Reinforcement Learning and Markov Decision Processes

Martijn van Otterlo and Marco Wiering

Abstract Situated in between supervised learning and unsupervised learning, the paradigm of reinforcement learning deals with learning in sequential decision making problems in which there is limited feedback. This text introduces the intuitions and concepts behind Markov decision processes and two classes of algorithms for computing optimal behaviors: reinforcement learning and dynamic programming. First the formal framework of Markov decision process is defined, accompanied by the definition of value functions and policies. The main part of this text deals with introducing foundational classes of algorithms for learning optimal behaviors, based on various definitions of optimality with respect to the goal of learning sequential decisions. Additionally, it surveys efficient extensions of the foundational algorithms, differing mainly in the way feedback given by the environment is used to speed up learning, and in the way they concentrate on relevant parts of the problem. For both model-based and model-free settings these efficient extensions have shown useful in scaling up to larger problems.

This text was assembled from the initial chapters of *The Logic of Adaptive Behavior* by Martijn van Otterlo (2008), later published at IOS Press (2009). The purpose of this chapter is to show i) which kinds of algorithms, concepts, techniques etc. one can assume when writing chapters on specific topics (i.e. concepts such as MDPs and Q-learning can be assumed, and do not have to be explained again), and ii) the type of notation that is expected (e.g. the AI-style of RL, as opposed to the Bertsekas and Tsitsiklis (1996) OR-style. The specifics and length of this chapter are – of course – subject to change.

Martijn van Otterlo
Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium, e-mail: martijn.vanotterlo@cs.kuleuven.be

Marco Wiering
Department of Artificial Intelligence, University of Groningen, The Netherlands e-mail: m.a.wiering@rug.nl

1 Introduction

Markov Decision Processes (MDP) [Puterman(1994)] are an intuitive and fundamental formalism for *decision-theoretic planning* (DTP) [Boutilier et al(1999)Boutilier, Dean, and Hanks, Boutilier(1999)], reinforcement learning (RL) [Bertsekas and Tsitsiklis(1996), Sutton and Barto(1998), Kaelbling et al(1996)Kaelbling, Littman, and Moore] and other learning problems in stochastic domains. In this model, an environment is modelled as a set of states and actions can be performed to control the system's state. The goal is to control the system in such a way that some performance criterium is maximized. Many problems such as (stochastic) planning problems, learning robot control and game playing problems have successfully been modelled in terms of an MDP. In fact MDPs have become the *de facto* standard formalism for learning sequential decision making.

DTP [Boutilier et al(1999)Boutilier, Dean, and Hanks], e.g. planning using decision-theoretic notions to represent uncertainty and plan quality, is an important extension of the AI *planning paradigm*, adding the ability to deal with *uncertainty* in action effects and the ability to deal with less-defined *goals*. Furthermore it adds a significant dimension in that it considers situations in which factors such as resource consumption and uncertainty demand solutions of *varying quality*, for example in *real-time* decision situations. There are many connections between AI planning, research done in the field of *operations research* [Winston(1991)] and *control theory* [Bertsekas(1995)], as most work in these fields on *sequential decision making* can be viewed as instances of MDPs. The notion of a *plan* in AI planning, i.e. a series of actions from a start state to a goal state, is extended to the notion of a *policy*, which is mapping from *all* states to an (optimal) action, based on decision-theoretic measures of *optimality* with respect to some goal to be optimized.

As an example, consider a typical planning domain, involving boxes to be moved around and where the goal is to move some particular boxes to a designated area. This type of problems can be solved using AI planning techniques. Consider now a slightly more realistic extension in which some of the actions can fail, or have uncertain side-effects that can depend on factors beyond the operator's control, and where the goal is specified by giving credit for how many boxes are put on the right place. In this type of environment, the notion of a plan is less suitable, because a sequence of actions can have many different outcomes, depending on the effects of the operators used in the plan. Instead, the methods in this chapter are concerned about *policies* that map states onto actions in such a way that the *expected* outcome of the operators will have the intended effects. The expectation over actions is based on a decision-theoretic expectation with respect to their probabilistic outcomes and credits associated with the problem goals. The MDP framework allows for online solutions that *learn* optimal policies gradually through *simulated trials*, and additionally, it allows for *approximated* solutions with respect to resources such as computation time. Finally, the model allows for numeric, decision-theoretic measurement of the *quality* of policies and learning *performance*. For example, policies

environment	You are in state 65. You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 7 units. You are now in state 15.
	You have 2 possible actions.
agent	I'll take action 1.
environment	You have received a reward of -4 units. You are now in state 65.
	You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 5 units. You are now in state 44.
	You have 5 possible actions.
...	...

Fig. 1 Example of interaction between an agent and its environment, from a RL perspective.

can be ordered by how much credit they receive, or by how much computation is needed for a particular performance.

This chapter will cover the broad spectrum of methods that have been developed in the literature to compute good or optimal policies for problems modelled as an MDP. The term RL is associated with the more difficult setting in which no (prior) knowledge about the MDP is presented. The task then of the algorithm is to *interact*, or *experiment* with the environment (i.e. the MDP), in order to gain knowledge about how to optimize its behavior, being guided by the evaluative feedback (rewards). The model-based setting, in which the full transition dynamics and reward distributions are known, is usually characterized by the use of *dynamic programming* (DP) techniques. However, we will see that the underlying basis is very similar, and that mixed forms occur.

2 Learning Sequential Decision Making

RL is a general class of algorithms in the field of machine learning that aims at allowing an *agent* to learn how to behave in an environment, where the only feedback consists of a *scalar* reward signal. RL should not be seen as characterized by a particular class of learning methods, but rather as a learning *problem* or a *paradigm*. The *goal* of the agent is to perform *actions* that maximize the reward signal in the long run.

The distinction between the *agent* and the *environment* might not always be the most intuitive one. We will draw a boundary based on *control* [Sutton and Barto(1998)]. Everything the agent cannot control is considered part of the environment. For example, although the motors of a robot agent might be considered part of the agent, the exact functioning of them in the environment is beyond the agent's control. It can give commands to gear up or down, but their physical realization can be influenced by many things.

An example of interaction with the environment is given in Figure 1. It shows how the interaction between an *agent* and the *environment* can take place. The agent

can choose an action in each state, and the *perceptions* the agent gets from the environment are the environment's state after each action plus the scalar reward signal at each step. Here a discrete model is used in which there are distinct numbers for each state and action. The way the interaction is depicted is highly general in the sense that one just talks about states and actions as discrete *symbols*. In the rest of this book we will be more concerned about interactions in which states and actions have more *structure*, such that a state can be something like *there are two blue boxes and one white one and you are standing next to a blue box*. However, this figure clearly shows the *mechanism* of sequential decision making.

There are several important aspects in learning sequential decision making which we will describe in this section, after which we will describe formalizations in the next sections.

Approaching Sequential Decision Making.

There are several classes of algorithms that deal with the problem of sequential decision making. In this book we deal specifically with the topic of *learning*, but some other options exist.

The first solution is the *programming* solution. An intelligent system for sequential decision making can – in principle – be *programmed* to handle all situations. For each possible state an appropriate or optimal action can be specified *a priori*. However, this puts a heavy burden on the designer or programmer of the system. All situations should be foreseen in the design phase and programmed into the agent. This is a tedious and almost impossible task for most interesting problems, and it only works for problems which can be modelled completely. In most realistic problems this is not possible due to the sheer *size* of the problem, or the intrinsic *uncertainty* in the system. A simple example is *robot control* in which factors such as lighting or temperature can have a large, and unforeseen, influence on the behavior of camera and motor systems. Furthermore, in situations where the problem changes, for example due to new elements in the description of the problem or changing dynamic of the system, a programmed solution will no longer work. Programmed solutions are *brittle* in that they will only work for completely known, static problems with fixed probability distributions.

A second solution uses *search* and *planning* for sequential decision making. The successful chess program *Deep Blue* [Schaeffer and Plaat(1997)] was able to defeat the human world champion Gary Kasparov by smart, brute force search algorithms that used a model of the dynamics of chess, tuned to Kasparov's style of playing. When the dynamics of the system are known, one can *search* or *plan* from the current state to a desirable goal state. However, when there is uncertainty about the action outcomes standard search and planning algorithms do not apply. *Admissible heuristics* can solve some problems concerning the reward-based nature of sequential decision making, but the probabilistic effects of actions pose a difficult problem. Probabilistic planning algorithms exist, e.g. [**?**], but their performance is not as good as their deterministic counterparts. An additional problem is that planning and

search focus on specific start and goal states. In contrast, we are looking for *policies* which are defined for all states, and are defined with respect to *rewards*.

The third solution is *learning*, and this will be the main topic of this book. Learning has several advantages in sequential decision making. First, it relieves the designer of the system from the difficult task of deciding upon everything in the design phase. Second, it can cope with uncertainty, goals specified in terms of reward measures, and with changing situations. Third, it is aimed at solving the problem for every state, as opposed to a mere plan from one state to another. Additionally, although a model of the environment can be used or learned, it is not necessary in order to compute optimal policies, such as is exemplified by RL methods. Everything can be learned from interaction with the environment.

Online versus Off-line Learning.

One important aspect in the learning task we consider in this book is the distinction between *online* and *off-line* learning. The difference between these two types is influenced by factors such as whether one wants to control a *real-world* entity – such as a robot player robot soccer or a machine in a factory – or whether all necessary information is available. Online learning performs learning directly on the problem instance. Off-line learning uses a *simulator* of the environment as a cheap way to get many training examples for *safe* and *fast* learning.

Learning the controller directly on the real task is often not possible. For example, the learning algorithms in this chapter sometimes need millions of training instances which can be too time-consuming to collect. Instead, a simulator is much faster, and in addition it can be used to provide arbitrary training situations, including situations that rarely happen in the real system. Furthermore, it provides a "safe" training situation in which the agent can explore and make mistakes. Obtaining negative feedback in the real task in order to learn to avoid these situations, might entail destroying the machine that is controlled, which is unacceptable. Often one uses a simulation to obtain a reasonable policy for a given problem, after which some parts of the behavior are *fine-tuned* on the real task. For example, a simulation might provide the means for learning a reasonable robot controller, but some *physical* factors concerning variance in motor and perception systems of the robot might make additional fine-tuning necessary. A simulation is just a *model* of the real problem, such that small differences between the two are natural, and learning might make up for that difference. Many problems in the literature however, *are* simulations of games and optimization problems, such that the distinction disappears.

Credit Assignment.

An important aspect of sequential decision making is the fact that deciding whether an action is "good" or "bad" cannot be decided upon right away. The appropriateness of actions is completely determined by the *goal* the agent is trying to pursue.

The real problem is that the effect of actions with respect to the goal can be much *delayed*. For example, the opening moves in chess have a large influence on winning the game. However, between the first opening moves and receiving a reward for winning the game, a couple of tens of moves might have been played. Deciding how to give *credit* to the first moves – which did not get the immediate reward for winning – is a difficult problem called the *temporal credit assignment* problem. Each move in a winning chess game contributes more or less to the success of the last move, although some moves along this path can be less optimal or even bad. A related problem is the *structural credit assignment* problem, in which the problem is to distribute feedback over the *structure* representing the agent's policy. For example, the policy can be represented by a structure containing parameters (e.g. a neural network). Deciding which parameters have to be updated forms the structural credit assignment problem.

The Exploration-Exploitation Trade-off.

If we know a complete model of dynamics of the problem, there exist methods (e.g. DP) that can compute optimal policies from this model. However, in the more general case where we do not have access to this knowledge (e.g. RL), it becomes necessary to *interact* with the environment to learn by *trial-and-error* a correct policy. The agent has to *explore* the environment by performing actions and perceiving their consequences (i.e. the effects on the environments and the obtained rewards). The only feedback the agent gets are rewards, but it does not get information about what is the right action. At some point in time, it will have a policy with a particular performance. In order to see whether there are possible improvements to this policy, it sometimes has to *try out* various actions to see their results. This might result in worse performance because the actions might also be less good than the current policy. However, without trying them, it might never find possible improvements. In addition, if the world is not stationary, the agent has to explore to keep its policy up-to-date. So, in order to *learn* it has to *explore*, but in order to *perform well* it should *exploit* what it already knows. Balancing these two things is called the *exploration-exploitation problem*.

Feedback, Goals and Performance.

Compared to supervised learning, the amount of feedback the learning system gets in RL, is much less. In supervised learning, for every learning sample the correct output is given in a training set. The performance of the learning system can be measured relative to the number of correct answers, resulting in a *predictive accuracy*. The difficulty lies in learning this mapping, and whether this mapping *generalizes* to new, unclassified, examples. In unsupervised learning, the difficulty lies in constructing a useful partitioning of the data such that classes naturally arise. In reinforcement there is only some information available about performance, in the

form of one *scalar* signal. This feedback system is *evaluative* rather than being *instructive*. Using this limited signal for feedback renders a need to put more effort in using it to evaluate and improve behavior during learning.

A second aspect about feedback and performance is related to the stochastic nature of the problem formulation. In supervised and unsupervised learning, the data is usually considered *static*, i.e. a data set is given and performance can be measured with respect to this data. The learning samples for the learner originate from a fixed distribution, i.e. the data set. From a RL perspective, the data can be seen as a *moving target*. The learning process is driven by the current policy, but this policy will change over time. That means that the *distribution* over states and rewards will change because of this. In machine learning the problem of a changing distribution of learning samples is termed *concept drift* [Maloof(2003)] and it demands special features to deal with it. In RL this problem is dealt with by exploration, a constant interaction between evaluation and improvement of policies and additionally the use of *learning rate adaption schemes*.

A third aspect of feedback is the question *"where do the numbers come from?"*. In many sequential decision tasks, suitable reward functions present themselves quite naturally. For games in which there are winning, losing and draw situations, the reward function is easy to specify. In some situations special care has to be taken in giving rewards for states or actions, and also their *relative size* is important. When the agent will encounter a large negative reward before it finally gets a small positive reward, this positive reward might get *overshadowed*. All problems posed will have *some* optimal policy, but it depends on whether the reward function is in accordance with the right goals, whether the policy will tackle the *right* problem. In some problems it can be useful to provide the agent with rewards for reaching intermediate *subgoals*. This can be helpful in problems which require very long action sequences.

Representations.

One of the most important aspects in learning sequential decision making is *representation*. Two central issues are *what* should be represented, and *how* things should be represented. The first issue is dealt with in this chapter. Key components that can or should be represented are models of the dynamics of the environment, reward distributions, value functions and policies. For some algorithms all components are explicitly stored in tables, for example in classic DP algorithms. Actor-critic methods keep separate, explicit representations of both value functions and policies. However, in most RL algorithms just a value function is represented whereas policy decisions are derived from this value function online. Methods that search in policy space do not represent value functions explicitly, but instead an explicitly represented policy is used to compute values when necessary. Overall, the choice for *not* representing certain elements can influence the choice for a type of algorithm, and its efficiency.

The question of *how* various structures can be represented is dealt with extensively in this book, starting from the next chapter. Structures such as policies, transition functions and value functions can be represented in more compact form by using various *structured* knowledge representation formalisms and this enables much more efficient solution mechanisms and scaling up to larger domains.

3 A Formal Framework

The elements of the RL problem as described in the introduction to this chapter can be formalized using the *Markov decision process* (MDP) framework. In this section we will formally describe components such as *states* and *actions* and *policies*, as well as the *goals* of learning using different kinds of *optimality criteria*. MDPs are extensively described in [Puterman(1994)] and [Boutilier et al(1999)Boutilier, Dean, and Hanks]. They can be seen as stochastic extensions of finite automata and also as *Markov processes* augmented with actions.

Although general MDPs may have infinite (even uncountable) state and action spaces, we limit the discussion to finite-state and finite-action problems. In the next chapter we will encounter continuous spaces and in later chapters we will encounter situations arising in the first-order logic setting in which infinite spaces can quite naturally occur.

3.1 Markov Decision Processes.

MDPs consist of states, actions, transitions between states and a reward function definition. We consider each of them in turn.

States.

The set of environmental states S is defined as the finite set $\{s^1,\ldots,s^N\}$ where the *size* of the state space is N, i.e. $|S| = N$. A *state* is a unique characterization of all that is important in a state of the problem that is modeled. For example, in chess a complete configuration of board pieces of both black and white, is a state. In the next chapter we will encounter the use of *features* that *describe* the state. In those contexts, it becomes necessary to distinguish between *legal* and *illegal* states, for some combinations of features might not result in an actually existing state in the problem. In this chapter, we will confine ourselves to the discrete state set S in which each state is represented by a *distinct symbol*, and all states $s \in S$ are legal.

Actions.

The set of actions A is defined as the finite set $\{a^1,\ldots,a^K\}$ where the *size* of the action space is K, i.e. $|A| = K$. Actions can be used to *control* the system state. The set of actions that can be applied in some particular state $s \in S$, is denoted $A(s)$, where $A(s) \subseteq A$. In some systems, not all actions can be applied in every state, but in general we will assume that $A(s) = A$ for all $s \in S$. In more structured representations (e.g. by means of *features*), the fact that some actions are not applicable in some states, is modeled by a *precondition function* $\text{pre} : S \times A \to \{\texttt{true},\texttt{false}\}$, stating whether action $a \in A$ is applicable in state $s \in S$.

The Transition Function.

By applying action $a \in A$ in a state $s \in S$, the system makes a *transition* from s to a new state $s' \in S$, based on a probability distribution over the set of possible transitions. The transition function T is defined as $T : S \times A \times S \to [0,1]$, i.e. the probability of ending up in state s' after doing action a in state s is denoted $T(s,a,s')$. It is required that for all actions a, and all states s and s', $T(s,a,s') \geq 0$ and $T(s,a,s') \leq 1$. Furthermore, for all states s and actions a, $\sum_{s' \in S} T(s,a,s') = 1$, i.e. T defines a *proper probability distribution over possible next states*. Instead of a precondition function, it is also possible to set[1] $T(s,a,s') = 0$ for all states $s' \in S$ if a is not applicable in s. For talking about the *order* in which actions occur, we will define a discrete *global clock*, $t = 1,2,\ldots$. Using this, the notation s_t denotes the state at time t and s_{t+1} denotes the state at time $t+1$. This enables to compare different states (and actions) occurring ordered in time during interaction.

The system being controlled is *Markovian* if the result of an action does not depend on the previous actions and visited states (history), but only depends on the current state, i.e.

$$P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots) \;=\; P(s_{t+1} \mid s_t, a_t) \;=\; T(s_t, a_t, s_{t+1})$$

The idea of Markovian dynamics is that the current state s gives enough information to make an optimal decision; if is not important which states and actions preceded s. Another way of saying this, is that if you select an action a, the probability distribution over next states is the same as the last time you tried this action in the same state. More general models can be characterized by being *k-Markov*, i.e. the last k are states sufficient, such that *Markov* is actually 1-*Markov*. Though, each *k-Markov* problem can be transformed into an equivalent *Markov* problem. The *Markov property* forms a boundary between the MDP and more general models such as POMDPs.

[1] Although this is the same, the explicit distinction between an action not begin applicable in a state and a zero probability for transitions with that action, is lost in this way.

The Reward Function.

The *reward function*[2] specifies rewards for being in a state, or doing some action in a state. The *state reward* function is defined as $R : S \rightarrow \mathbb{R}$, and it specifies the reward obtained in states. However, two other definitions exist. One can define either $R : S \times A \rightarrow \mathbb{R}$ or $R : S \times A \times S \rightarrow \mathbb{R}$. The first one gives rewards for performing an action in a state, and the second gives rewards for particular transitions between states. All definitions are interchangeable though the last one is convenient in *model-free* algorithms (see Section 7), because there we usually need both the starting state and the resulting state in backing up values. Throughout this book we will mainly use $R(s,a,s')$, but deviate from this when more convenient.

The reward function is an important part of the MDP that specifies implicitly the *goal* of learning. For example, in episodic tasks such as in the games *Tic-Tac-Toe* and chess, one can assign all states in which the agent has won a positive reward value, all states in which the agent loses a negative reward value and a zero reward value in all states where the final outcome of the game is a draw. The goal of the agent is to reach positive valued states, which means winning the game. Thus, the reward function is used to give *direction* in which way the system, i.e. the MDP, should be controlled. Often, the reward function assigns non-zero reward to non-goal states as well, which can be interpreted as defining *sub-goals* for learning.

The Markov Decision Process.

Putting all elements together results in the definition of a *Markov decision process*, which will be the base model for the large majority of methods described in this book.

Definition 3.1
*A **Markov decision process** is a tuple $\langle S,A,T,R\rangle$ in which S is a finite set of states, A a finite set of actions, T a transition function defined as $T : S \times A \times S \rightarrow [0,1]$ and R a reward function defined as $R : S \times A \times S \rightarrow \mathbb{R}$.*

The transition function T and the reward function R together define the *model* of the MDP. Often MDPs are depicted as a state transition graph for an example) where the nodes correspond to states and (directed) edges denote transitions. A typical domain that is frequently used in the MDP literature is the *maze* [Matthews(1922)], in which the reward function assigns a positive reward for reaching the exit state.

There are several distinct types of systems that can be modelled by this definition of an MDP. In *episodic tasks*, there is the notion of *episodes* of some length, where

[2] Although we talk about *rewards* here, with the usual connotation of something positive, the reward function merely gives a *scalar* feedback signal. This can be interpreted as negative (*punishment*) or positive (*reward*). The various origins of work in MDPs in the literature creates an additional confusion with the reward function. In the *operations resarch* literature, one usually speaks of a *cost function* instead and the goal of learning and optimization is to *minimize* this function.

the goal is to take the agent from a starting state to a *goal state*. An *initial state distribution* $I : S \to [0,1]$ gives for each state the probability of the system being started in that state. Starting from a state s the system progresses through a sequence of states, based on the actions performed. In episodic tasks, there is a specific subset $G \subseteq S$, denoted *goal state area* containing states (usually with some distinct reward) where the process *ends*. We can furthermore distinguish between *finite, fixed horizon* tasks in which each episode consists of a fixed number of steps, *indefinite horizon* tasks in which each episode can end but episodes can have arbitrary length, and *infinite* horizon tasks where the system does not end at all. The last type of model is usually called a *continuing task*.

Episodic tasks, i.e. in which there so-called *goal states*, can be modelled using the same model defined in Definition 3.1. This is usually modelled by means of *absorbing states* or *terminal states*, e.g. states from which every action results in a transition to that same state with probability 1 and reward 0. Formally, for an absorbing state s, it holds that $T(s,a,s) = 1$ and $R(s,a,s') = 0$ for all states $s' \in S$ and actions $a \in A$. When entering an absorbing state, the process is reset and restarts in a new starting state. Episodic tasks and absorbing states can in this way be elegantly modelled in the same framework as continuing tasks.

3.2 Policies

Given an MDP $\langle S,A,T,R\rangle$, a policy is a computable function that outputs for each state $s \in S$ an action $a \in A$ (or $a \in A(s)$). Formally, a *deterministic* policy π is a function defined as $\pi : S \to A$. It is also possible to define a *stochastic* policy as $\pi : S \times A \to [0,1]$ such that for each state $s \in S$, it holds that $\pi(s,a) \geq 0$ and $\sum_{a \in A} \pi(s,a) = 1$. We will assume deterministic policies in this book unless stated otherwise.

Application of a policy to an MDP is done in the following way. First, a start state s_0 from the initial state distribution I is generated. Then, the policy π suggest the action $a_0 = \pi(s_0)$ and this action is performed. Based on the transition function T and reward function R, a transition is made to state s_1, with probability $T(s_0,a,s_1)$ and a reward $r_0 = R(s_0,a_0,s_1)$ is received. This process continues, producing $s_0,a_0,r_0,s_1,a_1,r_1,s_2,a_2,\ldots$. If the task is episodic, the process ends in state s_{goal} and is restarted in a new state drawn from I. If the task is continuing, the sequence of states can be extended indefinitely.

The policy is part of the agent and its aim is to *control* the environment modeled as an MDP. A fixed policy induces a stationary transition distribution over the MDP which can be transformed into a *Markov system*[3] $\langle S', T'\rangle$ where $S' = S$ and $T'(s,s') = T(s,a,s')$ whenever $\pi(s) = a$.

[3] In other words, if π is fixed, the system behaves as a stochastic transition system with a stationary distribution over states.

3.3 Optimality Criteria and Discounting

In the previous sections, we have defined the environment (the MDP) and the agent (i.e. the controlling element, or policy). Before we can talk about algorithms for computing *optimal* policies, we have to define what that means. That is, we have to define what the *model of optimality* is. There are two ways of looking at optimality. First, there is the aspect of *what* is actually being optimized, i.e. what is the *goal* of the agent? Second, there is the aspect of how optimal the way in which the goal is being optimized, is. The first aspect is related to *gathering reward* and is treated in this section. The second aspect is related to the efficiency and optimality of algorithms, and this is briefly touched upon and dealt with more extensively in Section 5 and further.

The goal of learning in an MDP is to gather rewards. If the agent was only concerned about the immediate reward, a simple optimality criterion would be to optimize $E[r_t]$. However, there are several ways of taking into account the future in how to behave now. There are basically three models of optimality in the MDP, which are sufficient to cover most of the approaches in the literature. They are strongly related to the types of tasks that were defined in Section 3.1.

$$E\left[\sum_{t=0}^{h} r_t\right] \qquad E\left[\sum_{t=1}^{\infty} \gamma^t r_t\right] \qquad \lim_{h\to\infty} E\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right]$$

Fig. 2 Optimality: **a)** finite horizon, **b)** discounted, infinite horizon, **c)** average reward.

The *finite horizon* model simply takes a finite horizon of length h and states that the agent should optimize its expected reward over this horizon, i.e. the next h steps (see Figure 2**a)**). One can think of this in two ways. The agent could in the first step take the *h-step optimal action*, after this the $(h-1)$*-step optimal action*, and so on. Another way is that the agent will always take the *h-step optimal action*, which is called *receding-horizon control*. The problem, however, with this model, is that the (optimal) choice for the horizon length h is not always known.

In the *infinite-horizon model*, the long-run reward is taken into account, but the rewards that are received in the future are discounted according to how far away in time they will be received. A *discount factor* γ, with $0 \leq \gamma < 1$ is used for this (see Figure 2**b)**). Note that in this *discounted* case, rewards obtained later are discounted more than rewards obtained earlier. Additionally, the discount factor ensures that – even with infinite horizon – the sum of the rewards obtained is finite. In episodic tasks, i.e. in tasks where the horizon is finite, the discount factor is not needed or can equivalently be set to 1. If $\gamma = 0$ the agent is said to be *myopic*, which means that it is only concerned about immediate rewards. The discount factor can be interpreted in several ways; as an interest rate, probability of living another step, or the mathematical trick for bounding the infinite sum. The discounted, infinite-horizon model

is mathematically more convenient, but conceptually similar to the finite horizon model. Most algorithms in this book use this model of optimality.

A third optimality model is the *average-reward* model, maximizing the long-run *average reward* (see Figure 2**c)**). Sometimes this is called the *gain optimal* policy and in the limit, as the discount factor approaches 1, it is equal to the infinite-horizon discounted model. A difficult problem with this criterion that we cannot distinguish between two policies in which one receives a lot of reward in the initial phases and another one which does not. This initial difference in reward is hidden in the long-run average. This problem can be solved in using a *bias optimal* model in which the long-run average is still being optimized, but policies are preferred if they additionally get initially extra reward. See [Mahadevan(1996)] for a survey on average reward RL.

Choosing between these optimality criteria can be related to the learning problem. If the length of the episode is known, the finite-horizon model is best. However, often this is not known, or the task is continuing, the infinite-horizon model is more suitable. [Koenig and Liu(2002)] gives an extensive overview of different modelings of MDPs and their relationship with optimality.

The second kind of optimality in this section is related to the more general aspect of the optimality of the learning process itself. We will encounter various concepts in the remainder of this book. We will briefly summarize three important notions here.

Learning optimality can be explained in terms of *what* the end result of learning might be. A first concern is whether the agent is able to obtain *optimal performance* in principle. For some algorithms there are proofs stating this, but for some not. In other words, is there a way to ensure that the learning process will reach a global optimum, or merely a local optimum, or even an oscillation between performances? A second kind of optimality is related to the *speed* of converging to a solution. We can distinguish between two learning methods by looking at how many interactions are needed, or how much computation is needed per interaction. And related to that, what will the performance be after a certain period of time? In supervised learning the optimality criterion is often defined in terms of *predictive accuracy* which is different from optimality in the MDP setting. Also, it is important to look at how much *experimentation* is necessary, or even allowed, for reaching optimal behavior. For example, a learning robot or helicopter might not be allowed to make many mistakes during learning. A last kind of optimality is related to how much reward is *not* obtained by the learned policy, as compared to an optimal one. This is usually called the *regret* of a policy.

4 Value Functions and Bellman Equations

In the preceding sections we have defined MDPs and optimality criteria that can be useful for learning optimal policies. In this section we define *value functions*, which are a way to link the optimality criteria to policies. Most learning algorithms for

MDPs compute optimal policies by learning value functions. A value function represents an estimate *how good* it is for the agent to be in a certain state (or how good it is to perform a certain action in that state). The notion of *how good* is expressed in terms of an optimality criterion, i.e. in terms of the expected return. Value functions are defined for particular policies.

The *value of a state s under policy* π, denoted $V^\pi(s)$ is the expected return when starting in s and following π thereafter. We will use the infinite-horizon, discounted model in this section, such that this can be expressed[4] as:

$$V^\pi(s) = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s\right\} \tag{1}$$

A similar *state-action value function* $Q : S \times A \to \mathbb{R}$ can be defined as the expected return starting from state s, taking action a and thereafter following policy π:

$$Q^\pi(s,a) = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s, a_t = a\right\}$$

One fundamental property of value functions is that they satisfy certain recursive properties. For any policy π and any state s the expression in Equation 1 can recursively be defined in terms of a so-called *Bellman Equation* [Bellman(1957)]:

$$\begin{aligned} V^\pi(s) &= E_\pi\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots |s_t = t\right\} \\ &= E_\pi\left\{r_t + \gamma V^\pi(s_{t+1})|s_t = s\right\} \\ &= \sum_{s'} T(s,\pi(s),s')\Big(R(s,a,s') + \gamma V^\pi(s')\Big) \end{aligned} \tag{2}$$

It denotes that the expected value of state is defined in terms of the immediate reward and values of possible next states weighted by their transition probabilities, and additionally a discount factor. V^π is the unique solution for this set of equations. Note that multiple policies can have the same value function, but for a given policy π, V^π is unique.

The goal for any given MDP is to find a *best* policy, i.e. the policy that receives the most reward. This means maximizing the value function of Equation 1 for all states $s \in S$. An *optimal policy*, denoted π^*, is such that $V^{\pi^*}(s) \geq V^\pi(s)$ for all $s \in S$ and all policies π. It can be proven that the optimal solution $V^* = V^{\pi^*}$ satisfies the following Equation:

$$V^*(s) = \max_{a\in A}\sum_{s'\in S} T(s,a,s')\Big(R(s,a,s') + \gamma V^*(s')\Big) \tag{3}$$

[4] Note that we use E_π for the *expected value under policy* π.

This expression is called the *Bellman optimality equation*. It states that the value of a state under an optimal policy must be equal to the expected return for the best action in that state. To select an optimal action given the optimal state value function V^* the following rule can be applied:

$$\pi^*(s) = \arg\max_a \sum_{s' \in S} T(s,a,s') \Big(R(s,a,s') + \gamma V^*(s') \Big) \tag{4}$$

We call this policy the *greedy policy*, denoted $\pi_{greedy}(V)$ because it greedily selects the best action using the value function V. An analogous optimal state-action value is:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \Big(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \Big)$$

Q-functions are useful because they make the weighted summation over different alternatives (such as in Equation 4) using the transition function unnecessary. No forward-reasoning step is needed to compute an optimal action in a state. This is the reason that in model-free approaches, i.e. in case T and R are unknown, Q-functions are learned instead of V-functions. The relation between Q^* and V^* is given by

$$Q^*(s,a) = \sum_{s' \in S} T(s,a,s') \Big(R(s,a,s') + \gamma V^*(s') \Big) \tag{5}$$

$$V^*(s) = \max_a Q^*(s,a) \tag{6}$$

Now, analogously to Equation 4, optimal action selection can be simply put as:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{7}$$

That is, the best action is the action that has the highest expected utility based on possible next states resulting from taking that action. One can, analogous to the expression in Equation 4, define a greedy policy $\pi_{greedy}(Q)$ based on Q. In contrast to $\pi_{greedy}(V)$ there is no need to consult the model of the MDP; the Q-function suffices.

5 Solving Markov decision processes

Now that we have defined MDPs, policies, optimality criteria and value functions, it is time to consider the question of *how* to compute optimal policies. *Solving* a given MDP means computing an optimal policy π^*. Several dimensions exist along which algorithms have been developed for this purpose. The most important distinction is that between *model-based* and *model-free* algorithms.

Model-based algorithms exist under the general name of DP. The basic assumption in these algorithms is that a *model* of the MDP is known beforehand, and can

be used to compute value functions and policies using the Bellman equation (see Equation 3). Most methods are aimed at computing state value functions which can, in the presence of the model, be used for optimal action selection. In this chapter we will focus on *iterative* procedures for computing value functions and policies.

Model-free algorithms, under the general name of RL, do not rely on the availability of a perfect model. Instead, they rely on *interaction* with the environment, i.e. a *simulation* of the policy thereby generating *samples* of state transitions and rewards. These samples are then used to estimate state-action value functions. Because a model of the MDP is not known, the agent has to *explore* the MDP to obtain information. This naturally induces a *exploration-exploitation* trade-off which has to be balanced to obtain an optimal policy.

A very important underlying mechanism, the so-called *generalized policy iteration* (GPI) principle, present in all methods is depicted in Figure 3. This principle consists of two interaction processes. The *policy evaluation* step estimates the utility of the current policy π, that is, it computes V^π. There are several ways for computing this. In model-based algorithms, one can use the model to compute it directly or iteratively approximate it. In model-free algorithms, one can *simulate* the policy and estimate its utility from the sampled execution traces. The main purpose of this step is to gather information about the policy for computing the second step, the *policy improvement* step. In this step, the values of the actions are evaluated for every state, in order to find possible improvements, i.e. possible other actions in particular states that are better than the action the current policy proposes. This step computes an improved policy π' from the current policy π using the information in V^π. Both the evaluation and the improvement steps can be implemented in various ways, and interleaved in several distinct ways. The bottom line is that there is a policy that drives value learning, i.e. it determines the value function, but in turn there is a value function that can be used by the policy to select good actions. Note that it is also possible to have an *implicit* representation of the policy, which means that only the value function is stored, and a policy is computed on-the-fly for each state based on the value function when needed. This is common practice in model-free algorithms (see Section 7). And vice versa it is also possible to have implicit representations of value functions in the context of an explicit policy representation. Another interesting aspect is that in general a value function does not have to be perfectly accurate. In many cases it suffices that sufficient distinction is present between suboptimal and optimal actions, such that small errors in values do not have to influence policy optimality. This is also important in *approximation* and *abstraction* methods discussed in the next chapter.

Planning as a RL Problem.

The MDP formalism is a general formalism for *decision-theoretic planning*, which entails that standard (deterministic) *planning* problems can be formalized as such too. All the algorithms in this chapter can – in principle – be used for these planning problems too. In order to solve planning problems in the MDP framework we

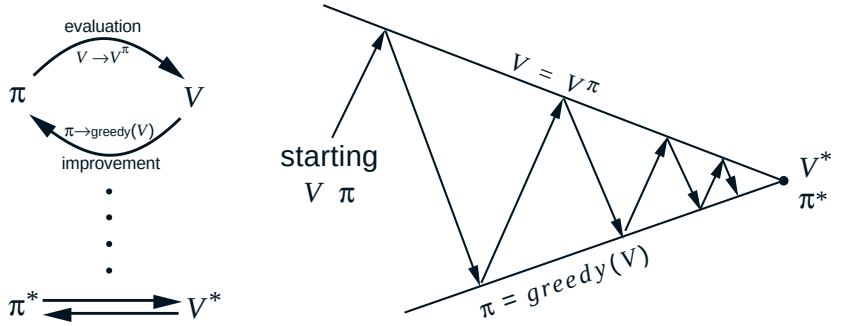

Fig. 3 a) The algorithms in Section 5 can be seen as instantiations of *Generalized Policy Iteration* (GPI) [Sutton and Barto(1998)]. The *policy evaluation* step estimates V^{π}, the policy's performance. The *policy improvement* step improves the policy π based on the estimates in V^{π}. b) The gradual convergence of both the value function and the policy to optimal versions.

have to specify *goals* and *rewards*. We can assume that the transition function T is given, accompanied by a *precondition function*. In planning we are given a *goal* function $G : S \rightarrow \{\texttt{true}, \texttt{false}\}$ that defines which states are goal states. The planning task is compute a sequence of actions $a_t, a_{t+1}, \ldots, a_{t+n}$ such that applying this sequence from a *start* state will lead to a state $s \in G$. All transitions are assumed to be deterministic, i.e. for all states $s \in S$ and actions $a \in A$ there exists only one state $s' \in S$ such that $T(s,a,s') = 1$. All states in G are assumed to be absorbing. The only thing left is to specify the reward function. We can specify this in such a way that a positive reinforcement is received once a goal state is reached, and zero otherwise:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 1, & \text{if } s_t \notin G \text{ and } s_{t+1} \in G \\ 0, & \text{otherwise} \end{cases}$$

Now, depending on whether the transition function and reward function are known to the agent, one can solve this planning task with either model-based or model-free learning. The difference with classic planning is that the learned policy will apply to all states.

6 Dynamic Programming: Model-based Solution Techniques

The term DP refers to a class of algorithms that is able to compute optimal policies in the presence of a perfect model of the environment. The assumption that a model is available will be hard to ensure for many applications. However, we will see that from a theoretical viewpoint, as well as from an algorithmic viewpoint, DP are very relevant because they define fundamental computational mechanisms which are also used when no model is available. The methods in this section all assume a standard MDP $\langle S, A, T, R \rangle$, where the state and action sets are finite and discrete such that

they can be stored in tables. Furthermore, transition, reward and value functions are assumed to store values for all states and actions separately.

6.1 *Fundamental* DP *Algorithms*

Two core DP methods are *policy iteration* [Howard(1960)] and *value iteration* [Bellman(1957)]. In the first, the GPI mechanism is clearly separated into two steps, whereas the second represents a tight integration of policy evaluation and improvement. We will consider both these algorithms in turn.

6.1.1 Policy Iteration

Policy iteration (PI) [Howard(1960)] iterates between the two phases of GPI. The *policy evaluation* phase computes the value function of the current policy and the *policy improvement* phase computes an improved policy by a maximization over the value function. This is repeated until converging to an optimal policy.

Policy Evaluation: The Prediction Problem.

A first step is to find the value function V^π of a fixed policy π. This is called the *prediction problem*. It is a part of the complete problem, that of computing an *optimal* policy. Remember from the previous sections that for all $s \in S$,

$$V^\pi(s) = \sum_{s' \in S} T(s, \pi(s), s') \Big(R(s, \pi(s), s') + \gamma V^\pi(s') \Big) \tag{8}$$

If the dynamics of the system is known, i.e. a model of the MDP is given, then these equations form a system of $|S|$ equations in $|S|$ unknowns (the values of V^π for each $s \in S$). This can be solved by linear programming (LP). However, an *iterative* procedure is possible, and in fact common in DP and RL. The Bellman equation is transformed into an *update rule* which updates the current value function V_k^π into V_{k+1}^π by 'looking one step further in the future', thereby extending the planning horizon with one step:

$$\begin{aligned} V_{k+1}^\pi(s) &= E_\pi \Big\{ r_t + \gamma V_k^\pi(s_{t+1}) | s_t = s \Big\} \\ &= \sum_{s'} T(s, \pi(s), s') \Big(R(s, \pi(s), s') + \gamma V_k^\pi(s') \Big) \end{aligned} \tag{9}$$

The sequence of approximations of V_k^π as k goes to infinity can be shown to converge. In order to converge, the update rule is applied to each state $s \in S$ in each

iteration. It replaces the old value for that state by a new one that is based on the expected value of possible successor states, intermediate rewards and weighted by the transition probabilities. This operation is called a *full backup* because it is based on all possible transitions from that state.

A more general formulation can be given by defining a *backup operator* B^π over arbitrary real-valued functions φ over the state space (e.g. a value function):

$$(B^\pi \varphi)(s) = \sum_{s' \in S} T(s, \pi(s), s') \Big(R(s, \pi(s), s') + \gamma \varphi(s') \Big) \tag{10}$$

The value function V^π of a fixed policy π satisfies the *fixed point* of this backup operator as $V^\pi = B^\pi V^\pi$. A useful special case of this backup operator is defined with respect to a fixed action a:

$$(B^a \varphi)(s) = R(s) + \gamma \sum_{s' \in S} T(s, a, s') \varphi(s')$$

Now LP for solving the *prediction problem* can be stated as follows. Computing V^π can be accomplished by solving the Bellman equations (see Equation 3) for all states. The *optimal* value function V^* can be found by using a LP problem solver that computes $V^* = \arg\max_V \sum_s V(s)$ subject to $V(s) \geq (B^a V)(s)$ for all a and s.

Policy Improvement.

Now that we know the value function V^π of a policy π as the outcome of the policy evaluation step, we can try to improve the policy. First we identify the value of all actions by using:

$$Q^\pi(s,a) = E_\pi \Big\{ r_t + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a \Big\} \tag{11}$$

$$= \sum_{s'} T(s,a,s') \Big(R(s,a,s') + \gamma V^\pi(s') \Big) \tag{12}$$

If now $Q^\pi(s,a)$ is larger than $V^\pi(s)$ for some $a \in A$ then we could do better by choosing action a instead of the current $\pi(s)$. In other words, we can *improve* the current policy by selecting a different, better, action in a particular state. In fact, we can evaluate all actions in all states and choose the best action in all states. That is, we can compute the *greedy* policy π' by selecting the best action in each state, based on the current value function V^π:

Require: $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
  {POLICY EVALUATION}
  repeat
    $\Delta := 0$
    for each $s \in S$ **do**
      $v := V^{\pi}(s)$
      $V(s) := \sum_{s'} T(s,\pi(s),s') \Big(R(s,\pi(s),s') + \gamma V(s') \Big)$
      $\Delta := \max(\Delta, |v - V(s)|)$
  until $\Delta < \sigma$
  {POLICY IMPROVEMENT}
  policy-stable := true
  for each $s \in S$ **do**
    $b := \pi(s)$
    $\pi(s) := \arg\max_a \sum_{s'} T(s,a,s') \Big(R(s,a,s') + \gamma \cdot V(s') \Big)$
    if $b \neq \pi(s)$ **then** policy-stable := false
  if policy-stable **then** stop; **else** go to POLICY EVALUATION

Algorithm 1: Policy Iteration [Howard(1960)]

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^{\pi}(s,a) \\ &= \arg\max_a E\Big\{ r_t + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = a \Big\} \\ &= \arg\max_a \sum_{s'} T(s,a,s') \Big(R(s,a,s') + \gamma V^{\pi}(s') \Big) \end{aligned} \tag{13}$$

Computing an improved policy by greedily selecting the best action with respect to the value function of the original policy is called *policy improvement*. If the policy cannot be improved in this way, it means that the policy is already optimal and its value function satisfies the Bellman equation for the optimal value function. In a similar way one can also perform these steps for stochastic policies by blending the action probabilities into the expectation operator.

Summarizing, *policy iteration* [Howard(1960)] starts with an arbitrary initialized policy π_0. Then a sequence of iterations follows in which the current policy is evaluated after which it is improved. The first step, the *policy evaluation* step computes V^{π_k}, making use of Equation 9 in an iterative way. The second step, the *policy improvement* step, computes π_{k+1} from π_k using V^{π_k}. For each state, using equation 4, the optimal action is determined. If for all states s, $\pi_{k+1}(s) = \pi_k(s)$, the policy is *stable* and the policy iteration algorithm can stop. Policy iteration generates a sequence of alternating policies and value functions

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \pi_2 \rightarrow V^{\pi_2} \rightarrow \pi_3 \rightarrow V^{\pi_3} \rightarrow \ldots \rightarrow \pi^*$$

The complete algorithm can be found in Algorithm 1.
For finite MDPs, i.e. state and action spaces are finite, policy iteration converges after a finite number of iterations. Each policy π_{k+1} is a strictly better policy than

Require: initialize V arbitrarily (e.g. $V(s) := 0, \forall s \in S$)
  repeat
    $\Delta := 0$
    for each $s \in S$ **do**
      $v := V(s)$
      for each $a \in A(s)$ **do**
        $Q(s,a) := \sum_{s'} T(s,a,s') \Big(R(s,a,s') + \gamma V(s') \Big)$
      $V(s) := \max_a Q(s,a)$
      $\Delta := \max(\Delta, |v - V(s)|)$
  until $\Delta < \sigma$

Algorithm 2: Value Iteration [Bellman(1957)]

π_k unless in case $\pi_k = \pi^*$, in which case the algorithm stops. And because for a finite MDP, the number of different policies is finite, policy iteration converges in finite time. In practice, it usually converges after a small number of iterations. Although policy iteration computes the optimal policy for a given MDP in finite time, it is relatively inefficient. In particular the first step, the policy evaluation step, is computationally expensive. Value functions for all intermediate policies $\pi_0, \ldots, \pi_k, \ldots, \pi^*$ are computed, which involves multiple sweeps through the complete state space per iteration. A bound on the number of iterations is not known [Littman et al(1995)Littman, Dean, and Kaelbling] and depends on the MDP transition structure, but it often converges after few iterations in practice.

6.1.2 Value Iteration

The policy iteration algorithm completely separates the evaluation and improvement phases. In the evaluation step, the value function must be computed in the limit. However, it is not necessary to wait for full convergence, but it is possible to stop evaluating earlier and improve the policy based on the evaluation so far. The extreme point of truncating the evaluation step is the *value iteration* [Bellman(1957)] algorithm. It breaks off evaluation after just one iteration. In fact, it immediately blends the policy improvement step into its iterations, thereby purely focusing on estimating directly the value function. Necessary updates are computed on-the-fly. In essence, it combines a truncated version of the policy evaluation step with the policy improvement step, which is essentially Equation 3 turned into one update rule:

$$V_{t+1}(s) = \max_a \sum_{s'} T(s,a,s') \Big(R(s,a,s') + \gamma V_t(s') \Big) \tag{14}$$

$$= \max_a Q_{t+1}(s,a). \tag{15}$$

Using Equations (14) and (15), the value iteration algorithm (see Figure 2) can be stated as follows: starting with a value function V_0 over all states, one iteratively

updates the value of each state according to (14) to get the next value functions V_t ($t = 1,2,3,\ldots$). It produces the following sequence of value functions:

$$V_0 \rightarrow V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \rightarrow \ldots V^*$$

Actually, in the way it is computed it also produces the intermediate Q-value functions such that the sequence is

$$V_0 \rightarrow Q_1 \rightarrow V_1 \rightarrow Q_2 \rightarrow V_2 \rightarrow Q_3 \rightarrow V_3 \rightarrow Q_4 \rightarrow V_4 \rightarrow \ldots V^*$$

Value iteration is guaranteed to converge in the limit towards V^*, i.e. the Bellman optimality Equation (3) holds for each state. A deterministic policy π for all states $s \in S$ can be computed using Equation 4. If we use the same general *backup operator* mechanism used in the previous section, we can define value iteration in the following way.

$$(B^*\varphi)(s) = \max_a \sum_{s' \in S} T(s,a,s') \Big\{ R(s,a,s') + \gamma\varphi(s) \Big\} \tag{16}$$

The backup operator B^* functions as a *contraction mapping* on the value function. If we let π^* denote the *optimal* policy and V^* its value function, we have the relationship (fixed point) $V^* = B^*V^*$ where $(B^*V)(s) = \max_a(B^aV)(s)$. If we define $Q^*(s,a) = B^aV^*$ then we have $\pi^*(s) = \pi_{greedy}(V^*)(s) = \arg\max_a Q^*(s,a)$. That is, the algorithm starts with an arbitrary value function V^0 after which it iterates $V_{t+1} = B^*V^t$ until $\|V_{t+1} - V_t\|_S < \varepsilon$, i.e. until the distance between subsequent value function approximations is small enough.

6.2 *Efficient* DP *Algorithms*

The policy iteration and value iteration algorithms can be seen as spanning a *spectrum* of DP approaches. This spectrum ranges from complete *separation* of evaluation and improvement steps to a complete *integration* of these steps. Clearly, in between the extreme points is much room for variations on algorithms. Let us first consider the computational complexity of the extreme points.

Complexity.

Value iteration works by producing successive approximations of the optimal value function. Each iteration can be performed in $O(|A||S|^2)$ steps, or faster if T is sparse. However, the *number* of iterations can grow exponentially in the discount factor cf. [Bertsekas and Tsitsiklis(1996)]. This follows from the fact that a larger γ implies that a longer sequence of future rewards has to be taken into account, hence a larger number of value iteration steps because each step only extends the

horizon taking into account in V by one step. The complexity of value iteration is linear in number of actions, and quadratic in the number of states. But, usually the transition matrix is sparse. In practice policy iteration converges much faster, but each evaluation step is expensive. Each iteration has a complexity of $O(|A||S|^2 + |S|^3)$, which can grow large quickly. A worst case bound on the number of iterations is not known [Littman et al(1995)Littman, Dean, and Kaelbling]. Linear programming is is a common tool that can be used for the evaluation too. In general, the number of iterations and value backups can quickly grow extremely large when the problem size grows. The state spaces of games such as backgammon and chess consist of too many states to perform just one full sweep. In this section we will describe some efficient variations on DP approaches. Detailed coverage of complexity results for the solution of MDPs can be found in [Littman et al(1995)Littman, Dean, and Kaelbling, Bertsekas and Tsitsiklis(1996), Boutilier et al(1999)Boutilier, Dean, and Hanks].

The efficiency of DP can be roughly improved along two lines. The first is a *tighter integration* of the evaluation and improvement steps of the GPI process. We will discuss this issue briefly in the next section. The second is that of using (*heuristic*) *search* algorithms in combination with DP algorithms. For example, using search as an exploration mechanism can highlight important parts of the state space such that value backups can be concentrated on these parts. This is the underlying mechanism used in the methods discussed briefly in Section 6.2.2

6.2.1 Styles of Updating

The full backups updates in DP algorithms can be done in several ways. We have assumed in the description of the algorithms that in each step an old and a new value function are kept in memory. Each update puts a new value in the new table, based on the information of the old. This is called *synchronous*, or *Jacobi-style* updating [Sutton and Barto(1998)]. This is useful for explanation of algorithms and theoretical proofs of convergence. However, there are two more common ways for updates. One can keep a single table and do the updating directly in there. This is called *in-place* updating [Sutton and Barto(1998)] or *Gauss-Seidel* [Bertsekas and Tsitsiklis(1996)] and usually speeds up convergence, because during one sweep of updates, some updates use already newly updated values of other states. Another type of updating is called *asynchronous updating* which is an extension of the *in-place* updates, but here updates can be performed in any order. An advantage is that the updates may be distributed unevenly throughout the state(-action) space, with more updates being given to more important parts of this space. For all these methods convergence can be proved under the general condition that values are updated infinitely often but with a finite frequency.

Modified Policy Iteration.

Modified policy iteration (MPI) [Puterman and Shin(1978)] strikes a middle ground between value and policy iteration. MPI maintains the two separate steps of GPI, but both steps are not necessarily computed in the limit. The key insight here is that for policy improvement, one does not need an *exactly* evaluated policy in order to improve it. For example, the policy estimation step can be approximative after which a policy improvement step can follow. In general, both steps can be performed quite independently *by different means*. For example, instead of iteratively applying the Bellman update rule from Equation 15, one can perform the policy estimation step by using a sampling procedure such as *Monte Carlo* estimation [Sutton and Barto(1998)]. These general forms in which mixed forms of estimation and improvements is captured by the generalized policy iteration mechanism depicted in Figure 3. Policy iteration and value iteration are both the extreme cases of modified policy iteration, whereas MPI is a general method for asynchronous updating.

6.2.2 Heuristics and Search

In many realistic problems, only a fraction of the state space is relevant to the problem of reaching the goal state from some state s. This has inspired a number of algorithms that focus computation on states that seem most relevant for finding an optimal policy from a start state s. These algorithms usually display good *anytime behavior*, i.e. they produce good or reasonable policies fast, after which they are gradually improved. In addition, they can be seen as implementing various ways of *asynchronous* DP.

Envelopes and Fringe States.

One form of *asynchronous* methods is the PLEXUS system [Dean et al(1995)Dean, Kaelbling, Kirman, and Nicholson]. It was designed for goal-based reward functions, i.e. episodic tasks in which only goal states get positive reward. It starts with an approximated version of the MDP in which not the full state space is contained. This smaller version of the MDP is called an *envelope* and it includes the agent's current state and the goal state. A special OUT state represents all the states outside the envelope. The initial envelope is constructed by a forward search until a goal state is found. The envelope can be extended by considering states outside the envelope that can be reached with high probability. The intuitive idea is to include in the envelope all the states that are likely to be reached on the way to the goal. Once the envelope has been constructed a policy is computed through policy iteration. If at any point the agent leaves the envelope, it has to replan by extending the envelope. This combination of learning and planning

still uses policy iteration, but on a much smaller (and presumably more relevant with respect to the goal) state space.

A related method proposed in [Tash and Russell(1994)] considers goal-based tasks too. However, instead of the single OUT state, they keep a *fringe* of states on the edge of the envelope and use a heuristic to estimate values of the other states. When computing a policy for the envelope, all fringe states become absorbing states with the heuristic set as their value. Over time the heuristic values of the fringe states converge to the optimal values of those states.

Similar to the previous methods the LAO* algorithm [Hansen and Zilberstein(2001)] also alternates between an expansion phase and a policy generation phase. It too keeps a fringe of states outside the envelope such that expansions can be larger than the envelope method in [Dean et al(1995)Dean, Kaelbling, Kirman, and Nicholson]. The motivation behind LAO* was to extend the classic search algorithm AO* cf. [**?**] to *cyclic* domains such as MDPs.

Search and Planning in DP.

Real-time dynamic DP (RTDP) [Barto et al(1995)Barto, Bradtke, and Singh] combines forward search with DP too. It is used as an alternative for value iteration in which only a subset of values in the state space are backed up in each iteration. RTDP performs *trials* from a randomly selected state to a goal state, by simulating the greedy policy using an *admissable heuristic* function as the initial value function. It then backups values *fully* only along these trials, such that backups are concentrated on the *relevant* parts of the state. The approach was later extended into *labeled* RTDP [Bonet and Geffner(2003b)] where some states are *labeled* as *solved* which means that their value has already converged. Furthermore, it was recently extended to *bounded* RTDP[Brendan McMahan et al(2005)Brendan McMahan, Likhachev, and Gordon] which keeps lower and upper *bounds* on the optimal value function. Other recent methods along these lines are *focussed* DP [Ferguson and Stentz(2004)] and *heuristic search-DP* [Bonet and Geffner(2003a)]

7 Reinforcement Learning: Model-Free Solution Techniques

The previous section has reviewed several methods for computing an optimal policy for an MDP assuming that a (perfect) model is available. RL is primarily concerned with how to obtain an optimal policy when such a model is not available. RL adds to MDPs a focus on approximation and incomplete information, and the need for sampling and exploration. In contrast with the algorithms discussed in the previous section, *model-free* methods do not rely on the availability of priori known transition and reward models, i.e. a *model of the MDP*. The lack of a model generates a need to *sample* the MDP to gather statistical knowledge about this unknown model. Many model-free RL techniques exist that probe the environment by doing actions,

for each episode **do**
  $s \in S$ is initialized as the *starting* state
  $t := 0$
  repeat
    choose an action $a \in A(s)$
    perform action a
    observe the new state s' and received reward r
    update $\tilde{T}$, $\tilde{R}$, $\tilde{Q}$ and/or $\tilde{V}$
    using the experience $\langle s, a, r, s' \rangle$
    $s := s'$
  until s' is a goal state

Algorithm 3: A general algorithm for online RL

thereby estimating the same kind of state value and state-action value functions as model-based techniques. This section will review model-free methods along with several efficient extensions.

In model-free contexts one has still a choice between two options. The first one is first to *learn* the transition and reward model from interaction with the environment. After that, when the model is (approximately or sufficiently) correct, all the DP methods from the previous section apply. This type of learning is called *indirect* RL. The second option, called *direct* RL, is to step right into estimating values for actions, without even estimating the model of the MDP. Additionally, mixed forms between these two exists too. For example, one can still do model-free estimation of action values, but use an approximated model to speed up value learning by using this model to perform more, and in addition, full backups of values (see Section 7.3). Most model-free methods however, focus on direct estimation of (action) values.

A second choice one has to make is what to do with the *temporal credit assignment*. It is difficult to assess the utility of some action, if the real effects of this particular action can only be perceived much later. One possibility is to wait until the ”end” (e.g. of an episode) and punish or reward specific actions along the path taken. However, this will take a lot of memory and often, with ongoing tasks, it is not known beforehand whether, or when, there will be an ”end”. Instead, one can use similar mechanisms as in *value iteration* to adjust the estimated value of a state based on the immediate reward and the estimated (discounted) value of the next state. This is generally called *temporal difference learning* which is a general mechanism underlying the model-free methods in this section. The main difference with the update rules for DP approaches (such as Equation 14) is that the transition function T and reward function R cannot appear in the update rules now. The general class of algorithms that interact with the environment and update their estimates after each experience is called *online*.

A general template for *online* RL is depicted in Figure 3. It shows an interaction loop in which the agent selects an action (by whatever means) based on its current state, gets feedback in the form of the resulting state and an associated reward, after which it updates its estimated values stored in $\tilde{V}$ and $\tilde{Q}$ and possibly statistics concerning $\tilde{T}$ and $\tilde{R}$ (in case of some form of indirect learning). The selection of the

action is based on the current state s and the value function (either Q or V). To solve the exploration-exploitation problem, usually a separate *exploration* mechanism ensures that sometimes the best action (according to current estimates of action values) is taken (exploitation) but sometimes a different action is chosen (exploration). Various choices for exploration, ranging from random to sophisticated, exist and we will see some examples in Section 7.3.

Exploration.

One important aspect of model-free algorithms is that there is a need for *exploration*. Because the model is unknown, the learner has to try out different actions to see their results. A learning algorithm has to strike a balance between *exploration* and *exploitation*, i.e. in order to gain a lot of reward the learner has to exploit its current knowledge about good actions, although it sometimes must try out different actions to explore the environment for possible better actions. The most basic exploration strategy is the ε-greedy policy, i.e. the learner takes its current best action with probability $(1-\varepsilon)$ and a (randomly selected) other action with probability ε. There are many more ways of doing exploration (see [Wiering(1999), Reynolds(2002), Ratitch(2005)] for overviews) and in Section 7.3 we will see some examples. One additional method that is often used in combination with the algorithms in this section is the *Boltzmann* (or: *softmax*) exploration strategy. It is only slightly more complicated than the ε-greedy strategy. The action selection strategy is still random, but selection probabilities are weighted by their relative Q-values. This makes it more likely for the agent to choose very good actions, whereas two actions that have similar Q-values will have almost the same probability to get selected. Its general form is

$$P(a_n) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_i e^{\frac{Q(s,a_i)}{T}}} \tag{17}$$

in which $P(a_n)$ is the probability of selecting action a_n and T is the *temperature* parameter. Higher values of T will move the selection strategy more towards a purely random strategy and lower values will move to a fully greedy strategy. A combination of both ε-greedy and Boltzmann exploration can be taken by taking the best action with probability $(1-\varepsilon)$ and otherwise an action computed according to Equation 17 [Wiering(1999)].

Another simple method to stimulate exploration is *optimistic Q-values initialization*; one can initialize all Q-values to high values – e.g. an a priori defined upper-bound – at the start of learning. Because Q-values will decrease during learning, actions that have not been tried a number of times will have a large enough value to get selected when using Boltzmann exploration for example. Another solution with a similar effect is to keep counters on the number of times a particular state-action pair has been selected.

7.1 Temporal Difference Learning

Temporal difference learning algorithms learn estimates of values based on other estimates. Each step in the world generates a *learning example* which can be used to bring some value in accordance to the immediate reward and the estimated value of the next state or state-action pair. An intuitive example, along the lines of [Sutton and Barto(1998)](Chapter 6), is the following.

Imagine you have to predict at what time your guests can arrive for a small diner in your house. Before cooking, you have to go to the supermarket, the butcher and the wine seller, in that order. You have estimates of driving times between all locations, and you predict that you can manage to visit the two last stores both in 10 minutes, but given the crowdy time on the day, your estimate about the supermarket is a half hour. Based on this prediction, you have notified your guests that they can arrive no earlier than $18.00h$. Once you have found out while in the supermarket that it will take you only 10 minutes to get all the things you need, you can adjust your estimate on arriving back home with 20 minutes less. However, once on your way from the butcher to the wine seller, you see that there is quite some traffic along the way and it takes you 30 minutes longer to get there. Finally you arrive 10 minutes later than you predicted in the first place. The bottom line of this example is that you can adjust your estimate about what time you will be back home every time you have obtained new information about in-between steps. Each time you can adjust your estimate on how long it will still take based on actually experienced times of parts of your path. This is the main principle of TD learning: you do not have to wait until the end of a trial to make updates along your path.

TD methods learn their value estimates based on estimates of other values, which is called *bootstrapping*. They have an advantage over DP in that they do not require a model of the MDP. Another advantage is that they are naturally implemented in an online, incremental fashion such that they can be easily used in various circumstances. No full sweeps through the full state space are needed; only along experienced paths values get updated, and updates are effected after each step.

TD(0).

TD(0) is a member of the family of TD learning algorithms [Sutton(1988)]. It solves the prediction problem, i.e. it estimates V^π for some policy π, in an online, incremental fashion. $TD(0)$ can be used to evaluate a policy and works through the use of the following update rule[5]:

$$V_{k+1}(s) := V_k(s) + \alpha\Big(r + \gamma V_k(s') - V_k(s)\Big)$$

[5] The learning parameter α should comply with some criteria on its value, and the way it is changed. In the algorithms in this section, one often chooses a small, fixed learning parameter, or it is decreased every iteration.

Require: discount factor γ, learning parameter α
  initialize Q arbitrarily (e.g. $Q(s,a) = 0, \forall s \in S, \forall a \in A$)
  for each episode **do**
    s is initialized as the *starting* state
    repeat
      choose an action $a \in A(s)$ based on an exploration strategy
      perform action a
      observe the new state s' and received reward r
      $Q(s,a) := Q(s,a) + \alpha\left(r + \gamma \cdot \max_{a' \in A(s')} Q(s',a') - Q(s,a)\right)$
      $s := s'$
    until s' is a goal state

Algorithm 4: Q-Learning [Watkins and Dayan(1992)]

where $\alpha \in [0,1]$ is the *learning rate*, that determines by how much values get updated. This backup is performed after experiencing the transition from state s to s' based on the action a, while receiving reward r. The difference with DP backups such as used in Equation 14 is that the update is still done by using bootstrapping, but it is based on an *observed* transition, i.e. it uses a *sample backup* instead of a full backup. Only the value of one successor state is used, instead of a weighted average of all possible successor states. When using the value function V^π for action selection, a model is needed to compute an expected value over all action outcomes (e.g. see Equation 4).

The learning rate α has to be decreased appropriately for learning to converge. Sometimes the learning rate can be defined for states separately as in $\alpha(s)$, in which case it can be dependent on how often the state is visited. The next two algorithms learn Q-functions directly from samples, removing the need for a transition model for action selection.

Q-learning.

One of the most basic and popular methods to estimate Q-value functions in a model-free fashion is the Q-learning algorithm [Watkins(1989), Watkins and Dayan(1992)], see Algorithm 4.

The basic idea in Q-learning is to incrementally estimate Q-values for actions, based on feedback (i.e. rewards) and the agent's Q-value function. The update rule is a variation on the theme of TD learning, using Q-values and a built-in max-operator over the Q-values of the next state in order to update Q_t into Q_{t+1}:

$$Q_{k+1}(s_t,a_t) = Q_k(s_t,a_t) + \alpha\left(r_t + \gamma \max_a Q_k(s_{t+1},a) - Q_k(s_t,a_t)\right) \qquad (18)$$

The agent makes a step in the environment from state s_t to s_{t+1} using action a_t while receiving reward r_t. The update takes place on the Q-value of action a_t in the state s_t from which this action was executed.

Q-learning is exploration-insensitive. It means that it will converge to the optimal policy regardless of the exploration policy being followed, under the assumption that each state-action pair is visited an infinite number of times, and the learning parameter α is decreased appropriately [Watkins and Dayan(1992), Bertsekas and Tsitsiklis(1996)].

SARSA.

Q-learning is an *off-policy* learning algorithm, which means that while following some exploration policy π, it aims at estimating the optimal policy π^*. A related *on-policy* algorithm that learns the Q-value function for the policy the agent is actually executing is the SARSA [Rummery and Niranjan(1994), Rummery(1995), Sutton(1996)] algorithm, which stands for **S**tate–**A**ction–**R**eward–**S**tate–**A**action. It uses the following update rule:

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha\Big(r_t + \gamma Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)\Big) \tag{19}$$

where the action a_{t+1} is the action that is executed by the current policy for state s_{t+1}. Note that the max-operator in Q-learning is replaced by the estimate of the value of the next action according to the policy. This learning algorithm will still converge in the limit to the optimal value function (and policy) under the condition that all states and actions are tried infinitely often and the policy converges in the limit to the greedy policy, i.e. such that exploration does not occur anymore. SARSA is especially useful in non-stationary environments. In these situations one will never reach an optimal policy. It is also useful if *function approximation* is used, because off-policy methods can diverge when this is used. However, off-policy methods are needed in many situations such as in learning using hierarchically structured policies.

Actor-Critic Learning.

Another class of algorithms that precede Q-learning and SARSA are *actor–critic* methods [Witten(1977), Barto et al(1983)Barto, Sutton, and Anderson, Konda and Tsitsiklis(2003)], which learn on-policy. This branch of TD methods keeps a *separate* policy independent of the value function. The policy is called the *actor* and the value function the *critic*. The critic – typically a state-value function – evaluates, or: criticizes, the actions executed by the actor. After action selection, the critic evaluates the action using the TD-error:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

The purpose of this error is to strengthen or weaken the selection of this action in this state. A *preference* for an action a in some state s can be represented as $p(s,a)$ such that this preference can be modified using:

$$p(s_t, a_t) := p(s_t, a_t) + \beta \delta_t$$

where a parameter β determines the size of the update. There are other versions of actor–critic methods, differing mainly in how preferences are changed, or experience is used (for example using *eligibility traces*, see next section). An advantage of having separate policy representation is that if there are many actions, or when the action space is continuous, there is no need to consider all actions' Q-values in order to select one of them. A second advantage is that they can learn *stochastic* policies naturally. Furthermore, a priori knowledge about policy constraints can be used, e.g. see [Främling(2005)].

Average Reward Temporal Difference Learning.

We have explained Q-learning and related algorithms in the context of discounted, infinite-horizon MDPs. Q-learning can also be adapted to the average-reward framework, for example in the R-learning algorithm [Schwartz(1993)]. Other extensions of algorithms to the average reward framework exist (see [Mahadevan(1996)] for an overview).

7.2 Monte Carlo Methods

Other algorithms that use more *unbiased* estimates are *Monte Carlo* (MC) techniques. They keep frequency counts of transitions and rewards and base their values on these estimates. MC methods only require samples to estimate average sample returns. For example, in MC policy evaluation, for each state $s \in S$ all returns obtained from s are kept and the value of a state $s \in S$ is just their average. In other words, MC algorithms treat the long-term reward as a random variable and take as its estimate the sampled mean. In contrast with one-step TD methods, MC estimates values based on *averaging sample returns* observed during interaction. Especially for episodic tasks this can be very useful, because samples from complete returns can be obtained. One way of using MC is by using it for the evaluation step in policy iteration. However, because the sampling is dependent on the current policy π, only returns for actions suggested by π are evaluated. Thus, exploration is of key importance here, just as in other model-free methods.

A distinction can be made between *every-visit* MC, which averages over all *visits* of a state $s \in S$ in all episodes, and *first-visit* MC, which averages over just the returns obtained from the first visit to a state $s \in S$ for all episodes. Both variants will converge to V^π for the current policy π over time. MC methods can also be applied to the problem of estimating action values. One way of ensuring enough exploration is to use *exploring starts*, i.e. each state-action pair has a non-zero probability of being selected as the initial pair. MC methods can be used for both on-policy and off-policy control, and the general pattern complies with the generalized policy

iteration procedure. The fact that MC methods do not *bootstrap* makes them less dependent on the *Markov assumption*. TD methods too focus on *sampled* experience, although they do use bootstrapping.

Learning a Model.

We have described MC methods in the context of learning value functions. Methods similar to MC can also be used to estimate a *model* of the MDP. An average over sample transition probabilities experienced during interaction can be used to gradually estimate transition probabilities. The same can be done for *immediate* rewards. *Indirect* RL algorithms make use of this to strike a balance between model-based and model-free learning. They are essentially model-free, but learn a transition and reward model in parallel with model-free RL, and use this model to do more efficient value function learning (see also the next section). An example of this is the DYNA model [Sutton(1991a)]. Another method that often employs model learning is *prioritized sweeping* [Moore and Atkeson(1993)]. Learning a model can also be very useful to learn in continuous spaces where the transition model is defined over a discretized version of the underlying (infinite) state space [Großmann(2000)].

Relations with Dynamic Programming.

The methods in this section solve essentially similar problems as DP techniques. RL approaches can be seen as *asynchronous* DP. There are some important differences in both approaches though.

RL approaches avoid the exhaustive sweeps of DP by restricting computation on, or in the neighborhood of, sampled trajectories, either real or simulated. This can exploit situations in which many states have low probabilities of occurring in actual trajectories. The backups used in DP are simplified by using sampling. Instead of generating and evaluating all of a state's possible immediate successors, the estimate of a backup's effect is done by sampling from the appropriate distribution. MC methods use this to base their estimates completely on the sample returns, without bootstrapping using values of other, sampled, states. Furthermore, the focus on learning (action) value functions in RL is easily amenable to *function approximation* approaches. Representing value functions and or policies can be done more compactly than lookup-table representations by using *numeric regression algorithms* without breaking the standard RL interaction process; one can just feed the update values into a regression engine.

An interesting point here is the similarity between Q-learning and *value iteration* on the one hand and SARSA and *policy iteration* on the other hand. In the first two methods, the updates immediately combine policy evaluation and improvement into one step by using the max-operator. In contrast, the second two methods separate evaluation and improvement of the policy. In this respect, value iteration can be considered as off-policy because it aims at directly estimating V^* whereas policy

iteration estimates values for the current policy and is on-policy. However, in the model-based setting the distinction is only superficial, because instead of samples that can be influenced by an on-policy distribution, a model is available such that the distribution over states and rewards is known.

7.3 Efficient Exploration and Value Updating

The methods in the previous section have shown that both prediction and control can be learned using samples from interaction with the environment, without having access to a model of the MDP. One problem with these methods is that they often need a large number of experiences to converge. In this section we describe a number of extensions used to speed up learning. One direction for improvement lies in the exploration. One can – in principle – use MC sampling until one knows everything about the MDP but this simply takes too long. Using more information enables more focused exploration procedures to generate experience more efficiently. Another direction is to put more efforts in using the experience for updating multiple values of the value function on each step. Improving exploration generates *better samples*, whereas improving updates will squeeze *more information* from samples.

Efficient Exploration.

We have already encountered ε-greedy and *Boltzmann* exploration. Although commonly used, these are relatively simple *undirected* exploration methods. They are mainly driven by *randomness*. In addition, they are *stateless*, i.e. the exploration is driven without knowing which areas of the state space have been explored so far.

A large class of *directed* methods for exploration have been proposed in the literature that use additional information about the learning process. The focus of these methods is to do more uniform exploration of the state space and to balance the relative benefits of discovering new information relative to exploiting current knowledge. Most methods use or learn a model of the MDP in parallel with RL. In addition they learn an *exploration value function*.

Several options for directed exploration are available. One distinction between methods is whether to work *locally* (e.g. exploration of individual state-action pairs) or *globally* by considering information about parts or the complete state-space when making a decision to explore. Furthermore, there are several other classes of exploration algorithms.

Counter-based or *recency-based* methods keep records of how often, or how long ago, a state-action pair has been visited. *Error-based* methods, of which *prioritized sweeping* [Moore and Atkeson(1993)] is one example, use an exploration bonus based on the error in the value of states. Other methods base exploration on the *uncertainty* about the value of a state, or the *confidence* about the state's current value. They decide whether to explore by calculating the probability that an

explorative action will encover a larger reward than already found. The *interval estimation* (IE) method [Kaelbling(1993)] is an example of this kind of methods. IE uses a statistical model to measure the degree of uncertainty of each $Q(s,a)$-value. An upper bound can be calculated on the likely value of each Q-value, and the action with the highest upper bound is taken. If the action taken happens to be a poor choice, the upper bound will be decreased when the statistical model is updated. Good actions will continue to have a high upper bound and will be chosen often. In contrast to counter- and recency-based exploration, IE is concerned with *action exploration* and not with *state space* exploration. [Wiering(1999)] (see also [**?**]) introduced an extension to model-based RL in the *model-based interval estimation* algorithm in which the same idea is used for estimates of transition probabilities.

Another recent method that deals explicitly with the exploration-exploitation trade-off is the E^3 method [Kearns and Singh(1998)]. E^3 stands for *explicit exploration and exploitation*. It learns by updating a model of the environment by collecting statistics. The state space is divided into *known* and *unknown* parts. On every step a decision is made whether the known part contains sufficient opportunities for getting rewards or whether the unknown part should be explored to obtain possibly more reward. An important aspect of this algorithm is that it was the first general near-optimal (tabular) RL algorithm with provable bounds on computation time. The approach was extended in [Brafman and Tennenholtz(2002)] into the more general algorithm R-MAX. It too provides a polynomial bound on computation time for reaching near-optimal policies. As a last example, [Ratitch(2005)] present an approach for efficient, directed exploration based on more sophisticated characteristics of the MDP such as an *entropy* measure over state transitions. An interesting feature of this approach is that these characteristics can be computed *before* learning and be used in combination with other exploration methods, thereby improving their behavior.

For a more detailed coverage of exploration strategies we refer the reader to [Ratitch(2005)] and [Wiering(1999)].

Guidance and Shaping.

Exploration methods can be used to speed up learning and focus attention to relevant areas in the state space. The exploration methods mainly use statistics derived from the problem before or during learning. However, sometimes more information is available that can be used to *guide* the learner. For example, if a reasonable policy for a domain is available, it can be used to generate more useful learning samples than (random) exploration could do. In fact, humans are usually very bad in specifying optimal policies, but considerably good at specifying reasonable ones[6].

The work in *behavioral cloning* [Bain and Sammut(1995)] takes an extreme point on the guidance spectrum in that the goal is to *replicate* example behavior from expert traces, i.e. to *clone* this *behavior*. This type of guidance moves learning more

[6] Quote taken from the invited talk by Leslie Kaelbling at the European Workshop on Reinforcement Learning EWRL, in Utrecht 2001.

in the direction of *supervised* learning. Another way to help the agent is by *shaping* [Mataric(1994), Dorigo and Colombetti(1997), Ng et al(1999)Ng, Harada, and Russell]. Shaping pushes the reward closer to the subgoals of behavior, and thus encourages the agent to incrementally improve its behavior by searching the policy space more effectively. This is also related to the general issue of giving rewards to appropriate subgoals, and the gradual increase in difficulty of tasks. The agent can be trained on increasingly more difficult problems, which can also be considered as a form of guidance.

Various other mechanisms can be used to provide guidance to RL algorithms, such as decompositions [**?**], heuristic rules for better exploration [Främling(2005)] and various types of *transfer* in which knowledge learned in one problem is transferred to other, related problems, e.g. see [**?**].

Eligibility Traces.

In MC methods, the updates are based on the entire sequence of observed rewards until the end of an episode. In TD methods, the estimates are based on the samples of immediate rewards and the next states. An intermediate approach is to use the *n-step-truncated-return* $R_t^{(n)}$, obtained from a whole sequence of returns:

$$R_t^{(n)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n V_t(s_{t+n})$$

With this, one can go to the approach of computing the updates of values based on several n-step returns. The family of TD(λ), with $0 \leq \lambda \leq 1$, combines n-step returns weighted proportionally to λ^{n-1}.

The problem with this is that we would have to wait indefinitely to compute $R_t^{(\infty)}$. This view is useful for theoretical analysis and understanding of n-step backups. It is called the *forward view of the* TD(λ) *algorithm*. However, the usual way to implement this kind of updates is called the *backward view of the* TD(λ) *algorithm* and is done by using *eligibility traces*, which is an incremental implementation of the same idea.

Eligibility traces are a way to perform n-step backups in an elegant way. For each state $s \in S$, an eligibility $e_t(s)$ is kept in memory. They are initialized at 0 and incremented every time according to:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

where λ is the *trace decay parameter*. The trace for each state is increased every time that state is visited and decreases exponentially otherwise. Now δ_t is the *temporal difference* error at stage t:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

On every step, all states are updated in proportion to their eligibility traces as in:

$$V(s) := V(s) + \alpha \delta_t e_t(s)$$

The forward and backward view on eligibility traces can be proved equivalent [Sutton and Barto(1998)]. For $\lambda = 1$, TD(λ) is essentially the same as MC, because it considers the complete return, and for $\lambda = 0$, TD(λ) uses just the immediate return as in all one-step RL algorithms. Eligibility traces are a general mechanism to learn from n-step returns. They can be combined with all of the model-free methods we have described in the previous section. [Watkins(1989)] combined Q-learning with eligibility traces in the $Q(\lambda)$-algorithm. [Peng and Williams(1996)] proposed a similar algorithm, and [Wiering and Schmidhuber(1998)] and [Reynolds(2002)] both proposed efficient versions of $Q(\lambda)$. The problem with combining eligibility traces with learning *control* is special care has to be taken in case of *exploratory* actions, which can break the intended meaning of the n-step return for the current policy that is followed. In [Watkins(1989)]'s version, eligibility traces are reset every time an exploratory action is taken. [Peng and Williams(1996)]'s version is, in that respect more efficient, in that traces do not have to be set to zero every time. SARSA(λ) [Sutton and Barto(1998)] is more safe in this respect, because action selection is on-policy. Another recent on-policy learning algorithm is the QV(λ) algorithm by [Wiering(2005)]. In QV(λ)-learning two value functions are learned; TD(λ) is used for learning a state value function V and one-step Q-learning is used for learning a state-action value function, based on V.

Learning and Using a Model: Learning and Planning.

Even though RL methods can function without a model of the MDP, such a model can be useful to speed up learning, or bias exploration. A learned model can also be useful to do more efficient value *updating*. A general guideline is when experience is costly, it pays off to learn a model. In RL model-learning is usually targeted at the specific learning task defined by the MDP , i.e. determined by the rewards and the goal. In general, learning a model is most often useful because it gives knowledge about the *dynamics* of the environment, such that it can be used for other tasks too (see [**?**] for extensive elaboration on this point).

The DYNA architecture [Sutton(1990), Sutton(1991b), Sutton(1991a), Sutton and Barto(1998)] is a simple way to use the model to amplify experiences. Algorithm 5 shows DYNA-Q which combines Q-learning with planning. In a continuous loop, Q-learning is interleaved with series of extra updates using a model that is constantly updated too. DYNA needs less interactions with the environment, because it *replays* experience to do more value updates.

A related method that makes more use of experience using a learned model is *prioritized sweeping* (PS) [Moore and Atkeson(1993)]. Instead of selecting states to be updated randomly (as in DYNA), PS prioritizes updates based on their change in values. Once a state is updated, the PS algortithm considers all states that can reach

Require: initialize Q and Model arbitrarily
  repeat
    $s \in S$ is the start state
    $a := \varepsilon$-greedy(s, Q)
    update Q
    update Model
    for $i := 1$ **to** n **do**
      $s :=$ randomly selected observed state
      $a :=$ random, previously selected action from s
      update Q using the model
  until Sufficient Performance

Algorithm 5: Dyna-Q [Sutton and Barto(1998)]

that state, by looking at the transition model, and sees whether these states will have to be updated as well. The order of the updates is determined by the size of the value updates. The general mechanism can be summarized as follows. In each step i) one remembers the old value of the current state, ii) one updates the state value with a full backup using the learned model, iii) one sets the priority of the current state to 0, iv) one computes the change δ in value as the result of the backup, v) one uses this difference to modify *predecessors* of the current state (determined by the model); all states leading to the current state get a priority update of $\delta \times T$. The number of value backups is a parameter to be set in the algorithm. Overall, PS focuses the backups to where they are expected to most quickly reduce the error. Another example of using planning in model-based RL is [Wiering(2002)].

References

[Bain and Sammut(1995)] Bain M, Sammut C (1995) A framework for behavioral cloning. Machine Learning 15

[Barto et al(1983)Barto, Sutton, and Anderson] Barto AG, Sutton RS, Anderson CW (1983) Neuronlike elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics 13:835–846

[Barto et al(1995)Barto, Bradtke, and Singh] Barto AG, Bradtke SJ, Singh SP (1995) Learning to act using real-time dynamic programming. Artificial Intelligence 72(1):81–138

[Bellman(1957)] Bellman RE (1957) Dynamic Programming. Princeton University Press, Princeton, New Jersey

[Bertsekas(1995)] Bertsekas D (1995) Dynamic Programming and Optimal Control, volumes 1 and 2. Athena Scientific, Belmont, MA

[Bertsekas and Tsitsiklis(1996)] Bertsekas DP, Tsitsiklis J (1996) Neuro-Dynamic Programming. Athena Scientific, Belmont, MA

[Bonet and Geffner(2003a)] Bonet B, Geffner H (2003a) Faster heuristic search algorithms for planning with uncertainty and full feedback. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp 1233–1238

[Bonet and Geffner(2003b)] Bonet B, Geffner H (2003b) Labeled rtdp: Improving the convergence of real-time dynamic programming. In: Proceedings of the International Conference on Artificial Planning Systems (ICAPS'03), pp 12–21

[Boutilier(1999)] Boutilier C (1999) Knowledge representation for stochastic decision processes. Lecture Notes in Computer Science 1600:111–152

[Boutilier et al(1999)Boutilier, Dean, and Hanks] Boutilier C, Dean T, Hanks S (1999) Decision theoretic planning: Structural assumptions and computational leverage. Journal of Artificial Intelligence Research 11:1–94

[Brafman and Tennenholtz(2002)] Brafman R, Tennenholtz M (2002) R-MAX - a general polynomial time algorithm for near-optimal reinforcement learning. Journal of Machine Learning Research 3:213–231

[Brendan McMahan et al(2005)Brendan McMahan, Likhachev, and Gordon] Brendan McMahan H, Likhachev M, Gordon GJ (2005) Bounded real-time dynamic programming: Rtdp with monotone upper bounds and performance guarantees. In: Proceedings of the International Conference on Machine Learning (ICML), pp 569–576

[Dean et al(1995)Dean, Kaelbling, Kirman, and Nicholson] Dean T, Kaelbling LP, Kirman J, Nicholson A (1995) Planning under time constraints in stochastic domains. Artificial Intelligence 76:35–74

[Dorigo and Colombetti(1997)] Dorigo M, Colombetti M (1997) Robot shaping: an experiment in behavior engineering. MIT Press, Cambridge, MA

[Ferguson and Stentz(2004)] Ferguson D, Stentz A (2004) Focussed dynamic programming: Extensive comparative results. Tech. Rep. CMU-RI-TR-04-13, Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania

[Främling(2005)] Främling K (2005) Bi-memory model for guiding exploration by pre-existing knowledge. In: Driessens K, Fern A, van Otterlo M (eds) Proceedings of the ICML-2005 Workshop on Rich Representations for Reinforcement Learning, pp 21–26

[Großmann(2000)] Großmann A (2000) Adaptive state-space quantisation and multi-task reinforcement learning using constructive neural networks. In: Meyer JA, Berthoz A, Floreano D, Roitblat HL, Wilson SW (eds) From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB), pp 160–169

[Hansen and Zilberstein(2001)] Hansen EA, Zilberstein S (2001) LAO*: A heuristic search algorithm that finds solutions with loops. Artificial Intelligence 129:35–62

[Howard(1960)] Howard RA (1960) Dynamic Programming and Markov Processes. The MIT Press, Cambridge, Massachusetts

[Kaelbling(1993)] Kaelbling LP (1993) Learning in Embedded Systems. MIT Press, Cambridge, MA

[Kaelbling et al(1996)Kaelbling, Littman, and Moore] Kaelbling LP, Littman ML, Moore AW (1996) Reinforcement learning: A survey. Journal of Artificial Intelligence Research 4:237–285

[Kearns and Singh(1998)] Kearns M, Singh S (1998) Near-optimal reinforcement learning in polynomial time. In: Proceedings of the International Conference on Machine Learning (ICML)

[Koenig and Liu(2002)] Koenig S, Liu Y (2002) The interaction of representations and planning objectives for decision-theoretic planning. Journal of Experimental and Theoretical Artificial Intelligence

[Konda and Tsitsiklis(2003)] Konda V, Tsitsiklis J (2003) Actor-critic algorithms. SIAM Journal on Control and Optimization 42(4):1143–1166

[Littman et al(1995)Littman, Dean, and Kaelbling] Littman ML, Dean TL, Kaelbling LP (1995) On the complexity of solving markov decision problems. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp 394–402

[Mahadevan(1996)] Mahadevan S (1996) Average reward reinforcement learning: Foundations, algorithms, and empirical results. Machine Learning 22:159–195

[Maloof(2003)] Maloof MA (2003) Incremental rule learning with partial instance memory for changing concepts. In: Proceedings of the International Joint Conference on Neural Networks, pp 2764–2769

[Mataric(1994)] Mataric M (1994) Reward functions for accelerated learning. In: Proceedings of the International Conference on Machine Learning (ICML)

[Matthews(1922)] Matthews WH (1922) Mazes and Labyrinths: A General Account of their History and Developments. Longmans, Green and Co., London, reprinted in 1970 by Dover Publications, New York, under the title 'Mazes & Labyrinths: Their History & Development

[Moore and Atkeson(1993)] Moore AW, Atkeson CG (1993) Prioritized sweeping: Reinforcement learning with less data and less time. Machine Learning 13(1):103–130

[Ng et al(1999)Ng, Harada, and Russell] Ng AY, Harada D, Russell SJ (1999) Policy invariance under reward transformations: Theory and application to reward shaping. In: Proceedings of the International Conference on Machine Learning (ICML), pp xx–xx

[Peng and Williams(1996)] Peng J, Williams RJ (1996) Incremental multi-step q-learning. Machine Learning 22:283–290

[Puterman(1994)] Puterman ML (1994) Markov Decision Processes—Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY

[Puterman and Shin(1978)] Puterman ML, Shin MC (1978) Modified policy iteration algorithms for discounted Markov decision processes. Management Science 24:1127–1137

[Ratitch(2005)] Ratitch B (2005) On characteristics of markov decision processes and reinforcement learning in large domains. PhD thesis, The School of Computer Science, McGill University, Montreal

[Reynolds(2002)] Reynolds SI (2002) Reinforcement learning with exploration. PhD thesis, The School of Computer Science, The University of Birmingham, UK

[Rummery(1995)] Rummery GA (1995) Problem solving with reinforcement learning. PhD thesis, Cambridge University, Engineering Department, Cambridge, England

[Rummery and Niranjan(1994)] Rummery GA, Niranjan M (1994) On-line Q-Learning using connectionist systems. Tech. Rep. CUED/F-INFENG/TR 166, Cambridge University, Engineering Department

[Schaeffer and Plaat(1997)] Schaeffer J, Plaat A (1997) Kasparov versus deep blue: The re-match. International Computer Chess Association Journal 20(2):95–101

[Schwartz(1993)] Schwartz A (1993) A reinforcement learning method for maximizing undiscounted rewards. In: Proceedings of the International Conference on Machine Learning (ICML), pp 298–305

[Sutton(1990)] Sutton R (1990) Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: Proceedings of the International Conference on Machine Learning (ICML), pp 216–224

[Sutton(1988)] Sutton RS (1988) Learning to predict by the methods of temporal differences. Machine Learning 3:9–44

[Sutton(1991a)] Sutton RS (1991a) DYNA, an integrated architecture for learning, planning and reacting. In: Working Notes of the AAAI Spring Symposium on Integrated Intelligent Architectures, pp 151–155

[Sutton(1991b)] Sutton RS (1991b) Reinforcement learning architectures for animats. In: From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB), pp 288–296

[Sutton(1996)] Sutton RS (1996) Generalization in reinforcement learning: Successful examples using sparse coarse coding. In: Touretzky DS, Mozer MC, Hasselmo ME (eds) Proceedings of the Neural Information Processing Conference (NIPS), vol 8, pp 1038–1044

[Sutton and Barto(1998)] Sutton RS, Barto AG (1998) Reinforcement Learning: an Introduction. The MIT Press, Cambridge

[Tash and Russell(1994)] Tash J, Russell S (1994) Control strategies for a stochastic planner. In: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94), pp 1079–1085

[Watkins(1989)] Watkins CJCH (1989) Learning from delayed rewards. PhD thesis, King's College, Cambridge, England

[Watkins and Dayan(1992)] Watkins CJCH, Dayan P (1992) Q-learning. Machine Learning 8(3/4), special Issue on Reinforcement Learning

[Wiering(1999)] Wiering MA (1999) Explorations in efficient reinforcement learning. PhD thesis, Faculteit der Wiskunde, Informatica, Natuurkunde en Sterrenkunde, Universiteit van Amsterdam

[Wiering(2002)] Wiering MA (2002) Model-based reinforcement learning in dynamic environments. Tech. Rep. UU-CS-2002-029, Institute of Information and Computing Sciences, University of Utrecht, The Netherlands

[Wiering(2005)] Wiering MA (2005) QV(λ)-Learning: A new on-policy reinforcement learning algorithm. In: Proceedings of the 7th European Workshop on Reinforcement Learning

[Wiering and Schmidhuber(1998)] Wiering MA, Schmidhuber JH (1998) Fast online q(λ). Machine Learning xx:xxx–xxx

[Winston(1991)] Winston WL (1991) Operations research applications and algorithms, 2nd edn. Thomson Information/Publishing Group, Boston

[Witten(1977)] Witten IH (1977) An adaptive optimal controller for discrete-time markov environments. Information and Control 34:286–295