1 Hierarchical decision making

M.R.K. RYAN
University of New South Wales
Sydney, Australia

Divide at impera.
Divide and rule.

—Ancient Roman maxim

1.1 INTRODUCTION

Reinforcement learning [50], [25] is a field of machine learning dedicated to building algorithms which learn to control the interaction of an agent with an environment, to achieve particular goals. For over a decade, since the advent of TD(λ) [47] and Q-Learning [55], most of the work in reinforcement learning has been based on the Markov Decision Process (MDP) model. While these algorithms have strong theoretical convergence properties, they have often performed poorly in practice. Optimal policies can be found for simple toy problems, but the algorithms are often difficult to scale up to realistic control problems.

Part of the problem is that MDPs model a system in fine detail. In recent years there has been a move from Markov Decision Processes to Semi-Markov Decision Processes, in an attempt to build models at higher levels of abstraction. This abstraction provides a number of advantages which allow us to apply reinforcement learning to a significantly larger collection of problems.

This field is known as Hierarchical Reinforcement Learning (HRL) and has been the subject of rapid growth in the past few years. Many techniques have arisen, all based on similar foundations but taking them in different directions. This profusion of different approaches can hide both the commonalities shared by these algorithms, and the different orthogonal improvements each approach contains. Many approaches combine several improvements into a single algorithm, which need not intrinsically be tied together.

In order to clarify this situation, we divide this review into three sections. In the first section we outline the problems with the standard MDP model for reinforcement learning in greater detail. In the second section we present the theory behind Hier-

archical Reinforcement Learning and describe some of the different improvements it can offer. In the third section we describe some of the actual hierarchical reinforcement learning algorithms which have been published and try to show how they combine the different elements we have previously described.

This review tries to capture the motivations underlying the movement towards hierarchical reinforcement learning, and some of the prominent examples thereof. It is not exhaustive. For an alternative presentation, with several recent inventions that are omitted here, we refer the reader to Barto and Mahadevan's survey [4].

1.2 REINFORCEMENT LEARNING AND THE CURSE OF DIMENSIONALITY

Reinforcement learning models an agent interacting with an environment, trying to optimise its choice of action according to some reward criterion. The agent operates over a sequence of discrete time-steps $(t, t+1, t+2, \ldots)$. (One time-step indicates the duration of a single action. This may or may not correspond to a fixed unit of time in the real-world.)

At each step the agent observes the state of the environment s_t and selects an appropriate action a_t. Executing the action produces a change in the state of the environment to s_{t+1}. It is generally assumed that the sets of possible states $\mathcal{S}$ and available actions $\mathcal{A}$ are both finite. This is not always the case in practice, but it greatly simplifies the theory, so we shall follow this convention.

The mapping of states to actions is done by an internal policy π. The initial policy is arbitrarily chosen, generally random, and it is modified and improved based on the agent's experiences. Each experience $\langle s_t, a_t, s_{t+1} \rangle$ is evaluated according to some fixed reward function, yielding a reward $r_t \in \Re$. The agent's objective is to modify its policy to maximise its long-term reward. There are several possible definitions of "long-term reward" but the one most commonly employed is the *expected discounted return* given by:

$$\begin{aligned} R_t &= E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots\right\} \\ &= E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right\} \end{aligned} \tag{1.1}$$

where γ is the *discount rate* that specifies the relative weight of future rewards, with $0 \leq \gamma < 1$. Should the agent reach some terminal state s_T, then the infinite sum is cut short: all subsequent rewards $r_{T+1}, r_{T+2}, \ldots$ are considered to be zero. (Another, less commonly used, reward criterion is *average reward* [30], [42], [44].)

To ensure that this optimisation problem is well-founded, most reinforcement learning algorithms place a strong constraint on the structure of the environment. They assume that it operates as a *Markov Decision Process* (MDP) [38]. An MDP describes a process that has no hidden state or dependence on history. The outcomes of every action, in terms of state transition and reward, obey fixed probability distributions that depend only on the current state and the action performed.

Formally an MDP can be described as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a transition function and $R : \mathcal{S} \times \mathcal{A} \times \Re \to [0, 1]$ is a reward function with:

$$T(s'|s, a) = P(s_{t+1} = s' \mid s_t = s, a_t = a) \tag{1.2}$$
$$R(r|s, a) = P(r_t = r \mid s_t = s, a_t = a) \tag{1.3}$$

which respectively express the probability of ending up in state s' and receiving reward r after executing action a in state s. These probabilities must be independent of any criteria other than the values of s and a. This is called the *Markov Property*. An in-depth treatment of the theory of Markov Decision Processes can be found in any of [6], [7], [8], [20], [38], or [50].

Given this simplifying assumption, the best action to choose in any state depends on that state alone. This means that the agent's policy can be expressed as a purely reactive mapping of states to actions, $\pi : \mathcal{S} \to \mathcal{A}$. Furthermore every state s can be assigned a value $V^\pi(s)$ that denotes the expected discounted return if the policy π is followed:

$$V^\pi(s) = E\{R_t \mid \varepsilon(\pi, s, t)\} \tag{1.4}$$
$$= E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid \varepsilon(\pi, s, t)\right\} \tag{1.5}$$
$$= \int_{-\infty}^{+\infty} rR(r|s, a)\,\mathrm{d}r + \gamma \sum_{s' \in \mathcal{S}} T(s'|s, a)\, V^\pi(s') \tag{1.6}$$

where $\varepsilon(\pi, s, t)$ denotes the event of policy π being initiated in state s at time t. V^π called the *state value function* for policy π.

An *optimal policy* can now be simply defined as a policy $\pi^\star$ that maximises $V^\pi(s)$ for all states s. The Markov property guarantees that there is such a globally optimal policy [50] although it may not be unique. We define the *optimal state-value function* $V^\star(s)$ as being the state value function of the policy $\pi^\star$:

$$\begin{aligned} V^\star(s) &= V^{\pi^\star}(s) \\ &= \max_\pi V^\pi(s) \end{aligned} \tag{1.7}$$

We can also define an *optimal state-action value function* $Q^\star(s, a)$ in terms of $V^\star(s)$ as:

$$Q^\star(s, a) = E\{r_t + \gamma V^\star(s_{t+1}) \mid s_t = s, a_t = a\} \tag{1.8}$$

This function expresses the expected discounted return if action a is executed in state s and an optimal policy is followed thereafter. If such a function is known then an optimal policy can be extracted from it simply:

$$\pi^\star(s) = \arg\max_{a \in \mathcal{A}} Q^\star(s, a) \tag{1.9}$$

Thus the reinforcement learning problem can be transformed from learning an optimal policy $\pi^\star$ to learning the optimal state-action value function $Q^\star$. [1] This turns out to be a relatively straightforward dynamic programming problem, which could in principle be solved by conventional DP algorithms.

There is, however, one complication. Standard techniques such as value-iteration or policy-iteration [6] rely on us already having an accurate model of the underlying MDP. However, in reinforcement learning we assume that this model is not (initially) available. Our only source of information about the MDP is experimental interaction with the environment. Given this we can proceed in two ways: we can attempt to learn a model and then applied standard DP methods to construct a policy, or we can attempt to learn $Q^\star$ directly. These are called "model-based" and "model-free" approaches respectively. Much work has been done in both these areas, but model-free approaches tend to dominate. We perpetuate this bias in this chapter, focusing on model-free reinforcement learning, which is where most of the work in hierarchical learning has been taking place. We briefly summarise some of the model-based hierarchical learning towards the end of the chapter.

1.2.1 Q-Learning

Q-Learning [55] is widely regarded as the archetypal model-free reinforcement learning algorithm. It is an online incremental learning algorithm that learns an *approximate state-action value function* $Q(s, a)$ that converges to the optimal function $Q^\star$ in Equation 1.8 above (under certain conditions outlined below). It is a simple algorithm which avoids the complexities of modelling the functions R and T of the MDP by learning Q directly from its experiences. It has significant practical limitations, but is theoretically sound and has provided a foundation for many more complex algorithms. Pseudocode for this algorithm is given in Algorithm 1.

Algorithm 1 Watkin's Q-Learning

function Q-LEARNING
  $t \leftarrow 0$
  Observe state s_t
  while s_t is not a terminal state **do**
    Choose action $a_t \leftarrow \pi(s_t)$ according to an exploration policy
    Execute a_t
    Observe resulting state s_{t+1} and reward r_t
    $Q(s_t, a_t) \xleftarrow{\alpha} r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a)$
    $t \leftarrow t + 1$
  end while
end Q-LEARNING

[1] In practice it is only necessary for the agent to optimise its reward in the states it actually encounters. Complete optimal policies are not needed to do this as some states may never be reached. This is one way in which reinforcement learning differs from classical MDP solution methods.

The approximate Q-function is stored in a table. Its initial values may be arbitrarily chosen, typically they are all zero or else randomly assigned. At each time-step an action is performed according to the policy dictated by the current Q-function:

$$a_t = \pi(s_t) = \arg\max_{a \in \mathcal{A}} Q(s_t, a) \tag{1.10}$$

The result of executing this action is used to update $Q(s_t, a_t)$, according to the temporal-difference rule:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) Q(s_t, a_t) + \alpha (r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a)) \tag{1.11}$$

where α is a learning rate, $0 \leq \alpha \leq 1$.[2]

The approximate state-action value function Q is proven to converge to the optimal function $Q^\star$ (and hence π to $\pi^\star$) given certain technical restrictions on learning rates ($\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$) and the requirement that all state-action pairs continue to be updated indefinitely [56], [54], [23]. This second requirement means that in executing the learnt policy the agent must also do a certain proportion of non-policy actions for the purposes of exploration. Exploration is important in all the algorithms that follow. The simplest approach to exploration is the ε-greedy algorithm which simply takes an exploratory action with some small probability ε, and a policy action otherwise. A large number of more complex alternatives exist (see [53] for a summary).

[2]The expression in Equation 1.11 is somewhat cumbersome. There are two operations being described simultaneously which are not clearly differentiated. The first operation is the temporal-difference step, which estimates the value of $Q(s_t, a_t)$ as:

$$Q_{new} = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a)$$

This value is the input to the second operation, which updates the existing value of $Q(s_t, a_t)$ towards this target value, using an exponentially weighted rolling average with learning rate α:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) Q(s_t, a_t) + \alpha Q_{new}$$

To simplify the equations we shall henceforth use the short-hand notation:

$$X \xleftarrow{\alpha} Y$$

to indicate that the value of X is adjusted towards the target value Y via an exponentially weighted rolling average with decay factor α, that is:

$$X \leftarrow (1 - \alpha) X + \alpha Y$$

Thus Equation 1.11 shall be written as:

$$Q(s_t, a_t) \xleftarrow{\alpha} r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) \tag{1.12}$$

This is non-standard notation, originally introduced by Baird in [3] but not widely adopted. I believe it captures the important elements of the formula more clearly and concisely.

1.2.2 The curse of dimensionality

As stated above, Q-Learning is theoretically guaranteed to converge to an optimal policy, however the guarantee is only true in the limit, and in practical problems of any significant size it has often been found to be impracticably slow. Without doing a full analysis of the algorithm, we can observe certain factors which contribute to this failure.

To find an optimal policy, a Q-value must be learnt for every state-action pair. This means, first of all, that every such pair needs to be explored at least once. So convergence time is at best $O(|S|.|A|)$. Real-world problems typically have large multi-dimensional state spaces. $|S|$ is exponential in the number of dimensions, so each extra dimension added to a problem multiplies the time it takes.

Furthermore, states are generally only accessible from a handful of close neighbours, so the distance between any pair of states in terms of action steps also increases with the size and dimensionality of the space. Yet a change in the value of one state may have consequences for the policy in a far distant state. As information can only propagate from one state to another through individual state transitions, the further apart two states are, the longer it will take for this information to be propagated. Thus the diameter of the state space is an additional factor in the time required to reach convergence.

A general-purpose solution to this problem has not yet been found. There have been many attempts to represent the table of Q-values more compactly by using one variety of function approximator or another. These have met with mixed success. Sometimes the resulting state-abstraction has enabled the learning algorithm to learn an effective (not necessarily optimal) policy in times faster than without abstraction by an order of magnitude or more for a particular domain (eg. [52] [57] [5]), but no such approach has proven to be a general-purpose solution. What works well in one domain will fail spectacularly in another. Furthermore even the simplest and most conservative forms of function approximation have been shown to break the convergence proofs, causing Q-values to becoming wildly divergent in certain situations [3]. For a summary of such attempts, see [25].

As a result of these difficulties researchers have turned from seeking general-purpose to special-purpose solutions. It has been recognised that a number of the most successful applications of reinforcement learning have used significant task-specific background knowledge tacitly incorporated into the agent's representation of its states and actions. Focus is shifting towards creating an architecture by which this tacit information can become explicit and can be represented in a systematic way. The aim is to create systems that can benefit from the programmer's task-specific knowledge whilst maintaining desirable theoretical properties of convergence.

1.3 HIERARCHICAL REINFORCEMENT LEARNING IN THEORY

Significant attention has recently been given to hierarchical decomposition as a means to this end. "Hierarchical reinforcement learning" (HRL) is the name given to a class

of learning algorithms that share a common approach to scaling up reinforcement learning.

Hierarchical decomposition has always been a natural approach to problem solving. "Divide-and-conquer" has long been a familiar motto in computer science. A complex problem can often be solved by decomposing it into a collection of smaller problems. The smaller problems can be solved more easily in isolation, and then recombined into a solution for the whole. Inspiration for hierarchical reinforcement learning came partly from behaviour-based techniques for robot programming [11] [29] [32] and partly from the hierarchical methods used in symbolic planning [41] [27] [22] [26].

Hierarchical reinforcement learning accelerates learning by forcing a structure on the policies being learnt. The reactive state-to-action mapping of Q-learning is replaced by a hierarchy of temporally-abstract actions. These are actions that operate over several time-steps. Like a subroutine or procedure call, once a temporally abstract action is executed it continues to control the agent until it terminates, at which point control is restored to the main policy. These actions (variously called *subtasks*, *behaviours*, *macros*, *options*, *activities* or *abstract machines* depending on the particular algorithm in question) must themselves be further decomposed into one-step actions that the agent can execute. We shall henceforth refer to one-step actions as *primitive actions* and temporally-abstract actions as *behaviours*. Policies learnt using primitive actions alone shall be called *monolithic* to distinguish them from *hierarchical* or *behaviour-based* policies.

How does this decomposition aid us? There are two different ways. One, it allows us to limit the choices available to the agent, even to the point of hard-coding parts of the policy; and two, it allows us to specify local goals for certain parts of the policy. Different HRL algorithms implement these features in different ways. Some implement one and not the other. We shall postpone describing specific algorithms until Section 1.4, and for the moment present these features in more general terms, with the aid of an example.

1.3.1 A Motivating Example

Figure 1.1 shows an example environment we shall use to illustrate the concepts in this chapter. Imagine that the learning agent is a house-hold robot in a house with the layout shown. Its purpose is to fetch objects from one room to another. It is able to know its location with a precision as shown by the cells of the grid, and its primitive actions enable it to navigate from a cell to any of its eight neighbours, with a small probability of error.

If the robot is in the same cell as an object, it can pick it up and carry it. There are two objects in the environment that we are interested in. In the kitchen in the north-west corner of the map is a machine which dispenses a cup of coffee. In the second bedroom there is a book, also indicated on the map. The robot starts at its docking location in the study. Its goal will vary from example to example as we consider different aspects of HRL (and later, of planning).

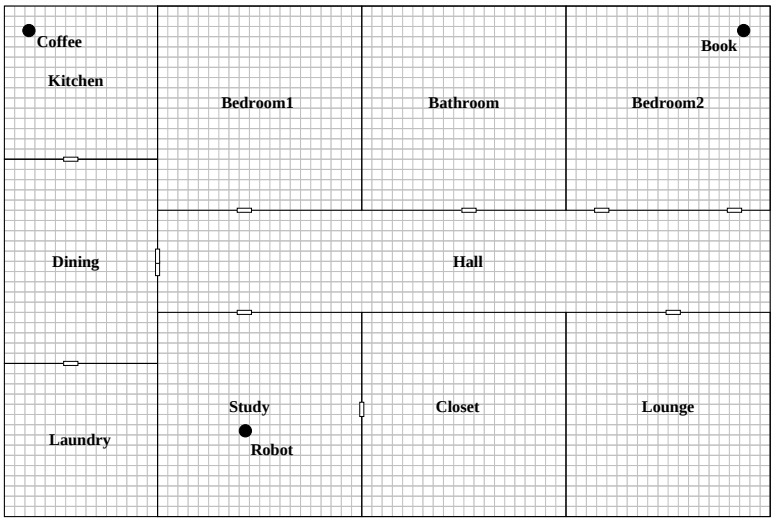


Fig. 1.1 An example environment.

In this environment we have 15,000 states (75×50 cells, with two different states for each object, depending on whether the robot is holding it or not[3]) and 9 primitive actions (each of the 8 compass directions, plus the pickup action). This is not in itself a complex environment, and most goals will be relatively easy to complete, but it is certainly one that can be made simpler by providing an appropriate set of behaviours.

The obvious behaviours to specify are: $\mathsf{Go}(\mathtt{Room1}, \mathtt{Room2})$ which moves the robot between two neighbouring rooms, and $\mathsf{Get}(\mathtt{Object}, \mathtt{Room})$ which moves towards and picks up the specified object when the robot is in the same room as it. We will discuss how these behaviours are implemented as we examine individual techniques.

1.3.2 Limiting the Agent's Choices

Since learning time is dominated by the number of state-action pairs that need to be explored, the obvious way to accelerate the process is to cut down the number of such pairs. Using background knowledge we can identify action choices which are plainly unhelpful and eliminate them from the set of possible policies. There is a variety of ways in which this can be done. Of course, such limitations must be applied with care, as overly broad limitations can prevent the agent from discovering policies that might otherwise be optimal.

[3]The positions of the book and the coffee machine in their respective rooms are also part of the state but since these positions are fixed, and the robot cannot drop these objects elsewhere, this information can be omitted.

Limiting Available Primitive Actions The simplest solution is to hard-code portions of the policy. Some or all of the internal operation of a behaviour can be written by hand by the system designer. This removes the need for the agent to do any kind of learning at all for significant portions of the state space, which will immediately improve performance. This assumes however that the system designer is able to do this. Part of the point of learning policies is to relieve the designer of the need to specify them, so this may be of limited use. Still, there are some situations in which simple behaviours might be wholly or partially specified, and algorithms have been designed to take advantage of this.

Less drastically, the internal policy of a behaviour could be learnt using only a limited subset of all available primitive actions. This is useful if the system designer knows that certain primitive actions are only suitable for particular behaviours and not for others. From the example, the $\mathsf{Go}()$ behaviours could reasonably be limited to only use the primitive actions which move the robot, and ignore the pickup action, which would be of no use to that behaviour.

Note that it is not strictly necessary to use hierarchy to limit primitive action choice. A simple extension to the MDP formalism above would allow the admissible action set to be a function of the state. The Q-Learning algorithm can be successfully applied to this problem with only a slight modification. However adding hierarchy allows us to specify these restrictions on the wider context of the subtask we currently executing, as well as the immediate state.

Limiting Available Behaviours Likewise, limits can be placed on which behaviours are available to the agent at different times. Behaviours are generally limited in scope, so they often can only be executed from a subset of all possible states. For instance the $\mathsf{Get}()$ behaviour can only be applied when the agent is it the same room as the target object. The set of states in which a behaviour B can be applied is called its *applicability space*, which we shall denote $\mathsf{B}.\mathrm{pre}$. Learning algorithms should not allow the agent to choose a behaviour in a state in which it is not applicable.

However this may not be limiting enough. As more ambitious problems are tackled, the repertoire of behaviours available to an agent is likely to become large, and many behaviours will have overlapping applicability spaces. It is of no use to limit the internal policy choices of behaviours if choosing between the behaviours becomes just as difficult.

To this end, most HRL algorithms implement some kind of *task hierarchy* to limit the choice of behaviours to those that are appropriate to the agent's situation. Consider the situation in the example environment when the robot is in hall with the goal of fetching both the book and the coffee. There are six applicable behaviours: $\mathsf{Go}(\texttt{hall}, \texttt{study})$, $\mathsf{Go}(\texttt{hall}, \texttt{dining})$, $\mathsf{Go}(\texttt{hall}, \texttt{bedroom1})$, $\mathsf{Go}(\texttt{hall}, \texttt{bathroom})$, $\mathsf{Go}(\texttt{hall}, \texttt{bedroom2})$, and $\mathsf{Go}(\texttt{hall}, \texttt{lounge})$. Of these, only two are appropriate: $\mathsf{Go}(\texttt{hall}, \texttt{dining})$, if the agent decides to fetch the coffee first, and $\mathsf{Go}(\texttt{hall}, \texttt{bedroom2})$ if the agent decides to fetch the book. Exploring the others is a waste of time. The system designer, who specified the behaviours, should realise this and incorporate it into the task hierarchy, limiting the agent's choices in this sit-

uation to one of these two behaviours. The larger an agent's repertoire of behaviours becomes, the more critical this kind of background knowledge.

Committing to Behaviours Finally, choices are limited by requiring long-term *commitment* to a behaviour. It is conceivable that a learning algorithm could be written which implemented hard-coded behaviours but allowed the agent to choose a different behaviour on every time step. Such an algorithm would hardly be any better than learning a primitive policy directly, and could easily be worse. Long-term commitment to behaviours has two benefits. First, a single behaviour can traverse a long sequence of states in a single "jump", effectively reducing the diameter of the state-space and propagating rewards more quickly. In the grid-world, for example, fetching both the coffee and the book takes 126 primitive actions, but can be done with a sequence of just 10 behaviours.

Second, a behaviour can "funnel" the agent into a particular set of terminating states. These states are then the launching points for new behaviours. If no behaviour ever terminates in a given state, then no policy needs to be learnt for that state. Again, referring to the grid-world, each Go() behaviour terminates in one of the six cells surrounding a doorway, in one of four possible configuration of what the robot is holding. There are 10 doors, so this yields 240 states. Each Get() behaviour terminates in the same location as the target object with 2 possible configurations of what the agent is holding, yielding a further 4 states. Plus 1 starting state gives a total of 245 states in which the agent needs to learn to choose a behaviour, out of a possible 15,000. This is a significant reduction in the size of the policy-space and will result in much faster learning.

Flexible limitations Limiting the policy space in this fashion will clearly have an effect on optimality. If the optimal policy does not fit the hierarchical structure, then any policy produced by a hierarchical reinforcement learner will be sub-optimal. This may well be satisfactory, but if not, it is possible to some degree to have the best of both worlds by imposing structure on the policy during the early phase of learning and relaxing it later. This allows the agent to learn a near-to-optimal policy quickly and then refine it to optimality in the long-term. Such techniques shall be described in more detail in Section 1.5.

1.3.3 Providing Local Goals

So far we have assumed that all choices the agent makes, at any point in the hierarchy, are made to optimise the one global reward function. Such a policy is said to be *hierarchically optimal* [16]. A hierarchically optimal policy is the best possible policy within the confines of the hierarchical structure imposed by the system designer.

Hierarchical optimality, however, contradicts part of the intuition of behaviour-based decomposition of problems. The idea that a problem can be decomposed into several independent subparts which can be solved separately and recombined no longer holds true. The solution to each subpart must be made to optimise the whole

policy, and thus depends on the solutions to every other subpart. The internal policy for a behaviour depends on its role in the greater task.

Consider, for example, the behaviour Go(hall, bedroom2) in the grid-world problem. Figure 1.2 shows two possible policies for this behaviour. Assume, for the moment that diagonal movement is impossible. Which of these policies is hierarchically optimal? The answer depends on the context in which it is being used. If the agent's overall goal was to reach the room as soon as possible, then the policy in Figure 1.2(a) is preferable. If, on the other hand, the goal is to pick up the book, then the policy in Figure 1.2(b) is better, as it will result in a shorter overall path to the book (from some starting locations). (Note that in both cases, the policy illustrated is only one of many available optimal policies.)

Furthermore, the same behaviour may have different internal policies in different parts of the problem. For instance, if the agent's goal is to fetch the book, carry it to another room and then return to the bedroom, then the first instance of Go(hall, bedroom2) will use the policy in Figure 1.2(b) and the second instance will use the policy in Figure 1.2(a).

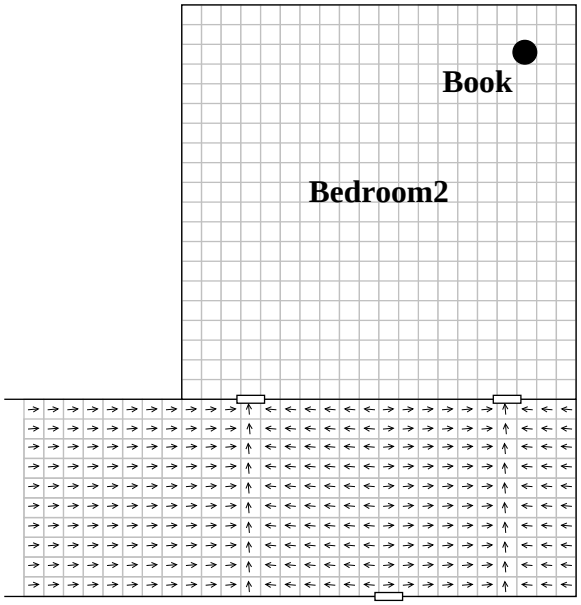
An alternative is to define local goals for each behaviour in terms of a behaviour-specific reward function. The behaviour's internal policy is learnt to optimise this local reward, rather than the global reward. This is called *recursive optimality*[16] and is a weaker form than hierarchical optimality. Recursively optimal policies make best use of the behaviours provided to them, but cannot control what the behaviours themselves do, and so cannot guarantee policies that are as efficient as hierarchically optimal policies.

The advantages of this approach, however, are several. First of all, learning an internal policy using a local reward function is likely to be much faster than learning with a global one. The behaviour can be learnt independently, without reference to the others. Local goals are generally simpler than global goals, and local rewards occur sooner than global ones. So each individual behaviour will be learnt more quickly.

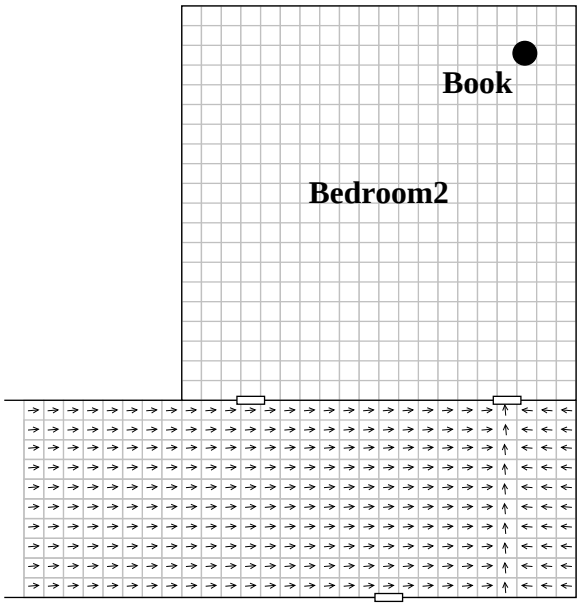
Furthermore, local goals often allow state abstraction. Elements of the state that are irrelevant to a local reward function can be ignored when learning the behaviour. So, for example, if the Go(hall, bedroom2) behaviour had a local reward function which rewarded the agent for arriving in the bedroom, then the internal policy for the behaviour could ignore what the robot is carrying. This would reduce the size of the state space for this behaviour by a factor of four.

Finally, local goals allow re-use. Once a behaviour has been learnt in one context, it can be used again in other contexts without having to re-learn its internal policy. This is useful when the same behaviour is employed several different times within the one policy.

The decision whether or not to include local goals is a trade-off between optimality and learning speed. In the ideal case, when local rewards exactly match the projected global rewards, the policies learnt will be identical. However this is unlikely to occur, and so we must decide which measure of performance is more important to us. In practice different researchers have chosen different approaches, as will become apparent in Section 1.4.



(a) A policy which optimises the number of steps to enter the room



(b) A policy which optimises the number of steps to reach the book

Fig. 1.2 Two different internal policies for the behaviour Go(`hall`, `bedroom2`).

1.3.4 Semi-Markov Decision Processes: A Theoretical Framework

So far we have described hierarchical reinforcement learning in abstract terms. We have assumed that choosing between behaviours can be done in much the same way as choosing primitive actions in monolithic reinforcement learning, to optimise the expected discounted return. There is, however, a fundamental difference between monolithic and hierarchical reinforcement learning: behaviours are temporally extended where primitive actions are not. Executing a behaviour will produce a sequence of state-transitions, yielding a sequence of rewards. The MDP model that was explained in Section 1.2 is limited insofar as it assumes each action will take a single time-step. A new theoretical model is needed to take this difference into account.

Semi-Markov Decision Processes [21] are an extension of the MDP model to include a concept of duration, allowing multiple-step actions. Formally an SMDP is a tuple $\langle \mathcal{S}, \mathcal{B}, T, R \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{B}$ is a set of behaviours (temporally-abstract actions), $T : \mathcal{S} \times \mathcal{B} \times \mathcal{S} \times \mathbb{N}^+ \to [0, 1]$ is a transition function (including duration of execution), and $R : \mathcal{S} \times \mathcal{B} \times \Re \to [0, 1]$ is a reward function:

$$T(s', k|s, \mathsf{B}) = P\left(\begin{array}{l} \mathsf{B}_t \text{ terminates in } s' \text{ at time } t+k \mid \\ s_t = s, \mathsf{B}_t = \mathsf{B} \end{array} \right) \tag{1.13}$$

$$R(r|s, \mathsf{B}) = P\left(\sum_{i=0}^{k-1} \gamma^i r_{t+i} = r \mid s_t = s, \mathsf{B}_t = \mathsf{B} \right) \tag{1.14}$$

when B_t is the behaviour executing at time t. T and R must both obey the Markov property, i.e. they can only depend on the behaviour and the state in which it was started. (This formulation of a SMDP is based on that described by Parr in [35]).

A policy is a mapping $\pi : \mathcal{S} \to \mathcal{B}$ from states to behaviours. A state-value function can be given as:

$$V^\pi(s) = \int_{-\infty}^{+\infty} rR(r|s, \pi(s))\, \mathrm{d}r + \sum_{s', k'} T(s', k'|s, \pi(s))\, \gamma^k V^\pi(s') \tag{1.15}$$

Semi-Markov Decision Processes are designed to model any continuous-time discrete-event system. Their purpose in hierarchical reinforcement learning is more constrained. Executing a behaviour results in a sequence on primitive actions being performed. The value of the behaviour is equal to the value of that sequence. Thus if behaviour B is initiated in state s_t and terminates sometime later in state s_{t+k} then the SMDP reward value R_t is equal to the accumulation of the one-step rewards received while executing B:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{k-1} r_{t+k-1} \tag{1.16}$$

Thus the state-value function in Equation 1.15 above becomes:

$$V^{\pi}(s) = E\left\{\sum_{i=0}^{\infty}\gamma^{i}r_{t+i} \mid \varepsilon(\pi, s, t)\right\} \tag{1.17}$$

which is identical to the state-value function for primitive policies shown previously in Equation 1.4. We can define an optimal behaviour-based policy $\pi^{\star}$ with the optimal state-value function $V^{\pi^{\star}}$ as:

$$V^{\pi^{\star}}(s) = \max_{\pi} V^{\pi}(s) \tag{1.18}$$

Since the value measure V^{π} for a behaviour-based policy π is identical to the value measure V^{π} for a primitive policy we know that $\pi^{\star}$ yields the optimal primitive policy over the limited set of policies that our hierarchy allows.

1.3.5 Learning behaviours

Learning internal policies of behaviours can be expressed along the same lines. Formally, let $\mathsf{B}.\pi$ be the policy of behaviour B, and $\mathsf{B}.\mathcal{A}$ be the set of sub-actions (either behaviours or primitives) available to B. Let Root indicate the root behaviour, with reward function equal to that of the original (MDP) learning task. The recursively optimal policy has:

$$\mathsf{B}.\pi^{\star}(s) = \arg\max_{a \in \mathsf{B}.\mathcal{A}} \mathsf{B}.Q^{\star}(s, a) \tag{1.19}$$

where $\mathsf{B}.Q^{\star}(s, a)$ is the optimal state-action value function for behaviour B according to its local reward function $\mathsf{B}.R$ (defined by the system designer in accordance to the behaviour's goals) .

In contrast, the hierarchically optimal policy has

$$\mathsf{B}.\pi^{\star}(\mathit{stack}, s) = \arg\max_{a \in \mathsf{B}.\mathcal{A}} \mathsf{Root}.Q^{\star}(\mathit{stack}, s, a) \tag{1.20}$$

where $\mathit{stack} = \{\mathsf{Root}, \ldots, \mathsf{B}\}$ is the calling stack of behaviours and $\mathsf{Root}.Q^{\star}$ is the state-action value function according to the root reward function. The stack is a necessary part of the input to an hierarchically optimal policy, as the behaviour may operate differently in different calling contexts. (Hierarchically optimal policies do not allow local goals for behaviours, so $\mathsf{B}.R$ and $\mathsf{B}.Q^{\star}$ are not defined.)

1.4 HIERARCHICAL REINFORCEMENT LEARNING IN PRACTICE

We have discussed the expected benefits of hierarchical reinforcement learning in abstract terms without referring to any particular algorithm, to show what motivates its exploration. Historically a large number of different implementations have been proposed ([14] [28] [24]) but only recently have they been developed into a strong theoretical framework that has been commonly agreed upon. Even so, there are

several current implementations that differ significantly in which elements they emphasise and how they approach the problem. We shall focus on four of the most recent offerings: SMDP Q-Learning, HSMQ-Learning, MAXQ-Q and HAMQ-Learning.

1.4.1 Semi-Markov Q-Learning

The simplest algorithm extends Watkins' Q-Learning to include temporally abstract behaviours. Bradtke and Duff [10] proposed such an algorithm, called *SMDP Q-Learning*. Assuming behaviours obey the Semi-Markov property, an optimal policy can be learnt in a manner analogous to Watkins' Q-Learning, but discounting based on the time taken by the behaviour, as shown in Algorithm 2.

Algorithm 2 SMDP Q-Learning

function SMDPQ
- $t \leftarrow 0$
- Observe state s_t
- **while** s_t is not a terminal state **do**
  - Choose behaviour $\mathsf{B}_t \leftarrow \pi(s_t)$ according to an exploration policy
  - $totalReward \leftarrow 0$
  - $discount \leftarrow 1$
  - $k \leftarrow 0$
  - **while** B_t has not terminated **do**
    - Execute B_t
    - Observe reward r
    - $totalReward \leftarrow totalReward + discount \times r$
    - $discount \leftarrow discount \times \gamma$
    - $k \leftarrow k + 1$
  - **end while**
  - Observe state s_{t+k}
  - $Q(s_t, \mathsf{B}_t) \xleftarrow{\alpha} totalReward + discount \times \max_{\mathsf{B} \in \mathcal{B}} Q(s_{t+k}, \mathsf{B})$
  - $t \leftarrow t + k$
- **end while**

end SMDPQ

Just as primitive Q-Learning learns a state-action value function, so SMDP Q-Learning learns a *state-behaviour value function* $Q : \mathcal{S} \times \mathcal{B} \rightarrow \Re$, which is an approximation to the *optimal state-behaviour value function* $Q^\star$:

$$Q^\star(s, \mathsf{B}) = E\left\{ \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V^\star(s_{t+k}) \mid \varepsilon(s, \mathsf{B}, t) \right\} \tag{1.21}$$

where $\varepsilon(s, \mathsf{B}, t)$ indicates the event of executing behaviour B in state s at time t, and k is a random variable expressing the duration of the behaviour B in this event (taken into account in the expectation).

The optimal policy is defined as before:

$$\pi^\star(s) = \arg\max_{\mathsf{B}\in\mathcal{B}} Q^\star(s,\mathsf{B}) \tag{1.22}$$

The approximation $Q(s,\mathsf{B})$ can be learnt via the update rule (analogous to the Q-Learning update rule in Equation 1.12):

$$Q(s_t,\mathsf{B}_t) \xleftarrow{\alpha} R_t + \gamma^k \max_{\mathsf{B}\in\mathcal{B}} Q(s_{t+k},\mathsf{B}) \tag{1.23}$$

where k is the duration of B_t and R_t is a discounted accumulation of all single-step reward values received while executing the behaviour:

$$R_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} \tag{1.24}$$

SMDP Q-Learning can be shown to converge to the optimal behaviour-based policy under circumstances similar to those for 1-step Q-Learning [35].

Sutton, Precup and Singh [51] applied SMDP Q-Learning to behaviours they called "options". An option B has a fixed internal policy $\mathsf{B}.\pi$ which is recursively constructed from other options and primitive actions. If this internal policy obeys the Markov property, then such behaviours are semi-Markov and can be used in SMDP Q-Learning. Other model-based dynamic programming techniques such as value-iteration and Monte Carlo methods can also be applied to options using the semi-Markov model [37]. Also, since different options are constructed from the same primitive building blocks, *intra-option* learning is possible, where experiences from executing one option can be applied to learning about another. This will be described in more detail in Section 1.6.

1.4.2 Hierarchical Semi-Markov Q-Learning

Hierarchical Semi-Markov Q-Learning (HSMQ) [17] is a recursively optimal learning algorithm that learns reactive behaviour-based policies, with a designer-specified task hierarchy. As shown in Algorithm 3 it is a simple elaboration of the SMDP Q-Learning algorithm. The SMDPQ update rule given in Equation 1.23 is applied recursively with local reward functions at each level of the hierarchy. TASKHIERARCHY is a function which returns a set of available actions (behaviours or primitives) that can be used by a particular behaviour in a given state. This hierarchy is hand-coded by the system designer based on knowledge of which actions are appropriate on what occasions.

HSMQ-Learning converges to a recursively optimal policy with the same kinds of requirements as SMDP Q-Learning, provided also that the exploration policy for behaviours is greedy in the limit [45].

Algorithm 3 HSMQ-Learning

function HSMQ(state s_t, action a_t)
returns sequence of state transtions $\{\langle s_t, a_t, s_{t+1}\rangle, \ldots\}$
  if a_t is primitive **then**
    Execute action a_t
    Observe next state s_{t+1}
    return $\{\langle s_t, a_t, s_t + 1\rangle\}$
  else
    sequence $S \leftarrow \{\}$
    behaviour $\mathsf{B} \leftarrow a_t$
    $\mathcal{A}_t \leftarrow \text{TASKHIERARCHY}(s_t, \mathsf{B})$
    while B is not terminated **do**
      Choose action $a_t \leftarrow \mathsf{B}.\pi(s_t)$ from $\mathcal{A}_t$
        according to an exploration policy
      sequence $S' \leftarrow \text{HSMQ}(s_t, a_t)$
      $k \leftarrow 0 \qquad totalReward \leftarrow 0$
      for each $\langle s, a, s'\rangle \in S'$ **do**
        $totalReward \leftarrow totalReward + \gamma^k \mathsf{B}.r(s, a, s')$
        $k \leftarrow k + 1$
      end for
      Observe next state s_{t+k}
      $\mathcal{A}_{t+k} \leftarrow \text{TASKHIERARCHY}(s_{t+k}, \mathsf{B})$
      $\mathsf{B}.Q(s_t, a_t) \xleftarrow{\alpha} totalReward + \gamma^k \max_{a \in \mathcal{A}_{t+k}} \mathsf{B}.Q(s_{t+k}, a)$
      $S \leftarrow S + S'$
      $t \leftarrow t + k$
    end while
    return S
  end if
end HSMQ

1.4.3 MAXQ-Q

A more sophisticated algorithm for learning recursively optimal policies is Dietterich's MAXQ-Q [16]. The policies it learns are equivalent to those of HSMQ, but it uses a special decomposition of the state-action value function in order to learn them more efficiently. MAXQ-Q relies on the observation that the value of a behaviour B as part of its parent behaviour P can be split into two parts: the reward expected while executing B, and the discounted reward of continuing to execute P after B has terminated. That is:

$$\mathsf{P}.Q(s, \mathsf{B}) = \mathsf{P}.I(s, \mathsf{B}) + \mathsf{P}.C(\mathsf{P}, s, \mathsf{B}) \tag{1.25}$$

where $\mathsf{P}.I(s, \mathsf{B})$ is the expected total discounted reward (according to the reward function of the parent behaviour P) that is received while executing behaviour B from initial state s, and $\mathsf{P}.C(\mathsf{B}_{\textit{parent}}, s, \mathsf{B}_{\textit{child}})$ is the expected total reward of continuing to execute behaviour $\mathsf{B}_{\textit{parent}}$ after $\mathsf{B}_{\textit{child}}$ has terminated, discounted appropriately to take into account the time spent in $\mathsf{B}_{\textit{child}}$. (Again with rewards calculated according to the behaviour P.)

Furthermore the $I(s, \mathsf{B})$ function can be recursively decomposed into I and C via the rule:

$$\mathsf{P}.I(s, \mathsf{B}) = \max_{a \in \mathsf{B}.\mathcal{A}} \mathsf{P}.Q(s, a) \tag{1.26}$$

There are several advantages to this decomposition, primarily of value in learning recursively optimal Q-values. The I and C functions can each be represented with certain state abstractions that do not apply to both parts. The explanation is complex and beyond the scope of this review. For full details and pseudocode see [16].

1.4.4 Q-Learning with Hierarchies of Abstract Machines

Q-Learning with Hierarchies of Abstract Machines (HAMQ) [36] is an hierarchically optimal learning algorithm that uses a more elaborate model to structure the policy space. Behaviours are implemented as *hierarchies of abstract machines* (HAMs) which resemble finite-state machines, in that they include an internal *machine state*. The state of the machine dictates the action it may take. Actions include: 1) performing primitive actions, 2) calling other machines as subroutines, 3) making choices, 4) terminating and returning control to the calling behaviour. Transitions between machine states may be deterministic, stochastic or may rely on the state of the environment. Learning takes place at choice states only, where the behaviour must decide which of several internal state transitions to make. HAMs represent a compromise between hard-coded policies and fully-learnt policies. Some transitions can be hard-coded into the machine while others can be learnt. Thus they provide a means for background knowledge in the form of partial solutions to be specified.

Behaviours in HAMQ are merely a typographic convenience. In effect they are compiled into a single abstract machine, consisting of action nodes and choice nodes only. Algorithm 4 shows the Pseudocode for learning in such a machine.

Andre and Russell [2] have extended the expressive power of HAMs by introducing parameterisation, aborts and interrupts, and memory variables. These Programmable HAMs allow quite complex programmatic description of behaviours, while also providing room for exploration and optimisation of alternatives.

Algorithm 4 HAMQ-Learning

function HAMQ
  $t \leftarrow 0$
  $node \leftarrow$ starting node
  $totalReward \leftarrow 0$
  $k \leftarrow 0$
  choice $a \leftarrow null$
  choice state $s \leftarrow null$
  choice node $n \leftarrow null$
  while s is not a terminal state **do**
    if $node$ is an action node **then**
      Execute action
      Observe reward r
      $totalReward \leftarrow totalReward + \gamma^k r$
      $k \leftarrow k + 1$
      $node \leftarrow node.next$
    else $node$ is a choice node
      Observe state s'
      if $n \neq null$ **then**
        $Q(n, s, a) \xleftarrow{\alpha} totalReward + \gamma^k \max_{a' \in \mathcal{A}} Q(node, s', a')$
        $totalReward \leftarrow 0$
        $k \leftarrow 0$
      end if
      $n \leftarrow node$
      $s \leftarrow s'$
      Choose transition $a \leftarrow \pi(n, s)$ according to an exploration policy
      $node \leftarrow a.destination$
    end if
  end while
end HAMQ

1.5 TERMINATION IMPROVEMENT

In Section 1.3.2 above, we discussed the importance of long-term commitment to behaviours. Without this, much of the benefit of using temporally abstract actions is lost. However it can also be an obstacle in the way of producing optimal policies. Consider the situation illustrated in Figure 1.3. The task is to navigate to the indicated goal location. Behaviours are represented by dotted circles and black dots indicating the applicability space and terminal states respectively. The heavy line shows a path from the starting location to the goal, using the behaviours provided. The path

travels from one termination state to the next, indicating that each behaviour is being executed all the way to completion.

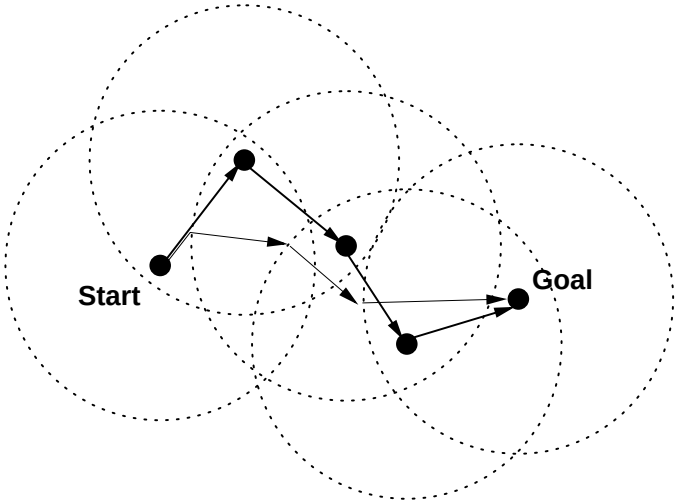


Fig. 1.3 A simple navigation task illustrating the advantage of termination improvement. The circles show the overlapping applicability spaces for a collection of hard-code navigation behaviours. Each behaviour moves the agent towards the central landmark location (the black dots). The heavy line indicates the standard policy with commitment to behaviours. The lighter line indicates the path taken by a termination improved policy.

Compare this with the path shown by the lighter line. In this case each behaviour is executed only until a more appropriate behaviour becomes applicable. "Cutting corners" in this way results in a significantly shorter path, and a policy much closer to the optimal one.

This example is taken from the work of [46] who call this process *termination improvement*. They show how to produce such corner-cutting policies using hard-coded behaviours. Having already learnt an optimal policy π using these behaviours, they transform it into an improved *interrupted policy* π' by prematurely interrupting an executing behaviour B whenever $Q(s, \mathsf{B}) < V(s)$, i.e. when there is a better alternative behaviour available. The resulting policy is guaranteed to be of equal or greater efficiency than the original.

A similar approach can be applied to policies learnt using MAXQ-Q [16]. While MAXQ-Q is a recursively optimal learning algorithm, it nevertheless learns a value for each primitive action using the global reward function. In normal execution, actions are chosen on the basis of the local Q-value assigned to each by its calling behaviour. However once such a recursively optimal policy has been learnt, it can be improved by switching to selecting primitive actions based on their global Q-value instead. There is no longer any commitment to behaviours. Execution reverts to the reactive semantics of monolithic Q-learning, and the hierarchy serves only s a means to assign Q-values to primitives. This is called the *hierarchical greedy policy*,

and is also guaranteed to be of equal or greater efficiency than the recursive policy. Furthermore, by continuing to update these Q-values, via *polling execution* [24], [15], this policy can be further improved.

In both these algorithms it is important that the transformation is made to the policy once an uninterrupted policy has already been learnt. If behaviours can be arbitrarily interrupted from the outset then the advantages of using temporally abstract actions are lost. However in my own TRQ algorithm [40], I show that selective termination improvement can be done while learning is still ongoing, if we limit it to those occasions in which we can prove that the executing behaviour is no longer appropriate. In TRQ this is done by giving behaviours abstract symbolic descriptions and using these descriptions to build high-level plans. A behaviour is only appropriate as long as it is recommended by the plan. If a behaviour is no longer appropriate then it may be interrupted before normal termination without adversing affecting learning performance.

1.6 INTRA-BEHAVIOUR LEARNING

When behaviours are constructed from a common set of primitive actions, in a common Markov model, experiences gathered while executing one behaviour may strongly overlap with those from another. If two behaviours B_1 and B_2 have overlapping applicability spaces and action sets, then when B_1 executes action $a \in \mathsf{B}_1.\mathcal{A} \cap \mathsf{B}_2.\mathcal{A}$ in state $s \in \mathsf{B}_1.\mathrm{pre} \cap \mathsf{B}_2.\mathrm{pre}$ then the resulting experience $\langle s, a, s' \rangle$ can often be used to learn about both B_1 and B_2.

There are two ways in which this information can be transferred from B_1 to B_2:

1. It can be used to update both the *internal* policy of B_2, through a method called *all-goals updating* [24], and

2. It can be used to update the *external* policy which is calling B_1 and B_2, through *intra-option learning* [37], [51].

1.6.1 All-goals updating

Q-Learning is what is termed an *off-policy* learning algorithm [50]. This means that the update rule for the state-action value function $Q(s, a)$ (Equation 1.12 above) does not rely on the action a to be drawn from the current policy. In particular, if we are working in a recursively-optimal framework, in which there are behaviour-specific reward functions specifying local goals, then any experience $\langle s, a, s' \rangle$ with $s, s' \in \mathsf{B}.\mathrm{pre}$ and $a \in \mathsf{B}.\mathcal{A}$ can be used to update the state-action value function $\mathsf{B}.Q(s, a)$ regardless of whether the experience was obtained while executing B itself or another behaviour.

As a result, when multiple behaviours have overlapping applicability spaces and action sets, then experiences from executing one behaviour can be used to improve the policy of others. This is what Kaelbling calls *all-goals updating* [24].

This is also the case for the hierarchical extensions of Q-learning which we have presented above, but is not true for all reinforcement learning algorithms. *On-policy* algorithms, such as SARSA [39], [49] are not amenable to this kind of experience sharing as they rely on the experiences used to update a policy being drawn from the execution of that policy.

1.6.2 Intra-option learning

Whereas all-goals updating concerns updating the internal policies of overlapping behaviours, *intra-option learning* [37], [51] updates the policy of the parent behaviour which calls the subtasks B_1 and B_2. In this case B_1 and B_2 are assumed to be fixed-policy behaviours (options), although the technique can probably also be extended to hierarchically-optimal or recursively-optimal learnt behaviours (no convergence proof is yet available).

Intra-option learning is based on the observation that if a behaviour B dictates primitive action a_t in state s_t, and then continues executing then:

$$Q^\star(s_t, \mathsf{B}) = E(r_t) + \gamma \sum_{s_{t+1}} T\left(s_{t+1}|s_t, a_t\right) Q^*(s_{t+1}, \mathsf{B}) \tag{1.27}$$

i.e. the value of executing B in s_t is the immediate reward for executing a_t plus the discounted value of continuing to execute B thereafter. Alternatively, if B terminates in state s_{t+1} then:

$$Q^\star(s_t, \mathsf{B}) = E(r_t) + \gamma \sum_{s_{t+1}} T\left(s_{t+1}|s_t, a_t\right) \max_{\mathsf{B}' \in \mathcal{B}} Q^*(s_{t+1}, \mathsf{B}') \tag{1.28}$$

i.e. the value of executing B is the the immediate reward for executing a_t and then the discounted value of the subsequently chosen behaviour.

Based on this fact, we can construct a *one-step intra-option Q-learning* update rule which updates the state-behaviour value $Q(s, \mathsf{B})$ with $\mathsf{B}.\pi(s) = a$ based on the experience $\langle s, a, r, s' \rangle$, as follows:

$$Q(s, \mathsf{B}) \xleftarrow{\alpha} r + \gamma U(s_{t+1}, \mathsf{B}) \tag{1.29}$$

where:

$$U(s, \mathsf{B}) = (1 - \beta(\mathsf{B}, s))Q(s, \mathsf{B}) + \beta(\mathsf{B}, s) \max_{\mathsf{B}' \in \mathcal{B}} Q(s, \mathsf{B}')$$

where $\beta(\mathsf{B}, s)$ denotes the probability B terminates in state s.

Once again, this update rule does not rely on the experience necessarily resulting from the execution of B as part of the parent policy. The experience may be drawn from the execution of behaviour B_1 and used to update the value of B_2 provided that $\mathsf{B}_1.\pi(s) = \mathsf{B}_2.\pi(s)$. So in this way we can improve our estimations of the value of several "overlapping" behaviours based on the experience of executing only one.

Both this all-goals updating and intra-option learning increase the amount of computation done based on each experience gathered. This can potentially mean better

performance with less experience, but it can also mean longer learning times due to excess computation. There is a tradeoff between the experience and computation. Generally it is assumed that experience is the bottleneck in learning, and that computation is relatively inexpensive, but as hierarchical methods grow more complex this is an issue that is going to need to be addressed.

1.7 CREATING BEHAVIOURS AND BUILDING HIERARCHIES

As stated earlier, typically the hierarchy of behaviours is defined by a human designer. Many researchers have pointed to the desirability of automating this task (eg. [9] [18]).

One approach to this problem is the HEXQ algorithm [19]. This algorithm is an extension of MAXQ-Q which attempts to automatically decompose a problem into a collection of subproblems. Sub-problems are created corresponding to particular variables in the state-vector. Variables that change infrequently inspire behaviours which aim to cause those variables to change.

A similar approach is used by acQuire [33]. It uses exploration to identify "bottlenecks" in the state-space – states which are part of many trajectories through the space. Bottleneck states are selected as subgoals for new behaviours.

Both these approaches implement a kind of uninformed behaviour invention, based only on the dynamics of the environment without any background knowledge. RACHEL [40] adopts alternative approach. This system combines hierarchical reinforcement learning with symbolic planning. Behaviours are specified by a training in abstract terms, in terms of their pre-conditions and goals. These specifications take the form of teleo-operators [34], which are used in conjunction with symbolic planning to build hierarchies of behaviours suited to a given task.

1.8 MODEL-BASED REINFORCEMENT LEARNING

As we said earlier, not all reinforcement learning algorithms are model-free like Q-Learning and it's derivatives. There are also many algorithms which attempt to learn models of the transition and reward functions $T(s'|s,a)$ and $R(s'|s,a)$, and then use these models to construct policies using dynamic programming methods such as value-iteration or policy-iteration (eg [48]). Such techniques have been somewhat less popular in practice as learning accurate models of T and R has been found to be more difficult than learning Q-values directly.

Nevertheless these technique have also been applied to the hierarchical reinforcement learning problem, and model-based hierarchical algorithms exist (eg H-Dyna [43], SMDP Planning [37], abstract MDPs [18], and discrete-event models [31]).

It is unfortunate that model-based methods have apparently acquired second-class status in this field as there are potentially many ways in which a complete model, at multiple levels of abstraction, could be exploited far beyond a particular task-specific set of Q-values. This is an area deserving further attention.

1.9 TOPICS FOR FUTURE RESEARCH

1.9.1 Recombination of different features

As we have described above, most hierarchical reinforcement learning algorithms are actually the combination of several independent features. These particular combinations are not necessarily forced, but are mainly accidental. Some of these features are:

1. Hierarchical vs Recursive optimality,
2. The MAXQ decomposition of the state-action value function,
3. The HAMQ programmable behaviour structure,
4. Termination improvement
5. Intra-option learning

It would certainly be worthwhile to investigate other recombinations of these features. There is no obvious reason, for example, why the MAXQ decomposition could not be applied to to a hierarchically optimal learning algorithm; or the HAMQ framework used to structure recursively optimal behaviours with termination improvement. Some combinations may not work, but we cannot be sure of this until we explore them.

1.9.2 Real-world Applications

As must be apparent from this chapter, the development of hierarchical reinforcement theory and algorithms has been proceeding strongly in recent years. However this development has far outstripped the use of these methods in non-trivial real-world situations. One piece of work stands out: Crites successful use of SMDP Q-learning to learn elevator control [12], [13]. Few others exist.

Hierarchical reinforcement learning is founded on the claim that hierarchy can help us overcome the curse of dimensionality. The field has matured, the algorithms are available. Now it is time to prove that these claims are not empty.

1.10 CONCLUSION

Hierarchy has proven to be a rich source of development in reinforcement learning, producing both theoretical and practical advances. It has found a solid theoretical model in Semi-Markov Decision Processes and has been growing strongly. Diverse algorithms have been built on this shared foundation, differing in terms of optimality criteria, action selection mechanisms, state-approximation, and, as always, terminology. In this chapter we have tried to unify these different approaches and show how their might be separated and recombined. It is our hope that this will give a coherent

foundation for future development and the application of these new ideas to solving greater problems.

Acknowledgements

I would like to thank Andrew Barto for reviewing several drafts of this report and providing many valuable recommendations.

References

1. *Proceedings of the Tenth International Conference on Machine Learning*, San Franciso, CA, 1993. Morgan Kaufmann.

2. David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems 12: Proceedings of the 1999 Conference*, pages 1019–1025, San Franciso, CA, 2000. Morgan Kaufmann.

3. Leemon C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37, San Franciso, CA, 1995. Morgan Kaufmann.

4. Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete-Event Dynamical Systems: Theory and Applications*, 13:341–379, 2003.

5. J. Baxter, A. Tridgell, and L. Weaver. Knightcap: A chess program that learns by combining TD(λ) with game-tree search. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 28–36, San Franciso, CA, 1998. Morgan Kaufmann.

6. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

7. D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

8. D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

9. Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

10. Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 393–400. The MIT Press, 1995.

11. Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

12. Robert Harry Crites. *Large-Scale Dynamic Optimization Using Teams of Reinforcement Learning Agents*. PhD thesis, University of Massachusetts Amherst, 1996.

13. Robert Harry Crites and Andrew G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2-3):235–262, 1998.

14. Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, 5:271–278, 1992.

15. Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126, San Franciso, CA, 1998. Morgan Kaufmann.

16. Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value dunction decomposition. *Artitificial Intelligence*, 13:227–303, 2000.

17. Thomas G. Dietterich. An overview of MAXQ hierarchical reinforcement learning. In B. Y. Choueiry and T. Walsh, editors, *Proceedings of the Symposium on Abstraction, Reformulation and Approximation SARA 2000, Lecture Notes in Artificial Intelligence*, pages 26–44, New York, NY, 2000. Springer Verlag.

18. Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *Uncertainty in Artificial Intelligence*, pages 220–229, 1998.

19. Bernhard Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 243–250, San Franciso, CA, 2002. Morgan Kaufmann.

20. Ronald A. Howard. *Dynamic Programming and Markov Processes*. The MIT Press, Cambridge, MA, 1960.

21. Ronald A. Howard. *Dynamic Probabilistic Systems: Semi-markov and Decision Processes*. John Wiley and sons, New York, NY, 1971.

22. Glenn A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989.

23. Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. In *Advances in Neural Information Processing Systems*, volume 6. The MIT Press, November 1994.

24. Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning* [1].

25. Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

26. Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, volume 2, pages 686–691, Anaheim, California, USA, 1991. AAAI Press/MIT Press.

27. Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

28. Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1993.

29. P. Maes. How to do the right thing. *Connection Science Journal, Special Issue on Hybrid Systems*, 1, 1990.

30. Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1-3):159–195, 1996.

31. Sridhar Mahadevan, Nikfar Khaleeli, and Nicholas Marchalleck. Designing agent controllers using discrete-event markov models. In *Working Notes of the AAAI Fall Symposium on Model-Directed Autonomous Systems*, Cambridge, Massachusetts, 1997.

32. Maja J. Matarić. Behaviour based control: Examples from navigation, learning and group behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2-3), 1996.

33. Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 361–368, San Franciso, CA, 2001. Morgan Kaufmann.

34. Nils J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

35. Ronald Parr. *Hierarchical Control and learning for Markov decision processes*. PhD thesis, University of California at Berkeley, 1998.

36. Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

37. Doina Precup. *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 2000.

38. Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.

39. G. A. Rummery and M. Niranjan. Online Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

40. Malcolm R. K. Ryan. *Hierarchical Reinforcement Learning: A Hybrid Approach*. PhD thesis, University of New South Wales, 2002.

41. Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

42. A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning* [1].

43. Satinder P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artifical Intelligence*, Cambridge, MA, 1992. MIT Press.

44. Satinder P. Singh. Reinforcement learning algorithms for average-payoff markovian decision processes. In *National Conference on Artificial Intelligence*, pages 700–705, 1994.

45. Satinder P. Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

46. R. S. Sutton, S. Singh, D. Precup, and B. Ravindran. Improved switching among temporally abstract actions. In *Advances in Neural Information Processing Systems*, volume 11. The MIT Press, 1999.

47. Richard S. Sutton. Implementation details for the TD(λ) procedure for the case of vector predictions and backpropagation. Technical Report TN87-509.1, GTE Laboratories, 1987.

48. Richard S. Sutton. Integrated architectures for learning, planning and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, San Franciso, CA, 1990. Morgan Kaufmann.

49. Richard S. Sutton. Generalisation in reinforcement learning successful examples using sparse coarse coding. *Advances in Neural Neural Information Processing Systems*, 1995.

50. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

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

52. G. Tesauro. Td-gammon, a self-teaching backgammon program achieves master-level play. *Neural Computation*, 6:215–219, 1994.

53. S. B. Thrun. The role of exploration in learning control with neural networks. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, Florence, Kentucky, 1992. Van Nostrand Reinhold.

54. John N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16(3), September 1994.

55. Christopher. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, England, 1989.

56. Christopher. J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

57. W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, San Franciso, CA, 1995. Morgan Kaufmann.