Chapter 9
Hierarchical Approaches

Bernhard Hengst

Abstract. Hierarchical decomposition tackles complex problems by reducing them to a smaller set of interrelated problems. The smaller problems are solved separately and the results re-combined to find a solution to the original problem. It is well known that the naïve application of reinforcement learning (RL) techniques fails to scale to more complex domains. This Chapter introduces hierarchical approaches to reinforcement learning that hold out the promise of reducing a reinforcement learning problems to a manageable size. Hierarchical Reinforcement Learning (HRL) rests on finding good re-usable temporally extended actions that may also provide opportunities for state abstraction. Methods for reinforcement learning can be extended to work with abstract states and actions over a hierarchy of subtasks that decompose the original problem, potentially reducing its computational complexity. We use a four-room task as a running example to illustrate the various concepts and approaches, including algorithms that can automatically learn the hierarchical structure from interactions with the domain.

9.1 Introduction

Artificial intelligence (AI) is about how to construct agents that act rationally. An agent acts rationally when it maximises a performance measure given a sequence of perceptions (Russell and Norvig, 1995). Planning and control theory can also be viewed from an agent perspective and included in this problem class. Reinforcement learning may at first seem like a seductive approach to solve the *artificial general intelligence (AGI)* problem (Hutter, 2007). While in principle this may be true, reinforcement learning is beleaguered by the "curse of dimensionality". The curse of dimensionality is a term coined by Bellman (1961) to refer to the exponential

Bernhard Hengst
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
e-mail: bernhardh@cse.unsw.edu.au

increase in the state-space with each additional variable or dimension that describes the problem. Bellman noted that sheer enumeration will not solve problems of any significance. It is unlikely that complex problems can be described by only a few variables, and so, it may seem we are at an impasse.

Fortunately the real world is highly structured with many constraints and with most parts independent of most other parts. Without structure it would be impossible to solve complex problems of any size (Russell and Norvig, 1995). Structure can significantly reduce the naïve state space generated by sheer enumeration. For example, if the transition and reward functions for two variables are independent, then a reinforcement learner would only need to explore a state-space the size of the addition of the state-space sizes of the variables instead of one the size of their product.

Reinforcement learning is concerned with problems represented by actions as well as states, generalising problem solving to systems that are dynamic in time. Hence we often refer to reinforcement learning problems as tasks, and sub-problems as sub-tasks.

This Chapter is concerned with leveraging hierarchical structure to try to reduce and solve more complex reinforcement learning problems that would otherwise be difficult if not impossible to solve.

Hierarchy

Empirically, a large proportion of complex systems seen in nature, exhibit hierarchical structure. By hierarchy we mean that the system is composed of interrelated sub-systems, that in turn have their own subsystems, and so on. Human societies have used hierarchical organisations to solve complex tasks dating back to at least Egyptian times. Hierarchical systems can be characterised as *nearly decomposable*, meaning that intra-component linkages are generally stronger than inter-component linkages (Simon, 1996). The property of near decomposability can help simplify their behaviour and description.

A heuristic from Polya (1945) for problem solving is *decomposing and recombining*, or *divide and conquer* in today's parlance. Many large problems have hierarchical structure that allows them to be broken down into sub-problems. The sub-problems, being smaller, are often solved more easily. The solutions to the sub-problems are recombined to provide the solution for the original larger problem. The decomposition can make finding the final solution significantly more efficient with improvements in the time and space complexity for both learning and execution.

There is another reason that hierarchy can simplify problem solving and make the task of learning more efficient. It is often the case that similar tasks need to be executed in different contexts. If the reinforcement learner is unaware of the underlying structure, it would relearn the same task in multiple different contexts, when it is clearly better to learn the task once and reuse it. Programmers use function calls and subroutines for just this purpose – to avoid repeating similar code fragments. In many ways *hierarchical reinforcement learning (HRL)* is about structuring reinforcement learning problems much like a computer program where subroutines are akin to subtasks of a higher-level reinforcement learning problem. Just as the main program calls subroutines, a higher level reinforcement problem invokes subtasks.

Four-Room Task

Throughout this Chapter we will use a simple four-room task as a running example to help illustrate concepts. Figure 9.1 (left) shows the agent view of reinforcement learning. The agent interacts with the environment in a sense-act loop, receiving a reward signal at each time-step as a part of the input.

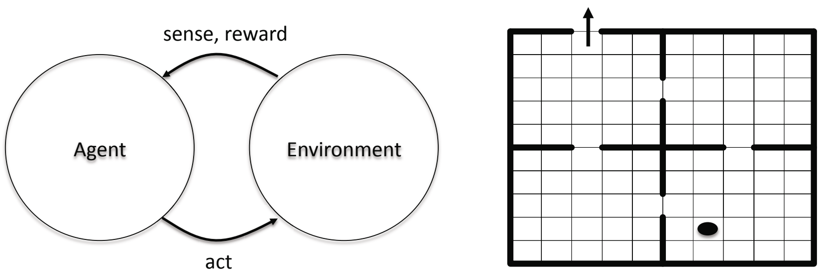

Fig. 9.1 Left: The agent view of Reinforcement Learning. Right: A four-room task with the agent in one of the rooms shown as a solid black oval.

In the four-room example, the agent is represented as a black oval at a grid location in the South-East room of a four-room house (Figure 9.1, right). The rooms are connected by open doorways. The North-West room has a doorway leading out of the house. At each time-step the agent takes an action and receives a sensor observation and a reward from the environment.

Each cell represents a possible agent position. The position is uniquely described by each room and the position in each room. In this example the rooms have similar dimensions and similar positions in each room are assumed to be described by the same identifier. The environment is *fully observable* by the agent which is able to sense both the room it occupies and its position in the room. The agent can move one cell-step in any of the four compass directions at each time-step. It also receives a reward of -1 at each time-step. The objective is to leave the house via the least-cost route. We assume that the actions are stochastic. When an action is taken there is an 80% chance that the agent will move in the intended direction and a 20% chance that it will stay in place. If the agent moves into a wall it will remain where it is.

Hierarchical Reinforcement Learning (HRL)

Our approach to HRL starts with a well specified reinforcement learning problem modelled as an Markov Decision Process (MDP) as described in Chapter 1. The reader can easily verify that the four-room task is such a reinforcement learning problem. We provide here an initial intuitive description of how HRL methods might be applied to the four-room task.

If we can find a policy to leave a room, say by the North doorway, then we could reuse this policy in any of the rooms because they are identical. The problem to leave a room by the North doorway is just another, albeit smaller, reinforcement learning problem that has inherited the position-in-room states, actions, transition and reward function from the original problem. We proceed to solve two smaller reinforcement learning problems, one to find a room-leaving policy to the North and another to leave a room through the West doorway.

We also formulate and solve a higher-level reinforcement learning problem that uses only the four room-states. In each room-state we allow a choice of executing one of the previously learnt room-leaving policies. For the higher-level problem these policies are viewed as *temporally extended* actions because once they are invoked they will usually persist for multiple time-steps until the agent exits a room. At this stage we simply specify a reward of -1 per room-leaving action. As we shall see later, reinforcement learning can be generalised to work with temporally extended actions using the formalism of semi-Markov Decision Processes (SMDP).

Once learnt, the execution of the higher-level house-leaving policy will determine the room-leaving action to invoke given the current room. Control is passed to the room-leaving sub-task that leads the agent out of the room through the chosen doorway. Upon leaving the room, the sub-task is terminated and control is passed back to the higher level that chooses the next room-leaving action until the agent finally leaves the house.

The above example hides many issues that HRL needs to address, including: safe state abstraction; appropriately accounting for accumulated sub-task reward; optimality of the solution; and specifying or even learning of the hierarchical structure itself. In the next sections we will discuss these issues and review several approaches to HRL.

9.2 Background

This section will introduce several concepts that are important to understanding hierarchical reinforcement learning (HRL). There is general agreement that temporally extended actions and the related semi-Markov Decision Process formalism are key ingredients. We will also discuss issues related to problem size reduction and solution optimality.

9.2.1 Abstract Actions

Approaches to HRL employ actions that persist for multiple time-steps. These temporally extended or *abstract actions* hide the multi-step state-transition and reward details from the time they are invoked until termination. The room-leaving actions discussed above for the four-room task are examples. Leaving a room may involve taking multiple single-step actions to first navigate to a doorway and then stepping through it.

Abstract actions are employed in many fields including AI, robotics and control engineering. They are similar to *macros* in computer science that make available a sequence of instructions as a single program statement. In planning, macros help decompose and solve problems (see for example solutions to the Fifteen Puzzle and Rubik's Cube (Korf, 1985)).

Abstract actions in an MDP setting extend macros in that the more primitive steps that comprise an abstract action may be modelled with stochastic transition functions and use stochastic polices (see Chapter 1). Abstract actions may execute a policy for a smaller Markov Decision Problem. Stochasticity can manifest itself in several ways. When executing an abstract action a stochastic transition function will make the sequence of states visited non-deterministic. The sequence of rewards may also vary, depending on the sequence of states visited, even if the reward function itself is deterministic. Finally, the time taken to complete an abstract action may vary. Abstract actions with deterministic effects are just classical macro operators.

Properties of abstract actions can be seen in the four-room task. In this problem the 20% chance of staying in situ, does not make it possible to determine beforehand how many time-steps it will take to leave a room when a room-leaving abstract action is invoked.

Abstract actions can be generalised for continuous-time (Puterman, 1994), however, this Chapter will focus on discrete-time problems. Special case abstract actions that terminate in one time-step are just ordinary actions and we refer to them as *primitive* actions.

9.2.2 Semi-Markov Decision Problems

We will now extend MDPs from Chapter 1 to MDPs that include abstract actions. MDPs that include abstract actions are called *semi Markov Decision Problems*, or SMDPs (Puterman, 1994). As abstract actions can take a random number of time-steps to complete, we need to introduce another variable to account for the time to termination of the abstract action.

We denote the random variable $N \geq 1$ to be the number of time steps that an abstract action a takes to complete, starting in state s and terminating in state s'.

The model of the SMDP, defined by the state transition probability function and the expected reward function now includes the random variable N.[1]

The transition function $T : S \times A \times S \times N \to [0,1]$ gives the probability of the abstract action a terminating in state s' after N steps, having been initiated in state s.

$$T(s,a,s',N) = Pr\{s_{t+N} = s' | s_t = s, a_t = a\} \tag{9.1}$$

The reward function for an abstract action accumulates single step rewards as it executes. The manner in which rewards are summed depends on the performance measure included with the SMDP. A common performance measure is to maximise the sum of future discounted rewards using a constant discount factor γ. The reward function $R : S \times A \times S \times N \to \mathbb{R}$ that gives the expected discounted sum of rewards when an abstract action a is started in state s and terminates in state s' after N steps is

$$R(s,a,s',N) = E\left\{\sum_{n=0}^{N-1} \gamma^n r_{t+n} | s_t = s, a_t = a, s_{t+N} = s'\right\} \tag{9.2}$$

The value functions and Bellman "backup" equations from Chapter 1 for MDPs can also be generalised for SMDPs. The value of state s for policy π, denoted $V^\pi(s)$ is the expected return starting in state s at time t, and taking abstract actions according to π.[2]

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

If the abstract action executed in state s is $\pi(s)$, persists for N steps and terminates we can write the value function as two series – the sum of the rewards accumulated for the first N steps and the remainder of the series of rewards.

$$V^\pi(s) = E_\pi\left\{(r_t + \gamma r_{t+1} + \ldots + \gamma^{N-1} r_{t+N-1}) + (\gamma^N r_{t+N} + \ldots) | s_t = s\right\}$$

Taking the expectation with respect to both s' and N with probabilities given by Equation 9.1, substituting the N-step reward for abstract action $\pi(s)$ from Equation 9.2, and recognising that the second series is just the value function starting in s' discounted by N steps, we can write

$$V^\pi(s) = \sum_{s',N} T(s,\pi(s),s',N))[R(s,\pi(s),s',N) + \gamma^N V^\pi(s')]$$

[1] This formulation of an SMDP is based on Sutton et al (1999) and Dietterich (2000), but has been changed to be consistent with notation in the Chapter 1.

[2] Note that we are overloading function π. $\pi(s,a)$ is the probability of choosing abstract action a in state s. $\pi(s)$ is the abstract action that is chosen in state s under a deterministic policy π.

The optimum value function for an SMDP (denoted by $*$) is also similar to that for MDPs with the sum taken with respect to s' and N.

$$V^*(s) = \max_a \sum_{s',N} T(s,a,s',N)[R(s,a,s',N) + \gamma^N V^*(s')]$$

Similarly, Q abstract action value functions can be written for SMDP policies. The definition of $Q^\pi(s,a)$ is the value of taking abstract action a in state s and then following the SMDP policy π thereafter. Hence

$$Q^\pi(s,a) = \sum_{s',N} T(s,a,s',N))[R(s,a,s',N) + \gamma^N Q^\pi(s',\pi(s'))] \tag{9.3}$$

The optimum SMDP value function is

$$\begin{aligned} Q^*(s,a) &= \sum_{s',N} T(s,a,s',N))[R(s,a,s',N) + \gamma^N V^*(s')] \\ \text{where } V^*(s') &= \max_{a'} Q^*(s',a') \end{aligned}$$

For problems that are guaranteed to terminate, the discount factor γ can be set to 1. In this case the number of steps N can be marginalised out in the above equations and the sum taken with respect to s alone. The equations are then similar to the ones for MDPs with the expected primitive reward replaced with the expected sum of rewards to termination of the abstract action.

All the methods developed for solving Markov decision processes in the Chapter 1 for reinforcement learning using primitive actions work equally well for problems using abstract actions. As primitive actions are just a special case of abstract actions we include them in the set of abstract actions.

The reader may well wonder whether the introduction of abstract actions buys us anything. After all we have just added extra actions to the problem and increased the complexity. The rest of this Chapter will show how abstract actions allow us to leverage structure in problems to reduce storage requirements and increase the speed of learning.

In a similar four-room example to that of Figure 9.1, Sutton et al (1999) show how the presence of abstract actions allows the agent to learn significantly faster proceeding on a room by room basis, rather than position by position[3]. When the goal is not in a convenient location, able to be reached by the given abstract actions, it is possible to include primitive actions as special-case abstract actions and still accelerate learning for some problems. For example, with room-leaving abstract actions alone, it may not be possible to reach a goal in the middle of a room.

Unless we introduce other abstract actions, primitive actions are still required when the room containing the goal state is entered. Although the inclusion of primitive actions guarantees convergence to the globally optimal policy, this may create extra work for the learner. Reinforcement learning may be accelerated because the

[3] In this example, abstract actions take the form of options and will be defined in Subsection 9.3.1

value function can be backed-up over greater distances in the state-space and the inclusion of primitive actions guarantees convergence to the globally optimal policy, but the introduction of additional actions increased the storage and exploration necessary.

9.2.3 Structure

Abstract actions and SMDPs naturally lead to hierarchical structure. With appropriate abstract actions alone we may be able to learn a policy with less effort than it would take to solve the problem using primitive actions. This is because abstract actions can skip over large parts of the state-space terminating in a small subset of states. We saw in the four-room task how room-leaving abstract actions are able to reduce the problem state-space to room states alone.

Abstract actions themselves may be policies from smaller SMDPs (or MDPs). This establishes a hierarchy where a higher-level *parent* task employs *child* subtasks as its abstract actions.

Task Hierarchies

The parent-child relationship between SMDPs leads to *task-hierarchies* (Dietterich, 2000). A task-hierarchy is a directed acyclic graph of sub-tasks. The root-node in the task-hierarchy is a top-level SMPD that can invoke its child-node SMDP policies as abstract actions. Child-node policies can recursively invoke other child subtasks, right down to sub-tasks that only invoke primitive actions. The designer judiciously specifies abstract actions that are available in states of each parent task, and specifies the active states and terminal states for each sub-task.

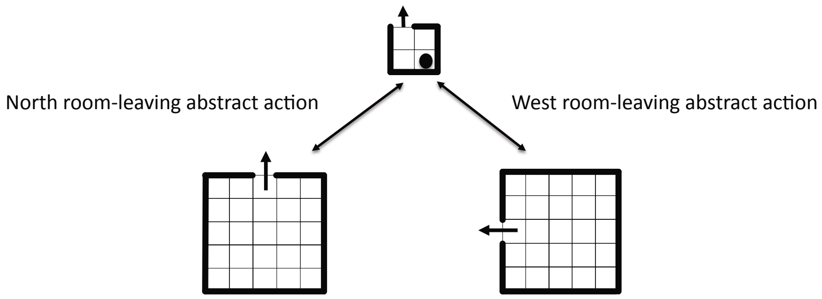
North room-leaving abstract action
West room-leaving abstract action

Fig. 9.2 A task-hierarchy decomposing the four-room task in Figure 9.1. The two lower-level sub-tasks are generic room-leaving abstract actions, one each for leaving a room to the North and West.

Figure 9.2 shows a task-hierarchy for the four-room task discussed in Section 9.1. The two lower-level sub-tasks are MDPs for a generic room, where separate policies are learnt to exit a room to the North and West. The arrows indicate transitions to terminal states. States, actions, transitions and rewards are inherited from the original MDP. The higher level problem (SMDP) consists of just four states representing the rooms. Any of the sub-tasks (room-leaving actions) can be invoked in any of the rooms.

Partial Programs

Manually specifying a task-hierarchy is a form of programming allowing the designer to include background knowledge to exploit the hierarchical structure of the problem. Not all sub-task policies need to be learnt. If a programmer already has knowledge of good sub-task polices these can be hand-coded and supplied in the form of stochastic finite state machines.

As computers are finite state machines and can approximate stochastic finite state machines, task-hierarchies may be able to be efficiently represented as partial programs using the full programming structure available. We will elaborate on this approach to HRL in Section 9.3.2.

9.2.4 State Abstraction

The benefit of decomposing a large MDP is that it will hopefully lead to *state abstraction* opportunities to help reduce the complexity of the problem. An abstracted state space is smaller than the state space of an original MDP. There are broadly two kinds of conditions under which state-abstractions can be introduced (Dietterich, 2000). They are situations in which:

- we can eliminate irrelevant variables, and
- where abstract actions "funnel" the agent to a small subset of states.

Eliminating Irrelevant Variables

When reinforcement learning algorithms are given redundant information they will learn the same value-function or policy for all the redundant states. For example, navigating through a red coloured room may be the same as for a blue coloured room, but a value function and policy treats each $(position\text{-}in\text{-}room, colour)$ as a different state. If colour has no effect on navigation it would simplify the problem by eliminating the colour variable from consideration.

More generally, if we can find a partition of the state-space of a subtask m such that all the transitions from states in one block of the partition have the same probability and expected reward to transitioning to each of the other blocks, we can reduce the subtask to one where the states become the blocks. The solution of this reduced subtask is the solution of the original subtask. This is the notion of *stochastic bisimulation homogeneity* for MDP model minimisation (Dean and Givan, 1997). The computational complexity of solving an MDP, being polynomial in $|S|$, will be reduced depending on the coarseness of the partition. The state abstraction can be substantial. Consider the subtask that involves walking. This skill is mostly independent of geography, dress, and objects in the world. If we could not abstract the walking subtask, we would be condemned to re-learn to walk every-time any one of these variables changed value.

Formally, if $P = \{B_1, \ldots, B_n\}$ is a partition of the state-space of an SMDP and for each: B_i, $B_j \in P$, $a \in A$, p, $q \in B_i$, number of time-steps to termination is N, then if and only if

$$\sum_{r \in B_j} T(p,a,r,N) = \sum_{r \in B_j} T(q,a,r,N)$$
$$\sum_{r \in B_j} R(p,a,r,N) = \sum_{r \in B_j} R(q,a,r,N)$$

the subtask can be minimised to one in which S is replaced by P,

$$T(B_i,a,B_j,N) = \sum_{r \in B_j} T(p,a,r,N)$$
$$R(B_i,a,B_j,N) = \sum_{r \in B_j} R(p,a,r,N)$$

Model minimisation is exemplified by the room-leaving abstract actions in the four-room task. For these subtasks the particular room is irrelevant and the room variable can be eliminated. In this case, the blocks of the partition of the total state-space comprise states with the same position-in-room value.

This type of state abstraction is used by Boutilier et al (1995) who exploit independence between variables in dynamic Bayes nets. It is introduced by Dietterich (2000) for Max Node and Leaf Irrelevance in the MAXQ graph (Section 9.3.3). Ravindran and Barto (2003) extend this form of model minimisation to state-action symmetry couched in the algebraic formalism of morphisms. They state conditions under which the minimised model is a homomorphic image and hence transitions and rewards commute.

Some states may not be reachable for some sub-tasks in the task-hierarchy. This Shielding condition (Dietterich, 2000) is another form of elimination of irrelevant variables, in this case resulting from the structure of the task-graph.

Funnelling

Funnelling (Dietterich, 2000) is a type of state abstraction where abstract actions move the environment from a large number of initial states to a small number of resulting states. The effect is exploited for example by Forestier and Varaiya (1978) in plant control and by Dean and Lin (1995) to reduce the size of the MDP.

Funnelling can be observed in the four-room task. Room-leaving abstract actions move the agent from any position in a room to the state outside the respective doorway. Funnelling allows the four-room task to be state-abstracted at the root node to just 4 states because, irrespective of the starting position in each room, the abstract actions have the property of moving the agent to another room state.

9.2.5 Value-Function Decomposition

The task-hierarchy for the four-room task in Figure 9.2 has two successful higher-level policies that will find a path out of the house from the starting position in the South-East room. They are to leave rooms successively either North-West-North or West-North-North. The latter is the shorter path, but the simple hierarchical reinforcement learner in Section 9.1 cannot make this distinction.

What is needed is a way to decompose the value function for the whole problem over the task-hierarchy. Given this decomposition we can take into account the rewards within a subtask when making decision at higher levels.

To see this, consider the first decision that the agent in the four-room task (Figure 9.1) has to make. Deciding whether it is better to invoke a North or West room-leaving abstract action does not just depend on the agent's room state – the South-West room in this case. It also depends on the current position in the room. In this case we should aim for the doorway that is closer, as once we have exited the room, the distance out of the house is the same for either doorway. We need both the room-state and the position-in-room state to decide on the best action.

The question now arises as to how to decompose the original value function given the task-hierarchy so that the optimal action can be determined in each state. This will be addressed by the MAXQ approach (Dietterich, 2000) in Section 9.3.3 with a two part decomposition of the value function for each subtask in the task-hierarchy. The two parts are value to termination of the abstract action and the value to termination of the subtask.

Andre and Russell (2002) have introduced a three part decomposition of the value function by including the component of the value to complete the problem after subtask terminates. The advantage of this approach is that context is taking into account, making solutions hierarchically optimal (see Section 9.2.6), but usually at the expense of less state abstraction. The great benefit of decomposition is state-abstraction opportunities.

9.2.6 Optimality

We are familiar with the notion of an optimal policy for an MDP from Chapter 1. Unfortunately, HRL cannot guarantee in general that a decomposed problem will necessarily yield the optimal solution. It depends on the problem and the quality of the decomposition in terms of the abstract actions available and the structure of the task hierarchy or the partial program.

Hierarchically Optimal

Policies that are *hierarchically optimal* are ones that maximise the overall value function consistent with the constraints imposed by the task-hierarchy. To illustrate this concept assume that the agent in the four-room task moves with a 70% probability in the intended direction, but slips with a 10% probability each of the other three directions. Executing the hierarchical optimal policy for the task-hierarchy shown in Figure 9.2 may not be optimal. The top level policy will choose the West room-leaving action to leave the room by the nearest doorway. If the agent should find itself near the North doorway due to stochastic drift, it will nevertheless stubbornly persist to leave by the West doorway as dictated by the policy of the West room-leaving abstract action. This is suboptimal. A "flat" reinforcement learner executing the optimal policy would change its mind and attempt to leave the South-West room by the closest doorway.

The task-hierarchy could once again be made to yield the optimal solution if we included an abstract action that was tasked to leave the room by either the West or North doorway.

Recursively Optimal

Dietterich (2000) introduced another form of optimality - *recursive optimality*. In this type of optimality sub-task policies to reach goal terminal states are *context free* ignoring the needs of their parent tasks. This formulation has the advantage that sub-tasks can be re-used in various contexts, but they may not therefore be optimal in each situation.

A recursively optimal solution cannot be better than a hierarchical optimal solution. A hierarchical optimality can be arbitrarily worse than the globally optimal solution and a recursive optimal solution can be arbitrarily worse than a hierarchical optimal solution. It depends on the design of the task hierarchy by the designer.

Hierarchical Greedy Optimality

As we have seen above the stochastic nature of MDPs means that the condition under which an abstract action is appropriate may have changed after the action's invocation and that another action may become a better choice because of the inherent stochasticity of transitions (Hauskrecht et al, 1998). A subtask policy proceeding to termination may be sub-optimal. By constantly interrupting the sub-task a better sub-task may be chosen. Dietterich calls this "polling" procedure *hierarchical greedy execution*. While this is guaranteed to be no worse than a hierarchical optimal solution or a recursively optimal solution and may be considerably better, it still does not provide any global optimality guarantees.

Hauskrecht et al (1998) discuss decomposition and solution techniques that make optimality guarantees, but unfortunately, unless the MDP can be decomposed into weakly coupled smaller MDPs the computational complexity is not necessarily reduced.

9.3 Approaches to Hierarchical Reinforcement Learning (HRL)

HRL continues to be an active research area. We will now briefly survey some of the work in HRL. Please also see Barto and Mahadevan (2003) for a survey of advances in HRL, Si et al (2004) for hierarchical decision making and approaches to concurrency, multi-agency and partial observability, and Ryan (2004) for an alternative treatment and motivation underlying the movement towards HRL.

Historical Perspective

Ashby (1956) talks about amplifying the regulation of large systems in a series of stages, describing these hierarchical control systems “ultra-stable”. HRL can be viewed as a gating mechanisms that, at a higher levels, learn to switch in appropriate and more reactive behaviours at lower levels. Ashby (1952) proposed such a gating mechanism for an agent to handle recurrent situations.

The subsumption architecture (Brooks, 1990) works along similar lines. It decomposes complicated behaviour into many simple tasks which are organised into layers, with higher level layers becoming increasingly abstract. Each layer’s goal subsumes that of its child layers. For example, obstacle avoidance is subsumed by a foraging for food parent task and switched in when an obstacle is sensed in the robot’s path.

Watkins (1989) discusses the possibility of hierarchical control consisting of coupled Markov decision problems at each level. In his example, the top level, like the navigator of an 18th century ship, provides a kind of gating mechanism, instructing the helmsman on which direction to sail. Singh (1992) developed a gating mechanism called Hierarchical-DYNA (H-DYNA). DYNA is a reinforcement learner that uses both real and simulated experience after building a model of the reward and state transition function. H-DYNA first learns elementary tasks such as to navigate to specific goal locations. Each task is treated as an abstract action at a higher level of control.

The “feudal” reinforcement learning algorithm (Dayan and Hinton, 1992) emphasises another desirable property of HRL - state abstraction. The authors call it “information hiding”. It is the idea that decision models should be constructed at coarser granularities further up the control hierarchy.

In the hierarchical distance to goal (HDG) algorithm, Kaelbling (1993) introduces the important idea of composing the value function from distance components along the path to a goal. HDG is modelled on navigation by landmarks. The idea is to learn and store local distances to neighbouring landmarks and distances between any two locations within each landmark region. Another function is used to store shortest-distance information between landmarks as it becomes available from local distance functions. The HDG algorithm aims for the next nearest landmark on the way to the goal and uses the local distance function to guide its primitive actions. A higher level controller switches lower level policies to target the next neighbouring landmark whenever the agent enters the last targeted landmark region. The agent therefore rarely travels through landmarks but uses them as points to aim for on its way to the goal. This algorithm provided the inspiration for both the MAXQ value function decomposition (to be discussed in more detail in Section 9.3.3) and value function improvement with hierarchical greedy execution (Paragraph 9.2.6).

Moore et al (1999) have extended the HDG approach with the "airport-hierarchy" algorithm. The aim is to find an optimal policy to move from a start state to a goal state where both states can be selected from the set of all states. This *multi-goal* problem requires an MDP with $|S|^2$ states. By learning a set of goal-state reaching abstract actions with progressively smaller "catchment" areas, it is possible to approximate an optimal policy to any goal-state using, in the best case, only order $N \log N$ states.

Abstract actions may be included in hybrid hierarchies with other learning or planning methods mixed at different levels. For example, Ryan and Reid (2000) use a hybrid approach (RL-TOP) to constructing task-hierarchies. In RL-TOP, planning is used at the abstract level to invoke reactive planning operators, extended in time, based on teleo-reactive programs (Nilsson, 1994). These operators use reinforcement learning to achieve their post-conditions as sub-goals.

Three paradigms predominate HRL. They are: *Options*, a formalisation of abstract actions; HAMQ, a partial program approach, and MAXQ value-function decomposition including state abstraction.

9.3.1 Options

One formalisation of an abstract action is the idea of an *option* (Sutton et al, 1999)[4].

Definition 9.3.1. *An **option** (in relation to an MDP $\langle S,A,T,R\rangle$) is a triple $\langle I,\pi,\beta\rangle$ in which $I \subseteq S$ is an initiation set, $\pi : S \times A \to [0,1]$ is a policy, and $\beta : S^+ \to [0,1]$ is a termination condition.*

An option can be taken in state $s \in S$ of the MDP if $s \in I$. This allows option initiation to be restricted to a subset of S. Once invoked the option takes actions as determined by the stochastic policy π. Options terminate stochastically according to function β that specifies the probability of termination in each state. Many tasks are *episodic*, meaning that they will eventually terminate. When we include a single

[4] Abstract actions can be formalised in slightly different ways. One such alternative is used with the HEXQ algorithm to be described later.

abstract terminal state in the definition of the state space of an MDP we write S^+ to denote the total state space including the terminal abstract state. If the option is in state s it will terminate with probability $\beta(s)$, otherwise it will continue execution by taking the next action a with probability $\pi(s,a)$.

When option policies and termination depend on only the current state s, options are called *Markov options*. We may wish to base a policy on information other then the current state s. For example, if we want the option to time-out after a certain number of time-steps we could add a counter to the state space. In general the option policy and termination can depended on the entire history sequence of states, actions and rewards since the option was initiated. Options of this sort are called *semi-Markov* options. We will later discuss examples of options that are formulated by augmenting an MDP by a stochastic finite state machine, such as a program.

It is easy to see that a primitive action is an option. A primitive action a is equivalent to the option $\langle I = s, \pi(s,a) = 1.0, \beta(s) = 1\rangle$ for all $s \in S$. It is possible to unify the set of options and primitive actions and take set A to be their union.

Just as with policies over primitive actions, we can define policies over options. The option policy $\pi : S \times A \to [1,0]$ is a function that selects an option with probability $\pi(s,a)$, $s \in S$, $a \in A$. Since options select actions, and actions are just special kinds of options, it is possible for options to select other options. In this way we can form hierarchical structures to an arbitrary depth.

9.3.2 HAMQ-Learning

If a small hand-coded set of abstract actions are known to help solve an MDP we may be able to learn a policy with much less effort than it would take to solve the problem using primitive actions. This is because abstract actions can skip over large parts to the state space terminating in a small subset of states. The original MDP may be made smaller because now we only need to learn a policy with the set of abstract actions over a reduced state space.

In the *hierarchy of abstract machines* (HAM) approach to HRL the designer specifies abstract actions by providing stochastic finite state automata called *abstract machines* that work jointly with the MDP (Parr and Russell, 1997). This approach explicitly specifies abstract actions allowing users to provide background knowledge, more generally in the form of partial programs, with various levels of expressivity.

An abstract machine is a triple $\langle \mu, I, \delta \rangle$, where μ is a finite set of machine states, I is a stochastic function from states of the MDP to machine states that determines the initial machine state, and δ is a stochastic next-state function, mapping machine states and MDP states to next machine states. The machine states are of different types. Action-states specify the action to be taken given the state of the MDP to be solved. Call-states execute another machine as a subroutine. Choice-states non-deterministically select the next machine state. Halt-states halt the machine.

The parallel action of the abstract machine and the MDP yields a discrete-time higher-level SMDP. Choice-states are states in which abstract actions can be initiated. The abstract machine's action-states and choice states generate a sequence of actions that amount to an abstract action policy. If another choice-state is reached before the current executing machine has terminated, this is equivalent to an abstract action selecting another abstract action, thereby creating another level in a hierarchy. The abstract action is terminated by halt-states. A judicious specification of a HAM may reduce the set of states of the original MDP associated with choice-points.

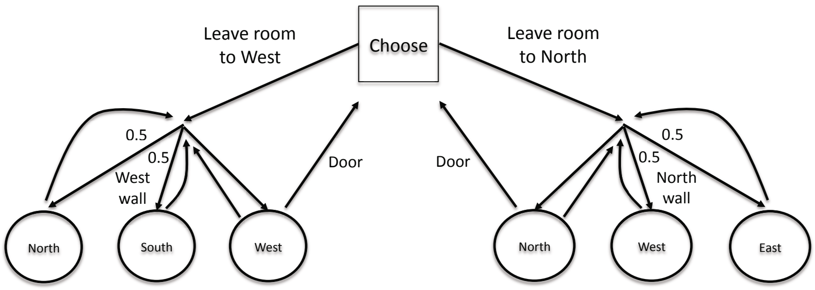

Fig. 9.3 An abstract machine (HAM) that provides routines for leaving rooms to the West and North of the house in Figure 9.1. The choice is between two abstract actions. One to leave the house through a West doorway, the other through a North doorway. If the West doorway is chosen, the abstract action keeps taking a West primitive action until it has either moved though the door and terminates or reaches a West wall. If at a West wall it takes only North and South primitive actions at random until the West wall is no longer observable, i.e. it has reached a doorway, whereupon it moves West through the doorway and terminates.

We can illustrate the operation of a HAM using the four-room example. The abstract machine in Figure 9.3 provides choices for leaving a room to the West or the North. In each room it will take actions that move the agent to a wall, and perform a random walk along the wall until it finds the doorway. Primitive rewards are summed between choice states. In this example we assume the agent's initial position is as shown in Figure 9.1. Only five states of the original MDP are states of the SMDP. These states are the initial state of the agent and the states on the other side of doorways where the abstract machine enters choice states. Reinforcement learning methods update the value function for these five states in the usual way with rewards accumulated since the last choice state.

Solving the SMDP yields an optimal policy for the agent to leave the house subject to the program constraints of the abstract machine. The best policy consists of the three abstract actions – sequentially leaving a room to the West, North and North again. In this case it is not a globally optimal policy because a random walk along walls to find a doorway is inefficient. The HAM approach is predicated on engineers and control theorists being able to design good controllers that will realise specific behaviours. HAMs are a way to partially specify procedural knowledge to transform an MDP to a reduced SMDP.

Programmable HRL

In the most general case a HAM can be a program that executes any computable mapping of the agent's complete sensory-action history (Parr, 1998).

Andre and Russell extended the HAM approach by introducing more expressible agent design languages for HRL – Programmable HAM (PHAM) (Andre and Russell, 2000) and ALisp, a Lisp-based high-level partial programming language (Andre and Russell, 2002).

Golog, is a logic programming language for agents that allow agents to reason about actions, goals, perceptions, other agents, etc., using situation calculus (Levesque et al, 1997). It has been extended as a partial program by embedding MDPs. Examples include *decision theoretic* Golog (DTGolog) (Boutilier et al, 2000) and Readylog (Ferrein and Lakemeyer, 2008) using the options framework.

In each case the partial program allows users to provide background knowledge about the problem structure using special choice-point routines that implement non-deterministic actions for the agent to learn the best action to take from experience. Programs of this kind leverage the expressiveness of the programming language to succinctly specify (and solve) an SMDP.

9.3.3 MAXQ

MAXQ is an approach to HRL where the value function is decomposed over the task hierarchy (Dietterich, 2000). It can lead to a compact representation of the value function and makes sub-tasks *context-free* or portable.

MAXQ abstract actions are crafted by classifying subtask terminal states as either goal states or non-goal states. Using disincentives for non-goal states, policies are learnt for each subtask to encourage termination in goal states. This termination predicate method may introduce an additional source of sub-optimality in the MDP as "pseudo" rewards can distort the subtask policy.

A key feature of MAXQ is that it represents the value of a state as a decomposed sum of sub-task completion values plus the expected reward for the immediate primitive action. A completion value is the expected (discounted) cumulative reward to complete the sub-task after taking the next abstract action.

We will derive the hierarchical decomposition following Dietterich (2000) and extend the above SMDP notation by including explicit reference to a particular subtask m. Equation 9.3 for subtask m becomes:

$$Q^{\pi}(m,s,a) = \sum_{s',N} T(m,s,a,s',N))[R(m,s,a,s',N) \\ +\gamma^N Q^{\pi}(m,s',\pi(s'))\,] \tag{9.4}$$

Abstract action a for subtask m invokes a child subtask m_a. The expected value of completing subtask m_a is expressed as $V^{\pi}(m_a,s)$. The *hierarchical policy*, π, is a set of policies, one for each subtask.

The *completion function*, $C^{\pi}(m,s,a)$, is the expected discounted cumulative reward after completing abstract action a, in state s in subtask m, discounted back to the point where a begins execution.

$$C^{\pi}(m,s,a) = \sum_{s',N} T(m,s,a,s',N))\gamma^N Q^{\pi}(m,s',\pi(s'))$$

The Q function for subtask m (Equation 9.4) can be expressed recursively as the value for completing the subtask that a invokes, m_a, plus the completion value to the end of subtask m.

$$Q^{\pi}(m,s,a) = V^{\pi}(m_a,s) + C^{\pi}(m,s,a) \tag{9.5}$$

The value of completing subtask m_a depends on whether a is primitive or not. In the event that action a is primitive, $V^{\pi}(m_a,s)$ is the expected reward after taking action a in state s.

$$V^{\pi}(m_a,s) = \begin{cases} Q^{\pi}(m_a,s,\pi(s)) & \text{if } a \text{ is abstract} \\ \sum_{s'} T(s,a,s'))R(s,a,s') & \text{if } a \text{ is primitive} \end{cases} \tag{9.6}$$

If the path of activated subtasks from root subtask m_0 to primitive action m_k is $m_0, m_1, \ldots, m_k$, and the hierarchical policy specifies that in subtask m_i, $\pi(s) = a_i$, then recursive Equations 9.5 and 9.6 decompose the value function $Q^{\pi}(m_0,s,\pi(s))$ as

$$Q^{\pi}(m_0,s,\pi(s)) = V^{\pi}(m_k,s) \;\; +C^{\pi}(m_{k-1},s,a_{k-1}) + \ldots \\ +C^{\pi}(m_1,s,a_1) + C^{\pi}(m_0,s,a_0)$$

To follow an optimal greedy policy given the hierarchy, the decomposition Equation 9.6 for the subtask implementing abstract action a is modified to choose the best action a, i.e. $V^*(m_a,s) = max_{a'} Q^*(m_a,s,a')$. The introduction of the *max* operator means that we have to perform a complete search through all the paths in the task-hierarchy to determine the best action. Algorithm 18 performs such a depth-first search and returns both the value and best action for subtask m in state s.

As the depth of the task-hierarchy increases, this exhaustive search can become prohibitive. Limiting the depth of the search is one way to control its complexity (Hengst, 2004). To plan an international trip, for example, the flight and airport-transfer methods need to be considered, but optimising which side of the bed to get out of on the way to the bathroom on the day of departure can effectively be ignored for higher level planning.

```
Require: $V(m,s)$ for primitive actions $m$, abstract actions $A_m$, $C(m,s,j)\ \forall j \in A_m$
1: if $m$ is a primitive action then
2:     return $\langle V(m,s),m\rangle$
3: else
4:     for each $j \in A_m$ do
5:         $\langle V(j,s),a_j\rangle = Evaluate(j,s)$
6:     $j^{greedy} = argmax_j\,[V(j,s)+C(m,s,j)]$
7:     return $\langle V(j^{greedy},s),a_{j^{greedy}}\rangle$
```

Algorithm 18. $Evaluate(m,s)$ [Dietterich (2000)]

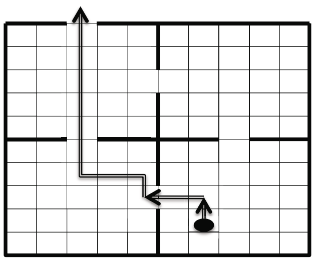

Fig. 9.4 The completion function components of the decomposed value function for the agent following an optimal policy for the four-room problem in Figure 9.1. The agent is shown as a solid black oval at the starting state.

For the four-room task-hierarchy in Figure 9.2, the decomposed value of the agent's state has three terms determined by the two levels in the task-hierarchy plus a primitive action. With the agent located in the state shown in Figure 9.4 by a solid back oval, the optimal value function for this state is the cost of the shortest path out of the house. It is composed by adding the expected reward for taking the next primitive action to the North, completing the lower-level sub-task of leaving the room to the West, and completing the higher-level task of leaving the house.

MAXQ Learning and Execution

Given a designer specified task-hierarchy, Algorithm 19 performs the equivalent of on-line Q-Learning (Chapter 1) for completion functions for each of the subtask SMDPs in a task-hierarchy. If the action is primitive it learns the expected reward for that action in each state (Line 3). For abstract actions the completion function following the abstract action is updated (Line 12). The learning rate parameter α is gradually reduced to zero in the limit. Algorithm 19 is a simplified version of the

MAXQ algorithm. For an extended version of the Algorithm, one that accelerates learning and distinguishes goal from non-goal terminal states using *pseduo-rewards*, please see (Dietterich, 2000). Algorithm 20 initiates the MAXQ process that proceeds to learn and execute the task-hierarchy.

```
Require: $V, C, A$, learning rate $\alpha$
 1: if $m$ is a primitive action then
 2:     execute $m$, receive reward $r$, and observe the result state $s'$
 3:     $V(m,s) \leftarrow (1-\alpha)V(m,s) + \alpha r$
 4:     return 1
 5: else
 6:     $count = 0$
 7:     while $m$ has not terminated do
 8:         choose an exploration action $a$
 9:         $N = MAXQ(a,s)$
10:         observe result state $s'$
11:         $\langle V(j^{greedy},s'),a_{j^{greedy}}\rangle \leftarrow Evaluate(m,s')$
12:         $C(m,s,a) \leftarrow (1-\alpha)C(m,s,a) + \alpha\gamma^N V(j^{greedy},s')$
13:         $count \leftarrow count + N$
14:         $s = s'$
15:         return $count$
```

Algorithm 19. $MAXQ(m,s)$ [Dietterich (2000)]

```
Require: root node $m_0$, starting state $s_0$, $V, C$
 1: initialise $V(m,s)$ and $C(m,s,a)$ arbitrarily
 2: $MAXQ(m_0,s_0)$
```

Algorithm 20. Main Program MAXQ

HRL Applied to the Four-Room Task

We will now put all the above ideas together and show how we can learn and execute the four-room task in Figure 9.1 when the agent can start in any state. The designer of the task-hierarchy in Figure 9.5 has recognised several state abstraction opportunities. Recall that the state is described by the tuple $(room, position)$.

The agent can leave each room by one of four potential doorways to the North, East, South, or West, and we need to learn a separate navigation strategy for each. However, because the rooms are identical, the room variable is irrelevant for intra-room navigation.

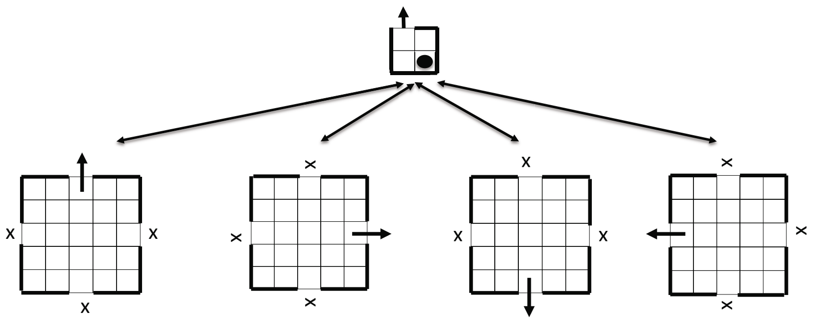

Fig. 9.5 A task task-hierarchy for the four-room task. The subtasks are room-leaving actions and can be used in any of the rooms. The parent-level root subtask has just four states representing the four rooms. “X” indicates non-goal terminal states.

We also notice that room-leaving abstract actions always terminate in one state. A room leaving abstract action is seen to “funnel” the agent through the doorway. It is for this reason that the position-in-room states can be abstracted away and the room state retained at the root level and only room leaving abstract actions are deployed at the root subtask. This means that instead of requiring 100 states for the root subtask we only require four, one for each room. Also, we only need to learn 16 (4 states $\times$ 4 actions) completion functions, instead of 400.

To solve the problem using the task-hierarchy with Algorithm 19, a main program initialises expected primitive reward functions $V(\cdot,\cdot)$ and completion functions $C(\cdot,\cdot,\cdot)$ arbitrarily, and calls function $MAXQ$ at the root subtask in the task-hierarchy for starting state s_0, i.e. $MAXQ(root,s_0)$. $MAXQ$ uses a Q-Learning like update rule to learn expected rewards for primitive actions and completion values for all subtasks.

With the values converged, α set to zero, and exploration turned off, $MAXQ$ (Algorithm 18) will execute a recursively optimal policy by searching for the shortest path to exit the four rooms. An example of such a path and its decomposed value function is shown in Figure 9.4 for one starting position.

9.4 Learning Structure

In most hierarchical reinforcement learning (HRL) applications the structure of the task-hierarchy or the partial program is provided as background knowledge by the designer. Automating the decomposition of a problem appears to be more difficult. It is desirable to have machines free designers from this task. Hierarchical structure requires knowledge of how a problem can best be decomposed and how to introduce state abstractions to balance complexity against loss of optimality.

Readers may be familiar with the mutilated checker-board problem showing that problem representation plays a large part in its solution (Gamow and Stern, 1958). In his seminal paper on six different representations for the missionaries and cannibals problem, Amarel (1968) demonstrated the possibility of making machine learning easier by discovering regularities and using them to formulating new representations. The choice of variables to represent states and actions in a reinforcement learning problem plays a large part in providing opportunities to decompose the problem.

Some researchers have tried to learn the hierarchical structure from the agent-environment interaction. Most approaches look for sub-goals or sub-tasks that try to partition the problem into near independent reusable sub-problems. Methods to automatically decompose problems include ones that look for sub-goal bottleneck or landmark states, and ones that find common behaviour trajectories or region policies.

Learning from the Bottom-Up

Automatic machine decomposition of complex tasks through interactions with the environment requires learning in stages. While a programmer can specify behaviour from the top down, learning seems to evolve from the bottom up.

Simon (1996) observed – "complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms". Clark and Thornton (1997) present a persuasive argument for modelling complex environments, namely, that it is necessary to proceed bottom-up and solve simple problems as intermediate representations. In their words,

> ... the underlying trick is always the same; to maximise the role of achieved representations, and thus minimise the space of subsequent search.

This is consistent with the principles advocated by Stone (1998) that include problem decomposition into multi-layers of abstraction, learning tasks from the lowest level to the highest in a hierarchy, where the output of learning from one layer feeds into the next layer. Utgoff and Stracuzzi (2002) point to the compression inherent in the progression of learning from simple to more complex tasks. They suggest a building block approach, designed to eliminate replication of knowledge structures. Agents are seen to advance their knowledge by moving their "frontier of receptivity" as they acquire new concepts by building on earlier ones from the bottom up. Their conclusion:

> "Learning of complex structures can be guided successfully by assuming that local learning methods are limited to simple tasks, and that the resulting building blocks are available for subsequent learning".

Some of the approaches use to learn abstract actions and structure include searching for common behaviour trajectories or common state region polices (Thrun and Schwartz, 1995; McGovern, 2002). Others look for bottleneck or landmark states

(Digney, 1998; McGovern, 2002; Menache et al, 2002). Şimşek and Barto (2004) use a relative novelty measure to identify sub-goal states. Interestingly Moore et al (1999) suggest that, for some navigation tasks, performance is insensitive to the position of landmarks and an (automatic) randomly-generated set of landmarks does not show widely varying results from ones more purposefully positioned.

9.4.1 HEXQ

We now describe one approach to automatic machine learning of hierarchic structure. The HEXQ (hierarchical exit Q function) approach is a series of algorithms motivated by MAXQ value-function decomposition and bottom-up structure learning. The algorithms rely on a finitely dimensioned (factored) base level state description. An underlying MDP is assumed but not known to the agent. The objective is to learn a task-hierarchy and find an optimal policy.

HEXQ constructs a hierarchy starting from base level states and primitive actions. It finds abstract actions in those parts of the state-space where some variables are irrelevant. It then recursively formulates reduced SMDPs at higher levels using the result-states from the "funnel" behaviour of the previously found abstract actions.

Learning Abstract Actions and the Task-Hierarchy

HEXQ searches for subspaces by exploring the transition and reward functions for the projected state space onto each state variable. Subspace states are included in the same block of a partition when transitions do not change the other variables and the transition and reward functions are independent of other variables. This is a stricter form of stochastic bisimulation homogeneity (Section 9.2.4), one where the context, in the guise of other than the projected variable, does not change. The associated state abstraction eliminates context variables from the subspace as they are irrelevant.

Whenever these condition are violated for a state transition, an *exit*, represented as a state-action pair, (s,a), is created. If exits cannot be reached from initial subspace states, the subspace is split and extra exits created. Creating exits is the mechanism by which subgoals are automatically generated.

This process partitions the projected state space for each variable. Each block of the partition forms a set of subtasks, one for each exit. The subtask is a SMDP with the goal of terminating on exit execution. HEXQ subtasks are like options, except that the termination condition is defined by a state-action pair. The reward on termination is not a part of the subtask, but is counted at higher levels in the task-hierarchy. This represents a slight reformulation of the MAXQ value function decomposition, but one that unifies the definition of the Q function over the task-hierarchy and simplifies the recursive equations.

Early versions of HEXQ learn a monolithic hierarchy after ordering state variables by their frequency of change (Hengst, 2002). In more recent versions, state variables are tackled in parallel (Hengst, 2008). The projected state-space for each variable is partitioned into blocks as before. Parent level states are new variables formed by taking the cross-product of block identifiers from the child variables. This method of parallel decomposition of the factored state-space generates *sequential (or multi-tasking)* actions, that invoke multiple child subtasks, one at a time. Sequential actions create *partial-order* task-hierarchies that have the potential for greater state abstraction. (Hengst, 2008).

The Four-room HEXQ Decomposition

For the four-room task, HEXQ is provided with the factored state variable pairs $(room, position)$. It starts by forming one module for each variable. The position module discovers one block with four exits. Exits are transitions that leave any room by one of the doorways. They are discovered automatically because it is only for these transitions that the room variable may change. The position module will formulate four room-leaving subtasks, one for each of the exits. The subtask policies can be learnt in parallel using standard off-policy Q-learning for SMDPs.

The room module learns a partition of the room states that consists of singleton state blocks, because executing a primitive action in any room may change the position variable value. For each block there are four exits. The block identifier is just the room state and is combined with the block identifier from the position module to form a new state variable for the parent module.

In this case the room module does not add any value in the form of state abstraction and could be eliminated with the room variable passed directly to the parent module as part of the input. Either way the parent module now represents abstract room states and invokes room-leaving abstract actions to achieve its goal. The machine generated task-hierarchy is similar to that shown in Figure 9.5.

Interestingly, if the four-room state is defined using coordinates (x,y), where x and y range in value from 0 to 9, instead of the more descriptive $(room,position)$, HEXQ will nevertheless create a higher level variable representing four rooms, but one that was not supplied by the designer. Each of the x and y variables are partitioned by HEXQ into two blocks representing the space inside each room. The cross-product of the block identifiers creates a higher-level variable representing the four rooms. Sequential abstract actions are generated to move East-West and North-South to leave rooms. In this decomposition the subspaces have a different meaning. They model individual rooms instead of a generic room (Hengst, 2008).

Summary of HEXQ

HEXQ automatically builds task-hierarchies from interactions with the environment assuming an underlying finite state factored MDP. It employs both variable elimination and funnel type state abstractions to construct a compact representation (Section 9.2.4). As one subtask is learnt for each exit, HEXQ solutions are hierarchically optimal (Section 9.2.6). HEXQ can use hierarchical greedy execution to try to improve on the hierarchically optimal solution, and the HEXQ policy converges to the globally optimal result for task-hierarchies where only the root-subtask has stochastic transition or reward functions. The introduction of parallel decomposition of variables and sequential actions allows HEXQ to create abstract state variables not supplied by the designer. The decompositions is guaranteed to be "safe" in the sense that the decomposed value function of any policy over the task-hierarchy is equivalent to the value function when that policy is executed with the original MDP.

9.5 Related Work and Ongoing Research

Hierarchical reinforcement learning (HRL) is not an isolated subfield. In combination with topics covered in other chapters in this book it has generated several contributions and presents ongoing research challenges. We briefly list some of this work.

Our above treatment of HRL assumes discrete time, but SMDPs are defined more generally with continuous-time temporally extended actions (Puterman, 1994). Ghavamzadeh and Mahadevan (2001) extend MAXQ to continuous-time models for both the discounted reward and average reward optimality models introduced in Chapter 1. Many real-world problems involve continuous variables. Ghavamzadeh and Mahadevan (2003) use a hierarchical policy gradient approach to tackle reinforcement learning problems with continuous state and action spaces. Jong and Stone (2009) combine function approximation and optimistic exploration to allow MAXQ to cope with large and even infinite state spaces.

We have seen how the elimination of irrelevant variables can lead to the implicit transfer of skills between different contexts. Hierarchical decomposition can play a significant role in *transfer learning* where experience gained in solving one task can help in solving a different, but related task. Konidaris and Barto (2007) build portable options that distinguishing between *agent-space* and *problem-space*. The former agent-centric representation is closely related to the notion of deictic representations (Agre and Chapman, 1987). Marthi et al (2007) use sample trajectories to learn MAXQ style hierarchies where the objective function measures how much planning or reinforcement learning is likely to be simplified on future similar environments by using this hierarchy. Mehta et al (2008a) transfer the value function between different SMPDs and find that transfer is especially effective in the hierarchical setting. Mehta et al (2008b) discover MAXQ task hierarchies by applying dynamic Bayesian network models to a successful trajectory from a source reinforcement learning task. An interesting conclusion is that transferring the knowledge about the structure of the task-hierarchy may be more important than transferring the value function. The latter can even hinder the learning on the new task as it has to unlearn the transferred policy. Taylor and Stone (2009) survey transfer learning in the reinforcement setting including abstract actions, state abstraction and hierarchical approaches. For effective knowledge transfer the two environments need to be "close enough". Castro and Precup (2010) show that transferring abstract actions is more successful than just primitive actions using a variant of the bisimulation metric.

State abstraction for MAXQ like task-hierarchies is hindered by discounted reward optimality models because the rewards after taking an abstract action are no longer independent of the time to complete the abstract action. Hengst (2007) has developed a method for decomposing discounted value functions over the task-hierarchy by concurrently decomposing multi-time models (Precup and Sutton, 1997). The twin recursive decomposition functions restore state-abstraction opportunities and allow problems with continuing subtasks in the task-hierarchy to be included in the class of problems that can be tackled by HRL.

Much of the literature in reinforcement learning involves one-dimensional actions. However, in many domains we wish to control several action variables simultaneously. These situations arise, for example, when coordinating teams of robots, or when individual robots have multiple degrees of articulation. The challenge is to decompose factored actions over a task-hierarchy, particularly if there is a chance that the abstract actions will interact. Rohanimanesh and Mahadevan (2001) use the options framework to show how actions can be parallelized with a SMPD formalism. Fitch et al (2005) demonstrate, using a two-taxi task, concurrent action task-hierarchies and state abstraction to scale problems involving concurrent actions. Concurrent actions require a subtask termination scheme (Rohanimanesh and Mahadevan, 2005) and the attribution of subtask rewards. Marthi et al (2005) extend partial programs (Section 9.2.3) to the concurrent action case using multi-threaded concurrent-ALisp.

When the state is not fully observable, or the observations are noisy the MDP becomes a *Partially Observable Markov Decision Problem* or POMDP (Chapter 12). Wiering and Schmidhuber (1997) decompose a POMDP into sequences of simpler subtasks, Hernandez and Mahadevan (2000) solve partially observable sequential decision tasks by propagating reward across long decision sequences using a memory-based SMDP. Pineau and Thrun (2002) present an algorithm for planning in structured POMDPs using an action based decomposition to partition a complex problem into a hierarchy of smaller subproblems. Theocharous and Kaelbling (2004) derive a hierarchical partially observable Markov decision problem (HPOMDP) from hierarchical hidden Markov models extending previous work to include multiple entry and exit states to represent the spatial borders of the sub-space.

New structure learning techniques continue to be developed. The original HEXQ decomposition uses a simple heuristic to determine an ordering over the state variables for the decomposition. Jonsson and Barto (2006) propose a Bayesian network model causal graph based approach – Variable Influence Structure Analysis (VISA) – that relates the way variables influence each other to construct the task-hierarchy. Unlike HEXQ this algorithm combines variables that influence each other and ignores lower-level activity. Bakker and Schmidhuber (2004)'s HASSLE algorithm discovers subgoals by learning how to transition between abstract states. In the process subgoal abstract actions are generalised to be reused or specialised to work in different parts of the state-space or to reach different goals. HASSLE is extended with function approximation by Moerman (2009). Strehl et al (2007) learn dynamic Bayesian network (DBN) structures as part of the reinforcement learning process using a factored state representation.

Mugan and Kuipers (2009) present a method for learning a hierarchy of actions in a continuous environment by learning a qualitative representation of the continuous environment and then find actions to reach qualitative states. Neumann et al (2009) learn to parameterize and order a set of motion templates to form abstract actions (options) in continuous time. Konidaris and Barto (2009) introduce a skill discovery method for reinforcement learning in continuous domains that constructs chains of skills leading to an end-of-task reward. The method is further developed to build skill trees faster from a set of sample solution trajectories Konidaris et al (2010).

Osentoski and Mahadevan (2010) extend automatic basis function construction to HRL. The approach is based on hierarchical spectral analysis of graphs induced on an SMDP's state space from sample trajectories. Mahadevan (2010) provides a brief review of more recent progress in general representation discovery. The review includes temporal and homomorphic state abstractions, with the latter generalised to representing value functions abstractly in terms of a basis functions.

9.6 Summary

Hierarchical Reinforcement Learning (HRL) decomposes a reinforcement learning problem into a hierarchy of sub-tasks such that higher-level parent-tasks invoke lower-level child tasks as if they were primitive actions. A decomposition may have multiple levels of hierarchy. Some or all of the sub-problems can themselves be reinforcement learning problems. When a parent-task is formulated as a reinforcement learning problem it is commonly formalised as a semi-Markov Decision Problem because its actions are child-tasks that persist for an extended period of time. The advantage of hierarchical decomposition is a reduction in computational complexity if the overall problem can be represented more compactly and reusable sub-tasks learned or provided independently. There is usually a trade-off between the reduction in problem complexity through decomposition and how close the solution of the decomposed problem is to optimal.

References

Agre, P.E., Chapman, D.: Pengi: an implementation of a theory of activity. In: Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI 1987, vol. 1, pp. 268–272. AAAI Press (1987)

Amarel, S.: On representations of problems of reasoning about actions. In: Michie, D. (ed.) Machine Intelligence, vol. 3, pp. 131–171. Edinburgh at the University Press, Edinburgh (1968)

Andre, D., Russell, S.J.: Programmable reinforcement learning agents. In: Leen, T.K., Dietterich, T.G., Tresp, V. (eds.) NIPS, pp. 1019–1025. MIT Press (2000)

Andre, D., Russell, S.J.: State abstraction for programmable reinforcement learning agents. In: Dechter, R., Kearns, M., Sutton, R.S. (eds.) Proceedings of the Eighteenth National Conference on Artificial Intelligence, pp. 119–125. AAAI Press (2002)

Ashby, R.: Design for a Brain: The Origin of Adaptive Behaviour. Chapman & Hall, London (1952)

Ashby, R.: Introduction to Cybernetics. Chapman & Hall, London (1956)

Bakker, B., Schmidhuber, J.: Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In: Proceedings of the 8-th Conference on Intelligent Autonomous Systems, IAS-8, pp. 438–445 (2004)

Barto, A.G., Mahadevan, S.: Recent advances in hiearchical reinforcement learning. Special Issue on Reinforcement Learning, Discrete Event Systems Journal 13, 41–77 (2003)

Bellman, R.: Adaptive Control Processes: A Guided Tour. Princeton University Press, Princeton (1961)

Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence, vol. 2, pp. 1104–1111. Morgan Kaufmann Publishers Inc., San Francisco (1995)

Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-theoretic, high-level agent programming in the situation calculus. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, pp. 355–362. AAAI Press (2000)

Brooks, R.A.: Elephants don't play chess. Robotics and Autonomous Systems 6, 3–15 (1990)

Castro, P.S., Precup, D.: Using bisimulation for policy transfer in mdps. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2010, vol. 1, pp. 1399–1400. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2010)

Clark, A., Thornton, C.: Trading spaces: Computation, representation, and the limits of uninformed learning. Behavioral and Brain Sciences 20(1), 57–66 (1997)

Dayan, P., Hinton, G.E.: Feudal reinforcement learning. In: Advances in Neural Information Processing Systems (NIPS), vol. 5 (1992)

Dean, T., Givan, R.: Model minimization in Markov decision processes. In: AAAI/IAAI, pp. 106–111 (1997)

Dean, T., Lin, S.H.: Decomposition techniques for planning in stochastic domains. Tech. Rep. CS-95-10, Department of Computer Science Brown University (1995)

Dietterich, T.G.: Hierarchical reinforcement learning with the MAXQ value function decomposition. Journal of Artificial Intelligence Research 13, 227–303 (2000)

Digney, B.L.: Learning hierarchical control structures for multiple tasks and changing environments. From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behaviour SAB (1998)

Ferrein, A., Lakemeyer, G.: Logic-based robot control in highly dynamic domains. Robot Auton. Syst. 56(11), 980–991 (2008)

Fitch, R., Hengst, B., šuc, D., Calbert, G., Scholz, J.: Structural Abstraction Experiments in Reinforcement Learning. In: Zhang, S., Jarvis, R.A. (eds.) AI 2005. LNCS (LNAI), vol. 3809, pp. 164–175. Springer, Heidelberg (2005)

Forestier, J., Varaiya, P.: Multilayer control of large Markov chains. IEEE Tansactions Automatic Control 23, 298–304 (1978)

Gamow, G., Stern, M.: Puzzle-math. Viking Press (1958)

Ghavamzadeh, M., Mahadevan, S.: Continuous-time hierarchial reinforcement learning. In: Proc. 18th International Conf. on Machine Learning, pp. 186–193. Morgan Kaufmann, San Francisco (2001)

Ghavamzadeh, M., Mahadevan, S.: Hierarchical policy gradient algorithms. In: Marine Environments, pp. 226–233. AAAI Press (2003)

Hauskrecht, M., Meuleau, N., Kaelbling, L.P., Dean, T., Boutilier, C.: Hierarchical solution of Markov decision processes using macro-actions. In: Fourteenth Annual Conference on Uncertainty in Artificial Intelligence, pp. 220–229 (1998)

Hengst, B.: Discovering hierarchy in reinforcement learning with HEXQ. In: Sammut, C., Hoffmann, A. (eds.) Proceedings of the Nineteenth International Conference on Machine Learning, pp. 243–250. Morgan Kaufmann (2002)

Hengst, B.: Model Approximation for HEXQ Hierarchical Reinforcement Learning. In: Boulicaut, J.-F., Esposito, F., Giannotti, F., Pedreschi, D. (eds.) ECML 2004. LNCS (LNAI), vol. 3201, pp. 144–155. Springer, Heidelberg (2004)

Hengst, B.: Safe State Abstraction and Reusable Continuing Subtasks in Hierarchical Reinforcement Learning. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 58–67. Springer, Heidelberg (2007)

Hengst, B.: Partial Order Hierarchical Reinforcement Learning. In: Wobcke, W., Zhang, M. (eds.) AI 2008. LNCS (LNAI), vol. 5360, pp. 138–149. Springer, Heidelberg (2008)

Hernandez, N., Mahadevan, S.: Hierarchical memory-based reinforcement learning. In: Fifteenth International Conference on Neural Information Processing Systems, Denver (2000)

Hutter, M.: Universal algorithmic intelligence: A mathematical top$\rightarrow$down approach. In: Artificial General Intelligence, pp. 227–290. Springer, Berlin (2007)

Jong, N.K., Stone, P.: Compositional models for reinforcement learning. In: The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (2009)

Jonsson, A., Barto, A.G.: Causal graph based decomposition of factored mdps. Journal of Machine Learning 7, 2259–2301 (2006)

Kaelbling, L.P.: Hierarchical learning in stochastic domains: Preliminary results. In: Proceedings of the Tenth International Conference Machine Learning, pp. 167–173. Morgan Kaufmann, San Mateo (1993)

Konidaris, G., Barto, A.G.: Building portable options: skill transfer in reinforcement learning. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence, pp. 895–900. Morgan Kaufmann Publishers Inc., San Francisco (2007)

Konidaris, G., Barto, A.G.: Skill discovery in continuous reinforcement learning domains using skill chaining. In: Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C.K.I., Culotta, A. (eds.) Advances in Neural Information Processing Systems, vol. 22, pp. 1015–1023 (2009)

Konidaris, G., Kuindersma, S., Barto, A.G., Grupen, R.: Constructing skill trees for reinforcement learning agents from demonstration trajectories. In: Advances in Neural Information Processing Systems NIPS, vol. 23 (2010)

Korf, R.E.: Learning to Solve Problems by Searching for Macro-Operators. Pitman Publishing Inc., Boston (1985)

Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: Golog: A logic programming language for dynamic domains. Journal of Logic Programming 31, 59–84 (1997)

Mahadevan, S.: Representation discovery in sequential descision making. In: 24th Conference on Artificial Intelligence (AAAI), Atlanta, July 11-15 (2010)

Marthi, B., Russell, S., Latham, D., Guestrin, C.: Concurrent hierarchical reinforcement learning. In: Proc. IJCAI 2005 Edinburgh, Scotland (2005)

Marthi, B., Kaelbling, L., Lozano-Perez, T.: Learning hierarchical structure in policies. In: NIPS 2007 Workshop on Hierarchical Organization of Behavior (2007)

McGovern, A.: Autonomous Discovery of Abstractions Through Interaction with an Environment. In: Koenig, S., Holte, R.C. (eds.) SARA 2002. LNCS (LNAI), vol. 2371, pp. 338–339. Springer, Heidelberg (2002)

Mehta, N., Natarajan, S., Tadepalli, P., Fern, A.: Transfer in variable-reward hierarchical reinforcement learning. Mach. Learn. 73, 289–312 (2008a), doi:10.1007/s10994-008-5061-y

Mehta, N., Ray, S., Tadepalli, P., Dietterich, T.: Automatic discovery and transfer of maxq hierarchies. In: Proceedings of the 25th International Conference on Machine Learning, ICML 2008, pp. 648–655. ACM, New York (2008b)

Menache, I., Mannor, S., Shimkin, N.: Q-Cut - Dynamic Discovery of Sub-goals in Reinforcement Learning. In: Elomaa, T., Mannila, H., Toivonen, H. (eds.) ECML 2002. LNCS (LNAI), vol. 2430, pp. 295–305. Springer, Heidelberg (2002)

Moerman, W.: Hierarchical reinforcement learning: Assignment of behaviours to subpolicies by self-organization. PhD thesis, Cognitive Artificial Intelligence, Utrecht University (2009)

Moore, A., Baird, L., Kaelbling, L.P.: Multi-value-functions: Efficient automatic action hierarchies for multiple goal mdps. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1316–1323. Morgan Kaufmann, San Francisco (1999)

Mugan, J., Kuipers, B.: Autonomously learning an action hierarchy using a learned qualitative state representation. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence, pp. 1175–1180. Morgan Kaufmann Publishers Inc., San Francisco (2009)

Neumann, G., Maass, W., Peters, J.: Learning complex motions by sequencing simpler motion templates. In: Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, pp. 753–760. ACM, New York (2009)

Nilsson, N.J.: Teleo-reactive programs for agent control. Journal of Artificial Intelligence Research 1, 139–158 (1994)

Osentoski, S., Mahadevan, S.: Basis function construction for hierarchical reinforcement learning. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2010, vol. 1, pp. 747–754. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2010)

Parr, R., Russell, S.J.: Reinforcement learning with hierarchies of machines. In: NIPS (1997)

Parr, R.E.: Hierarchical control and learning for Markov decision processes. PhD thesis, University of California at Berkeley (1998)

Pineau, J., Thrun, S.: An integrated approach to hierarchy and abstraction for pomdps. CMU Technical Report: CMU-RI-TR-02-21 (2002)

Polya, G.: How to Solve It: A New Aspect of Mathematical Model. Princeton University Press (1945)

Precup, D., Sutton, R.S.: Multi-time models for temporally abstract planning. In: Advances in Neural Information Processing Systems, vol. 10, pp. 1050–1056. MIT Press (1997)

Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Whiley & Sons, Inc., New York (1994)

Ravindran, B., Barto, A.G.: SMDP homomorphisms: An algebraic approach to abstraction in semi Markov decision processes. In: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003 (2003)

Rohanimanesh, K., Mahadevan, S.: Decision-theoretic planning with concurrent temporally extended actions. In: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, pp. 472–479. Morgan Kaufmann Publishers Inc., San Francisco (2001)

Rohanimanesh, K., Mahadevan, S.: Coarticulation: an approach for generating concurrent plans in Markov decision processes. In: ICML 2005: Proceedings of the 22nd international conference on Machine learning, pp. 720–727. ACM Press, New York (2005)

Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River (1995)

Ryan, M.R.K.: Hierarchical Decision Making. In: Handbook of Learning and Approximate Dynamic Programming. IEEE Press Series on Computational Intelligence. Wiley-IEEE Press (2004)

Reid, M.D., Ryan, M.: Using ILP to Improve Planning in Hierarchical Reinforcement Learning. In: Cussens, J., Frisch, A.M. (eds.) ILP 2000. LNCS (LNAI), vol. 1866, pp. 174–190. Springer, Heidelberg (2000)

Si, J., Barto, A.G., Powell, W.B., Wunsch, D.: Handbook of Learning and Approximate Dynamic Programming. IEEE Press Series on Computational Intelligence. Wiley-IEEE Press (2004)

Simon, H.A.: The Sciences of the Artificial, 3rd edn. MIT Press, Cambridge (1996)

Şimşek, O., Barto, A.G.: Using relative novelty to identify useful temporal abstractions in reinforcement learning. In: Proceedings of theTwenty-First International Conference on Machine Learning, ICML 2004 (2004)

Singh, S.: Reinforcement learning with a hierarchy of abstract models. In: Proceedings of the Tenth National Conference on Artificial Intelligence (1992)

Stone, P.: Layered learning in multi-agent systems. PhD, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1998)

Strehl, A.L., Diuk, C., Littman, M.L.: Efficient structure learning in factored-state mdps. In: Proceedings of the 22nd National Conference on Artificial Intelligence, vol. 1, pp. 645–650. AAAI Press (2007)

Sutton, R.S., Precup, D., Singh, S.P.: Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence 112(1-2), 181–211 (1999)

Taylor, M.E., Stone, P.: Transfer learning for reinforcement learning domains: A survey. Journal of Machine Learning Research 10(1), 1633–1685 (2009)

Theocharous, G., Kaelbling, L.P.: Approximate planning in POMDPS with macro-actions. In: Advances in Neural Information Processing Systems 16 (NIPS-2003) (2004) (to appear)

Thrun, S., Schwartz, A.: Finding structure in reinforcement learning. In: Tesauro, G., Touretzky, D., Leen, T. (eds.) Advances in Neural Information Processing Systems (NIPS), vol. 7. MIT Press, Cambridge (1995)

Utgoff, P.E., Stracuzzi, D.J.: Many-layered learning. In: Neural Computation. MIT Press Journals (2002)

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

Wiering, M., Schmidhuber, J.: HQ-learning. Adaptive Behavior 6, 219–246 (1997)