mathbf{e}_k : c_k = z\} \text{ and } \{\mathbf{e}_k : c_k > z\}$$

respectively. We can also choose $d-1$ linear equations from (1.22), omitting $\mathbf{e}_i^T\mathbf{x} = 0$ and $\mathbf{e}_j^T\mathbf{x} = 0$, and the two equations from (1.23), (1.24). Then $\mathbf{x}$ has only two nonzero components x_i and x_j. Furthermore $c_i \neq c_j$, otherwise the $d+1$ equations would be linearly dependent. So we obtain the two linearly independent equations

$$\begin{aligned} x_i + x_j &= 1, \\ c_i x_i + c_j x_j &= z. \end{aligned}$$

The unique solution

$$x_i = \frac{z - c_j}{c_i - c_j} \text{ and } x_j = \frac{c_i - z}{c_i - c_j}$$

is positive, that is, it lies on the standard simplex and is not contained in the previous case, if and only if $c_j < z$ and $c_i > z$. ■

We call the vertices $\text{vert}(C^\pi_=(s))$ *equivalent vertices* and $\text{vert}(C^\pi_\geq(s))$ *improving vertices* for policy π in state s. In Figure 1.3 the equivalent vertices are $\mathbf{v}_{12}$ and $\mathbf{v}_{32}$. The polytope $C^\pi_=(s)$ is the line segment between them.

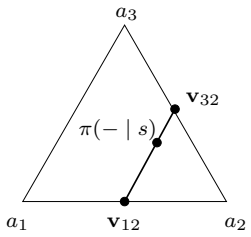

Figure 1.3: Equivalent policies

Using strictly improving actions Equation (1.26) reads as follows

$$\text{vert}(C^\pi_\geq(s)) = A_>(s) \cup \text{vert}(C^\pi_=(s)). \tag{1.27}$$

See Figure 1.4, where $A^\pi_>(s) = \{a_1, a_3\}$, $\text{vert}(C^\pi_=(s)) = \{\mathbf{v}_{12}, \mathbf{v}_{32}\}$ and the polytope $C^\pi_\geq(s)$ is the shaded area, the side marked by the small arrows.

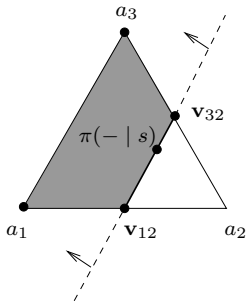

Figure 1.4: Improving policies

Using the value function and action-values function the equivalent vertices from (1.25) read as follows

$$\delta_a, \quad \text{with } a \in A(s) \text{ and } Q^\pi(a, s) = V^\pi(s)$$

and

$$\frac{V^\pi(s) - Q^\pi(\tilde{a}, s)}{Q^\pi(a, s) - Q^\pi(\tilde{a}, s)}\delta_a + \frac{Q^\pi(a, s) - V^\pi(s)}{Q^\pi(a, s) - Q^\pi(\tilde{a}, s)}\delta_{\tilde{a}},$$

with $a, \tilde{a} \in A(s)$ and $Q^\pi(a,s) > V^\pi(s)$, $Q^\pi(\tilde{a},s) < V^\pi(s)$. The improving vertices (1.26) are the equivalent vertices and

$$\delta_a, \quad \text{with } a \in A(s) \text{ and } Q^\pi(a,s) > V^\pi(s).$$

Observe that each vertex of $C^\pi_=(s)$ lies on an edge between two actions. We obtain an equivalent policy by choosing an equivalent vertex for each state $s \in S$ by Corollary 14. This implies the following theorem.

Theorem 18. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π be a policy for E. Then there exists an equivalent policy $\tilde{\pi}$ with the property that for each state $s \in S$ we have*

$$\tilde{\pi}(a \mid s) > 0 \text{ for at most two different actions.}$$

The above theorem is useful in applications since for every stochastic policy there exists an equivalent policy using two actions a and $\tilde{a}$ only in each s with $\pi(a \mid s) + \pi(\tilde{a} \mid s) = 1$. Such policies are determined by a triple containing two actions and the probability of taking the first action, that is π is given by $(a, \tilde{a}, p)$ with $a, \tilde{a} \in A(s)$ and $0 \leq p \leq 1$ for each $s \in S$. The second action is taken with probability $1 - p$.

1.8.4 Number of Improving Vertices

We give bounds for the number of equivalent and improving vertices, which are useful in applications. For example the required memory to store the equivalent and improving vertices can be estimated in terms of the number of actions.

Let $\lceil x \rceil$ denote the *ceiling function* for $x \in \mathbb{R}$, that is, the least integer greater than or equal to x. We denote the floor function by $\lfloor x \rfloor$. Graham, Knuth and Pathashnik [GKP94, pp. 67] give an extensive discussion of the floor and ceiling function, applications and related formulas. We use the following identity [GKP94, p. 96]

$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n+m-1}{m} \right\rfloor, \quad \text{for integers } n, m > 0. \tag{1.28}$$

From Equation (1.25) it follows that the number of equivalent vertices is

$$|\mathrm{vert}(C^\pi_=(s))| = |\{k : c_k = z\}| + |\{i : c_i > z\}| \cdot |\{j : c_j < z\}|.$$

So it is maximal (for $d \geq 2$) if we have

$$\left\lceil \frac{d+1}{2} \right\rceil \text{ indices } i \text{ with } c_i > z$$

and

$$\left\lfloor \frac{d+1}{2} \right\rfloor \text{ indices } j \text{ with } c_j < z$$

or vice versa. Thus

$$|\mathrm{vert}(C^{\pi}_{=}(s))| \leq \left\lceil \frac{d+1}{2} \right\rceil \left\lfloor \frac{d+1}{2} \right\rfloor = a(d+1),$$

with

$$a(n) = \left\lceil \frac{n}{2} \right\rceil \left\lfloor \frac{n}{2} \right\rfloor, \quad \text{for } n = 0, 1, 2, \ldots.$$

The first terms of the sequence $a(n)$ for $n = 0, \ldots, 10$ are

$$0, 0, 1, 2, 4, 6, 9, 12, 16, 20, 25. \tag{1.29}$$

Entering these numbers in Sloane's "On-Line Encyclopedia of Integer Sequences" [Slo04], see also Sloane [Slo03], we find that the sequence $a(n)$ (sequence `A002620`) is called "quarter-squares", since

$$a(n) = \left\lfloor \frac{n^2}{4} \right\rfloor.$$

Various applications and references are given. Furthermore, we find that the sequence satisfies several recurrences, for example

$$a(n) = a(n-1) + \left\lfloor \frac{n}{2} \right\rfloor, \quad \text{for } n > 0.$$

Hence with Equation (1.28) we conclude

$$a(n+1) = a(n) + \left\lfloor \frac{n+1}{2} \right\rfloor = a(n) + \left\lceil \frac{n}{2} \right\rceil, \quad \text{for } n \geq 0.$$

Using Equation (1.26) we see that the number of improving vertices is

$$\left|\mathrm{vert}(C^{\pi}_{\geq}(s))\right| = |\{i : c_i > z\}| + |\mathrm{vert}(C^{\pi}_{=}(s))|.$$

So it is maximal if we have

$$\left\lceil \frac{d+1}{2} \right\rceil \text{ indices } i \text{ with } c_i > z$$

and

$$\left\lfloor \frac{d+1}{2} \right\rfloor \text{ indices } j \text{ with } c_j < z.$$

Thus

$$\left|\mathrm{vert}(C^\pi_\geq(s))\right| \leq \left\lceil \frac{d+1}{2} \right\rceil \left\lfloor \frac{d+1}{2} \right\rfloor + \left\lceil \frac{d+1}{2} \right\rceil$$

and

$$\left|\mathrm{vert}(C^\pi_\geq(s))\right| \leq a(d+2).$$

See Figure 1.4, where the are $4 = a(4)$ improving and $2 = a(3)$ equivalent vertices, which is maximal.

Recall McMullen's upper bound $f(m,d)$ for the maximal number of vertices of a polyhedron defined by m linear inequalities in $\mathbb{R}^d$, see (1.21). Computing the sequence

$$f(d+3,d+1) = \binom{d+3-\left\lfloor \frac{d+2}{2} \right\rfloor}{2} + \binom{d+2-\left\lfloor \frac{d+1}{2} \right\rfloor}{2}$$

for $d = 0, \ldots, 8$ gives

$$2, 4, 6, 9, 12, 16, 20, 25, 30.$$

Considering the cases d even and d odd one sees that $f(d+3, d+1) = a(d+3)$ and we conclude that the polytope of improving policies can have the maximal possible number of vertices.

1.9 Policy Iteration

The *policy iteration* algorithm computes an optimal policy for a given Markov decision process. It is usually attributed to Howard [How65, p. 38] and Bellman [Bel57], compare also Kallenberg in [FS02, p. 34] and Denardo [Den03, p. 167].

The main idea of the algorithm is to combine policy evaluation with policy improvement steps. We start with an arbitrary policy and compute its value function and action-values. Then we try to improve the policy by choosing a strictly improving action in as many states as possible, see Section 1.6. Once improved we evaluate the new policy and again try to improve it. The algorithm terminates if there are no strictly improving actions. Then the policy is optimal due to (iv) of Theorem 15. A formal description of policy iteration for a given Markov decision process is shown in Algorithm 2.

We choose an arbitrary strictly improving action $a \in A^\pi_>(s)$ to improve the policy π. A usual choice is an action, that maximizes the action-value, that is

$$a \in \arg\max_a Q^\pi(a,s). \tag{1.30}$$

Output: an optimal policy π
  choose a starting policy π
  repeat
    compute V^π and Q^π
    for all $s \in S$ **do**
      compute $A^\pi_>(s)$
      if $A^\pi_>(s) \neq \emptyset$ **then**
        choose $a \in A^\pi_>(s)$
        $\pi(- \mid s) \leftarrow a$
  until $A^\pi_>(s) = \emptyset$ for all $s \in S$

Algorithm 2: Policy iteration

The resulting policy is called a *greedy policy* for Q^π, see Sutton and Barto [SB98, p. 96]. It selects an action that appears best according to the action-values.

If we change a policy by an improvement step in a state the policy becomes deterministic in this state and remains unchanged in all others, in particular a deterministic policy remains deterministic. Since the number of deterministic policies is finite, policy iteration terminates after finitely many steps.

Theorem 19. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Then there exists an optimal deterministic policy π.*

Proof. We run the policy iteration algorithm with a deterministic starting policy. We obtain a deterministic optimal policy in finitely many iteration. ■

A geometric interpretation of one step of the policy iteration algorithm for three states is shown in Figure 1.5. In state s_1 the set of strictly improving

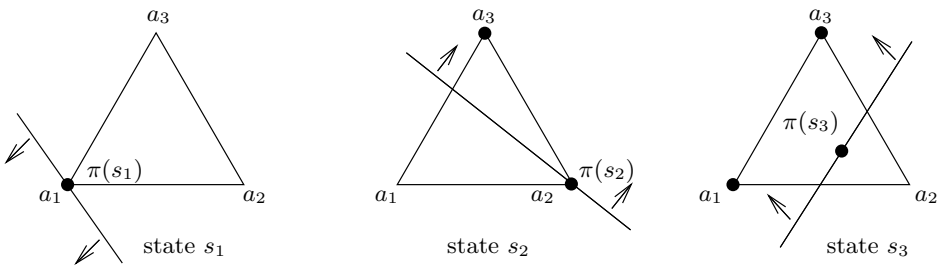

Figure 1.5: Policy iteration and strictly improving actions

actions is empty. In state s_2 we have one strictly improving action a_3. In

state s_3 we can choose between the two strictly improving actions a_1 and a_3. Note that the policy in this state is stochastic, which can occur when we run the algorithm with a stochastic starting policy.

1.10 Optimal Value Function and Actions

In this section we give a criterion for optimal policies by means of optimal actions and describe the set of all optimal policies geometrically.

Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π^* be an optimal policy for E. Then $[\pi^*]$ is the greatest element of the set of equivalence classes of policies and all optimal policies share the same value function by definition. We define the *optimal value function* of the MDP $(E, \mathbf{R}, \gamma)$ by

$$V^*(s) = V^{\pi^*}(s), \quad \text{for } s \in S.$$

Since $V^\pi \leq V^{\pi^*}$ for all policies π, we see that

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s), \quad \text{for } s \in S. \tag{1.31}$$

The *optimal action-value* for action $a \in A(s)$ in state $s \in S$ is defined by

$$Q^*(a, s) = R(a, s) + \gamma \sum_{s' \in S} V^*(s') P(s' \mid a, s) = Q^{\pi^*}(a, s). \tag{1.32}$$

Criterion (v) of Theorem 15 gives

$$V^*(s) = \max_{a \in A(s)} Q^*(a, s), \quad \text{for } s \in S. \tag{1.33}$$

Combining the two last equations yields

$$V^*(s) = \max_{a \in A(s)} \Big(R(a, s) + \gamma \sum_{s' \in S} V^*(s') P(s' \mid a, s) \Big), \quad \text{for } s \in S. \tag{1.34}$$

This equation is often referred to as *Bellman optimality equation* for the optimal value function, for example in Sutton and Barto [SB98, p. 76] or Puterman [Put94, p. 147]. Bellman calls it the "basic functional equation" [Bel57].

We obtain an analogous equation for the optimal action-values

$$Q^*(a, s) = R(a, s) + \gamma \sum_{s' \in S} \max_{a' \in A(s')} Q^*(a', s') P(s' \mid a, s). \tag{1.35}$$

Corollary 14 implies the following characterization of optimal policies.

Corollary 20. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Then a policy π is optimal if and only if*

$$\sum_{a \in A(s)} Q^*(a,s)\pi(a \mid s) = V^*(s) \quad \textit{for all } s \in S.$$

We define the *set of optimal actions* in state s by

$$A^*(s) = \{a \in A(s) : Q^*(a,s) = V^*(s)\} = \arg\max_{a \in A(s)} Q^*(a,s).$$

The next theorem gives a characterization of optimal policies by the sets of optimal actions. It says that a policy is optimal if it chooses optimal actions only.

Theorem 21. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π be a policy for E. Then π is optimal if and only if*

$$\pi(a \mid s) > 0 \textit{ implies } a \in A^*(s) \quad \textit{for all } s \in S.$$

Proof. Suppose that there exists an $s \in S$ and an action $a \in A(s)$ such that $\pi(a \mid s) > 0$ and $a \notin A^*(s)$. Then

$$Q^*(a,s) < \max_{a \in A(\tilde{s})} Q^*(a,s) = V^*(s)$$

and

$$\sum_{a \in A(s)} Q^*(a,s)\pi(a \mid s) < \Big(\max_{a \in A(s)} Q^*(a,s)\Big) \sum_{a \in A(s)} \pi(a \mid s) = V^*(s).$$

By the preceding corollary π is not optimal. Let now π be a policy such that $\pi(a \mid s) > 0$ implies $a \in A^*(s)$, for $s \in S$. Then

$$\sum_{a \in A(s)} Q^*(a,s)\pi(a \mid s) = \sum_{a \in A^*(s)} V^*(s)\pi(a \mid s) = V^*(s), \quad \text{for } s \in S$$

and π is optimal by the above corollary. ■

Corollary 22. *Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π^* be an optimal policy for E. Then*

$$[\pi^*] = \prod_{s \in S} \operatorname{conv}(\delta_a : a \in A^*(s)) \subset \prod_{s \in S} C(s).$$

Given the optimal value function V^* we can compute the optimal action-values Q^* via Equation (1.32). If we construct a greedy policy π^* for Q^*, that is

$$\pi^*(- \mid s) = a \in \arg\max_{a \in A(s)} Q^*(a,s) \tag{1.36}$$

for each $s \in S$, then π^* is an optimal deterministic policy by the previous theorem.

1.11 Value Iteration

We give a short description of *value iteration*, an algorithm that approximates directly the optimal value function and optimal policies. For an extensive discussion of several aspects of the algorithm, its variants and references we refer to Puterman [Put94, pp. 158].

Recall the Bellman optimality equation (1.34) from the previous section

$$V^*(s) = \max_{a \in A(s)} \Big(R(a,s) + \gamma \sum_{s' \in S} V^*(s') P(s' \mid a, s) \Big).$$

We define the map $T \colon \mathbb{R}^S \to \mathbb{R}^S$ for $V \in \mathbb{R}^S$ and $s \in S$ by

$$(TV)(s) = \max_{a \in A(s)} \Big(R(a,s) + \gamma \sum_{s' \in S} V(s') P(s' \mid a, s) \Big). \tag{1.37}$$

We show that T is a contraction mapping with respect to the maximum norm. This implies that the optimal value function V^* is its unique fixed point.

Lemma 23. *Let $f, g \in \mathbb{R}^B$ with B a finite set. Then*

$$\Big| \max_b f(b) - \max_b g(b) \Big| \leq \max_b |f(b) - g(b)| .$$

Proof. We may assume without loss of generality that $\max_b f(b) \geq \max_b g(b)$. Let $b_1, b_2 \in B$ such that $f(b_1) = \max_b f(b)$ and $g(b_2) = \max_b g(b)$ respectively. Then

$$\Big| \max_b f(b) - \max_b g(b) \Big| = f(b_1) - g(b_2) \leq$$
$$f(b_1) - g(b_1) \leq \max_b |f(b) - g(b)| .$$

■

Let $V_1, V_2 \in \mathbb{R}^S$ and $s \in S$. With Equation (1.37), using the previous lemma and the triangle inequality we see that

$$|T(V_1)(s) - T(V_2)(s)| \leq \gamma \max_{a \in A(s)} \sum_{s'} |(V_1(s') - V_2(s'))| \, P(s' \mid a, s) \leq$$
$$\gamma \, \|V_1 - V_2\|_\infty \max_{a \in A(s)} \sum_{s'} P(s' \mid a, s) = \gamma \, \|V_1 - V_2\|_\infty .$$

Hence

$$\|T(V_1) - T(V_2)\|_\infty \leq \gamma \, \|V_1 - V_2\|_\infty$$

and T is a contraction mapping with contraction factor γ. From the Banach Fixed-Point Theorem for contraction mappings we know that the sequence defined by choosing $V_0 \in \mathbb{R}^S$ arbitrarily and

$$V_{n+1} = T(V_n) = T^n(V_0) \quad \text{for } n = 1, 2, \ldots \tag{1.38}$$

converges to V^*.

Given an approximation $V \in \mathbb{R}^S$ of the optimal value function V^*, we approximate the optimal action-values by $Q \in \mathbb{R}^{\mathbf{A}_S}$ with

$$Q(a, s) = R(a, s) + \gamma \sum_{s' \in S} V(s') P(s' \mid a, s),$$

for $(a, s) \in \mathbf{A}_S$, compare Equation (1.32). Then a *greedy policy* π *for* the *approximation* Q, that is

$$\pi(- \mid s) = a \in \arg\max_{a \in A(s)} Q(a, s) \tag{1.39}$$

for each $s \in S$, is an approximation of an optimal policy, compare Equation (1.36). The iterative algorithm derived from (1.38) followed by computing a greedy policy leads to *value iteration*, see Puterman [Put94, p. 161].

If the approximation V of the optimal value function is sufficiently good then the policy π is close to optimal. See Puterman [Put94, p. 161] for a precise statement in terms of ϵ-optimal policies. A policy π is called *ϵ-optimal* for $\epsilon > 0$ if

$$V^\pi(s) \geq V^*(s) - \epsilon, \quad \text{for } s \in S.$$

A Gauss-Seidel variant of value iteration, where updated values of the approximation are used as soon as they become available, is given in Algorithm 3. See Bertsekas [Ber95, pp. 28], Sutton and Barto [SB98, p. 102] and Puterman [Put94, p. 166]. For a discussion of asynchronous value iteration in general we refer to Bertsekas and Tsitsiklis [BT96, pp. 26].

The optimal action-values can also be interpreted as the fixed point of a contraction mapping. Recall the optimality equation (1.35) for the action-values. We define the map $\widetilde{T} \colon \mathbb{R}^{\mathbf{A}_S} \to \mathbb{R}^{\mathbf{A}_S}$ for $Q \in \mathbb{R}^{\mathbf{A}_S}$ and $(a, s) \in \mathbf{A}_S$ by

$$\widetilde{T}(Q)(a, s) = R(a, s) + \gamma \sum_{s' \in S} \max_{a' \in A(s')} Q(a', s') P(s' \mid a, s). \tag{1.40}$$

Then one sees as above that $\widetilde{T}$ is a contraction mapping with respect to the maximum norm and the optimal action-values Q^* its unique fixed point.

Input: $\epsilon > 0$
Output: an ϵ-optimal policy π
  initialize $V \in \mathbb{R}^S$ arbitrarily
  repeat
    $\Delta = 0$
    for all $s \in S$ **do**
      $v \leftarrow V(s)$
      $V(s) \leftarrow \max_{a \in A(s)} \Big(R(a,s) + \gamma \sum_{s' \in S} V(s') P(s' \mid a, s) \Big)$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  until $\Delta < \epsilon(1-\gamma)/2\gamma$
  for all $s \in S$ **do**
    $Q(a,s) \leftarrow R(a,s) + \gamma \sum_{s' \in S} V(s') P(s' \mid a, s)$ for all $a \in A(s)$
    choose $a \in \arg\max_{a \in A(s)} Q(a,s)$
    $\pi(- \mid s) \leftarrow a$

Algorithm 3: Value iteration

1.12 Model-free Methods

In this section we consider methods that approximate the value function and action-values and find optimal or suboptimal policies in MDPs, whenever there is no explicit model of the environment. Such methods are called *model-free* methods, compare Bertsekas and Tsitsiklis [BT96, pp. 179] and Kaelbling, Littman and Moore [KLM96, pp. 251].

Model-free methods are of major importance in the context of real world applications, where in general the dynamics of the environment can only be observed.

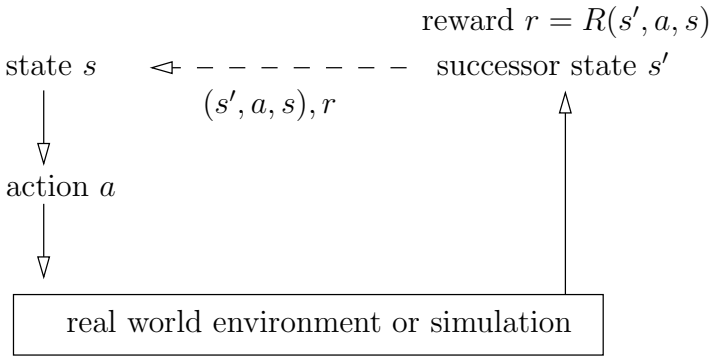

Figure 1.6: Interaction between agent and environment

Figure 1.6 summarizes the interaction between an agent and the (real world) environment or simulation, which provide the observations compare Sutton and Barto [SB98, p. 52]. We start in state s, apply an action a and observe the successor state s' and the reward $r = R(s', a, s)$. The resulting triple (s', a, s) and the reward r are then used to estimate the value function or action-values.

In the following sections we consider basic forms of different model-free algorithms. We first focus on methods to approximate the value-function and action-values for a given policy. Then we discuss approximate policy iteration and describe Q-learning to approximate the optimal action-values.

1.12.1 Temporal Differences

The term temporal difference learning was introduced by Sutton [Sut88], where he describes methods to approximate the value function for a policy. The main idea is to update the value function stepwise by using an observed estimate of it.

Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π be a policy for E and V^π the value function for policy π. Recall the Bellman equation (1.7)

$$V^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in S} V^\pi(s') P^\pi(s' \mid s), \quad \text{for } s \in S. \tag{1.41}$$

Suppose that $V \in \mathbb{R}^S$ is an approximation of the value function V^π. Let $s \in S$. We choose action $a \in A(s)$ according to policy π. Let $s' \in S$ denote the successor state observed after applying action a in state s and $r = R(s', a, s) \in \mathbb{R}$ the reward. Then r is an estimate of $R^\pi(s)$ and

$$r + \gamma V(s')$$

an estimate of the value function $V^\pi(s)$ in s derived from the current observation, compare Equation (1.41). The *temporal difference* is the difference between the estimate observed and the old estimate, that is

$$r + \gamma V(s') - V(s).$$

Based on the temporal difference we define a new estimate $\tilde{V} \in \mathbb{R}^S$ of the value function as

$$\tilde{V}(\tilde{s}) = \begin{cases} V(s) + \alpha(r + \gamma V(s') - V(s)), & \text{if } \tilde{s} = s, \\ V(\tilde{s}), & \text{otherwise,} \end{cases}$$

where α is a positive *step-size parameter* which influences the importance of the temporal difference and where $\tilde{V}$ differs from V in state s only. In other terms, we define the *update rule*

$$V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s)). \tag{1.42}$$

The *assignment operator* $\leftarrow$ means that we change V in state s only.

The algorithm derived from this update is a *stochastic approximation algorithm*, Bertsekas and Tsitsiklis [BT96, p. 133], or more specifically a Robbins-Monro stochastic approximation algorithm, Robbins and Monro [RM51]. We state Algorithm 4 usually called TD(0), refer to the bibliographical remarks Section 1.13.

Input: a policy π
Output: an approximation of V^π
  initialize V arbitrarily
  repeat
    choose $s \in S$
    choose $a \in A(s)$ according to π
    apply action a, observe s' and obtain r
    $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$

Algorithm 4: Temporal differences TD(0)

To state the stochastic convergence results for Algorithm 4 we introduce the following probability spaces and *random vectors*.

Let P be a probability distribution on S. One update step of the algorithm consists of choosing a state s with probability $P(s)$, choosing an action $a \in A(s)$ according to π and obtaining the successor state s'. The probability of obtaining the pair (s', s) is

$$P(s', s) = P^\pi(s' \mid s)P(s).$$

These experiments are repeated independently of each other.

Let $T \in \mathbb{N}_0$ and

$$\Omega_T = (S \times S)^{T+1}$$

denote the set of all sequences

$$\omega_T = (s'_T, s_T, s'_{T-1}, s_{T-1}, \ldots, s'_0, s_0) \in \Omega_T.$$

Since the experiments are independent, the probability of a sequence ω_T is

$$\mathbb{P}_T(\omega_T) = \prod_{t=0}^{T} P^\pi(s'_t \mid s_t)P(s_t)$$

Let

$$\Omega_\infty = (S \times S)^{\mathbb{N}}$$

denote the set of all infinite sequences

$$\omega = (\ldots, s'_T, s_T, s'_{T-1}, s_{T-1}, \ldots, s'_0, s_0) \in \Omega_\infty.$$

Let $\mathbb{P}_\infty$ be the unique probability on Ω_∞ with marginal probabilities $\mathbb{P}_T$, that is

$$\mathbb{P}_\infty \left\{\omega = (\ldots, s'_T, s_T, \omega_{T-1}) \in \Omega_\infty\right\} = \mathbb{P}_{T-1}(\omega_{T-1}),$$

for $T \in \mathbb{N}$ and $\omega_{T-1} \in \Omega_{T-1}$.

Let $V_0 \in \mathbb{R}^S$ be arbitrary, also considered as a constant random vector

$$V_0 \colon \Omega \to \mathbb{R}^S,\ \omega \mapsto V_0.$$

Inductively we define random vectors

$$V_{t+1} \colon \Omega \to \Omega_t \to \mathbb{R}^S$$

for $t = 0, 1, \ldots$ by

$$V_{t+1}(\omega_t)(s) = V_{t+1}(s'_t, s_t, \omega_{t-1})(s) =$$
$$\begin{cases} V_t(\omega_{t-1})(s_t) + \alpha_t(R^\pi(s) + \gamma V_t(\omega_{t-1})(s'_t) - V_t(\omega_{t-1})(s_t)), & \text{if } s = s_t, \\ V_t(\omega_{t-1})(s), & \text{otherwise,} \end{cases}$$

where α_t denotes the step-size parameter at iteration t.

Then the random vectors V_t converge to V^π with probability one provided that each state is chosen with a nonzero probability, that is $P(s) > 0$ for $s \in S$, and the step-size parameter decreases to zero under conditions explained below. The proof is based on convergence results for stochastic approximation methods for contraction mappings, compare Bertsekas and Tsitsiklis [BT96, pp. 199]. Recall that V^π is the fixed point of T^π, see Equation (1.9).

The rather difficult proof of convergence with probability one can for example be found in Bertsekas and Tsitsiklis [BT96, pp. 208]. The weaker result that the expectation $\mathbb{E}(V_t)$ of V_t converges to V^π is easily shown, see Sutton [Sut88].

The conditions for the decreasing step-size parameters α_t are

$$\sum_{t=1}^{\infty} \alpha_t = \infty \text{ and } \sum_{t=1}^{\infty} \alpha_t^2 < \infty. \tag{1.43}$$

The first condition allows the change in the update to be big enough to overcome any initial bias. The second condition ensures convergence. Compare Bertsekas and Tsitsiklis [BT96, p. 135] and Sutton and Barto [SB98, p. 39]. A usual choice for the step-size parameter that satisfies (1.43) is

$$\alpha_t = \frac{c}{t}, \quad \text{where } c \text{ is a positive constant.} \tag{1.44}$$

1.12.2 Approximate Policy Evaluation

In the model-free case we cannot compute the action-values directly from the value-function, since the rewards and transition probabilities are unknown. If we have an approximation of the action-values we can approximate the value-function, see the following section. We call methods to approximate the action-values for a given policy *approximate policy evaluation.*

Let $(E, \mathbf{R}, \gamma)$ be an MDP. Let π be a policy for E. Equation (1.16)

$$Q^\pi(a,s) = R(a,s) + \gamma \sum_{s' \in S} \sum_{a' \in A(s')} Q^\pi(a',s')\pi(a' \mid s')P(s' \mid a,s)$$

motivates the following update rule analogous to the TD(0) algorithm from the previous section.

Suppose that $Q \in \mathbb{R}^{\mathbf{A}_S}$ is an approximation of the action-values of Q^π. We start in state $s \in S$ and apply action $a \in A(s)$. Let $s' \in S$ denote the successor state and $r = R(s',a,s) \in \mathbb{R}$ the reward. Then r is an estimate of $R(a,s)$ and

$$r + \gamma \sum_{a' \in A(s')} Q(a',s')\pi(a' \mid s')$$

an estimate of the action-value $Q^\pi(a,s)$. We update the approximation of the action-value Q by the update rule

$$Q(a,s) \leftarrow Q(a,s) + \alpha(r + \gamma \sum_{a' \in A(s')} Q(a',s')\pi(a' \mid s') - Q(a,s)), \tag{1.45}$$

where α is a step-size parameter. The resulting algorithm is described in Algorithm 5.

Approximate policy evaluation and TD(0) are closely related and converge under the same conditions, since Q^π is the fixed point of the contraction mapping $\widetilde{T}^\pi$, see Equation (1.17).

We can define a probability space and random vectors Q_t related to the approximations Q generated by the algorithm similarly to the construction of V_t in the previous section. Then the random vectors Q_t converge to Q^π

Input: a policy π
Output: an approximation of Q^π
  initialize Q arbitrarily
  repeat
    choose $s \in S$
    choose $a \in A(s)$
    apply action a, observe s' and obtain r
    $Q(a,s) \leftarrow Q(a,s) + \alpha(r + \gamma \sum_{a' \in A(s')} Q(a',s')\pi(a' \mid s') - Q(a,s))$

Algorithm 5: Approximate policy evaluation

with probability one, provided that each state and each action is chosen with nonzero probability and the step-size parameter decreases to zero under conditions (1.43). Compare also Bertsekas and Tsitsiklis [BT96, p. 338].

1.12.3 Approximate Policy Iteration

Approximate policy iteration combines approximate policy evaluation with policy improvement following the idea of policy iteration, compare Section 1.9.

Let $(E, \mathbf{R}, \gamma)$ be an MDP and π be a policy for E. Let $Q \in \mathbb{R}^{\mathbf{A}_S}$ be an approximation of the action-values Q^π. To approximate the value function V^π we use the Bellman Equation (1.13) with approximation Q. Then $V \in \mathbb{R}^S$ with

$$V(s) = \sum_{a \in A(s)} Q(a,s)\pi(a \mid s), \quad \text{for } s \in S, \tag{1.46}$$

is an approximation of the value function V^π.

We define improving policies, strictly improving actions and improving vertices for policy π and approximation Q using the approximations instead of the exact action-values and value function. For example, we define the *improving policies* for policy π and *approximation* Q in state s by

$$C_{\geq}^{\pi,Q}(s) = \{\tilde{\pi}(- \mid s) : \sum_{a \in A(s)} Q(a,s)\tilde{\pi}(a \mid s) \geq V(s)\}$$

and the *strictly improving actions* for policy π and *approximation* Q in state s by

$$A_{>}^{\pi,Q}(s) = \{a \in A(s) : Q(a,s) > V(s)\}.$$

Note that $\pi(- \mid s) \in C_{\geq}^{\pi}(s)$ by Equation (1.46).

We start with an arbitrary policy and approximate the action-values with Algorithm 5 and the value function as described above. Then we compute

the set of strictly improving actions for the approximation and improve the policy. The resulting algorithm is given in Algorithm 6.

Output: a (sub)optimal policy π
  choose a starting policy π
  repeat
    compute approximation Q of Q^π
    compute V
    for all $s \in S$ **do**
      compute $A_>^{\pi,Q}(s)$
      if $A_>^{\pi,Q}(s) \neq \emptyset$ **then**
        choose $a \in A_>^{\pi,Q}(s)$
        $\pi(- \mid s) \leftarrow a$

Algorithm 6: Approximate policy iteration

In general approximate policy iteration does not converge to an optimal policy. Empirically, the policy improves well in the first iteration steps and then begins to oscillate around the optimal policy, compare Bertsekas and Tsitsiklis [BT96, pp. 282] and see Section 3.5.4 for computational experiments. The oscillation is constrained by error bounds discussed in Bertsekas and Tsitsiklis [BT96, pp. 275] and Munos [Mun03]. Perkins and Precup [PP02] give a convergent form of approximate policy iteration under special conditions. In practical applications the starting policy is usually chosen as good as possible, often through heuristic considerations.

1.12.4 Q-learning

Q-learning approximates the optimal action-values. It was introduced by Watkins [Wat89], who together with Dayan discussed convergence [WD92].

Let $(E, \mathbf{R}, \gamma)$ be an MDP. Equation (1.35) for the optimal action-values

$$Q^*(a,s) = R(a,s) + \gamma \sum_{s' \in S} \max_{a' \in A(s')} Q^*(a',s') P(s' \mid a,s)$$

motivates the following update rule.

Suppose that $Q \in \mathbb{R}^{\mathbf{A}_S}$ is an approximation of the optimal action-values of Q^*. We start in state $s \in S$ and apply action $a \in A(s)$. Let $s' \in S$ denote the successor state and $r = R(s',a,s) \in \mathbb{R}$ the reward. Then r is an estimate of $R(a,s)$ and

$$r + \gamma \max_{a' \in A(s')} Q(a',s')$$

an estimate of the optimal action-value $Q^*(a, s)$. We update the approximation of the optimal action-value Q by the update rule

$$Q(a, s) \leftarrow Q(a, s) + \alpha(r + \gamma \max_{a' \in A(s')} Q(a', s') - Q(a, s)). \tag{1.47}$$

The resulting algorithm is given in Algorithm 7.

Output: an approximation of Q^*
  initialize Q arbitrarily
  repeat
    choose $s \in S$
    choose $a \in A(s)$
    apply action a, observe s' and obtain r
    $Q(a, s) \leftarrow Q(a, s) + \alpha(r + \gamma \max_{a'} Q(a', s') - Q(a, s))$.

Algorithm 7: Q-learning

For an approximation Q of the optimal action-values Q^* the greedy policy π for Q, that is

$$\pi(- \mid s) = a \in \arg\max_{a \in A(s)} Q(a, s)$$

for each $s \in S$, is an approximation of an optimal policy, compare Equation (1.36).

We can define a probability space and random vectors Q_t related to approximations Q generated by the algorithm similarly to construction in Section 1.12.1. Then the random vectors Q_t converge to Q^* with probability one, provided that each state and each action is chosen with nonzero probability and the step-size parameter decreases to zero under conditions (1.43).

We refer to Tsitsiklis [Tsi94] and Bertsekas and Tsitsiklis [BT96, pp. 247] for a proof based on the fact that Q^* is the fixed point of the contraction mapping $\widetilde{T}$, see Equation (1.40). Further convergence results can be found in Jaakkola, Jordan and Singh [JJS94]. See also Sutton and Barto [SB98, p. 229] for a discussion of this algorithm in the context of planning.

1.13 Bibliographical Remarks

Ad Section 1.1: There is no standard terminology for policies, see Feinberg and Shwartz ([FS02, p. 4]). Puterman [Put94, p. 21] distinguishes between policies and decision rules. He uses the term Markovian randomized decision rules for what we call policies in state s. Compare also Filar and Vrieze [FV97, p. 51]. Hinderer [Hin70] discusses nonstationary dynamic programming, where in particular history and time

depending policies are considered.
In the reinforcement learning literature the terms *reward function* or *reinforcement function* are often used to describe the family of rewards, see Kaelbling, Littman and Moore [KLM96] and Santos [San99].

Ad Section 1.2.5: In the reinforcement learning literature action-values are often called *Q-values*. This is due to the impact of Q-learning, where action-values play a fundamental role, see also Sutton [Sut03] for a discussion on this terminology.

Ad Section 1.3.1: Sutton and Barto [SB98, p. 75] define a preorder over policies not taking into account the equivalence classes of policies.

Ad Section 1.3.2: The optimal value function is often directly defined by Equation (1.31). Optimal policies are then defined via the optimal value function, see Bertsekas and Tsitsiklis [BT96, p. 13], Feinberg and Shwartz [FS02, p. 23], Sutton and Barto [SB98, p. 75] or White [Whi93, p. 27]. This is equivalent to our definition.

Ad Section 1.4: For deterministic policies and without the notion of action-values the result on the difference of policies can be found in Howard [How65, p. 84]. Denardo [Den03, p. 167] and Kallenberg in [FS02, p. 34] use what we call the local difference for policy iteration with deterministic policies.

Ad Section 1.6: Kallenberg in [FS02, p. 34] defines and discusses improving actions. Sutton and Barto [SB98, p. 97] give a short description of policy improvement for stochastic policies. They suggest to improve a policy by a stochastic policy, where all submaximal actions are given zero probability. We describe all possible stochastic improving policies.

Ad Section 1.8.2: In game theory it is common to identify mixed strategies with points on a standard simplex, see for example von Neumann and Morgenstern [vNM47, pp. 143] or Ekeland [Eke74, p. 24]. Compare also Filar and Vrieze [FV97, pp. 343]. In contrast, in the theory of reinforcement learning and MDPs the interpretation of policies as points on standard simplices is not customary.

Ad Section 1.10: Denardo [Den03, p. 161] discusses the set of deterministic optimal policies.

Ad Section 1.12.1: We discuss TD(0) that looks ahead one step to compute the temporal difference. This method is a special case of temporal

difference methods TD(λ), see Bertsekas and Tsitsiklis [BT96, pp. 193] and Sutton and Barto [SB98, pp. 169]

Chapter 2

MDP Package

The `MDP` package for the computer algebra system Maple 9, `http://www.maplesoft.com`, is a collection of procedures for Markov decision processes (MDPs). Since Maple provides exact arithmetics all computations with the package can be done exactly, provided that the transition probabilities and rewards are rational numbers. Furthermore, symbolic computations can be performed.

For numerical computations with MDPs, see the matlab, `http://www.mathworks.com`, toolboxes:

- `http://www.ai.mit.edu/~murphyk/Software/MDP/mdp.html`
  by Kevin Murphy and
- `http://www.inra.fr/bia/T/MDPtoolbox`
  by M.-J. Cros, Frédérick Garcia and Régis Sabbadin.

The package implements the computation of value functions and action-values, (strictly) improving vertices and actions, and policy iteration. Functions to generate random transition probabilities and rewards and to plot improving polytopes are also provided. Help pages with examples are available. See Section 11.2 for a list of all functions with a short description and examples. The package with the help files, its source code, information on how to use it and an example worksheet can be found on the attached CD-ROM, see Section 11.3. The functions for several realizations are described in Section 7.

The following is a list of functions from the `MDP` package and the corresponding equations, theorems and algorithms from the previous section:

- `ValueFunction`, Equation (1.8).
- `ActionValues`, Equation (1.12).

- `IsOptimal`, Theorem 15 (v).
- `StrictlyImprovingActions`, `EquivalentVertices`, `ImprovingVertices`, Theorem 17.
- `PolicyImprovement`, `PolicyIteration`, Algorithm 2.

2.1 Transition Probabilities, Policies and Rewards

The following examples give an introduction to the main functions of the package and how to use them.

We make the functions from the `MDP` package available.

```
> restart;
> with(MDP):
```

We define transition probabilities with two states and two actions in each state.

```
> a:=2:  s:=2:
> P:=Array(1..s,1..a,1..s):
```

The first state

```
> P[1..s,1..a,1]:=<<3/4,1/4>|<1/3,2/3>>;
```

$$P_{1..2,1..2,1} := \begin{bmatrix} \frac{3}{4} & \frac{1}{3} \\ \frac{1}{4} & \frac{2}{3} \end{bmatrix}$$

and the second state.

```
> P[1..s,1..a,2]:=<<1/6,5/6>|<1/2,1/2>>;
```

$$P_{1..2,1..2,2} := \begin{bmatrix} \frac{1}{6} & \frac{1}{2} \\ \frac{5}{6} & \frac{1}{2} \end{bmatrix}$$

Are these valid transition probabilities?

```
> IsTransitionProbability(P);
```

true

Rewards

```
> R:=Array(1..s,1..a,1..s):
```

for the first

```
> R[1..s,1..a,1]:=<<1,-1>|<1,0>>;
```

$$R_{1..2,\,1..2,\,1} := \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$$

and the second state.

```
> R[1..s,1..a,2]:=<<0,1>|<0,-1>>;
```

$$R_{1..2,\,1..2,\,2} := \begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix}$$

Expected rewards for `P` and `R`.

```
> ER:=ExpectedReward(P,R);
```

$$ER := \begin{bmatrix} \frac{1}{2} & \frac{5}{6} \\ \frac{1}{3} & \frac{-1}{2} \end{bmatrix}$$

A policy

```
> Pol:=<<0,1>|<1/2,1/2>>;
```

$$Pol := \begin{bmatrix} 0 & \frac{1}{2} \\ 1 & \frac{1}{2} \end{bmatrix}$$

its expected rewards

```
> ERp:=ExpectedRewardPolicy(ER,Pol);
```

$$ERp := \left[\frac{1}{3}, \frac{1}{6}\right]$$

and transition matrix.

```
> Pp:=TransitionMatrix(P,Pol);
```

$$Pp := \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{2}{3} \end{bmatrix}$$

2.2 Value Function and Action-Values

A discount rate.

```
> ga:=1/2;
```

$$ga := \frac{1}{2}$$

The value function for policy `Pol` and discount rate `ga` is computed with the expected rewards `ERp` and the transition matrix `Pp`.

```
> Vp:=ValueFunction(ERp,Pp,ga);
```

$$Vp := \left[\frac{5}{9}, \frac{7}{18}\right]$$

To compute the action-values we need the transition probabilities `P`, the expected rewards `ER`, the value function `Vp` and the discount rate `ga`.

```
> Qp:=ActionValues(P,ER,Vp,ga);
```

$$Qp := \begin{bmatrix} \frac{109}{144} & \frac{25}{24} \\ \frac{5}{9} & \frac{-19}{72} \end{bmatrix}$$

The Bellman equation (1.13).

```
> [seq(LinearAlgebra:-Column(Pol,i).\
>          LinearAlgebra:-Column(Qp,i),i=1..s)];
```

$$[\frac{5}{9}, \frac{7}{18}]$$

2.3 Policy Improvement and Iteration

We improve policy `Pol`.

```
> imPol:=PolicyImprovement(Qp,Vp,Pol,'improved');
```

$$imPol := \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

Do we have a better policy?

```
> improved;
```

$$1$$

We compute the difference between the value functions of the improved and the original policy.

```
> Pimp:=TransitionMatrix(P,imPol):
> ERimp:=ExpectedRewardPolicy(ER,imPol):
> Vimp:=ValueFunction(ERimp,Pimp,ga);
```

$$Vimp := \left[\frac{19}{17}, \frac{27}{17}\right]$$

```
> Vimp-Vp;
```

$$\left[\frac{86}{153}, \frac{367}{306}\right]$$

Is the improved policy optimal?

```
> Qimp:=ActionValues(P,ER,Vimp,ga);
```

$$Qimp := \begin{bmatrix} \frac{19}{17} & \frac{27}{17} \\ \frac{107}{102} & \frac{3}{17} \end{bmatrix}$$

```
> IsOptimal(Qimp,Vimp);
```

$$true$$

We generate a random MDP with two states and three actions in each state for the policy iteration algorithm.

```
> s:=2:  a:=3:
> P:=RandomTransitionProbability(s,a,10):
> R:=RandomReward(s,a,-1,1):
> ER:=ExpectedReward(P,R);
```

$$ER := \begin{bmatrix} -1 & -1 \\ \frac{2}{5} & 1 \\ \frac{3}{10} & \frac{-4}{5} \end{bmatrix}$$

A random policy.

```
> Pol:=RandomStochasticMatrix(a,s,10);
```

$$Pol := \begin{bmatrix} \frac{1}{5} & \frac{4}{5} \\ \frac{1}{10} & \frac{1}{5} \\ \frac{7}{10} & 0 \end{bmatrix}$$

Now we compute an optimal policy with starting policy `Pol`.

```
> optPol:=PolicyIteration(P,ER,Pol,ga,'steps');
```

$$optPol := \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}$$

How many improvement steps were used to obtain this optimal policy?

```
> steps;
```

$$2$$

2.4 Improving Policies

We compute the strictly improving actions for policy `Pol` in the first state. The value function and action-values.

```
> ERp:=ExpectedRewardPolicy(ER,Pol):
> Pp:=TransitionMatrix(P,Pol):
> Vp:=ValueFunction(ERp,Pp,ga);
```

$$Vp := \left[\frac{-3}{35}, \frac{-29}{35}\right]$$

```
> Qp:=ActionValues(P,ER,Vp,ga);
```

$$Qp := \begin{bmatrix} \frac{-73}{70} & \frac{-443}{350} \\ \frac{73}{350} & \frac{23}{25} \\ \frac{51}{350} & \frac{-321}{350} \end{bmatrix}$$

Action-values in the first state.

```
> Qps:=LinearAlgebra:-Column(Qp,1);
```

$$Qps := \begin{bmatrix} \frac{-73}{70} \\ \frac{73}{350} \\ \frac{51}{350} \end{bmatrix}$$

```
> StrictlyImprovingActions(Qps,Vp[1]);
```

$$[2,\ 3]$$

The equivalent and improving vertices.

```
> EquivalentVertices(Qps,Vp[1]);
```

$$\left[\left[\begin{array}{c} \frac{103}{438} \\ \frac{335}{438} \\ 0 \end{array}\right], \left[\begin{array}{c} \frac{81}{416} \\ 0 \\ \frac{335}{416} \end{array}\right]\right]$$

```
> ImprovingVertices(Qps,Vp[1]);
```

$$\left[\left[\left[\begin{array}{c} 0 \\ 1 \\ 0 \end{array}\right], \left[\begin{array}{c} 0 \\ 0 \\ 1 \end{array}\right]\right], \left[\left[\begin{array}{c} \frac{103}{438} \\ \frac{335}{438} \\ 0 \end{array}\right], \left[\begin{array}{c} \frac{81}{416} \\ 0 \\ \frac{335}{416} \end{array}\right]\right]\right]$$

We plot the polytope of improving policies for the two states. The red dot represents the policy, the blue area the set of improving policies and the thick black line the set of equivalent policies.

```
> for i from 1 to s do
>    improvingpolicyplot3d(ImprovingVertices(\
>    LinearAlgebra:-Column(Qp,i),Vp[i]),\
>    LinearAlgebra:-Column(Pol,i));
> end do;
```

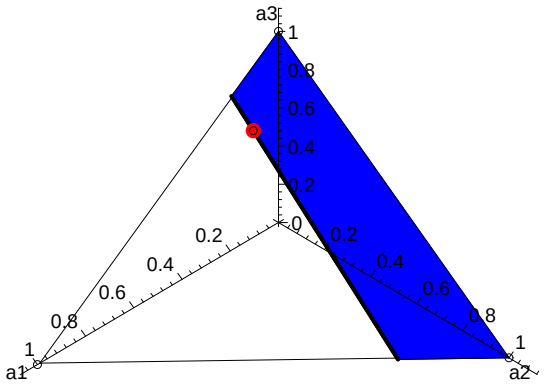

Figure 2.1: Improving Policies for policy `Pol` in state 1

And after one step of policy improvement.

```
> imPol:=PolicyImprovement(Qp,Vp,Pol);
```

$$imPol := \left[\begin{array}{cc} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{array}\right]$$

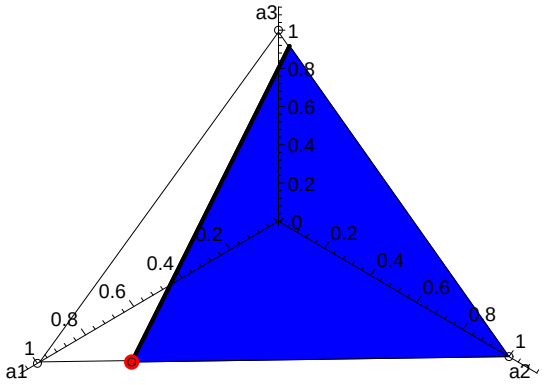

Figure 2.2: Improving Policies for policy `Pol` in state 2

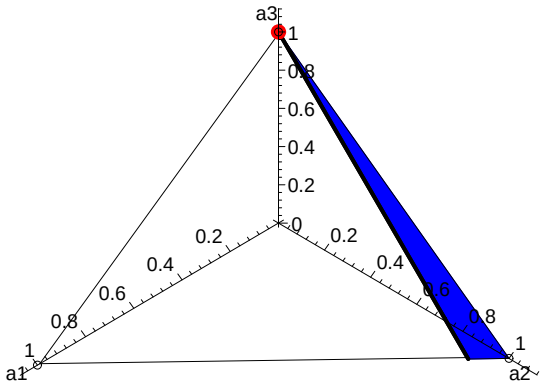

Figure 2.3: Improving Policies for policy `imPol` in state 1

```
> Pimp:=TransitionMatrix(P,imPol):
> ERimp:=ExpectedRewardPolicy(ER,imPol):
> Vimp:=ValueFunction(ERimp,Pimp,ga):
> Qimp:=ActionValues(P,ER,Vimp,ga):
> for i from 1 to s do
>    improvingpolicyplot3d(ImprovingVertices(\
>    LinearAlgebra:-Column(Qimp,i),Vimp[i]),\
>    LinearAlgebra:-Column(imPol,i));
> end do;
```

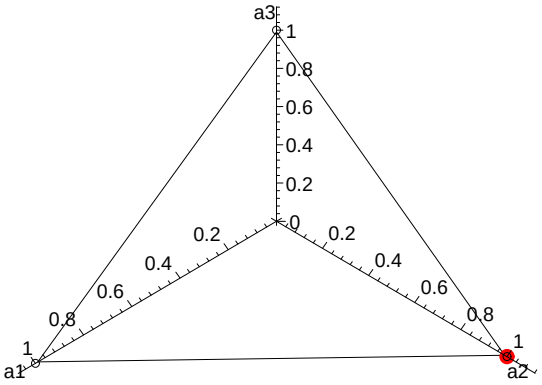

Figure 2.4: Improving Policies for policy `imPol` in state 2

2.5 Symbolic Computations

We give two examples of how to use symbolic computations with the MDP package.

2.5.1 Discount Rate

First we compute the value function of a policy with the discount rate as a variable. We use the MDP from the first example with two states and actions.

```
> a:=2:  s:=2:
> P:=Array(1..s,1..a,1..s):
> P[1..s,1..a,1]:=<<3/4,1/4>|<1/3,2/3>>:
> P[1..s,1..a,2]:=<<1/6,5/6>|<1/2,1/2>>:
> R:=Array(1..s,1..a,1..s):
> R[1..s,1..a,1]:=<<1,-1>|<1,0>>:
> R[1..s,1..a,2]:=<<0,1>|<0,-1>>:
> ER:=ExpectedReward(P,R):
```

A policy.

```
> Pol:=<<0,1>|<1/2,1/2>>;
```

$$Pol := \begin{bmatrix} 0 & \frac{1}{2} \\ 1 & \frac{1}{2} \end{bmatrix}$$

```
> ERp:=ExpectedRewardPolicy(ER,Pol):
```

```
> Pp:=TransitionMatrix(P,Pol):
```

The discount rate as variable

```
> ga:=g;
```

$$ga := g$$

and we obtain the value function.

```
> Vp:=ValueFunction(ERp,Pp,ga);
```

$$Vp := \left[\frac{-3+2\,g}{9\,(-1+g)} - \frac{g}{9\,(-1+g)}, -\frac{g}{9\,(-1+g)} + \frac{-3+g}{18\,(-1+g)}\right]$$

2.5.2 Same Action-Values and different Value Functions

Now we want to construct a simple MDP and two policies with the same action-values and different value functions. Again we consider two states and two action in each state.

```
> a:=2:  s:=2:
> P:=Array(1..s,1..a,1..s):
```

We choose symmetric transition probabilities.

```
> P[1..s,1..a,1]:=<<1/2|1/2>,<1/2|1/2>>;
```

$$P_{1..2,\,1..2,\,1} := \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

```
> P[1..s,1..a,2]:=<<1/2|1/2>,<1/2|1/2>>:
```

Rewards with variables `r1, r2, r3, r4`.

```
> R:=Array(1..s,1..a,1..s):
> R[1..s,1..a,1]:=<<r4|1>,<1|r1>>;
```

$$R_{1..2,\,1..2,\,1} := \begin{bmatrix} r4 & 1 \\ 1 & r1 \end{bmatrix}$$

```
> R[1..s,1..a,2]:=<<1|r3>,<r2|1>>;
```

$$R_{1..2,\,1..2,\,2} := \begin{bmatrix} 1 & r3 \\ r2 & 1 \end{bmatrix}$$

```
> ER:=ExpectedReward(P,R):
```

A first (deterministic) policy and its action-values.

```
> POL1:=<<0,1>|<1,0>>;
```

$$POL1 := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

```
> Pp1:=TransitionMatrix(P,POL1);
```

$$Pp1 := \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

```
> ERp1:=ExpectedRewardPolicy(ER,POL1):
> ga:=1/2:
> Vp1:=ValueFunction(ERp1,Pp1,ga):
> Qp1:=ActionValues(P,ER,Vp1,ga);
```

$$Qp1 := \begin{bmatrix} 1+\frac{1}{2}\,\overline{r4}+\frac{r1}{4}+\frac{r2}{4} & 1+\frac{1}{2}\,\overline{r2}+\frac{r1}{4}+\frac{r2}{4} \\ 1+\frac{1}{2}\,\overline{r1}+\frac{r1}{4}+\frac{r2}{4} & 1+\frac{1}{2}\,\overline{r3}+\frac{r1}{4}+\frac{r2}{4} \end{bmatrix}$$

And a second policy

```
> POL2:=<<1,0>|<0,1>>;
```

$$POL2 := \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

with the same transition matrix.

```
> Pp2:=TransitionMatrix(P,POL2);
```

$$Pp2 := \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

```
> ERp2:=ExpectedRewardPolicy(ER,POL2):
> Vp2:=ValueFunction(ERp2,Pp2,1/2):
> Qp2:=ActionValues(P,ER,Vp2,ga);
```

$$Qp2 := \begin{bmatrix} 1+\frac{1}{2}\,\overline{r4}+\frac{r4}{4}+\frac{r3}{4} & 1+\frac{1}{2}\,\overline{r2}+\frac{r4}{4}+\frac{r3}{4} \\ 1+\frac{1}{2}\,\overline{r1}+\frac{r4}{4}+\frac{r3}{4} & 1+\frac{1}{2}\,\overline{r3}+\frac{r4}{4}+\frac{r3}{4} \end{bmatrix}$$

We see that if

```
> r1+r2=r3+r4;
```

$$r1 + r2 = r3 + r4$$

then the action-values of the two policies are equal. Furthermore, we want the two value functions to be different. We set for example the value function of the first policy in the first state

```
> Vp1[1]=0;
```

$$1 + \frac{3\,r1}{4} + \frac{r2}{4} = 0$$

and the value function of the second policy in second the state.

```
> Vp2[2]=0;
```

$$1 + \frac{r4}{4} + \frac{3\,r3}{4} = 0$$

Then we have three linear equations for four unknowns. We solve the system

```
> sol:=solve({r1+r2=r3+r4,Vp1[1]=0,Vp2[2]=0});
```

$$sol := \{r4 = -4 - 3\,r3,\ r1 = r3,\ r2 = -4 - 3\,r3,\ r3 = r3\}$$

and choose `r1=r3` such that the value function for the second policy in the first state is not zero.

```
> Vp2[1];
```

$$1 + \frac{3\,r4}{4} + \frac{r3}{4}$$

```
> subs({r4=-4-3*r1,r3=r1},%);
```

$$-2 - 2\,r1$$

For example

```
> r1:=0; r3:=0;
```

$$r1 := 0$$
$$r3 := 0$$

and

```
> r4:=-4-3*r1; r2:= -4-3*r1;
```

$$r4 := -4$$
$$r2 := -4$$

We have found rewards and policies with the property that the action-values are equal and the value functions different. We check our solution.

```
> R[1..2,1..2,1];
```

$$\begin{bmatrix} -4 & 1 \\ 1 & 0 \end{bmatrix}$$

```
> R[1..2,1..2,2];
```

$$\begin{bmatrix} 1 & 0 \\ -4 & 1 \end{bmatrix}$$

```
> ER:=ExpectedReward(P,R):
> ERp1:=ExpectedRewardPolicy(ER,POL1);
```

$$ERp1 := \left[\frac{1}{2}, \frac{-3}{2}\right]$$

```
> ERp2:=ExpectedRewardPolicy(ER,POL2);
```

$$ERp2 := \left[\frac{-3}{2}, \frac{1}{2}\right]$$

```
> Vp1:=ValueFunction(ERp1,Pp1,1/2);
```

$$Vp1 := [0, -2]$$

```
> Vp2:=ValueFunction(ERp2,Pp2,1/2);
```

$$Vp2 := [-2, 0]$$

```
> Qp1:=ActionValues(P,ER,Vp1,1/2);
```

$$Qp1 := \begin{bmatrix} -2 & -2 \\ 0 & 0 \end{bmatrix}$$

```
> Qp2:=ActionValues(P,ER,Vp2,1/2);
```

$$Qp2 := \begin{bmatrix} -2 & -2 \\ 0 & 0 \end{bmatrix}$$

Chapter 3

SimRobo

To illustrate the theory and test the algorithms described and proposed in this thesis we developed a simple grid world simulator called `SimRobo`. It implements the algorithms introduced in Section 1 for discounted MDPs: policy evaluation, policy iteration, and value iteration, Algorithms 1, 2, 3, and the model-free methods approximate policy evaluation and iteration and Q-learning, Algorithms 5, 6, 7. The implementation of algorithms for several environments is described in Section 8.

We have used `SimRobo` for publications, talks and public presentations, where it turned out to be a useful didactic tool. We also wrote a general tutorial on reinforcement learning in connection with the simulator. The states, actions, environments and rewards in `SimRobo` can be easily adapted for different tasks. The executable program and the source code can be found on the attached CD-ROM, see Section 11.3. In the following we give an introduction to `SimRobo`, its specifications, implementation details and computational experiments.

3.1 The Robot and its World

3.1.1 Grid World

`SimRobo` is based on simple rectangular grid worlds. Each field of the grid either is empty or holds an obstacle. Figure 3.1 shows three 10×10 sample grid worlds.

The grid worlds are stored in files, ending with the extension `.env`. They can be edited and modified with any text editor. The first two parameters stored in the file are the length and width of the grid world, then a matrix follows, where the value of zero represents an empty field and one an obstacle.

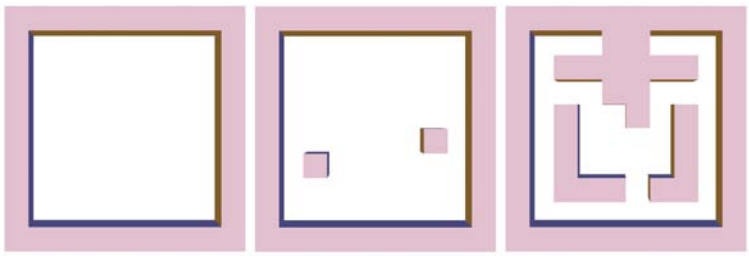

Figure 3.1: Three grid worlds

The file for the second grid world of Figure 3.1 is, for example:

```
10 10
1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 0 1
1 0 1 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1
```

3.1.2 Directions and Coordinates

The simulated robot is represented by a filled hexagonal cone. The cone points in the direction where the robot is heading. In our case the robot heads to one of four possible *directions* d, encoded as follows

- direction $d = 1$, *upwards,*
- direction $d = 2$, *downwards,*
- direction $d = 3$, *left,*
- direction $d = 4$, *right.*

The robot's *positions* are represented by the robot's coordinates and the direction. Let $(x, y) \in \mathbb{N}^2$ be the coordinates where the robot is located, with $(0, 0)$ being the upper left obstacle and $(1, 1)$ the upper left corner. Let d

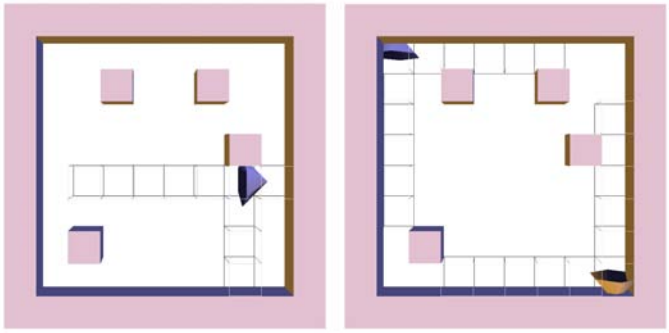

Figure 3.2: *left:* The robot and its sensors *right:* Same sensor values and different positions

be the direction the robot is heading. Then a position is represented by the triple (x, y, d). In Figure 3.2 the robot faces right being in coordinates $(7, 5)$ and thus the position is $(7, 5, 4)$.

3.1.3 Sensors

The robot has a predefined number of sensors. We implement four sensors: front, left, right and back. The sensors measure the distance to the obstacles within a predefined range. We use a fixed number of five blocks maximum range for each sensor. The sensor values vary between zero and five, where a value of five means that there is a block right in front and a zero value means that there is no block in sight. Thus we encode the sensors by

$$(s_0, s_1, s_2, s_3), \quad \text{with } s_i = 0, \ldots, 5,$$

where s_0 is the front, s_1 the left, s_2 the right and s_3 the back sensor. See Figure 3.2 *left*, where the forward sensor is four, the left sensor is five, the right sensor is two and the back sensor is zero, represented by $(4, 5, 2, 0)$.

The same sensor values can result from different robot positions in the environment. For an example in Figure 3.2, take the position $(1, 1, 1)$ that represents the robot being in the upper left corner looking upwards. This position shares the same sensor values $(5, 5, 0, 0)$ with the position $(8, 8, 2)$ of looking downwards in the lower right corner.

3.1.4 Actions

The robot has a predefined number of actions in each position. We consider the same three actions a in all positions, encoded as follows:

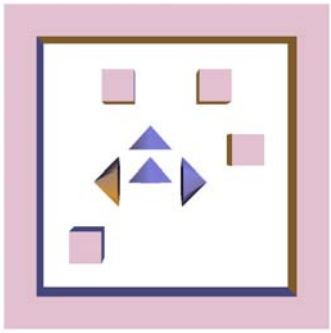

Figure 3.3: The robot and its actions

- action $a = 0$, *move one block forward,*
- action $a = 1$, *turn left and move one block forward,*
- action $a = 2$, *turn right and move one block forward.*

Note that the result of an action depends on the start position, that is the coordinates and direction. Figure 3.3 shows the result of each action, starting in a position looking upwards. In the case that there is an obstacle blocking the field where the action is heading, the robot bumps into the wall and does not move, but it may turn.

3.2 Rewards

We consider two basic behavior we want the robot to learn, *obstacle avoidance* and *wall following*. The rewards depend on the sensor values $s = (s_0, s_1, s_2, s_3)$ of the starting position, the applied action a and the sensor values $s' = (s'_0, s'_1, s'_2, s'_3)$ of the resulting position. Three different values are used: reward $+1$, punishment -1, and neutral reinforcement 0. In general it is sufficient to have two different rewards only, see [Mat00, p. 31] and Santos, Matt and Touzet [SMT01].

3.2.1 Obstacle Avoidance

The robot should move around while avoiding obstacles. If the robot stays far away from obstacles it gets rewarded, if it bumps into an obstacle it gets a punishment. Additionally, if it moves away from obstacles it will be rewarded, if it moves towards obstacles it will be punished and in any other

case it gets a neutral reinforcement. The formal description of the rewards R_{oa} for obstacle avoidance is given by:

$$R_{oa}(s', a, s) = \begin{cases} +1, & \text{if } s'_i \leq 3, \text{ for } i = 0, \ldots, 3, \\ -1, & \text{if } s_i = 5 \text{ and } a = i, \text{ for } i = 0, 1, 2, \end{cases}$$

and in any other case

$$R_{oa}(s', a, s) = \begin{cases} +1, & \text{if } \sum_i (s'_i - s_i) \leq -1, \\ -1, & \text{if } \sum_i (s'_i - s_i) > 0, \\ 0, & \text{otherwise.} \end{cases}$$

3.2.2 Wall Following

The robot should follow the wall, represented by a sequence of obstacles, with its right side while not bumping into obstacles. We punish the robot if it bumps into an obstacle. If its right side is close to the wall it gets rewarded, otherwise punished.

$$R_{wf}(s', a, s) = -1, \quad \text{if } s_i = 5 \text{ and } a = i, \text{ for } i = 0, 1, 2,$$

and in any other case

$$R_{wf}(s', a, s) = \begin{cases} +1, & \text{if } s'_2 = 5, \\ -1, & \text{otherwise.} \end{cases}$$

3.3 Environments and MDPs

We describe how we model the robot in a grid world as an MDP. We define states, actions and discuss how the transition probabilities are derived from the grid world to obtain an environment. We consider two different environments: one based on positions (coordinates and directions) and one based on sensors. The family of rewards $\mathbf{R}$ is defined by obstacle avoidance, R_{oa}, or wall following, R_{wf}, from the previous section.

3.3.1 Positions

We consider an $m \times n$ grid world. The positions of the robot in the grid world define the set of states, that is a subset S of all possible positions

$$\{(x, y, d) : x = 0, \ldots, m-1, \ y = 0, \ldots, n-1 \text{ and } d = 1, \ldots, 4\}.$$

The sets of actions

$$A(s) = \{0, 1, 2\} \quad \text{for all } s \in S$$

define the family $\mathbf{A} = (A(s))_{s \in S}$ of actions. The transition probabilities can be computed from the grid world following the definition of actions in Section 3.1.4. If the robot applies an action $a \in A(s)$ in state $s = (x, y, d) \in S$ it gets to the unique successor state $s' = (x', y', d') \in S$. This data gives rise to deterministic transition probabilities $\mathbf{P} = (P(- \mid a, s))_{s \in S, a \in A(s)}$, with

$$P(\tilde{s}' \mid \tilde{a}, \tilde{s}) = \begin{cases} 1, & \text{if } \tilde{s} = s,\ \tilde{s}' = s' \text{ and } \tilde{a} = a, \\ 0, & \text{otherwise.} \end{cases}$$

The transition probabilities completely describe the dynamics of the grid world. We obtain the environment $E = (S, \mathbf{A}, \mathbf{P})$ and the *position-based MDP* $(E, \mathbf{R}, \gamma)$ with a discount rate γ.

3.3.2 Sensors

The sensor values in all positions of the robot in a grid world define the set of states, that is a subset S of all possible sensor values

$$\{(s_0, s_1, s_2, s_3) : \ s_i = 0, \ldots, 5 \text{ for } i = 1, \ldots, 4\},$$

The sets of actions

$$A(s) = \{0, 1, 2\} \quad \text{for all } s \in S$$

define the family $\mathbf{A} = (A(s))_{s \in S}$ of actions. Recall that different positions may have the same sensor values. Thus applying an action $a \in A(s)$ in state $s = (s_0, s_1, s_2, s_3) \in S$ may lead to different successor states $s' = (s'_0, s'_1, s'_2, s'_3) \in S$.

For example consider $s = (5, 5, 0, 0)$ and action $a = 2$ to move to the right, see Figure 3.2 *right*. If the robot is in the upper left corner looking upwards, that is position $(1, 1, 1)$, the successor state is $(0, 5, 0, 4)$. If it is in the lower right corner looking downwards, that is position $(8, 8, 2)$, the successor state is $(0, 5, 2, 4)$.

If we fix a probability on all positions, then we can compute the transition probabilities for sensor-based states. We assume a uniform distribution over all positions and construct the transition probabilities as follows. For each possible position (x, y, d) we compute its sensor values $s = s(x, y, d) \in S$, apply each action $a \in A(s)$ and observe the successor position (x', y', d') and

its successor sensor values $s' \in S$. We denote the number of times that we observed s by n_s and the number of times that s' was observed when action a was applied in s by $n_{s'|(a,s)}$. We set

$$P(s' \mid a, s) = \frac{n_{s'|(a,s)}}{n_s}.$$

The construction gives an environment $E = (S, \mathbf{A}, \mathbf{P})$ and we obtain the *sensor-based MDP* $(E, \mathbf{R}, \gamma)$.

The transition probabilities depend on the probability distribution over all possible positions. So, the MDP we construct is not an exact model of the sensor-based grid world. As an interpretation of the constructed MDP we can say that the robot switches between positions that are represented by the same sensor values.

A more appropriate model for this situation is a partially observable MDP (*POMDP*), where only stochastic observations of an underlying MDP are available, see Kaelbling, Littman and Cassandra [KLC98], Meuleau et al. [MKKC99] and Cassandra [Cas03]. In our example the underlying MDP is the position-based MDP and the (deterministic) observations are the sensor values.

3.4 Implementation

3.4.1 Methodology

We applied several aspects of the software development method *extreme programming*, an agile methodology that favors informal and immediate design and test techniques over detailed specifications and planning, see `http://www.extremeprogramming.org` and `http://www.agilealliance.com`.

In particular, the technique of pair programming was used for the implementation of `SimRobo`. Two programmers work together on one computer. One programmer controls keyboard and mouse and actively implements the program and the other programmer identifies errors and thinks about the overall program design. If required, the programmers brainstorm or switch roles.

Williams et al. [WKCJ00] report that pair programming increases efficiency and quality. For a case-study of extreme and pair programming and its adaptation for scientific research see Wood and Kleb [WK03]. Our experience with pair programming is that, apart from producing better designed code for complex problems faster and with fewer errors, it is truly enjoyable.

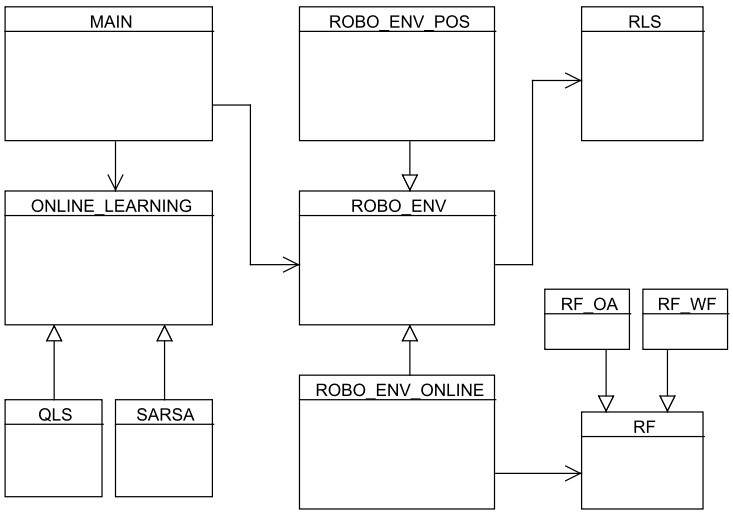

Figure 3.4: `SimRobo` class structure for one MDP

3.4.2 Programming Details

`SimRobo` is implemented in `C++` using object-oriented techniques and is compatible with the `ISO` Standard, see Stroustrup [Str97]. We worked with the `Visual C++ 6.0` development environment. To keep the code small and efficient the container classes and algorithms provided by the standard library are used throughout the program, see Stroustrup [Str97, pp. 427]. In particular for vector operations we rely on the class `valarray`, specifically designed for speed of numeric computations [Str97, pp. 662].

For 3D graphics we used `OpenGL`, see Section 3.4.6 for details. The program has approximately 7000 lines of source code. In the following sections, we describe the class structure, classes and member functions of `SimRobo`. Figure 3.4 shows all classes and their relations in a UML-diagram. Refer to `http://www.uml.org` for an introduction to UML (unified modeling language).

3.4.3 Reinforcement Learning System

The computations for MDPs are done by the class `RLS` (reinforcement learning system), which is designed for fast computations. Within this class states and actions are enumerated and transition probabilities and expected rewards are represented by `valarrays`.

For efficiency reasons, three different types of policies are implemented. Deterministic policies are represented by a `valarray` of `integers`. Extended

policies additionally include possible random actions, that is, all actions are chosen with the same probability. Finally, stochastic (probabilistic) policies are represented by a `valarray` of `double` values.

The optimal value function is computed by `value_iteration`, see Algorithm 3. The expected rewards and transition probabilities for a given policy can be computed for a deterministic, extended or stochastic policy respectively with the functions `rew_dpol`, `prob_dpol`, `rew_epol`, `prob_epol`, `rew_ppol` and `prob_ppol`. The function `pol_eval` computes the value function for a policy with the expected rewards and transition probabilities for a policy, see Algorithm 1.

The action-values and a greedy policy are computed by the functions `qval` and `pol_improv`. The optimal value function, optimal actions and an optimal policy are accessed by functions `optval`, `optact` and `optpol`. The discounted utility of the optimal and random policy by functions `dutilopt` and `dutilran` respectively. For details consult the header file Listing 1.

3.4.4 Environments and Rewards

The main objective of the class `robo_env` is the interaction between robot, grid world and the underlying MDP. It contains information concerning the grid world and its transition probabilities, it includes functions to compute the representations of states, actions and transition probabilities for the class `RLS`. Furthermore, helper functions for the different types of policies are provided.

The grid world is stored in a member variable and can be loaded and displayed in simple text format. The function `perform_a` computes the successor position (coordinates and direction) for a given position and action in the grid world. The sensor values of a given position are provided by the function `get_se`. The functions `setos` and `stose` convert the sensor values to a natural number and vice versa.

The function `pos_dir` decides whether a position is possible to reach with the defined actions or not. For example a position where the back of the robot is touching the wall is only reachable if the robot stands in a corner. It is used by the function `random_position` that generates random coordinates and a random direction.

Value iteration and policy evaluation are run by calling the appropriate functions in the class `RLS`, that are `value_iteration`, `dpol_eval`, `epol_eval`, `ppol_eval`, `ppol_eval`, and `xepol_eval`.

For position-based MDPs, see Section 3.3.1, we implemented the class `robo_env_pos` derived from `robo_env`. It provides a function to map positions to states and different constructors to initialize all necessary data.

The abstract class `RF` to compute rewards is implemented by the classes `RF_OA` for obstacle avoidance and `RF_WF` for wall following as described in Section 3.2. Since we use function objects, see Stroustrup [Str97, p. 514], different rewards can be easily added. They just have to be derived from the class `RF` and overload the `operator()`, that returns a `double` value given the state-action-state triple (s', a, s). Consult the header files Listings 2 and 3 for further details.

3.4.5 Model-free Methods

In the class `online_learning` action-values play an important role. An approximation of action-values is a member of the class. A greedy policy for an approximation is computed by the function `get_greedypolicy`. An approximation of the value function is computed from an approximation of the action-values and a given policy by the function `get_values`, see Equation (1.46). The action-values are initialized with zero values by the function `initialize`.

Derived from `online_learning` the classes `SARSA` and `QLS` perform the updates of the model-free methods approximate policy evaluation and Q-learning respectively by implementing the `update` rule, see Equations (1.45) and (1.47). The class `SARSA` contains two different updates, one for deterministic policies and one for stochastic policies. In the class `QLS` an `update` function for deterministic policies is available. See Listings 5, 6 and 7.

To compare results of the model-free methods and exact computations we implemented the class `robo_env_online` for sensor-based MDPs. It is derived from the class `robo_env` and additionally provides functions to deal with the classes `SARSA` and `QLS`. Approximate policy evaluation and Q-learning for a given number of update steps can be done by the functions `sarsa_poleval` and `qlearning`, see Algorithms 5 and 7. States and actions are chosen with respect to the uniform distribution over the set of states and actions. See Listing 8.

3.4.6 Graphics and User Interface

The 3D graphics is implemented using the `OpenGl` utility toolkit `GLUT`, a window system independent programming interface for `OpenGL`. `GLUT` contains window management, callback and idle routines, basic functions to generate various solid and wire frame objects, and it supports bitmap fonts, see Kilgard [Kil96].

In the `main` function of `SimRobo` first the window specifications are given, then a general lighting and two variable lights are enabled. The 3D to 2D

perspective projection is set and the main loop function called. The graphics are displayed in a double-buffered window.

The `GLUI` graphical user interface based on `GLUT`, see Rademacher [Rad99], takes care of the keyboard and mouse interaction with the user and provides a series of different buttons and controllers. The simulated robot can be steered with the keyboard, keys 8, 4 and 6 for forward, left and right by first clicking with the mouse into the grid world window or using the appropriate buttons in the lower control panel. The lower control panel contains a rotation button to change the angle to look at the grid world and a rotation button to change the light settings. The grid world can be zoomed and moved using translation buttons. `Start Position` sets the default angle and coordinates of the grid world. With the radio buttons `Sensors` and `Positions` the mutually exclusive options of using the sensor-based MDP or the position-based MDP can be set.

The right control panel provides buttons to change the displayed grid world and current policies, it contains advanced options and control buttons to run algorithms for one environment. In the left control panel the equivalent buttons to run algorithms for several environments can be found. They are explained in Section 8.3. Figure 3.5 displays the `SimRobo` window with all three control panels. The controls are very useful to set and change options to conduct the different experiments. In the next section most of the control buttons are shortly described.

3.5 Experiments

In the following experiments we consider 10×10 grid worlds. We set the discount rate $\gamma = 0.95$. In all experiments first the grid world and rewards are fixed, then the MDP is derived, see Section 3.3. Except for the first experiment, all of them are conducted using the sensor-based MDP. The experiments can be repeated using `SimRobo` and following the instructions given in each experiment.

To evaluate and compare policies we use the utility of a policy, see Section 1.3.3. We define the *random policy* π^{rand}, which chooses all actions with the same probability in each state, by

$$\pi^{\text{rand}}(a \mid s) = \frac{1}{|A(s)|}, \quad s \in S \text{ and } a \in A(s).$$

We compute the utilities $V(\pi^*)$ and $V(\pi^{\text{rand}})$ of the optimal policy π^* and the random policy π^{rand}. We normalize the utility of an arbitrary policy π

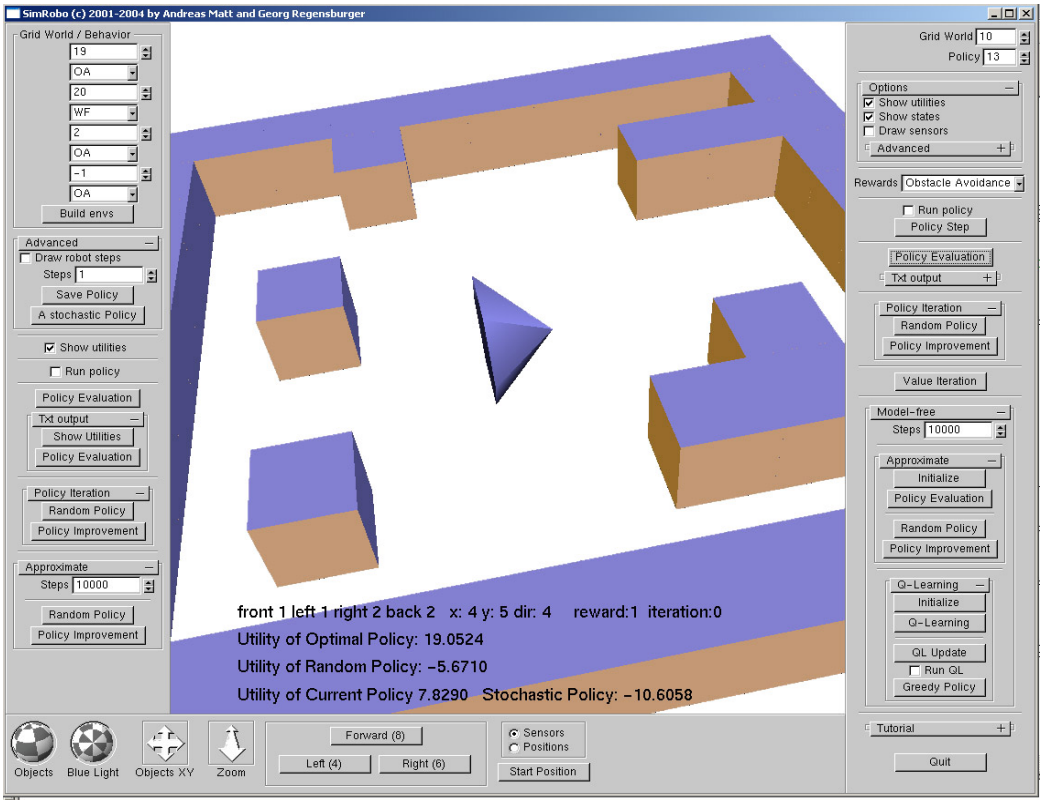

Figure 3.5: `SimRobo` and its control panels

with 1 being the utility of an optimal policy and 0 of the random policy. We denote the *normalized utility* by

$$\overline{V}(\pi) = \frac{V(\pi) - V(\pi^{\text{rand}})}{V(\pi^*) - V(\pi^{\text{rand}})}.$$

To observe the progress of the algorithms we display the normalized utilities of the obtained policies. The precision of policy evaluation and value iteration is set to $\epsilon = 10^{-8}$.

3.5.1 Wall Following

We want the robot to follow the wall in the grid world displayed in Figure 3.6 *left*. The rewards for wall following are defined in Section 3.2.2. First we derive the position-based MDP. We start with the random policy. Then policy iteration is run by consecutively evaluating and improving the policy, see Algorithms 2 and 1. We choose a greedy policy in the improvement step,

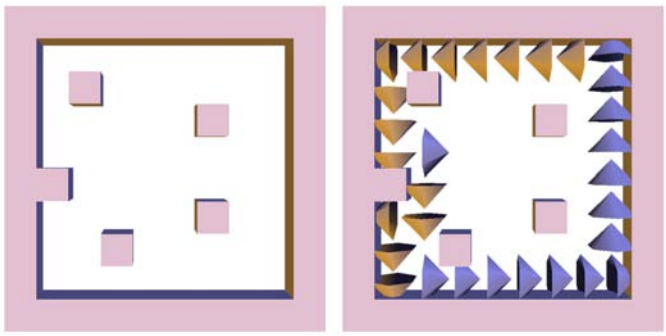

Figure 3.6: *left:* Grid world to learn wall following *right:* The robot following an optimal policy

see Equation (1.30). Refer to the experiment in Section 8.3.6, where policy improvement is run using a randomly chosen strictly improving action in the improvement step. Figure 3.7 *left* shows the normalized utility of the policy after each improvement step. After four iterations the policy is optimal where one iteration consists of policy evaluation and improvement.

Figure 3.6 *right* shows the robot following the optimal policy obtained. Wherever the robot starts, it moves to the wall and follows the displayed loop, where only positive rewards are experienced.

Now we derive the sensor-based MDP and repeat the above process. Figure 3.7 *right* shows the normalized utility of the policy after each improvement step. The policy is optimal after four iterations.

To repeat the experiment with `SimRobo` use the buttons on the right and lower control panel. Choose `Grid World` number 5 and `Wall Following` as `Rewards`. Put the checkerbox on `Positions` or `Sensors` depending on the MDP you want to use. Select `Show utilities` in the `Options` rollout. Open the `Policy Iteration` rollout and press `Random Policy` to set the current policy to the random policy. Now press the `Policy Improvement` and `Policy Evaluation` buttons consecutively while observing the utility of the current (improved) policy until it gets optimal. If you want to see the robot following the current policy select `Run policy`. To set the speed of the moving robot change `Speed` in the `Advanced` rollout. To see the progress of the utilities you can redirect the output of the utility to the console window by pressing `Policy Evaluation` in the `Console Output` rollout.

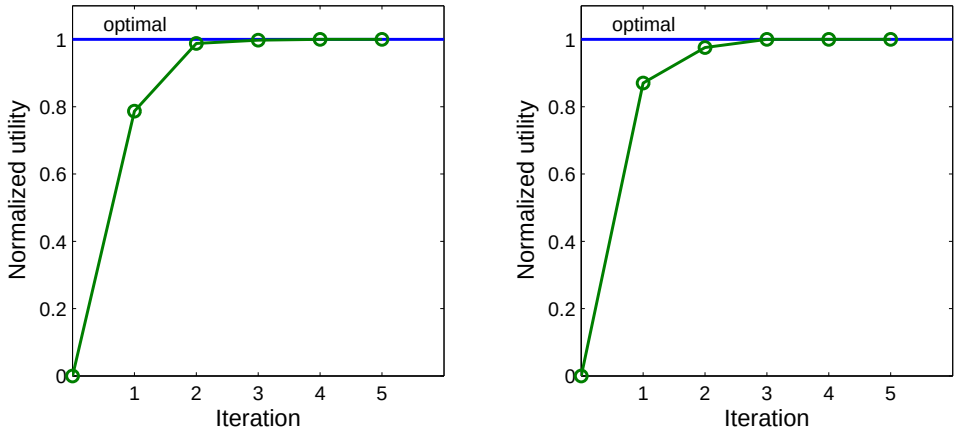

Figure 3.7: *left:* Progress of policy iteration in the position-based MDP *right:* Progress in the sensor-based MDP

3.5.2 Obstacle Avoidance

The robot should learn to avoid obstacles in the grid world displayed in Figure 3.8 *left* for the derived sensor-based MDP. The rewards for obstacle avoidance are defined in Section 3.2.1. As described above we start with the random policy and repeat policy evaluation and policy improvement. The following table shows the utility and normalized utility of the policy after each improvement step. After one improvement step the policy is almost optimal, after two it is optimal.

	utility	normalized
optimal	19.172	1.0
random	−3.489	0.0
iteration 1	19.154	0.992
iteration 2	19.172	1.0

Figure 3.8 *right* shows the robot following an optimal policy. Wherever the robot starts, it moves to the center of the grid world. There are different loops where the robot gathers only positive rewards.

In `SimRobo` this experiment can be repeated similarly to the previous experiment. Just choose `Grid World` number 3 and `Obstacle Avoidance` as `Rewards` and leave the checkerbox on `Sensors`. As an alternative to policy iteration, that is policy evaluation and policy improvement, value iteration can be run to directly obtain an optimal policy by pressing the button `Value Iteration`, see Algorithm 3.

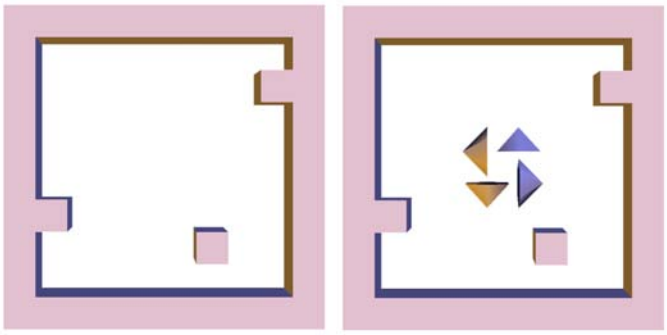

Figure 3.8: *left:* Grid world to learn obstacle avoidance *right:* The robot following an optimal policy

3.5.3 Approximate Policy Evaluation

We apply approximate policy evaluation to evaluate a policy in the model-free case, see Algorithm 5. We choose a number of update steps n and set the step-size parameter α_t for iteration t,

$$\alpha_t = 1 - \frac{n}{t}, \quad \text{for } t = 0, \ldots, n-1,$$

see Section 1.12.1. We use this step-size choice in all model-free experiments.

To compare value functions we use the maximum norm and the L_1-*norm* on $\mathbb{R}^S$, that is

$$\|x\|_1 = \sum\nolimits_{s \in S} |x(s)|, \quad \text{for } x \in \mathbb{R}^S.$$

Let π be a policy and V^π its value function. Let V be an approximation of V^π. We call $\|V^\pi - V\|_\infty$ the *maximum error* and $\|V^\pi - V\|_1$ the L_1 *error* of approximation V.

We set the grid world and rewards as in the previous experiment. We choose the random policy π^{rand} and approximate its action-values. The algorithm is run with 10000 update steps. We obtain an approximation Q of the action-values $Q^{\pi^{\text{rand}}}$. We compute an approximation of the value function by Equation (1.46), and display its error. Figure 3.9 shows the maximum and L_1 error in twelve experiments. Note that there are 149 states in the derived MDP.

The experiment is repeated with different numbers of update steps: 1000, 5000, 10000 and 50000. In Figure 3.10 the average maximum error of twelve experiments for each number of update steps is shown. We see that the approximation gets closer to the exact value function if the number of update steps is increased.

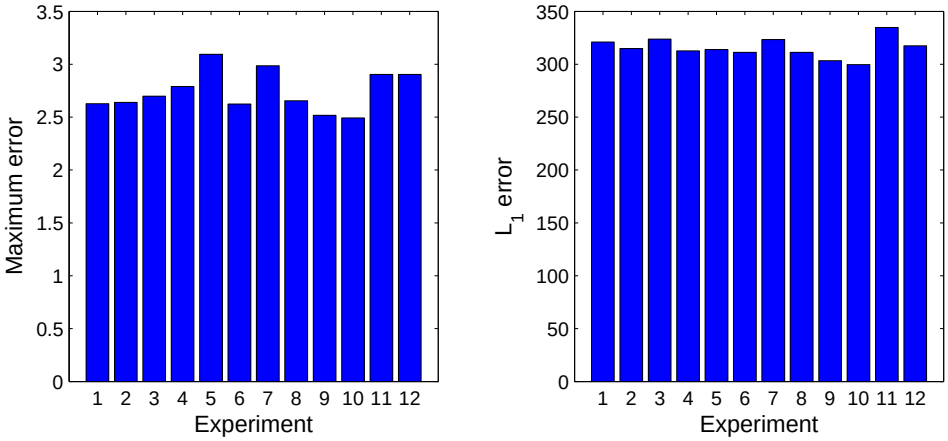

Figure 3.9: *left:* Maximum error for approximate policy evaluation with 10000 update steps *right:* L1 error

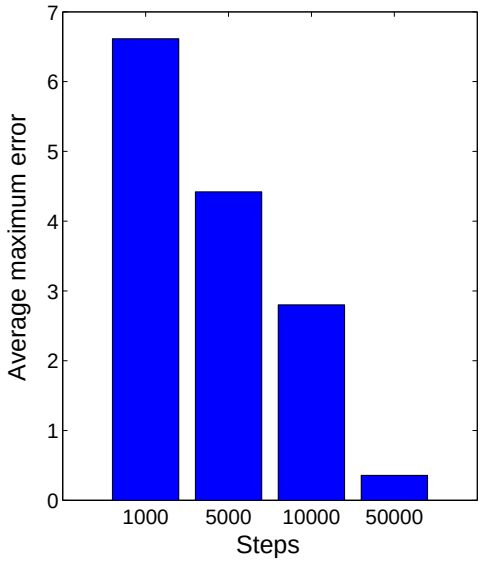

Figure 3.10: Average maximum error for approximate policy evaluation

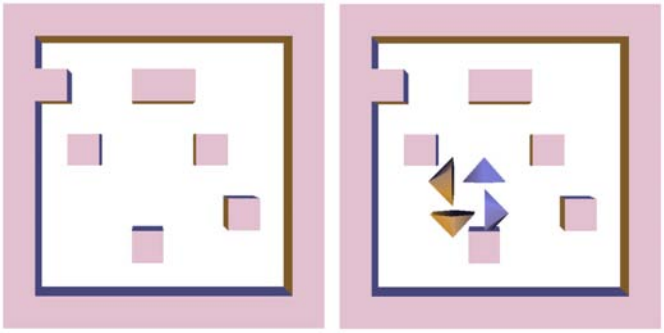

Figure 3.11: *left:* Grid world to learn obstacle avoidance *right:* The robot following an optimal policy

To carry out this experiment with `SimRobo` choose `Grid World` number 3 and `Obstacle Avoidance` as `Rewards`. Leave the checkerbox on `Sensors`. Open the rollouts `Model-free` and `Approximate` and press `Initialize` to set the approximation of the action-values to zero and `Random Policy`. Choose the desired number of `steps` and press the `Policy Evaluation` button in the `Approximate` rollout. The maximum and L_1 error can be observed in the console window. To display the number of states in the current grid world press the button `Grid World` in the `Console Output` rollout.

3.5.4 Approximate Policy Iteration

Obstacle avoidance in the grid world displayed in Figure 3.11 *left* should be learned in the model-free case. Approximate policy iteration is run by consecutively applying approximate policy evaluation and policy improvement, see Algorithm 6.

We carry out three experiments with different numbers of update steps for approximate policy evaluation. First we use 500 and 1000 update steps. We compute the utility of the obtained policy after each improvement step. Figure 3.12 shows the progress of approximate policy iteration. The policy may become worse after an improvement step depending on the error of the approximation of the action-values. If we increase the number of update steps for approximate policy evaluation, the obtained policies are almost optimal after few iterations. The oscillations around the optimal policy can be observed in Figure 3.13 for 10000 and 50000 update steps, compare Section 1.12.3.

To repeat the experiment with `SimRobo`, choose `Grid World` number 7 and `Obstacle Avoidance` as `Rewards`. Leave the checkerbox on `Sensors`.

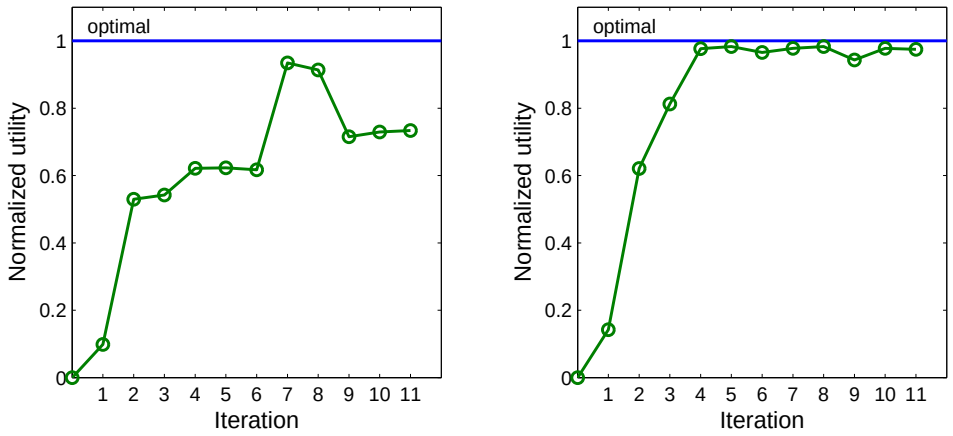

Figure 3.12: *left:* Progress of approximate policy iteration for 500 update steps *right:* Progress for 1000 update steps

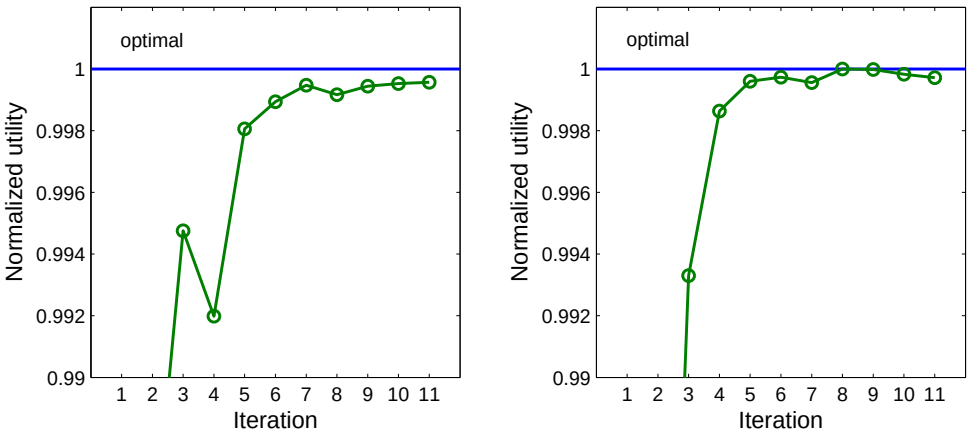

Figure 3.13: *left:* Progress of approximate policy iteration for 10000 update steps *right:* Progress for 50000 steps

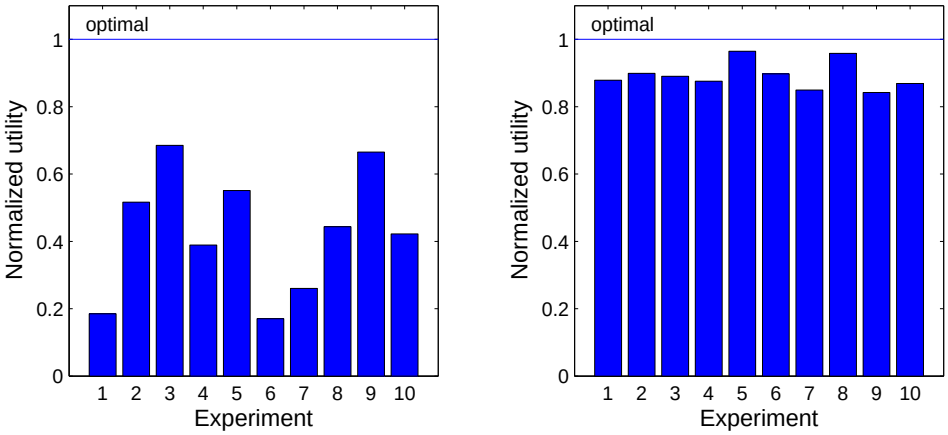

Figure 3.14: *left:* Performance of the obtained policies for Q-learning with 500 update steps *right:* Performance with 1000 update steps

Open the rollouts `Model-free` and `Approximate`, press `Initialize` to set the approximation of the action-values to zero and `Random Policy` to set the current policy to the random policy. Choose the desired number of `steps` for approximate policy evaluation.

Select `Show utilities` in the `Options` rollout. Press the button `Policy Improvement` in the `Approximate` rollout to run approximate policy evaluation and then improve the policy. To observe the exact utility of the current (improved) policy press the `Policy Evaluation` button above the `Console Output` rollout, not the one in the `Approximate` rollout. Repeat this process. If you want to see the robot following the current policy select `Run policy`. To see the progress of the utilities you can redirect the output of the utility to the console window by pressing `Policy Evaluation` in the `Console Output` rollout.

3.5.5 Q-learning

We want the robot to learn wall following in the grid world displayed in Figure 3.6 left, as in the first experiment. The Q-learning algorithm is run, see Algorithm 7. We experiment with four different numbers of update steps. First ten sample runs with 500 and 1000 update steps are conducted. Figure 3.14 shows the normalized utilities of the obtained policy for each experiment. We see that the policies perform much better than the random policy and are already very good after 1000 update steps. Then ten sample runs with 10000 and 50000 update steps are carried out. We see that the utilities of

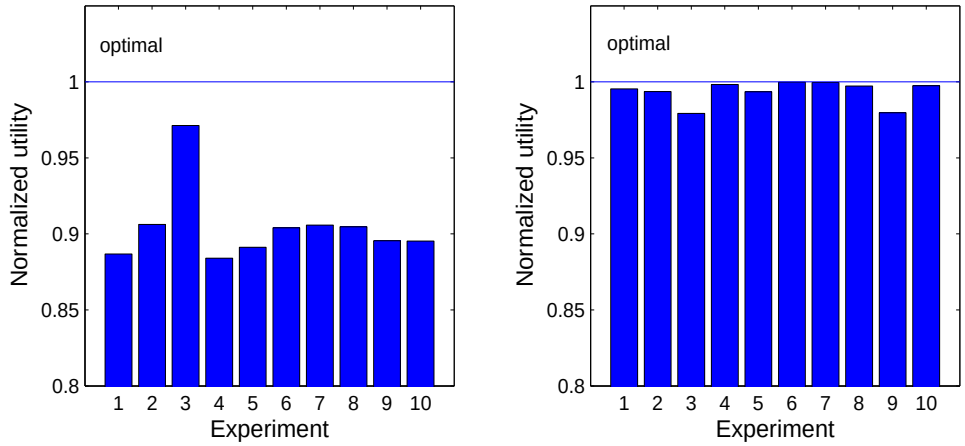

Figure 3.15: *left:* Performance of the obtained policies for Q-learning with 10000 update steps *right:* Performance with 50000 update steps

the obtained policies increase and are almost optimal for 50000 update steps, see Figure 3.15.

To repeat the experiment with `SimRobo` choose `Grid World` number 5 and `Wall Following` as `Rewards`. Leave the checkerbox on `Sensors`. Select `Show utilities` in the `Options` rollout. Open the rollouts `Model-free` and `Q-learning` and press `Initialize` and `Random Policy`. Choose the desired amount of `steps` and press the `Q-learning` button. The obtained policy can be evaluated with the button `Policy Evaluation`. The output of the utilities can be redirected to the console window by pressing `Policy Evaluation` in the `Console Output` rollout.

Chapter 4

RealRobo

It is effective to validate control algorithms using a simulator, but the simplifications involved are too restrictive for the results to be conclusive in real world applications. Therefore experiments in the real world are essential to show the success of control algorithms. An affordable approach is to conduct experiments using miniature robots.

We use the mobile robot Khepera to test the proposed theory. We experienced that practical applications require a considerable amount of trial and error and often success is obtained using methods or a combination of methods whose properties and interrelations are not fully understood. Intuition and heuristic considerations as well as manual parameter tuning play an important role. Thus the sections on `RealRobo` emphasize a specific implementation and ideas rather than formal methods that can be applied generally.

We implemented the program `RealRobo`. Similarly to `SimRobo` it is implemented in `C++` and uses the `OpenGL` graphics interface and the `GLUI` user interface. `RealRobo` contains functions to control the robot and implements adaptations of the model-free algorithms approximate policy iteration and Q-learning, see Algorithms 6 and 7. Approximate policy iteration and an implementation for several realizations are described in Section 9.

4.1 The Mobile Robot Khepera and its World

4.1.1 The Robot

Khepera is a miniature mobile robot produced by the Swiss company *K-Team*, `http://www.k-team.com`, see Figure 4.1 *left.* It has a cylindric shape, is 55 mm in diameter, 30 mm high and weighs approximately 70 g. Two

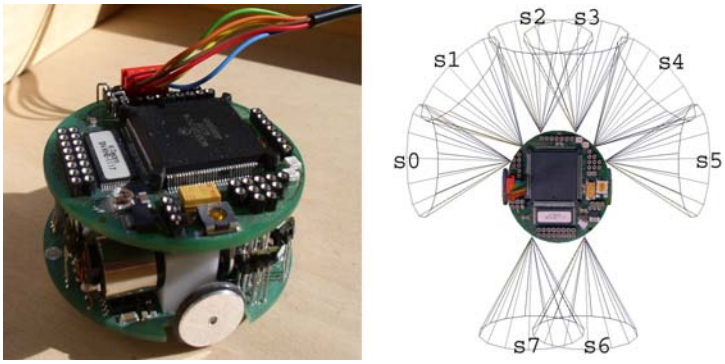

Figure 4.1: *left:* The mobile robot Khepera *right:* Range and distribution of the eight proximity sensors

Figure 4.2: Three sample wooden boxes

wheels that can be controlled independently allow the robot to move around. Their speed is set by integer values between -127 and 127 with a maximum speed of one meter per second. It is equipped with analogue infrared proximity sensors to measure the distance to obstacles.

The robot can act autonomously by means of a pre-compiled program or can be connected, via the `Rs232` serial port, to a computer, where a program to interact with the robot is installed. The program contains the learning algorithms. It manages the memory needed and reacts in real-time by receiving data from the robot and sending back commands the robot should carry out. See Mondada, Franzi and Ienne [MFI94] for technical details of the robot.

To conduct experiments we put the robot in a *wooden box*, that can be filled with obstacles, see Figure 4.2 for three sample boxes.

4.1.2 Sensors

Eight infrared proximity sensors can detect obstacles up to a range of 50 mm. The values received are between 0 and 1023 and then normalized between 0.0 and 1.0. Thus the set of possible sensor values is

$$\{0, 1/1023, \ldots, 1022/1023, 1\}.$$

The value represents the distance to the perceived obstacle, where a value of 0 means that there is no obstacle in sight and 1 that there is an obstacle right in front.

The infrared detectors are sensitive to color and reflection of the obstacle and to the general light configuration. Figure 4.1 *right* shows a graphic of the distribution, enumeration and range of the six front sensors $s_0, \ldots, s_5$ and the two back sensors s_6 and s_7. The sensors differ from each other in terms of range and activation sensibility. Note that different physical positions of the robot in a wooden box may have the same sensor values, compare Section 3.1.3.

4.1.3 Motors

The robot can move around and turn by means of two wheels. If one wheel turns forward and the other backward with the same speed, the robot turns around its center. If both wheels turn forward or backward with the same speed, the robot moves straight forward or backward respectively. To control the robot we consider only these two basic movements, to turn by a specific angle and to move a specific distance straight forward or backward.

4.2 Rewards

We want the robot to move around while avoiding obstacles. If it moves away from obstacles it will be rewarded, if it gets too close to an obstacle it will be punished, and in any other case it gets a neutral reinforcement. We put the robot in a position and observe the values of the eight proximity sensors $s_0, \ldots, s_7$ measured in this position. Then the robot conducts a movement and gets to a successor position where we observe the sensor values $s'_0, \ldots, s'_7$. The formal description of the rewards R_{oa} for obstacle avoidance is then given by:

$$R_{oa}(s', s) = \begin{cases} +1, & \text{if } \sum_{i=0}^{7}(s_i - s'_i) > 0.04, \end{cases}$$

and in any other case

$$R_{oa}(s', s) = \begin{cases} -1, & \sum_{i=0}^{7} s'_i > 0.94, \\ 0, & \text{otherwise.} \end{cases}$$

These rewards for obstacle avoidance are due to Santos [San99, p. 90].

4.3 States, Actions and Transition Probabilities

We describe how we model the robot in its world as an MDP. We define states and actions and discuss how the transition probabilities are provided by the wooden box.

The values of the eight proximity sensors of the robot in a wooden box define the set of states, that is a subset S of all possible values

$$\{(s_0, ..., s_7) : s_i \in \tilde{S} \quad \text{for } i \in 0 \dots 7\},$$

where $\tilde{S} = \{0, 1/1023, \dots, 1022/1023, 1\}$. In every state we consider the same actions, that consist of turning by an integer angle between -90 and 90 degrees, and then moving forward a fixed distance of approximately 4 mm. Hence the set of actions is

$$A(s) = \{a : a \in -90 \dots 90\} \quad \text{for all } s \in S.$$

The family of rewards $\mathbf{R}$ is defined by obstacle avoidance R_{oa} from the previous section.

In theory, if we fix a probability on all physical positions of the robot then transition probabilities can be derived from the wooden box by putting the robot into the physical positions given by the distribution, observing the sensor values, applying the actions and observing the successor positions and their sensor values. Compare Section 3.3.2 where transition probabilities are deduced from the grid world. This process turns out to be infeasible since it is difficult to measure the physical position of the robot and since there are too many different states and actions, approximately 10^{24} states and 181 actions.

If we consider model-free methods only, we do not need an explicit representation of the transition probabilities. The transitions can be observed in sample steps. Similarly the possible states $s \in S$ are observed while interacting.

4.4 Q-learning

In the following two sections we discuss a variant of Q-learning, Section 1.12.4, and a method to apply it for large state and action spaces. The considerations can be applied generally in the model-free case. They are presented based on our robot experiments.

The goal is to compute a good approximation Q of the optimal action-value function Q^* and then to derive an approximation π of an optimal deterministic policy π^* according to Equation (1.39), that is by

$$\pi(- \mid s) = a \in \arg\max_{a \in A(s)} Q(a, s), \quad \text{for } s \in S. \tag{4.1}$$

Note that V defined by

$$V(s) = \max_{a \in A(s)} Q(a, s)$$

is an approximation of the optimal value function V^* by Equation (1.33). The online approximation of Q^* is also called the *learning phase*.

One experimental run of the robot is described by a theoretically infinite sequence

$$\omega = (\ldots, a_{t+1}, s_{t+1}, a_t, s_t, \ldots, s_1, a_0, s_0)$$

of states

$$s_t = (s_0^t, \ldots, s_7^t) \in S \subset [0, 1]^8$$

and actions $a_t \in A(s_t)$. At the discrete time step t action a_t is applied to state s_t to yield the next state s_{t+1}.

Let Ω be the space of all sequences ω. Approximate action-value vectors $Q_t \in \mathbb{R}^{\mathbf{A}S}$ are computed at each time step t starting with $Q_0 = 0$. The sequence Q_t is then defined inductively by the update rule

$$Q_{t+1}(a, s) = \begin{cases} Q_t(a_t, s_t) + \\ \alpha_t(r_t + \gamma \max_{a' \in A(s_{t+1})} Q_t(a', s_{t+1}) - Q_t(a_t, s_t)), & \text{if } (a, s) = (a_t, s_t), \\ Q_t(a, s), & \text{otherwise,} \end{cases} \tag{4.2}$$

where r_t is the reward and $\alpha_t > 0$ the step-size parameter at time t, compare Equation (1.43). The maps

$$Q_t : \Omega \to \mathbb{R}^{\mathbf{A}S}, \ \omega \mapsto Q_t, \tag{4.3}$$

are *random vectors* depending on the past history

$$(a_{t-1}, s_{t-1}, \ldots, a_0, s_0)$$

only. When a_t is applied to s_t the probability $P(s_{t+1} \mid a_t, s_t)$ is completely determined by the environment, that is, the wooden box.

The choice of a_{t+1} depends on the current approximation of the optimal action-values Q_t and a random term. The dependence of the chosen actions on the approximation allows the robot to use current knowledge during learning and is called *exploitation*. The modification by a random term is called *exploration*, see Thrun [Thr92] and Bertsekas and Tsitsiklis [BT96, p. 251].

Let $\tilde{\pi}_{t+1}(- \mid s_{t+1})$ be a greedy policy for the current approximation Q_{t+1} in state s_{t+1}, that is

$$\tilde{\pi}_{t+1}(- \mid s_{t+1}) = a \in \arg \max_{a \in A(s_{t+1})} Q_{t+1}(a, s_{t+1}). \tag{4.4}$$

Exploration signifies that $\tilde{\pi}_{t+1}(- \mid s_{t+1})$ is changed randomly, for instance by

$$\pi_{t+1}(- \mid s_{t+1}) = (1 - \beta_t)\tilde{\pi}_{t+1}(- \mid s_{t+1}) + \beta_t \frac{1}{|A(s_{t+1})|}, \tag{4.5}$$

with $0 < \beta_t \leq 1$ and where the β_t form a zero-sequence. The next action a_{t+1} is chosen according to $\pi_{t+1}(- \mid s_{t+1})$. Note that $\pi_{t+1}(- \mid s_{t+1})$ becomes a greedy policy for the approximation Q_{t+1} for large t.

In our experiment we specifically add a random number $b \in [-b_t, b_t]$ to a greedy action $a \in -90 \ldots 90$ where $b_t > 0$ is a zero-sequence and take the action $a_{t+1} \in -90 \ldots 90$ which is nearest to $a + b$. In any case we obtain a probability $\pi_{t+1}(- \mid s_{t+1})$ on $A(s_{t+1})$ according to which the action a_{t+1} is selected.

The choice of the random disturbance of a current greedy action and hence of $\pi_{t+1}(- \mid s_{t+1})$ implies a probability P_Ω on the space Ω which by the Kolmogorov extension theorem is uniquely determined by the marginal probabilities

$$\begin{gathered} P_\Omega(s_{t+1} \mid a_t, s_t, \ldots, a_0, s_0) = P(s_{t+1} \mid a_t, s_t),\ P_\Omega(s_0) > 0 \\ P_\Omega(a_{t+1} \mid s_{t+1}, a_t, s_t, \ldots, a_0, s_0) = \pi(a_{t+1} \mid s_{t+1}). \end{gathered} \tag{4.6}$$

The latter conditional probability depends on Q_{t+1} and therefore on the whole history $(s_{t+1}, a_t, s_t, \ldots, a_0, s_0)$ and hence Ω is not a *Markov chain*.

Convergence of Q-learning according to Algorithm 7 has been proved if the state-action pairs (a_t, s_t) are chosen according to a pre-selected probability P with $P(a, s) > 0$ for all pairs $(a, s) \in \mathbf{A}_S$. This choice signifies that all

pairs (a, s) appear with positive probability among the (a_t, s_t). Exploration is chosen to imitate this behavior in the situation of the robot experiment.

As far as we know the $P_\Omega-$almost sure convergence of the random vectors Q_t to the optimal action-value Q^* has not yet been established, but is an interesting and important problem. Compare also Singh et al. [SJLS00]. Our robot experiments suggest that this convergence takes place, and we hence assume this convergence in the following considerations. Hence

$$\lim_{t\to\infty} \max_{a\in A(s)} Q_{t+1}(a, s) = \max_{a\in A(s)} Q^*(a, s) = V^*(s), \quad \text{for } s \in S$$

and a greedy policy with respect to Q_{t+1} becomes an optimal policy for large t. Since action a_{t+1} is chosen greedily for Q_{t+1} for large t we have

$$a_{t+1} \in A^*(s_{t+1}), \quad \text{for large } t,$$

where

$$A^*(s) = \{a \in A(s) : Q^*(a, s) = V^*(s)\}$$

is the set of optimal actions in a state s, see Section 1.10. In other terms, for large t

$$(s_t, a_t, Q_t(a_t, s_t)) \sim (s_t, a^*, V^*(s_t)), \quad \text{with } a^* \in A^*(s_t). \tag{4.7}$$

Thus if we run the experiment for a long time the chosen actions become optimal actions.

4.5 Data Compression by Clustering

Approximation methods are needed to implement learning methods for large state and action spaces. See Bertsekas and Tsitsiklis [BT96, pp. 59] for an introduction and overview. In our experiment we have $2^{80} \sim 10^{24}$ states

$$s = (s_0, \ldots, s_7) \in S \subset [0, 1]^8$$

and 181 actions in $A(s) = \{-90, \ldots, 90\}$. Altogether the vectors Q are contained in $\mathbb{R}^{2^{80}\cdot 181}$. It is obvious that vectors of such a high dimension cannot be computed and stored. This problem is often referred to as "the curse of dimensionality", see Bellman [Bel57, p. ix]. To implement the Q-learning method discussed in the previous section we apply data compression.

We use a *clustering method* proposed by Santos [San99, pp. 85] for the Khepera robot, see also Bendersky [Ben03, pp. 35]. Weissensteiner [Wei03, pp. 107] applied the presented method to find good consumption-investment

decisions in a life cycle approach. The clustering method is a variant of *vector quantization* as introduced by Kohonen [Koh95, ch. 6].

By the scaling map

$$\{-90, \ldots, 90\} \to [0,1], a \mapsto (a+90)/180,$$

we map $A(s)$ into the unit interval. In the following we identify

$$A = A(s) = \{i/180 : i = 0, \ldots, 180\}.$$

Recall that in an experimental run of the robot we obtain, for each time step t, a triple

$$(s_t, a_t, Q_t(a_t, s_t)) = (s_0^t, \ldots, s_7^t, a_t, Q_t(a_t, s_t)) \in S \times A \times \mathbb{R} \subset [0,1]^9 \times \mathbb{R} \subset \mathbb{R}^{10}.$$

Following the considerations from the previous section these triples converge to $(s_t, a^*, V^*(s_t))$ with $a^* \in A^*(s_t)$, that is

$$\lim_{t\to\infty}((s_t, a_t, Q_t(a_t, s_t)) - (s_t, a^*, V^*(s_t))) = 0, \quad \text{almost surely in } \Omega.$$

We encode the wanted data a^* and V^* in a matrix $W \in \mathbb{R}^{m\times 10}$ with a suitable number m of rows

$$w(j), \quad \text{for } j = 1, \ldots, m,$$

of the form

$$w(j) = (w_s(j), w_a(j), w_q(j)) \in [0,1]^{8+1} \times \mathbb{R}$$

and the following desired property. If s is a state then there is a row j with $w_s(j)$ nearby, $w_q(j)$ is a good approximation of $V^*(s)$ and $w_a(j)$ is close to an optimal action $a^* \in A^*(s)$. Or, in more precise terms,

$$\min_j \parallel s - w_s(j) \parallel \ll 1,$$

and for

$$j^* = \arg\min_j \parallel s - w_s(j) \parallel$$

we have

$$|V^*(s) - w_q(j^*)| \ll 1$$

and

$$\arg\min_a |a - w_a(j^*)| \in A^*(s).$$

Here $\| - \|$ is any norm, in our implementation the L_2-*norm*, that is

$$\|x\|_2 = (\sum_{i=1}^{n} x_i^2)^{\frac{1}{2}}, \quad \text{for } x \in \mathbb{R}^n.$$

In the terminology of neural science or artificial intelligence the matrix W is called a *network* with m *units* $j = 1, \ldots, m$ with *center* $w(j)$.

Learning Phase

The network W is determined by one or more experimental runs

$$(\ldots, a_t, s_t, \ldots, a_0, s_0)$$

of the robot as the limit of networks $W_t,\ t = 0, 1, \ldots$ with m_t units where the m_t are an increasing, but not strictly increasing sequence of row dimensions. In neural science terminology one talks about an *adaptive network* W since the number of units can change.

The sequence $(\ldots, a_t, s_t, \ldots, a_0, s_0)$ and the $m_t \times 10$-matrices W_t with rows

$$w_t(j),\ j = 1, \ldots, m_t,$$

are obtained as follows. We start with a randomly chosen state-action pair (a_0, s_0) and the 1×10-matrix W_0 with the row

$$w_0(1) = (s_0, a_0, 0).$$

After t iterations the history $(a_t, s_t, \ldots, s_0)$ has been observed and the network $W_t \in \mathbb{R}^{m_t \times 10}$ has been obtained. The rows of this matrix W_t are interpreted as a good approximation of the vectors

$$(s_k, a_k, Q_k(a_k, s_k)), \quad k = 0, \ldots, t,$$

where, however, $m_t \ll t$ in general. The matrix W_t is updated as explained below.

For the update we need to select greedy actions based on the current approximation. In a state s we choose a row j^* with $w_s(j^*)$ close to s and $w_q(j^*)$ as high as possible. We choose a "distance function" $d_{s,q}(x, y)$ of a nonnegative real variable x for the distance between s and w_s and a real variable y for the approximate action-value w_q, with the property that it is strictly monotonically increasing in x and decreasing in y. A possible choice would be $d_{s,q}(x, y) = x - ay$ with a positive constant a.

In our actual implementation action-values q are kept in the unit interval by cutting them off below zero and above one, and we use the distance function

$$d_{s,q}(x,q) = x^2 + \frac{(1-q)^2}{2}.$$

Moreover, we need to select a row k^* for the current state-action pair (s,a) with $w_s(k^*)$ close to s and $w_a(k^*)$ close to a. Again we choose a distance, in our case the L_2-norm. To decide when to add a new unit during the update we additionally choose two small constants $\delta_{s,a} > 0$ and $\delta_{s,q} > 0$. The choice of the distances and constants have to be made appropriately.
The actual update proceeds in the following steps.

1. Apply action a_t to the state s_t, observe the new state s_{t+1} and compute the reward $r_t = R(s_{t+1}, a_t, s_t)$.

2. Compute

$$j^* = \arg\min_j d_{s,q}(\| \, s_{t+1} - w_{t,s}(j) \, \|, w_{t,q}(j)).$$

   The properties of $d_{s,q}$ imply that the vector $(w_{t,s}(j^*), w_{t,q}(j^*))$ can be used as an approximation of $(s_{t+1}, \max_{a'} Q_t(a', s_{t+1}))$ in the update rule Equation (4.2).

3. Compute

$$\delta_1 = \min_k \|(s_t, a_t) - (w_{t,s}(k), w_{t,a}(k))\| \, .$$

   If $\delta_1 \leq \delta_{s,a}$ then compute

$$k^* = \arg\min_k \|(s_t, a_t) - (w_{t,s}(k), w_{t,a}(k))\| \, .$$

   If $\delta_1 > \delta_{s,a}$ then add to W_t a new row or unit

$$w_t(m_t + 1) = (s_t, a_t, r_t + \gamma w_{t,q}(j^*))$$

   and set

$$k^* = m_t + 1.$$

   The vector $w_t(k^*)$ is used as an approximation of $(s_t, a_t, Q_t(s_t, a_t))$ in the update rule Equation (4.2).

4. Define the new network W_{t+1} by

$$w_{t+1}(k) = w_t(k), \quad \text{for } k \neq k^*$$

and update the row

$$w_t(k^*) = (w_{t,s}(k^*), w_{t,a}(k^*), w_{t,q}(k^*))$$

by

$$\begin{cases} w_{t+1,q}(k^*) = w_{t,q}(k^*) + \alpha_t(r_t + \gamma w_{t,q}(j^*) - w_{t,q}(k^*)), \\ w_{t+1,s}(k^*) = w_{t,s}(k^*) + \eta_{t,s}(s_t - w_{t,s}(k^*)), \\ w_{t+1,a}(k^*) = w_{t,a}(k^*) + \eta_{t,a}(a_t - w_{t,a}(k^*)). \end{cases} \tag{4.8}$$

The sequence α_t denotes step-size parameters for Q-learning and $\eta_{t,s}, \eta_{t,a}$ are zero-sequences of positive step-size parameters for the network.
The first of the equations (4.8) is determined by equation (4.2) with the approximate values from the preceding steps.
The other two equations of (4.8) are typical for clustering or vector quantization. Since the vector $(w_{t+1,s}(k^*), w_{t+1,a}(k^*))$ is intended to be a better approximation of the observed state action pair (s_t, a_t) than $(w_{t,s}(k^*), w_{t,a}(k^*))$ it is chosen in the interior of the line segment between the two latter vectors and hence the updated center moves towards (s_t, a_t).
In the actual *finite* runs of the robot the parameters are chosen constant with $\alpha_t = 0.7,\ \eta_{t,s} = 0.01, \eta_{t,q} = 0.2$.

5. Finally the action a_{t+1} is chosen. Compute

$$\delta_2 = \min_i d_{s,q}(\parallel s_{t+1} - w_{t+1,s}(i) \parallel, w_{t+1,q}(i)).$$

If $\delta_2 \leq \delta_{s,q}$ then compute

$$i^* = \arg\min_i d_{s,q}(\parallel s_{t+1} - w_{t+1,s}(i) \parallel, w_{t+1,q}(i))$$

and use $w_{t+1,a}(i^*)$ as an approximation of the greedy action a in Equation (4.4). For exploration we draw a random number $b \in [-b_t, b_t]$ and choose

$$a_{t+1} = \arg\min_a |a - w_{t+1,a}(i^*) - b|.$$

Here b_t is a zero sequence of positive numbers.
If $\delta_2 > \delta_{s,q}$ choose a_{t+1} randomly and add to the matrix W_{t+1} the new row or unit

$$w_{t+1}(m_{t+1} + 1) = (s_{t+1}, a_{t+1}, 0).$$

In our experiment we fix a number of iterations T for the learning phase. For exploration in step 5 we choose $b_0 = 1$ and

$$b_{t+1} = b_t - \frac{1}{T}. \tag{4.9}$$

Execution phase

After T iteration steps where T is sufficiently large the matrix $W = W_T$ with $m = m_T$ rows is used to construct an approximation of an optimal deterministic policy. The rows $w(j)$ in the network approximate the optimal action-values for large T. Thus a greedy policy π with respect to the network which can be computed for each state as follows is a (sub)optimal policy. For state s first the row

$$i^* = \arg\min_i d_{s,q}(\parallel s - w_s(i) \parallel, w_q(i)) \tag{4.10}$$

is selected and then choose action

$$a = \arg\min_a |a - w_a(i^*)|.$$

Thus the policy π is given by the network W and can be executed and evaluated.

4.6 Implementation

`RealRobo` was implemented in `Visual C++ 6.0`. Compare Sections 3.4.2 and 3.4.6 for general programming details. The program has approximately 4500 lines of source code. Figure 4.3 shows the `RealRobo` window including the two control panels. The robot is visualized using texture images, taken with a digital camera.

4.6.1 Robot Control

The function `InitializeComPort` sets the baud rate, `COM port` number and parameters and opens the serial connection to the robot. The computer and the robot are communicating with text messages, that represent the different control commandos. For a complete list of all commandos consult the Khepera user manual by K-Team [KT99, pp. 35].

The most important functions are `Set_Speed` to set the speed of the two motors and `Read_Sensors` that returns the values of the eight proximity sensors. These two functions are used by `Turn` and `Move_Forward` to turn an angle and move forward or backward. For function prototypes see Listing 12.

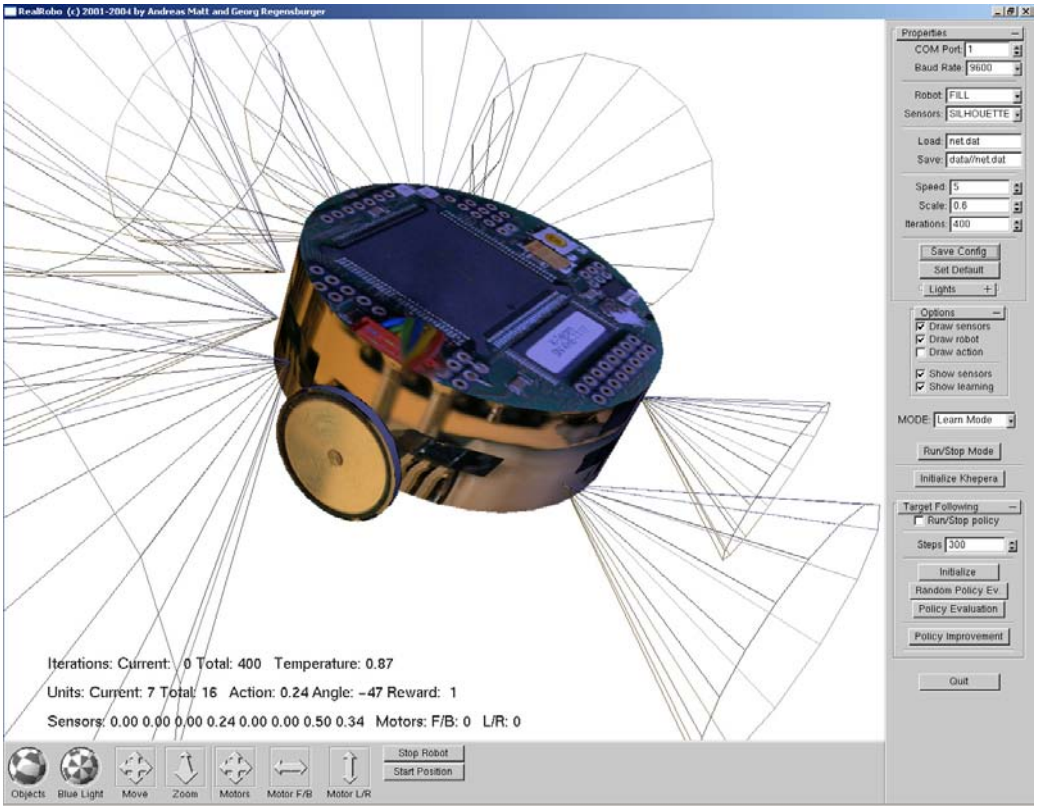

Figure 4.3: `RealRobo` and its control panels

4.6.2 Network

We use the functions `privateDistSQ` and `privateDistSA` to compute the distances in the learning phase from Section 4.5. The closest units are computed by the functions `privateNearestSQ` and `privateNearestSA` respectively. A new unit is added by the function `privateAddNeuron` and the update of the centers is done by the function `QUpdate`. For function prototypes consult Listing 12.

4.7 Experiments

We first describe how we implement experiments in general and then describe a specific implementation of an obstacle avoidance experiment.

4.7.1 Implementation Process

Implementing an experiment with a robot is a time consuming task. First a physical environment has to be specified, in our case a wooden box with obstacles and light settings. In our experiments we rely on natural light since the sensors of the robot are very sensitive to artificial light and heavy oscillation occurs. Then the actions of the robot have to be defined and implemented in the control program. Usually a test run is carried out. The robot is put in a state, random actions are applied and the sensor values are observed. We ensure that the connection to the robot is established, the actions are applied correctly and the sensors respond.

To define rewards whose optimal policy corresponds to a desired behavior is a complex task, compare Santos [San99]. Moreover, the rewards depend on the observed sensor values which themselves depend on the light, surface and material of the obstacles perceived in the wooden box. The sensor values are observed in a series of test runs where random actions are applied. The sensor values are analyzed and the rewards adapted if necessary.

Once the physical environment and rewards are fixed a representation of the states, actions and action-values has to be chosen, for example the adaptive network described in Section 4.5 or the discretization in 9.2.2. Then a model-free learning method and exploration technique have to be implemented according to the chosen representation.

The number of iterations for the learning phase has to be selected. Sample learning runs with different numbers of iterations are conducted and action-values are observed to estimate the appropriate number of iterations. During the learning phase the robot has to be observed. It may possibly act in a wrong way due to technical interferences, or it may get stuck in certain physical positions. To overcome such problems the learning can be stopped or the robot can be corrected manually.

In this phase of the implementation process reflexes are designed. A reflex is an instantly applied rule that lets the robot apply an action if it is in a certain state or perceives certain sensor values. Reflexes let the robot avoid certain states. A reflex can also be applied by the user by correcting the robot manually, for example, when it bumps into a wall. Note that the reflexes influence the sequence of visited states and thus the whole learning process.

After learning the execution phase begins and the learned policy is applied. Depending on the representation of states, actions and action-values the policy is derived and represented differently. Different criteria are used to decide its quality.

One criterion concerns the quality of the policy with respect to the desired

behavior, that is whether the rewards are well chosen for the intended task. For instance, we can measure the average distance to the obstacles for the obstacle avoidance task. In other terms, whereas the optimal policy is, by definition, optimal with respect to the optimal value function determined by the chosen rewards it may not be optimal with respect to another quality criterion.

To compare and decide the quality of a policy with respect to fixed rewards we use an approximation of its utility, see Section 9.4. For a high number of iterations the approximation of the utility obtained is good. In robot experiments relatively few iterations can be done, since the robot takes about a second to apply an action. An experiment may take from 30 minutes to an hour for a few thousand iterations, while, for example, using the simulator an experiment with 50000 iterations is carried out in less than a second.

Instead of using the approximation obtained, we can estimate the value function by observing the rewards in test runs. The robot is put in a starting state and the policy is applied for a fixed number T of iterations. The resulting sequence of states and actions is used to compute the (discounted) return up to the chosen iteration, see Section 1.2.2. This process can be repeated and the obtained returns averaged. The average return then serves as an estimate of the (discounted) value function for the starting state.

If the policy obtained does not perform sufficiently well the implementation process can be repeated from the beginning and the physical environment, rewards, state-action representation or number of iterations adjusted.

4.7.2 Obstacle Avoidance

We want the robot to avoid obstacles using the rewards from Section 4.2 in the wooden box displayed in Figure 4.4. The box is 22.5 cm long and 20 cm wide. An obstacle is situated in the middle of the long side attached to the wall, it is 2 cm wide and 4.5 cm long. The wall and the obstacle are 5 cm high.

We consider the states and actions defined in Section 4.3 and apply Q-learning described in Section 4.4 using the network implementation from Section 4.5. The experiments are carried out with natural morning light to illuminate the wooden box. They can be repeated using `RealRobo` and following the instructions given below.

The robot is put in the center of the wooden box looking towards the obstacle. Then the learning phase begins. While learning, the robot is provided with two reflexes. If it gets too close to an obstacle it undoes the last action executed. This reflex prevents the robot from future collisions. If

Figure 4.4: Wooden box to learn obstacle avoidance

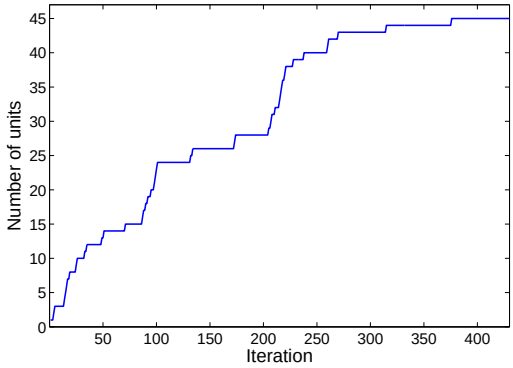

Figure 4.5: Number of units of the network during the learning phase

there is no obstacle in sight it moves straight forward until it again perceives an obstacle. This reflex gives the robot a strategy to leave empty areas.

The exploration technique used lets the robot act randomly at the beginning of the learning phase. With increasing iterations the actions of the robot become the greedy actions with respect to the current network. The network starts with zero units. New units are added to obtain a representation of states, actions and action-values. Figure 4.5 shows the increasing number of units of the network during the learning phase for an experiment with 430 iterations.

We conduct an experiment with 300 iterations. To compare the obtained policies we use estimates of the value function derived from sample runs. The robot is put to a start position. Then the policy is executed for 100 steps and the rewards are observed. The average sum of the rewards is computed. This sum can be seen as an estimate of the undiscounted value function for the starting state. Compare *Monte Carlo* methods in Sutton and Barto [SB98,

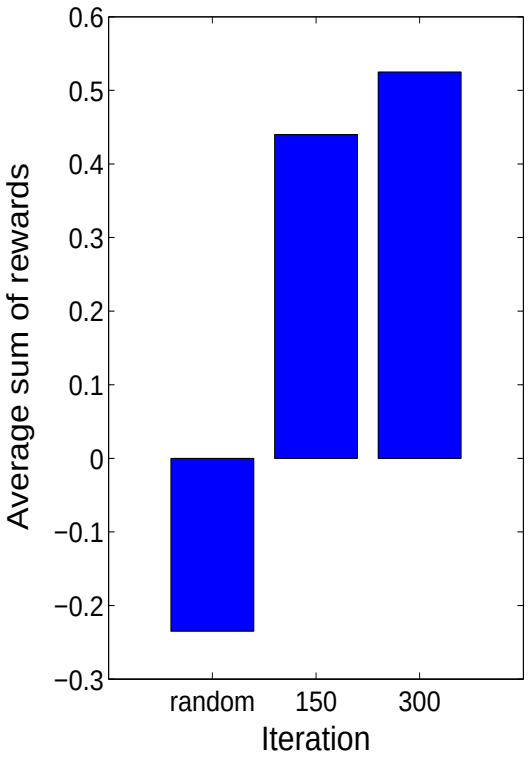

Figure 4.6: Average sum of rewards for the random policy and the policy obtained after 150 and 300 Q-learning iterations

pp. 113] and Bertsekas and Tsitsiklis [BT96, pp. 181].

For the random policy, a policy obtained after 150 iterations and a policy obtained after 300 iterations we conduct three sample runs each and average the obtained estimates. In Figure 4.6 these averages for the three policies are displayed. We observe that the average sums of rewards raise with the iterations. Figure 4.7 shows the robot following the policy obtained after 300 iterations. The robot starts on the left side looking towards the wall. It turns right and proceeds to the center approaching the wall. It turns right and continues straight towards the right corner. There it turns right and gets quite close to the obstacle, since its corner is difficult to perceive with the sensors. Then the robot turns right again and proceeds to the center of the wooden box.

In Figure 4.8 the actions chosen by the learned policy in two sample states are displayed. The arrows show the actions of the policy. In the state displayed on the left, the obstacles are quite near on the right side of the

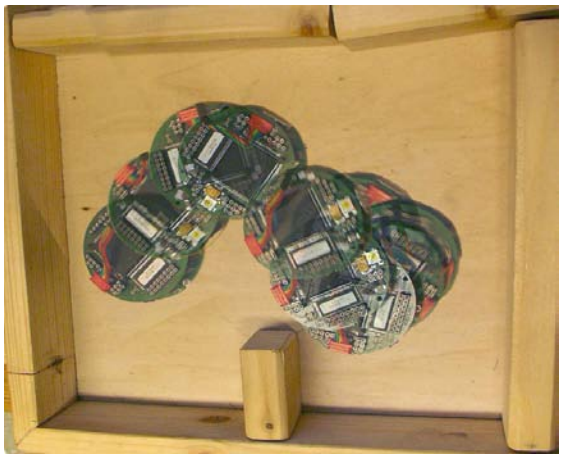

Figure 4.7: The robot following a learned policy

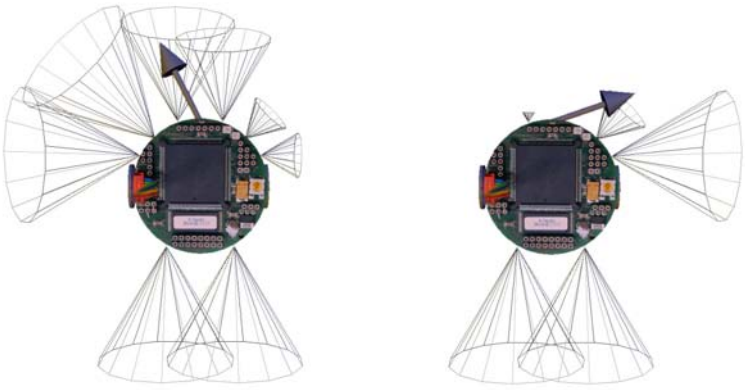

Figure 4.8: The actions chosen by the learned policy in two sample states

robot and the action is to turn 27 degrees to the left. In the right state the robot stands close to obstacles on its left and front side. The action is to turn 71 degrees to the right.

To repeat this experiment connect your Khepera mobile robot to the serial port and start `RealRobo`. Open the `Properties` rollout and choose the appropriate `COM port` and `baud rate`. The settings of the parameters of the `Properties` and the `Options` rollout can be saved by pressing the button `Save Config`. The next time `RealRobo` is started the new settings will be loaded automatically. To restore the default settings press the button `Set Default`.

The connection with the robot is initialized by pressing `Initialize Khepera`. Now the sensor cones show the current sensor values graphically. They can be turned off and on by switching the checkerbox `Draw sensors`. With `Show sensors` the current sensor values are displayed in the window. Turn on `Show Learning` to display details on the current iteration, exploration, number of units and actions executed. The temperature value is the random factor for the exploration, compare Equation 4.9.

Now put the robot in a wooden box and switch the `Mode` to `Learn Mode`. Choose the desired amount of 300 iterations in `Iterations` spinner in the `Properties` rollout. To start the learning phase press the button `Run/Stop Mode`. After each 10 iterations the current network is saved with the name given in the `Save` editbox in the `Properties` rollout. The number of iteration is added to the file name, for example filename `net150.dat` for the network after 150 iterations.

To run a policy choose the file name of the network you want to load in the `Load` editbox. Then go to `Run Mode` and press the button `Run/Stop Mode`. Now the policy derived from the loaded network will be executed for the number of iterations set in the `Iterations` spinner in the `Properties` rollout. The observed rewards are displayed in the console window. To run the random policy switch the `Mode` to `Random Angle` and press the button `Run/Stop Mode`.

Part II

Several Environments

Chapter 5

Reinforcement Learning

Several methods to find optimal or suboptimal policies for one Markov decision process are discussed in the first part. Now we investigate a general view of behavior, independently from a single environment. As an example imagine that a robot should learn to avoid obstacles. What we have in mind is a behavior suitable for not only one, but several environments simultaneously, see Figure 5.1.

Obviously, a policy for several environments cannot – in general – be optimal for each one of them. Improving a policy for one environment may result in a worse performance in another. Nevertheless it is often possible to improve a policy for all environments.

As far as we know, the problem of learning a policy for several MDPs has not yet been investigated. The special case of finding a policy for several rewards is often referred to as multi-objective or multi-criteria reinforcement learning, see Section 5.8 for a discussion.

5.1 The State Action Space and Realizations

To apply one policy to different environments we introduce the following notions. A *state action space (SAS)* $\mathbb{E}$ is given by

- A finite set S of states.
- A family $\mathbf{A} = (A(s))_{s \in S}$ of finite sets of actions.

Let $\mathbb{E} = (S, \mathbf{A})$ be an SAS. We call an environment $E = (S_E, \mathbf{A}_E, \mathbf{P}_E)$ a *realization* of the state action space $\mathbb{E}$ if the set of states is a subset of S and the actions are the same as in the SAS, that is $S_E \subset S$ and

$$A_E(s) = A(s) \quad \text{for all } s \in S_E.$$

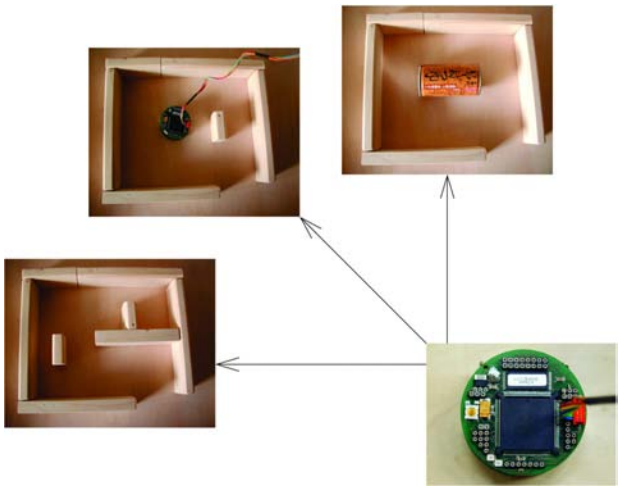

Figure 5.1: Obstacle avoidance for several environments simultaneously

We call an MDP $(E, \mathbf{R}, \gamma)$ a realization of the SAS $\mathbb{E}$ if the environment E is a realization of $\mathbb{E}$.

Since the set of actions for each state is given by the state action space, we can define policies for $\mathbb{E}$, which can be applied to any realization. A *policy* for $\mathbb{E}$ is given by

- A family $\pi = (\pi(- \mid s))_{s \in S}$ of probabilities $\pi(- \mid s)$ on $A(s)$.

Let $E = (S_E, \mathbf{A}_E, \mathbf{P}_E)$ be a realization for $\mathbb{E}$. Then the restriction

$$\pi|_E = (\pi(- \mid s))_{s \in S_E}$$

is a policy for E. A family of rewards for $\mathbb{E}$ is given by

- $\mathbf{R} = (R(s', a, s))_{s', s \in S, a \in A(s)}$ with $R(s', a, s) \in \mathbb{R}$.

Rewards can be restricted to any realization E of $\mathbb{E}$,

$$\mathbf{R}|_E = (R(s', a, s))_{s', s \in S_E, a \in A(s)}.$$

We write again π and $\mathbf{R}$ for the restrictions $\pi|_E$ and $\mathbf{R}|_E$.

Let $\mathbb{E} = (S, \mathbf{A})$ be an SAS. Let $\mathbf{E} = (E_i)_{i \in I}$ be a family of realizations of $\mathbb{E}$, where $E_i = (S_i, \mathbf{A}_i, \mathbf{P}_i)$. We denote by

$$S_{\mathbf{E}} = \bigcup_{i \in I} S_i \subset S$$

the set of all states for the family of realizations $\mathbf{E}$. Again since the set of actions for each state is given by the state action space, we can define policies and a family of rewards for $\mathbf{E}$, which can be restricted to any realization E_i. A *policy* for a family of realizations $\mathbf{E}$ is given by

- A family $\pi = (\pi(- \mid s))_{s \in S_{\mathbf{E}}}$ of probabilities $\pi(- \mid s)$ on $A(s)$.

Figure 5.2 shows the relation of the state action space and its realizations graphically. Note that one MDP is obviously a realization of the SAS defined

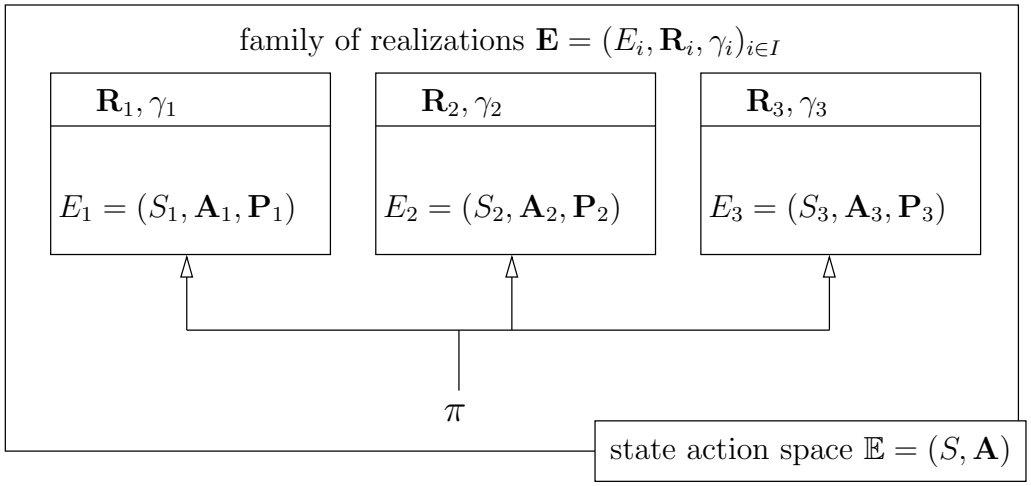

Figure 5.2: A state action space and a family of realizations

by its states and actions.

Let

$$\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$$

be a family of realizations of an SAS. In particular the following cases are included in our model: Several environments with a fixed family of rewards $\mathbf{R}$ for $\mathbf{E}$, that is

$$\mathbf{E} = (E_i, \mathbf{R}, \gamma)_{i \in I}.$$

Several rewards and discount rates for one fixed environment E, that is

$$\mathbf{E} = (E, \mathbf{R}_i, \gamma_i)_{i \in I}.$$

The second case is called a *multi-criteria* reinforcement problem, see bibliographical remarks in Section 5.8.

5.2 Improving Policies and Policy Improvement

We extend the notions of improving policies to families of realizations. Let $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ be a family of realizations of an SAS and π a policy for $\mathbf{E}$. We denote the value function and action-values for realization $(E_i, \mathbf{R}_i, \gamma_i)$ by V_i^π and Q_i^π and likewise for all other definitions from the first part.

We define the set of *improving policies* for policy π and the family of realizations $\mathbf{E}$ in state $s \in S_{\mathbf{E}}$ by

$$C_{\geq}^{\pi,\mathbf{E}}(s) = \bigcap_{i \in I, s \in S_i} C_{i,\geq}^{\pi}(s),$$

where $C_{i,\geq}^{\pi}(s)$ denotes the set of improving policies for E_i in state $s \in S_i$. See Section 1.6.

The set of *equivalent policies* for policy π and the family of realizations $\mathbf{E}$ in state s is defined by

$$C_{=}^{\pi,\mathbf{E}}(s) = \bigcap_{i \in I, s \in S_i} C_{i,=}^{\pi}(s).$$

Note that

$$\pi(- \mid s) \in C_{=}^{\pi,\mathbf{E}}(s) \subset C_{\geq}^{\pi,\mathbf{E}}(s).$$

We define the set of *strictly improving policies* for policy π and the family of realizations $\mathbf{E}$ in state s by

$$C_{>}^{\pi,\mathbf{E}}(s) = C_{\geq}^{\pi,\mathbf{E}}(s) \setminus C_{=}^{\pi,\mathbf{E}}(s).$$

Observe that the strictly improving policies are the improving policies in s, that are contained in the set of strictly improving policies for at least one realization, that is

$$C_{>}^{\pi,\mathbf{E}}(s) = \left\{ \tilde{\pi}(- \mid s) \in C_{\geq}^{\pi,\mathbf{E}}(s) : \tilde{\pi}(- \mid s) \in C_{i,>}^{\pi}(s) \text{ for at least one } i \in I \right\}.$$

With these definitions Corollaries 13 and 14 imply the following propositions for families of realizations.

Corollary 24. *Let $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ be a family of realizations of an SAS. Let π and $\tilde{\pi}$ be policies for $\mathbf{E}$. Then*

$$\tilde{\pi}(- \mid s) \in C_{\geq}^{\pi,\mathbf{E}}(s) \quad \textit{for all } s \in S_{\mathbf{E}}$$

implies

$$V_i^{\tilde{\pi}} \geq V_i^{\pi} \text{ for all } i \in I.$$

If additionally

$$\tilde{\pi}(- \mid s) \in C_{>}^{\pi,\mathbf{E}}(s) \quad \text{for at least one } s \in S_{\mathbf{E}}$$

then

$$V_i^{\tilde{\pi}} > V_i^{\pi} \text{ for at least one } i \in I.$$

Corollary 25. *Let* $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ *be a family of realizations of an SAS. Let* π *and* $\tilde{\pi}$ *be policies for* $\mathbf{E}$*. Then*

$$V_i^{\tilde{\pi}} = V_i^{\pi} \text{ for all } i \in I$$

if and only if

$$\tilde{\pi}(- \mid s) \in C_{=}^{\pi,\mathbf{E}}(s) \quad \text{for all } s \in S_{\mathbf{E}}.$$

Recall from Section 1.6 that the sets of (strictly) improving policies for one Markov decision process are invariant if we multiply the family of rewards by a positive real number and the equivalent policies are invariant if we multiply the rewards with a nonzero real number. Hence the same property holds for the sets of (strictly) improving and equivalent policies for a family of realizations.

5.3 Balanced Policies

Let $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ be a family of realizations of an SAS. We call a policy π for $\mathbf{E}$ *balanced* if there are no strictly improving policies, that is

$$C_{>}^{\pi,\mathbf{E}}(s) = \emptyset \quad \text{for all } s \in S_{\mathbf{E}}.$$

For one environment the notions of balanced and optimal coincide by Theorem 15 (iii). Again, the notion of balanced policies is invariant if the rewards are multiplied by a positive number.

We give a characterization of balanced policies. For $s \in S_{\mathbf{E}}$ let π_s denote the set of all policies that are arbitrary in s and equal π otherwise, that is

$$\pi_s = \{\tilde{\pi} : \tilde{\pi}(- \mid \tilde{s}) = \pi(- \mid \tilde{s}) \text{ for all } \tilde{s} \in S_{\mathbf{E}} \setminus s\}.$$

The following theorem says that if we change a balanced policy in one state the value functions remain unchanged in all realizations or it is worse in at least one. Compare the notion of equilibrium points in game theory Nash [Nas51] and for example Owen [Owe95].

Theorem 26. *Let* $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ *be a family of realizations of an SAS and* π *be a policy for* $\mathbf{E}$*. Then* π *is balanced if and only if for all* $s \in S_{\mathbf{E}}$ *and all* $\tilde{\pi} \in \pi_s$ *either*

$$\begin{aligned} V_i^{\tilde{\pi}} &= V_i^{\pi} \quad \text{for all } i \in I \text{ or} \\ V_i^{\tilde{\pi}} &< V_i^{\pi} \quad \text{for at least one } i \in I. \end{aligned} \tag{5.1}$$

Proof. Let π be balanced. Let $s \in S$ and $\tilde{\pi} \in \pi_s$. Since π is balanced we have $C_{>}^{\pi,\mathbf{E}}(s) = \emptyset$. So either $\tilde{\pi}(- \mid s) \in C_{\geq}^{\pi,\mathbf{E}}(s)$ and hence

$$\tilde{\pi}(- \mid s) \in C_{=}^{\pi,\mathbf{E}}(s) = C_{\geq}^{\pi,\mathbf{E}}(s)$$

and

$$V_i^{\tilde{\pi}} = V_i^{\pi} \quad \text{for all } i \in I$$

by Theorem 24, or $\tilde{\pi}(- \mid s)$ is not contained in $C_{\geq}^{\pi,\mathbf{E}}(s)$. Then there exists an $i \in I$ such that $s \in S_i$ and $\tilde{\pi}(- \mid s)$ is not contained in $C_{i,\geq}^{\pi}(s)$, that is

$$\sum_{a \in A(s)} Q_i^{\pi}(a, s)\tilde{\pi}(a \mid s) < V_i^{\pi}(s).$$

Corollary 10 (ii) implies $V_i^{\tilde{\pi}} < V_i^{\pi}$. Suppose now that π is not balanced. Then there exists an $s \in S_{\mathbf{E}}$ with $C_{>}^{\pi,\mathbf{E}}(s) \neq \emptyset$. We define a policy $\tilde{\pi}$ by choosing

$$\tilde{\pi}(- \mid s) \in C_{>}^{\pi,\mathbf{E}}(s)$$

and set $\tilde{\pi}(- \mid \tilde{s}) = \pi(- \mid \tilde{s})$ for all $\tilde{s} \in S_{\mathbf{E}} \setminus s$. Note that $\tilde{\pi} \in \pi_s$. Theorem 24 gives $V_i^{\tilde{\pi}} \geq V_i^{\pi}$ for all $i \in I$ and $V_i^{\tilde{\pi}} > V_i^{\pi}$ for at least one $i \in I$, a contradiction to (5.1). ■

5.4 Joint Optimal Policies

In this section we show that there exists a policy for a family of realizations that is optimal for all of them if and only if the intersection of the optimal actions is not empty.

Let $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ be a family of realizations of an SAS. We denote the optimal actions for realization $(E_i, \mathbf{R}_i, \gamma_i)$ in state s by $A_i^*(s)$.

Theorem 27. *There exists a policy* π *for the family of realizations* $\mathbf{E}$ *such that* π *is optimal for each* $(E_i, \mathbf{R}_i, \gamma_i)$ *if and only if*

$$\bigcap_{i \in I, s \in S_i} A_i^*(s) \neq \emptyset \quad \text{for all } s \in S_{\mathbf{E}}.$$

Proof. Let be π a policy for $\mathbf{E}$ such that π is optimal for each $(E_i, \mathbf{R}_i, \gamma_i)$. Let $s \in S_{\mathbf{E}}$. Choose an $a \in A(s)$ with $\pi(a \mid s) > 0$. Then $a \in A_i^*(s)$ for $i \in I$ and $s \in S_i$ by Theorem 21. If the intersection of optimal actions is not empty for all states, we define a deterministic policy π by

$$\pi(- \mid s) = a \in \bigcap_{i \in I, s \in S_i} A_i^*(s) \quad \text{for all } s \in S_{\mathbf{E}}$$

Then π is optimal for each realization $(E_i, \mathbf{R}_i, \gamma_i)$, again by Theorem 21. ■

Note that a policy for a family of realizations that is optimal for each realization is balanced. If such a policy exists then we can find a deterministic policy with the same property by choosing an action in the intersection of optimal actions for each state.

The proof of Theorem 21 characterizing all optimal policies for an MDP can be adapted to prove the next result characterizing all policies for a family of realizations that are optimal in each one of them.

Theorem 28. *Let π be a policy for $\mathbf{E}$. Then π is optimal for each $(E_i, \mathbf{R}_i, \gamma_i)$ if and only if*

$$\pi(a \mid s) > 0 \text{ implies } a \in \bigcap_{i \in I, s \in S_i} A_i^*(s) \quad \text{for all } s \in S_{\mathbf{E}}.$$

5.5 Policies and Cones

In this section we discuss geometrical aspects of the sets of (strictly) improving and equivalent policies of a family of realizations. Since we allow infinite families, we recall some concepts and results for arbitrary convex sets. The notation and references are as in Section 1.8. Further references on *convexity* used for this sections are Rockafellar [Roc70] and Webster [Web94].

5.5.1 Faces and Extreme Points

The concept of faces for polyhedra can be generalized to arbitrary convex sets. A *face* of a convex subset $K \subset \mathbb{R}^d$ is a convex subset $F \subset K$ such that every line segment in K with a relative interior point in F has both endpoints in F, that is, whenever

$$\lambda \mathbf{x} + (1 - \lambda)\mathbf{y} \in F, \quad \text{with } \mathbf{x}, \mathbf{y} \in K \text{ and } 0 < \lambda < 1$$

then $\mathbf{x}, \mathbf{y} \in F$. For polyhedra this definition of faces is equivalent to the definition with valid inequalities given in Section 1.8.1.

The zero-dimensional faces of K are called *extreme points*. Clearly, a point $\mathbf{x} \in K$ is an extreme point of K if it is not a convex combination of two other points K; that is, $\mathbf{x} \in K$ is an extreme point if it has the property that whenever

$$\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}, \quad \text{with } \mathbf{y}, \mathbf{z} \in K \text{ and } 0 < \lambda < 1,$$

then $\mathbf{x} = \mathbf{y} = \mathbf{z}$. For polyhedra the notion of extreme points and vertices coincide.

Recall that polyhedra are defined as subsets given by a finite intersection of halfspaces. For closed convex subsets we have the following theorem, see Rockafellar [Roc70, p. 99] or Webster [Web94, p. 70].

Theorem. *Every closed convex subset of $\mathbb{R}^d$ is the intersection of the halfspaces containing it.*

Every polytope is the convex hull of its vertices, see Theorem 1.8.1. This is a special case of the Krein-Milman theorem. For a proof in $\mathbb{R}^d$ see for example Webster [Web94, p. 86] and in an arbitrary vector spaces Lang [Lan93, pp. 83].

Theorem. *Every compact convex subset of $\mathbb{R}^d$ is the convex hull of its extreme points.*

5.5.2 Convex Cones

A nonempty subset $C \subset \mathbb{R}^d$ is a *(convex) cone* if with any two points $\mathbf{x}, \mathbf{y} \in C$ it contains all their linear combinations with nonnegative coefficients, that is

$$\lambda\mathbf{x} + \mu\mathbf{y} \in C, \quad \text{for } \lambda, \mu \geq 0.$$

In particular, every cone contains the origin $\mathbf{0} \in \mathbb{R}^d$. The intersection of any number of cones is again a cone. The *conical hull* of a subset $X \subset \mathbb{R}^d$, $\text{cone}(X)$, is defined as the intersection of all cones in $\mathbb{R}^d$ that contain X. It is given by

$$\text{cone}(X) = \{\lambda_1\mathbf{x}_1 + \cdots + \lambda_n\mathbf{x}_n : n \geq 1, \mathbf{x}_i \in X, \lambda_i \geq 0\}.$$

The conical hull of the empty set is defined as $\{\mathbf{0}\}$. The only bounded cone is $\{\mathbf{0}\}$.

Let $\mathbf{a} \in \mathbb{R}^d$ be a nonzero row vector. A set of the form $\{\mathbf{x} \in \mathbb{R}^d : \mathbf{a}\mathbf{x} \leq 0\}$ is a *linear halfspaces*. Linear halfspaces are obviously cones. For a proof of the following result see Rockafellar [Roc70, p. 101].

Theorem. *Every closed convex cone in $\mathbb{R}^d$ is the intersection of the linear halfspaces containing it.*

For convex cones the notion of extreme points is not useful, since the origin is the only possible candidate. Instead one considers *extreme rays*, which are one-dimensional faces, that are halflines starting from the origin.

The *lineality space* L of a convex cone C is defined as $L = C \cap -C$. It is the largest linear subspace contained in C. A cone is *pointed* if its lineality space is $\{\mathbf{0}\}$. Every cone C is the direct sum of its lineality space L and the pointed cone $C \cap L^\perp$, that is

$$C = \left(C \cap L^\perp\right) \oplus L,$$

where $L^\perp$ denotes the orthogonal complement. For a proof of the next result see Rockafellar [Roc70, p. 167].

Theorem. *Every pointed and closed convex cone in $\mathbb{R}^d$ not $\{\mathbf{0}\}$ is the conical hull of its extreme rays.*

A cone C is called *finitely generated* if $C = \text{cone}(X)$ for a finite set of vectors X. A cone $C \subset \mathbb{R}^d$ is *polyhedral* if it is a polyhedron given by the intersection of finitely many linear halfspaces, that is

$$C = P(A, \mathbf{0}) = \left\{\mathbf{x} \in \mathbb{R}^d : A\mathbf{x} \leq \mathbf{0}\right\}$$

for a matrix A. The following theorem shows that the concepts of polyhedral and finitely generated cones are equivalent, compare the finite basis theorem for polytopes in Section 1.8.1. For a proof we refer to Schrijver [Sch86, p. 87] or Ziegler [Zie98, pp. 30].

Theorem. *A cone is finitely generated if and only if it is polyhedral.*

5.5.3 Improving Policies

Let $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ be a family of realizations of an SAS, π a policy for $\mathbf{E}$ and $s \in S_{\mathbf{E}}$.

Recall from Section 5.2 that the set of improving policies $C_{\geq}^{\pi,\mathbf{E}}(s)$ for π and $\mathbf{E}$ in state s is defined as the intersection of the improving policies for each realization $(E_i, \mathbf{R}_i, \gamma_i)$ with $s \in S_i$,

$$C_{\geq}^{\pi,\mathbf{E}}(s) = \bigcap_{i \in I, s \in S_i} C_{i,\geq}^{\pi}(s).$$

The set of improving policies $C^{\pi}_{i,\geq}(s)$ for realization $(E_i, \mathbf{R}_i, \gamma_i)$ is given by the intersection of the standard simplex $C(s)$ in $\mathbb{R}^{A(s)}$ and the halfspace

$$H^{\pi}_{i,\geq}(s) = \{x \in \mathbb{R}^{A(s)} : \sum_{a \in A(s)} Q^{\pi}_i(a,s)x(a) \geq V^{\pi}_i(s)\},$$

that is

$$C^{\pi}_{i,\geq}(s) = H^{\pi}_{i,\geq}(s) \cap C(s),$$

see Section 1.8.3. Let

$$H^{\pi,\mathbf{E}}_{\geq}(s) = \bigcap_{i \in I, s \in S_i} H^{\pi}_{i,\geq}(s).$$

Then

$$C^{\pi,\mathbf{E}}_{\geq}(s) = H^{\pi,\mathbf{E}}_{\geq}(s) \cap C(s).$$

Recall that by the Bellman equation (1.7) we have

$$\sum_{a \in A(s)} Q^{\pi}_i(a,s)\pi(a \mid s) = V^{\pi}_i(s)$$

for each realization $(E_i, \mathbf{R}_i, \gamma_i)$ with $s \in S_i$. Hence

$$H^{\pi}_{i,\geq 0}(s) = \{x \in \mathbb{R}^{A(s)} : \sum_{a \in A(s)} Q^{\pi}_i(a,s)x(a) \geq 0\} = H^{\pi}_{i,\geq}(s) - \pi(- \mid s).$$

We denote the closed convex cone given by the intersection of these linear halfspaces by

$$H^{\pi,\mathbf{E}}_{\geq 0}(s) = \bigcap_{i \in I, s \in S_i} H^{\pi}_{i,\geq 0}(s).$$

Then

$$H^{\pi,\mathbf{E}}_{\geq}(s) = \bigcap_{i \in I, s \in S_i} H^{\pi}_{i,\geq}(s) = \bigcap_{i \in I, s \in S_i} (\pi(- \mid s) + H^{\pi}_{i,\geq 0}(s)) = \pi(- \mid s) + H^{\pi,\mathbf{E}}_{\geq 0}(s).$$

Summing up we note that the set of improving policies in state s is the intersection of $\pi(- \mid s) + H^{\pi,\mathbf{E}}_{\geq 0}(s)$ with the standard simplex.

The set of equivalent policies $C^{\pi,\mathbf{E}}_{=}(s)$ for π and $\mathbf{E}$ in state s is defined in Section 5.2 as

$$C^{\pi,\mathbf{E}}_{=}(s) = \bigcap_{i \in I, s \in S_i} C^{\pi}_{i,=}(s).$$

Recall from Section 1.8.3 that the set of equivalent policies is the intersection of an affine hyperplane with the standard simplex,

$$C^{\pi}_{i,=}(s) = H^{\pi}_{i,=}(s) \cap C(s),$$

with

$$H^{\pi}_{i,=}(s) = \{x \in \mathbb{R}^{A(s)} : \sum_{a \in A(s)} Q^{\pi}_i(a,s)x(a) = V^{\pi}_i(s)\}.$$

Hence

$$C^{\pi,\mathbf{E}}_{=}(s) = H^{\pi,\mathbf{E}}_{=}(s) \cap C(s)$$

with

$$\pi(- \mid s) \in H^{\pi,\mathbf{E}}_{=}(s) = \bigcap_{i \in I, s \in S_i} H^{\pi}_{i,=}(s).$$

See Figure 5.3 for an example with three actions and two realizations. The

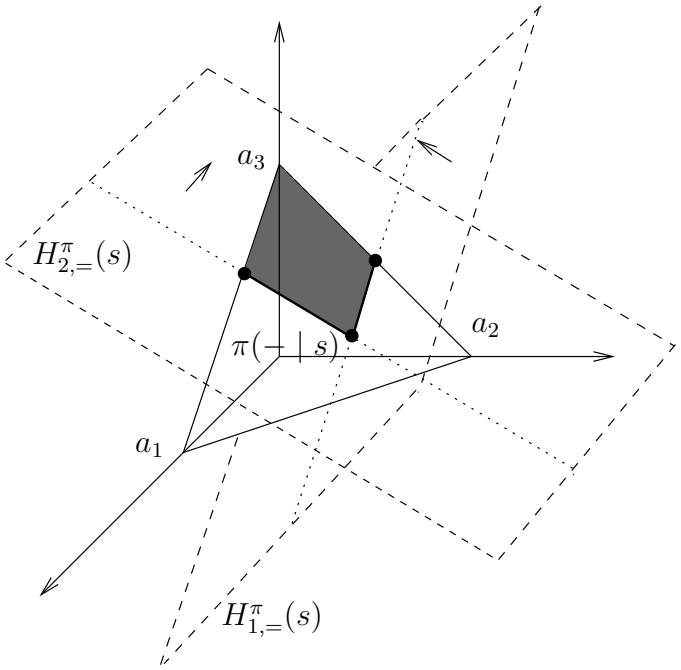

Figure 5.3: Improving polices for two realizations

halfspaces $H^{\pi}_{1,\geq}(s)$ and $H^{\pi}_{2,\geq}(s)$ are marked by the small arrows and the set of improving policies $C^{\pi,\mathbf{E}}_{\geq}(s)$ is the shaded area.

The set $H^{\pi,\mathbf{E}}_{=}(s)$ is an affine subspace (or flat) given by the intersection of affine hyperplanes. The associated linear subspace

$$H^{\pi,\mathbf{E}}_{=0} = H^{\pi,\mathbf{E}}_{=}(s) - \pi(- \mid s)$$

is the lineality space of the cone $H^{\pi,\mathbf{E}}_{\geq 0}$ and is given by

$$H^{\pi,\mathbf{E}}_{=0} = \bigcap_{i \in I, s \in S_i} \{x \in \mathbb{R}^{A(s)} : \sum_{a \in A(s)} Q^{\pi}_i(a,s)x(a) = 0\}.$$

Thus the set of equivalent policies is the intersection of an affine subspace through $\pi(- \mid s)$ with the standard simplex.

The set of strictly improving policies $C_{>}^{\pi,\mathbf{E}}(s)$ for π and $\mathbf{E}$ in state s is defined as

$$C_{>}^{\pi,\mathbf{E}}(s) = C_{\geq}^{\pi,\mathbf{E}}(s) \setminus C_{=}^{\pi,\mathbf{E}}(s).$$

Geometrically, this is the intersection of the standard simplex with the cone $H_{\geq 0}^{\pi,\mathbf{E}}$ without its lineality space plus the point $\pi(- \mid s)$.

5.5.4 Improving Vertices

We consider finite family of realizations. Then the set of (strictly) improving and equivalent policies are defined by finitely many (in)equalities and are thus polytopes. We discuss geometrical and computational aspects for this case specializing the results from the previous section.

Let $\mathbf{E} =(E_i, \mathbf{R}_i, \gamma_i)_{i\in I}$ be a finite family of realizations of an SAS. Let π be a policy for $\mathbf{E}$ and $s \in S_{\mathbf{E}}$. The set of improving policies $C_{\geq}^{\pi,\mathbf{E}}(s)$ for π and $\mathbf{E}$ in state s is a polytope given by the intersection of a polyhedral cone plus the point $\pi(- \mid s)$ with the standard simplex.

We call the vertices $\operatorname{vert}(C_{\geq}^{\pi,\mathbf{E}}(s))$ *improving vertices* for policy π and the family of realizations $\mathbf{E}$ in state s. We define the *strictly improving vertices* for π and $\mathbf{E}$ in state s by

$$\operatorname{vert}(C_{>}^{\pi,\mathbf{E}}(s)) = \operatorname{vert}(C_{\geq}^{\pi,\mathbf{E}}(s)) \cap C_{>}^{\pi,\mathbf{E}}(s).$$

The set of equivalent policies $C_{=}^{\pi,\mathbf{E}}(s)$ for π and $\mathbf{E}$ in state s is a polytope given by the intersection of an affine subspace through $\pi(- \mid s)$ with the standard simplex. We call the vertices $\operatorname{vert}(C_{=}^{\pi,\mathbf{E}}(s))$ *equivalent vertices* for policy π and the family of realizations $\mathbf{E}$ in state s.

A polytope representing the set of improving policies in a state for two realizations is shown in Figure 5.4. The set of improving policies is the

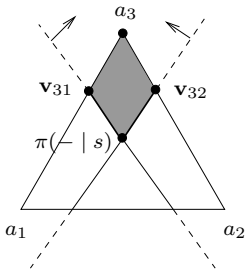

Figure 5.4: Improving policies for two realizations

shaded area, the strictly improving vertices $\mathrm{vert}(C^\pi_>(s)) = \{\mathbf{v}_{31}, a_3, \mathbf{v}_{32}\}$ and the equivalent vertices $\mathrm{vert}(C^{\pi,\mathbf{E}}_=(s)) = \{\pi(- \mid s)\}$. Note that a_3 is an action that improves the policy in both environments. Figure 5.5 shows that it may happen that all improving policies in a state are stochastic. In particular,

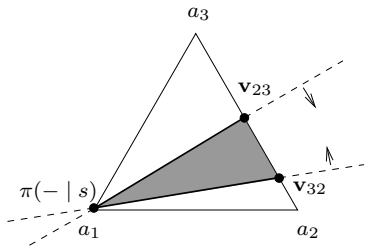

Figure 5.5: (Stochastic) improving policies for two realizations

the strictly improving vertices $\mathrm{vert}(C^\pi_>(s)) = \{\mathbf{v}_{23}, \mathbf{v}_{32}\}$ represent stochastic policies in a state.

For one MDP the strictly improving vertices are just the strictly improving actions. To see this, recall Equation (1.27)

$$\mathrm{vert}(C^\pi_\geq(s)) = A_>(s) \cup \mathrm{vert}(C^\pi_=(s)).$$

Thus

$$\begin{aligned}\mathrm{vert}\,(C^\pi_>(s)) &= \mathrm{vert}(C^\pi_\geq(s)) \cap C^\pi_>(s) \\ &= (\mathrm{vert}(C^\pi_=(s)) \cup A^\pi_>(s)) \cap C^\pi_>(s) = A^\pi_>(s). \quad (5.2)\end{aligned}$$

There exist several algorithms to compute all vertices of a polytope given by a system of linear inequalities. See Fukuda [Fuk00] for a discussion and related software. Bremner [Bre00] gives an extensive annoted bibliography on vertex and facet enumeration and related questions. Linear programming methods to find strictly improving vertices are discussed in the following section.

Two Realizations

We discuss how all strictly improving vertices for two realizations can be computed. We consider a family $\mathbf{E}$ of two realizations

$$(E_1, \mathbf{R}_1, \gamma_1) \text{ and } (E_2, \mathbf{R}_2, \gamma_2)$$

of an SAS. Let π be a policy for $\mathbf{E}$. We denote the value function and action-values for realization $(E_i, \mathbf{R}_i, \gamma_i)$ by V_i^π and Q_i^π. Let s be a state contained in both realizations, that is $s \in S_1 \cap S_2 \subset S_\mathbf{E}$. Let $A(s) = \{a_1, \ldots, a_{d+1}\}$,

$$\begin{aligned} \mathbf{c}_1 &= (Q_1^\pi(a_1, s), \ldots, Q_1^\pi(a_{d+1}, s)), \quad z_1 = V_1^\pi(s), \\ \mathbf{c}_2 &= (Q_2^\pi(a_1, s), \ldots, Q_2^\pi(a_{d+1}, s)), \quad z_2 = V_2^\pi(s). \end{aligned}$$

The polytope of improving policies $C_\geq^{\pi,\mathbf{E}}(s)$ is defined by the (in)equalities for the standard $d+1$-simplex,

$$\mathbf{1}\mathbf{x} = 1, \ \mathbf{x} \geq \mathbf{0}$$

with $\mathbf{1} = (1, \ldots, 1) \in \mathbb{R}^{d+1}$ and the additional inequalities

$$\mathbf{c}_1\mathbf{x} \geq z_1 \text{ and } \mathbf{c}_2\mathbf{x} \geq z_2. \tag{5.3}$$

A strictly improving vertex is a vertex of $C_\geq^{\pi,\mathbf{E}}(s)$ such that at least one from the above inequalities is strict. Recall from Section 1.8.3 that the improving policies $C_{i,\geq}^\pi(s)$ are given by the (in)equalities for the standard simplex and

$$\mathbf{c}_i\mathbf{x} \geq z.$$

Theorem 29. *We have*

$$\operatorname{vert}(C_>^{\pi,\mathbf{E}}(s)) = \big(\operatorname{vert}(C_{1,\geq}^\pi(s)) \cup \operatorname{vert}(C_{2,\geq}^\pi(s))\big) \cap C_>^{\pi,\mathbf{E}}(s).$$

Proof. We use the characterization of vertices from Corollary 16. So to compute the vertices of $C_\geq^{\pi,\mathbf{E}}(s)$, we have to choose all subsets of $d+1$ linearly independent rows from the $d+4$ (in)equalities for the standard simplex and Equation (5.3). Then we compute the unique solution $\mathbf{x} \in \mathbb{R}^{d+1}$ of the resulting system of linear equations and test whether the solution satisfies the remaining (in)equalities. For a strictly improving vertex at least one of the inequalities (5.3) must be strict. Hence the above remark on the improving policies $C_{i,\geq}^\pi(s)$ and computations in the proof of Theorem 17 imply the result. ■

This result gives us a method to compute all strictly improving vertices for two realizations. We first compute the improving vertices for the two realizations with the formulas given in Theorem 17 and then test for each vertex if one of the inequalities (5.3) is strict. Note that improving vertices in one realization correspond to policies in a state s with $\pi(a \mid s) > 0$ for at most two different actions, compare Theorem 18. Thus the strictly improving vertices for two realizations have the same property.

Three actions

We finally describe how all strictly improving vertices can be computed directly for a finite family of realizations $(E_i, \mathbf{R}_i, \gamma_i)_{i\in I}$ of an SAS if we have only three actions $A(s) = \{a_1, a_2, a_3\}$ in a state. We use this observation for the simulator `SimRobo`, see Section 8.

Let $s \in S_{\mathbf{E}}$ and J be the set of indices j with s a state of environment E_j, that is

$$J = \{j \in I : s \in S_j\}.$$

Let

$$\mathbf{c}_j = (Q_j^\pi(a_1, s), Q_j^\pi(a_2, s), Q_j^\pi(a_3, s)) \text{ and } z_j = V_j^\pi(s), \quad \text{for } j \in J.$$

Then the polytope of improving policies $C_{\geq}^{\pi,\mathbf{E}}(s)$ is defined by the inequalities

$$\mathbf{c}_j\mathbf{x} \geq z_j, \quad \text{for } j \in J, \tag{5.4}$$

and the (in)equalities for the standard simplex.

Theorem 30. *We have*

$$\operatorname{vert}(C_{>}^{\pi,\mathbf{E}}(s)) = (\bigcup_{j\in J} \operatorname{vert}(C_{j,\geq}^{\pi}(s))) \cap C_{>}^{\pi,\mathbf{E}}(s).$$

Proof. We use the characterization of vertices from Corollary 16. So to compute the vertices of $C_{\geq}^{\pi,\mathbf{E}}(s)$, we have to choose all subsets of 3 linearly independent rows from (in)equalities for the standard simplex and (5.4). Then we compute the unique solution $\mathbf{x} \in \mathbb{R}^3$ of the resulting system and test whether the solution satisfies the remaining (in)equalities. For a strictly improving vertex at least one of the inequalities $\mathbf{c}_j\mathbf{x} \geq z_j$ must be strict. We consider two cases to choose 3 linearly independent rows. For the first case we choose three linear independent equations either of the form

$$\mathbf{c}_j\mathbf{x} = z_j \quad \text{with } j = j_1, j_2, j_3 \in J$$

or

$$\mathbf{c}_j\mathbf{x} = z_j \quad \text{with } j = j_1, j_2 \in J \text{ and } \mathbf{1}\mathbf{x} = 1.$$

Recall that the policy $\mathbf{x} = \pi(- \mid s)$ in state s satisfies

$$\mathbf{c}_j\mathbf{x} = z_j \quad \text{for all } j \in J \tag{5.5}$$

by the Bellman equation (1.13). Hence for any possible choices of linear independent equation above the policy $\pi(- \mid s)$ is the unique solution of the resulting system but it is not an improving vertex by Equation (5.5).

For the second case we choose one equation of the form

$$\mathbf{c}_j\mathbf{x} = z_j, \quad \text{with } j \in J$$

and two equations from the (in)equalities for the standard simplex. The possible solutions for the resulting system are just the improving vertices $\text{vert}(C^{\pi}_{j,\geq}(s))$, see the proof of Theorem 17. ■

We obtain the following method to compute the strictly improving vertices. We first compute the improving vertices $\text{vert}(C^{\pi}_{j,\geq}(s))$ with the formulas given in Theorem 17 and then test for each vertex if all inequalities (5.4) are satisfied and one them is strict.

5.5.5 Linear Programming

Linear Programming methods can be used to decide whether there exists a strictly improving vertex and to find one. We refer to Fourer [Fou00] for references and a discussion of available software. See Chvátal [Chv83] for a description of Dantzig's simplex method [Dan51] and Schrijver [Sch86] for a survey of linear programming methods, complexity analysis and historical informations. We discuss the related linear programming problems to find strictly improving vertices.

Let $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i\in I}$ be a finite family of realizations of an SAS and π a policy for $\mathbf{E}$. Let $s \in S_{\mathbf{E}}$ and J be the set of indices j with s a state of environment E_j, that is

$$J = \{j \in I : s \in S_j\}.$$

We denote the value function and action-values for realization $(E_i, \mathbf{R}_i, \gamma_i)$ by V^{π}_i and Q^{π}_i. Let

$$\mathbf{c}_j = (Q^{\pi}_j(a_1, s), \ldots, Q^{\pi}_j(a_{d+1}, s)) \text{ and } z_j = V^{\pi}_j(s), \quad \text{for } j \in J,$$

Then the polytope of improving policies $C^{\pi,\mathbf{E}}_{\geq}(s)$ is defined by

$$\begin{aligned} \mathbf{c}_j\mathbf{x} &\geq z_j, \quad \text{for } j \in J, \\ \mathbf{1}\mathbf{x} &= 1, \text{ and } \mathbf{x} \geq \mathbf{0}. \end{aligned}$$

Let $i \in J$. We consider the following linear program

$$\begin{aligned} &\text{maximize } \mathbf{c}_i\mathbf{x} \\ &\text{subject to } \mathbf{c}_j\mathbf{x} \geq z_j, \quad \text{for } j \in J \setminus \{i\}, \\ &\mathbf{1}\mathbf{x} = 1, \text{ and} \\ &\mathbf{x} \geq \mathbf{0}. \end{aligned} \tag{5.6}$$

Let P_i be the polytope defined by the (in)equalities (5.6). Then the above linear program is

$$\max\{\mathbf{c}_i\mathbf{x} : \mathbf{x} \in P_i\}.$$

The maximum is attained in a vertex of P_i. If

$$\max\{\mathbf{c}_i\mathbf{x} : \mathbf{x} \in P_i\} > z_i,$$

then a vertex $\mathbf{v}$ of P_i that maximizes $\mathbf{c}_i\mathbf{x}$, is a strictly improving vertex.

If

$$\max\{\mathbf{c}_i\mathbf{x} : \mathbf{x} \in P_i\} = z_i, \quad \text{for all } i \in J,$$

then

$$\operatorname{vert}(C_>^{\pi,\mathbf{E}}(s)) = \emptyset$$

and thus the set of strictly improving policies is empty.

5.6 Policy Iteration

We introduce *policy iteration* for a finite family of realizations of an SAS, which computes a balanced stochastic policy, see [MR01a] and [MR01b].

We start with an arbitrary policy and compute its value functions and action-values for all realizations. Then we try to improve the policy by choosing a strictly improving vertex in as many states as possible, see Section 5.2 and 5.5.4. Once improved we evaluate the new policy in all realizations and again try to improve it. The algorithm terminates if there are no strictly improving vertices. Then the policy is balanced by definition. A formal description is given in Algorithm 8.

Input: a finite family of realizations $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i\in I}$ of an SAS
Input: a policy π for $\mathbf{E}$
Output: a balanced policy $\tilde{\pi}$ with $V_i^{\tilde{\pi}} \geq V_i^{\pi}$ for $i \in I$

```
π̃ ← π
repeat
    compute V_i^π̃ and Q_i^π̃ for i ∈ I
    for all s ∈ S_E do
        if vert(C_>^{π̃,E}(s)) ≠ ∅ then
            choose π'(− | s) ∈ vert(C_>^{π̃,E}(s))
            π̃(− | s) ← π'(− | s)
until vert(C_>^{π̃,E}(s)) = ∅ for all s ∈ S_E
```

Algorithm 8: Policy Iteration for several realizations

The balanced policies obtained depend on the starting policy. Different choices of strictly improving vertices may also lead to different balanced policies. Compare the experiment in Section 8.3.7. Since for one realization the improving vertices are just the improving actions (5.2), Algorithm 8 coincides with policy iteration for one MDP.

A geometric interpretation of one step of policy iteration for three states and two realizations can be seen in Figure 5.6. In state s_1 there are no strictly

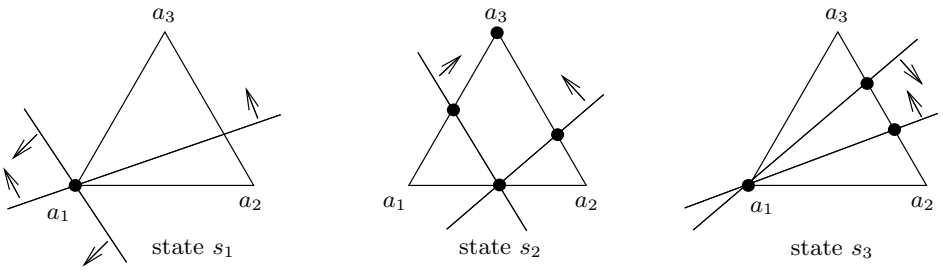

Figure 5.6: Strictly improving vertices in policy iteration for two MDPs

improving vertices. In state s_2 there are three strictly improving vertices, one of them is the action a_3. In state s_3 there are two, both of them a stochastic combination of a_2 and a_3.

In all our computational experiments the algorithm terminated after finitely many steps. So far, we have no formal proof of this proposition. Note however that the sequence of the value functions produced by the algorithm is increasing and bounded and therefore converges componentwise for each realization.

Conceptionally, policy iteration can be extended to the case of infinite families of realizations using extreme points of the set of strictly improving policies, see Section 5.5.3.

5.7 Approximate Policy Iteration

We formulate *approximate policy iteration* [MR03a] for a finite family of realizations of an SAS in the model-free case, when the environments are not explicitly known. The idea is analogous to approximate policy iteration for one MDP, see Section 1.12.3. We use approximate policy evaluation for each realization, Algorithm 5, and then improve the policy with the approximations.

Let $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i \in I}$ be a finite family of realizations of an SAS and π be a policy for $\mathbf{E}$. Let $i \in I$ and $Q_i \in \mathbb{R}^{\mathbf{A}_{S_i}}$ an approximation of the action-

values Q_i^π. We can compute an approximation $V_i \in \mathbb{R}^S$ of the value functions V_i^π by Equation (1.46).

For each realization we consider (strictly) improving policies and vertices for policy π using the approximations instead of exact action-values and value functions. Let $s \in S_{\mathbf{E}}$. We define the *improving policies* for π, $\mathbf{E}$ and *approximations* Q_i in state s by

$$C_{\geq}^{\pi,\mathbf{E},Q}(s) = \bigcap_{i\in I, s\in S_i} C_{i,\geq}^{\pi,Q}(s).$$

A formal description of the algorithm is given in Algorithm 9.

Input: a finite family of realizations $\mathbf{E} = (E_i, \mathbf{R}_i, \gamma_i)_{i\in I}$ of an SAS
Input: a policy π for $\mathbf{E}$
Output: a (balanced) policy $\tilde{\pi}$
  $\tilde{\pi} \leftarrow \pi$
  repeat
    approximate Q_i^π for $i \in I$
    compute V_i^π for $i \in I$
    for all $s \in S$ **do**
      compute $C_{>}^{\pi,\mathbf{E},Q}(s)$
      if $C_{>}^{\pi,\mathbf{E},Q}(s) \neq \emptyset$ **then**
        choose $\pi'(- \mid s) \in \mathrm{vert}(C_{>}^{\pi,\mathbf{E},Q}(s))$
        $\pi(- \mid s) \leftarrow \pi'(- \mid s)$

Algorithm 9: Approximate policy iteration for several realizations

Empirically, the policy improves well in the first iteration steps and then begins to oscillate, see the computational experiments in Section 8.3.9.

5.8 Bibliographical Remarks

Multi-criteria problems are discussed in the literature on reinforcement learning and on MDPs, where they are also called *vector-valued MDPs*. We summarize several approaches. In all proposed methods only deterministic policies are considered.

One approach is to find policies that maximize a positive linear combination of the different rewards, see for example White [Whi93, pp. 159].

Let $E = (S, \mathbf{A}, \mathbf{P})$ be an environment, $(\mathbf{R}_i)_{i\in I}$ a finite family of rewards, $(\alpha_i)_{i\in I}$ positive real numbers and γ a discount rate. Then an MDP is defined by the environment E, the rewards

$$\mathbf{R} = \sum\nolimits_{i\in I} \alpha_i \mathbf{R}_i,$$

and the discount rate γ. Now all algorithms including model-free methods can be applied to find (sub)optimal deterministic policies for this MDP. A problem of this approach is that the learned policies depend on the particular choice of the weights α_i. Note that policy improvement as described in Theorem 24 improves a policy for all possible families of weights $(\alpha_i)_{i\in I}$, since the set of improving policies is invariant if the rewards are multiplied by positive numbers.

The work of Singh and Cohn [SC98] is related to this approach. They merge different MDPs into a composite MDP that is based on the cross product of all states and actions and the sum of rewards. Shelton [She01] balances the rewards to restrict their influence on the learned policy.

Several authors discuss methods where action-values or policies are first learned for each reward and then a compromising policy is constructed based on heuristic considerations. Humphrys [Hum96] and Karlsson [Kar97] use Q-learning for each reward and then apply different action selection methods to identify a policy. Sprague and Ballard[SB03] apply Sarsa(0) instead of Q-learning to find policies that maximize the sum of rewards in a composite MDP. Mariano and Morales [MM00] apply distributed Q-learning agents for each reward and negotiate the resulting policy after each learning episode.

Gábor, Kalmár and Szepesvári [GKS98] fix a total order on the different rewards, discuss the existence of optimal policies and describe solution methods for this case.

For vector-valued MDPs Wakuta [Wak95] discusses a variant of policy iteration to find optimal deterministic policies, where in his context a policy is optimal if its value function is Pareto optimal for all states. An algorithm to find optimal deterministic policies for vector-valued MDPs was first proposed by Furukawa [Fur80]. The considerations are limited to one MDP with several rewards and deterministic policies.

White [Whi98] describes different vector-maximal sets of policies. Feinberg and Shwartz [FS94] discuss the case of several rewards and different discount rates. Novak [Nov89] uses multi-objective linear programming to find nondominated deterministic policies for vector-valued MDPs.

Finding good policies for multi-criteria MDPs can be interpreted as a special case of multiple criteria optimization, for a recent overview on this topic see Ehrgott and Gandibleux [MG02].

Chapter 6

Generalization over Environments

The fact that a policy learned in one environment can successfully be applied in other environments has been observed, but not investigated in detail. We discuss the influence of the environment on the ability to generalize over environments in [MR02] and [MR03b].

Given a policy for a family of realizations of an SAS, we consider the average utility of the policy in all realizations. We learn an optimal policy in one realization and then extend it to a policy for the family of realizations. We say that a realization, with an optimal policy having a high average utility, generalizes well over other environments. How can we find such an environment? Are there criteria to decide if an environment generalizes well over others? Figure 6.1 motivates this question, see the attached paper in Section 11.5.2 for details.

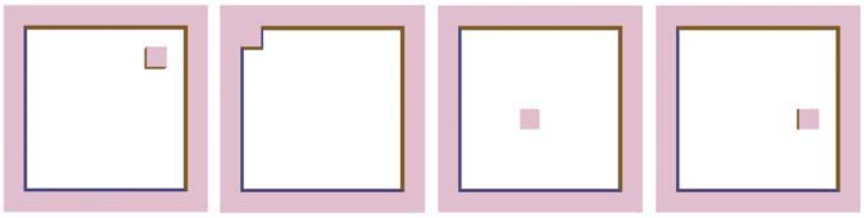

Figure 6.1: Four one-block grid worlds. Which one would you choose to learn in?

Chapter 7

MDP Package

The `MDP` package provides functions for two realizations with the same number of states and actions in each state. It implements the computation of strictly improving vertices and policy improvement and iteration. See Section 11.2.8 for a short description of all functions and examples.

The functions from the `MDP` package and the corresponding theorems and algorithms:

- `StrictlyImprovingVertices2`, Theorem 29.
- `PolicyImprovement2 PolicyIteration2`, Algorithm 8.

The following examples show the functions of the package for two realizations and how to use them, see the example file on the attached CD-ROM.

7.1 Two Realizations

Transition probabilities for two realizations with three states and three actions in each state.

```
> s:=3:a:=3:
> P:=[RandomTransitionProbability(s,a,10),\
>         RandomTransitionProbability(s,a,10)]:
> R:=[RandomReward(s,a,-2,2),RandomReward(s,a,-2,2)]:
> ER:=[ExpectedReward(P[1],R[1]),\ % rev1 p.120:  added linebreak
>         ExpectedReward(P[2],R[2])]:
```

A random policy.

```
> Pol:=RandomStochasticMatrix(a,s,10);
```

$$Pol := \begin{bmatrix} \frac{9}{10} & \frac{1}{10} & \frac{1}{5} \\ 0 & \frac{1}{10} & 0 \\ \frac{1}{10} & \frac{4}{5} & \frac{4}{5} \end{bmatrix}$$

Two discount rates.

```
> ga:=[1/2,1/2]:
> ERp:=[ExpectedRewardPolicy(ER[1],Pol),\
>         ExpectedRewardPolicy(ER[2],Pol)]:
> Pp:=[TransitionMatrix(P[1],Pol),\
>         TransitionMatrix(P[2],Pol)]:
```

The value functions

```
> Vp:=[ValueFunction(ERp[1],Pp[1],ga[1]),\
>         ValueFunction(ERp[2],Pp[2],ga[2])];
```

$$Vp := [\left[\frac{-1011238}{1005375}, \frac{-267634}{143625}, \frac{-2224538}{1005375}\right], \left[\frac{-65829}{132025}, \frac{-41979}{132025}, \frac{58271}{132025}\right]]$$

and action-values

```
> Qp:=[ActionValues(P[1],ER[1],Vp[1],ga[1]),\
>         ActionValues(P[2],ER[2],Vp[2],ga[2])]:
```

We compute the strictly improving vertices for the policy in the first state. The action-values for the two realizations in the first state

```
> Qps:=[LinearAlgebra:-Column(Qp[1],1),\
>         LinearAlgebra:-Column(Qp[2],1)];
```

$$Qps := \left[\begin{bmatrix} \frac{-133817}{143625} \\ \frac{815686}{1005375} \\ \frac{-1681909}{1005375} \end{bmatrix}, \begin{bmatrix} \frac{-193869}{264050} \\ \frac{126161}{264050} \\ \frac{428241}{264050} \end{bmatrix}\right]$$

and the value function

```
> Vps:=[Vp[1][1],Vp[1][2]];
```

$$Vps := [\frac{-1011238}{1005375}, \frac{-267634}{143625}]$$

Then the strictly improving vertices are

```
> StrictlyImprovingVertices2(Qps,Vps);
```

$$\left[\left[\begin{array}{c}1\\0\\0\end{array}\right],\left[\begin{array}{c}0\\1\\0\end{array}\right],\left[\begin{array}{c}\frac{9}{10}\\0\\\frac{1}{10}\end{array}\right],\left[\begin{array}{c}0\\\frac{670671}{2497595}\\\frac{1826924}{2497595}\end{array}\right]\right]$$

With the policy iteration algorithm for several realizations we compute a balanced policy with starting policy `Pol`

```
> balPol:=PolicyIteration2(P,ER,Pol,ga,'steps');
```

$$balPol := \left[\begin{array}{ccc}0 & 0 & 1\\ \frac{670671}{2497595} & \frac{1148507007892665}{2166047408033131} & 0\\ \frac{1826924}{2497595} & \frac{1017540400140466}{2166047408033131} & 0\end{array}\right]$$

How many improvement steps were used to obtain this balanced policy?

```
> steps;
```

$$2$$

Is this policy really balanced?

```
> ERp:=[ExpectedRewardPolicy(ER[1],balPol),\
>         ExpectedRewardPolicy(ER[2],balPol)]:
> Pp:=[TransitionMatrix(P[1],balPol),\
>         TransitionMatrix(P[2],balPol)]:
> Vp:=[ValueFunction(ERp[1],Pp[1],ga[1]),\
>         ValueFunction(ERp[2],Pp[2],ga[2])]:
> Qp:=[ActionValues(P[1],ER[1],Vp[1],ga[1]),\
>         ActionValues(P[2],ER[2],Vp[2],ga[2])]:
> IsBalanced2(Qp,Vp);
```

$$true$$

Chapter 8

SimRobo

The implementation and experiments for several realizations with the simulator `SimRobo`, see Section 3, are described. Figure 8.1 shows the complete UML-diagram of `SimRobo`, including classes for several realizations.

8.1 State Action Space and Realizations

We describe how we model the robot in several environments. The states, actions and rewards are as described in Sections 3.1 and 3.2. We consider sensor-based MDPs only, see Section 3.3.2. Then the state action space is given by the set S of all possible sensor values, that is

$$S = \{(s_0, s_1, s_2, s_3) : \ s_i = 0, \ldots, 5 \text{ for } i = 0, \ldots, 3\},$$

and the family $\mathbf{A} = (A(s))_{s \in S}$ of all possible actions, where

$$A(s) = \{0, 1, 2\} \quad \text{for all } s \in S.$$

We obtain an SAS $\mathbb{E} = (S, \mathbf{A})$. Let now a finite number of grid worlds and rewards be given. We can derive a realization of $\mathbb{E}$ from each grid world and family of rewards and obtain a family of realizations $\mathbf{E}$, see Section 5.1. Policies can then be defined on the set $S_{\mathbf{E}}$ of all states for the family of realizations $\mathbf{E}$.

In the `SimRobo` implementation, first the grid worlds and rewards are specified, then the MDPs are derived. They form a family of realizations and the set of all states for the family is computed.

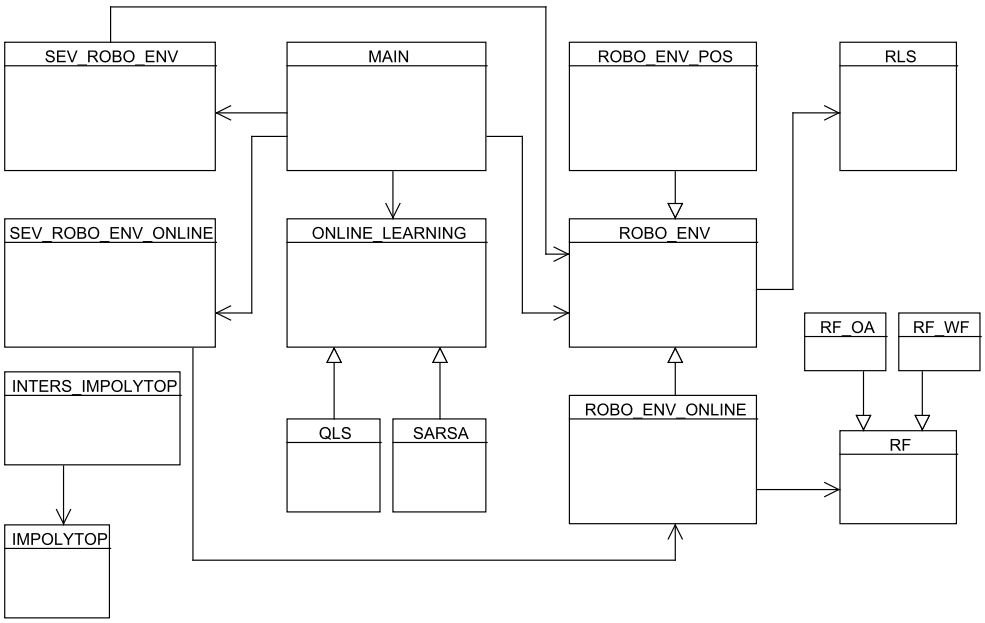

Figure 8.1: `SimRobo` class structure

8.2 Implementation

8.2.1 Environments

The family of realizations is represented by a `vector` of pointers to `robo_env` objects. The class `sev_robo_env` contains this vector. In the constructor, the list of all states of the realizations is computed. The function `pol_improv` computes an improved policy for all `robo_env` objects by first computing the action-values and value function of each realization and then choosing randomly an improving vertex in each state, compare one iteration of Algorithm 8. There are two versions of `pol_improv`, one that improves a policy by choosing strictly improving vertices, one that chooses only strictly improving actions to improve the policy as long as it is possible. The latter version is a deterministic variant of Algorithm 8. For details, see the header file Listing 10.

8.2.2 Improving Polytopes and Strictly Improving Vertices

Improving polytopes and (strictly) improving vertices are implemented. They are represented by the class `impolytop`, that contains the value function, the

action-values and the (strictly) improving and equivalent vertices for one realization, see Section 1.6. In the class `inters_impolytop` the strictly improving vertices are computed when the constructor is called. See the remarks in Section 5.5.4 on how to compute the strictly improving vertices for a finite family of realizations with three actions. The polytopes are given by a vector of `impolytop` objects. The function `get_simprovpol` returns a vector of strictly improving vertices. In the class `sev_robo_env`, `inters_impolytop` and `impolytop` objects are used for computations. The header files can be found in Listing 9.

8.2.3 Model-free

For the model-free case, the class `sev_robo_env_online` is implemented. It contains a vector of pointers to `robo_env_online` objects. The function `pol_improv` runs approximate policy evaluation for each `robo_env_online` object and then approximate policy improvement, see one step of Algorithm 9. It uses `inters_impolytop` and `impolytop` objects to compute the (strictly) improving vertices for the approximations. Consult the header file Listing 11 for details.

8.2.4 User Interface

The left control panel is designed to set options and run algorithms for several environments. It provides spinners and edit boxes to choose up to four grid worlds and rewards. The MDPs and the family of realizations is derived from the chosen grid worlds and rewards by pressing `Build Envs`. See Figure 3.5, where `SimRobo` with all control panels is shown. The policies obtained by learning in several environments are stochastic and stored in a different format. To evaluate or show stochastic policies use the buttons `Policy Evaluation` and `Run Policy` in the left panel. The controls are shortly explained in the next section, where they are used to conduct a series of experiments.

8.3 Experiments

We use the settings for the discount rate, step-size parameters and precision and the notion of normalized utilities described in Section 3.5. In all experiments first the grid worlds and rewards are fixed to derive the MDPs that form a family of realizations for which policies are learned. All experiments are conducted using sensor-based MDPs.

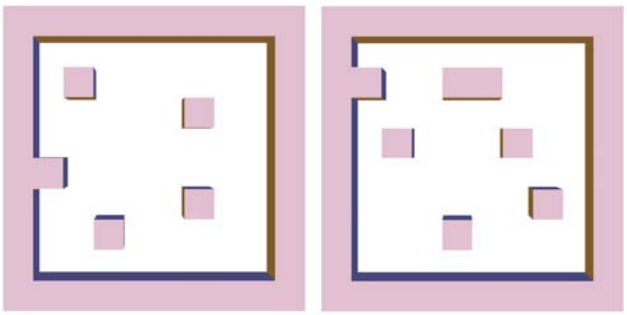

Figure 8.2: Two grid worlds to learn obstacle avoidance

The experiments can be repeated using `SimRobo` and following the instructions given in each experiment. See [MR01a], [MR03a] for other experiments with `SimRobo`.

8.3.1 Obstacle Avoidance - two Realizations

We want the robot to learn obstacle avoidance in the two grid worlds displayed in Figure 8.2. The rewards for obstacle avoidance are defined in Section 3.2.1. We derive two sensor-based MDPs and obtain two realizations that form the family of realizations to learn with. We start with the random policy. Then policy iteration is run by consecutively evaluating the policy in each realization and then improving the policy for all realizations, see Algorithm 8. In the policy improvement step we randomly choose an improving vertex, see Section 5.5.4.

The experiment is carried out twice. Figure 8.3 *left* shows the normalized utility of the policy for both realization after each improvement step for the first experiment and Figure 8.3 *right* for the second experiment. The solid line represents the progress of the policy in the MDP derived from the left grid world, the dashed line shows the progress in the MDP derived from the right grid world. We see that the policies improve for both realizations after each improvement step. In these experiments the algorithm terminated with different balanced policies that are close to optimal in both grid worlds after four iterations where one iteration consists of policy evaluation and improvement.

The following table shows the utilities of the balanced policies obtained in both realizations for the two experiments:

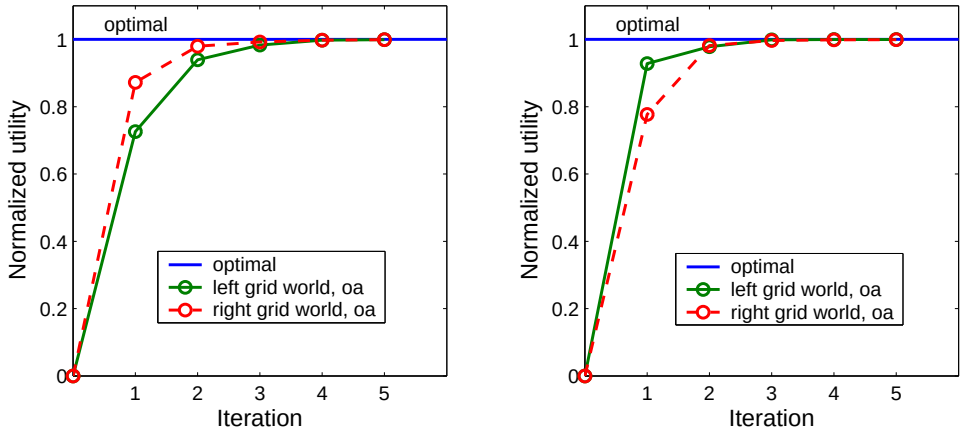

Figure 8.3: *left:* Progress of policy iteration for obstacle avoidance in two realizations, first experiment *right:* Second experiment

policy, in grid world:	*left*	*right*
balanced, experiment 1	18.428	8.867
balanced, experiment 2	18.435	8.866

Comparing the utilities of the balanced policies with the utility of optimal policies in each realization given in the following table, we see that they are close to optimal in both.

policy, in grid world:	*left*	*right*
optimal	18.449	8.882
random	−6.060	−6.960

To repeat this and the following experiments with `SimRobo` use the buttons on the left and lower control panel. To set the realizations use the grid world and reward spinners and list boxes in the panel `Grid World / Rewards`, where up to four realizations can be initialized. If the grid world is set to −1 it will not be considered, when the family of realizations is built using the button `Build Envs`. Leave the checkerbox on `Sensors` for all experiments.

For this experiment choose grid worlds number 5 and 7 and rewards `OA` for obstacle avoidance for both. Leave the third and fourth grid world set to −1. Now press `Build Envs` to build the two MDPs and the family of realizations. Select `Show utilities`. Open the `Policy Iteration` rollout and press `Random Policy`. Change the currently displayed grid world by selecting grid world number 5 or 7 and the `Rewards` to `Obstacle Avoidance` in the right control panel. Note that only one grid world can be displayed at

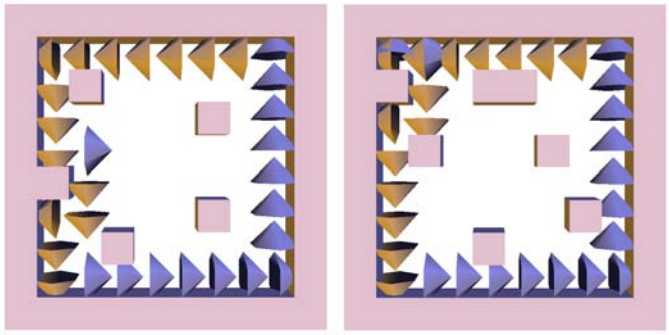

Figure 8.4: The robot following a balanced policy learned for both grid worlds

the same time and that for policy evaluation the according rewards have to be set.

Now press the `Policy Improvement` and `Policy Evaluation` buttons consecutively while observing the utility of the stochastic (improved) policy in the currently displayed grid world. If you want to see the robot following the current policy in the currently displayed grid world select `Run policy`. The utilities for both realizations can be printed to the console window by pressing `Policy Evaluation` in the `Console Output` rollout.

8.3.2 Wall Following - two Realizations

Now we want the robot to learn wall following for the two grid worlds of the experiment above. We choose the rewards for wall following described in Section 3.2.2 and derive two MDPs. Again we run policy iteration. In Figure 8.4 an obtained balanced policy is displayed in both grid worlds.

The following table shows the utility of a balanced policy for both realizations.

policy, in grid world:	*left*	*right*
balanced	10.060	9.900

Comparing the utility of the balanced policy with the utility of optimal policies in each realization, we see that it is close to optimal in both.

policy, in grid world:	*left*	*right*
optimal	10.098	9.920
random	−12.765	−11.282

The experiment can be repeated with `SimRobo` similar to the experiment above. Just choose `Grid Worlds` number 5 and 7 and rewards `WF` for wall following. Then repeat the steps described in the previous experiment.

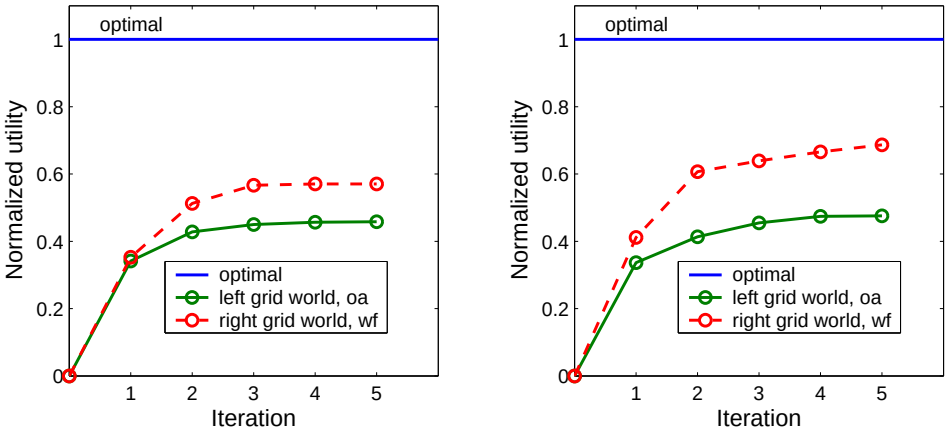

Figure 8.5: *left:* Progress of policy iteration for obstacle avoidance and wall following in one environment, first experiment *right:* Second experiment

8.3.3 Obstacle Avoidance and Wall Following - one Environment

We learn a policy that performs obstacle avoidance and wall following simultaneously in one grid world. Obviously it is difficult to find one policy that satisfies both tasks. This is a multi-criteria reinforcement problem, where we have two rewards in one environment, see Section 5.1.

We choose the grid world displayed in Figure 8.2 *right* and derive two realizations from the grid world, one with rewards for obstacle avoidance and one with rewards for wall following. We start with the random policy and run policy iteration. The progress of policy iteration is shown in Figure 8.5 *left* for the first experiment and in Figure 8.5 *right* for the second experiment.

In general balanced policies can be stochastic. Figure 8.6 shows two states, in which the obtained balanced policy chooses between different actions. In the state displayed on the left, the robot chooses to move forward with probability 0.675 and to move left with probability 0.325. If the robot moves forward it receives a reward of $+1$ for wall following, but -1 for obstacle avoidance. If it moves left it receives a value of -1 for both rewards. On the long run it is better for obstacle avoidance to turn left. In the state displayed on the right side, it chooses to go forward, left or right with the same probability of one third as in the random policy. So the policy in this state did not change.

In `SimRobo` choose `Grid World` number 7 with `OA` for obstacle avoidance and `Grid World` number 7 with `WF` for wall following in the `Grid World /`

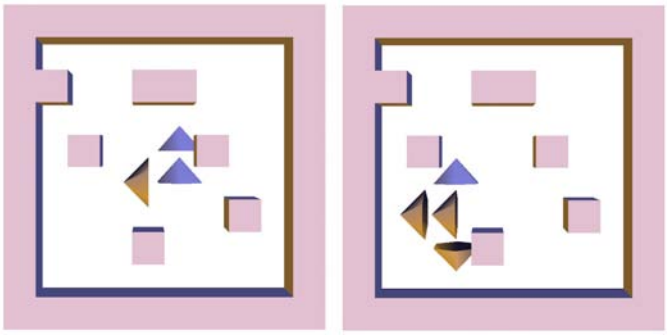

Figure 8.6: Two states where the obtained balanced policy is stochastic

`Rewards` panel. Leave the third and fourth `Grid World` set to -1. Now press `Build Envs` to build the two MDPs.

Select `Show utilities`, open the `Policy Iteration` rollout and press `Random Policy`. Press the `Policy Improvement` and `Policy Evaluation` buttons consecutively while observing the utility of the stochastic (improved) policy in the currently displayed grid world with the according rewards. To see if a policy is stochastic or deterministic in a state, open the `Advanced` rollout and select `Draw robot steps`. Now the number of policy steps given by `Steps` will be applied while showing the robot in the successor positions. Choose one step and observe the successor position. If the policy is stochastic various successor position are shown since each action is taken according to the probability given by the policy.

8.3.4 Obstacle Avoidance and Wall Following - two Realizations

The robot should learn to avoid obstacles in the grid world displayed in Figure 8.7 *left* and to follow the wall in the grid world displayed in Figure 8.7 *right*. Figure 8.8 *left* and *right* shows the progress of policy iteration for two experiments. In both experiments the policies became balanced after four iterations and cope well with the possibly contradicting task to do obstacle avoidance in one grid world and wall following in another grid world.

To repeat this experiment in `SimRobo` choose grid world number 19 with `OA` for obstacle avoidance and grid world number 20 with `WF` for wall following in the `Grid World / Rewards` panel. Leave the third and fourth grid world set to -1 and press `Build Envs` to build the two MDPs. Open the `Policy Iteration` rollout and press `Random Policy`. Now proceed as described in the previous experiments to run policy iteration and show the obtained utility

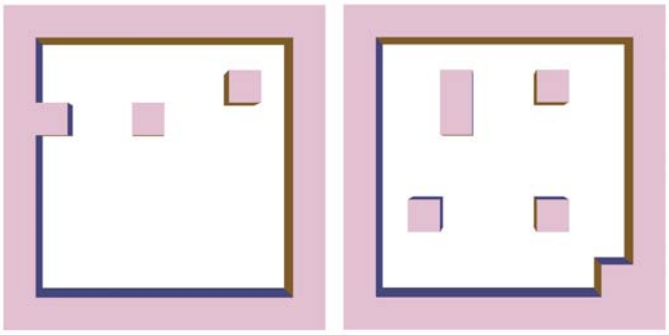

Figure 8.7: *left:* Grid world to learn obstacle avoidance *right:* Grid world to learn wall following

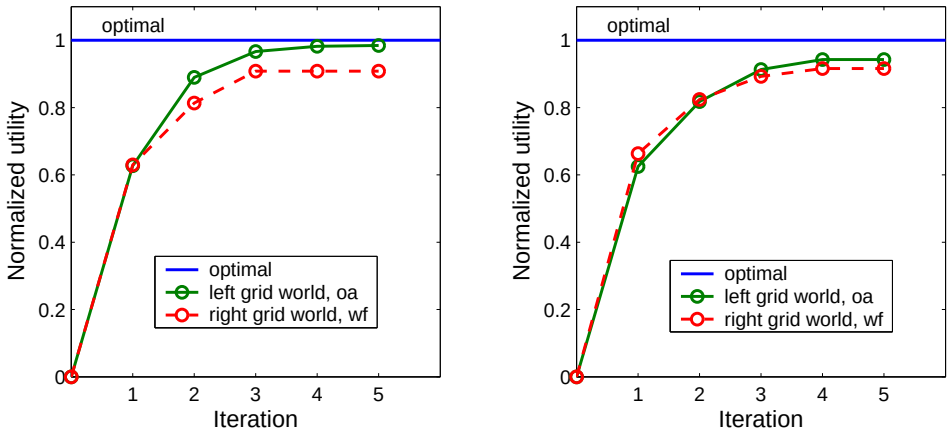

Figure 8.8: *left:* Progress of policy iteration for obstacle avoidance and wall following in two realizations, first experiment *right:* Second experiment

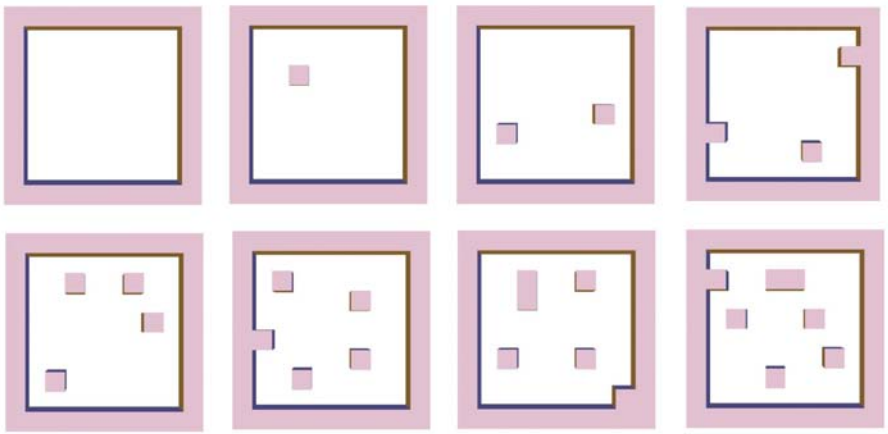

Figure 8.9: Eight grid worlds to learn obstacle avoidance

after each policy iteration step.

8.3.5 Many Realizations

Policy iteration is applied to learn obstacle avoidance for eight different grid worlds, see Figure 8.9.

After deriving the eight MDPs from the grid worlds with rewards for obstacle avoidance we obtain a family of realization and apply policy iteration, starting with the random policy. The progress of the normalized utility for one experiment is shown in Figure 8.10. We see that the utility of the policy increases dramatically in the first improvement step for all realizations. The algorithm terminated after five iterations.

We conduct several experiments and obtain different balanced policies. The normalized utilities of two balanced policies are shown in the following table.

policy, in grid world	0	1	2	3	4	5	6	7
balanced 1	0.999	0.903	0.912	0.959	0.690	0.949	0.817	0.932
balanced 2	0.999	0.908	0.899	0.956	0.690	0.953	0.815	0.934

In `SimRobo` the eight realizations can be built by pressing the button `Grid Worlds 0-7,OA` in the `Advanced` Rollout. It sets the eight grid worlds 0 to 7 with rewards for obstacle avoidance and builds the family of realizations.

Select `Show utilities`, open the `Policy Iteration` rollout and press `Random Policy`. Change the currently displayed grid world by selecting one of the grid worlds, number 0 to 7, and `Obstacle Avoidance` as `Rewards` in the right control panel. Now press the `Policy Improvement` and `Policy`

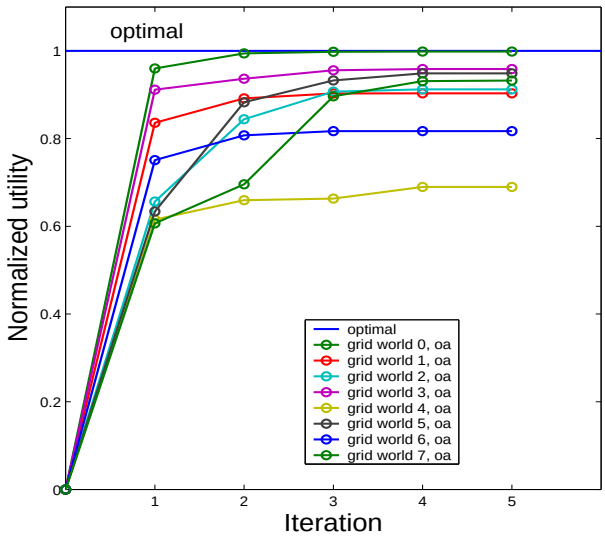

Figure 8.10: Progress of policy iteration for obstacle avoidance in eight realizations

`Evaluation` buttons consecutively while observing the utility of the stochastic (improved) policy in the currently displayed grid world. If you want to see the robot following the current policy in the currently displayed grid world select `Run policy`. The utilities for all realizations can be printed to the console window by pressing `Policy Evaluation` in the `Console Output` rollout.

8.3.6 One Realization

As a special case the algorithms for several realizations can be used for one realization. The robot should follow the wall in the grid world displayed in Figure 8.2 *left*. The difference to the experiment discussed in Section 3.5.1 is the choice of the strictly improving actions in the policy improvement step. In this experiment a randomly chosen improving action is selected instead of the greedy choice. The progress of the utilities was different for each run and the algorithm terminated with the optimal policy after five, six and four iterations for the three experiments. Figure 8.11 shows the progress of policy iteration. The progress of the normalized utilities is given in the following table.

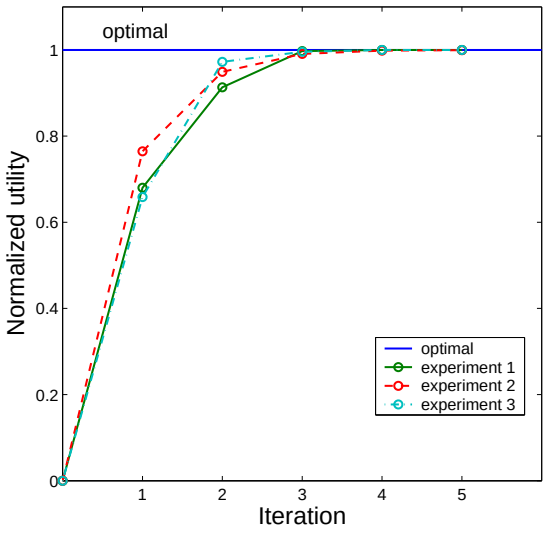

Figure 8.11: Progress of policy iteration for obstacle avoidance in one realization, three experiments

iteration	experiment 1	experiment 2	experiment 3
1	0.680	0.765	0.659
2	0.913	0.949	0.972
3	0.997	0.990	0.996
4	0.999	0.998	1.0
5	1.0	0.999	1.0
6	1.0	1.0	1.0

To repeat the experiment with `SimRobo` choose grid world number 7 with `WF` for wall following as rewards. Leave the second, third and fourth grid world set to -1. Press `Build Envs` to build the MDP. Change the currently displayed `Grid World` to grid world 7 and `Rewards` to `Wall Following` in the right control panel. Open the `Policy Iteration` rollout and press `Random Policy`. Now apply policy iteration as described in the previous experiments.

8.3.7 Improvement of an Optimal Policy

We consider wall following in the two grid worlds displayed in Figure 8.7. We want the robot to learn a policy that is optimal in the left grid world and as good as possible in the right grid world. After deriving the two MDPs we first learn an optimal policy in the left grid world by applying policy iteration in the corresponding MDP. Then we try to improve the obtained policy for the other MDP. So instead of starting policy iteration with the random policy

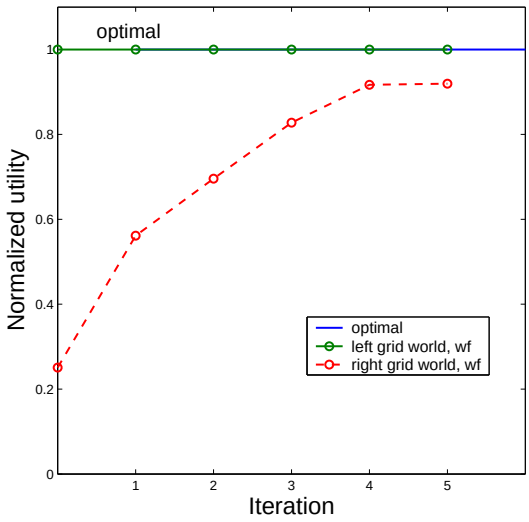

Figure 8.12: Progress of policy iteration for two realizations with the starting policy being optimal in one realization

we start with the obtained optimal policy. We have to extend the optimal policy to be defined in states that are contained only in the second MDP. We choose the random policy in these states. Figure 8.12 shows the progress of policy iteration for one experiment. Observe that the policy remains optimal in the first MDP and gets better in the second MDP.

To repeat this experiment in `SimRobo` first we have to learn an optimal policy for one MDP. Choose grid world number 19 with `WF` for wall following as rewards in the `Grid World / Rewards` panel. Leave the second, third and fourth grid world set to -1. Now press `Build Envs` to build the MDP. Select `Show utilities`. Change the currently displayed grid world by selecting grid world number or 19 and the `Rewards` to `Wall Following` in the right control panel. Now apply policy iteration using the left control panel as described in the previous experiment to obtain an optimal policy.

To improve the obtained optimal policy we have to build the two realizations. Choose grid worlds number 19 and 20 with `WF` for wall following as rewards in the `Grid World / Rewards` panel and leave the third and fourth grid world set to -1. Now press `Build Envs` to build the two realizations. Instead of pressing `Random Policy` we directly proceed with policy iteration described in the above experiments.

8.3.8 Joint Optimal Policies

Obstacle avoidance for two grid worlds should be learned. We select the first two grid worlds displayed in Figure 8.9, the empty and the grid world with one obstacle. Again we apply policy iteration and observe that the obtained balanced policy is optimal for each realization. See Section 5.4, where joint optimal policies are discussed. In the following table we give the progress of the utility for two experiments. In the first experiment the policy becomes optimal for both realizations after three iterations, in the second experiment after four iterations.

	experiment 1		experiment 2	
iteration	one obstacle	empty	one obstacle	empty
1	0.967	0.96	0.978	0.944
2	1.0	0.997	0.996	0.993
3	1.0	1.0	1.0	0.999
4	1.0	1.0	1.0	1.0

The experiment can be repeated with `SimRobo` as follows. Choose grid worlds number 0 and 1 and `OA` for both as rewards. Press `Build Envs` to build the two MDPs and the family of realizations. Select `Show utilities`. Change the currently displayed grid world to grid world 0 or 1 and the `Rewards` to `Obstacle Avoidance` in the right control panel. Now apply policy iteration as described in the previous experiments while observing the utility of the stochastic (improved) policy.

8.3.9 Approximate Policy Iteration - two Realizations

The experiment described in Section 8.3.4 is repeated for the model-free case. The robot should avoid obstacles in the grid world displayed in Figure 8.7 *left* and follow the wall in the grid world displayed in Figure 8.7 *right*. We apply approximate policy iteration by consecutively applying approximate policy evaluation and improving the policy with the obtained approximations, see Algorithms 1.12.2 and 9.

Approximate policy evaluation is performed for each realization as described in Section 3.5.3. We carry out several experiments with different numbers of update steps for approximate policy evaluation. After each improvement step the exact utility of the obtained policy is computed in each realization. In Figure 8.13 the normalized utilities for experiments with 500 and 1000 update steps are shown. We observe that the utility can decrease due to the error of the approximation.

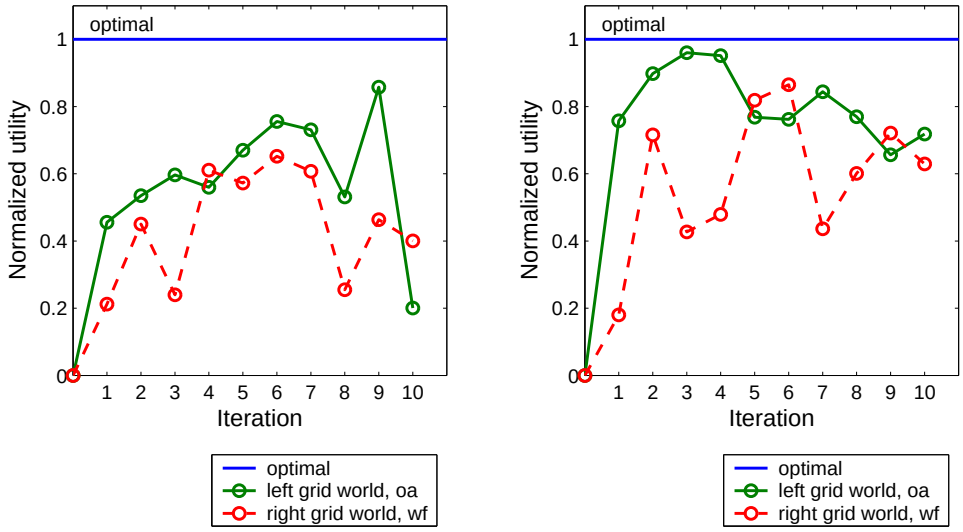

Figure 8.13: *left:* Progress of approximate policy iteration for two realizations for 500 update steps *right:* Progress for 1000 update steps

Figure 8.14 shows the progress of approximate policy iteration for 10000 and 50000 update steps. The policies get more stable with a high number of update steps for approximate policy evaluation.

To repeat this experiment in `SimRobo` choose grid world number 19 with rewards `OA` and grid world number 20 with rewards `WF`. Leave the third and fourth grid world set to -1. Now press `Build Envs` to build the two MDPs and the family of realizations. Select `Show utilities`. Change the currently displayed grid world by selecting grid world number 19 or 20 and the corresponding reward in the right control panel.

Now open the `Approximate` rollout and press `Random Policy`. Choose the desired number of `Steps` for approximate policy evaluation. Press the `Policy Improvement` button in the `Approximate` rollout and the `Policy Evaluation` button consecutively while observing the utility of the stochastic (improved) policy in the currently displayed grid world. If you want to see the robot following the current policy in the currently displayed grid world select `Run policy`. The utilities for both realizations can be printed to the console window by pressing `Policy Evaluation` in the `Console Output` rollout.

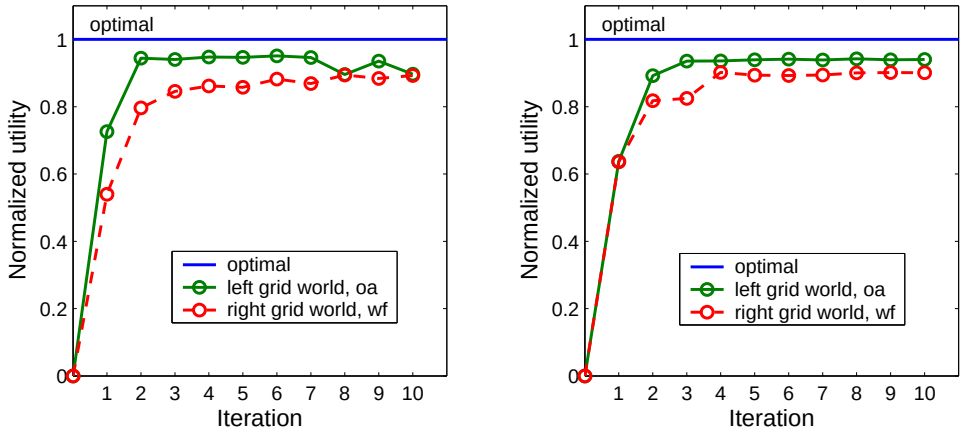

Figure 8.14: *left:* Progress of approximate policy iteration for two realizations for 10000 update steps *right:* Progress for 50000 update steps

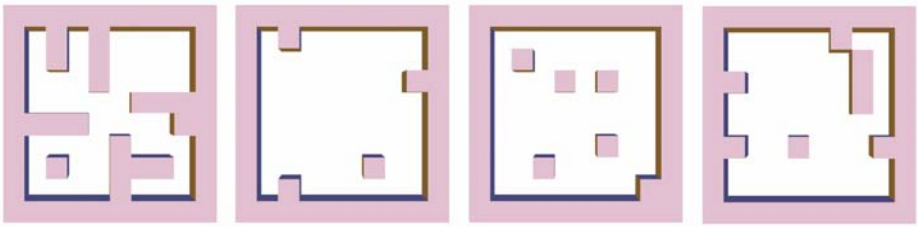

Figure 8.15: Four grid worlds to learn wall following

8.3.10 Approximate Policy Iteration - four Realizations

Wall following should be achieved in the model-free case for the four different grid worlds displayed in Figure 8.15. We use the rewards for obstacle avoidance for the four grid worlds and obtain four realizations. Again policy iteration is performed for 10000 and 50000 policy evaluation update steps. The progress of the normalized utilities is shown in Figure 8.16.

To repeat this experiment in `SimRobo` choose grid world numbers 11, 14, 16 and 18 with rewards `WF`. Press `Build Envs` to build the four MDPs and the family of realizations. Select `Show utilities`. Change the currently displayed grid world by selecting grid world number 11, 14, 16 or 18 and `Rewards` to `Obstacle Avoidance` in the right control panel.

Now open the `Approximate` rollout and repeat the process described in the previous experiment. Note that the policy improvement step may take

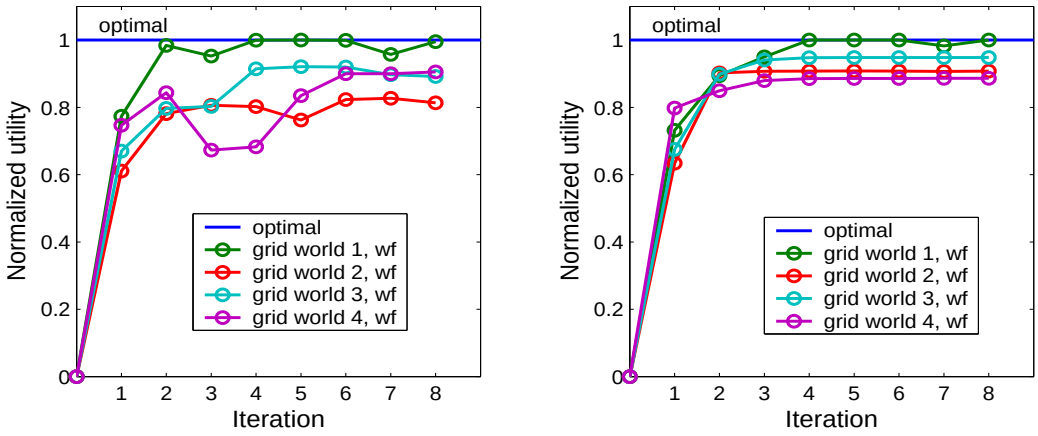

Figure 8.16: *left:* Progress of approximate policy iteration for four realizations for 10000 update steps *right:* Progress for 50000 update steps

some time depending on your computer for computing the 50000 policy evaluation updates in the four MDPs.

Chapter 9

RealRobo

`RealRobo`, see Section 4, is used to conduct an experiment for several realizations. We first describe how the sensors of the robot and the chosen actions define an SAS and how realizations are obtained. Then we focus on a specific experiment in which we want the robot to learn a *target following* behavior, inspired by Bendersky [Ben03] and Bendersky and Santos [BS03]. It consists of learning a policy for one environment and two different families of rewards.

9.1 State Action Space and Realizations

Consider the robot and its infrared sensors to perceive obstacles. All possible sensor values represent the possible states for the robot. Thus an SAS is given by the set S of all possible sensor values, that is

$$S = \{(s_0, ..., s_7) : s_i \in \tilde{S} \quad \text{for } i \in 0 \dots 7\},$$

where $\tilde{S} = \{0, 1/1023, \dots, 1022/1023, 1\}$ and a family of possible actions. For example we choose the actions defined in Section 4.3, where

$$A(s) = \{a : a \in -90 \dots 90\}, \quad \text{for all } s \in S.$$

Let now a finite number of wooden boxes and rewards be given. We can derive a realization of the SAS from each box and family of rewards and obtain a family of realizations. In real world applications policies are generally defined for the SAS.

9.2 Target Following

Bendersky [Ben03] considers two different families of rewards to achieve a target following behavior, one to keep a certain distance and one to keep a specified angle with respect to a target.

For keeping distance only forward and backward movements, for keeping the angle, only turns by a given angle are considered. The robot is put in front of a fixed target and Q-learning is applied for keeping distance and for keeping the angle separately. Then a new policy is built by combining the two (sub)optimal policies obtained by having the new policy execute the actions of the two policies consecutively in each state. The new policy performs the desired target following behavior, also for a moving target. This concept of a policy does not fit into the previously described model. It works with actions that are a combination of two subactions. In this context the term *task decomposition* is used, see Brooks [Bro86].

Since the two actions are learned independently, they may be contradictory in terms of the received rewards. Imagine that the robot applies a backward movement and turns to the left. This may be good to keep the angle but bad to keep the distance. Using an adaptation of approximate policy iteration for several realizations a policy can be learned that satisfies both rewards simultaneously.

9.2.1 Rewards

The two rewards for keeping distance and keeping the angle are defined. For keeping distance, the robot should look towards the target and keep a certain distance to it. If its maximal front sensor value is in a certain range, it gets rewarded. If the maximal sensor value is close to the desired range, it gets a neutral reinforcement. In any other case it will be punished.

The rewards depend only on the successor sensor values $s' = (s'_0, \ldots, s'_7)$. Let

$$m = \arg\max_{i \in 1 \ldots 4} s'_i$$

be the number of the sensor with the maximal value of the four front sensors. We define the rewards for keeping distance by:

$$R_{kd}(s') = \begin{cases} 1, & \text{if } 0.2 \leq s'_m \leq 0.8, \\ 0, & \text{if } 0.1 \leq s'_m < 0.2, \\ 0, & \text{if } 0.8 < s'_m \leq 0.9, \\ -1, & \text{otherwise.} \end{cases}$$

For keeping the angle, the robot should look towards the target and keep a certain angle to it. If the target is far away it will receive a punishment.

If the target is straight ahead, it gets rewarded. If the target is a bit to the left or right a neutral reinforcement is given. In any other case it will be punished. The rewards for keeping the angle are defined as follows:

$$R_{ka}(s') = \ -1, \quad \text{if } s'_m \leq 0.2,$$

and in any other case

$$R_{ka}(s') = \begin{cases} +1, & \text{if } m \in \{2,3\}, \\ 0, & \text{if } m \in \{1,4\}, \\ -1, & \text{otherwise.} \end{cases}$$

See Bendersky [Ben03].

9.2.2 States and Actions

We model the target following task described above. The four front sensors are considered to define the states. The actions are to turn an integer angle between -90 and 90 degrees followed by moving forward or backward between -100 and 100 steps, where one step is $1/12$ mm. We have $2^{40} \approx 10^{12}$ states and 36381 actions and the action values Q are contained in $\mathbb{R}^{2^{40} \cdot 36381}$.

In practice, to implement and apply a new algorithm in a task with a high dimensional state and action space, often a discrete representation of states and actions is considered first. This way the action-values can be stored in a table and the algorithm and its functionality can be tested more easily. Note that the sets of actions are assumed to be equal for each state.

Once the algorithm is implemented adaptive networks or function approximation methods can be applied, see Section 4.4. In the following we describe the discrete representation used to test approximate policy iteration for two realizations in the target following task.

To decide which states and actions are used for the discretization we choose a fixed number of actions uniformly distributed among the action space. Then a sample robot run is conducted by putting the robot in a start position, applying random actions and observing the sensor values received for a given number of iterations. The received sensor values are recorded and used to find representative states, that are states close to the sensor values received using a distance. Once a list of states is selected, the sample run is repeated and the sensor values and their corresponding states are observed. If a state occurs very often a new state close to it can be added, if a state does not occur very often it can be omitted.

In another sample run, the successor states and rewards are recorded. If applying an action in a state leads to many different successor states, then

the discretization of states and/or actions is usually made finer. Also if different rewards are observed when the same action is applied in the same state yielding to the same successor state. Compare Bertsekas and Tsitsiklis [BT96, pp. 341].

We use a simplified representation by mapping the sensor values to an integer value between 0 and 10. Let

$$\tilde{s} = (\tilde{s}_1, ..., \tilde{s}_4) \in \{0, 1/1023, \ldots, 1022/1023, 1\}^4$$

be the vector of the four front sensor values. Then we define the integer value representation of $\tilde{s}$ by

$$\hat{s} = (\hat{s}_1, ...\hat{s}_4) \text{ with } \hat{s}_i = \lfloor 10 \cdot \tilde{s}_i \rfloor, \quad \text{for } i = 1 \ldots 4.$$

For target following we select the 13 integer vectors given in the following table to be the representative states:

$(0,0,0,0)$	$(0,0,0,3)$	$(0,0,0,6)$	$(0,0,0,9)$	$(0,0,6,6)$
$(9,9,9,9)$	$(3,0,0,0)$	$(6,0,0,0)$	$(9,0,0,0)$	$(6,6,0,0)$
	$(9,9,0,0)$	$(6,9,9,6)$	$(0,0,9,9)$	

A vector of sensor values $\tilde{s}$ is represented by the vector of the table that is closest to the integer sensor values $\hat{s}$ using the L_2-norm. The vectors in the table define the set of states S.

We select the following degrees for turning by an angle

$$-20° \quad -10° \quad 0° \quad 10° \quad 20°$$

and the following steps to move

$$-60 \quad -30 \quad 0 \quad 30 \quad 60$$

to form the representative actions. We used small angles, since for larger angles many different successor states occurred in our discretization when actions were applied in the same states.

The resulting 25 representative actions define the set $A = A(s)$ of actions for all states s. Note that the robot is also allowed to stand still. Using this discretization the state and action space is reduced and the action values can be computed by a table in $\mathbb{R}^{13 \cdot 25}$.

We consider the two families of rewards from the previous section and observe the transitions by applying actions in the wooden box.

9.3 Approximate Policy Iteration

Approximate policy iteration for several realizations is applied by consecutively applying approximate policy evaluation and policy improvement, see Algorithms 5 and 9. A variant of approximate policy iteration arises as in Section 4.4. Exploration techniques have to be used. For target following we use the simple exploration technique of applying a policy that consecutively chooses greedy actions for 6 iterations and then random actions for 2 iterations.

The approximate policy iteration algorithm is implemented similar to `SimRobo`, see Section 8.2. We represent stochastic policies by a triple

$$(a, \tilde{a}, p) \in A^2 \times [0, 1].$$

To apply the random policy we use a flag. If it is set the robot chooses random actions, if not it chooses the action a with probability p and the action $\tilde{a}$ with probability $(1 - p)$. This representation is suitable since the strictly improving vertices for two realizations can be described by such a triple, compare the remarks on two realizations in Section 5.5.4. A policy π is represented by a table, where for each $s \in S$ a triple $(a, \tilde{a}, p)$ is stored.

To compute the approximation of the action-values we use another table, where for each $s \in S$ and $a \in A$ the approximation $q_1 \in \mathbb{R}$ of the action-value for policy π in the first realization and the approximation $q_2 \in \mathbb{R}$ of the action-value for policy π in the second realization are stored.

9.4 Target Following Experiment

We put the robot in an empty wooden box with a fixed target, see Figure 9.1. The target is a cylindric shaped paper with a diameter of 55 mm.

The idea of using a fixed target to learn a target following behavior is due to Bendersky [Ben03, pp. 32]. With a moving target the robot would loose it very often especially at the beginning of the learning phase and thus observe very few positive rewards being in state $(0, 0, 0, 0)$ most of the time.

While learning, the robot is provided with a reflex. If it gets too close to the target or if it gets too far from the target, it undoes the last action executed. This reflex assures that the robot does not loose the target.

We start with the random policy and run approximate policy evaluation for a fixed number of steps for the two realizations. The two approximations of the action-values q_1 and q_2 for the policy are updated simultaneously with respect to the corresponding rewards. Then the policy is improved by approximate policy improvement for two realizations and again evaluated.

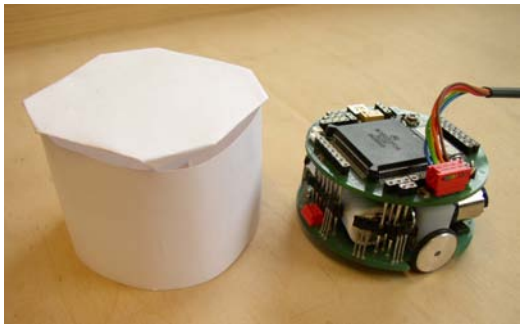

Figure 9.1: The robot and a fixed target to learn target following

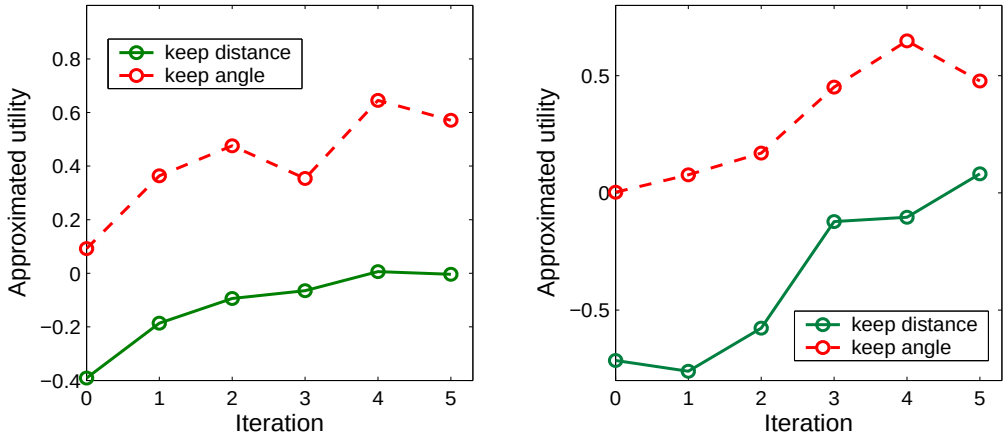

Figure 9.2: *left:* Progress of approximate policy iteration for two realizations for 100 update steps *right:* Progress for 200 update steps

We carry out two experiments with different numbers of update steps for approximate policy evaluation. To compare the policies we use the utility of the obtained policies. Let π be a policy and V an approximation of V^π. We define the approximated utility of a policy π and the approximation V by

$$\frac{1}{|S|}\sum_{s\in S} V(s).$$

After each improvement step the approximated utility of the obtained policy is computed in each realization. Figure 9.2 shows the approximated utility for experiments with 100 and 200 update steps.

We observe that the utility can decrease due to the error of the approximation. The obtained policies perform well in both realizations. The robot

receives positive rewards for both families of rewards quite often. The keep distance behavior was more difficult to learn, since only certain sensor values are positively rewarded. The sensor values are most of the time either close to zero or close to one.

We look at the actions of the obtained policy in two states. In state $(9, 9, 9, 9)$, the robot chooses the action to move backward 60 steps and to turn 0 degrees. In state $(6, 6, 0, 0)$, it moves backward 30 steps and turns -20 degrees.

When applying the obtained policies with a moving target the robot does not always follow the target well. The obtained behavior may be improved with a finer discretization of states and actions and by applying more of the time consuming approximate policy evaluation steps. The main purpose of the experiments is to show that it is possible to adapt approximate policy iteration for several realizations to real world applications. First promising tests with a variant of the network described in Section 4.5 have been made.

To repeat this experiment connect your Khepera mobile robot to the serial port and start `RealRobo`. Open the `Properties` rollout and choose the appropriate `COM port` and `baud rate`. Initialize the connection with the robot by pressing `Initialize Khepera`. Now the sensor cones show the current sensor values graphically. They can be turned off and on by switching the checkerbox `Draw sensors`. With `Show sensors` the current sensor values are displayed in the window.

Now put the robot in front of a target. Open the `Target Following` rollout. Choose the desired number of approximate policy evaluation `Steps` and press the button `Initialize` to set all action-values to zero. Press `Random Policy Ev.` to run approximate policy evaluation for the random policy. Wait for the robot to carry out the number of steps. Then press the button `Policy Improvement` to apply policy improvement for both realizations. The rewards and information about the improved states are displayed in the console window. To evaluate the improved policy press `Policy Evaluation`.

To run a policy choose `Run/Stop policy` and observe the rewards displayed in the console window.

Chapter 10

Discussion and Future Work

One of the main results of this thesis is policy improvement and policy iteration for a family of realizations, see Section 5.6. In all our computational experiments policy iteration, Algorithm 8, terminated after a finite number of improvement steps. We could not yet prove this conjecture.

In the policy improvement step strictly improving vertices were chosen randomly. Their choice influences the resulting balanced policy and the number of iterations until termination; details are to be investigated. A characterization of all balanced policies and its connection to Pareto optimality are of further interest, also methods to compute strictly improving vertices.

Policy improvement is formulated for an infinite number of realizations, see Section 5.2. In the future we would like to discuss how to find strictly improving extreme points and to apply a variant of policy iteration for problems with infinitely many realizations, for example considering parametrized transition probabilities or rewards.

We consider discounted MDPs only. Several aspects of the theory and algorithms could possibly be adapted for other models and criteria of MDPs, like finite horizon MDPs, undiscounted and average expected rewards, see for example Puterman [Put94] and Kallenberg in [FS94, pp. 23].

For model-free methods our future research will include a further investigation of the underlying theory. Since for several realizations there is no unique optimal value or action-value function methods such as value iteration and Q-learning cannot be applied. Moreover, there is no natural method to derive a policy for several realizations from a single value or action-value function.

Approximate policy iteration for several realizations requires a good approximation of the action-values for each realization in the approximate policy evaluation step. In this context the number of update steps and the choice of the states and actions that are updated play an important role.

Especially in real world applications, updates can be very costly and time consuming. *Optimistic policy iteration* methods that work with incomplete evaluations of the policy could be adapted and applied, see Tsitsiklis [Tsi03].

For problems with several realizations in real world applications we present an adaptation of approximate policy iteration for a target following behavior. We use a discrete state and action representation. Experiments using function approximation are a future focus of our work. The method from Section 4.5 has been applied in a first test. It was changed to represent the action-values of a policy that has to be explicitly given and included in the implementation. First promising results have been obtained.

The proposed theory for several realizations can be used in a variety of research topics and applications. For discounted *stochastic games* a single strategy can be found that performs well against several players who follow fixed strategies, see Filar and Vrieze [FV97]. Several applications to partially observable MDPs seem to be possible. For example, policies can be improved for several probability distributions over all states of the underlying MDP simultaneously, compare with Section 3.3.2. Our theory can also be applied in the field of constrained MDPs by including linear constraints on policies in the improvement step.

We plan an implementation of the simplex algorithm to exactly compute balanced policies for a finite number of realizations for the `MDP` package. The simulator `SimRobo` was not only programmed to test our algorithms. It is also meant to serve as a didactic tool for lectures on reinforcement learning. We intend to implement additional classes for sensors and actions and hope that in the future new functions, like the simplex algorithm, will be added not only by the authors.

For `RealRobo` an object-oriented implementation of the robot control and its functions is in progress. Also a general redesign and the object-oriented implementation of the network are projected.

Refer to the website of our project, `http://mathematik.uibk.ac.at/users/rl`, for the latest versions of the `MDP` package, `SimRobo` and `RealRobo`.

Chapter 11

Appendix

11.1 Listings

The header files of all classes from `SimRobo`. The full source code can be found on the attached CD-ROM.

11.1.1 One Environment

Listing 1: class RLS

```
class RLS {
public:
// number of states (ns, ns2=ns*ns)
// number of actions (na) (with ns>=na)
// transitions probabilities, P(s'|a,s) = [a*ns2+s'*ns+s]
// rewards, R(s',a,s) = [a*ns2+s'*ns+s]
RLS(int,int,const map<int,double>&,
RLS(int,int,const map<int,double>&,
const map<int,double>&,double);
// as above
// expected rewards, R(a,s) = [a*ns+s]
RLS(int,int,const map<int,double>&,
const valarray<double>&,double);
RLS() {}

map<int,double> transprob() const;        // transition
double transprob(int,int,int) const;      // probabilities
valarray<double> rewas() const;           // expected reward
double get_gamma() const;                 // discount rate
// optimal policies:
void value_iteration();          // runs value iteration
double dutilopt();               // discounted utility
valarray<double> optval();       // computes value function
```

```
valarray<int> optpol();          // an optimal policy
multimap<int,int> optact();      // optimal actions
// random policy:
void ran_eval();                 // computes value function
double dutilran();               // discounted utility

// policy improvement
// returns greedy policy for action-values
valarray<int> pol_improv(const valarray<double>&) const;

// computes R(s), P(s'|s) of:
// deterministic policies, action a in state [s] = a
valarray<double> rew_dpol(const valarray<int>&) const;
valarray<double> prob_dpol(const valarray<int>&) const;

// extended policies, random action in state [s] = -1
valarray<double> rew_epol(const valarray<int>&) const;
valarray<double> prob_epol(const valarray<int>&) const;

// stochastic policies, pi(a,s) = [a*ns+s]
valarray<double> rew_ppol(const valarray<double>&) const;
valarray<double> prob_ppol(const valarray<double>&) const;

// action-values of policy with value function V,
// Q(a,s)=[a*ns+s]
valarray<double> qval(const valarray<double>&) const;
void save_rls(const char*) const;            // saves MDP
void save_res(const char* filename) const;   // saves results
void load_res(const char* filename);         // loads results

private:
// MDP:
int ns,ns2;                      // number of states,  ns2=ns*ns
int na;                          // number of actions
map<int,double> tprob;           // transition probabilities
valarray<double> ras;            // expected rewards
double gamma;                    // discount rate
// optimal policies:
bool optimal;                    // computed
valarray<double> oval;           // value function
valarray<int> opol;              // an optimal policy
multimap<int,int> oact;          // optimal actions
double duo;                      // discounted utility
// random policy:
bool random;                     // computed
double dur;                      // discounted utility
};
```

```
// policy evaluation: R(s), P(s'|s), precision
valarray<double> pol_eval(const valarray<double>&,
const valarray<double>&,double,double);
// undiscounted value function up to time T : R(s), P(s'|s), T
valarray<double> uval(const valarray<double>&,
const valarray<double>&,int);
```

Listing 2: class RF

```
class RF {
public:
// successor state s', action a, state s
virtual int operator()(int,int,int) = 0;
virtual RF* clone() const = 0;
virtual string get_name() const { return name; }
protected:
string name;
};

// obstacle avoidance:
class RF_OA : public RF {
public:
RF_OA() { name="Obstacle avoidance"; }
int operator() (int,int,int);
RF* clone() const;
};

// wall following:
class RF_WF : public RF {
public:
RF_WF() { name="Wall following"; }
int operator() (int,int,int);
RF* clone() const;
};
```

Listing 3: class robo_env

```
// from pos to coordinates x,y
// and vice versa
// environment
//      dimx
//           x
//      XXXXXXX (x,y) |-> x+y*dimx=pos
//      X
//      X   X    pos |-> (x=pos mod dimx, y=(pos-x)/dimx)
//     yX
//      X
```

```
// number of directions (directions 1,...,ND)
const int ND=4;
// directions       1
// of robot      3   4      1: A, 2: V, 3: <, 4: >
//                  2

const int NSE=4;    // number of sensors
const int FRONT=5; // range of sensors
const int LEFT=5;
const int RIGHT=5;
const int BACK=5;

//  >X          front sensor = 5
//  > X         front sensor = 4
//  >  X        front sensor = 2
//  >   X       front sensor = 1
//  >    X      front sensor = 0
//  >      X    front sensor = 0 ...

const int NA=3;              // number of actions

// actions, move:
// 0 forward
// 1 to the left
// 2 to the right

const int M=6;           // M=max(range of sensors) + 1
const int M2=6*6;        // M^2
const int M3=6*6*6;      // M^3

// sensors -> state
// sensors:front    se[0]    state:
//         left     se[1]    |-> se[0]*M3+se[1]*M2+se[2]*M+se[3]
//         right    se[2]
//         back     se[3]

class robo_env {
public:
// environment
// dimension x
// reinforcement function (derived from RF)
// discount rate
robo_env(const valarray<int>&,int,RF*,double);
// environment
// dimension x
// rls file
robo_env(const valarray<int>&,int,const char*);
robo_env() {}
```

```
valarray<int> get_env() const;              // environment
int get_dimx() const;                       // dimension x
int get_ns() const;                         // number of states
list<int> get_states() const;               // list of states
bool isstate(int) const;                    // is a state of env
map<int,double> transprob() const;          // transition
double transprob(int,int,int) const;        // probabilities
double get_gamma() const;                   // discount rate

// optimal policies:
void value_iteration();                  // runs value iteration
double dutilopt();                       // discounted utility
map<int,double> optval();                // computes value function
map<int,int> optpol();                   // an optimal policy
multimap<int,int> optact();              // optimal actions
// random optimal policy
map<pair<int,int>,double> ran_optpol();
// random policy:
void ran_eval();                         // computes value function
double dutilran();                       // discounted utility
double dlearnpot();                      // learning potential

// policy improvement
// returns improved deterministic policy
map<int,int> pol_improv(const map<int,int>&) const;

// policy evaluation:
// returns value function of (V(s)=[number of s]):
// deterministic policies, action a in state [s]=a
valarray<double> dpol_eval(const map<int,int>&) const;
// extended policies, random action in s: [s]=-1
valarray<double> epol_eval(const map<int,int>&) const;
// stochastic policies, pi(a,s)=[(a,s)]
valarray<double> ppol_eval(const map<pair<int,int>,
double>&) const;
// returns value function of (V(s)=[s]):
// extended policies, random action in s: [s]=-1
map<int,double> xepol_eval(const map<int,int>&) const;

// action-values for value function V(s)=[number of s]
map<pair<int,int>,double> qval(const valarray<double>&) const;

// deterministic or extended policy to extended policy
map<int,int> pol2epol(const map<int,int>&) const;

void save_rls(const char* filename) const;  // saves MDP
void save_res(const char* filename) const;  // saves results
void load_res(const char* filename);        // loads results
```

```
// actions, sensors, positions:
// perform action: pos, direction, action
// return pos, direction
pair<int,int> perform_a(int,int,int) const;
// get sensors: pos, direction
valarray<int> get_se(int,int) const;
// can robot get into this position by allowed actions?
// pos, direction
bool pos_dir(int,int) const;
// chooses random position: x,y,dir
void random_position(int&,int&,int&);

// text format:
void view_env() const;                  // environment
void view_results();                    // results
void view_rob(int,int,int) const;       // robot (x,y,dir)

protected:
valarray<int> env;          // environment
int dimx;                   // dimension x
int ns,ns2;                 // number of states, ns2=ns*ns
list<int> states;           // list of states
map<int,int> stons;         // state to number of state

RLS rls;                    // reinforcement learning system
};

int setos(const valarray<int>&);    // sensors to state
valarray<int> stose(int);           // state to sensors

// functions for policies:

// stochastic policy = probabilistic policy
// generates a random stochastic policy for list of states
map<pair<int,int>,double> generate_ppol(const list<int>&);
// extends stochastic policy for list of states
map<pair<int,int>,double> extend_ppol(map<pair<int,int>,double>,
const list<int>&);
// determinstic to stochastic policy for list of states
map<pair<int,int>,double> dp2pp(const map<int,int>&,
const list<int>&);
// loads deterministic and stochastic policies, filename
map<int,int> load_pol(const char*);
map<pair<int,int>,double> load_ppol(const char*);
// saves deterministic and stochastic policies, filename, policy
void save_pol(const char*,map<int,int>);
void save_ppol(const char*,map<pair<int,int>,double>);
```

Listing 4: class robo_env_pos

```
// environment:
//      dimx
//           x
//      XXXXXXX (x,y) |-> x+y*dimx=pos
//      X
//      X   X    pos |-> (x=pos mod dimx, y=(pos-x)/dimx)
//     yX
//      X

const NDIR=4;  // (directions 1,...,ND)

// position and direction  -> state
// position = i            -> i*NDIR+(j-1)
// direction = j

// and vice versa
// state   -> position and direction
// pos = state/NDIR
// direction = state

// directions      1
// of robot      3   4     1: A, 2: V, 3: <, 4: >
//                 2

class robo_env_pos : public robo_env {
public:
// environment
// dimension x
// reinforcement function (derived from RF)
// discount rate
robo_env_pos(const valarray<int>&,int,RF*,double);
// environment
// dimension x
// rls file
robo_env_pos(const valarray<int>&,int,const char*);
robo_env_pos() {}
};

// pos and direction to state
int posdirtos(int,int);
```

11.1.2 Model-free Methods

Listing 5: class online_learning

```
class online_learning {
public:
// number of actions, discount rate
```

```
online_learning(int,double);
online_learning() {}

double get_gamma() const;    // discount rate
map<pair<int,int>,double>    // action-values
get_qvalues() const;
void initialize();           // set qvalues to zero

// greedy policy from approximation for states list
// extended policy: random action for unknown states
map<int,int> get_greedypolicy(const list<int>&) const;
// approximation of the value function for state list
// 0.0 for unknown states
// extended policies
valarray<double> get_values(const list<int>&,
const map<int,int>&) const;
// stochastic policies
valarray<double> get_values(const list<int>&,
const map<pair<int,int>,double>&) const;

protected:
int na;                                  // number of actions
double q_gamma;                          // discount rate
map<pair<int,int>,double> qvalues;       // approximation
// of action-values
};
```

Listing 6: class SARSA

```
class SARSA : public online_learning {
public:
SARSA(int,double);
SARSA() {}
// update for approximate policy evaluation
// for deterministic policies
// action in successor state a', successor state s',
// action a, state s, reward r,
// step-size parameter
void update(int,int,int,int,int,double);
// update for approximate policy evaluation
// for stochastic policies
// stochastic policy in s', successor state s',
// action a, state s, reward r,
// step-size parameter
void update(const vector<double>&,int,int,int,int,double);
};
```

Listing 7: class QLS

```
class QLS : public online_learning {
```

```
public:
// number of actions, discount rate
QLS(int,double);
QLS() {}
// update rule for Q-learning:
// successor state s', action a, state s, reward r,
// step-size parameter
void update(int,int,int,int,double);
};
```

Listing 8: class robo_env_online

```
class robo_env_online : public robo_env {
public:
// environment
// dimension x
// reinforcement function (derived from RF)
// discount rate
robo_env_online(const valarray<int>&,int dx,RF*,double);
// environment
// dimension x
// rls file
// reinforcement function (derived from RF)
robo_env_online(const valarray<int>&,int dx,const char*,RF*);
robo_env_online() {}
// copy constructor
robo_env_online(const robo_env_online&);
// assignment
robo_env_online& operator=(const robo_env_online&);
~robo_env_online();      // destructor

// get name of reinforcement function
string get_rf_name() const { return rf->get_name(); }

// Q-Learning:
// steps, starting step-size parameter, discount rate, QLS
QLS qlearning(int,double,double,QLS);
// steps, starting step-size parameter, discount rate
QLS qlearning(int,double,double);

// approximate policy evaluation for stochastic policies:
// policy,
// steps, starting step-size parameter, discount rate, SARSA
SARSA sarsa_poleval(const map<pair<int,int>,double>&,
int,double,double,SARSA);
// policy,
// steps, starting step-size parameter, discount rate
SARSA sarsa_poleval(const map<pair<int,int>,double>&,
int,double,double);
```

```
// text format:
void view_policy(const map<int,int>&) const;// optimal policy
void view_optact();                          // optimal actions

private:
RF* rf;
};
```

11.1.3 Several Environments

Listing 9: class impolytop and class inters_impolytop

```
// list of actions and probabilities
typedef list<pair<int,double> > vertex;

typedef vertex::const_iterator CIV;
typedef list<vertex>::const_iterator CILV;

class impolytop {
public:
// action-values and value function in state s
// Q(a_1,s),...,Q(a_d+1,s),V(s)
impolytop(const valarray<double>&,double);
impolytop() {}

valarray<double> get_q() const; // action-values
double get_v() const;           // value function V(s)

// improving, strictly improving and
// equivalent policies for a MDP
vector<vertex> get_improvpol() const;
vector<vertex> get_simprovpol() const;
vector<vertex> get_equpol() const;

private:
valarray<double> q;     // action-values
double v;               // value function V(s)
vector<vertex> impol;   // improving
vector<vertex> simpol;  // strictly improving
vector<vertex> equpol;  // equivalent policies
};

class inters_impolytop {
public:
// improving policies
inters_impolytop(const vector<impolytop>&);
// strictly improving policies
// for a finite family of MDPs
vector<vertex> get_simprovpol() const;
```

```
private:
vector<impolytop> polytops; // improving policies
vector<vertex> simpol;      // strictly
};

double round(double wert);
```

Listing 10: class sev_robo_env

```
class sev_robo_env {
public:
// environments
sev_robo_env(const vector<robo_env*>&);
sev_robo_env() { }
// copy constructor
sev_robo_env(const sev_robo_env&);
// assignment
sev_robo_env& operator=(const sev_robo_env&);
~sev_robo_env();    // destructor

int get_ns() const;              // number of states
list<int> get_states() const;    // list of states
int get_numenv() const;          // number of environments
robo_env get_env(int) const;     // robo environment
// number (0...ne-1)
// discounted utility
vector<double> dutilopt();       // optimal policies
vector<double> dutilran();       // random policy

// policy improvement for several MDPs
// for deterministic policies
// (chooses only strictly improving actions)
// returns improved policy
map<int,int> pol_improv(const map<int,int>& pol) const;
// policy improvement for several MDPs
// for stochastic policies
// (chooses strictly improving vertices)
// returns improved policy
map<pair<int,int>,double>
pol_improv(const map<pair<int,int>,double>& pol);

// stochastic policies, pi(a,s)=[(a,s)]
// returns utility for each environment
vector<double> ppol_eval(const map<pair<int,int>,
double>&) const;

private:
vector<robo_env*> env;  // environments
int ne;                 // number of environments
int ns,ns2;             // number of states, ns2=ns*ns
```

```
list<int> states;        // list of states
map<int,int> stons;      // state to number of state
};
```

Listing 11: class sev_robo_env_online

```
class sev_robo_env_online {
public:
// online environments
sev_robo_env_online(const vector<robo_env_online*>&);
sev_robo_env_online() {}
// copy constructor
sev_robo_env_online(const sev_robo_env_online&);
// assignment
sev_robo_env_online& operator=(const sev_robo_env_online&);
~sev_robo_env_online(); // destructor

int get_ns() const;              // number of states
list<int> get_states() const;    // list of states
int get_numenv() const;          // number of environments
robo_env_online get_env(int) const;    // robo environment
// number (0...ne-1)
// discounted utility
vector<double> dutilopt();       // optimal policies
vector<double> dutilran();       // random policy

// approximate policy improvement for several MDPs
// for stochastic policies
// returns "improved" policy
// policy,
// steps, starting step-size parameter, discount rate
map<pair<int,int>,double>
pol_improv(const map<pair<int,int>,double>& pol,
int steps, double q_lambda, double q_gamma);

// stochastic policies, pi(a,s)=[(a,s)]
// returns utility for each environment
vector<double> ppol_eval(const map<pair<int,int>,
double>&) const;

private:
vector<robo_env_online*> env;    // environments
int ne;                          // number of environments
int ns,ns2;                      // number of states, ns2=ns*ns
list<int> states;                // list of states
map<int,int> stons;              // state to number of state
};
```

11.1.4 RealRobo

Listing 12: RealRobo

```
// COM port variables
static DCB dcb;
static HANDLE hCom, hStdout;
static DWORD dwError;
static BOOL fSuccess;
static COMMTIMEOUTS comt;

static char comm[10]="COM2";
static unsigned int bauds=9600;

int serialport=2;       // serial port
int serialspeed=9600;  // baud rate 9600,19200 or 38400

static char tbuff[256];     // string buffer to send data
static char rbuff[512];     // string buffer to receive data

// Khepera commandos
// set Speed of motors left=b[0], right=b[1] (K-Commando D)
void Set_Speed(int b[2]);
// read position counter of two motors (K-Commando H)
void Read_Position(int motors[2]);
// read 8 proximity sensors to s (K-Commando N)
void Read_Sensors(float s[CTD_SENSORES]);
// set position counters for motors (K-Commando G)
void Set_Position_Counter(int c_p[2]);
// set position counters to reach for motors (K-Commando C)
void Set_Position_To_Reach(int c_p[2]);
// read status of motion controller to c_estado (K-Comando K)
void Read_Motion_Controller(int c_estado[6]);

void Turn(int ang); // Turn an integer angle

// convert angle from [0,1] to [-90 and 90]
int  Value_To_Angle(float value);
// move steps forward/backward, 1 step=0.08mm
void Move_Forward(int steps);

// reflex 1 move straight when no obstacles
void Reflejo_1_oa();
// for reflex 2, returns 1 if obstacle, 0 else
int  Colision();

// network functions
void leer_red(char *filename);                  // load weights
void save_red(char *nombre_archivo,int epoca); // save weights
```

```
// set winner unit closest to s,Q=1
void privateNearestSQ(float s[CTD_SENSORES],float Q, int umbral);
// compute distance s,Q=1
float privateDistSQ(float x[ctd_entradas+CTD_ACCIONES],int nodo);
// set winner unit closest to s,a
void privateNearestSA(float s[8], float acc[CTD_ACCIONES]);
// compute distance s,a
float privateDistSA(float x[ctd_entradas+CTD_ACCIONES],int nodo);
// update weights
void QUpdate(float sensor_a[CTD_SENSORES],
float a[CTD_ACCIONES], float sensor[CTD_SENSORES], int r);
// add new unit
void privateAddNeuron(float sensor[CTD_SENSORES],
float acc[CTD_ACCIONES], double q);
```

11.2 MDP Package

The following is list of all functions in the `MDP` Maple package with short descriptions and examples.

11.2.1 Rewards and Transition Matrix

ExpectedReward - *compute the expected rewards*

Calling Sequence

ExpectedReward(**P**,**R**)

Parameters

`P` - Array; transition probabilities

`R` - Array; rewards

Description

The ExpectedReward(**P**,**R**) function computes the expected rewards for transition probabilities **P** and rewards **R**.

Examples

```
> with(MDP):
> P:=Array(1..2,1..2,1..2):
> P[1..2,1..2,1]:=<<1/4,3/4>|<2/3,1/3>>;
```

$$P_{1..2,\,1..2,\,1} := \begin{bmatrix} \frac{1}{4} & \frac{2}{3} \\ \frac{3}{4} & \frac{1}{3} \end{bmatrix}$$

```
> P[1..2,1..2,2]:=<<1/6,5/6>|<3/5,2/5>>:
> R:=Array(1..2,1..2,1..2):
> R[1..2,1..2,1]:=<<1,-1>|<1,0>>;
```

$$R_{1..2,\,1..2,\,1} := \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$$

```
> R[1..2,1..2,2]:=<<0,1>|<1,-1>>:
> ER:=ExpectedReward(P,R);
```

$$ER := \begin{bmatrix} \frac{-1}{2} & \frac{5}{6} \\ \frac{2}{3} & \frac{1}{5} \end{bmatrix}$$

ExpectedRewardPolicy - ***compute the expected rewards for a policy***

Calling Sequence

ExpectedRewardPolicy(**ER**,**Pol**)

Parameters

`ER` - Matrix; expected rewards
`Pol` - Matrix; policy

Description

The ExpectedRewardPolicy(**ER**,**Pol**) function computes the expected rewards for policy **Pol** and expected rewards **ER**.

Examples

```
> with(MDP):
> P:=Array(1..2,1..2,1..2):
> P[1..2,1..2,1]:=<<1/4,3/4>|<2/3,1/3>>:
> P[1..2,1..2,2]:=<<1/6,5/6>|<3/5,2/5>>:
> R:=Array(1..2,1..2,1..2):
> R[1..2,1..2,1]:=<<1,-1>|<1,0>>:
> R[1..2,1..2,2]:=<<0,1>|<1,-1>>:
> ER:=ExpectedReward(P,R);
```

$$ER := \begin{bmatrix} \frac{-1}{2} & \frac{5}{6} \\ \frac{2}{3} & \frac{1}{5} \end{bmatrix}$$

```
> Pol:=<<1/2,1/2>|<1/2,1/2>>;
```

$$Pol := \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

```
> ERp:=ExpectedRewardPolicy(ER,Pol);
```

$$ERp := \left[\frac{1}{12}, \frac{31}{60}\right]$$

TransitionMatrix - *compute the transition matrix for a policy*

Calling Sequence

TransitionMatrix(**P**,**Pol**)

Parameters

`P` - Array; transition probabilities
`Pol` - Matrix; policy

Description

The TransitionMatrix(**P**,**Pol**) function computes the transition matrix for policy **Pol** and transition probabilities **P**.

Examples

```
> with(MDP):
> P:=Array(1..2,1..2,1..2):
> P[1..2,1..2,1]:=<<1/4,3/4>|<2/3,1/3>>:
> P[1..2,1..2,2]:=<<1/6,5/6>|<3/5,2/5>>:
> Pol := <<1/2,1/2>|<1/2,1/2>>;
```

$$Pol := \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

```
> Pp:=TransitionMatrix(P,Pol);
```

$$Pp := \begin{bmatrix} \frac{11}{24} & \frac{23}{60} \\ \frac{13}{24} & \frac{37}{60} \end{bmatrix}$$

```
> IsStochastic(Pp);
```

$$true$$

11.2.2 Value Function and Action-Values

ValueFunction - *compute the discounted value function*

Calling Sequence

ValueFunction(**ERp**,**Pp**,**ga**)

Parameters

`ERp` - Vector[row]; expected rewards for a policy
`Pol` - Matrix; transition matrix for a policy
`ga` - algebraic; discount rate

Description

The ValueFunction(**ERp**,**Pp**,**ga**) command computes the discounted value function for a policy given the expected rewards **ERp** the transition matrix **Pp** and a discount rate **ga**.

Examples

```
> with(MDP):
> P:=Array(1..2,1..2,1..2):
> P[1..2,1..2,1]:=<<1/4,3/4>|<2/3,1/3>>:
> P[1..2,1..2,2]:=<<1/6,5/6>|<3/5,2/5>>:
> R:=Array(1..2,1..2,1..2):
> R[1..2,1..2,1]:=<<1,-1>|<1,0>>:
> R[1..2,1..2,2]:=<<0,1>|<1,-1>>:
> ER:=ExpectedReward(P,R):
> Pol:=<<1/2,1/2>|<1/2,1/2>>:
> ERp:=ExpectedRewardPolicy(ER,Pol);
```

$$ERp := \left[\frac{1}{12}, \frac{31}{60}\right]$$

```
> Pp:=TransitionMatrix(P,Pol);
```

$$Pp := \begin{bmatrix} \frac{11}{24} & \frac{23}{60} \\ \frac{13}{24} & \frac{37}{60} \end{bmatrix}$$

```
> ga:=1/2;
```

$$ga := \frac{1}{2}$$

```
> Vp:=ValueFunction(ERp,Pp,ga);
```

$$Vp := \left[\frac{569}{1386}, \frac{1193}{1386}\right]$$

ActionValues - *compute action-values*

Calling Sequence

ActionValues(**P**,**ER**,**Vp**,**ga**)

Parameters

`P` - Array; transition probabilities
`ER` - Matrix; expected rewards
`Vp` - Vector[row]; value function for a policy
`ga` - expression; discount rate

Description

The ActionValues(**P**,**ER**,**Vp**,**Vp**) command computes the action-values for a policy given the transition probabilities **P**, the expected rewards **ER**, the value function **Vp** and a discount rate **ga**.

Examples

```
> with(MDP):
> P:=Array(1..2,1..2,1..2):
> P[1..2,1..2,1]:=<<1/4,3/4>|<2/3,1/3>>:
> P[1..2,1..2,2]:=<<1/6,5/6>|<3/5,2/5>>:
> R:=Array(1..2,1..2,1..2):
> R[1..2,1..2,1]:=<<1,-1>|<1,0>>:
> R[1..2,1..2,2]:=<<0,1>|<1,-1>>:
> ER:=ExpectedReward(P,R);
```

$$ER := \begin{bmatrix} -\frac{1}{2} & \frac{5}{6} \\ \frac{2}{3} & \frac{1}{5} \end{bmatrix}$$

```
> Pol:=<<1/2,1/2>|<1/2,1/2>>:
> ERp:=ExpectedRewardPolicy(ER,Pol):
> Pp:=TransitionMatrix(P,Pol);
```

$$Pp := \begin{bmatrix} \frac{11}{24} & \frac{23}{60} \\ \frac{13}{24} & \frac{37}{60} \end{bmatrix}$$

```
> ga:=1/2;
```

$$ga := \frac{1}{2}$$

```
> Vp:=ValueFunction(ERp,Pp,ga);
```

$$Vp := \left[\frac{569}{1386}, \frac{1193}{1386}\right]$$

```
> Qp:=ActionValues(P,ER,Vp,ga);
```

$$Qp := \begin{bmatrix} \frac{-349}{2772} & \frac{103}{84} \\ \frac{125}{132} & \frac{1373}{2772} \end{bmatrix}$$

11.2.3 Improving Actions and Vertices

StrictlyImprovingActions - *compute strictly improving actions*

Calling Sequence

StrictlyImprovingActions(**Qps**,**Vps**)

Parameters

`Qps` - Vector; action-values in a state

`Vps` - algebraic; value function in a state

Description

The StrictlyImprovingActions(**Qps**,**Vps**) function returns a list of strictly improving actions given the action-values **Qps** and the value function **Vps** in a state.

Examples

```
> with(MDP):
> Qps:=<0,2,4,7>;
```

$$Qps := \begin{bmatrix} 0 \\ 2 \\ 4 \\ 7 \end{bmatrix}$$

```
> Vps:=2;
```

$$Vps := 2$$

```
> StrictlyImprovingActions(Qps,Vps);
```

$$[3, 4]$$

EquivalentVertices - *compute equivalent vertices*

Calling Sequence

EquivalentVertices(**Qps**,**Vps**)

Parameters

`Qps` - Vector; action-values in a state

`Vps` - algebraic; value function in a state

Description

The EquivalentVertices(**Qps**,**Vps**) function returns a list of equivalent vertices given the action-values **Qps** and the value function **Vps** in a state.

Examples

```
> with(MDP):
> Qps:=<0,2,4,7>;
```

$$Qps := \begin{bmatrix} 0 \\ 2 \\ 4 \\ 7 \end{bmatrix}$$

```
> Vps:=2;
```

$$Vps := 2$$

```
> EquivalentVertices(Qps,Vps);
```

$$\left[\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \frac{1}{2} \\ 0 \\ \frac{1}{2} \\ 0 \end{bmatrix}, \begin{bmatrix} \frac{5}{7} \\ 0 \\ 0 \\ \frac{2}{7} \end{bmatrix}\right]$$

ImprovingVertices - *compute improving vertices*

Calling Sequence

ImprovingVertices(**Qps**,**Vps**)

Parameters

`Qps` - Vector; action-values in a state

`Vps` - algebraic; value function in a state

Description

The ImprovingVertices(**Qps**,**Vps**) function returns a list of improving vertices given the action-values **Qps** and the value function **Vps** in a state. The first element is a list of improving vertices corresponding to strictly improving actions and the second element the list of equivalent vertices.

Examples

```
> with(MDP):
> Qps:=<0,2,4,7>;
```

$$Qps := \begin{bmatrix} 0 \\ 2 \\ 4 \\ 7 \end{bmatrix}$$

```
> Vps:=2;
```

$$Vps := 2$$

```
> IV:=ImprovingVertices(Qps,Vps);
```

$$IV := \left[\left[\left[\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right], \left[\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \frac{1}{2} \\ 0 \\ \frac{1}{2} \\ 0 \end{bmatrix}, \begin{bmatrix} \frac{5}{7} \\ 0 \\ 0 \\ \frac{2}{7} \end{bmatrix} \right] \right] \right]$$

```
> IV[1];
```

$$\left[\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right]$$

```
> IV[2];
```

$$\left[\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \frac{1}{2} \\ 0 \\ \frac{1}{2} \\ 0 \end{bmatrix}, \begin{bmatrix} \frac{5}{7} \\ 0 \\ 0 \\ \frac{2}{7} \end{bmatrix} \right]$$

11.2.4 Stochastic Matrices

IsStochastic - ***test if a Matrix is stochastic***

Calling Sequence

IsStochastic(**A**)

Parameters
`A` - Matrix

Description
The IsStochastic(**A**) function determines if **A** is a stochastic Matrix (non-negative and column sums equal to 1).

Examples

```
> with(MDP):
> A:=<<1/2,1/2>|<1/3,2/3>>;
```

$$A := \begin{bmatrix} \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{2}{3} \end{bmatrix}$$

```
> IsStochastic(A);
```

true

IsTransitionProbability - ***test if an Array defines valid transition probabilities***

Calling Sequence
IsTransitionProbability(**P**)

Parameters
`P` - Array; transition probabilities

Description
The IsTransitionProbability(**P**) function determines if **P** defines valid transition probabilities.

Examples

```
> with(MDP):
> P:=Array(1..2,1..2,1..2):
> P[1..2,1..2,1]:=<<1/4,3/4>|<2/3,1/3>>;
```

$$P_{1..2,\,1..2,\,1} := \begin{bmatrix} \frac{1}{4} & \frac{2}{3} \\ \frac{3}{4} & \frac{1}{3} \end{bmatrix}$$

```
> IsStochastic(%);
```

true

```
> P[1..2,1..2,2]:=<<1/6,5/6>|<3/5,2/5>>:
> IsTransitionProbability(P);
```

true

11.2.5 Random Rewards and Transition Probabilities

RandomReward - *construct random rewards*

Calling Sequence

RandomReward(**s**,**a**,**m**,**n**)

Parameters

`s` - positive integer; number of states
`a` - positive integer; number of actions
`m` - integer; lower bound
`n` - integer; upper bound

Description

The RandomReward(**s**,**a**,**m**,**n**) command returns an Array representing rewards for an environment with **s** states and **a** actions in each state. The entries are integers between **m** and **n**.

Examples

```
> with(MDP):
> R:=RandomReward(3,3,-1,1);
```

RandomStochasticVector - *construct a random stochastic Vector*

Calling Sequence

RandomStochasticVector(**d**,**n**)

Parameters

`d` - non-negative integer; dimension of the resulting Vector
`n` - positive integer; maximal denominator

Description

The RandomStochasticVector(**d**,**n**) command returns a random stochastic column Vector (non-negative and sum of entries equal to 1) of dimension **d** with rational entries and denominator less or equal **n**.

Examples

```
> with(MDP):
> RandomStochasticVector(3,5);
```

$$\begin{bmatrix} 0 \\ \frac{1}{5} \\ \frac{4}{5} \end{bmatrix}$$

RandomStochasticMatrix - *construct a random stochastic Matrix*

Calling Sequence

RandomStochasticMatrix(**r**,**c**,**n**)

Parameters

`r` - non-negative integer; row dimension of the resulting Matrix
`c` - non-negative integer; column dimension of the resulting Matrix
`n` - positive integer; maximal denominator

Description

The RandomStochasticMatrix(**r**,**c**,**n**) command returns a random stochastic (non-negative and column sums equal to 1) r x c Matrix with rational entries and denominator less or equal **n**.

Examples

```
> with(MDP):
> A:=RandomStochasticMatrix(3,3,10);
```

$$A := \begin{bmatrix} \frac{1}{2} & \frac{3}{5} & \frac{2}{5} \\ 0 & \frac{1}{10} & \frac{2}{5} \\ \frac{1}{2} & \frac{3}{10} & \frac{1}{5} \end{bmatrix}$$

```
> IsStochastic(A);
```

true

RandomTransitionProbability - *construct random transition probabilities*

Calling Sequence

RandomTransitionProbability(**s**,**a**,**n**)

Parameters

`s` - positive integer; number of states
`a` - positive integer; number of actions
`n` - positive integer; maximal denominator

Description

The RandomTransitionProbability(**s**,**a**,**n**) command returns an Array representing random transition probabilities for an environment with **s** states and **a** actions in each state. The entries are non-negative rational numbers with a denominator less or equal **n**.

Examples

```
> with(MDP):
> P:=RandomTransitionProbability(3,3,5):
> IsTransitionProbability(P);
```

true

11.2.6 Policy Improvement and Iteration

PolicyImprovement - *improve a policy*

Calling Sequence

PolicyImprovement(**Qp**,**Vp**,**Pol**)
PolicyImprovement(**Qp**,**Vp**,**Pol**,**'improved'**)

Parameters

`Qp` - Matrix; action-values for a policy
`Vp` - Vector[row]; value function for a policy
`Pol` - Matrix; policy
`improved` - (optional) name; is improved

Description

The PolicyImprovement(**Qp**,**Vp**,**Pol**) function returns an improved policy given action-values **Qp** and the value function **Vp** for policy **Pol**. The optional parameter **'improved'** is set to 1 if the policy is improved and 0 otherwise (that is, the policy is optimal).

Examples

```
> with(MDP):
> s:=3:  a:=3:
> P:=RandomTransitionProbability(s,a,5):
```

```
> R:=RandomReward(s,a,-1,1):
> ER:=ExpectedReward(P,R):
> Pol:=RandomStochasticMatrix(a,s,5);
```

$$Pol := \begin{bmatrix} \frac{3}{5} & \frac{3}{5} & \frac{3}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{2}{5} \\ \frac{1}{5} & \frac{1}{5} & 0 \end{bmatrix}$$

```
> ERp:=ExpectedRewardPolicy(ER,Pol):
> Pp:=TransitionMatrix(P,Pol):
> ga:=1/2:
> Vp:=ValueFunction(ERp,Pp,ga);
```

$$Vp := \left[\frac{-1447}{5665}, \frac{-5002}{5665}, \frac{1223}{5665}\right]$$

```
> Qp:=ActionValues(P,ER,Vp,ga):
> imPol:=PolicyImprovement(Qp,Vp,Pol,'improved');
```

$$imPol := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

```
> improved;
```

$$1$$

```
> Pimp:=TransitionMatrix(P,imPol):
> ERimp:=ExpectedRewardPolicy(ER,imPol):
> Vimp:=ValueFunction(ERimp,Pimp,ga);
```

$$Vimp := \left[\frac{257}{335}, \frac{-98}{335}, \frac{357}{335}\right]$$

```
> Vimp-Vp;
```

$$\left[\frac{77626}{75911}, \frac{44820}{75911}, \frac{64508}{75911}\right]$$

PolicyIteration - *policy iteration*

Calling Sequence

PolicyIteration(**P**,**ER**,**Pol**,**ga**)
PolicyIteration(**P**,**ER**,**Pol**,**ga**,**'steps'**)

Parameters

`P` - Array; transition probabilities
`ER` - Matrix; expected rewards
`Pol` - Matrix; policy
`ga` - algebraic; discount rate
`steps` - (optional) name; number of improvement steps

Description

The PolicyIteration(**P**,**ER**,**Pol**,**ga**,**'steps'**) command performs policy iteration and returns an optimal policy given transition probabilities **P**, expected rewards **ER**, a starting policy **Pol** and a discount rate **ga**. The optional parameter **'steps'** is set to the number of improvement steps used to obtain the returned optimal policy.

Examples

```
> with(MDP):
> s:=3:  a:=3:
> P:=RandomTransitionProbability(s,a,5):
> R:=RandomReward(s,a,-1,1):
> ER:=ExpectedReward(P,R):
> Pol:=RandomStochasticMatrix(a,s,5):
> ga:=1/2:
> optPol:=PolicyIteration(P,ER,Pol,ga,'steps');
```

$$optPol := \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

```
> steps;
```

$$2$$

```
> Pp:=TransitionMatrix(P,optPol):
> ERp:=ExpectedRewardPolicy(ER,optPol):
> Vp:=ValueFunction(ERp,Pp,ga):
> Qp:=ActionValues(P,ER,Vp,ga):
> IsOptimal(Qp,Vp);
```

$$true$$

IsOptimal - *test if a policy is optimal*

Calling Sequence

IsOptimal(**Qp**,**Vp**)

Parameters

`Qp` - Matrix; action-values for a policy

`Vp` - Vector[row]; value function for a policy

Description

The IsOptimal(**Qp**,**Vp**) function tests if a policy with action-values **Qp** and value function **Vp** is optimal.

Examples

```
> with(MDP):
> s:=3:  a:=3:
> P:=RandomTransitionProbability(s,a,5):
> R:=RandomReward(s,a,-1,1):
> ER:=ExpectedReward(P,R):
> Pol:=RandomStochasticMatrix(a,s,5):
> ga:=1/2:
> optPol:=PolicyIteration(P,ER,Pol,ga):
> ERp:=ExpectedRewardPolicy(ER,optPol):
> Pp:=TransitionMatrix(P,optPol):
> Vp:=ValueFunction(ERp,Pp,ga);
```

$$Vp := \left[\frac{20}{13}, \frac{10}{13}, \frac{92}{91}\right]$$

```
> Qp:=ActionValues(P,ER,Vp,ga);
```

$$Qp := \begin{bmatrix} \frac{20}{13} & \frac{16}{35} & \frac{32}{65} \\ \frac{358}{455} & \frac{10}{13} & \frac{92}{91} \\ \frac{334}{455} & \frac{-4}{13} & \frac{-3}{13} \end{bmatrix}$$

```
> IsOptimal(Qp,Vp);
```

$$true$$

11.2.7 Plots

improvingpolicyplot3d - *plot improving policies*

Calling Sequence
improvingpolicyplot3d(**IVs**,**Pols**)

Parameters
`IVs` - list; improving vertices in a state
`Pols` - Vector[column]; policy in a state

Description
The improvingpolicyplot3d(**IVs**,**Pols**) command plots the improving policies given the improving vertices **IVs** and the policy **Pols** in a state.

Examples

```
> with(MDP):
> s:=3:  a:=3:
> P:=RandomTransitionProbability(s,a,20):
> R:=RandomReward(s,a,-1,1):
> ER:=ExpectedReward(P,R):
> Pol:=RandomStochasticMatrix(a,s,10):
> ERp:=ExpectedRewardPolicy(ER,Pol):
> Pp:=TransitionMatrix(P,Pol):
> ga:=1/2:
> Vp:=ValueFunction(ERp,Pp,ga):
> Qp:=ActionValues(P,ER,Vp,ga):
> for i from 1 to s do
>    improvingpolicyplot3d(ImprovingVertices(\
>    LinearAlgebra:-Column(Qp,i),Vp[i]),\
>    LinearAlgebra:-Column(Pol,i));
> end do;
```

11.2.8 Two Realizations

StrictlyImprovingVertices2 - *compute strictly improving vertices*

Calling Sequence
StrictlyImprovingVertices2(**Qps**,**Vps**)

Parameters
`Qps` - list(Vector); action-values in a state

`Vps` - list(algebraic); value functions in a state

Description

The StrictlyImprovingVertices2(**Qps**,**Vps**) function returns a list of strictly improving vertices given the action-values **Qps** and the value functions **Vps** in a state for two realizations.

Examples

```
> with(MDP):
> Qps:=[<1,2,3,8>,<3,4,7,3>];
```

$$Qps := \left[\begin{bmatrix}1\\2\\3\\8\end{bmatrix}, \begin{bmatrix}3\\4\\7\\3\end{bmatrix}\right]$$

```
> Vps:=[2,4];
```

$$Vps := [2,\ 4]$$

```
> StrictlyImprovingVertices2(Qps,Vps);
```

$$\left[\begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}\frac{1}{2}\\0\\\frac{1}{2}\\0\end{bmatrix}, \begin{bmatrix}0\\0\\\frac{1}{4}\\\frac{3}{4}\end{bmatrix}\right]$$

PolicyImprovement2 - *improve a policy*

Calling Sequence

PolicyImprovement2(**Qp**,**Vp**,**Pol**)
PolicyImprovement2(**Qp**,**Vp**,**Pol**,**'improved'**)

Parameters

`Qp` - list(Matrix); action-values for a policy
`Vp` - list(Vector[row]); value functions for a policy
`Pol` - Matrix; policy
`improved` - (optional) name; is improved

Description

The PolicyImprovement2(**Qp**,**Vp**,**Pol**) function returns an improved policy given action-values **Qp** and value functions **Vp** for policy **Pol** for two

realizations. The optional parameter **'improved'** is set to 1 if the policy is improved and 0 otherwise (that is, the policy is balanced).

Examples

```
> with(MDP):
> s:=3:a:=3:
> P:=[RandomTransitionProbability(s,a,5),\
>         RandomTransitionProbability(s,a,5)]:
> R:=[RandomReward(s,a,-1,1),RandomReward(s,a,-1,1)]:
> ER:=[ExpectedReward(P[1],R[1]),\
>         ExpectedReward(P[2],R[2])]:
> Pol:=RandomStochasticMatrix(a,s,5);
```

$$Pol := \begin{bmatrix} 0 & \frac{2}{5} & \frac{2}{5} \\ \frac{4}{5} & \frac{1}{5} & \frac{2}{5} \\ \frac{1}{5} & \frac{2}{5} & \frac{1}{5} \end{bmatrix}$$

```
> ERp:=[ExpectedRewardPolicy(ER[1],Pol),\
>         ExpectedRewardPolicy(ER[2],Pol)]:
> Pp:=[TransitionMatrix(P[1],Pol),\
>         TransitionMatrix(P[2],Pol)]:
> ga:=[1/2,1/2]:
> Vp:=[ValueFunction(ERp[1],Pp[1],ga[1]),\
>         ValueFunction(ERp[2],Pp[2],ga[2])];
```

$$Vp := [\begin{bmatrix} \frac{-461}{4275}, & \frac{-1687}{1425}, & \frac{-163}{450} \end{bmatrix}, \begin{bmatrix} \frac{-12097}{18725}, & \frac{6128}{18725}, & \frac{-6322}{18725} \end{bmatrix}]$$

```
> Qp:=[ActionValues(P[1],ER[1],Vp[1],ga[1]),\
>         ActionValues(P[2],ER[2],Vp[2],ga[2])]:
> imPol:=PolicyImprovement2(Qp,Vp,Pol,'improved');
```

$$imPol := \begin{bmatrix} \frac{448}{905} & \frac{14753}{18825} & 0 \\ 0 & 0 & \frac{13331}{20170} \\ \frac{457}{905} & \frac{4072}{18825} & \frac{6839}{20170} \end{bmatrix}$$

```
> improved;
```

$$1$$

```
> ERimp:=[ExpectedRewardPolicy(ER[1],imPol),\
>         ExpectedRewardPolicy(ER[2],imPol)]:
```

```
> Pimp:=[TransitionMatrix(P[1],imPol),\
>        TransitionMatrix(P[2],imPol)]:
> Vimp:=[ValueFunction(ERimp[1],Pimp[1],ga[1]),\
>        ValueFunction(ERimp[2],Pimp[2],ga[2])]:
> Vimp[1]-Vp[1];
```

$$\left[\frac{48749710091603}{295058147899275}, \frac{7081447992017}{32784238655475}, \frac{21121785852899}{31058752410450}\right]$$

```
> Vimp[2]-Vp[2];
```

$$\left[\frac{69401197803974914}{78467072456007025}, \frac{17673362669093164}{78467072456007025}, \frac{15723509098557564}{78467072456007025}\right]$$

PolicyIteration2 - *policy iteration*

Calling Sequence

PolicyIteration2(**P**,**ER**,**Pol**,**ga**)
PolicyIteration2(**P**,**ER**,**Pol**,**ga**,**'steps'**)

Parameters

`P` - list(Array); transition probabilities
`ER` - list(Matrix); expected rewards
`Pol` - Matrix; policy
`ga` - list(algebraic); discount rates
`steps` - (optional) name; number of improvement steps

Description

The PolicyIteration2(**P**,**ER**,**Pol**,**ga**,**'steps'**) command performs policy iteration for two realizations. It returns a balanced policy given transition probabilities **P**, expected rewards **ER** and discount rates **ga** for two realizations and a starting policy **Pol**. The optional parameter **'steps'** is set to the number of improvement steps used to obtain the returned balanced policy.

Examples

```
> with(MDP):
> s:=3:a:=3:
> P:=[RandomTransitionProbability(s,a,10),\
>        RandomTransitionProbability(s,a,10)]:
> R:=[RandomReward(s,a,-2,2),RandomReward(s,a,-2,2)]:
> ER:=[ExpectedReward(P[1],R[1]),\
>        ExpectedReward(P[2],R[2])]:
> Pol:=RandomStochasticMatrix(a,s,10);
```

$$Pol := \begin{bmatrix} \frac{9}{10} & \frac{1}{10} & \frac{1}{10} \\ \frac{1}{10} & \frac{2}{5} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{2}{5} \end{bmatrix}$$

```
> ga:=[1/2,1/2]:
> balPol:=PolicyIteration2(P,ER,Pol,ga,'steps');
```

$$balPol := \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{3463063}{7173190} & \frac{832053429779429}{3400439142403598} \\ 0 & \frac{3710127}{7173190} & \frac{2568385712624169}{3400439142403598} \end{bmatrix}$$

```
> steps;
```

$$2$$

```
> ERp:=[ExpectedRewardPolicy(ER[1],balPol),\
>         ExpectedRewardPolicy(ER[2],balPol)]:
> Pp:=[TransitionMatrix(P[1],balPol),\
>         TransitionMatrix(P[2],balPol)]:
> Vp:=[ValueFunction(ERp[1],Pp[1],ga[1]),\
>         ValueFunction(ERp[2],Pp[2],ga[2])]:
> Qp:=[ActionValues(P[1],ER[1],Vp[1],ga[1]),\
>         ActionValues(P[2],ER[2],Vp[2],ga[2])]:
> IsBalanced2(Qp,Vp);
```

$$true$$

IsBalanced2 - *test if a policy is balanced*

Calling Sequence

IsBalanced2(**Qp**,**Vp**)

Parameters

`Qp` - list(Matrix); action-values for a policy

`Vp` - list(Vector[row]); value functions for a policy

Description

The IsBalanced2(**Qp**,**Vp**) function tests if a policy with action-values **Qp** and value functions **Vp** for two realizations is balanced.

Examples

```
> with(MDP):
```

```
> s:=3:a:=3:
> P:=[RandomTransitionProbability(s,a,5),\
>         RandomTransitionProbability(s,a,5)]:
> R:=[RandomReward(s,a,-2,2),RandomReward(s,a,-2,2)]:
> ER:=[ExpectedReward(P[1],R[1]),\
>         ExpectedReward(P[2],R[2])]:
> Pol:=RandomStochasticMatrix(a,s,5):
> ga:=[1/2,1/2]:
> balPol:=PolicyIteration2(P,ER,Pol,ga);
```

$$balPol := \begin{bmatrix} 0 & 0 & \frac{2}{5} \\ 1 & 1 & \frac{3}{5} \\ 0 & 0 & 0 \end{bmatrix}$$

```
> ERp:=[ExpectedRewardPolicy(ER[1],balPol),\
>         ExpectedRewardPolicy(ER[2],balPol)]:
> Pp:=[TransitionMatrix(P[1],balPol),\
>         TransitionMatrix(P[2],balPol)]:
> Vp:=[ValueFunction(ERp[1],Pp[1],ga[1]),\
>         ValueFunction(ERp[2],Pp[2],ga[2])];
```

$$Vp := [\begin{bmatrix} \frac{272}{1075}, & \frac{952}{1075}, & \frac{-58}{1075} \end{bmatrix}, \begin{bmatrix} \frac{-17}{20}, & \frac{-41}{35}, & \frac{-11}{10} \end{bmatrix}]$$

```
> Qp:=[ActionValues(P[1],ER[1],Vp[1],ga[1]),\
>         ActionValues(P[2],ER[2],Vp[2],ga[2])];
```

$$Qp := \begin{bmatrix} \begin{bmatrix} \frac{-416}{215} & \frac{1079}{1075} & \frac{-2014}{1075} \\ \frac{272}{1075} & \frac{952}{1075} & \frac{1246}{1075} \\ \frac{229}{215} & \frac{-2014}{1075} & \frac{-29}{1075} \end{bmatrix}, \begin{bmatrix} \frac{-5}{7} & \frac{-137}{70} & \frac{-191}{280} \\ \frac{-17}{20} & \frac{-41}{35} & \frac{-193}{140} \\ \frac{-193}{140} & \frac{-67}{70} & \frac{-83}{56} \end{bmatrix} \end{bmatrix}$$

```
> IsBalanced2(Qp,Vp);
```

$$true$$

11.3 Content of the CD-ROM

The attached CD-ROM contains the programs and source codes for

- `MDP` package, Sections 2 and 7,
- `SimRobo`, Sections 3 and 8,
- `RealRobo`, Sections 4 and 9.

Moreover, the publications [MR01a], [MR01b], [MR02], [MR03a], [MR03b] and the dissertation are included.

All programs can be installed by copying the `programs` directory with all its subdirectories to your hard disc. For details see the installation notes `info.txt` in each directory. For the `MDP` package the program `Maple 9` is required. `RealRobo` and `SimRobo` run on a Microsoft Windows system.

The source code for all programs can be found in the `sources` directory. The `Visual C++` project file for `SimRobo` is `ROBO3D.DSW` and `realrobo.dsw` for `RealRobo`.

Refer to the website of our project, `http://mathematik.uibk.ac.at/users/rl`, for the latest versions of the `MDP` package, `SimRobo` and `RealRobo`.

The following is an overview of the directories on the CD-ROM:

```
publications

programs              sources
  MDP-package           MDP-package
  RealRobo              RealRobo
    data                  data
                          LIB

  SimRobo               SimRobo
    slots                 slots
      mdpoa                 mdpoa
      mdpwf                 mdpwf
      oa                    oa
      wf                    wf
```

11.4 Contributions

In the following we give a list of the contributions of each author.

Both authors:

- Introduction, Bibliographical Remarks and Discussion
- Sections: 1.1, 1.3-1.7, 1.10, 3.1-3.4, 5.1-5.4, 5.6-5.7, 8.1-8.2.
- Simulator `SimRobo`, Sections 11.1.1-11.1.3.
- Publications in Section 11.5 and Chapter 6.

Andreas Matt:

- Chapters: 4 and 9.
- Sections: 1.9, 1.12.2-1.12.4, 3.5, 8.3.
- Program `RealRobo`, Section 11.1.4.

Georg Regensburger:

- Chapters: 2 and 7.
- Sections: 1.2, 1.8, 1.11, 1.12.1, 5.5.
- `MDP` package, Section 11.2.

Andreas Matt: Studies in Mathematics and Computer Science in Innsbruck, Vienna, Sevilla and Buenos Aires. Scholarship from the University of Innsbruck. Scholarships for short term research abroad. Teaching assistant at the Institute of Computer Science, University of Innsbruck.

Georg Regensburger: Studies in Mathematics in Innsbruck and Madrid. Scholarship from the University of Innsbruck. Research assistant at the Institute of Computer Science, Prof. Otmar Scherzer. Teaching assistant at the Institute of Computer Science, University of Innsbruck.

11.5 Publications

The publications are also available on our website `http://mathematik.uibk.ac.at/users/rl`.

11.5.1 Policy Improvement for several Environments

Abstract. In this paper we state a generalized form of the policy improvement algorithm for reinforcement learning. This new algorithm can be used to find stochastic policies that optimize single-agent behavior for several environments and reinforcement functions simultaneously. We first introduce a geometric interpretation of policy improvement, define a framework to apply one policy to several environments, and propose the notion of balanced policies. Finally we explain the algorithm and present examples.

See [MR01a] and the extended version [MR01b].

11.5.2 Generalization over Environments in Reinforcement Learning

Abstract. We discuss the problem of reinforcement learning in one environment and applying the policy obtained to other environments. We first state a method to evaluate the utility of a policy. Then we propose a general model to apply one policy to different environments and compare them. To illustrate the theory we present examples for an obstacle avoidance behavior in various block world environments.

See [MR03b] and [MR02].

11.5.3 Approximate Policy Iteration for several Environments and Reinforcement Functions

Abstract. We state an approximate policy iteration algorithm to find stochastic policies that optimize single-agent behavior for several environments and reinforcement functions simultaneously. After introducing a geometric interpretation of policy improvement for stochastic policies we discuss approximate policy iteration and evaluation. We present examples for two block-world environments and reinforcement functions.

See [MR03a].

Policy Improvement for several Environments

Andreas Matt ANDREAS.MATT@UIBK.AC.AT
Georg Regensburger GEORG.REGENSBURGER@UIBK.AC.AT
Institute of Mathematics[1], University of Innsbruck, Austria

Abstract

In this paper we state a generalized form of the policy improvement algorithm for reinforcement learning. This new algorithm can be used to find stochastic policies that optimize single-agent behavior for several environments and reinforcement functions simultaneously. We first introduce a geometric interpretation of policy improvement, define a framework to apply one policy to several environments, and propose the notion of balanced policies. Finally we explain the algorithm and present examples.

1. Idea

Until now reinforcement learning has been applied to learn behavior within one environment. Several methods to find optimal policies for one environment are known (Kaelbling et al., 1996; Sutton & Barto, 1998).

In our research we focus on a general point of view of behavior that appears independently from a single environment. As an example imagine that a robot should learn to avoid obstacles, a behavior suitable for more than one environment. Obviously a policy for several environments cannot - in general - be optimal for each one of them. Improving a policy for one environment may result in a worse performance in an other. Nevertheless it is often possible to improve a policy for several environments. Compared to multiagent reinforcement learning as in Bowling and Veloso (2000), where several agents act in one environment, we have one agent acting in several environments.

2. Equivalent and Improving Policies

We fix a finite Markov Decision Process $(S, \mathbf{A}, \mathbf{P}, \mathbf{R})$, use the standard definitions of value function V and Q-value and write $\pi(a \mid s)$ for the probability that action a is chosen in state s. We say that two policies π and $\tilde{\pi}$ are equivalent if their value functions coincide, i.e. $V^\pi = V^{\tilde{\pi}}$.

Theorem 1 *Two policies $\tilde{\pi}$ and π are equivalent if and only if*

$$\sum\nolimits_{a\in A} Q^\pi(a,s)\tilde{\pi}(a \mid s) = V^\pi(s) \text{ for all } s \in S.$$

This gives us a description of the equivalence class of a policy. We interpret $\pi(- \mid s)$ as a point on a standard simplex and the equivalence class as the intersection of the hyperplane H defined by $Q^\pi(- \mid s)$ and $V^\pi(s)$ with the simplex. See Figure 1 *left* for an example with three actions a_1, a_2 and a_3. The following theorem is

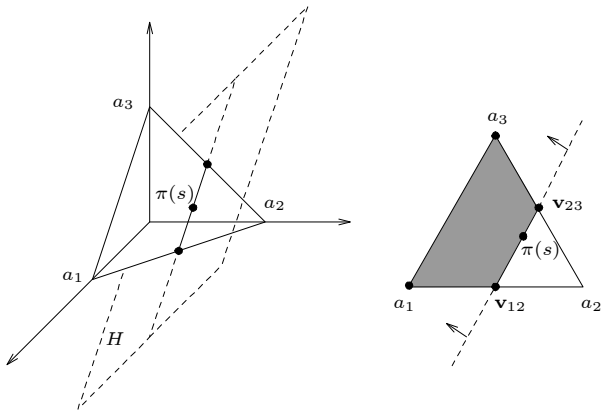

Figure 1. left: A policy in state s and its equivalence class *right:* Improving policies in state s

a general version of the policy improvement theorem.

Theorem 2 *Let π and $\tilde{\pi}$ be policies such that*

$$\sum\nolimits_{a\in A} Q^\pi(a,s)\tilde{\pi}(a \mid s) \geq V^\pi(s) \text{ for all } s \in S.$$

Then $V^{\tilde{\pi}} \geq V^\pi$. If additionally there exists an $s \in S$ such that $\sum Q^\pi(a,s)\tilde{\pi}(a \mid s) > V^\pi(s)$ then $V^{\tilde{\pi}} > V^\pi$.

We define the *set of improving policies* for π in s by

$$C^\pi_{\geq}(s) = \left\{\tilde{\pi}(- \mid s) : \sum Q^\pi(a,s)\tilde{\pi}(a \mid s) \geq V^\pi(s)\right\},$$

[1]We wish to thank Prof. Ulrich Oberst for his motivation, comments and support. This research was partially supported by "Forschungsstipendien an österreichische Graduierte" and Project Y-123 INF.

and in analogy the *set of strictly improving policies* $C^\pi_>(s)$ and the *set of equivalent policies* $C^\pi_=(s)$ for π in s. We define the *set of strictly improving actions* of π in s by $A^\pi_>(s) = \{a : Q^\pi(a,s) > V^\pi(s)\}$.

The set of improving policies $C^\pi_\geq(s)$ is a polytope given by the intersection of a half-space and a standard simplex. Its vertices are $\text{vert}\left(C^\pi_\geq(s)\right) = \text{vert}(C^\pi_=(s)) \cup A^\pi_>(s)$. See Figure 1 *right*, where $A^\pi_>(s) = \{a_1, a_3\}$, $\text{vert}(C^\pi_=(s)) = \{\mathbf{v}_{12}, \mathbf{v}_{23}\}$ and $C^\pi_\geq(s)$ is the shaded area, the side marked by the small arrows.

3. Policies for several Environments

Consider a robot and its sensors to perceive the world. All possible sensor values together represent all possible states for the robot. In each of these states the robot can perform some actions. We call all possible states and actions the *state action space (SAS)* $\mathbb{E} = (S, \mathbf{A})$. Now we put the robot in a physical environment, where we can observe all possible states for this environment, a subset $S_E \subset S$ of all possible states in general, and the transition probabilities $\mathbf{P}_E$. We call $E = (S_E, \mathbf{A}, \mathbf{P}_E)$ a *realization* of an SAS.

Let $\mathbf{E} = (E_i, \mathbf{R}_i)_{i=1\ldots n}$ be a finite family of realizations of an SAS with rewards $\mathbf{R}_i$. Since the actions are given by the SAS it is clear what is meant by a policy π for $\mathbf{E}$. For each $(E_i, \mathbf{R}_i)$ we can calculate the value function, which we denote by V_i^π. We define the *set of improving policies* of π in $s \in S$ by

$$C^\pi_\geq(s) = \bigcap\nolimits_{i\in[n],\ s\in S_i} C^\pi_{i,\geq}(s)$$

and the *set of strictly improving policies* of π in s by

$$C^\pi_>(s) = \left\{ \begin{array}{c} \tilde\pi(- \mid s) \in C^\pi_\geq(s) \text{ such that} \\ \exists\ i \in [n] \text{ with } \tilde\pi(- \mid s) \in C^\pi_{i,>}(s) \end{array} \right\},$$

where $[n] = \{1, \ldots, n\}$. The set of improving policies of π in s is the intersection of a finite number of half-spaces through a point with a standard simplex.

Theorem 3 *Let $\tilde\pi$ be a policy for $\mathbf{E}$ such that*

$$\tilde\pi(- \mid s) \in C^\pi_\geq(s) \text{ for all } s \in S.$$

Then $V_i^{\tilde\pi} \geq V_i^\pi$ for all $i \in [n]$. If additionally there exist an $s \in S$ with $\tilde\pi(- \mid s) \in C^\pi_>(s)$ then there exists an $i \in [n]$ such that $V_i^{\tilde\pi} > V_i^\pi$.

In order to describe $C^\pi_\geq(s)$ and find an $\tilde\pi(- \mid s) \in C^\pi_>(s)$ we consider its vertices. We call the vertices of $C^\pi_\geq(s)$ *improving vertices* and define the *strictly improving vertices* by $\text{vert}(C^\pi_>(s)) = \text{vert}(C^\pi_\geq(s)) \cap C^\pi_>(s)$. There exist several algorithm to find all vertices of a polytope (Fukuda, 2000). Linear Programming methods can be used to decide whether there exist strictly improving vertices and to find one (Schrijver, 1986). Observe that for a single environment the strictly improving vertices are just the set of strictly improving actions.

Let $s \in S$. We define π_s as the set of all policies that are arbitrary in s and equal π otherwise. We call a policy *balanced* if and only if for all $s \in S$ and all $\tilde\pi \in \pi_s$ either $V_i^{\tilde\pi} = V_i^\pi$ for all $i \in [n]$ or there exists $i \in [n]$ such that $V_i^{\tilde\pi} < V_i^\pi$. This means that if one changes a balanced policy in one state s it is the same for all environments or it gets worse in at least one. Compare to the notion of an equilibrium point in game theory (Nash, 1951). Note that for one environment the notions of optimal and balanced policies coincide.

Theorem 4 *A policy π is balanced if and only if there are no strictly improving policies, i.e. $C^\pi_>(s) = \emptyset$ for all $s \in S$.*

4. General Policy Improvement

We state a generalized form of the policy improvement algorithm for a family of realizations of an SAS which we call *general policy improvement* (algorithm 1). The idea is to improve the policy by choosing in each state a strictly improving vertex. If there are no strictly improving vertices the policy is balanced and the algorithm terminates.

Input: a policy π and a family of realizations $(E_i, \mathbf{R}_i)$
Output: a balanced policy $\tilde\pi : V_i^{\tilde\pi} \geq V_i^\pi$ for all $i \in [n]$

```
π̃ ⟵ π
repeat
    calculate V_i^π̃ and Q_i^π̃ for all i ∈ [n]
    for all s ∈ S do
        if vert(C_>^π̃(s)) ≠ ∅ then
            choose π'(− | s) ∈ vert(C_>^π̃(s))
            π̃(− | s) ⟵ π'(− | s)
until vert(C_>^π̃(s)) = ∅ for all s ∈ S
```

Algorithm 1: General Policy Improvement

In each step of the algorithm we try to choose a strictly improving vertex. Different choices may result in different balanced policies and influence the number of improvement steps before termination. The algorithm includes policy improvement for one environment as a special case.

A geometric interpretation of one step of the general policy improvement algorithm for three states and two realizations can be seen in Figure 2. In state s_1 there are no strictly improving vertices. In state s_2 there are three strictly improving vertices, one of them is the action a_3. In state s_3 there are only two, both of

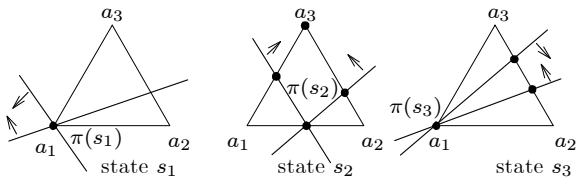

Figure 2. Improving policies for two realizations

them a stochastic combination of a_2 and a_3.

5. Examples

All experiments are made with a 10x10 gridworld simulator to learn an obstacle avoidance behavior. The robot has 4 sensors, forward, left, right, and back, with a range of 5 blocks each. There are 3 actions in each state: move forward, left and right. The robot gets rewarded if it moves away from obstacles, it gets punished if it moves towards obstacles.

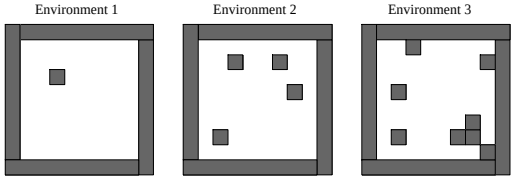

Figure 3. Three different environments

We choose three environments (see Figure 3) with the same reinforcement function and run the algorithm. In all experiments we start with the random policy. We calculate in each step all strictly improving vertices and choose one randomly. In order to evaluate and compare policies we consider the average utilities of all states, and normalize it, with 1 being an optimal and 0 the random policy in this environment. Four sample experiments show performances in each environment of the different balanced policies learned.

Experiment:	1	2	3	4
Environment 1	0.994	0.771	0.993	0.826
Environment 2	0.862	0.876	0.788	0.825
Environment 3	0.872	0.905	0.975	0.878

Figure 4 shows the progress of the algorithm for each environment in experiment 2. In all experiments the algorithm terminates after 6 to 8 iteration steps.

6. Discussion

The general policy improvement algorithm can be used to improve a policy for several realizations of a state

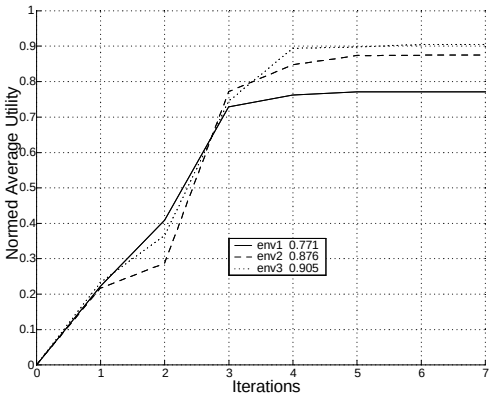

Figure 4. The progress of general policy improvement for each environment.

action space simultaneously. This means that it can be used to learn a policy for several environments and several reinforcement functions together. A useful application is to add a new environment or behavior to an already optimal policy, without changing its performance. We have already implemented Value Iteration for several realizations which leads to an extension of Q-learning. Our future research focuses on the implementation of on-line algorithms, methods to find the strictly improving vertices and to decide which of them are best regarding to learning speed. For more detailed information please consult the extended version of this paper on `http://mathematik.uibk.ac.at/~rl`.

References

Bowling, M., & Veloso, M. (2000). *An analysis of stochastic game theory for multiagent reinforcement learning* (Technical Report CMU-CS-00-165).

Fukuda, K. (2000). Frequently asked questions in polyhedral computation. `http://www.ifor.math.ethz.ch/~fukuda/polyfaq/polyfaq.html`.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, pp. 237–285.

Nash, J. (1951). Non-cooperative games. *Ann. of Math. (2)*, *54*, 286–295.

Schrijver, A. (1986). *Theory of linear and integer programming*. Chichester: John Wiley & Sons Ltd.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.

ARTÍCULO

Generalization over Environments in Reinforcement Learning

Andreas Matt and Georg Regensburger.

Institute of Mathematics .University of Innsbruck.
Technikerstr. 25/7. A-6020 Innsbruck. Austria
{andreas.matt, georg.regensburger}@uibk.ac.at

Keywords: reinforcement learning, multiple environments and objectives, generalization, obstacle avoidance.

.

We discuss the problem of reinforcement learning in one environment and applying the policy obtained to other environments. We first state a method to evaluate the utility of a policy. Then we propose a general model to apply one policy to different environments and compare them. To illustrate the theory we present examples for an obstacle avoidance behavior in various block world environments.

Generalization over Environments in Reinforcement Learning

Andreas Matt and Georg Regensburger
Institute of Mathematics
University of Innsbruck
Technikerstr. 25/7
A-6020 Innsbruck
Austria
{andreas.matt, georg.regensburger}@uibk.ac.at

Abstract

We discuss the problem of reinforcement learning in one environment and applying the policy obtained to other environments. We first state a method to evaluate the utility of a policy. Then we propose a general model to apply one policy to different environments and compare them. To illustrate the theory we present examples for an obstacle avoidance behavior in various block world environments.

Keywords: reinforcement learning, multiple environments and objectives, generalization, obstacle avoidance.

1 Idea

Until now reinforcement learning has been applied to abstract behavior within one environment. Several methods to find optimal or near optimal policies are known, see Bertsekas and Tsitsiklis [2], Kaelbling et al. [4], Sutton and Barto [6]. The fact that a policy learned in one environment can be successfully applied to other environments has been observed, but not investigated in detail.

In our research we focus on a general point of view of behavior that appears independently from a single environment. As an example imagine that a robot should learn to avoid obstacles. What we have in mind is a behavior suitable for any kind of environment.

We give a method to optimize single-agent behavior for several environments and reinforcement functions by learning in several environments simultaneously in [5]. Now we address the problem of learning in one and applying the policy obtained to other environments. We discuss the influence of the environment on the ability to generalize over other environments. How do good learning environments look like?

2 Preliminaries

We propose the following notation for reinforcement learning models, which allows us to treat several environments and reinforcement functions. The main difference to the standard definition of a Markov decision process, see Howard [3] or White [8], is, that we separate the reinforcement function from the environment.

An *environment* is given by

- A finite set S of *states*.

- A family $\mathbf{A} = (A(s))_{s \in S}$ of finite sets of *actions*.
  The set $A(s)$ is interpreted as the set of all allowed actions in state s.
- A family of probabilities $\mathbf{P} = P(- \mid a, s)$ on S with (a, s) such that $s \in S$ and $a \in A(s)$.
  We interpret $P(s' \mid a, s)$ as the *transition probability* that performing action a in state s leads to the successor state s'.

Let $E = (S, \mathbf{A}, \mathbf{P})$ be an environment. A *policy* for E is given by

- A family of probabilities $\pi = \pi(- \mid s)$ on $A(s)$ for $s \in S$.
  We interpret $\pi(a \mid s)$ as the probability that action a is chosen in state s.

A *Markov decision process (MDP)* is given by an environment $E = (S, \mathbf{A}, \mathbf{P})$ and

- A family $\mathbf{R} = R(s', a, s) \in \mathbb{R}$ with (s', a, s) such that s', $s \in S$ and $a \in A(s)$.
  The value $R(s', a, s)$ represents the *expected mean reward* if performing a in state s leads to the *successor state* s'. The family $\mathbf{R}$ is called a *reinforcement function*.

The goal of reinforcement learning is to find policies that maximize the sum of the expected rewards. Let $(E, \mathbf{R})$ be a MDP and $0 \leq \gamma < 1$ be a discount rate. Let π be a policy for E. The *value function* or *utility* of policy π in state s is given by

$$V^\pi(s) = \sum_a \pi(a \mid s) \cdot \left(R(a, s) + \gamma \sum_{s'} V^\pi(s') P(s' \mid a, s) \right),$$

where

$$R(a, s) = \sum_{s' \in S} R(s', a, s) P(s' \mid a, s)$$

is the expected reward of action a in state s. The value function describes the expected (discounted) sum of rewards following a certain policy starting in state s. The *action–value* of π

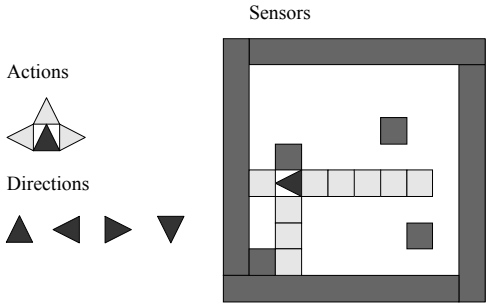

Figure 1: States and actions used in the simulator. The set of actions $A(s) = \{$forward, left, right$\}$ and the sensor values in this example are $s = (4, 2, 5, 0)$

of action $a \in A(s)$ in state s is defined by

$$Q^\pi(a, s) = R(a, s) + \gamma \sum_{s'} V^\pi(s') P(s' \mid a, s).$$

The action value is the expected (discounted) sum of rewards if action a is chosen in state s and afterwards policy π is followed. A policy π^* is *optimal* if $V^\pi(s) \leq V^{\pi^*}(s)$ for all policies π and all $s \in S$. There exists at least one deterministic optimal policy and all optimal policies share the same value function, which we denote by V^*.

3 The Blockworld Simulator Robo

To illustrate the theory developed in this paper we use the simple blockworld simulator Robo[1] to learn an obstacle avoidance behavior. The simulated robot acts in a 10x10 blockworld and has four sensors, forward, left, right, and back, with a range of 5 blocks each. The sensor values are encoded in a vector $s = (s_1, ..., s_4)$. The values vary between 0 and 5, where 5 means that there is a block right in front and 0 means that there is no block in the next 5 fields. There are three actions in each state, move forward, left and right one block (see Fig. 1).

The robot gets rewarded if it is far from obstacles or moves away from obstacles, it gets punished if it bumps into an obstacle or it moves

[1] More information on the blockworld simulator Robo is available at `http://mathematik.uibk.ac.at/users/rl`.

towards obstacles. Let $s = (s_1, \ldots, s_4)$ and $s' = (s'_1, \ldots, s'_4)$ be the sensor values for the state and the successor state. The reinforcement function is defined as follows:

$$R(s', a, s) = \begin{cases} +1 & \text{if } s'_i \leq 3 \text{ for } i = 1 \ldots 4 \\ & \text{or } \sum s'_i - s_i \leq -1, \\ -1 & \text{if it bumps into the wall} \\ & \text{or } \sum s'_i - s_i \geq 0.0, \\ 0 & \text{else.} \end{cases}$$

The optimal value function is calculated using value iteration, see Bellman [1] or Sutton and Barto [6]. In all examples and experiments we use the discount rate $\gamma = 0.95$ and accuracy 10^{-6}.

4 Utility of a Policy

Let $(E, \mathbf{R})$ be a MDP and γ a discount rate. Let π be a policy for E. We define the *utility* of π, denoted by $V(\pi)$, by

$$V(\pi) = \frac{1}{|S|} \sum_{s \in S} V^\pi(s).$$

It is the average utility of all states.

Let π^* be an optimal policy. Then $V(\pi) \leq V(\pi^*)$ for all policies π. A policy is optimal if and only if its utility is maximal. The discounted utility allows us to describe a policy with one number and thus to easily compare all policies.

Example 1 *We calculate the utilities of the random policy and the optimal policy in a sample blockworld (see Fig. 2) with the reinforcement function of Sect. 3. The discounted utility of the random policy* π^{rand}*, that is all actions are chosen with the same probability,* $V(\pi^{\text{rand}}) = -4.712$ *and the discounted utility of the optimal policy* $V(\pi^*) = 18.511$.

5 The State Action Space

To apply a policy to different environments we need some new notions, which are motivated by the following example.

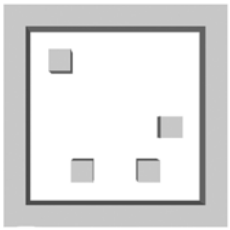

Figure 2: A sample blockworld

Consider a robot with its sensors to perceive the world. All possible sensor values together represent all possible states for the robot. In each of these states the robot can perform some actions. We call all possible states and actions the state action space.

A *state action space* $\mathbb{E} = (S, \mathbf{A})$ is given by

- A finite set S of states.
- A family $\mathbf{A} = (A(s))_{s \in S}$ of finite sets of actions.

Since the actions are given by the state action space it is clear what is meant by a policy for $\mathbb{E}$. Now we imagine the robot in a physical environment, where we can observe all possible states for this environment, a subset $S_E \subset S$ of all possible states in general, and the transition probabilities $\mathbf{P}_E$. We call an environment $E = (S_E, \mathbf{A}_E, \mathbf{P}_E)$ a *realization* of a state action space $\mathbb{E}$ if

$$S_E \subset S \text{ and } \mathbf{A}_E(s) = \mathbf{A}(s) \text{ for all } s \in S.$$

We call a MDP $(E, \mathbf{R})$ a *realization* of $\mathbb{E}$ if the environment E is a realization of $\mathbb{E}$.

Let $\mathbb{E} = (S, \mathbf{A})$ be a state action space and $E = (S_E, \mathbf{A}_E, \mathbf{P}_E)$ be a realization of $\mathbb{E}$. In order to apply the policy π for E to another realization of $\mathbb{E}$ we (randomly) extend the policy π to a policy π^e for $\mathbb{E}$ in the following way:

$$\pi^e = \begin{cases} \pi^e(a \mid s) = \pi(a \mid s) & \text{for } s \in S_E, \\ \pi^e(a \mid s) = \frac{1}{|A(s)|} & \text{for } s \in S \setminus S_E, \end{cases}$$

and $a \in A(s)$. This means that in unknown states the actions are chosen randomly. Once extended, the policy π^e can be applied to all realizations of $\mathbb{E}$. We simply write π for π^e.

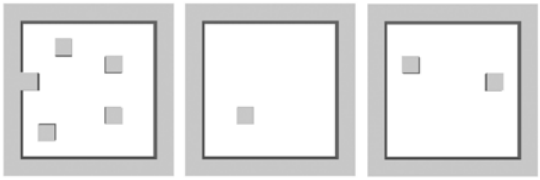

Figure 3: Sample blockworlds for the realizations B_1, B_2 and B_3

Example 2 *In the blockworld simulator the state action space is given by all possible sensor values $S = \{(s_1, \ldots, s_4)$ with $0 \leq s_i \leq 5\}$ and in each state the actions forward, left, right. We consider three different blockworlds (see Fig. 3). By observing the possible states and transition probabilities in each blockworld we obtain realizations B_1, B_2 and B_3 with the reinforcement function of Sect. 3.*
We extend an optimal policy π_1^* for B_1 and apply it to the realizations B_2 and B_3. Evaluating the utilities of π_1^* in all realizations, and the optimal and random policies, π_i^* resp. π_i^{rand}, for each realization B_i, we obtain:

Results	π_1^*	π_i^*	π_i^{rand}
B_1	18.448	18.448	−6.060
B_2	15.190	18.877	−4.873
B_3	3.081	18.951	−3.678

(1)

Note that π_1 performs relatively well in B_2 and bad in B_3.

Example 3 *Let the state action space and B_2 be as in Example 2. We consider the empty blockworld and obtain realization B_0. Comparing the utilities of their extended optimal policies π_i^* we obtain:*

Results	π_2^*	π_0^*	π_i^{rand}
B_2	18.877	9.360	−4.873
B_0	19.264	19.264	−4.473

Observe that the optimal policy for B_2 remains optimal in B_0, but not viceversa.

6 Optimal Actions

Let $\mathbb{E} = (S, \mathbf{A})$ be a state action space and $(E, \mathbf{R})$ be a realization of $\mathbb{E}$. There can be several optimal policies for this realization. While they are equally optimal in $(E, \mathbf{R})$ they can perform better or worse in other realizations of $\mathbb{E}$. Since we do not know which of the optimal policies performs best in other realizations we propose a random choice between all optimal actions.

Let $(E, \mathbf{R})$ be a MDP and γ a discount rate. Let $s \in S$. Let V^* be the optimal value function and Q^* be the optimal action values. We define the *set of optimal actions* in s by

$$A^*(s) = \{a^* \in A(s) \text{ with } Q^*(a^*, s) = V^*(s)\}.$$

We define the *random optimal policy* $\pi^{\text{rand}\,*}$ by

$$\pi^{\text{rand}\,*} = \begin{cases} \pi^*(a \mid s) = \frac{1}{|A^*(s)|} & \text{for } s \in S_E \\ & \text{and } a \in A^*(s), \\ \pi^*(- \mid s) = 0 & \text{else.} \end{cases}$$

Example 4 *We repeat the Example 2 with the random optimal policy $\pi_1^{\text{rand}\,*}$ for B_1 and obtain:*

Results	$\pi_1^{\text{rand}\,*}$	π_i^*	π_i^{rand}
B_1	18.448	18.448	−6.060
B_2	15.166	18.877	−4.873
B_3	3.202	18.951	−3.678

(2)

Comparing (1) and (2) we find that $\pi_1^{\text{rand}\,*}$ performs better in B_3 and slightly worse in B_2.

7 Utility of a Policy for a Family of Realizations

Let $\mathbb{E} = (S, \mathbf{A})$ be a state action space. Let $\mathbf{E} = (E_i, \mathbf{R}_i)_{i=1\ldots n}$ be a finite family of realizations of $\mathbb{E}$ and $(\gamma_i)_{i=1\ldots n}$ discount rates. Note that the reinforcement functions and discount rates can be different for each realization. Let π be a policy for $\mathbb{E}$. We calculate for each $(E_i, \mathbf{R}_i)$ and γ_i the utility of π, which we denote by $V_i(\pi)$.

We define the utility of π for a finite family of realizations $\mathbf{E}$ by

$$V_{\mathbf{E}}(\pi) = \frac{1}{n} \sum_{i=1\ldots n} V_i(\pi),$$

the average utility of π in all realizations. This utility is used to measure the performance of policies for a family of realizations. We say that a policy π performs better in $\mathbf{E}$ than policy π' if $V_{\mathbf{E}}(\pi) > V_{\mathbf{E}}(\pi')$.

8 Generalizing over Environments

Recall our main idea to learn a policy in one environment and apply it to other environments. We say that an environment generalizes over other environments if a policy learned in this environment performs well in the others.

In the framework defined above we can formulate this more precisely using random optimal policies: Let $\mathbb{E} = (S, \mathbf{A})$ be a state action space and $\mathbf{E} = (E_i, \mathbf{R}_i)_{i=1\ldots n}$ be a finite family of realizations of $\mathbb{E}$. We say that $(E_i, \mathbf{R}_i)$ *generalizes* better over $\mathbf{E}$ than $(E_j, \mathbf{R}_j)$ if $V_{\mathbf{E}}(\pi_i^{\mathrm{rand}\,*}) > V_{\mathbf{E}}(\pi_j^{\mathrm{rand}\,*})$.

This general approach includes generalization over reinforcement functions. In our context generalization is established without using methods such as neural networks.

9 Experiments

We observed in the above examples that the environment influences the utility of its optimal policies for other realizations. Our question now is, how to choose an environment that generalizes well over other environments. We conduct three experiments.

In all experiments we use the blockworld simulator as described in Sect. 3. We choose a family $\mathbf{E}$ of realizations of the state action space in Example 2. We calculate the random optimal policies (Sect. 6) for all realizations of $\mathbf{E}$ and compare their utilities for $\mathbf{E}$ (Sect. 7).

9.1 How Many Blocks?

In the first experiment we discuss how many blocks we should distribute in an empty blockworld. We generate environments with one to ten blocks distributed randomly. We generate ten environments for each number of blocks and obtain a family $\mathbf{E}$ of 100 realizations. The utility of the random optimal policies of the n-block environments for all realizations of $\mathbf{E}$ are then averaged for $n = 1 \ldots 10$. The results of two experiments are shown in Fig. 4.

We observe that the average utility first increases with the number of blocks and then decreases if there are more than three blocks. This confirms the intuitive idea that a "good" environment to generalize should be neither too simple nor too complex, more on the simple side though.

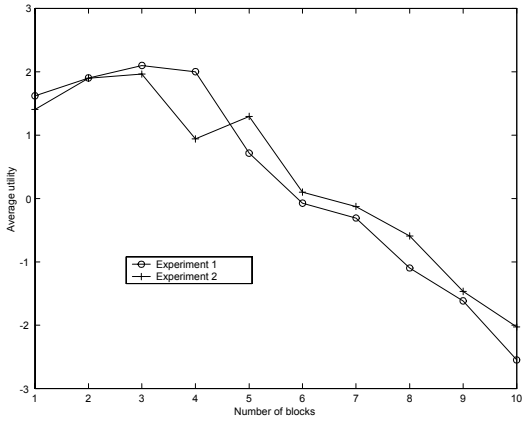

Figure 4: Averaged utility of the random optimal policies of the n-block environments for all realizations

9.2 One-Block Environments

We consider all 64 possible one-block environments. The question is now, on which position we put the block in order to have a good environment to generalize over all one-block environments. Before you read on, you can think by yourself where to put the block.

The checkerboard in Fig. 5 represents the utilities. The brighter the color of the block the higher is the utility of the random optimal policy for all one-block environments of the environment with a block on this position. The utilities vary between 13.584 and 14.873. Note the symmetry of the environments and the eight best ones.

The checkerboard in Fig. 6 shows the number of different states in each one-block environment. Again the brighter the color of the block the higher is the number. The number of states lies between 65 and 104. The best one-block environment has 100 different states.

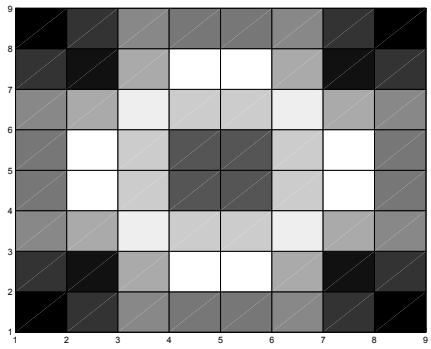

Figure 5: The checkerboard shows the utility for all 64 one-block environments

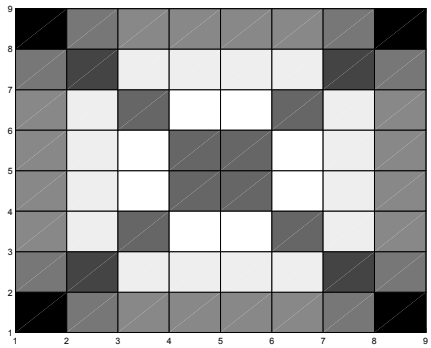

Figure 6: The checkerboard represents the number of different states for all 64 one-block environments

9.3 Two-Block Environments

We extend the above experiment to all two-block environments. There are 2016 different environments to compare. Using symmetries we end up with 632 which still results in a fair amount of calculations. The best and worst environment are shown in Fig. 7.

The utility of the random optimal policy of the best environment is 11.195, the utility of the worst environment 3.404. The number of states ranges from 51 to 143. The best environment has 126 different states, the worst 60. We observe that "good" environments to generalize have a high number of different states.

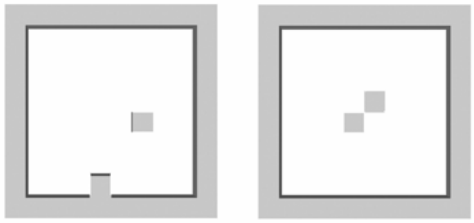

Figure 7: The *left* blockworld is the best, the *right* one the worst environment generalizing over all two-block environments

10 Discussion

We give a method to apply policies to different environments and compare them. The results in Sect. 9 emphasize the influence of the realization, that is the environment with its states and transition probabilities and the reinforcement function, on the ability to generalize over other realizations. Example 3 even shows that an environment may include all the necessary information to learn an optimal policy for other environments.

To obtain one policy suitable for many realizations it is important to choose an appropriate environment to learn in. Our future work consists in finding a priori criteria, such as the number of states, complexity of transitions, rewards, etc., to predict the ability of an environment to generalize.

In Sect. 6 we extend the policy randomly for unknown states. Applying neural networks to choose similar actions for similar unknown states as in Touzet [7] could improve the method.

Acknowledgements

This research was partially supported by the Austrian Science Fund (FWF), Project Y-123 INF.

References

[1] Bellman, R. E.: Dynamic Programming. Princeton University Press. Princeton (1957)

[2] Bertsekas, D. P., Tsitsiklis, J. N.: Neuro-dynamic programming. Athena Scientific. Belmont, MA (1996)

[3] Howard, R. A.: Dynamic Programming and Markov Processes. MIT Press. Cambridge, MA (1960)

[4] Kaelbling, L. P., Littman, M. L., Moore, A. W.: Reinforcement learning: A survey. Journal of Artificial Intelligence Research **4** (1996) 237–285

[5] Matt, A., Regensburger, G.: Policy Improvement for Several Environments. Proceedings of the 5th European Workshop on Reinforcement Learning (EWRL-5) (2001) 30–32

[6] Sutton, R. S., Barto, A. G.: Reinforcement Learning: An Introduction. MIT Press. Cambridge, MA (1998)

[7] Touzet, C.: Neural Reinforcement Learning for Behavior Synthesis. Robotics and Autonomous Systems **22** (1997) 251–281

[8] White, D. J.: Markov Decision Processes. John Wiley & Sons Ltd. Chichester (1993)

Approximate Policy Iteration for several Environments and Reinforcement Functions

Andreas Matt ANDREAS.MATT@UIBK.AC.AT
Georg Regensburger GEORG.REGENSBURGER@UIBK.AC.AT
Institute of Mathematics, University of Innsbruck, Austria

Abstract

We state an approximate policy iteration algorithm to find stochastic policies that optimize single-agent behavior for several environments and reinforcement functions simultaneously. After introducing a geometric interpretation of policy improvement for stochastic policies we discuss approximate policy iteration and evaluation. We present examples for two blockworld environments and reinforcement functions.

1. Introduction

Reinforcement learning methods usually achieve optimal policies for one reinforcement function in one environment (Bertsekas & Tsitsiklis, 1996; Kaelbling et al., 1996). Multicriteria reinforcement learning is concerned with optimizing several reinforcement functions in one environment. Gábor et al. (1998) order the different criteria, Wakuta (1995) discusses policy improvement to find optimal deterministic policies for vector-valued Markov decision processes. In our research we focus on finding stochastic policies, which can perform better than deterministic policies, for several environments and reinforcement functions.

2. MDP's and State Action Space

An *environment* $E = (S, \mathbf{A}, \mathbf{P})$ is given by a finite set S of *states*, a family $\mathbf{A} = (A(s))_{s\in S}$ of finite sets of *actions* and a family $\mathbf{P} = P(- \mid a, s)_{s\in S, a\in A(s)}$ of *transition probabilities* on S. A *policy* for E is given by a family $\pi = \pi(- \mid s)_{s\in S}$ of probabilities on $A(s)$. A *Markov decision process (MDP)* is given by an environment $E = (S, \mathbf{A}, \mathbf{P})$ and a family $\mathbf{R} = R(s', a, s)_{s',s\in S, a\in A(s)}$ of *rewards* in $\mathbb{R}$. Let $(E, \mathbf{R})$ be a MDP and $0 \leq \gamma < 1$ be a discount rate. Let π be a policy for E. We denote by $V^\pi(s)$ the (discounted) *value function* or *utility* of policy π in state s and write $Q^\pi(a, s)$ for the (discounted) *action-value*. The goal of reinforcement learning is to find policies that maximize the value function.

To apply one policy to different environments we introduce the following notions. A *state action space* $\mathbb{E} = (S, \mathbf{A})$ is given by a finite set S of states and a family $\mathbf{A} = (A(s))_{s\in S}$ of finite sets of actions. We call an environment $E = (S_E, \mathbf{A}_E, \mathbf{P}_E)$ and a MDP $(E, \mathbf{R})$ a *realization* of a state action space if the set of states is a subset of S and the actions for all states are the same as in the state action space, that is $S_E \subset S$ and $\mathbf{A}_E(s) = \mathbf{A}(s)$ for all $s \in S_E$. We can define policies for a state action space which can be applied to any realization.

3. Policy Improvement

3.1 One Realization

The policy improvement theorem for stochastic policies gives a sufficient criterion to improve a given policy π. Let $\tilde{\pi}$ be a policy such that

$$\sum\nolimits_{a\in A(s)} Q^\pi(a, s)\tilde{\pi}(a \mid s) \geq V^\pi(s) \text{ for all } s \in S. \quad (1)$$

Then $V^{\tilde{\pi}} \geq V^\pi$, that is $V^{\tilde{\pi}}(s) \geq V^\pi(s)$ for all $s \in S$. If additionally there exists an $s \in S$ such that the inequality (1) is strict then $V^{\tilde{\pi}} > V^\pi$. A usual choice for *policy improvement* is $\tilde{\pi}(a \mid s) = 1$ for an action $a \in A(s)$ such that $Q^\pi(a, s) = \max_a Q^\pi(a, s)$. Repeating policy improvement leads to *policy iteration*.

Considering all stochastic policies satisfying (1), we define the *set of improving policies* for π in s by

$$C^\pi_{\geq}(s) = \left\{\tilde{\pi}(- \mid s) : \sum Q^\pi(a, s)\tilde{\pi}(a \mid s) \geq V^\pi(s)\right\}.$$

The *set of strictly improving policies* $C^\pi_{>}(s)$ and the *set of equivalent policies* $C^\pi_{=}(s)$ for π in s are defined analogously. We define the *set of strictly improving actions* of π in s by $A^\pi_{>}(s) = \{a : Q^\pi(a, s) > V^\pi(s)\}$.

The set of improving policies $C^\pi_{\geq}(s)$ is a polytope given by the intersection of a half-space and a standard simplex. Its vertices are

$$\operatorname{vert}\left(C^\pi_{\geq}(s)\right) = \operatorname{vert}(C^\pi_{=}(s)) \cup A^\pi_{>}(s). \quad (2)$$

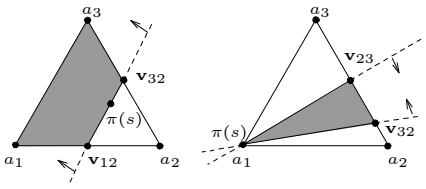

Figure 1. left: Improving policies for one realization *right:* Improving policies for two realizations

See Figure 1 *left*, where $C^\pi_\geq(s)$ is the shaded area, the side marked by the small arrows, $A^\pi_>(s) = \{a_1, a_3\}$ and $\text{vert}(C^\pi_=(s)) = \{\mathbf{v}_{12}, \mathbf{v}_{32}\}$. Let $A(s) = \{a_1, \ldots, a_n\}$, $\mathbf{c} = (c_1, \ldots, c_n) \in \mathbb{R}^n$ with $c_i = Q^\pi(a_i, s)$ and $b = V^\pi(s)$. Let $\mathbf{e}_i \in \mathbb{R}^n$ denote the ith standard basis vector. Then $\text{vert}\,(C^\pi_=(s))$ is $\{\mathbf{e}_k : c_k = b\} \cup$

$$\left\{\mathbf{v}_{ij} = \frac{b - c_j}{c_i - c_j}\mathbf{e}_i + \frac{c_i - b}{c_i - c_j}\mathbf{e}_j : c_i > b,\, c_j < b\right\}. \quad (3)$$

3.2 Several Realizations

Let $\mathbb{E} = (S, \mathbf{A})$ be a state action space. We want to improve a policy π for two realizations $(E_1, \mathbf{R}_1)$ and $(E_2, \mathbf{R}_2)$ of $\mathbb{E}$ with value functions V^π_i and Q^π_i for discount rates γ_i, $i = 1, 2$. Let $\tilde{\pi}$ be a policy such that

$$\sum\nolimits_{a \in A(s)} Q^\pi_1(a, s)\tilde{\pi}(a \mid s) \geq V^\pi_1(s) \text{ and} \quad (4)$$

$$\sum\nolimits_{a \in A(s)} Q^\pi_2(a, s)\tilde{\pi}(a \mid s) \geq V^\pi_2(s) \quad (5)$$

for all $s \in S_1 \cap S_2$ and that $\tilde{\pi}\,(- \mid s)$ satisfies (4) or (5) if s is only contained in S_1 or S_2 respectively. Then $V^{\tilde{\pi}}_1 \geq V^\pi_1$ and $V^{\tilde{\pi}}_2 \geq V^\pi_2$, see equation (1).

We define the *set of improving policies* $C^\pi_\geq(s) = C^\pi_{1,\geq}(s) \cap C^\pi_{2,\geq}(s)$ for π in $s \in S_1 \cap S_2$. The *set of strictly improving policies* $C^\pi_>(s)$ is given by all policies $\tilde{\pi}\,(- \mid s) \in C^\pi_\geq(s)$ such that one inequality (4) or (5) is strict. If s is only contained in S_1 or S_2 we use the definition from the previous subsection. Let $\tilde{\pi}$ a policy such that $\tilde{\pi}(- \mid s) \in C^\pi_\geq(s)$ for all $s \in S$ and $\tilde{\pi}(- \mid s) \in C^\pi_>(s)$ for at least one s. Then $\tilde{\pi}$ performs better than π since $V^{\tilde{\pi}}_i \geq V^\pi_i$ and $V^{\tilde{\pi}}_1 > V^\pi_1$ or $V^{\tilde{\pi}}_2 > V^\pi_2$ by the previous subsection.

We call a policy *balanced* if $C^\pi_>(s)$ is empty for all s. In general there exist several balanced policies which can be stochastic. In one environment a policy is optimal if and only if it is balanced. For further details and policy iteration for the general case with a finite family of realizations see Matt and Regensburger (2001).

To describe $C^\pi_\geq(s)$ and find a $\tilde{\pi}(- \mid s) \in C^\pi_>(s)$ we consider its vertices, see Figure 1 *right*. We call the vertices $\text{vert}(C^\pi_\geq(s))$ *improving vertices* and define the *strictly improving vertices* by $\text{vert}(C^\pi_>(s)) = \text{vert}(C^\pi_\geq(s)) \cap C^\pi_>(s)$. For one realization the strictly improving vertices are the strictly improving actions. To find all strictly improving vertices for two realizations and an $s \in S_1 \cap S_2$ we take all elements from $\text{vert}(C^\pi_{1,\geq}(s)) \cup \text{vert}(C^\pi_{2,\geq}(s))$ that are in $C^\pi_{1,>}(s)$ or $C^\pi_{2,>}(s)$, where $\text{vert}(C^\pi_{i,\geq}(s))$ are given by (2) and (3).

4. Approximate Policy Iteration

For policy iteration we need the action-values. If the model of the environment is not given explicitly we can approximate them. We use a SARSA related method, see algorithm 1, (Sutton & Barto, 1998).

repeat
  choose $s \in S$ and $a \in A(s)$ derived from π
  take action a and observe r and s'
  choose $a' \in A(s')$ according to π
  $Q(a, s) \leftarrow Q(a, s) + \alpha\,(r + \gamma Q(a', s') - Q(a, s))$

Algorithm 1: Approximate Policy Evaluation

We call Algorithm 2 *approximate policy iteration* for several realizations. We start with an arbitrary policy and approximate the action-values. The value function can be derived by the *Bellman equation*. Then we improve the policy according to Section 3.2 with the approximated values.

repeat
  approximate V^π_i and Q^π_i for $i \in [n]$
  for all $s \in S$ **do**
    if $\text{vert}(C^\pi_>(s)) \neq \emptyset$ **then**
      choose $\pi'(- \mid s) \in \text{vert}(C^\pi_>(s))$
      $\pi(- \mid s) \leftarrow \pi'(- \mid s)$

Algorithm 2: Approximate Policy Iteration

5. Experiments

All experiments are made with the simulator SimRobo[1]. The robot has four sensors, forward, left, right, and back, with a range of five blocks each. There are three actions in each state, move forward, left and right. The state action space is defined by all possible sensor values and actions. We consider two environments, see Figure 2, and two reinforcement functions. For *obstacle avoidance*, $\mathbf{R}_{oa}$, the robot gets rewarded if it moves away from obstacles and it gets punished if it moves towards them. For *wall following*, $\mathbf{R}_{wf}$, the robot gets rewarded if there is a block on its right side and punished otherwise.

[1] More information on the blockworld simulator is available at `http://mathematik.uibk.ac.at/users/rl`.

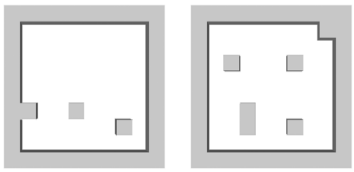

Figure 2. Environments E_1 and E_2

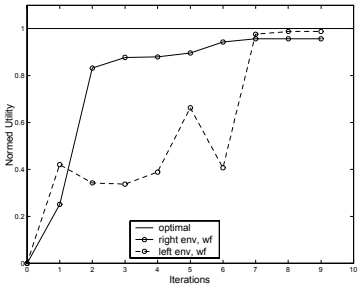

Figure 3. The performance of approximate policy iteration for wall following in two environments

In each experiment we consider two realizations and run Algorithm 2 starting with the random policy. For policy evaluation, Algorithm 1, we use discount rate $\gamma = 0.95$, start with learning rate $\alpha = 0.8$ and run 5000 iterations. We choose action a using the $\epsilon-$greedy policy derived from π. We define the *utility of a policy* by the average utilities of all states. To evaluate and compare the policies obtained after each improvement step we calculate the utilities of the policies exactly using value iteration. The utilities are then normalized, with 1 being an optimal and 0 the random policy in this realization.

5.1 Two Environments

We want to learn a policy for a wall following behavior for the environments $E_1 = (S_1, \mathbf{A}_1, \mathbf{P}_1)$ and $E_2 = (S_2, \mathbf{A}_2, \mathbf{P}_2)$. Thus we have the realizations $(E_1, \mathbf{R}_{wf})$ and $(E_2, \mathbf{R}_{wf})$. Figure 3 shows the utilities of the learned policy in each iteration step for each realization. Since the action values and value functions are only approximated the utility may decrease after a policy improvement step.

5.2 Two Environments and Two Reinforcement Functions

We look for a policy that avoids obstacles in E_1 and follows the wall in E_2. The realizations are $(E_1, \mathbf{R}_{oa})$ and $(E_2, \mathbf{R}_{wf})$. Figure 4 shows the utilities. Even tough wall following and obstacle avoidance together may be contradicting we obtain a stochastic policy that performs well in both realizations.

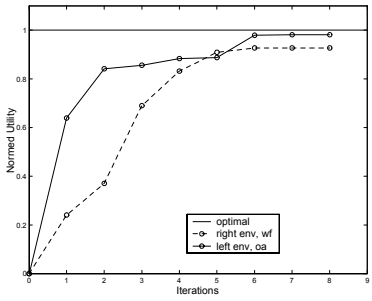

Figure 4. The performance of approximate policy iteration for wall following and obstacle avoidance in two environments

6. Discussion

Approximate policy iteration requires good approximations of all action-values in all realizations for the improvement step. Therefore the approximate policy evaluation step is critical and exploration plays a fundamental role. We note that the starting policy influences the policy learned by the algorithm. Our future research focuses on optimistic policy iteration methods, where the policy is improved after incomplete evaluation steps.

References

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming.* Belmont, MA: Athena Scientific.

Gábor, Z., Kalmár, Z., & Szepesvári, C. (1998). Multi-criteria reinforcement learning. *Proceedings of the 15th International Conference on Machine Learning (ICML 1998)* (pp. 197–205). Madison, WI: Morgan Kaufmann.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, pp. 237–285.

Matt, A., & Regensburger, G. (2001). Policy improvement for several environments. *Proceedings of the 5th European Workshop on Reinforcement Learning (EWRL-5)* (pp. 30–32). Utrecht, Netherlands.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* Cambridge, MA: MIT Press.

Wakuta, K. (1995). Vector-valued Markov decision processes and the systems of linear inequalities. *Stochastic Process. Appl.*, *56*, 159–169.

Notation

$\leftarrow$ Assignment operator, 38

$\text{aff}(X)$ Affine hull of the set $X \subset \mathbb{R}^d$, 22

$\text{cone}(X)$ Conical hull of the set $X \subset \mathbb{R}^d$, 106

$\text{conv}(X)$ Convex hull of the set $X \subset \mathbb{R}^d$, 21

$\text{vert}(P)$ Set of vertices of polytope P, 23

$(E, \mathbf{R}, \gamma)$ Discounted MDP with environment E, rewards $\mathbf{R}$ and discount rate γ, 9

$[\pi]$ Equivalence class of policy π, 13

α A step-size parameter, 38

α_t A step-size parameter at iteration t, 39

$\triangle_d$ Standard d-simplex, 23

δ_a Standard basis vector in $\mathbb{R}^{A(s)}$, 23

γ A discount rate, 5

$\lceil x \rceil$ Ceiling function for $x \in \mathbb{R}$, 28

$\lfloor x \rfloor$ Floor function for $x \in \mathbb{R}$, 23

$\|B\|_\infty$ Matrix norm of matrix B, 8

$\|x\|_\infty$ Maximum norm of vector x, 8

$\mathbb{E} = (S, \mathbf{A})$ A state action space with states S and actions $\mathbf{A}$, 99

$\mathbb{E}^\pi$ Expectation value, 5

$\mathbf{A}$ Family of actions, 2

$\mathbf{A}_S$ Set of allowed state-action pairs, 4

$\mathbf{E} = (E_i)_{i \in I}$ A family of realizations of a state action space, 100

$\mathbf{e}_i$ ith standard basis vector, 23

$\mathbf{P}$ Family of transition probabilities, 2

$\mathbf{R}$ Family of rewards, 3

$\pi(- \mid s)$ A policy in state s, that is, a probability on $A(s)$, 3

π^* An optimal policy, 32

Π Set of all policies, 3

π A policy, 2

π^{rand} Random policy, 69

$\widetilde{P}^\pi$ Transition Matrix of policy π for state-action pairs, 11

$\widetilde{T}$ Contraction mapping with fixed point Q^*, 35

$\widetilde{T}^\pi$ Contraction mapping with fixed point Q^π, 12

a An action, 2

$A(s)$ Set of available actions in state s, 2

$A^*(s)$ Set of optimal actions in state s, 33

$A^\pi_>(s)$ Set of strictly improving actions for policy π in state s, 18

$A^{\pi,Q}_>(s)$ Set of strictly improving actions for policy π and approximation Q in state s, 41

$C(s)$ Standard simplex in $\mathbb{R}^{A(s)}$, 23

$C^{\pi,\mathbf{E}}_=(s)$ Set of equivalent policies for policy π and the family of realizations $\mathbf{E}$ in state s, 102

$C^\pi_=(s)$ Set of equivalent policies for policy π in state s, 18

$C^{\pi,\mathbf{E}}_>(s)$ Set of strictly improving policies for policy π and the family of realizations $\mathbf{E}$ in state s, 102

$C^{\pi}_{>}(s)$ Set of strictly improving policies for policy π in state s, 18

$C^{\pi,\mathbf{E}}_{\geq}(s)$ Set of improving policies for policy π and the family of realizations $\mathbf{E}$ in state s, 102

$C^{\pi,Q}_{\geq}(s)$ Set of improving policies for policy π and approximation Q in state s, 41

$C^{\pi}_{\geq}(s)$ Set of improving policies for policy π in state s, 17

d Direction of the simulated robot in SimRobo, 60

$D^{\tilde{\pi},\pi}$ One-step difference between policies $\tilde{\pi}$ and π, 15

$E = (S, \mathbf{A}, \mathbf{P})$ Environment E with states S, actions $\mathbf{A}$ and transition probabilities $\mathbf{P}$, 2

H_{∞} Set of infinite histories, 5

H_T Set of histories up to time T, 4

h_T A history of states and actions up to time T, 4

I Identity Matrix, 8

$P(s' \mid a, s)$ Transition probability that action a in state s leads to successor state s', 2

$P^{\pi}(s' \mid s)$ Transition probability from state s to s' for policy π, 3

P^{π} Transition Matrix of policy π, 7

$Q^{*}(a, s)$ Optimal action-value of action a in state s, 32

$Q^{\pi}(a, s)$ Action-value of action a in state s for policy π, 11

$R(a, s)$ Expected reward of action a in state s, 3

$R(s', a, s)$ Reward if performing action a in state s leads to the successor state s', 3

$R^{\pi}(s)$ Expected reward for policy π in state s, 3

R^{γ}_{∞} Discounted return of infinite histories, 5

R_T Return of histories up to time T, 5

R_T^γ Discounted return of histories up to time T for discount rate γ, 5

S Set of states, 2

s A state, 2

$S_{\mathbf{E}}$ Set of states for the family of realizations $\mathbf{E}$, 101

T Contraction mapping with fixed point V^*, 34

T^π Contraction mapping with fixed point V^π, 9

$V(\pi)$ Discounted utility of policy π, 14

V^* Optimal value function, 32

V^π Discounted value function for policy π, 9

$V_T^{\pi,\gamma}(s)$ Discounted value function of state s for policy π up to time T, 6

$V_T^\pi(s)$ Undiscounted value function of state s for policy π up to time T, 6

xB Multiplication of row vector $x \in \mathbb{R}^S$ and matrix $B \in \mathbb{R}^{S \times S}$, 7

List of Figures

List of Algorithms

Listings

Bibliography

[Bel54] Richard Bellman, *The theory of dynamic programming*, Bull. Amer. Math. Soc. **60** (1954), 503–515. MR 16,732c 2

[Bel57] _____, *Dynamic programming*, Princeton University Press, Princeton, N. J., 1957. MR 19,820d 2, 30, 32, 85

[Bel84] _____, *Eye of the hurricane*, World Scientific Publishing Co., Singapore, 1984. MR 87i:01045 iii

[Bel03] _____, *Dynamic programming*, Dover Publications Inc., Mineola, NY, 2003, Reprint of the sixth (1972) edition. With an introduction by Eric V. Denardo. MR 1 975 026 2

[Ben03] Diego Bendersky, *Aprendizaje por refuerzo en robots autnomos para el problema de seguimiento de objetivos mviles y su aplicacin en formaciones*, Master's thesis, University of Buenos Aires, 2003. 85, 140, 141, 142, 144

[Ber95] Dimitri P. Bertsekas, *Dynamic programming and optimal control*, vol. II, Athena Scientific, Belmont, MA, 1995. 35

[Bor01] Karl Heinz Borgwardt, *Optimierung Operations Research Spieltheorie*, Birkhäuser Verlag, Basel, 2001. MR 2002b:90001 20

[Bre00] David Bremner, *Polytopebase*, URL: `http://www.cs.unb.ca/profs/bremner/PolytopeBase/`, 2000. 111

[Bro86] Rodney A. Brooks, *A robust layered control system for a mobile robot*, IEEE Journal of Robotics and Automation **2** (1986), no. 1, 14–23. 141

[BS03] Diego Ariel Bendersky and Juan Miguel Santos, *Robot formations as an emergent collective task using target-following behavior*, IberoAmerican Journal of Artificial Intelligence **21** (2003), 9–18. 140

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis, *Parallel and distributed computation: numerical methods*, Prentice-Hall, Inc., 1989, available online URL: `http://hdl.handle.net/1721.1/3719`. 9, 10

[BT96] ———, *Neuro-dynamic programming*, Athena Scientific, Belmont, MA, 1996. 35, 36, 38, 39, 40, 41, 42, 43, 44, 45, 84, 85, 95, 143

[Cas03] Anthony R. Cassandra, *Partially observable markov decision processes*, URL: `http://www.cassandra.org/pomdp`, 2003. 65

[Chv83] Vašek Chvátal, *Linear programming*, A Series of Books in the Mathematical Sciences, W. H. Freeman and Company, New York, 1983. MR 86g:90062 20, 22, 23, 114

[Dan51] George B. Dantzig, *Maximization of a linear function of variables subject to linear inequalities*, Activity Analysis of Production and Allocation, Cowles Commission Monograph No. 13, John Wiley & Sons Inc., New York, N. Y., 1951, pp. 339–347. MR 15,47k 114

[Den03] Eric V. Denardo, *Dynamic programming*, Dover Publications Inc., Mineola, NY, 2003, Corrected reprint of the 1982 original. MR 1 987 068 30, 44

[DH91] Peter Deuflhard and Andreas Hohmann, *Numerische Mathematik*, de Gruyter Lehrbuch. [de Gruyter Textbook], Walter de Gruyter & Co., Berlin, 1991, Eine algorithmisch orientierte Einführung. [An algorithmically oriented introduction]. MR 94f:65002 9

[DY79] E. B. Dynkin and A. A. Yushkevich, *Controlled Markov processes*, Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], vol. 235, Springer-Verlag, Berlin, 1979. MR 80k:90037 6

[Eke74] Ivar Ekeland, *La théorie des jeux et ses applications à l'èconomie mathématique*, Presses Universitaires de France, Vendôme, 1974. 44

[Fou00] Robert Fourer, *Linear programming frequently asked questions*, URL: `http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html`, 2000, Optimization Technology Center of Northwestern University and Argonne National Laboratory. 114

[FS94] Eugene A. Feinberg and Adam Shwartz, *Markov decision models with weighted discounted criteria*, Math. Oper. Res. **19** (1994), no. 1, 152–168. MR 95h:90162 118, 147

[FS02] Eugene A. Feinberg and Adam Shwartz (eds.), *Handbook of Markov decision processes*, International Series in Operations Research & Management Science, vol. 40, Kluwer Academic Publishers, Boston, MA, 2002. MR 2003a:90001 30, 43, 44

[Fuk00] Komei Fukuda, *Frequently asked questions in polyhedral computation*, URL: `http://www.ifor.math.ethz.ch/~fukuda/polyfaq/polyfaq.html`, 2000. 23, 111

[Fur80] Nagata Furukawa, *Vector-valued Markovian decision processes with countable state space*, Recent developments in Markov decision processes (London) (R. Hartley, L. C. Thomas, and D. J. White, eds.), Academic Press Inc. [Harcourt Brace Jovanovich Publishers], 1980, Institute of Mathematics and its Applications Conference Series, pp. xiv+334. MR 82k:90128 118

[FV97] Jerzy Filar and Koos Vrieze, *Competitive Markov decision processes*, Springer-Verlag, New York, 1997. MR 97g:90003 43, 44, 148

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete mathematics*, Addison-Wesley Publishing Company, Reading, MA, 1994. MR 97d:68003 28

[GKS98] Zoltán Gábor, Zsolt Kalmár, and Csaba Szepesvári, *Multi-criteria reinforcement learning*, Proceedings of the 15th International Conference on Machine Learning (ICML 1998) (Madison, WI) (Jude W. Shavlik, ed.), Morgan Kaufmann, 1998, pp. 197–205. 118

[Hin70] Karl Hinderer, *Foundations of non-stationary dynamic programming with discrete time parameter*, Lecture Notes in Operations Research and Mathematical Systems, Vol. 33, Springer-Verlag, Berlin, 1970. MR 42 #2791 43

[How65] Ronald A. Howard, *Dynamische Programmierung und Markov-Prozesse*, Deutsche Bearbeitung von Hans P. Künzi und Peter Kall, Verlag Industrielle Organisation, Zürich, 1965. MR 36 #4893 30, 44

[Hum96] Mark Humphrys, *Action Selection Methods using Reinforcement Learning*, From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (MA.), vol. 4, MIT Press/Bradford Books, 1996, pp. 135–144. 118

[JJS94] Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh, *On the convergence of stochastic iterative dynamic programming algorithms*, Neural Computation **6** (1994), 1185–1201. 43

[Kar97] Jonas Karlsson, *Learning to Solve Multiple Goals*, Ph.D. thesis, University of Rochester, Department of Computer Science, 1997, URL: `http://www.cs.rochester.edu/trs/ai-trs.html`. 118

[Kil96] Mark J. Kilgard, *The OpenGL Utility Toolkit (GLUT) programming interface, API version*, URL: `http://www.opengl.org/resources/libraries/glut/glut_downloads.html`, 1996. 68

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra, *Planning and acting in partially observable stochastic domains*, Artificial Intelligence **101** (1998), no. 1–2, 99–134. 65

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore, *Reinforcement learning: A survey*, Journal of Artificial Intelligence Research **4** (1996), 237–285. 36, 44

[Koh95] Teuvo Kohonen, *Self-organizing maps*, Springer Series in Information Sciences, vol. 30, Springer-Verlag, Berlin, 1995. MR 97c:68134 86

[KT99] K-Team, *Khepera user manual*, 1999, URL: `http://www.k-team.com/download/khepera.html`. 90

[Lan93] Serge Lang, *Real and functional analysis*, Graduate Texts in Mathematics, vol. 142, Springer-Verlag, New York, 1993. MR 94b:00005 106

[Mat00] Andreas Matt, *Time variable reinforcement learning and reinforcement function design*, Master's thesis, Universität Innsbruck, 2000, URL: `http://mathematik.uibk.ac.at/users/rl`. 6, 62

[McM70] Peter McMullen, *The maximum numbers of faces of a convex polytope*, Mathematika **17** (1970), 179–184. MR 44 #921 22

[MFI94] Francesco Mondada, Edoardo Franzi, and Paolo Ienne, *Mobile robot miniaturisation: A tool for investigation in control algorithms*, Experimental Robotics III (Kyoto), Springer-Verlag, 1994, pp. 501–513. 80

[MG02] Ehrgott Matthias and Xavier Gandibleux (eds.), *Multiple criteria optimization: State of the art annotated bibliographic surveys*, International Series in Operations Research and Management Science, vol. 58, Kluwer Academic Publishers, Boston, 2002. 118

[MKKC99] Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony R. Cassandra, *Solving POMDPs by searching the space of finite policies*, UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30-August 1, 1999 (Kathryn B. Laskey and Henri Prade, eds.), Morgan Kaufmann, 1999, pp. 417–426. 65

[MM00] Carlos Mariano and Eduardo Morales, *A New Approach for the Solution of Multiple Objective Optimization Problems Based on Reinforcement Learning*, MICAI'2000) (Acapulco, México) (Osvaldo Cairo, L. Enrique Sucar, and Francisco J. Cantu, eds.), Springer-Verlag, 2000, pp. 212–223. 118

[MR01a] Andreas Matt and Georg Regensburger, *Policy improvement for several environments*, Proceedings of the 5th European Workshop on Reinforcement Learning (EWRL-5) (Utrecht, Netherlands) (Marco A. Wiering, ed.), 2001, pp. 30–32. 115, 126, 183, 185

[MR01b] ———, *Policy improvement for several environments - extended version*, 2001, URL: `http://mathematik.uibk.ac.at/users/rl`. 115, 183, 185

[MR02] ———, *Generalization over environments in reinforcement learning*, Proceedings of the 4th Argentine Symposium on Artificial Intelligence (ASAI 2002) (Santa Fe, Argentina) (Juan Miguel Santos and Adriana Zapico, eds.), 2002, pp. 100–109. 119, 183, 185

[MR03a] ———, *Approximate policy iteration for several environments and reinforcement functions*, Proceedings of the 6th European Workshop on Reinforcement Learning (EWRL-6) (Nancy, France) (Alain Dutech and Olivier Buffet, eds.), 2003, pp. 15–17. 116, 126, 183, 185

[MR03b] ———, *Generalization over environments in reinforcement learning*, IberoAmerican Journal of Artificial Intelligence **21** (2003), 47–53. xi, 119, 183, 185

[Mun03] Rémis Munos, *Error bounds for approximate policy iteration*, Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA (Tom Fawcett and Nina Mishra, eds.), AAAI Press, 2003, pp. 560–567. 42

[Nas51] John Nash, *Non-cooperative games*, Ann. of Math. (2) **54** (1951), 286–295. MR 13,261g 103

[Nov89] Jiří Novák, *Linear programming in vector criterion Markov and semi-Markov decision processes*, Optimization **20** (1989), no. 5, 651–670. MR 90i:90113 118

[Owe95] Guillermo Owen, *Game theory*, Academic Press Inc., San Diego, CA, 1995. MR 96i:90003 103

[Pau02] Franz Pauer, *Lineare Optimierung*, 2002, Institut für Mathematik, Unviversität Innsbruck, lecture notes. 20

[PP02] Theodore J. Perkins and Doina Precup, *A convergent form of approximate policy iteration*, 2002, NIPS 2002, URL: `http://www-2.cs.cmu.edu/Groups/NIPS/NIPS2002/NIPS2002preproceedings/papers/CN11.html`. 42

[Put94] Martin L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*, Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics,

John Wiley & Sons Inc., New York, 1994. MR 95b:90152 ix, 9, 32, 34, 35, 43, 147

[Rad99] Paul Rademacher, *GLUI a GLUT-based User Interface Library*, URL: `http://gd.tuwien.ac.at/hci/glui`, 1999, maintained by Nigel Stewart. 69

[RM51] Herbert Robbins and Sutton Monro, *A stochastic approximation method*, Ann. Math. Statistics **22** (1951), 400–407. MR 13,144j 38

[Roc70] R. Tyrrell Rockafellar, *Convex analysis*, Princeton Mathematical Series, No. 28, Princeton University Press, Princeton, N.J., 1970. MR 43 #445 105, 106, 107

[San99] Juan Miguel Santos, *Contribution to the study and the design of reinforcement functions*, Ph.D. thesis, Universidad de Buenos Aires and Universit d'Aix-Marseille III, 1999. 44, 82, 85, 92

[SB98] Richard S. Sutton and Andrew G. Barto, *Reinforcement learning: An introduction*, MIT Press, Cambridge, MA, 1998, available online URL: `http://www-anw.cs.umass.edu/~rich/book/the-book.html`. ix, 7, 9, 10, 17, 31, 32, 35, 37, 40, 43, 44, 45, 95

[SB03] Nathan Sprague and Dana Ballard, *Multiple-goal reinforcement learning with modular Sarsa(0)*, International Joint Conference on Artificial Intelligence (Acapulco), August 2003. 118

[SC98] Satinder Singh and David Cohn, *How to Dynamically Merge Markov Decision Processes*, Advances in Neural Information Processing Systems 10, [NIPS Conference, Denver, Colorado, USA, 1997] (Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, eds.), The MIT Press, 1998. 118

[Sch86] Alexander Schrijver, *Theory of linear and integer programming*, Wiley-Interscience Series in Discrete Mathematics, John Wiley & Sons Ltd., Chichester, 1986, A Wiley-Interscience Publication. MR 88m:90090 20, 21, 107, 114

[Sch03] ———, *A course in combinatorial optimization*, URL: `http://homepages.cwi.nl/~lex/`, 2003, lecture notes. 20, 22

[She01] Christian R. Shelton, *Balancing multiple sources of reward in reinforcement learning*, Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA (Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, eds.), MIT Press, 2001, pp. 1082–1088. 118

[SJLS00] Singh Satinder, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári, *Convergence results for single-step on-policy reinforcement-learning algorithms*, Machine Learning **39** (2000), 287–308. 85

[Slo03] N. J. A. Sloane, *The on-line encyclopedia of integer sequences*, Notices Amer. Math. Soc. **50** (2003), no. 8, 912–915. MR 1 992 789 29

[Slo04] ———, *The on-line encyclopedia of integer sequences*, URL: `http://www.research.att.com/~njas/sequences/`, 2004. 29

[SMT01] Juan Miguel Santos, Andreas Matt, and Claude Touzet, *Tuning vector parametrized reinforcement functions*, Proceedings of the 5th European Workshop on Reinforcement Learning (EWRL-5) (Utrecht, Netherlands) (Marco A. Wiering, ed.), 2001, pp. 42–43. 62

[Str97] Bjarne Stroustrup, *The C++ Programming Language*, Third ed., Addision-Wesley Publishing Company, 1997. 66, 68

[Sut88] Richard S. Sutton, *Learning to predict by the methods of temporal differences*, Machine Learning **3** (1988), 9–44. 37, 39

[Sut03] Richard Sutton, *Reinforcement learning FAQ*, 2003, URL: `http://www.cs.ualberta.ca/~sutton/RL-FAQ.html`. 44

[Thr92] Sebastian B. Thrun, *Efficient exploration in reinforcement learning*, Tech. Report CMU-CS-92-102, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992. 84

[Tsi94] John N. Tsitsiklis, *Asynchronous stochastic approximation and Q-learning*, Machine Learning **16** (1994), no. 3, 185–202. 43

[Tsi03] ———, *On the convergence of optimistic policy iteration*, J. Mach. Learn. Res. **3** (2003), no. 1, 59–72. MR 2003m:90151 148

[vNM47] John von Neumann and Oskar Morgenstern, *Theory of Games and Economic Behavior*, 2d ed., Princeton University Press, Princeton, N. J., 1947. MR 9,50f 44

[Wak95] Kazuyoshi Wakuta, *Vector-valued Markov decision processes and the systems of linear inequalities*, Stochastic Process. Appl. **56** (1995), no. 1, 159–169. MR 95m:90148 118

[Wat89] Christopher J.C.H. Watkins, *Learning from delayed rewards*, Ph.D. thesis, Cambridge University, Cambridge, England, 1989. 10, 42

[WD92] Christopher J.C.H. Watkins and Peter Dayan, *Q-Learning*, Machine Learning **8** (1992), 279–292. 42

[Web94] Roger Webster, *Convexity*, Oxford Science Publications, The Clarendon Press Oxford University Press, New York, 1994. MR 98h:52001 105, 106

[Wei03] Alex Weissensteiner, *Besonderheiten einer lebenszyklusorientierten Vermgensverwaltung im Gegensatz zur klassischen Portfoliotheorie*, Ph.D. thesis, University of Innsbruck, 2003. 85

[Whi93] Douglas John White, *Markov decision processes*, John Wiley & Sons Ltd., Chichester, 1993. MR 95j:90003 44, 117

[Whi98] _____, *Vector maxima for infinite-horizon stationary Markov decision processes*, IMA J. Math. Appl. Bus. Indust. **9** (1998), no. 1, 1–17. MR 99a:90233 118

[WK03] William A. Wood and William L. Kleb, *Exploring XP for scientific research*, Software, IEEE **20** (2003), no. 3, 30–36. 65

[WKCJ00] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries, *Strengthening the case for pair programming*, Software, IEEE **17** (2000), no. 4, 19–25. 65

[Zie98] Günter M. Ziegler, *Lectures on polytopes*, revised first edition ed., Graduate Texts in Mathematics, Springer-Verlag, New York, 1998. MR 96a:52011 20, 21, 23, 107

Index