Hierarchical Reinforcement Learning in Multi-Agent Environment

Group d636a

June 17, 2005

Department of Computer Science Aalborg University

Department of Computer Science

TITLE:
Hierarchical Reinforcement Learning in Multi-Agent Environment

PROJECT PERIOD:
February 7, 2005 – June 17, 2005

PROJECT GROUP:
d636a

GROUP MEMBERS:
Tim Boesen
Dennis Kjærulff Pedersen

SUPERVISOR:
Uffe Kjærulff

ABSTRACT:

The purpose of this report is to explore the area of Hierarchical Reinforcement Learning. First a hierarchical reinforcement approached called the *MaxQ value function decomposition* is described in great detail. Using MaxQ the state space can be reduced considerably. To support the claim that MaxQ performs better than the basic reinforcement learning algorithm, a test comparing the two is performed. The results clearly show that as the complexity grows, so does the difference in performance between the two.
The MaxQ algorithm does not allow agents to cooperate. An extension to MaxQ is presented that allow agents with the same task decomposition to coordinate and cooperate at a high level of abstraction. We show that two agents using the new algorithm does in fact deliver better results than two agents using the basic MaxQ value function decomposition.
To further explore the area of multi-agent reinforcement learning, we propose two approaches that deals with heterogeneity in multi-agent environment. The first approach uses experience sharing to speed up learning, while the other expands the multi-agent hierarchical algorithm to allow agents with different task decompositions to cooperate.

Preface

This project was done by group d636a at Aalborg University, Department of Computer Science. We would like to thank Uffe Kjærulff for supervising the entire project.

Aalborg, June 17, 2005

Dennis Kjærulff Pedersen

Tim Boesen

Contents

1 Introduction **1**

1.1 Contribution 2

1.2 Related Work 2

1.3 Outline of the Report 3

2 Reinforcement Learning **5**

2.1 Markov Decision Process 6

2.2 Non-Determinism 7

2.3 Summary 9

3 Hierarchical Reinforcement Learning **11**

3.1 The Taxi Problem 12

3.2 Semi-Markov Decision Process 14

3.3 MAXQ Problem Decomposition 14

3.4 Hierarchical Policy 15

3.5 Projected Value Function 15

3.5.1 MAXQ Graph 18

3.6 Optimality in MAXQ 18

3.6.1 Hierarchical Optimality 19

3.7 The MAXQ-0 Learning Algorithm 20

3.7.1 Example Execution 20

3.7.2 Requirements to the Algorithm 23

3.8 The MAXQ-Q Learning Algorithm 24

3.8.1 All-States Updating 24

3.9 State Abstraction 26

3.9.1 Leaf Irrelevance 27

3.9.2 Subtask Irrelevance 28
3.9.3 Result Distribution Irrelevance 29
3.9.4 Termination 29
3.9.5 Shielding 30
3.10 Test of HRL 30
3.10.1 The Taxi Problem and Tabular Reinforcement Learning . 32
3.11 Summary 36

4 Homogeneous Multi-Agent Hierarchical Reinforcement Learning 37
4.1 Introduction to the Approach Used 38
4.2 Multi-agent Semi-Markov Decision Process 39
4.3 Task Decomposition 41
4.4 Hierarchical Multi-Agent Policy 41
4.5 Projected Value Function 41
4.6 A Hierarchical Multi-agent Reinforcement Learning Algorithm . 43
4.7 The Multi-agent Taxi Problem 46
4.7.1 An Example 47
4.7.2 State Abstraction 49
4.8 Testing of the MHRL Algorithm 52
4.8.1 Environmental Setup and Passenger Adding Rules 52
4.8.2 Exploration Policy Used During Learning 53
4.8.3 The Test 54
4.9 Summary 57

5 Adding Heterogeneity to Multi-Agent Environments 59
5.1 Heterogeneity in Method 60
5.1.1 The Concept of Guidance 60
5.1.2 Inter-Agent Guidance 61
5.1.3 The Independent Agents 62
5.1.4 The Dependent Agents 64
5.1.5 Expectations 65
5.2 Heterogeneity in Goal 67
5.2.1 Expanding the Taxi Problem 67
5.2.2 Communication Using Interest Groups 69
5.2.3 Projected Value Function 72
5.2.4 A Multi-agent Reinforcement Learning Algorithm for Heterogeneous Agents 73
5.3 Summary 75

6 Conclusion **77**

6.1 Future Work 78

A 5 Step Rewards for the HRL Implementation **79**

Chapter 1

Introduction

Machine learning is a fast growing field in computer science. Its influence can be seen in many aspects of our daily lives, from computer games to checking out groceries at the local supermarket. Within the field of machine learning we find Reinforcement Learning (RL).

In RL the goal is to teach agents the correct actions in a given situation. This is done in much the same way humans and animals learn, through trial and error. When an agent performs a good action, or a series of good actions it is rewarded. Likewise when it performs a bad action or a series of bad actions it is penalized. This is the same as when teaching your dog. When the dog does what it is told we give it a treat, and when it misbehaves we scold it.

RL is the underlying theory behind most of the work done in this report. Because RL in its basic tabular form, as described by Mitchell in [9], suffers from scaling problems, it, in itself, is not very useful in real life problems. However, much research has been done in this area, and several ideas building on the concept of RL have been developed. One such technique is Relational RL [4] (RRL). RRL combines traditional reinforcement learning with inductive logic programming. RRL seeks to generalise over states and objects present in a given problem domain to handle scaling and speed up learning.

Another path that has also been researched a lot is that of applying hierarchical structures to a problem, thereby decomposing the problem into a lot of smaller problems. This branch of RL is called Hierarchical RL [3] (HRL). HRL decomposes a problem into sub-problems, or subtasks as they are referred to in the rest of this report. What makes HRL interesting is the opportunities for state abstraction it presents. Because we can abstract in the individual subtasks, we can reduce the amount of space required to represent the value function. As a result, the learning speed as well as execution speed of a problem can be increased considerably when compared to regular RL.

The history of reinforcement learning has two main threads which together have become modern reinforcement learning [13].

One thread concerns the problem of optimal control and its solution using value functions and dynamic programming. The focus on optimal control dates back

to the late 1950s, and was used to describe the problem of designing a controller to minimise a measure of a dynamical system's behaviour over time. One of the approaches to this problem was developed in the mid-1950s by Richard Bellman and colleagues [1]. This approach uses the concept of a dynamical system's state and of a value function, to define a functional equation, now often called the Bellman equation.

The other thread is concerned with learning through trial and error and began in the psychology of animal training where "reinforcement" theories are common. The essence is that actions followed by good or bad outcomes have a tendency to be picked more or less frequently accordingly. Thorndike called this the "Law of Effect" [14] because it describes the effect of reinforcing events on the tendency to select actions. Harry Klopf [6, 7] introduced the trial and error to artificial intelligence as reinforcement learning. He recognised that an adaptive behaviour was missing, as researchers had almost only been focusing on supervised learning.

1.1 Contribution

In this report we will describe the MaxQ learning algorithm invented by Thomas Dietterich [3]. We will elaborate on the work done by Dietterich, and we will verify that HRL does in fact provide a substantial increase in performance when comparing it to basic tabular RL.

In Dietterichs work the focus is on single-agent environments. We will expand the MaxQ learning algorithm to include cooperating agents by using the Multi-agent HRL (MHRL) algorithm presented by M. Ghavamzadeh and S. Mahadevan in [8]. As with HRL we will elaborate on the concepts presented in their report as well as provide testing results of MHRL on a simple multi-agent problem.

The approach introduced by Ghavamzadeh and Mahadevan focuses on homogeneous agents. We will explore the area of Heterogeneity in multi-agent environments. To this end two approaches are introduced each dealing with a different aspect of heterogeneity. The first method uses something called inter-agent experience sharing, and is meant for agents that are heterogeneous in the sense that they use different learning algorithms to achieve the same goal. This algorithm also has the property of speeding up learning in cases where the value function is not shared between agents. The second approach expands the algorithm presented by Ghavamzadeh and Mahadevan to allow heterogeneous agents. This is done using a concept called interest groups.

1.2 Related Work

The concept of Q learning has been widely covered by several researchers. Some of them are Christopher J.C.H. Watkins and Peter Dayan in [15].

The problem of scaling using reinforcement learning has been addressed by, among others, Sašo Džeroski and Luc De Raedt and Kurt Driessens. In [4]

they explain relational reinforcement learning, which is a learning technique that combines reinforcement learning with inductive logic programming. The basic principle of RRL is to describe the state of an environment using relational properties instead of absolute properties. Often, this makes it possible to introduce state generalisations that reduces the number of essentially different states—thereby creating an opportunity to learn more complex problems. Instead of representing the behaviour function[1] in a table based manner (as is normally the case in traditional reinforcement learning), RRL makes use of first order logic and so-called logical classification trees. This enables a more compact representation.

1.3 Outline of the Report

The report is organised as follows: In Chapter 2, a short description of traditional RL and the scaling problems associated with the technique are presented.

In Chapter 3 we explain how HRL makes RL much more scaleable. To illustrate how this can be achieved a problem called the *Taxi Problem* is used.

Adding multi-agent support to HRL is done in Chapter 4. Here it is explained how HRL can, with a few modifications, be made to work with multiple agents.

Chapter 5 presents two new approaches for implementing heterogeneity into multi-agent settings. The first approach uses guidance to speed up learning, as well as allowing agents with different learning algorithms to learn from each other. The second approach builds on the concepts presented in Chapter 3 and Chapter 4, but allows agents with different goals to coordinate and cooperate.

Finally Chapter 6 concludes on the work and results presented in this report.

[1] In Q learning, this is known as the Q function.

Chapter 2

Reinforcement Learning

One of the most commonly used paradigms in machine learning is supervised learning. Supervised learning is a general method for approximating functions, and can e.g. be used to train neural networks. Training data in supervised learning consists of input/output pairs, some of which are supplied for training, while the rest are saved for testing the approximated function.

Supervised learning is very good for solving problems such as classification and problems where the desired behaviour is known. Many problems fit nicely within the supervised training paradigm. For instance, imagine that you would like an agent to learn when the weather is ideal for playing tennis [10]. Then you simply need to organise a set of examples in which playing tennis is a good idea, e.g. `[(sunny, weak wind),(overcast, weak wind)]`, and a set in which it is *not* a good idea, e.g. `[(rainy, weak wind),(rainy, strong wind)]`, and feed them to the supervised learning algorithm.

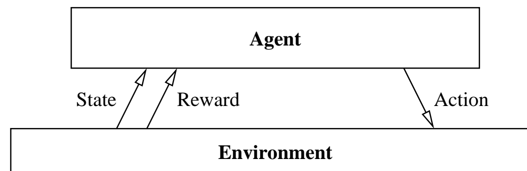


Figure 2.1: Agent interaction with the environment.

For some problems it can be very difficult (or even impossible) to define the optimal behaviour of an agent in advance. It is often easier to pick out specific elements from a problem and say: "if the agent is in state s and chooses action a, then it should be rewarded (or penalised)". This is the essence of reinforcement learning, where the agent acts on reinforcing stimuli from the environment in which it exists. Agents can start out with no knowledge of the environment, and learn the optimal strategy for reaching an unknown goal as illustrated in Figure 2.1 [9]. The agent learns in a trial and error manner by exploring the environment and by, to some extent, exploiting what it has already learnt. Reinforcement learning is in some of the literature also referred to as reward/penalty learning, or consequence learning.

This chapter reviews the formalism behind traditional Tabular Reinforcement Learning (TRL), where the behaviour function being learnt is approximated in a table based manner.

2.1 Markov Decision Process

A Markov Decision Process (MDP) [9] is a four-tuple $\langle S, A, r, \delta \rangle$, where S is a set of states, A is a set of actions, $r : S \times A \rightarrow \mathcal{R}$ is a reward function, and $\delta : S \times A \rightarrow S$ is a transition function. The process models the ability of being in a certain state, carrying out an action, and thus ending up in a new state. Having a state and an action as input, the transition function will provide the resulting state, and the reward function will provide an immediate reward.

The goal of reinforcement learning is to find a policy which maximizes the total expected reward retrieved over the course of a given task.

For a policy π, the value function V^{π}, is a function that tells, for each state, what the expected cumulative reward will be when executing π starting in state s_t:

$$\begin{aligned} V^{\pi}\left(s_t\right) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned} \tag{2.1}$$

where t is the current time step, and γ is a constant value between zero and one, meant to discount the value of future actions. This discount factor has the effect of making it more desirable to perform “good” actions now instead of later. An optimal policy, denoted π^*, is one that always chooses the action with the highest reward.

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}\left(s\right), (\forall s)$$

The optimal action $\pi^*(s)$ for a given state s can be defined as

$$\pi^*(s) = \arg\max_{a} \left[r\left(s, a\right) + \gamma V^*\left(\delta\left(s, a\right)\right)\right] \tag{2.2}$$

I.e. the action that maximizes the immediate reward plus the discounted maximal cumulative value of the state resulting from the transition function δ. Unfortunately, this requires access to the reward function r and the transition function δ, which the agent does not have. Therefore the agent will have to learn the transition function itself.

One way to do this, is to use the *Q learning algorithm*. It works by iteratively approximating the sum of future rewards from a given state. The more iterations performed, the more precise the approximation will become. The value of the function $Q\left(s, a\right)$ is defined as the maximal discounted cumulative reward obtainable from a given state s by performing a given action a, as follows:

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \tag{2.3}$$

Using $V^*(s) = \max_{a'} Q(s, a')$ a recursive version of Equation 2.3 can be defined as

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a') \quad (2.4)$$

Notice that the right hand side of Equation 2.3 is a part of Equation 2.2. Therefore it is possible to redefine π^* as shown in Equation 2.5 using Equation 2.3 and thereby become independent of the r and δ functions:

$$\pi^*(s) = \arg\max_a Q(s,a) \quad (2.5)$$

We use the symbol $\hat{Q}$ to describe the agent's current approximation of Q. Initially all $\hat{Q}$ values are set to random values[1] or simply zero whereafter the current state s is observed. Subsequently, the following is repeated infinitely: An action a is chosen and performed, the resulting immediate reward r and state s' are observed, and $\hat{Q}(s,a)$ is updated (for s, the previous state) using the current $\hat{Q}$ value for the new state. The algorithm is described in further detail in Algorithm 1.

Algorithm 1 The Q learning algorithm for deterministic actions and rewards.

```
 1: for all s and a do
 2:   Q̂(s,a) ← 0 (or a random value)
 3: end for
 4: loop
 5:   s ← an initial state
 6:   repeat
 7:     a ← a chosen action
 8:     Perform a
 9:     r ← r(s,a)
10:     s' ← δ(s,a)
11:     Q̂(s,a) ← r + γ max_a' Q̂(s',a')
12:     s ← s'
13:   until s is a goal state
14: end loop
```

The Q learning algorithm, except the initialisation of $\hat{Q}$, represents one *episode* of learning. For each episode, the agent's initial state is randomly chosen. An episode can end, for instance, when the agent enters an *absorbing state*—a state where, no matter which action the agent performs, it will stay in that state.

2.2 Non-Determinism

So far this chapter has concerned itself with deterministic environments only. If we are instead dealing with a non-deterministic environment, which will often be

[1] [4] explains why initialising to random values causes faster converging to Q.

the case, the transition function $\delta(s, a)$ and the reward function $r(s, a)$ may not always yield the same results given the same input, we will need to extend the Q learning algorithm. In order to allow non-determinism, the following changes to the deterministic definitions are made.

V^π is redefined to be the expected value (E) over its non-deterministic outcomes of the discounted cumulative reward received by applying policy π. Hence Equation 2.1 becomes:

$$V^\pi(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

Equation 2.3 is rewritten to express the expected Q value as:

$$\begin{aligned} Q(s,a) &\equiv E\left[r(s,a) + \gamma V^*(\delta(s,a))\right] \\ &= E\left[r(s,a)\right] + \gamma E\left[V^*(\delta(s,a))\right] \\ &= E\left[r(s,a)\right] + \gamma \sum_{s'} P(s'|s,a) V^*(s') \end{aligned}$$

where $P(s'|s, a)$ is the probability that action a in state s will result in state s'. The recursive definition of Q for non-determinism (analogous to Equation 2.4) is:

$$Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \quad (2.6)$$

If e.g. the reward function is non-deterministic i.e. returns different results for the same state/action pair, our stored $\hat{Q}$ value for a given s and a would change every time our deterministic training rule is applied, even if $\hat{Q}$ is already (close to) Q. In other words; it would not converge. Therefore, the line

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

in Algorithm1 is rewritten into

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n \left[r + \max_{a'} \hat{Q}_{n-1}(s',a')\right]$$

where α_n is the learning factor for the nth iteration of the algorithm. This is an estimate of $\hat{Q}_n(s, a)$ in the nth iteration of the algorithm. The learning factor can be defined as

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

where s and a are the state and action updated during the nth iteration, and where $visits_n(s, a)$ is the total number of times this state/action pair has been visited up to and including the nth iteration. Using the learning factor has the effect of putting more emphasis on the previous approximation of $\hat{Q}$, and less on new observations. As α_n approaches zero, the $\hat{Q}$ approximation stabilises.

The fact that $\hat{Q}$ converges to Q as n approaches infinity in a non-deterministic environment is proven in [15].

2.3 Summary

This chapter has reviewed the basic concepts of reinforcement learning. The Q function was defined to, basically, encapsulate the unknown transition and reward function. Training an agent becomes a matter of storing and updating Q values for each state/action pair in the environment.

The biggest problem in reinforcement learning is scaling. This problem is illustrated very well by Table 2.1. This table shows the maximum table size that will be needed to represent a grid world environment under different circumstances.

Grid Size	**Actions**	**Agents**	$\hat{Q}$ **Table Size**
6	4	1	24
100	4	2	$40,000$
200	4	3	$32 \cdot 10^6$
500	4	5	$125 \cdot 10^{12}$

Table 2.1: Scaling of tabular Q learning in the Navigation problem.

In a world with only one agent, where there are 6 grid locations and 4 actions, the table needed to represent the world is only 24 entries. However as can be seen it scales very poorly. This is called the curse of dimensionality. Handling the curse of dimensionality is the main focus of reinforcement learning today.

In the next chapter Hierarchical Reinforcement Learning (HRL) is introduced. HRL is a technique that attempts to handle the curse of dimensionality by imposing a hierarchical structure on the problem–thereby making it possible to solve more complex problems than what is possible with Tabular Reinforcement Learning.

Chapter 3

Hierarchical Reinforcement Learning

As stated in the previous chapter tabular reinforcement learning suffers from the curse of dimensionality. In this chapter an approach to ease the state explosion is presented.

When examining a Markov decision problem one might discover hierarchical structures. The aim of Hierarchical Reinforcement Learning (HRL) is to discover, and subsequently exploit these structures. One approach that exploit hierarchical structures is the *MaxQ* method first presented by Thomas G. Dietterich [2]. In this method each subtask is defined in terms of a termination predicate and a local reward function. These define in which states a subtask terminates, and what reward should be given when such a state is encountered.

In Section 3.1 a hierarchical problem domain called the *Taxi Problem* is presented. By decomposing the Taxi problem into a hierarchical structure some opportunities for state abstraction are presented. Section 3.9 will present five general abstraction rules that can be applied to the decomposed problem. By applying these rules to the Taxi problem the state space is reduced from the 3000 state/action pairs it would take to represent it in tabular Q learning to only 632 pairs.

In this report the task of discovering the hierarchical structures will not be described, instead a predefined problem will be used, where the hierarchical task structure is given in advance. Although the task hierarchy is given in advance by a programmer, it is up to the learning system to "learn" the optimal policy for each subtask.

HRL uses temporal abstraction which means that the time, as well as the number of actions needed by a child task to complete is of no concern to the parent task. In effect the parent task only looks at the outcome of the child task, thereby abstracting away from the time taken, and actions performed at the lower levels of the hierarchy. The problem with temporal abstraction is that it cannot be represented by a regular MDP. Therefore an extension to MDPs is introduced in Section 3.2. This extension is called Semi-MDP, and is used in the MaxQ algorithm.

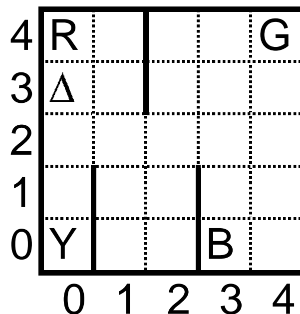


Figure 3.1: The Taxi problem.

Section 3.7 shows the MaxQ-0 learning algorithm, and Section 3.8 shows an extension to this algorithm called MaxQ-Q. When learning is complete each subtask will be an optimal policy to a sub-SMDP [1] of the original MDP. The policy for the entire problem will be a combination of the individual subtask policies.

Throughout this chapter the Taxi problem will be used frequently to elaborate on the problems encountered and to illustrate the principle behind value function decomposition.

3.1 The Taxi Problem

The Taxi problem consists of a 5×5 grid world, in which a taxi agent moves around. This world is shown in Figure 3.1. In the world there are 4 specific locations denoted with R, B, G, and Y. In each episode the taxi starts in one of the 25 grid locations. In one of the four locations(source) a passenger is waiting for the taxi to take it to a target location(destination). The agent must first navigate to the source location. Following this it must pick up the passenger. It must now navigate to the destination location, and put down the passenger. An episode ends when the passenger has been put down.

There are six primitive actions in the Taxi problem: First there are the four navigation actions, $North$, $South$, $East$, and $West$. Furthermore there are the $Pickup$ and the $Putdown$ actions. To add consequence to the primitive actions, each primitive action has a penalty of -1. Upon completion of the episode there is a reward of $+20$. If the taxi happens to navigate into a wall, then the state does not change, but a penalty of -1 is still given. Furthermore, if a $Pickup$ or $Putdown$ is performed in an illegal state the penalty given will be -10, otherwise it is the normal penalty of -1.

Overall there are 500 different states in this problem if using tabular RL: There are 25 different grid locations, 5 locations of the passenger (counting the pas-

[1] Note that an optimal sub-SMDP is not necessarily optimal in the context of the overall problem, or even in the context of the hierarchy. Section 3.6 describes this problem in further detail.

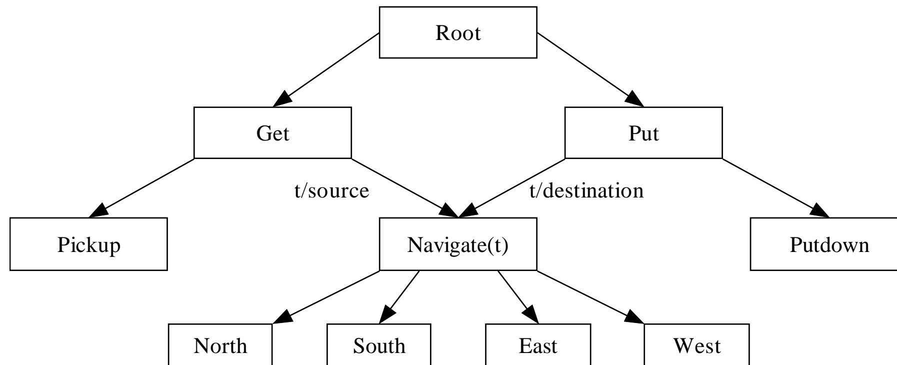


Figure 3.2: A task graph for the Taxi problem.

senger inside the Taxi as one location), and there are 4 destinations. This gives a total of 3000 state/action pairs.

Task Decomposition

It is the job of the problem developer to identify the individual subtasks that the problem should be decomposed into. In the Taxi problem the following four subtasks have been identified:

- $Navigate(t)$. The parameter t indicates which of the four locations that is the target location. In this subtask, the goal is to move the taxi from its current location to t.
- *Get*. The goal is to move the taxi from its current location to the source location, and pick up the passenger.
- *Put*. The goal is to move the taxi from the source location to the destination location, and put the passenger down.
- *Root*. This task is the overall task of the system, and represent the entire Taxi problem.

When defining a subtask there are a number of things that must be done. First a sub goal must be defined. When this goal is reached, the subtask is terminated, and control is returned to the calling task. Furthermore, it must be specified which descendants this task has, both primitive actions, and other subtasks. The decomposition of the Taxi problem is shown in Figure 3.2. The task hierarchy is a directed acyclic graph, where each node corresponds to either a composite or primitive action.

Abstractions and Subtask Sharing

State abstraction is an important concept in hierarchical learning. In the case of the Taxi problem not all state variables matter in all subtasks, e.g. when the

navigation subtask is being performed the only thing of importance is where the taxi is going, whether the passenger is in the taxi or waiting at a location cannot affect any decisions in the subtask once it has been initialised.

Now that we know that the location of the passenger does not affect the decisions in the navigate subtask, the *Get* and *Put* subtask can share the navigate subtask. This means that the system would only have to solve the navigate subtask once, meaning faster learning for the taxi agent, as well as fewer states.

Another abstraction method used is temporal abstraction. By applying temporal abstractions to the problem, actions that can take a different amount of time, depending on the state, can be seen as always taking the same amount of time. The reason why temporal abstractions is applied is to speed up the learning and planning techniques for MDPs, which is obtained by this simplification.

3.2 Semi-Markov Decision Process

Subtasks are said to be "semi-Markovian" because they can take a variable stochastic amount of time. Because of this ability an extension of MDPs is needed. A Semi-Markov Decision Process (SMDP) is a generalisation of MDPs where actions can take a variable amount of time to complete. SMDPs are introduced to handle temporal abstraction. Unlike MDPs, where state changes are only due to the action, in SMDP actions are chosen at discrete points in time and the state of the system may change continually between the selections of action.

An SMDP is a four tuple $\langle S, A, r, \delta \rangle$ where S, A and r are as in an MDP, the set of states, the set of actions, and the reward function. δ is the multi-step transition function, $\delta : S \times A \rightarrow S \times N$. SMDPs handle the varying amount of time by letting the variable N denote the number of time steps that action a requires when it is executed in state s. The transition function will then not only return a new state, but also the number of steps used to end up in that state.

3.3 MAXQ Problem Decomposition

The MAXQ decomposition takes an SMDP M and decomposes it into a finite set of subtasks$\{M_0, M_1, \ldots, M_n\}$, where M_0 is the root subtask

An unparameterised subtask is a four tuple, $\langle T_i, A_i, \widetilde{R}_i, S \rangle$ where

- T_i is the set of termination predicates that partitions S into a set of active states, S_i^A, and a set of terminal states, S_i^T. The policy for subtask M_i can only be executed if the current state s is in S_i^A. The subtask M_i terminates as soon as the SMDP enters a state in S_i^T, even if it is still executing a subtask.
- A_i is the set of actions that can be performed in subtask M_i. These actions can be both primitive actions and other subtasks, and are referred to as

the children of subtask M_i. The sets A_i define a directed acyclic graph over the subtask $M_0, \ldots, M_n$

If a child subtask has formal parameters, then this is interpreted as if the subtask occurred multiple times in A_i, with one occurrence for each possible tuple of actual value that could be bound to the formal parameters.

- $\widetilde{R}_i$ is the pseudo-reward function, which specifies a pseudo-reward for each transition to a terminal state in S_i^T. It is used to describe how desirable each of the terminating states are.

3.4 Hierarchical Policy

A hierarchical policy, π, is a set of policies containing a policy for each subtask: $\pi = \{\pi_0, \ldots \pi_n\}$.

Each policy π_i takes a state and returns an action, $\pi_i : S \rightarrow A$. The action A is either a primitive action or a subtask. Algorithm 2 gives a pseudo-code description of how a hierarchical policy can be executed. A hierarchical policy is executed using a stack. Each subtask policy takes a state and returns a primitive action to execute, or a subtask to invoke. When a subtask is invoked, its name is pushed on the stack and its policy is executed until it enters one of its terminating states. When a subtask terminates, its name is popped off the stack. If any subtask on the stack terminates, then all subtasks below the subtask in the graph are aborted, and control returns to the subtask that had invoked the terminating subtask. Hence, at any time, *Root* subtask is located at the bottom of the stack and the subtask which is currently in control is located at the top.

3.5 Projected Value Function

The projected value function of hierarchical policy π on subtask M_i, denoted $V^\pi(i,s)$, is the expected cumulative reward of executing π_i, and its descendants, starting in state s until M_i terminates. We will often denote a subtask by its index number to avoid cluttering the notations.

The purpose of the MAXQ value function decomposition is to decompose the projected value function of the root task in terms of all the subtasks.

We let $Q^\pi(i,s,a)$ be the expected cumulative reward for subtask M_i of performing action a in state s and following hierarchical policy π until subtask M_i terminates. Action a can either be a primitive action or a child subtask.

The projected value function can then be restated as

$$Q^\pi(i,s,a) = V^\pi(a,s) + \sum_{s',N} P_i^\pi(s',N|s,a)\gamma^N Q^\pi(i,s',\pi_i(s')) \tag{3.1}$$

where N is the number of steps required to complete M_i, s' is the resulting state, and $\pi_i(s')$ is the best action for M_i in state s' according to the current policy.

Algorithm 2 Pseudo-code for execution of a hierarchical policy.

1: s_t is the state of the world at time t.
2: K_t is the state of the execution stack at time t.
3: let $t = 0$; K_t = the empty stack; observe s_t.
4: push $(0, nil)$ onto stack K_t(invoke the root task with no parameters).
5: **repeat**
6:     **while** $top(K_t)$ is not a primitive action **do**
7:         let $(i, f_i) := top(K_t)$, where i is the "current" subtask, and f_i gives the parameter bindings for i.
8:         let $(a, f_a) := \pi(s, f_i)$, where a is the action and f_a gives the parameter bindings chosen by policy π.
9:         push (a, f_a) onto the stack K_t.
10:     **end while**
11:     let $(a, nil) := pop(K_t)$ be the primitive action on the top of the stack.
12:     execute primitive action a, observe s_{t+1}, and receive reward $R(s_{t+1}|s_t, a)$.
13:     **if** any subtask on K_t is terminated in (s_{t+1}) **then**
14:         let M' be the terminated subtask that is highest (closest to the root) on the stack.
15:         **while** $top(K_t) \neq M'$ **do**
16:             $pop(K_t)$.
17:         **end while**
18:         $pop(K_t)$.
19:     **end if**
20:     $K_{t+1} := k_t$ is the resulting execution stack.
21: **until** K_{t+1} is empty

The right-most term in this equation is the expected discounted reward of completing task M_i after executing action a in state s. This term is called the completion function $C^\pi(i, s, a)$.

Using this completion function the Q-function can then be expressed as

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a). \tag{3.2}$$

The V-function can be expressed as

$$V^\pi(i, s) = \begin{cases} Q^\pi(i, s, \pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s, i)R(s'|s, i) & \text{if } i \text{ is primitive} \end{cases} \tag{3.3}$$

If i is a composite action further decomposition is needed. If i is a primitive action a leaf node has been reached and further decomposition is not possible.

Equation 3.1, 3.2 and 3.3 are called the *decomposition equations* because they recursively decomposes the projected value function for the root task, $V^\pi(0, s)$, into the projected value function for the individual subtasks and the individual completion functions, and are therefore known as the decomposition equations. The only thing that needs to be stored are the C values for all non-primitive subtasks and the V value for all primitive actions. Using this decomposition and the stored values all Q values in the hierarchy can be calculated recursively.

The value of $Q^\pi(Root, s, Get)$ is the cost of executing *Get* in state s plus what it takes for *Root* to terminate once *Get* has completed.

$$Q^\pi(root, s, get) = V^\pi(get, s) + C^\pi(root, s, get)$$

The completion cost C for each node is stored. Therefore the only thing that needs to be calculated is the value of $V^\pi(Get, s)$. Applying the decomposition equations, we have that $V^\pi(Get, s) = Q^\pi(Get, s, \pi_{Get}(s))$. This decomposition continues until a primitive action is encountered. When this happens the combined values are returned recursively. Consider the scenario shown in Figure 3.1, where the taxi must navigate to R, pickup a passenger and then navigate to B and put down the passenger. Using the decomposition equations we get the following:

$$\begin{aligned} Q^\pi(root, s, get) = V^\pi(north) + C^\pi(navigate(R), s, north) + \\ C^\pi(get, s, navigate(R)) + C^\pi(root, s, get) \end{aligned} \tag{3.4}$$

This means that the complete cost of completing *Root* amounts to the cost of the primitive action $North$, plus the cost of completing $Navigate$ once a north action has been performed, plus the cost of completing *Get* once $Navigate$ has been performed, plus the cost of completing *Root* once a get has been performed. The resulting value of this using an optimal policy is

$$Q^\pi(root, s, get) = -1 + 0 + (-1) + (-8) = -10$$

Furthermore a reward of 20 is given for completing the problem resulting in an overall reward of 10.

3.5.1 MAXQ Graph

In [2] the author has developed a graphical representation of the task graph, that he has named the *MAXQ Graph*. A MAXQ graph for the Taxi problem is shown in Figure 3.3. The graph contains two kinds of nodes. Max nodes and Q nodes. The Max nodes correspond to the subtasks in the task decomposition. There is one Max node for each primitive action and one for each subtask. Each primitive Max node i stores the value of $V^{\pi}(i, s)$. The Q nodes correspond to the actions that are available for each subtask. Each Q node for parent task i, state s and subtask a stores the value of $C^{\pi}(i, s, a)$. In the MAXQ graph the children are unordered.

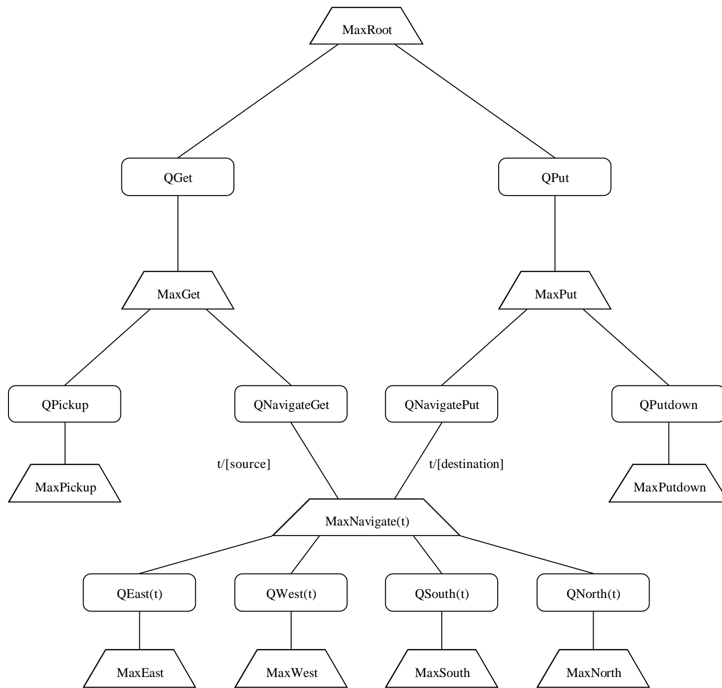


Figure 3.3: A MAXQ graph for the Taxi problem.

3.6 Optimality in MAXQ

When using MAXQ the best policy that can be expected is a recursively optimal policy. A recursively optimal policy is a policy where for each individual subtask, the policies assigned to that tasks descendants are optimal. Recursive optimality is a kind of local optimality. There is no guarantee that the overall policy resulting from this will be globally optimal, in fact often it will not be.

Consider the scenario shown in Figure 3.4, where the agent must reach the location marked with a G. The two subtasks of interest here are the task of getting out of the left room, and navigating to G in the right room. To exit

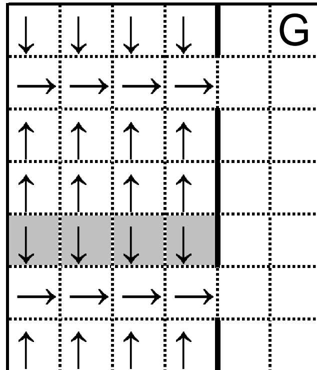


Figure 3.4: The two room maze problem.

the left room the agent must use one of the two doors. The problem when using recursive optimality is clearly illustrated by the shaded locations. Here the agent chooses the fastest way out of the room, which in terms of recursive optimality is the correct action. However, in the context of the overall policy this behaviour is suboptimal.

The major advantage of recursive optimality is that each subtask is independent of objectives that must be reached later in the execution. For example in Figure 3.4, it has no consequence where in the second room the G is located, because all that matters in recursive optimality is to find the local optima. Because of this, the policy for room one remains the same no matter where in room two the destination is located. This type of state abstraction is discussed in further detail in Section 3.9.

An advantage of recursive optimality seen from an implementation perspective is that, because of the independence of other tasks, each subtask(on the same level), can be distributed to different processes such that learning the individual policies can happen in parallel independent of parent tasks, thus optimising the speed at which learning can be done.

3.6.1 Hierarchical Optimality

Another stricter form of optimality is hierarchical optimality [11]. In hierarchical optimality the best possible overall policy given a hierarchical structure is found. To achieve this, it is necessary to provide outside information to each subtask. Consider again the scenario shown in Figure 3.4. Here it is easy to see that in the shaded area it would be wiser to move north instead of south. To achieve this the subtask needs information about which door would be the best to use from a given state. This could easily be implemented into MAXQ by adding different pseudo rewards for exiting through a certain door, i.e. different terminating states. However there is a major drawback to this approach. Implementing this reward would ruin the chance for state abstraction that exists when using recursive optimality. To illustrate this point, consider what would happen if the goal G were moved. In this case we would have to learn one policy for

each location of G, instead of the single policy needed when using recursive optimality.

So there is a tradeoff between having a high degree of state abstraction, and the quality of the created policies. In the rest of this report the recursive optimality solution is used.

One last thing to notice is that hierarchical optimality does not guarantee global optimality. The hierarchical optimality is restricted by the imposed hierarchical structure, meaning that if the hierarchical structure is not constructed correctly then a global optimal policy cannot be achieved.

3.7 The MAXQ-0 Learning Algorithm

MAXQ-0 is a recursive function that executes the current exploration policy starting at Max node i in state s. Algorithm 3 gives the pseudo code for MAXQ-0. In the algorithm t represents time steps.

The algorithm performs actions until it reaches a terminating state, at which point it returns a count of the total number of primitive actions that have been executed. To execute an action, MAXQ-0 calls itself recursively with the chosen action as a parameter. If the action is primitive, it is executed and the one-step reward $V(i, s)$ is updated and the value 1 is returned, which is used in line 10 and 12 to discount the rewards properly. If the action is not primitive, an action is chosen according to the current exploration policy. The algorithm then calls itself with the recursive call at line 10 and receives the number of steps, i.e. the number of primitive actions executed by the chosen subtask. It then uses this number to update its completion cost. If it is not in a terminating state, a new action is chosen, otherwise it terminates by returning the total number of steps used.

3.7.1 Example Execution

In the following a high level description of the MaxQ-0 algorithm is given. This description focuses on how tasks are pushed on and popped off the execution stack. When a task is popped of the stack the completion function for that task is updated. An example of this is when $North$ is popped off the stack the completion function for $C(Navigate(t), s, North)$ is updated, where s is the state as it was before the $North$ action.

When a subtask is called in the MaxQ-0 algorithm it is pushed onto an execution stack. For all problems the *Root* task will always be at the bottom of the stack, and the current task will be at the top. Consider the scenario in Figure 3.1. The taxi is just one location away from being able to pick up the passenger waiting at location R. In this example the algorithm has already chosen to go down the *Get* branch, and then further down the $Navigate(R)$ branch, meaning the execution stack consists of $[Root, Get, Navigate(R)]$, with *Root* being the bottom element, and $Navigate(R)$ the top element.

The next action to be performed is a $North$ action. Because this is a primitive action it is pushed onto the stack, and popped off after it has been executed.

Algorithm 3 Pseudo-code for the MAXQ-0 learning algorithm.

```
1:  Function MAXQ-0(MaxNode i, State s)
2:  if i is a primitive MaxNode then
3:     execute i, receive reward r, and observe result state s'.
4:     V^π_{t+1}(i,s) := (1 − α_t(i)) · V^π_t(i,s) + α_t(i) · r_t.
5:     return 1.
6:  else
7:     let count = 0.
8:     while T_i(s) is false do
9:        choose an action a according to the current exploration policy π_x(s).
10:       let N = MAXQ-0(a, s).
11:       observe result state s'.
12:       C^π_{t+1}(i,s,a) := (1 − α_t(i)) · C^π_t(i,s,a) + α_t(i) · γ^N V_t(i,s')
13:       count := count + N
14:       s:=s'
15:    end while
16:    return count
17: end if
18:
19: Main program
20: initialise V^π(i,s) and C^π(i,s,j) arbitrarily.
21: MAXQ-0(root node 0, starting state s_0)
```

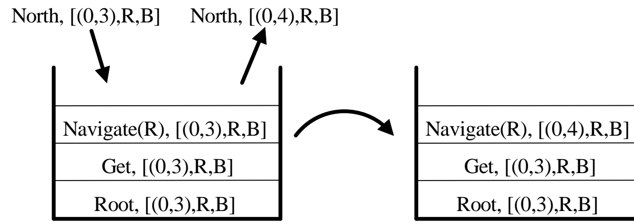
Figure 3.5(a) illustrates this. Whenever a task is popped of the stack the state s of the new top task in the execution stack, in this case $Navigate(R)$, is updated to the current state $[(0,4), R, B]$. Notice that the elements below the new top element still think s remains the same as it was when it was pushed onto the stack.

Figure 3.5(b) shows what happens next. Now that the taxi is at the location of the passenger, $Navigate(R)$ can be popped off the stack, and *Get* is updated with the new state.

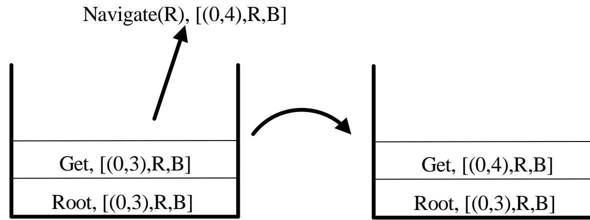
Next a *Pickup* action is executed, updating the top stack elements state to $[(0,4), T, B]$, where T means that the passenger is located inside the taxi. This update is seen in Figure 3.5(c).

Because the passenger is inside the taxi the *Get* subtask terminates, and is popped off the stack, leaving only the *Root* task on the stack. This action is shown in Figure 3.5(d).

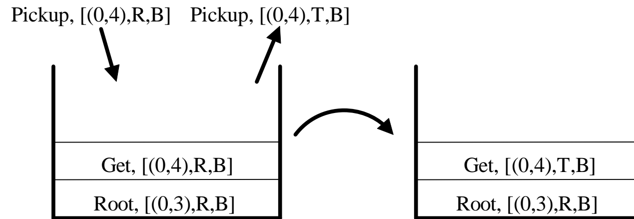
Figure 3.5(e) shows the action taken by *Root*. Because the passenger is inside the taxi the best action is *Put*. Following this the $Navigate(B)$ will be pushed onto the stack. When this is done a number of primitive actions will be called and ultimately result in $[(0,3), T, B]$. Now $Navigate(B)$ can be popped off the stack, and a *Putdown* action can be executed. Now the passenger will have reached its destination, and all that is left to do is to pop the remaining elements off the stack.



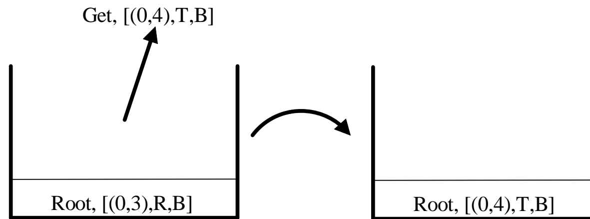
(a) Executing a *North* action.



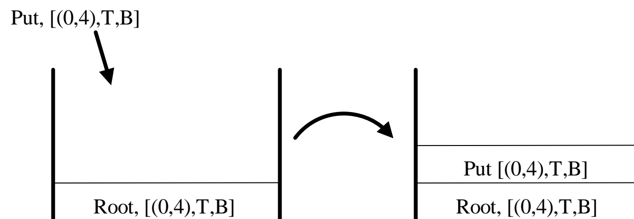
(b) Removing the *Navigate*(R) subtask from the execution stack.



(c) Executing a *Pickup* action.



(d) Removing the *Get* subtask from the execution stack.



(e) Pushing the *Put* subtask onto the execution stack.

Figure 3.5: The stack changes occurring when solving the taxi problem.

3.7.2 Requirements to the Algorithm

The pseudo code does not show how to handle "ancestor termination". When an "ancestor" subtask terminates, the descendant subtasks are interrupted and no C values are updated in any of the interrupted subtasks. An exception is if the interrupted subtask also is in a terminating state. Then the C value is updated. Ancestor termination where the C value should not be updated does not occur in the Taxi problem.

It is also not specified in the pseudo code how to calculate $V_t(i, s')$ for composite actions, since it is not stored in the Max nodes. This is solved by changing the decomposition equations to Equation 3.5 and Equation 3.6:

$$V_t(i,s) = \begin{cases} max_a Q_t(i,s,a) & \text{if } i \text{ is composite} \\ R_t(i,s) & \text{if } i \text{ is primitive} \end{cases} \tag{3.5}$$

where R is the reward function. The Q function is changed to

$$Q_t(i,s,a) = V_t(a,s) + C_t(i,s,a). \tag{3.6}$$

There are two changes compared to the decomposition equations. First $V_t(i, s)$ is defined in terms of the Q value of the best action a. This is done, because no optimal policy is present. Second, there are no π superscript, because the current value function is not based on a fixed hierarchical policy.

To find the best action a in Equation 3.5 a complete search of all paths through the MAXQ graph is needed, starting at node i and ending at the leaf nodes. Algorithm 4 gives pseudo code for a recursive function that implements a depth-first search.

Algorithm 4 Pseudo-code for greedy execution of the MAXQ graph.

1: **Function** EvaluateMaxNode(i, s)
2: **if** i is a primitive MaxNode **then**
3:    **return** $\langle V_t(i,s), i\rangle$.
4: **else**
5:    **for all** $j \in A_i$ **do**
6:       let $\langle V_t(j,s), a_j\rangle$ = EvaluateMaxNode(j,s).
7:    **end for**
8:    let $j^{best} = argmax_j V_t(j,s) + C(i,s,j)$.
9:    **return** $\left\langle (V_t(j^{best},s) + C_t(i,s,j^{best}))), a_{j^{best}} \right\rangle$
10: **end if**

The pseudo-code returns both $V_t(j^{best}, s) + C_t(i, s, j^{best}))$ and the action $a_{j^{best}}$ at the leaf node that achieves this value.

The exploration policy π_x from the Pseudo-code of MAXQ-0 also needs to be specified. It must be an ordered GLIE policy.

An ordered GLIE policy (Greedy in the Limit with Infinite Exploration) is a policy that converges in the limit to an ordered greedy policy, which is a greedy policy that imposes an arbitrary fixed order ω on the available actions and break ties in favour of the action a that appears earliest in that order.

This property ensures that MAXQ-0 converges to a uniquely defined recursively optimal policy. A problem with recursive optimality is that in general, each Max node i will have a choice of many different locally optimal policies. These different policies will all achieve the same locally optimal value function, but can result in different probability transition functions $P(s', N|s, i)$. These differences may lead to better or worse policies at higher levels of the MAXQ graph, even though they make no difference inside subtask i. When using an ordered GLIE policy it is certain that the same local optimal policy is obtained each time thus avoiding the problem.

3.8 The MAXQ-Q Learning Algorithm

The MAXQ-0 learning algorithm only works when the pseudo-reward function, $\widetilde{R}$, is zero. When it is different from zero the MAXQ-Q algorithm should be used instead. The MAXQ-Q algorithm maintains two completion functions. One, $\widetilde{C}$, to use inside the subtasks, and one, C, to use outside of the subtasks. $C(i, s, a)$ is the completion function which is also used in the MAXQ-0 algorithm and is the one used by the parent tasks to compute $V(i, s)$. The other completion function, $\widetilde{C}(i, s, a)$, is only used inside task i to find the locally optimal policy. This function will use both the real reward function, $R(s'|s, a)$, and the pseudo-reward function, $\widetilde{R}(s'|s, a)$. The pseudo code for MAXQ-Q is shown in Algorithm 5.

The MAXQ-Q algorithm maintains a stack containing the sequence of states visited in each node. If the algorithm is called with a primitive action the state s is pushed on the stack, the action is performed and the value function is updated. Then the stack is returned. If the algorithm is called with a composite action an action a is chosen using the current exploration policy. The algorithm then calls itself recursively storing all the states visited while executing the chosen action a. Then the completion functions are updated with the states visited. If the resulting state is a terminating state the stack is returned.

3.8.1 All-States Updating

In Algorithm 5 *All-States Updating* is performed. All-states updating is a way of speeding up learning by updating the completion function for more than one state at a time. When action a is chosen for Max node i the execution of a will move the environment through a series of states (one if the action is primitive). A property of SMDPs is that action selection for subtask i in state s is independent of previous states of the environment. This means that no matter in which of the intermediate states between s and the resulting state s' the agent starts in, the result will always be s'. Knowing this, we can update the completion function for all of the intermediate states using lines 15-17 in Algorithm 5.

This approach can be applied to MaxQ-0 as well.

Algorithm 5 Pseudo-code for the MAXQ-Q learning algorithm.

1: **Function** MAXQ-Q(MaxNode i, State s)
2: let seq =() be the sequence of states visited while executing i.
3: **if** i is a primitive MaxNode **then**
4: execute i, receive reward r, and observe result state s'.
5: $V_{t+1}(i, s) := (1 - \alpha_t(i)) \cdot V_t(i, s) + \alpha_t(i) \cdot r_t$.
6: push s onto the beginning of seq
7: **else**
8: let $count = 0$.
9: **while** $T_i(s)$ is false **do**
10: choose an action a according to the current exploration policy $\pi_x(s)$
11: let $childseq$ = MAXQ-Q(a, s), where $childseq$ is the sequence of states visited while executing action a.
12: observe result state s'
13: let $a^* = argmax_{a'}[\widetilde{C}_t(i, s', a') + V_t(a', s')]$.
14: let $N = 1$.
15: **for all** $s \in childseq$ **do**
16: $\widetilde{C}_{t+1}(i, s, a) := (1 - \alpha_t(i)) \cdot \widetilde{C}_t(i, s, a) + \alpha_t(i) \cdot \gamma^N[\widetilde{R}_i + \widetilde{C}_t(i, s', a^*) + V_t(a^*, s')]$.
17: $C_{t+1}(i, s, a) := (1 - \alpha_t(i)) \cdot C_t(i, s, a) + \alpha_t(i) \cdot \gamma^N[C_t(i, s', a^*) + V_t(a^*, s')]$.
18: **end for**
19: **append** $childseq$ onto the front of seq.
20: $s := s'$.
21: **end while**
22: **end if**
23: **return** seq.

3.9 State Abstraction

As mentioned in the introduction to this chapter one of the motivating factors for decomposing a problem into a hierarchical structure is the opportunities it presents for state abstraction in the individual subtasks. In the following, five rules under which abstraction can be safely applied are presented. To illustrate the use of these rules the Taxi problem will be used to show how they can be applied in practice.

A state s is represented as a vector of state variables. In the Taxi problem the state variables it takes to describe the entire problem are

$$[(x, y), P, D]$$

where (x, y) is the coordinate of the taxi. In the current setup of the taxi problem the world consists of a 5×5 grid resulting in 25 possible locations of the taxi. The P represents the location of the passenger. The passenger can be in five different locations namely on R, G, B, Y, or inside the taxi T. The destination of the passenger is represented by D. There are four possible destinations of the passenger; R, G, B, and Y.

The values that needs to be represented are the values of the value function for each primitive action, as well as what it takes to complete a task once a subtask has completed.

In a state, all state variables are not necessary for all subtasks. State abstraction is the task of only representing the state variables that are relevant for a given subtask and abstracting away from the others; e.g. the destination of the passenger is irrelevant for the *Pickup* task. The state abstractions are used for the state given to the completion function and the state given to the value function of each primitive action. The job of state abstraction is to reduce the number of values needed for each of these. The list below shows in which categories the values are distributed.

$C^\pi(Root, s, Get)$	$C^\pi(Root, s, Put)$
$C^\pi(Get, s, Navigate(t))$	$C^\pi(Get, s, Pickup)$
$C^\pi(Put, s, Navigate(t))$	$C^\pi(Put, s, Putdown)$
$C^\pi(Navigate(t), s, North)$	$C^\pi(Navigate(t), s, South)$
$C^\pi(Navigate(t), s, East)$	$C^\pi(Navigate(t), s, West)$
$V^\pi(North, s)$	$V^\pi(South, s)$
$V^\pi(East, s)$	$V^\pi(West, s)$
$V^\pi(Pickup, s)$	$V^\pi(Putdown, s)$

Without state abstraction the states would look as in Figure 3.6. The number shown after each completion function and value function is the number of possible values for the given completion function or value function.

As can be seen the total number of values to be represented by the completion functions and value functions when no abstraction is applied will far exceed that which is required by tabular learning. In tabular learning we have the six actions, the 25 locations of the taxi, the five locations of the passenger and the four destinations for a total of 3000 values. For each completion function, the

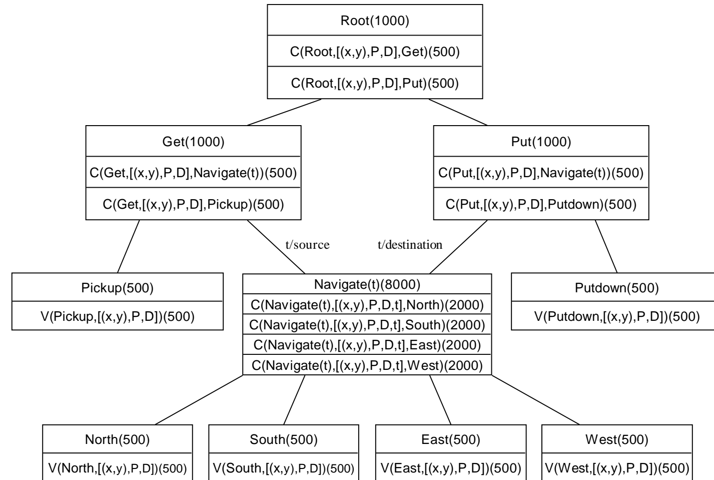


Figure 3.6: The states of each task when no state abstraction is applied.

taxi location, the passenger location, and the destination would be needed, thus the number of states in each subtask would be 500. For *Root* with two child tasks the completion functions would require a total of 1000 values. The *Get* and *Put* subtasks each has two child actions. This would mean $2 \cdot 500$ values for each task. The *Navigate* subtask has 4 child subtasks, but the parameter t can also take on four different values, bringing the number of values needed for navigate up to 8000. Finally, for each primitive action 500 values would be needed, bringing the total number of values to be represented in the taxi problem up to 14000.

In the following we will show how, by applying the safe abstraction rules, the total number of values can be reduced to only 632. In some of the rules we will refer to the state variable *Source*. Source is a special case of the passenger variable. It is used in cases where the passenger can never be found in the taxi, thus there is no need for a fifth value.

3.9.1 Leaf Irrelevance

A set of state variables is irrelevant to a value function for a primitive action if, no matter the configuration of the variables, the reward remains the same.

For all the primitive navigation actions the only penalty that can be given is -1, therefore it is only necessary to represent one value in each of these actions. For the pickup and putdown actions there are two values, namely, -1 if the taxi is in a legal state, and -10 if it is in an illegal state. For pickup a legal state is when the taxi is in the same location as the passenger. For putdown it is when the taxi is at the destination, and the passenger is in the taxi. By applying leaf irrelevance we have reduced the number of values needed to represent the value functions from 3000 to only 8. Figure 3.7 shows the updated task graph.

There are no explicit state variables called *Legal* or *Illegal*. These variables are

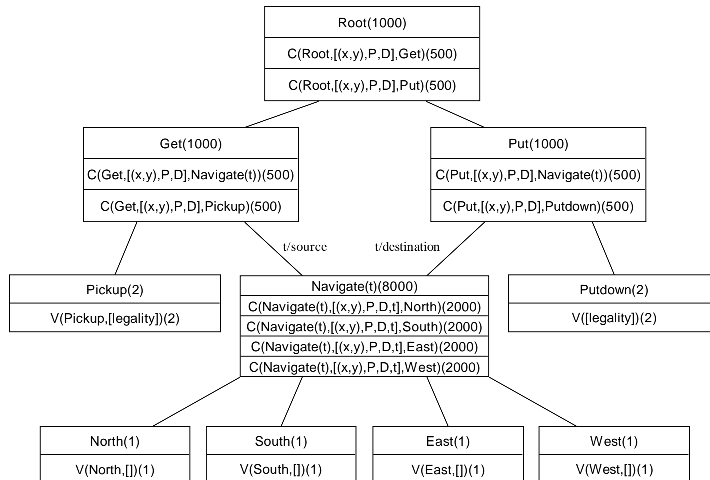


Figure 3.7: The states of each task after leaf irrelevance has been applied.

thought of as abstractions over a set of state configurations. Any given state can be categorised as either a legal state, or an illegal state.

3.9.2 Subtask Irrelevance

There are two conditions that must be satisfied before a variable can be abstracted away using subtask irrelevance:

1. *If a variable is not needed to represent the parent task itself, then the variable can be removed.*
2. *If a state variable is not used by any of the child tasks, then that variable can be removed using subtask irrelevance.*

Together these two conditions make up the subtask irrelevance abstraction rule.

Starting from the bottom of the task graph the first place where subtask irrelevance can be applied is in the completion functions for $Navigate$. Applying the first rule P and D are not needed in the parent task. Because the child subtasks does not have any variables in its state, the best state we can achieve in $Navigate$ by applying subtask irrelevance is $[(x, y), t]$.

In *Get* the destination of the passenger can be abstracted away, leaving only the passenger location. Furthermore, in the case where the passenger is inside the taxi does not need to be represented because whenever the passenger is in the taxi *Get* will have terminated. Much the same happens in Put, where the passenger location can be removed.

For *Root* no variables can be removed using subtask irrelevance because according to condition (2) no variables used in a subtask can be removed. Figure 3.8 updates the graph to also include subtask irrelevance.

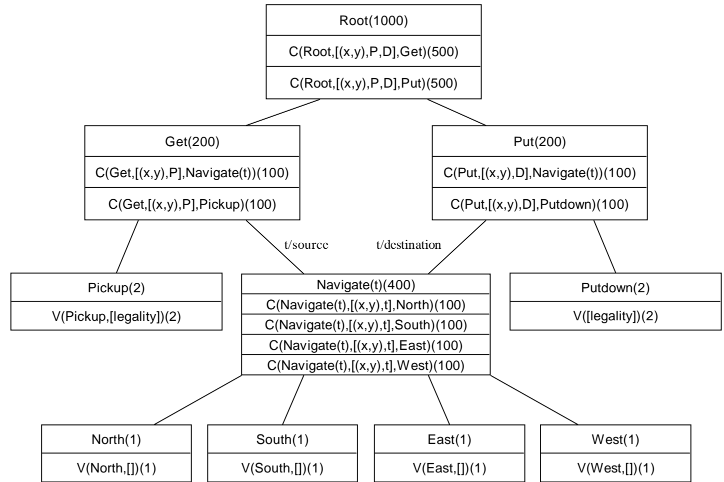


Figure 3.8: The states of each task after subtask irrelevance has been applied.

At this point, after applying leaf and subtask irrelevance the total number of values have been reduced to 1808.

3.9.3 Result Distribution Irrelevance

The rule of result distribution irrelevance states that if the state variables can be divided into two disjoint sets X and Y. The variables in Y are irrelevant for the result distribution of an action if the actions result state remains the same no matter the value of Y. When this is the case, Y can be abstracted away.

For the completion function for *Put* and *Get* when a navigate is completed, it is true that no matter what location the taxi starts in, it will always end up in the same state, thus it will always yield the same result. Therefore the taxi location can be abstracted away for both *Get* and *Put* in the *navigate* child instance. Doing so leaves only the source for *Get*, and the destination for *Put*.

In the *Root* subtask result distribution irrelevance can also be applied. Here the taxi location can be abstracted away, leaving only the source location and the destination. The graph created after applying result distribution irrelevance is shown in Figure 3.9.

At this point, after applying leaf and subtask irrelevance the total number of states have been reduced to 1808. Now that result distribution irrelevance has also been applied, the total number of values needed to be represented is 648.

3.9.4 Termination

The termination rule can be applied when termination of a subtask guarantees the termination of the parent subtask. If this is the case, then the completion cost $C^{\pi}(i, s, a)$ equals zero, and does not need to be represented.

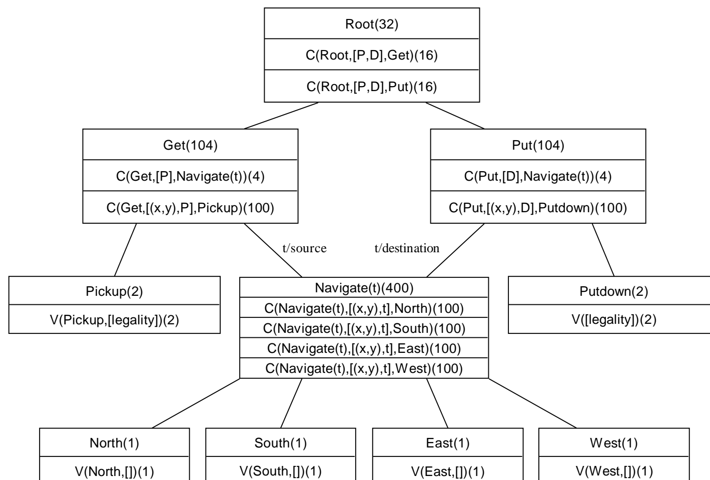


Figure 3.9: The states of each task after result distribution irrelevance has been applied.

In the taxi task it holds that, in all states where the passenger is located at the destination, *Put* will succeed and result in *Root* terminating. What this rule is saying is that if the termination predicate is the same for both parent and child the termination rule can be applied.

3.9.5 Shielding

If there for subtask M_i exists a state s, such that for all paths from the root of the MaxQ graph down to, and including, M_i exists a subtask that is terminated, then there is no need to represent the completion cost for this state because it can never be executed.

In the *Put* task it is the case that whenever the passenger is not in the taxi the completion cost of *Put* or any of its ancestor tasks (i.e. *root*) will be zero. Because of this and that the termination rule *Put* does not need to represent any completion costs at all. No matter the state, the cost will be zero.

The termination and shielding rules in combination makes it so that no values needs to be represented for the completion cost for root once a put has been completed. This was the last termination rule. The final result can be seen in Figure 3.10, here the total number of values needed to be represented is 632.

3.10 Test of HRL

In order to test HRL, the Taxi problem from Section 3.1 was implemented. During learning the taxi can start in any of the 25 locations. The source and the destination are randomly chosen(the source location is never the same as the destination location).

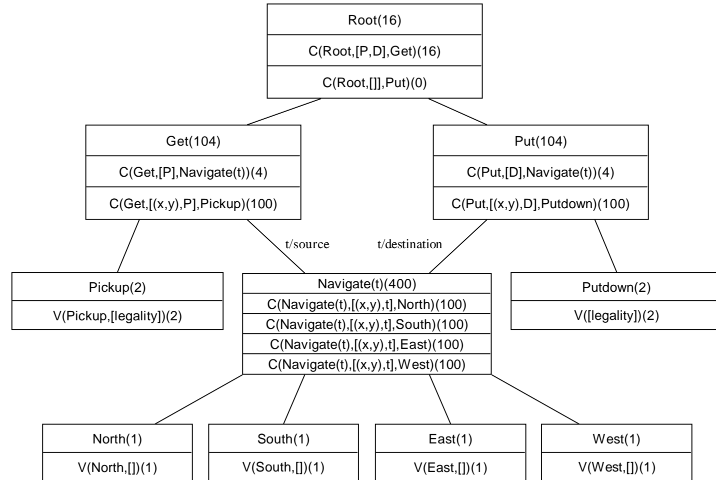


Figure 3.10: The resulting states after all abstraction rules has been applied.

When testing the learnt policy, the taxi will start in location $(2,2)$ and navigate to the Y location $(0,0)$, pickup the passenger, and then navigate to the B location $(3,0)$.

A trial run consists of first a learning episode. When this episode is completed the learnt policy is tested. If the testing goal has not been reached another learning episode followed by a test is performed.

Each experiment consists of a number of trials. This is because we want to take into consideration cases where the policy converges extraordinarily fast and extraordinarily slow.

When initialising a trail, the first episode chooses its actions completely at random. For each learning episode following this an exploration variable x decreases thus reducing the chance of a random action being chosen. The way x is calculated is shown in Equation 3.7

$$x = \begin{cases} (1 - \frac{iterations}{500}) & \text{if } (1 - \frac{iterations}{500}) > 0.2 \\ 0.2 & \text{if } (1 - \frac{iterations}{500}) \leq 0.2 \end{cases} \tag{3.7}$$

when selecting the action x is compared to a random number between 0 and 1. If x is bigger than this number a random action is chosen.

The learning factor α from line 12 in Algorithm 3 is shown in Equation 3.8.

$$\alpha = \begin{cases} (1 - \frac{visits}{f}) & \text{if } (1 - \frac{visits}{f}) > 0.2 \\ 0.2 & \text{if } (1 - \frac{visits}{f}) \leq 0.2 \end{cases} \tag{3.8}$$

The *visits* parameter indicates how many times the state/action pair has been visited previously, and f is a constant. In the tests f is changed in an attempt to find the fastest convergence for the given starting state. As the state is visited

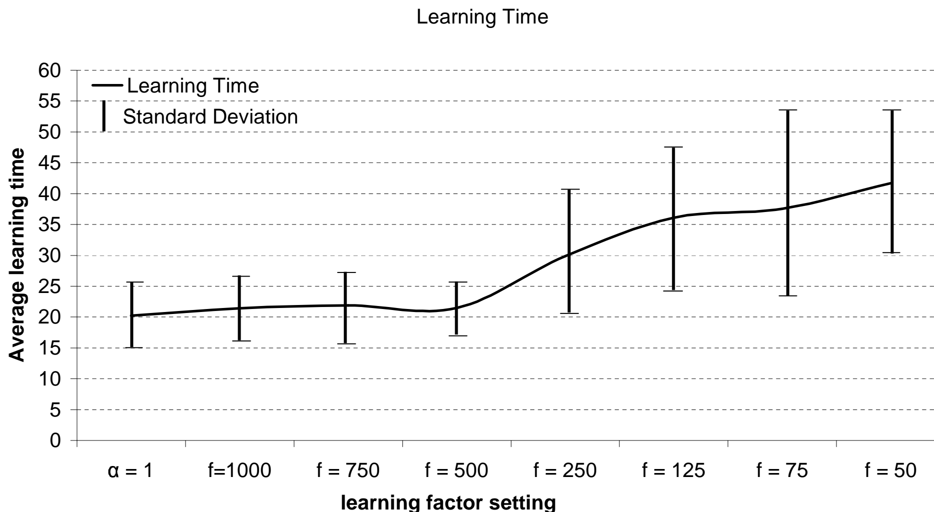


Figure 3.11: Results for different settings of f, and the standard deviation for each setting.

more and more times, the value returned becomes lower and lower, meaning more emphasis is put on old data and less on new data.

Because the taxi problem is deterministic, meaning any one state/action pair leads to only one resulting state, the learning factor only serves to delay convergence. Figure 3.11 shows the average number of episodes needed for an optimal policy to be found under different settings of f. The values for each setting is an average found from 100 trails. As can be seen the value of the deviation more or less increases as the value of f decreases.

On average when running the algorithm with α set to 1, meaning no emphasis on old data, the average number of episodes to find an optimal policy was 20.22, whereas the closest competitor was $f = 1000$, which converged at an average of 21.43. It is also clear that the lower the value of f, the higher the convergence time. Therefore the optimal update function in Algorithm 3 would be:

$$C^{\pi}_{t+1}(i,s,a) \;=\; (1-1)\cdot C^{\pi}_{t}(i,s,a) + 1\cdot \gamma^{N}\cdot V_t(s',i) \;=\; \gamma^{N}\cdot V_t(s',i) \quad (3.9)$$

3.10.1 The Taxi Problem and Tabular Reinforcement Learning

To show how the HRL algorithm performs compared to the tabular approach, the TRL algorithm described in Chapter 2 has been implemented. In the following the two implementations are compared. In the implementation of the tabular approach the taxi is given a reward of 1 at the end of the episode instead of a reward each time the taxi performs an action.

Furthermore, some post execution optimising have been made to speed up learning. First instead of updating $\widehat{Q}$ during execution of the algorithm each

state/action pair encountered is pushed onto a stack. Upon completion of the algorithm the state/action pairs are popped off the stack and each is updated according to their place N in the stack, meaning the top pair is given a reward of 1, $r_t = 1$, while the next is given a reward of $r_{t+1} = r_t \cdot \gamma^N$, where . The state/action pairs along with their assigned values are then compared to the values in $\widehat{Q}$, and if they are higher they replace the old. Using this setup the values propagate a lot faster through the $\widehat{Q}$ table, which in turn means faster convergence.

In the HRL experiment the exploration policy selects an exploration variable based on the number of trials performed, thus increasing the chance for a learnt action to be chosen. In the TRL experiment the probability distribution seen in [4] is used. This formula is shown in Equation 3.10.

$$P(a_i|s) = \frac{k^{-\widehat{Q}(s,a_i)}}{\sum_j k^{-\widehat{Q}(s,a_j)}} \qquad where \quad k > 0 \tag{3.10}$$

In Equation 3.10 k is a constant. Setting k close to zero means more exploiting, and closer to one means more exploration. In this experiment k was set to 0.5.

In average the TRL algorithm converged in 28.3 episodes. This is more than 8 episodes more than what required by HRL. However it should be mentioned that the execution of a single episode is faster in TRL. This could, in part, be due to the implementations, but mainly because there are much fewer calculations in the tabular approach. This advantage of TRL is expected to diminish as the table gets bigger, and each lookup becomes more expensive.

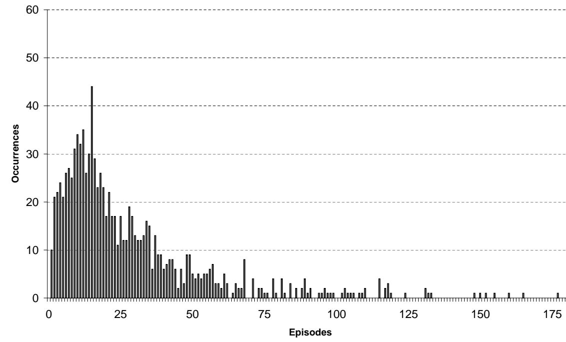
Looking at the distribution of trials over the number of episodes it takes to reach a perfect path for the taxi in the testing example, there is a clear difference between the two algorithms. Figure 3.12 shows the result distribution collected based on 1000 trials.

The two figures does not show how long it takes for the entire problem to converge but how long it takes the algorithm to find an optimal policy for the problem mentioned in the beginning of Section 3.10 i.e. for navigating from $(2,2)$ to $(0,0)$, picking up the passenger, and then navigating to $(3,0)$ and putting down the passenger.

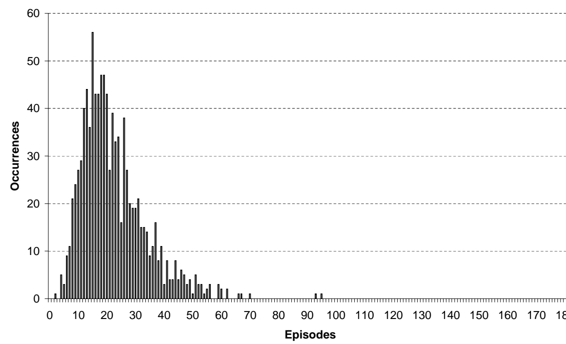
In Figure 3.12 the x axis shows the number of episodes it takes for a trial to converge. The y axis shows how many times each of these trials occurs. It is clear that the result distribution from the TRL algorithm is much more scattered than that of HRL, meaning that for HRL, a fewer number of iterations is needed for the average episode count to stabilise.

One might argue that a reduction of 8 time steps in the HRL approach compared to the TRL approach does not make up for the added number of calculations. In fact, this claim is confirmed when looking at the time it takes to execute a trial. in the tabular approach a single trial takes on average 80.05 miliseconds, whereas the hierarchical approach takes 331.98. Because of this the tabular approach will be the best to use *on this problem* if the number of trials, and the space used is of no consequence.

This experiment is not very encouraging for the hierarchical approach. However, this problem might be too small to really demonstrate the power of hierarchical



(a) TRL occurrence distribution



(b) HRL occurrence distribution

Tabular	
Time in milliseconds for one trial	80,05
Mean	28,0961
Median	19
Standard Deviation	26,96299
Minimum	1
Maximum	182
Count	1000

(c) Table based statistics

Hierarchical	
Time in milliseconds for one trial	331,98
Mean	22,35335
Median	20
Standard Deviation	11,28952
Minimum	2
Maximum	93
Count	1000

(d) Hierarchical based statistics

Figure 3.12: Occurrence distribution and statistics for the 5 by 5 grid Taxi problem.

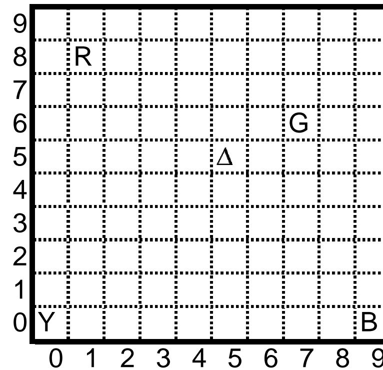


Figure 3.13: The 10 by 10 grid.

reinforcement learning. The next section compares HRL and TRL on a scaled up instance of the taxi problem.

Testing on a Larger Problem

To test the scaling abilities of both approaches, the locations that the taxi can navigate was increased from 25 to 100, that is, from a 5×5 grid to a 10×10 grid. Doing this increases the number of values to be represented by TRL to 12000, $(10 \cdot 10 \cdot 5 \cdot 4 \cdot 6)$, and the number of values for HRL to 2432. In HRL the only changes occurs in completion functions where the state requires the taxi location.

As a consequence of the new grid, the source and destination locations needs to be moved. The change is as seen in Figure 3.13.

When testing the learnt policy the taxi will start in location $(5, 5)$ navigate to Y, pickup the passenger and navigate to B, where it will put the passenger down.

Figure 3.14 shows the statistics generated for the TRL and HRL experiment. As can be seen the HRL implementation is a lot faster than its tabular counterpart. In fact the hierarchical implementation only uses in average 61.914 episodes to find an optimal path, whereas the tabular implementation uses in average 477.79.

What is even more interesting is the average time it takes to execute a trial. In HRL a trial takes on average 786.43 milliseconds, whereas TRL takes on average 12888.23 milliseconds. The hierarchical solution only increased the time spend with a factor of 2.37 when changing from the small board to the big board. The tabular solution increased the time spent with a factor of 161.1. Looking at these numbers it is not difficult to imagine what will happen when problems get even larger.

The explanation of why the tabular implementation performs so much worse than its hierarchical counterpart when scaling up, has to do with the number of foolish actions that can be performed by TRL as opposed to that of HRL. As

TRL	
Time in milliseconds for one trial	12888,23
Mean	477,79
Median	299
Standard Deviation	539,9542
Minimum	1
Maximum	3234
Count	500

(a) Table based statistics.

HRL	
Time in milliseconds for one trial	786,43
Mean	61,914
Median	59
Standard Deviation	20,26158
Minimum	17
Maximum	133
Count	500

(b) Hierarchical based statistics.

Figure 3.14: Statistics for the 10 by 10 grid Taxi problem.

an example consider the state $[(6,6), Y, B]$. In this state the taxi is in location $(6,6)$. The passenger is at location Y, meaning it has not yet been picked up. The destination is at B. In HRL when in the *Get* subtask there are only 2 possible actions, namely *Pickup* and *Putdown*. For the same state in TRL there are 6 possible actions, meaning the possibility of performing a foolish action is much greater. Because the actions can be performed in all states, the number of actions performed in TRL, when not much have been learnt yet, will be much larger than what will be the case in HRL. Because the number of grid locations is 4 times greater in the 10×10 instance the disadvantages of TRL becomes that much more apparent than they were in the 5×5 instance.

3.11 Summary

In this chapter the hierarchical reinforcement learning algorithm introduced by Dietterich was presented. As can be seen by the results this method offers a lot compared to tabular Q learning when it comes to time and space issues. The advantages just becomes more and more apparent when the problems gets more complex.

In the next chapter this algorithm will be expanded to handle more than one agent. The transition from single to multi-agent learning presents a number of new challenges that needs to be addressed. These challenges are described in great detail.

Chapter 4

Homogeneous Multi-Agent Hierarchical Reinforcement Learning

Optimal behaviour for an agent in a multi-agent system will usually depend on the behaviour of the other agents in the system and their behaviour will often not be predictable.

Multi-agent learning studies algorithms for selecting actions in environments, where multiple agents coexist. One thing that makes this a complicated problem is that the behaviour of other agents will change as they adapt their policy to achieve their own goals. Therefore, the environment will be highly non-stationary at the beginning of the learning process, and only gradually become stable.

Multi-agent learning is challenging for two main reasons: *Curse of dimensionality*, the number for parameters to be learned increases dramatically as the number of agents increases, and *partial observability*, the states and actions of the other agents which are required for an agent to make optimal decisions may not be fully observable and communication between the agents can be costly. We seek to minimise the effect of these two problems by using HRL. An initial hierarchical task structure is assumed given in advance. Since we in this report are only interested in cooperative agents, we define a subtask to be either a cooperative subtask, in which the agents can communicate, or a non-cooperative subtask, in which the agents are unaware of each other. Cooperative subtasks are usually defined at the highest levels of the hierarchy, because the lower level the tasks are, the more primitive the task will be. In the Taxi problem it would be strange to let the lowest level consist of cooperative subtasks, since primitive actions should not depend on the other agents.

4.1 Introduction to the Approach Used

The work in this chapter on *Multi-Agent Hierarchical Reinforcement Learning* (MHRL) elaborates on the work done by *Ghavamzadeh* and *Madadevan* in [8]. It also expands the framework for HRL presented in Chapter 3. In *Multi-Agent Hierarchical Reinforcement Learning* the authors presents Algorithm 7 and the projected value function in the MHRL setting.

The idea is that by letting cooperative agents see the high level actions (i.e. the actions located high in the task graph) of other agents, coordination can be achieved. Because the agent only knows the high level actions of the other agents, there is no coordination of primitive tasks, thus reducing both the complexity of the implementation as well as the communication cost and hence the time needed to learn an optimal policy.

Below four advantages to this approach are listed.

- The explicit task structure of HRL will speed up the learning because of improved scaling ability.
- Since it would be normal only to communicate in high-level tasks and that high-level tasks can take a long time to complete, communication is needed less often.
- That the agents only communicate at the higher levels will also speed up the learning, since the agents will not have to worry about the low level details of other agents.
- The agents only need local state information, the locations of the other agents do not need to be known. This is based on the idea that the agent can get a rough idea of what states the other agents might be in just by knowing about the high level action being performed by the other agents.

However this approach is not with out drawbacks. The single largest drawback of using this approach is

- Since agents only coordinate at a high level, the policy achieved might not be the optimal policy.

This drawback is often encountered when trying to limit coordination between agents. It is a tradeoff between how fast we want to learn and how close to an optimal policy we want to get.

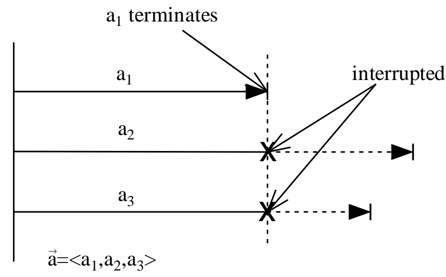
When executing a multi-agent learning algorithm the communication needed between the agents amounts to finding out what high level actions that are being performed by other agents. We assume communication is free, reliable, and instantaneous, i.e. the communication cost is zero, no messages are lost in the environment, and each agent receives information about the other agents before taking the next action.

4.2 Multi-agent Semi-Markov Decision Process

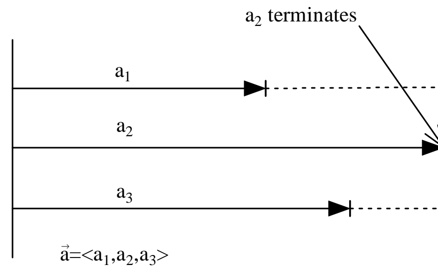
A multi-agent SMDP (MSMDP) [5] is a six tuple, $\langle \alpha, S, A, \delta, R, \tau \rangle$ where

- A is the set of actions.
- α is the set of n agents, with each agent $j \in \alpha$ having a finite set A^j of individual actions. An element $\vec{a} = \langle a^1, \dots, a^n \rangle$ of the joint-action space $\vec{A} = A^1 \times \dots \times A^j$ represents the concurrent execution of action a^j by each agent j.
- S and R are as in a SMDP, the set of states and the reward function.
- δ is the multi-step transition function, $\delta : S \times \vec{A} \rightarrow S \times N$. The difference from the transition function in SMDP is the $\vec{A}$ which consists of an action from each agent in the environment. The variable N denotes the number of time steps that the joint action $\vec{a}$ requires when it is executed in state s.
- τ is the termination strategy. Since the actions in the joint-action are temporally extended, they may not terminate at the same time. Therefore δ depends on how the decision epochs are defined.

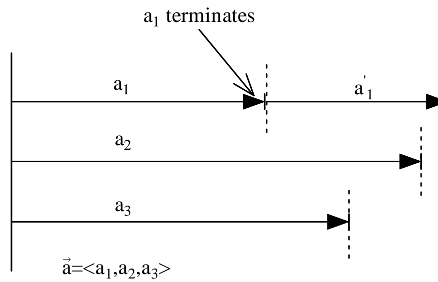
Three termination strategies τ_{any}, τ_{all} and $\tau_{continue}$ for temporally extended joint-actions were investigated in [12]. In the τ_{any} termination strategy the decision epoch terminates when one of the actions in the joint-action has terminated, thereby interrupting the other actions, meaning that when one agent finishes an action in a shared subtask all agents will terminate their actions in the shared subtask, even if they have not completed their tasks. In the τ_{all} termination strategy the decision epoch terminates when all of the actions in the joint-action have terminated. When an agent finishes its action it is idle until all the agents have finished their actions. In the $\tau_{continue}$ termination strategy the decision epoch ends when the first action within the joint-action currently being executed terminates. As opposed to τ_{any} the other agents are not interrupted in their actions and only the terminated agent selects a new action. This means that the agents do not necessarily share decision epochs. Figure 4.1 illustrates the difference between the three termination strategies. The termination strategies can be categorised as either synchronous or asynchronous. τ_{any} and τ_{all} are synchronous because all agents make a decision at every decision epoch. $\tau_{continue}$ is asynchronous since an agent does not need to wait for the other agents before choosing its next action. The choice of termination strategy depends on the problem at hand. Since it is natural to let the action selections in the Taxi problem be asynchronous we assume in the rest of this report that the $\tau_{continue}$ termination strategy is used, meaning that an agents epoch is not affected by another agent terminating its task, except for the influence it will have on its policy.



(a) τ_{any}: The decision epoch ends when an action terminates. The terminating action a_1 causes actions a_2 and a_3 to be interrupted. A new decision epoch begins afterwards.



(b) τ_{all}: The decision epoch ends when all the actions have terminated. Afterwards a new decision epoch can begin. No new actions are chosen before all agents have terminated their current action.



(c) $\tau_{continue}$: Each agent has its own decision epochs and starts a new one after each action.

Figure 4.1: The three termination strategies.

4.3 Task Decomposition

While an MSMDP provides the theoretical basic for temporal abstraction in the multi-agent setting it does not specify how tasks can be broken up into subtasks. A subtask is defined in one of two ways. Either a subtask is cooperative or it is non-cooperative. In a cooperative subtasks the agents are able to communicate. In the non-cooperative the agents does not need any knowledge of each other and are therefore not able to communicate. In this report we are only interested in cooperative agents and therefore assume that the agents do not wants to work against each other. If a subtask is a non-cooperative subtask, it is represented as in the single agent environment. If the subtask is a cooperative subtask, it is represented as a six tuple $\left\langle \alpha, T_i, A_i, \widetilde{R}_i, S_i, \tau_{continue} \right\rangle$. The difference from the non-cooperative subtask is the two added elements α and $\tau_{continue}$. S_i is defined as the single-agent state space. This is an approximation that greatly simplifies learning, and it is based on the thought that an agent can get a rough idea of what state the other agents are in just by knowing which high-level action they are performing [5]. A_i is the joint action space over the actions available to the agents in subtask i.

4.4 Hierarchical Multi-Agent Policy

Algorithm 6 gives a pseudo-code description of how a hierarchical policy can be executed in a multi-agent environment. In the algorithm each agent makes one primitive action before passing the control to the next agent. The primitive action is found like in Algorithm 2.

4.5 Projected Value Function

The projected value function of a hierarchical policy π on subtask M_i in the multi-agent setting, denoted $V^{\pi}(i, s, \vec{a})$, is the expected cumulative reward of executing π_i, and its descendents, starting in state s until M_i terminates.

The function decomposition is like in Section 3.5 except for the completion function for a cooperative subtask.

The joint completion function for agent j, $C^j(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j)$ is the expected discounted cumulative reward of completing cooperative subtask i after taking action a^j in state s while other agents are performing subtasks $a^k, \forall k \in 1, \ldots, n, k \neq j$. The reward is discounted back to the point in time where a^j begins execution.

The projected value function can be restated as Equation 4.1 if a^j is a non-cooperative subtask.

$$Q^{j^{\pi}}(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) = V^{j^{\pi}}(a^j, s) + C^{j^{\pi}}(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) \quad (4.1)$$

Algorithm 6 Pseudo-code for execution of a hierarchical multi-agent policy.

1: α is the set of n agents.
2: s_t is the state of the world at time t.
3: $K_t^1, \ldots, K_t^n$ are the states of the execution stacks at time t for the agents $1, \ldots, n$.
4: let $t = 0$; $K_t^1, \ldots, K_t^n$ = the empty stack; observe s_t.
5: push $(0, nil)$ onto each stack $K_t^1, \ldots, K_t^n$(invoke the root task with no parameters).
6: **repeat**
7: Choose an agent j with a non-empty execution stack according to a round robin method.
8: **while** $top(K_t^j)$ is not a primitive action **do**
9: let $(i, f_i) := top(K_t^j)$, where i is the "current" subtask, and f_i gives the parameter bindings for i.
10: **if** i is a non-cooperative task **then**
11: let $(a, f_a) := \pi(s, f_i)$, where a is the action and f_a gives the parameter bindings chosen by policy π.
12: **else**
13: let $(a, f_a) := \pi(s, f_i, \vec{a})$, where a is the action and f_a gives the parameter bindings chosen by policy π.
14: **end if**
15: push (a, f_a) onto the stack K_t^j.
16: **end while**
17: let $(a, nil) := pop(K_t^j)$ be the primitive action on the top of the stack.
18: execute primitive action a, observe s_{t+1}, and receive reward$R(st + 1|s_t, a)$.
19: **if** any subtask on K_t^j is terminated in (s_{t+1}) **then**
20: let M' be the terminated subtask that is highest (closest to the root) on the stack.
21: **while** $top(K_t^j) \neq M'$ **do**
22: $pop(K_t^j)$.
23: **end while**
24: $pop(K_t)$.
25: **end if**
26: $K_{t+1}^j := k_t^j$ is the resulting execution stack.
27: **until** all stacks are empty

If a^j is a cooperative subtask it is restated as Equation 4.2, where $\widetilde{a}^1, \ldots, \widetilde{a}^{j-1}, \widetilde{a}^{j+1}, \ldots, \widetilde{a}^n$ denotes the actions made by the other agents in subtask a^j.

$$Q^{j^\pi}(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) = V^{j^\pi}(a^j, s, \widetilde{a}^1, \ldots, \widetilde{a}^{j-1}, \widetilde{a}^{j+1}, \ldots, \widetilde{a}^n) + C^{j^\pi}(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) \quad (4.2)$$

If i is a non-cooperative subtask but a^j is, it is restated as Equation 4.3, where $\widetilde{a}^1, \ldots, \widetilde{a}^{j-1}, \widetilde{a}^{j+1}, \ldots, \widetilde{a}^n$ denotes the actions made by the other agents in subtask a^j.

$$Q^{j^\pi}(i, s, a^j) \quad = \quad V^{j^\pi}(a^j, s, \widetilde{a}^1, \ldots, \widetilde{a}^{j-1}, \widetilde{a}^{j+1}, \ldots, \widetilde{a}^n) \; + \; C^{j^\pi}(i, s, a) \quad (4.3)$$

4.6 A Hierarchical Multi-agent Reinforcement Learning Algorithm

The algorithm used in the multi-agent setting is an extended version of Algorithm 3. The extension handles the case where the subtask is a cooperative subtask, where the updating takes the other agents actions into account. There are however two cases that are not taken into account. First there is the case where the current task is not a cooperative subtask but the childtask is. The second is the case where both the task and subtask are cooperative. If only the childtask is a cooperative subtask the best action should be found as in Equation 4.4 and the completion function should be updated as in Equation 4.5.

$$a^* = \underset{a' \in A_i}{\arg\max} \left[C_t^{j^\pi}(i, s', a) + V_t^{j^\pi}(a', s', \widetilde{a}^1, \ldots, \widetilde{a}^n) \right] \quad (4.4)$$

$$C_{t+1}^{j^\pi}(i, s, a^j) := (1 - \alpha_t^j(i)) C_t^{j^\pi}(i, s, a^j) + \alpha_t^j(i) \gamma^N \left[C_t^{j^\pi}(i, s', a^*) + V_t^{j^\pi}(a^*, s', \widetilde{a}^1, \ldots, \widetilde{a}^n) \right] \quad (4.5)$$

If it is both the task and childtask that are cooperative the best action should be found as in Equation 4.6 and the completion function should be updated as in Equation 4.7 where $\widehat{a}^1, \ldots, \widehat{a}^{j-1}, \widehat{a}^{j+1}, \ldots, \widehat{a}^n$ are the actions made by the other agents in state s'.

$$a^* = \underset{a' \in A_i}{\arg\max} \left[C_t^{j^\pi}(i, s', a, a^1 \ldots a^n) + V_t^{j^\pi}(a', s', \widetilde{a}^1, \ldots, \widetilde{a}^n) \right] \quad (4.6)$$

$$C_{t+1}^{j^\pi}(i, s, a^j) := (1 - \alpha_t^j(i)) C_t^{j^\pi}(i, s, a^j, a^1 \ldots a^n) + \alpha_t^j(i) \gamma^N \left[C_t^{j^\pi}(i, s', a^*, \widehat{a}^1, \ldots, \widehat{a}^{j-1}, \widehat{a}^{j+1}, \ldots, \widehat{a}^n) + V_t^{j^\pi}(a^*, s', \widetilde{a}^1, \ldots, \widetilde{a}^n) \right] \quad (4.7)$$

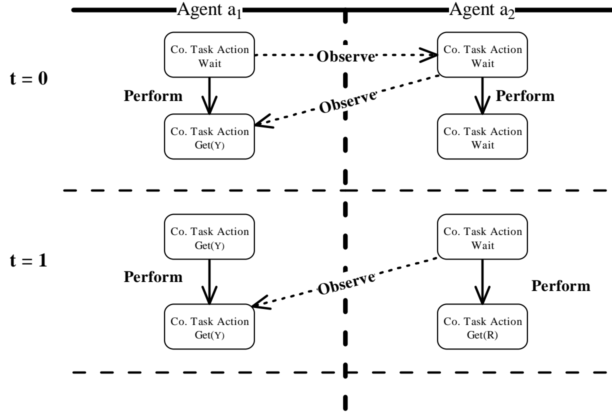


Figure 4.2: Execution of the hierarchical multi-agent algorithm.

In Algorithm 7 a stack *seq* is maintained. This stack contains the sequence of states visited and the actions of the shared subtasks being performed by the other agents. If the algorithm is called with a primitive action the state s is pushed onto the stack along with the actions of the other agents. The action is performed and the value function is updated. Then the stack is returned. If the algorithm is called with a cooperative composite action an action a is chosen using the current exploration policy. The algorithm then calls itself recursively storing all the states visited and the actions of the other agents while executing the chosen action a. It then updates the completion function. If the resulting state is a terminating state the stack is returned. If the algorithm is called with a composite action that is not cooperative, the only difference is that the actions from the other agents are not used.

To illustrate how the algorithm works consider Figure 4.2. Here we have two agents a_1 and a_2, where a_1 performs a primitive action in time t before a_2. When starting out both agents have their cooperative subtask set to execute a $Wait$ action. The environment seen from a_1 looks as follows:

$$[(2,2), _, _, \{Y_R, _, _, _\}], Wait$$

This configuration means that the agent is located in grid location $(2,2)$. The two following variables are blank, The first indicating that the agent has not yet selected a passenger to retrieve and therefore the second is also blank because there is no passenger destination. Following these variables are the 4 locations variables, Y, R, G, and B. Only the first has a non blank variable. The first location represents the Y location. This variable has the value Y_R meaning the passenger at location Y would like to go to location R. $Wait$ indicates that the other agent is performing a $Wait$ action. The agent observes the action of agent a_2 to make sure that this agent is not performing a $Get(Y)$. Since this is not the case (agent a_2 is performing the *wait* action), a_1 can choose to retrieve the passenger at Y i.e. performing a $Get(Y)$ action. Doing this causes the state as seen from a_1 to change:

$$[(2,2), Y, R, \{Y_R, _, _, _\}], Wait$$

Algorithm 7 The Cooperative HRL Algorithm

1: **Function** Cooperative-HRL(Agent a, Task i at the lth level of the hierarchy, State s)
2: let Seq = () be the sequence of (state-visited, actions in $\bigcup_{k=1}^{L} U_k$ being performed by other agents) while executing i where L is the number of levels in the hierarchy
3: **if** i is a primitive action **then**
4: execute action i, receive reward r and observe result state s'
5: $V_{t+1}^{j^\pi}(i,s) := (1-\alpha_t^j(i)) \cdot V_t^{j^\pi}(i,s) + \alpha_t^j(i) \cdot r_t$
6: push (state s, actions in $\{ U_l | l$ is a cooperation level $\}$ being performed by the other agents) onto the front of Seq
7: **else**
8: **while** i has not terminated **do**
9: **if** i is a cooperative subtask **then**
10: Choose action a^j according to the current exploration policy $\pi_i^j(s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n)$
11: let $ChildSeq$ = Cooperative-HRL(j, a^j, s), where $ChildSeq$ is the sequence of (state-visited, actions in $\bigcup_{k=1}^{L} U_k$ being performed by the other agents) while executing action a^j
12: observe result state s' and $\widehat{a}^1, \ldots, \widehat{a}^{j-1}, \widehat{a}^{j+1}, \ldots, \widehat{a}^n$ actions in U_l being performed by the other agents
13: let $a^* = \arg\max_{a' \in A_i} \left[C_t^{j^\pi}(i, s', \widehat{a}^1, \ldots, \widehat{a}^{j-1}, \widehat{a}^{j+1}, \ldots, \widehat{a}^n, a') + V_t^{j^\pi}(a', s') \right]$
14: let $N = 0$
15: **for** each $(s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n)$ in $ChildSeq$ from the beginning **do**
16: $N = N + 1$
17: $C_{t+1}^{j^\pi}(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) :=$ $(1-\alpha_t^j(i)) C_t^{j^\pi}(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) +$ $\alpha_t^j(i)\gamma^N \left[C_t^{j^\pi}(i, s', \widehat{a}^1, \ldots \widehat{a}^{j-1}, \widehat{a}^{j+1}, \ldots, \widehat{a}^n, a') + V_t^{j^\pi}(a^*, s') \right]$
18: **end for**
19: **else**
20: choose action a^j according to the current exploration policy $\pi_i^j(s)$
21: let $ChildSeq$ = Cooperative-HRL(j, a^j, s), where $ChildSeq$ is the sequence of (state-visited, actions in $\bigcup_{k=1}^{L} U_k$ being performed by the other agents) while executing action a^j
22: observe result state s'
23: let $a^* = \arg\max_{a' \in A_i} \left[C_t^{j^\pi}(i, s', a,) + V_t^{j^\pi}(a', s') \right]$
24: let $N = 0$
25: **for** each state s in $ChildSeq$ from the beginning **do**
26: $N = N + 1$
27: $C_{t+1}^{j^\pi}(i, s, a^j) := (1 - \alpha_t^j(i)) C_t^{j^\pi}(i, s, a^j) +$ $\alpha_t^j(i)\gamma^N \left[C_t^{j^\pi}(i, s', a^*) + V_t^{j^\pi}(a^*, s') \right]$
28: **end for**
29: **end if**
30: append $ChildSeq$ onto the front of Seq
31: $s = s'$
32: **end while**
33: **end if**
34: **return** Seq

Once agent a_1 performs a primitive action the initiative is given to agent a_2. The view that a_2 has of the environment looks similar to that of a_1:

$$[(1,4), _, _, \{Y_R, _, _, _\}], Get(Y)$$

a_2 discovers that there is a passenger at location Y, but after observing the action of a_1 it decides to not attempt to retrieve the passenger. Instead a $Wait$ action is executed.

Now that both agents have performed a primitive action the time step is incremented by one, and another round of actions can begin.

After a given number of time steps the environment will add more passengers for the taxies to service. In this case the environment adds a passenger at location R (destination G). For agent a_1 this information is irrelevant, it just continues to navigate for the passenger at location Y. For a_2 this means that instead of continuing to perform $Wait$ actions it can perform a $Get(R)$. The environment seen from a_2 at time step $t = 1$ when a_1 has performed its action looks as follows:

$$[(1,4), _, _, \{Y_R, R_G, _, _\}], Get(Y)$$

The above example demonstrates how the time progresses in the environment, but also how the agents are supposed to coordinate with each other to avoid competing for the same passenger.

4.7 The Multi-agent Taxi Problem

To test the MHRL algorithm the Taxi problem is extended with a second taxi. Let us assume that at every 10th time step a passenger will arrive at one of the four locations (no passenger can be at that location already). This passenger will want to be taken to one of the other three locations. Once the passenger has been delivered it disappears and the taxi is ready to service another passenger. This process will continue for a set number of time steps. The goal is to maximise the throughput of passengers over a given period.

The challenge is to coordinate the taxis so that they do not compete for the same passenger, but instead cooperate so that as many passengers as possible reach their destinations within the allocated time steps, thereby increasing the combined overall reward for the two taxies. To simplify the problem two taxies can be located at the same grid location (think of a grid location as a road, with two lanes) such that collision detection is not needed.

In the single Taxi problem each primitive action has a penalty of -1, and performing a *Pickup* or *Putdown* in an illegal state is penalised with -10. In the MHRL version of the Taxi problem the primitive navigation actions have a penalty of -2, likewise with the *Pickup* and *Putdown* actions. The reward in illegal states does not change. These changes was made to accommodate a new primitive action, the $Wait$ action. In the extended Taxi problem a $Wait$ action is penalised with -1. The reward for completing the *Root* task is increased to 40. There are two main reasons for the change in rewards.

- If $Wait$ has no penalty, then the agent will most likely learn that the best action in any given situation will be a $Wait$. This will happen because when starting out to learn the cumulative number of penalties received will be higher than the positive reward given for completing the task, thus yielding an overall penalty. This will mean that the most favourable action in *Root* will be a $Wait$ action. However because the exploration policy is an ordered GLIE policy all paths are guaranteed to be explored, so eventually an optimal policy will be found, but it will take much more time than by setting the value of $Wait$ to something else.

- If $Wait$ has the same penalty as the other actions, then there will be no point in having this action, because it will be more favourable to just have all taxies navigate to the passenger, and let the one who gets there first pick it up. This will mean that learning inter-agent coordination will be impossible.

In this problem the $\tau_{continue}$ termination strategy from Section 4.2 is employed, meaning that the termination of a subtask in one agent does not affect the execution of other agents.

Figure 4.3 shows the decomposition of the multi-agent taxi problem. Notice that as this is the hierarchy used by all agents in the problem, the agents are homogeneous. The problem of incorporating heterogeneous agents into the multi-agent algorithm is the topic of Chapter 5.

There are only two differences between the multi-agent taxi graph and the single-agent taxi graph. First, the *Get* subtask has the parameter t. This parameter is necessary because the children of any shared subtasks are known to other agents, and other agents need to know what the agent is doing. Simply knowing that the agent is doing a *Get* action is not enough, they need to know where the agent is getting the passenger from, so that it can learn not to perform the same action. The second difference is the $Wait$ action. This is, as mentioned, a primitive action that does nothing except use time steps. This action is useful when there are no passengers to collect. The agent can then choose to reduce the penalties given by waiting instead of navigating to an empty location, or a location already serviced by another taxi.

4.7.1 An Example

Consider the scenario shown in Figure 4.4. Agent 1 is located at position (0,3) and agent 2 is at (3,2) and both agents have learnt an optimal policy. Now a passenger arrives at Y. Agent 1 is first to act and checks its policy to see what it must do. Because agent 2 has not yet performed its first action, agent 1 sees it as performing a $Wait$ action. The policy lookup $\pi_{root}(s, Wait)$ results in a $Get(Y)$ action, because agent 1 sees that there is a passenger at location Y and it also sees that agent 2 is not trying to get the passenger.

The projected value function for agent 1 in the subtask *Root* is as follows:

$$V(Root, s, \{Wait\}) = V(Get(Y), s) + C(Root, s, \{Wait\}, Get(Y))$$

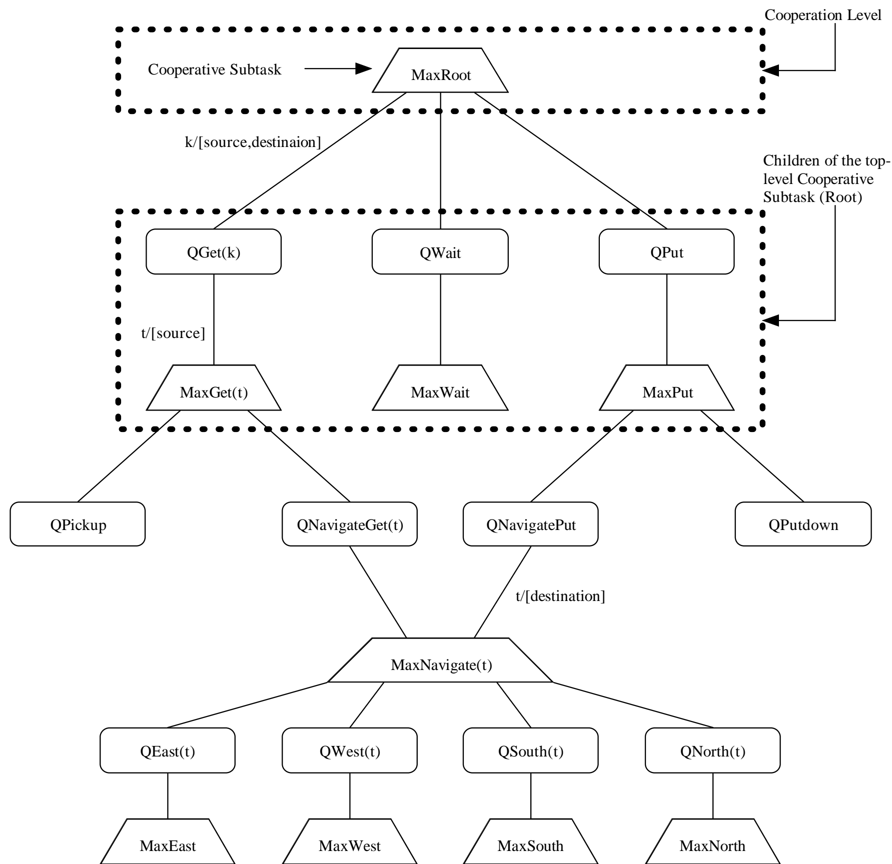


Figure 4.3: Multi-agent Taxi problem decomposition

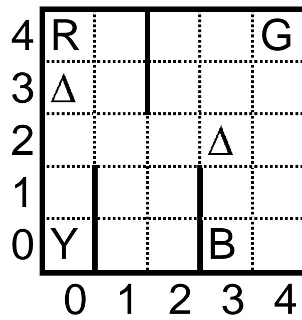


Figure 4.4: The multi-agent Taxi problem.

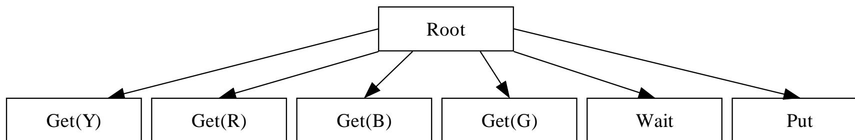


Figure 4.5: The *Root* task.

Because *Root* is a shared task the actions of shared tasks of all other agents must be considered in the value and completion function of *Root*. Because $Get(Y)$ is not a shared task, the value function of this task is like the one described in Chapter 3.

Once agent 1 has performed a primitive action, control is given to agent 2. Agent 2 now checks at what locations there are passengers. Because the environment is still at time step 1, and new passengers only arrive every 10th step, the only passenger available is the passenger at location Y. Agent 2 consults its policy to find the correct action $\pi_{root}(s, Get(Y))$. The policy will return $Wait$ because it has learnt that if another taxi is performing a $Get(Y)$ action, and no other passengers are listed as waiting for a taxi, the best overall action will be to wait, and not try to beat the other taxi's to the passenger.

The problem with this approach is that sometimes the taxi standing closest to the passenger has to wait because it is not first to act. This problem could be fixed by letting each agent know the whereabouts of other agents. However, this would result in a much larger state space.

In our current setup the order in which agents execute actions remains the same through the entire episode. Once an agent has performed one primitive action the agent stops executing, and the initiative is given to the next agent in line. Using this approach each agent gets one primitive action per time step.

4.7.2 State Abstraction

When expanding to MHRL the number of parameters needed to represent each task remains the same for all non-cooperative tasks. What is changed are the parameters needed. Because there can be many passengers in the environment a source and a destination parameter attached to the environment is no longer enough. Instead, each taxi has a passenger variable and a destination variable. If there is no passenger in the taxi, and the taxi has not been assigned a passenger to pickup, the two variables are blank. If the variables are not blank the *Get* task is called with the passenger variable as parameter. Using this setup the number of variables it takes to represent each taxi remains the same, the only thing that is changed is the way the variables are found.

A difference between HRL and MHRL when it comes to state abstraction is when considering the state/action pairs possible. In HRL there were only two possible actions from *Root* in the Taxi problem where there in MHRL now are six. Furthermore there are the shared actions of the other agents to consider. The 6 possible actions are shown in Figure 4.5.

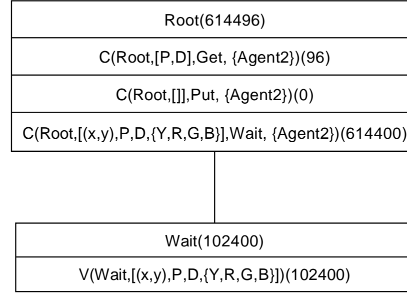


Figure 4.6: The added branch to the task graph of Figure 3.10.

In HRL there were only 16 different values (4 source and 4 destination) for the completion cost of *Root* once a *Get* had been completed, whereas in MHRL there are 4 source, 4 destinations, and the 6 different actions that the other agent can perform, resulting in a total of 96 values for $QGet(t)$.

The major difference when refactoring the state abstraction scheme to work with MHRL is in the state abstraction necessary in the $Wait$ task. The $Wait$ action can be performed at two distinct times in the execution of the problem, namely before *Get*, where no passenger has been assigned to the taxi, and before *Put*, where a passenger is located inside the taxi. However, it does not make any sense to perform a $Wait$ action if there is a passenger in the taxi, therefore performing a $Wait$ before *Put* is not possible.

If no state abstraction were applied the following variables would be needed.

- The location of the taxi (25)
- The source and destination for the taxi (4×4)
- If there is any passenger at location Y, R, G, B, and what is their destination (4^4)

In total this yields 102400 states. Couple this with the actions of the other taxies a total of 614400 values need to be represented in the case of one other taxi. Figure 4.6 shows the added branch to the task graph from Figure 3.10.

In total, adding $Wait$ adds 716800 values to the problem. Naturally this huge number compared to what it takes to represent the rest of the graph is unacceptable. Therefore state abstraction is even more important here than it is in single-agent HRL.

By applying the leaf irrelevance rule from Section 3.9.1 we can reduce the number of values in the value function for *wait* from 102400 to only 1. This can be done because we know that no matter the configuration of the state, the only penalty that a $Wait$ action can yield is -1, thus all variables can be removed.

By applying the subtask irrelevance rule from Section 3.9.2 we can remove the source and destination from the taxi state. Subtask irrelevance states, that if a variable is not needed by a child task, or by the task itself, then the variable

can be removed. This can be done because we know that $Wait$ can only be performed before Get, meaning the source and destination variables will be blank. Now that these two variables have been removed, the completion function looks as follows:

$$C(Root, [(x, y), \{Y, R, G, B\}], \{Agent2\ action\}, Wait)$$

This leaves us with a total of $25 \cdot 4^4 \cdot 6$ values. Applying the last three rules from Section 3.9.3 to Section 3.9.5 would change nothing. So the best that the safe abstraction rules can offer is a reduction in the wait branch from 716800 values to only 38400

Further Abstractions

As stated above no more abstraction using the 5 safe rules from Section 3.9 can be applied to the $Wait$ task without loosing precision. But consider what will happen if we were willing to sacrifice some precision for a sizeable reduction in states. The idea is to, instead of having the destination of the passenger attached to each of the four locations remove the destination so all that the location tells the agent is whether or not there is a passenger there. This abstraction would mean a reduction from $25 \cdot 4^4 \cdot 6$ values to only $25 \cdot 4^2 \cdot 6$, 38400 values to only 2400 values. The consequence of doing this is that when choosing which passenger to navigate to, the destination of that passenger is not taken into account, meaning that it might have been more profitable to navigate to a passenger farther away, because this passenger did not need to travel so far, meaning the total reward would be higher. But by removing the destination from the state this consideration is not possible. Instead we can expect the penalty for navigating to a specific destination will become an average of navigating to each of the other 3 locations. That is if the passenger is at Y, we can expect that the completion cost for root once a $Get(Y)$ has been performed will be an average over the distance from Y to the other locations.

In the implementation of the multi-agent taxi task this abstraction is applied. All in all the number of values to be represented is shown in Figure 4.7 and amounts to 3113 which is the total number of different value functions and completion functions in the task graph.

Tabular Q Learning

If the same problem were to be represented in a flat table implementation the following would be required:

- For each of the four locations there are four settings, namely, no passenger, and the three other locations.
- For each taxi, a total of 225 settings. This number is achieved by looking at whether a passenger is in the taxi or not. If a passenger is in the taxi, only the location of the taxi and the destination of the passenger is relevant. Likewise if there is no passenger in the taxi, then the relevant information will be the location of the taxi, and the location the passenger, if any, the taxi has singled out.

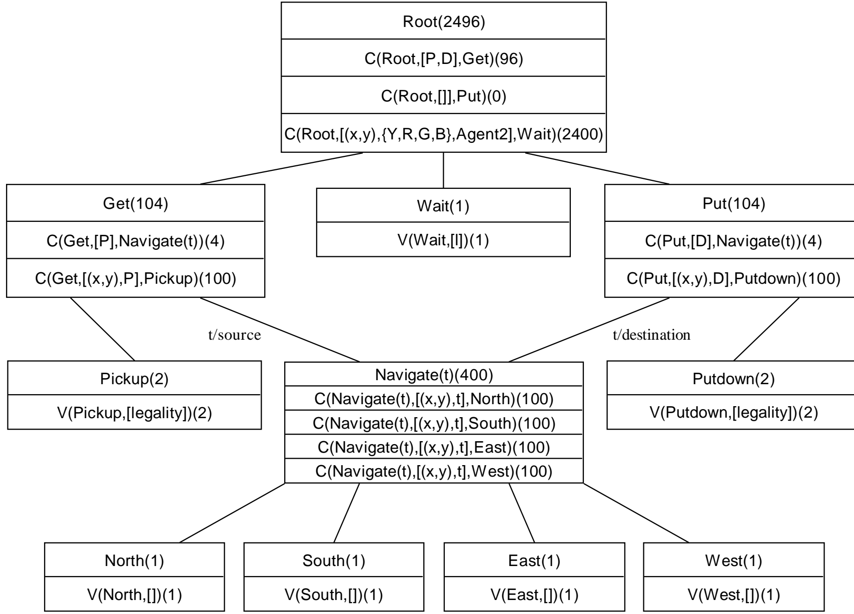


Figure 4.7: The complete task graph with state abstraction applied.

- The number of taxis.

This results in the following formula for calculating the number of states in the tabular multi-agent Taxi problem:

$$states = loc^{loc} * (board * loc + (board * (loc + 1)))^{agents}$$

In the case where the board area is 25 grids, the number of locations is 4 and the number of agents is 2, the total number of states will be just below 13 millions, all in all resulting in a total number of values to be represented in the neighbourhood of 90 millions.

4.8 Testing of the MHRL Algorithm

In this section the MHRL algorithm is tested. To this end the Taxi problem presented previously will be used.

4.8.1 Environmental Setup and Passenger Adding Rules

When initialising a learning episode, two taxies starts at a random location. Each taxi has no reference to any passenger. Furthermore the environment is initialised with one passenger at one of the 4 locations. The destination of this passenger is chosen at random. At certain intervals new passengers are added

to the board. The interval at which new passengers are added is set in the individual tests.

When testing the learnt policy the two taxies both start in location $(2, 2)$. A passenger will be located at location Y wanting to go to G. When adding a new passenger to the board the following update rules are applied.

- The locations are prioritised with Y as the most important, followed by R then G and finally B.
- When adding a passenger this passenger is added to the highest prioritised location that does not have a passenger associated with it. Meaning that if for example Y does not have any passenger, then a passenger is added to this location. If Y already has a passenger we look to the next location.
- The destination of the passenger is as follows:
  - Source: Y: destination R.
  - Source: R: destination G.
  - Source: G: destination B.
  - Source: B: destination Y.

Using this scheme we can calculate how many time steps it will take the taxi to deliver a passenger, assuming a passenger has already been picked up.

- Pickup at Y: 5 time steps to deliver.
- Pickup at R: 9 time steps to deliver.
- Pickup at G: 6 time steps to deliver.
- Pickup at B: 8 time steps to deliver.

With this knowledge we can calculate the optimal value of a test, by simply tracing the optimal steps.

An episode will consist of 250 time steps, and a trial will consist of 50 episodes. An episode terminates when time runs out. Whenever this happens the agents stops acting, and any task that they were doing is cut short, and no more values will be updated once time runs out.

The value of the learnt policy is measured by the total rewards and penalties given to all agents during the course of the episode.

4.8.2 Exploration Policy Used During Learning

While executing the learning algorithm, an agent must repeatedly choose the next action to execute. This action can be chosen at random, but that will most likely lead to slow learning, since an agent might search non profitable paths again and again. That is, the agent will keep exploring the environment and not exploit what is already learnt. On the other hand, if an agent continuously

chooses the action with the highest payoff, it risks converging to a sub-optimal policy—simply because there might exist better undiscovered paths (with a higher payoff) that are never explored.

To solve this dilemma, a prior probability distribution over the possible actions from the currently known values is calculated. Actions with a high payoff get a higher probability of being chosen than actions with a low payoff. However, all actions get a probability greater than zero. The probability distribution is defined as follows:

$$P(i, s, a) = \frac{k^{-Q(i,s,a)}}{\sum_j k^{-Q(i,s,a_j)}} \tag{4.8}$$

where $P(a_i|s)$ is the probability of selecting action a_i given state s. The constant $k > 0$ determines how strongly the selection favours actions with high $\hat{Q}$ values. Lower values of k encourage exploiting while higher values increase the probability of exploring. Before calculating the probability distribution, the range of Q values should be normalized to a number between zero and one. High Q values might otherwise result in very large numbers. Figure 4.8 shows the effect of setting the constant to different values. In the experiments the constant k is set to 0.5.

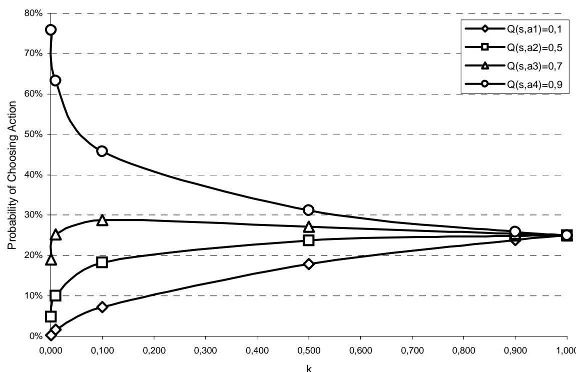


Figure 4.8: The effects of constants.

Using this technique ensures that all actions have a possibility of being selected while learning, thus increasing the chance that all states will be visited. When testing the learnt policy a greedy policy was used. If two values had the same value the tie was broken in favour of the first ordered action, just as is the case with the ordered GLIE.

4.8.3 The Test

In this experiment a passenger will be added every 5 time step. By adding a passenger every 5 time step we are ensured that all taxies are kept busy at all times, except for the first 5 time steps, where there is only one passenger on

the board. The board that the taxies are navigating on looks like the boards presented earlier, except that there are no walls except for the those surrounding the board. By removing the walls we open up for the possibility that the agents might be able to find some advanced policy that would otherwise not be possible. Below we describe a reasonable scenario, and in fact this scenario would represent the optimal policy had there been walls between the locations.

Optimal behaviour in this case would be that in the beginning, one of the agents will head for the passenger already on the board. This passenger is at location Y. Starting in location $(2, 2)$, it takes 4 navigation actions for a taxi to reach Y and one action to pickup the passenger, for a total of 5 actions. This takes 5 time steps in total. All the while the other agent has been performing wait actions. When 5 time steps have passed another passenger is added to the board. because Y is the location with the highest priority the passenger is added to this location. Now the first agent will head to R to deliver the passenger while the second agent will head to Y to retrieve a passenger. After 5 time steps, one taxi will have just delivered a passenger at R, and one will just have picked up a passenger at Y. Following this every 5 time step one agent will be picking up a passenger while another will be putting down a passenger. This continues until all the time steps have been spent.

Using this reasonable behaviour means that at any point except the first 5 time steps a passenger will be delivered, and a passenger will be picked up. Counting the reward given for 5 such steps yields the following:

- 8 primitive navigation actions each with a penalty of -2.
- 1 *Pickup* action, and one *Putdown* action, each with a penalty of -2.
- A reward of 40 for delivering a passenger.

In total such a 5 block has a total reward of 20. For the first 5 time steps however, there can be no pickup. therefore, the total penalty is $-15(-5$ for the 5 wait actions from one agent, and -10 for the other agent (4 navigate and the pickup)). In total there are 49 of the blocks that yields 20 in reward. In total this will result in a reward of 965.

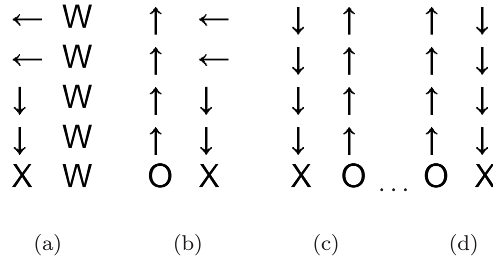


Figure 4.9: The steps taken using the optimal policy for the 5 steps scenario.

Figure 4.9 shows the primitive actions taken using an optimal policy when there are walls between locations as presented in previous chapters. In this figure an

X represents a pickup, an **O** is a putdown, and a **W** is a wait action. An arrow represents a navigation action.

Sequence of Rewards

When looking at the rewards returned from a trial a typical sequence might be.

$$\ldots \quad 900 \quad 935 \quad -990 \quad 965 \quad \ldots$$

The −990 in the middle of positive values might seem puzzling at first. The explanation for this is very simple. During execution of a test the action believed to be the best is the action that has the highest Q value. It will often be the case that the optimal actions value is very close to the value of an action that will result in a non optimal sequence, or even a sequence with looping behaviour. A small change in values could change the outcome of the policy query.

Versus Two HRL Agents

To have something to compare the algorithm with, an implementation with two agents using HRL was made. Each implementation was run for 100 trials, each trial containing 50 learning episodes interleaved with 50 testing episodes [1]. In Figure 4.10 the average reward of each episode is shown. The curve with a *Cooperative* predicate attached represents the average reward of the two MHRL algorithms. The *Uncooperative* curve represents the two HRL agents.

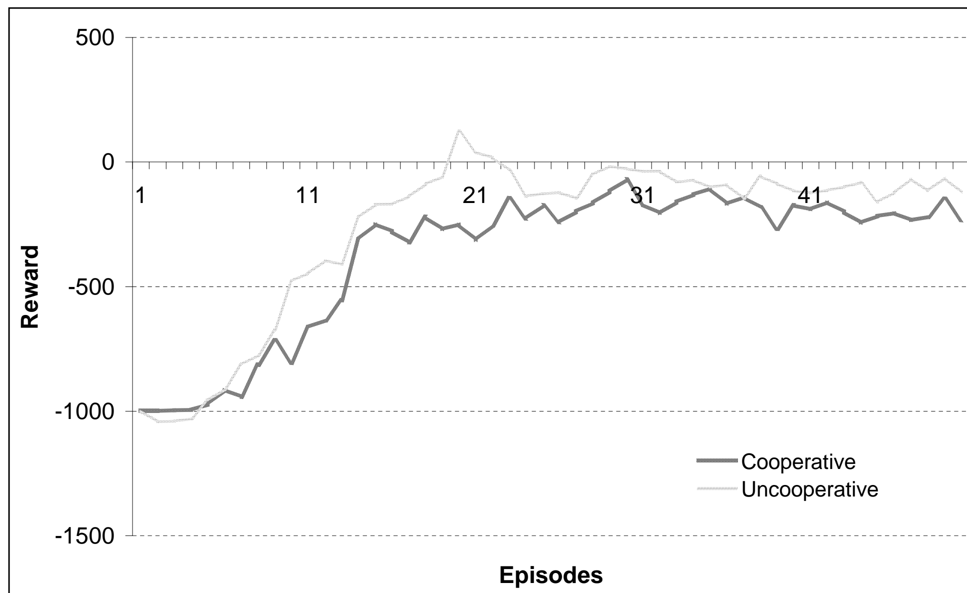


Figure 4.10: Average reward.

[1] The test data can be found on the CD.

In most cases the average reward found is negative. This is due to the problem explained above. However another disheartening fact appears. It seems that, except for the early stages of learning, the two HRL agents are getting better average rewards. A plausible explanation for this is that, because the MHRL agents have more values in their completion functions, in that they are each keeping track of the actions of the other agent, it will take longer for their values to stabilise.

However, it has been assumed that, by using the MHRL algorithm we can achieve some policies that are not possible for the two HRL agents that are not cooperating. To examine this we need to look at the maximum rewards given to each of the agents. Figure 4.11 shows the maximum rewards given in the two implementations.

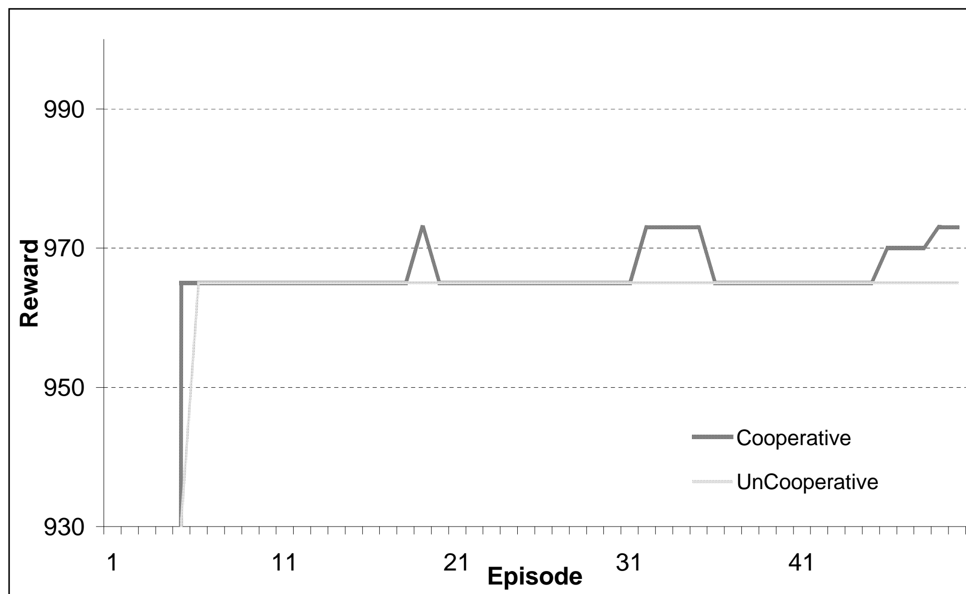


Figure 4.11: Maximum reward.

Looking at the figure, we can clearly see that the two HRL agents never receive more than 965 in reward. This reward is achieved by using the policy described in the beginning of Section 4.8.3. The MHRL implementation however does achieve some rewards that are higher. In fact the highest reward given to the MHRL implementation is 973. This reward is achieved by using a much more complex policy. A trace of the optimal route for the HRL and the MHRL implementation shown in Appendix A.

4.9 Summary

In this chapter the multi-agent hierarchical reinforcement learning algorithm introduced by Mohammad Ghavamzadeh and Sridhar Mahadevan was presented.

The algorithm expands on Algorithm 3 from Chapter 3 by splitting up the composite actions in cooperative and non-cooperative tasks. By doing this cooperative tasks were able to see the cooperative task of other agents. This had the effect that complex inter-agent strategies could be devised. During testing it was shown that using this algorithm did in fact produce policies that was better than what could be achieved using regular HRL agents.

In this chapter all the agent where homogeneous. In the next chapter the changes needed to allow heterogeneity are presented.

Chapter 5

Adding Heterogeneity to Multi-Agent Environments

The assumption that agents are homogeneous is becoming increasingly unrealistic, since agents will often be designed by different individuals, with different ideas of what is the correct way to decompose a problem and which is not. Also, as problems become bigger and more complex, agents will undoubtedly be given different goals. In this situation the communication scheme used in the previous chapter is not always usable.

In the previous chapter we introduced an algorithm for handling cooperative multi-agent environments. This algorithm assumed that:

- All agents operate with the same goal, and are able to perform the same primitive actions.
- All agents use the MHRL task decomposition, and the decomposition of all agents are the same.

In this chapter we will introduce two very different approaches to dealing with heterogeneity. In the first approach we focus on heterogeneity in agents that have the same goals and primitive actions. The heterogeneity can be found in the learning algorithms and the settings of the agents. This approach will allow agents with different learning algorithms, or distributed value functions to share knowledge. Better learning under this form of heterogeneity can be expected by having agents share sequences of particularly good state/action pairs. A sequence of state/action pairs is given an educational value. An educational value is a value that indicates how much can be learnt from a sequence. How this value is calculated is determined by the programmer. An example could be to give all the sequences random values. This approach is called *Heterogeneity in Method*, and is the topic of Section 5.1.

The second approach allows agents with different goals to work together. In large and complex problem domains we will often want to have more than one type of agent, and often we need these different agents to coordinate and cooperate.

This could e.g. be because a task needs a component from each agent delivered at the same time, or perhaps we just do not want the agents to compete for resources. To achieve this form of coordination and cooperation, a concept called interest groups is introduced. This approach is called *Heterogeneity in Goal* and is the subject of Section 5.2.

5.1 Heterogeneity in Method

So far this report has explored the area of hierarchical reinforcement learning in single and multi-agent environments. In HRL and MHRL it is possible for agents to share value functions and completion functions. This is not always possible either because of a large physical distance, where communication is costly, or for some other reason. In these cases learning will be slow because only one agent will be updating the completion function. To speed up learning when faced with this problem we introduce two different approaches that both use inter-agent guidance.

Besides being useful to the MHRL algorithm, this approach can also be applied in cases where the learning algorithm differs, e.g. agent A is using a genetic learning algorithm, while agent B is using tabular Q learning.

5.1.1 The Concept of Guidance

In very large domains, even with the abstraction and decomposition possible, it might be difficult to learn anything because the goal/terminal states might be so sparsely distributed in the state space that they will almost never be encountered using random exploration. In the Taxi problem an example of this is the navigate subtask. Here there is only one location that will terminate the subtask. The more grid locations there are, the less likely it will be that the taxi encounters this state.

This problem can be solved, or at the very least made less of a problem by applying guidance. An agent can for instance be guided by providing it with a *reasonable* policy in the beginning of the learning process. One way to create this policy is by logging the behaviour of a human expert and having the agent mimic this behaviour in its first episode. The agent can, based on the learnt policy, explore the state space, and thus be directed towards the terminating states and/or reward states.

Another strategy is to supply guidance interleaved with the normal exploration policy of the agent, meaning that guidance is provided at certain points during the execution. One benefit of the interleaving strategy is that the agent can request guidance whenever it thinks that it is needed. This guidance could be provided by a human expert, or it could be another process that has experience with the problem. The advantage of this approach is that the agent can decide when it wants guidance, so if it is in a state that it does not have a lot of experience with, it can request help. This approach allows the guidance to focus on areas of the state space which have not yet been explored.

5.1.2 Inter-Agent Guidance

The idea in inter-agent guidance is to take a sequence of state/action pairs from one agent, and give this sequence to another agent, that will then execute the state/action pairs, and update its value function accordingly.

There are several cases where doing this can be quite beneficial to the learning process:

- When a state/action pair, or a sequence of state/action pairs are not visited often, and the chance that another agent will stumble upon this is minimal. The sequence can then be propagated to other agents.
- When a particular good sequence of state/action pairs has been found, the agent can share this with other agents, and thereby accelerate learning.
- When agent A and agent B are using different learning algorithms, or even in cases where only the settings of the same algorithm are different, providing guidance to another agent might prove useful. This could be because agent A is very good at learning in one part of the state space, while agent B is not. In cases such as this, assistance will accelerate the overall learning rate.

By using inter-agent guidance the amount of data transmitted between agents is reduced because we only have to transmit data from one agent to another a few times, instead of every time a value function is updated.

A Motivating Example

Consider a scenario where we have two taxi agents. These agents are acting on a 100×100 board. Both agents are using the algorithm presented in Chapter 4. However, the two agents are not sharing value function.

Both agents are navigating around trying to find a specific location, which is a difficult task due to the size of the board. If an agent is lucky and finds a path to this target location, we would like this path to be known by the other agent as well. If we take the state/action pairs from this path and give it to the other agent, the knowledge of how to find the target location is shared.

The agent receiving the state/action pairs from the path, now only has to execute each state/action pair, observe the resulting state and the resulting reward, and update its value function.

In the following, two different approaches to sharing sequences of state/action pairs are presented.

In the first approach it is assumed that actions of other agents cannot interfere with the resulting state of a primitive action execution. If this is the case, then each agent can execute a sequence of state/action pairs independently of other agents.

The second approach does not have the same assumption as the above. In this case the execution of a state/action pair might be dependent on the other

agents behaviour. Therefore all agents needs to be executing a sequence found in the same time steps. This insures that the resulting states are correct if the environment is deterministic.

5.1.3 The Independent Agents

The first approach is called *The Independent Agents*. First an episode is generated. From this episode each agent stores the sequence of states and actions that they encountered. Each agent then calls the function in Algorithm 8, and a sub-sequence is returned. Algorithm 8 describes how this sub-sequence is selected.

Algorithm 8 The independent agent sequence selection algorithm.

```
1:  Function I-SequenceSelection(Sequence Seq)
2:  let P be the empty sequence planner
3:  let l be the length of Seq
4:  let y = 0
5:  while y < l do
6:     Let t = y + 1
7:     while  t < l do
8:        let x = the educational value of a sequence starting with element y and
          ending in element t, according to a chosen sequence evaluation function
9:        add (x, y, t) to P
10:       t = t + 1
11:    end while
12: end while
13: find a tuple (x, y_x, t_x) in P with the highest value of x
14: return  (x, the sequence of elements from Seq_{y_x} to Seq_{t_x})
```

I-SequenceSelection takes the sequence of state/action pairs from an agent as a parameter. It then assigns an educational value to each possible sub-sequence of this sequence according to some chosen heuristic. The sub-sequence with the highest educational value is then returned. If two sub-sequences ties for the highest educational value, then one is selected at random.

Once all agents have found a sub-sequence, sequence sharing occurs. Each agent passes the sub-sequence that they have found along to another agent. Now each agent can execute their new sequence according to Algorithm 9.

I-SequenceExecution iterates through the state/action pairs of the sequence Seq. For each element the state/action pair is executed, and the resulting state and reward are observed. The value function of the agent is now updated as it would have been, had the agent been performing a regular learning episode.

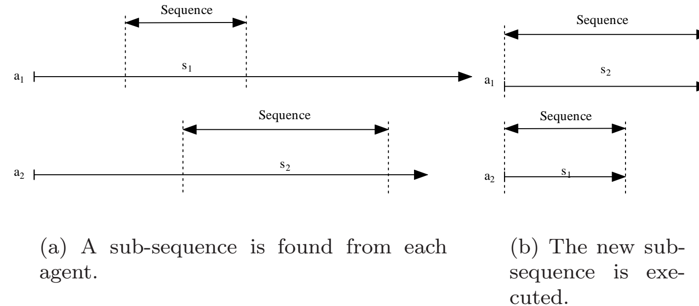
The variable l is the length of the original sequence from which the guidance sequence is taken. l needs to be available in case the learning algorithm of the agent needs it, this could be for the discount factor or for some other purpose.

Once all elements in the sequence have been executed the function returns, and the agent is ready for another learning episode, or exploration episode as the case may be.

Algorithm 9 The independent agent sequence execution algorithm.

1: **Function** I-SequenceExecution(Sequence Seq)
2: let l be the length of the original sequence, made available in the case that the update value function of the algorithm needs them
3: let t be the index in the original sequence
4: let k be the length of Seq
5: Seq is the sequence of elements Seq_i, where $0 \leq i < k$
6: $j = 0$
7: **while** j<k **do**
8: execute the state/action pair found in Seq_j, Observe reward r
9: update the value function according to the learning function of the agent.
10: $j = j + 1$
11: **end while**

In Figure 5.1 the basic idea behind the independent agents approach is shown. Figure 5.1(a) shows the sequences of states and actions performed by each agent. From each of these sequences a sub-sequence is selected to be an educational sequence. The two sub-sequences selected are denoted s_1 and s_2. The agents then execute the new sub-sequence. Figure 5.1(b) shows this. When an agent reaches the end of the sequence, execution is terminated.



(a) A sub-sequence is found from each agent.

(b) The new sub-sequence is executed.

Figure 5.1: Guidance for the independent agent guidance scheme.

By using this approach each agent chooses a sub-sequence and passes it along to another agent. The advantage of this approach is that the sub-sequence chosen by each agent is likely the most educational sub-sequence. Furthermore, because the sequence chosen by each agent does not have to be from the same point in time, an agent that has superior learning abilities at the beginning of an episode can share its knowledge with an agent that is superior at the end of an episode, and vice versa. A further advantage of this approach is that because an agent is performing the received sub-sequence independent of the execution of other agents, it has the option to decline the sub-sequence if it is deemed not educational enough.

5.1.4 The Dependent Agents

The second approach is called *The Dependent Agents*. First an episode is performed. From this episode each agent stores the sequence of states and actions that they encountered. The function D-SequenceSelection in Algorithm 10 takes all of these sequences as parameter and returns the sub-sequences that has been chosen to be shared. These sub-sequences all start at the same time step, and end at the same time step.

Algorithm 10 The dependent agent sequence selection algorithm.

1: **Function** D-SequenceSelection(Sequence List $SeqList$)
2: $SeqList$ is a list of sequences where $SeqList_i \in SeqList$ is the sequence of state/action pairs found for the ith agent, $0 \leq i < l$
3: let $(x, l, t, Seq) = \max_x(\textit{I-SequenceSelection}(SeqList_i))$
4: let $SeqExec$ = a list of empty sequences
5: let k = the number of elements in Seq
6: **for all** $SeqList_i$ such that $1 \leq i \leq n$ **do**
7: add to $SeqExec_i$ the sequence of state/action pairs from $SeqList_i$ that are located from index $(t - k)$ to index t
8: **end for**
9: **return** $SeqExec$

Finding the sequences that are to be used, is done by finding the most educational sub-sequence selected among the best sub-sequences from each agent. Line 3 of Algorithm 10 shows this.

Once this sequence is found, all the sequences occurring at the same interval are selected. These sub-sequences are returned.

Before calling the function in Algorithm 11 the returned sub-sequences are distributed among the agents, such that each agent receives a sub-sequence that it did not generate.

Now each agent can execute their new sequence according to Algorithm 11. The order of the execution of sub-sequences should be the same as when they where created. Meaning that the sequence that used to belong to the first executing agent should continue to belong to the first executing agent, of course this agent should not be the same agent as in the original execution.

The function D-SequenceExecution is called with a list of sequences and a list of agents participating. The function then iterates through the time steps of the sequences. For each time step each agent executes the state/action pair attached to their sequence at that time step. They observe the resulting state, and the resulting reward, and perform the updates that they would have done had it been a regular learning episode.

Once the sequences have been iterated through, it is possible to continue normal execution. This means that all agents can continue execution from the resulting state of the last state/action pair execution in their sub-sequence.

By using this form of guidance we get the situation shown in Figure 5.2, where a sequence of state/action pairs from time t to time $t + k$ is taken from each

Algorithm 11 The dependent agent sequence execution algorithm

```
1: Function D-SequenceExecution(SequenceList SeqList, agentList α)
2: let (x_i, l_i, t_i, Seq_i) ∈ SeqList be the sequence list along with the stored
   variables of agent i
3: let j = 0
4: let n = the number of elements in Seq_0
5: while j < n do
6:    for all active agents i do
7:       let (s_i, a_i) be the stored state/action pair of element j from sequence
         Seq_i
8:       execute s_i/a_i, Observe reward r
9:       update the appropriate values according to the learning function of the
         agent.
10:   end for
11:   j = j + 1
12: end while
```

agent and given to another agent. The agents then start executing in the first state/action pair of the sequence. When the last state/action pair has been executed the episode continue as it would if no guidance was applied.

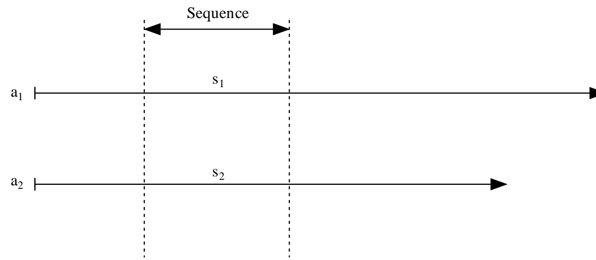
In the dependent approach the only agent that should be able to reject a sequence is the agent that receives the sequence with the highest educational value. If this is the case, then all agents should reject the sequence, and execution of the guidance algorithm should terminate.

Using this approach agents can share knowledge even when the interleaving of other agents have influence on the outcome of an action. When changing sub-sequences in environments where the agents can affect each other the sub-sequences have to represent the same time interval. If they do not, an agent can change the environment in such a way that it gets inconsistent with another agents sub-sequence. An example of this is if two agents, in the same episode, picks up a passenger at the same location but at different time steps, the sub-sequences could be selected such that both agents will be trying to pick up the passenger at the same time step and thus render the sub-sequences inconsistent.

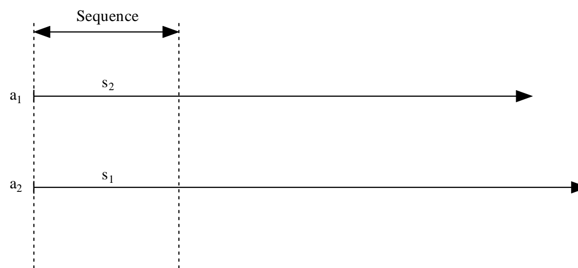
If we want to use this form of guidance and have the agents continue with execution once the shared sub-sequence has been executed, then this approach is the only one that will work. The independent agents approach cannot be used because it does not insure that the sub-sequences end at the same time step.

5.1.5 Expectations

We expect that learning will be faster when using inter-agent guidance, than what would be the case if no guidance were applied and the agents did not share value functions. By using the approaches described the agents will help each other explore the environment. When one agent finds something rewarding, it will be spread to the other agents as a result of the inter-agent guidance. However, we expect that learning will be slower than what would be the case when agents share completion and value function. If these functions were shared,



(a) The sequence from the first episode is saved and used as guidance by another agent in the next run.



(b) Each agent starts with a sequence from another agent from the previous episode. After this section has been traversed the agent returns to its normal exploration policy.

Figure 5.2: Guidance by the other agents. This process is repeated until the values have converged.

then all agents would be updating the same functions instead only one agent. So if one agent does something very rewarding, all agents would gain this knowledge immediately, thus rendering the need for experience sharing through guidance useless.

5.2 Heterogeneity in Goal

In this section we introduce a communication scheme aimed at solving the difficulties introduced when using agents that have different goals and different task structures, but are using the MaxQ value function decomposition model. This approach is an expansion to MHRL. Therefore, unless otherwise stated things are as they were in Chapter 4.

To illustrate the need for an algorithm that can handle agents with different behaviour the taxi problem will be expanded with a third agent. This agent, which we will call the *Janitor* agent, will unlike the two taxi agents not service the passengers, but have an entirely different goal. This problem is introduced in Section 5.2.1. In Section 5.2.2 we will introduce the idea behind *Interest Groups*, and present the learning algorithm in Section 5.2.4.

5.2.1 Expanding the Taxi Problem

In the expanded Taxi problem the four locations, Y, R, G and B will have a dirtiness level attached to them. As time passes, the locations become more and more dirty. At some point the status of a location will change from clean to dirty. If a location is classified as dirty, then no passengers can be picked up from that location. The goal of the janitor is to clean up the different locations, thereby changing the classification of a location from dirty to clean.

The challenge in the expanded taxi problem is to, not only coordinate between the taxies, but also coordinate between the taxies and the janitor, and if more than one janitor exists, between the janitors as well.

For the janitor the challenge is to learn that when a taxi is navigating to a location, then navigating to that same location might cause a conflict between the two, and should thus be avoided. If, however, a taxi is performing a $Pickup(t)$ action at a location, the janitor should learn that the taxi will soon be gone from this location, and navigating to that location will have a low risk of causing a conflict. Furthermore, the janitor should learn to only navigate to locations that are classified as dirty.

To reinforce this behaviour we add a penalty to the janitor agent if it attempts to clean a location on which a taxi is performing a pickup. Furthermore we increase the time it takes for a taxi to perform a $Pickup(t)$ from 1 time step to 4 time steps to make a conflict more possible.

The taxi on the other hand must learn that when a janitor is navigating to a location, then it should not try to navigating to that same location. To reinforce this behaviour a clean action will take 4 time steps. Furthermore when a clean action is being performed then no passenger can be picked up from that location.

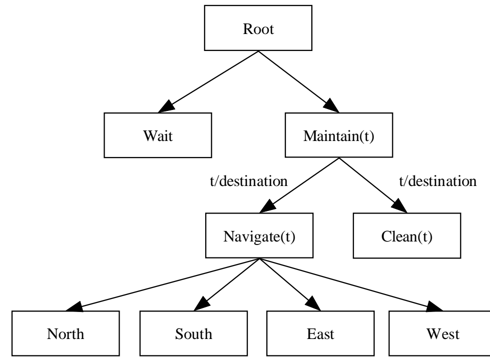


Figure 5.3: The janitor agent.

To further reinforce the above mentioned behaviour, a penalty is given to the janitor if it attempts to clean a location where a pickup is being performed.

Performing a clean action in a legal state has a reward of 20. A legal state is where the janitor is located in one of the four passenger locations, and that location is classified as dirty. Furthermore, no taxi can be performing a pickup in that location. Performing a clean action in an illegal state has a penalty of -20. Each of the navigating actions of the janitor has a penalty of -2 and the *wait* action has a penalty of -1.

For the taxi agent, performing a *Pickup*(t) in a location that is being cleaned carries the same penalty as performing a *Pickup*(t) in any other illegal state, and does not result in the passenger being picked up.

Task Decomposition

In the janitor agent, the following three subtasks have been identified:

- $Maintain(t)$. This task consist of navigating to location t and cleaning it.
- $Navigate(t)$. The parameter t indicates which of the four locations is the target location. In this subtask, the goal is to move the janitor from its current location to t.
- *Root*. This task is the overall task of the system, and represents the entire janitor task.

The task graph for the janitor is shown in Figure 5.3, and the taxi graph is shown in Figure 5.4 for easy reference.

In summary, the janitor should learn to prefer dirty locations where either no taxi is headed to pick up a passenger or a location where a taxi is in the middle of a *Pickup*, and will most likely be done by the time the janitor gets there. The taxi, in turn, should learn to prefer the locations where there is a passenger. Furthermore, the taxi should not start heading to a location that a janitor is headed to. If the janitor is in the middle of cleaning a location, the taxi should

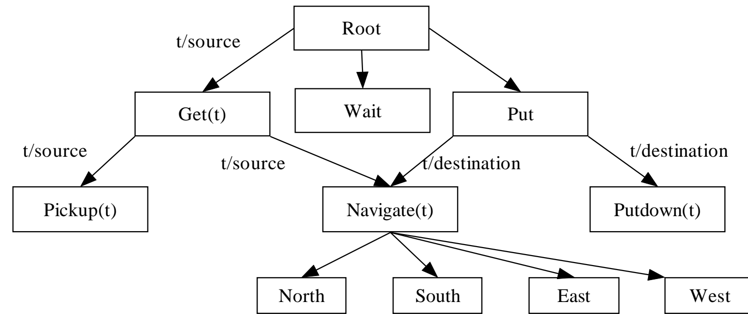


Figure 5.4: The taxi agent.

learn that heading there is safe, because the janitor will most likely be done by the time the taxi gets there.

In the expanded Taxi problem we clearly see a need for an algorithm that can handle agents with different task decompositions. In the following section a communication scheme for handling the problems introduced by heterogeneous agents is presented.

5.2.2 Communication Using Interest Groups

Instead of using the levels in the hierarchy to decide what information is valuable in a communication task, we introduce a new concept called *Interest Groups* (IG).

The idea is that in a given subtask, called an *Interest Task*, certain actions of other agents are of great importance to what action the agent should perform. Important actions are not necessarily children of the same task. Furthermore actions that are important to one type of agent might not be important to another type of agent. This is where the interest group approach differ from subtask sharing. In subtask sharing it is the actions of *one* task that are visible to other agents. When using interest groups it is not the actions of a specific task that are visible, but the actions of importance to the observing agent that are visible.

To illustrate this difference we will use the expanded Taxi problem. In this problem a taxi needs to know which actions other taxis are performing. If another taxi is performing a $Get(R)$ action the agent should learn that it is unfeasible to also perform a $Get(R)$. If all other taxis are performing either $Wait$ actions, Put actions, or Get actions to another locations, the agent should learn that it is safe to navigate for a passenger.

An interest task has an *Interest Group* attached to it. This interest group consists of the actions from the other agents that can affect which action to choose.. The taxi agent has only one interest task which is the *Root* task. The interest group attached to this interest task consists of the actions $Get(t)$,Put, and $Wait$ from the other taxi agents, and $Maintain(t)$ and $Clean(t)$ from the

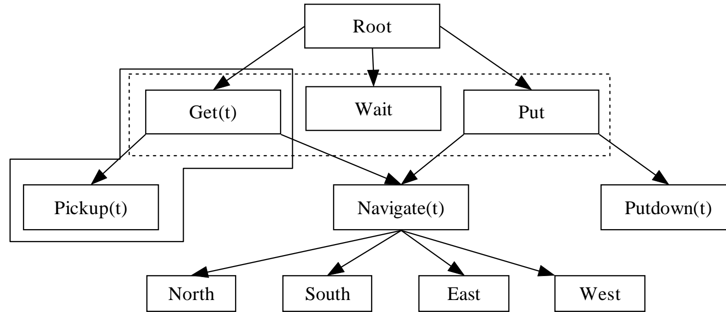


Figure 5.5: The taxi agent.

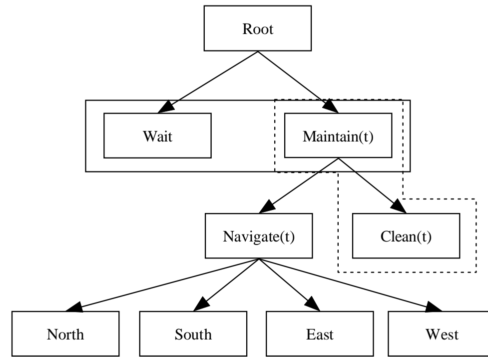


Figure 5.6: The janitor agent.

janitor agents. Using the interest group *Root* can see which actions the other agents are performing and choose its action accordingly.

Figure 5.5 shows the task decomposition of the taxi agent. The actions of importance to the interest task *Root* are marked with a dotted line. Likewise in the task graph of the janitor agent shown in Figure 5.6, the actions of importance to the taxi are marked with a dotted line.

A janitor observing a taxi agent is not interested in whether the taxi is waiting or on its way to put down a passenger. It is only interested in whether the agent is on its way to pick up a passenger, or if the agent is in the middle of picking up a passenger. This means that the only tasks of interest are the $Get(t)$ and $Pickup(t)$. If the problem contains two janitors, then each janitor needs to know what the other janitor is doing. The tasks of importance in this case are the $Maintain(t)$ task, and the $Wait$ task. In Figures 5.5 and 5.6 these tasks are represented with a filled line.

To further illustrate the concept of interest groups consider the Taxi problem board shown in Figure 5.7. In this figure we have two taxi agents, and two janitor agents. First assume that taxi agent t_1 is picking up a passenger from location Y, and that taxi agent t_2 is waiting. Now assume that janitor agent j_1 is cleaning the dirty location G(the grey shading indicates dirty locations).

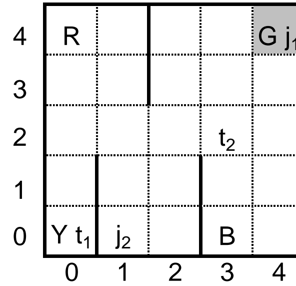


Figure 5.7: A taxi board with two taxies and two janitors.

The other janitor agent j_2 is waiting.

In this scenario the visible actions from the interest group for each agent would contain the following.

For agent t_1 the visible action from the other taxi agent would be $Wait$, while from janitor j_1 it would see $Clean(G)$. From janitor agent j_2 there would be no visible action, because it is performing an action that is of no importance to the taxi agent.

Agent t_2 would see the same janitor actions as taxi agent t_1. The visible action from t_1 is a $Pickup(Y)$.

For a janitor agent observing the two taxi agents, the visible actions would be a $Pickup(Y)$ for taxi agent t_1, because the taxi is picking up a passenger at location Y. For taxi agent t_2 there is no visible action. The visible action that j_1 can observe from janitor j_2 is a $Wait$. Janitor j_2 observes $Maintain(G)$ from janitor agent j_1.

Table 5.1 shows the observable actions of each agent. From this table the difference between a taxi observing a taxi, and a janitor observing a taxi becomes clear.

	obs. t_1	obs. t_2	obs j_1	obs. j_2
t_1		$Wait$	$Clean(G)$	
t_2	$Get(Y)$		$Clean(G)$	
j_1	$Pickup(Y)$			$Wait$
j_2	$Pickup(Y)$		$Maintain(G)$	

Table 5.1: The observable actions for the agents in Figure 5.7.The agents in the first column are the observing agents.

When a janitor observes a taxi it will either see the taxi as performing a $Pickup(t)$ action, as performing a $Get(t)$ action, or it will not be able to observe anything. When looking at the taxi graph we see that $Pickup(t)$ is actually a

descendent of *Get*(t) i.e. *Pickup*(t) occurs further down the task graph. In cases such as this where there are several tasks from the same interest group occurring in the same branch the rule is that the task furthest down the graph is the observable task.

In this case there are only two actions in the *Get*(t) task. A question is then why did we not just choose the *Navigate*(t) task as the action in the interest group? First and foremost, we cannot let *Navigate*(t) occur in the interest group, because we cannot know if it is initialised from a *Get* action, or if it initialised from a *Put* action. We could however just let the two actions be called *pickup*(t) and *NavigateGet*(t) and avoid this problem.

Imagine that the janitor agent is interested in if a taxi agent is in the *Get*(t) task subtree. If the taxi is in this part of the graph, the janitor is interested in whether the taxi is performing a pick up or if it is performing an child action to *Get*(t) that is not a pick up. If the action is not a pick up, then the janitor is not interested in the identity of the action, only that it is located in the *Get*(t) sub-tree. In this case having only *Pickup*(t) and *Get*(t) is a solution.

5.2.3 Projected Value Function

As is the case in the homogeneous setting, the projected value function of a hierarchical policy π on subtask M_i, denoted $V^{\pi}(i,s)$, is changed when the task i is an interest task. $V^{\pi}(i,s,\vec{u})$ is the expected cumulative reward of executing π_i, and its descendents, starting in state s until M_i terminates when i is an interest task. $\vec{u}$ are the actions being executed from the interest group of i.

The joint completion function for agent j, $C^j(i,s,u^1,\ldots u^n,a^j)$ is the expected discounted cumulative reward of completing interest task i after taking action a^j in state s while other agents are performing the observable action u^k from the interest group attached to i.

Because there can be several interest tasks in a task graph, an interest task can be found both as the current task i and as the selected action a, thus we must have a function decomposition for each of these cases.

In the case where both the current task and the chosen action are interest tasks, the decomposition can be restated as Equation 5.1.

$$Q^{j^{\pi}}(i,s,u^1\ldots u^n,a^j) = V^{j^{\pi}}(a^j,s,\widetilde{u}^1\ldots\widetilde{u}^n) + C^{j^{\pi}}(i,s,u^1\ldots u^n,a^j) \tag{5.1}$$

In this case both the value function for the chosen action a^j, and the completion function for the current task i must be redefined. In order for the functions to be dependent on the actions of other agents, the executing actions of other agents found in the interest groups of interest task i and interest task a^j must be represented. The actions in the interest group belonging to a^j are represented as $\widetilde{u}^1\ldots\widetilde{u}^n$. For the interest group belonging to i the actions are represented as $u^1\ldots u^n$.

The second case is when the current task i is an interest task, but the chosen action a^j is not. In this case only the completion function needs to rely on the

actions of other agents. The value function for a^j is independent of the other agents actions. Equation 5.2 shows the decomposition.

$$Q^{j^\pi}(i, s, u^1 \dots u^n, a^j) = V^{j^\pi}(a^j, s) + C^{j^\pi}(i, s, u^1 \dots u^n, a^j) \tag{5.2}$$

It might also be the case that the chosen action a^j is an interest task, while the current task i is not. In this case it is only the value function for a^j that depends on the actions of the other agents. Equation 5.3 shows this case.

$$Q^{j^\pi}(i, s, a^j) = V^{j^\pi}(a^j, s, \widetilde{u}^1 \dots \widetilde{u}^n) + C^{j^\pi}(i, s, a^j) \tag{5.3}$$

In the final case neither the current task i nor the chosen action a^j is an interest task. In this case Equation 5.4 can be applied. This is the same decomposition found in Chapter 3.

$$Q^{j^\pi}(i, s, a^j) = V^{j^\pi}(a^j, s) + C^{j^\pi}(i, s, a^j) \tag{5.4}$$

5.2.4 A Multi-agent Reinforcement Learning Algorithm for Heterogeneous Agents

The Heterogeneous learning algorithm is an extension to the MaxQ and the Cooperative MaxQ learning algorithm. Algorithm 12 shows this algorithm. The algorithm is initialised with the agent name as parameter, as well as the current task of that agent, and the current state. When initialising an episode, the current task will be the topmost task. In the janitor and taxi agent case this will be the *Root* task.

The first thing that happens is that all the interest groups $G_i \in G$ attached to agent a are found. For each interest group the observable actions currently being executed by other agents are found. The executing observable action in interest list k for interest task i is denoted u_i^k. For each interest group attached to the agent one action is observed.

When entering an iteration of the algorithm, a sequence, *Seq*, of elements is maintained. Each element consists of a state and the observable actions of each interest group of that agent being executed at the time the state was encountered. Each time the algorithm returns from an action, the sequence *childSeq*, which is the sequence encountered during the execution of the returning action, is added to *Seq*.

If the current task i is a primitive action then the action is executed, the value function for the primitive action is updated, and the state along with the observable executing actions of other agents are added to *Seq*.

If i is a composite task, then there are four special cases:

- i is an interest task, and the chosen action a^j is an interest task.
- i is an interest task, and the chosen action a^j is not.
- i is not an interest task, but the chosen action a^j is.

Algorithm 12 The Heterogeneous-CHRL Algorithm.

1: **Function** He-CHRL(Agent a, Task i, State s)
2: let G be a list of interest groups attached to Agent a
3: for each interest list IL_j in G_i where $G_i \in G$, add action u_i to U_i, where u_i is the action currently being executed in IL_j.
4: let $Seq = ()$ be the sequence of (state-visited, for each $G_i \in G$ add U_i)
5: **if** i is a primitive action **then**
6: execute action i, receive reward r and observe result state s'
7: $V_{t+1}^{j^\pi}(i,s) := (1-\alpha_t^j(i)) \cdot V_t^{j^\pi}(i,s) + \alpha_t^j(i) \cdot r_t$
8: push (state s, for each $G_i \in G$ add U_i) onto the front of Seq
9: **else**
10: **while** i has not terminated **do**
11: **if** i is an interest task **then**
12: Choose action a^j according to the current exploration policy $\pi_i^j(s, u_i^1 \dots u_i^n)$
13: let $ChildSeq$ = He-CHRL(j, a^j, s)
14: observe result state s' and current interest group actions $\widehat{u}_i^1, \dots, \widehat{u}_i^n$ for interest task i, and $\widehat{u}_j^1, \dots, \widehat{u}_j^n$ if a^j is an interest task.
15: **if** a_j is an interest task **then**
16: $a^* = \arg\max_{a' \in A_i} \left[C_t^{j^\pi}(i, s', \widehat{u}_i^1, \dots, \widehat{u}_i^n, a') + V_t^{j^\pi}(a', s', \widehat{u}_j^1, \dots, \widehat{u}_j^n) \right]$
17: **else**
18: $a^* = \arg\max_{a' \in A_i} \left[C_t^{j^\pi}(i, s', \widehat{u}_i^1, \dots, \widehat{u}_i^n, a') + V_t^{j^\pi}(a', s') \right]$
19: **end if**
20: let $v_x = V_t^{j^\pi}(a^*, s', \widehat{u}_j^1 \dots \widehat{u}_j^n)$ if a^j is an interest task, else let $v_x = V_t^{j^\pi}(a^*, s')$
21: let $N = 0$
22: **for** each element in $ChildSeq$ from the beginning **do**
23: $N = N + 1$
24: $C_{t+1}^{j^\pi}(i, s, u_i^1 \dots u_i^n, a^j) := (1-\alpha_t^j(i)) C_t^{j^\pi}(i, s, u_i^1 \dots u_i^n, a^j) + \alpha_t^j(i) \gamma^N \left[C_t^{j^\pi}(i, s', \widehat{u}_i^1 \dots \widehat{u}_i^n, a') + v_x \right]$
25: **end for**
26: **else**
27: choose action a^j according to the current exploration policy $\pi_i^j(s)$
28: let $ChildSeq$ = Heterogeneous-CHRL(j, a^j, s)
29: observe result state s', and current interest group actions $\widehat{u}_j^1, \dots, \widehat{u}_j^n$
30: **if** a_j is an interest task **then**
31: let $a^* = \arg\max_{a' \in A_i} \left[C_t^{j^\pi}(i, s', a) + V_t^{j^\pi}(a', s', \widehat{u}_j^1, \dots, \widehat{u}_j^n) \right]$
32: **else**
33: let $a^* = \arg\max_{a' \in A_i} \left[C_t^{j^\pi}(i, s', a) + V_t^{j^\pi}(a', s') \right]$
34: **end if**
35: let $v_x = V_t^{j^\pi}(a^*, s', \widehat{u}_j^1 \dots \widehat{u}_j^n)$ if a^j is an interest task, else let $v_x = V_t^{j^\pi}(a^*, s')$
36: let $N = 0$
37: **for** each state s in $ChildSeq$ from the beginning **do**
38: $N = N + 1$
39: $C_{t+1}^{j^\pi}(i, s, a^j) := (1-\alpha_t^j(i)) C_t^{j^\pi}(i, s, a^j) + \alpha_t^j(i) \gamma^N \left[C_t^{j^\pi}(i, s', a^*) + v_x \right]$
40: **end for**
41: **end if**
42: append $ChildSeq$ onto the front of Seq
43: $s = s'$
44: **end while**
45: **end if**
46: **return** Seq

- i is not an interest task, and neither is the chosen action a^j.

In the case where i is an interest task, the completion function will be dependant on the actions of other agents. In the case where a^j is an interest task, the value function for a^j will be dependant of the actions of other agents.

Line 16 of Algorithm 12 shown below as well as line 18, 31 and 33 have the same function as *EvaluateMaxNode* function from Algorithm 4

$$a^* = \arg\max_{a' \in A_i} \left[C_t^{j^\pi}(i, s', \widehat{u}_i^1, \dots, \widehat{u}_i^n, a') + V_t^{j^\pi}(a', s', \widehat{u}_j^1, \dots, \widehat{u}_j^n) \right]$$

5.3 Summary

In this chapter we presented two approaches that each dealt with an aspect of heterogeneity in multi-agent environments. First the inter-agent guidance approach was presented. In this approach we presented algorithms that was meant to increase the learning rate of agents that for one reason or another could not, or should not communicate too often. The second approach showed how agents with different goals could be made to work together. This approach builds upon Algorithm 7 presented in Chapter 4.

Chapter 6

Conclusion

At the beginning of this report we introduce a direction in machine learning called *Reinforcement Learning*. Reinforcement Learning tries to let agents learn using the same learning principles used in every day life. In its most basic form reinforcement learning suffers from scaling problems.

Even in the simplest and smallest domains, teaching a machine the skills needed to complete a task is not trivial. As the state space becomes larger, so does the amount of computation power needed, and in fact adding just a little complexity to a domain can cause a massive state explosion.

In an attempt to solve the scaling problem a technique called *Hierarchical Reinforcement Learning* is introduced. This technique decomposes a problem into a number of smaller problems. In each of these smaller problems state abstraction can be applied. In this report we showed that on a problem called *The Taxi Problem* the number of state/action pairs, when using a 5×5 board, could be reduced from 3000 when using regular table based reinforcement learning to only 632. When increasing the board size to 10×10, the difference became even more apparent. Now the regular approach required 12000 state/action pairs, whereas hierarchical reinforcement learning only required 2432. This shows, that as the problem becomes bigger so does the advantage of using hierarchical reinforcement learning.

Both approaches were tested on the Taxi problem, with the board size set to 10×10. The results of this test was very convincing. In this setting the HRL algorithm converged using just under $\frac{1}{8}$ of the time steps needed for regular Q learning. Furthermore, HRL converged 16 times faster in real time.

Having shown that hierarchical reinforcement learning does in fact provide a reduction in the state space as well as in the time used to converge, we applied it in a multi-agent setting.

Instead of just having two agents using the hierarchical learning algorithm a new approach was used. This approach, first introduced by Ghavamzadeh and Mahadevan, allows agents to coordinate at a high level of abstraction. This approach allows the agents to ignore low level details, while still getting an idea of what state the other agent is in, and what it is doing. Having knowledge

of other agents intentions should help the agents coordinate, and thus achieve a higher level of cooperation. To prove this claim an implementation of the approach was made, and subsequently compared to an implementation with multiple agents the regular hierarchical approach. As it turned out, the new approach was able to find policies that yielded higher rewards than what was possible using the regular hierarchical approach.

Chapter 5 presented two very different approaches that each added a form of heterogeneity to multi-agent environments. The first approach dealt with the heterogeneity introduced when agents are using different learning algorithms but are otherwise identical(including identical goals). We believe this approach will speed up learning when the value functions of the agents are distributed. The approach is based on the idea of experience sharing. What this means is, that when one agent has performed some sequence of actions that is believed to be good, then this sequence can be shared with other agents, thus improving the policy of other agents. The second approach dealt with agents that have different goals, but use the MaxQ function decomposition. In this approach the concept of *Interest Groups* were introduced. By using interest groups it is possible for two agents with entirely different goals to coordinate, so that they do not compete for the same resources. This approach builds upon the ideas presented in Chapter 3 and Chapter 4.

6.1 Future Work

First and foremost, testing and validation of the two approaches presented in Chapter 5 should be conducted. Furthermore testing of HRL and MHRL on other problems should be conducted. These tests should be made to validate that the approaches does in fact work on a variety of problem domains.

One issue that was not addressed in this report was how to find a suitable hierarchical structure to represent a problem. It could be interesting to investigate what it will take to create an algorithm that could handle this decomposition automatically.

One of the biggest problems with the MaxQ method is the time it takes to compute $V(i, s)$. The approach used in this report is shown in Algorithm 4. The problem with this approach is that calculating $V(i, s)$ requires a depth first search, where several paths through the task graph must be searched. The bigger the task graph the more branches must be searched, and the longer time it will take. It would be interesting to see if a more optimal approach to calculating $V(i, s)$ could be found.

While HRL and MHRL does provide a substantial reduction in the state space and an increase in the execution speeds, it still suffers somewhat when the problems gets bigger. It would be interesting to see if combining *Relational Reinforcement Learning* with hierarchical structures will yield improvements. It should be possible to reduce the state space even further because similar states can be abstracted away.

Appendix A

5 Step Rewards for the HRL Implementation

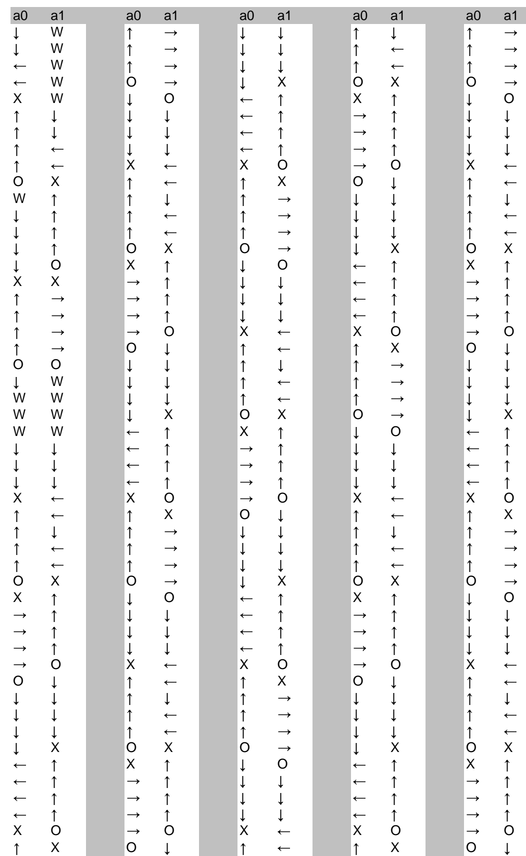
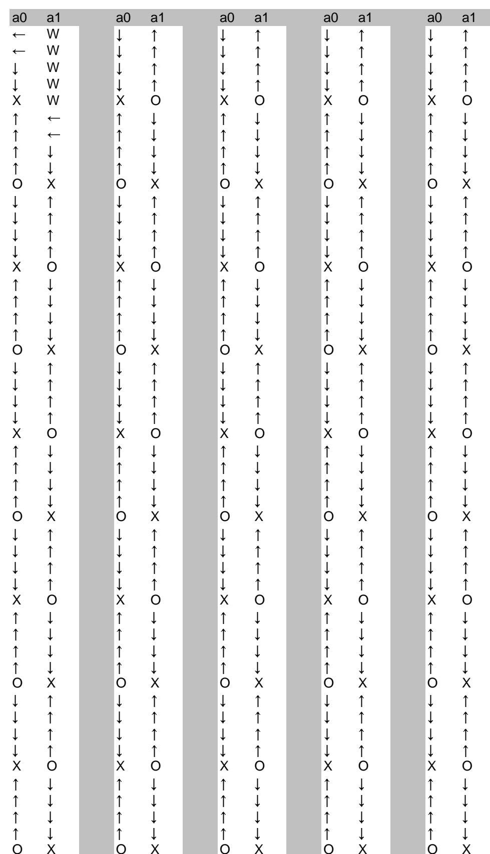


Figure A.1: Optimal sequence for 5 step MHRL agents.



a0	a1	a0	a1	a0	a1	a0	a1	a0	a1
←	W	↓	↑	↓	↑	↓	↑	↓	↑
←	W	↓	↑	↓	↑	↓	↑	↓	↑
↓	W	↓	↑	↓	↑	↓	↑	↓	↑
↓	W	↓	↑	↓	↑	↓	↑	↓	↑
X	W	X	O	X	O	X	O	X	O
↑	←	↑	↓	↑	↓	↑	↓	↑	↓
↑	←	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
O	X	O	X	O	X	O	X	O	X
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
X	O	X	O	X	O	X	O	X	O
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
O	X	O	X	O	X	O	X	O	X
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
X	O	X	O	X	O	X	O	X	O
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
O	X	O	X	O	X	O	X	O	X
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
X	O	X	O	X	O	X	O	X	O
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
O	X	O	X	O	X	O	X	O	X
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
↓	↑	↓	↑	↓	↑	↓	↑	↓	↑
X	O	X	O	X	O	X	O	X	O
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
↑	↓	↑	↓	↑	↓	↑	↓	↑	↓
O	X	O	X	O	X	O	X	O	X

Figure A.2: Optimal sequence for 5 step HRL agents.

Bibliography

[1] Richard Bellman. *Dynamic Programming*. Princeton University, 1957.

[2] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *CoRR*, cs.LG/9905014, 1999.

[3] Thomas G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *J. Artif. Intell. Res. (JAIR)*, 13:227–303, 2000.

[4] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational Reinforcement Learning. *Machine Learning*, 43(1/2):7–52, 2001.

[5] Mohammad Ghavamzadeh and Sridhar Mahadevan. Learning to communicate and act using hierarchical reinforcement learning. In *AAMAS*, pages 1114–1121, 2004.

[6] Harry Klopf. Brain Function and Adaptive Systems—A Heterostatic Theory. Technical report, Air Force Cambridge Research Laboratories, 1972. AFCRL-72-0164.

[7] Harry Klopf. *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere, Washington, D.C., December 1982. ISBN 089116202X.

[8] S. Mahadevan M. Ghavamzadeh. Hierarchical multiagent reinforcement learning. *CMPSCI Technical Report 04-02*, 2004.

[9] Tom M. Mitchell. *Machine Learning*, chapter 13 – Reinforcement Learning. McGraw-Hill, New York, 1997.

[10] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 0-07-115467-1.

[11] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1997.

[12] Khashayar Rohanimanesh and Sridhar Mahadevan. Learning to take concurrent actions. In *NIPS*, pages 1619–1626, 2002.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Bradford Books, 1998. ISBN 0-262-19398-1.

[14] Edward L. Thorndike. *Animal Intelligence*. Thoemmes Continuum, 1911. ISBN 185506698X.

[15] Christopher J.C.H. Watkins and Peter Dayan. Technical Note: Q-Learning. *Machine Learning*, 8(3-4):279–292, May 1992.

Summary

In this report we start out by describing the principles behind regular Q-learning. Following this short introduction to reinforment learning we describe how hierarchical structures can be applied to reinforcement learning. To this end we present the MaxQ reinforcement learning algorithm invented by Thomas Dietterich [3].

This algorithm decomposes a problem into smaller problems by applying a hierarchical structure on the problem. In each of the sub-problems it is very likely that some part of the overall state does matter, and can therefore be abstracted away. Because of the great abstraction ability of MaxQ scaling becomes less of a problem. To test that this is infact so a test was performed. In this test an agent using the MaxQ algorithm was compared to an agent using regular Q-learning. The results of this test was very convincing. In this setting the HRL algorithm converged using just under $\frac{1}{8}$ of the time steps needed for regular Q learning. Furthermore, HRL converged 16 times faster in real time.

In Dietterichs work the focus is on single-agent environments. We have expanded the MaxQ learning algorithm to include cooperating agents by using the Multi-agent HRL (MHRL) algorithm presented by M. Ghavamzadeh and S. Mahadevan in [8]. This report gives an in depth description of this approach. The idea in this approach is that by letting other agents know which high level action an agent is performing they can get a rough idea of in which state the agent is in. The knowledge of other agents states and actions can be used to coordinate the behaviour and thereby increase the agents cooperation skills. That this approach does indeed improve the agents cooperation skills is shown by comparing the policy found from two agents using this approach with two agents just using regular hierarchical reinforcement learning. The results showed that the cooperating agents was able to find policies that yieled a higher total reward than what was possible for the two non-cooperating agents.

The approach introduced by Ghavamzadeh and Mahadevan focuses on homogeneous agents. We explore the area of Heterogeneity in multi-agent environments. To this end two approaches that each deals with a different aspect of heterogenity is introduced. The first method uses something called inter-agent guidance, and is meant for agents that are heterogeneous in the sense that they use different learning algorithms to achieve the same goal. This algorithm also has the property of speeding up learning in cases where the value function is not shared between agents. The second approach expands the algorithm presented by Ghavamzadeh and Mahadevan to allow agents with different goals to cooperate. This is achieved by using a concept called interest groups.