Cooperative Reinforcement Learning for Independent Learners

Author: Bilal Hashem Kalil Abed-Alguni
BCS, MCS

A thesis submitted to the
University of Newcastle, NSW, Australia
for the degree of
Doctor of Philosophy
(Computer Science)

October 2014
Statement of Originality

The thesis contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. I give consent to the final version of my thesis being made available worldwide when deposited in the University’s Digital Repository**, subject to the provisions of the Copyright Act 1968. **Unless an Embargo has been approved for a determined period.

(Bilal Hashem Abed-Alguni)
# Contents

Statement of Originality ii  
List of Figures ix  
Abstract xi  
Acknowledgements xiv  
Abbreviations xv  

1 Introduction 1  
1.1 Cooperative Reinforcement Learning for Independent Learners . . 2  
1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4  
1.3 Research Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . 6  
1.4 Research Methodology . . . . . . . . . . . . . . . . . . . . . . . . . 9  
1.5 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  
1.6 Structure of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . . 12  

2 Agent-based Model 15  
2.1 Motivating Problem: Hunter Prey Problem . . . . . . . . . . . . . . 15  
2.2 Intelligent Agents . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16  
2.3 Agent Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  
2.4 Single-agent vs. Multi-agent Systems . . . . . . . . . . . . . . . . . 19  
  2.4.1 Single-agent Systems . . . . . . . . . . . . . . . . . . . . . . . 20  
  2.4.2 Multi-agent Systems . . . . . . . . . . . . . . . . . . . . . . . 20  
2.5 Multi-agent Systems Agendas . . . . . . . . . . . . . . . . . . . . . 21  
2.6 Multi-agent Systems Classifications . . . . . . . . . . . . . . . . . 23  
2.7 Multi-agent Learning . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
  2.7.1 Cooperative Multi-agent Learning Techniques . . . . . . . . 23  
  2.7.2 Team Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . 24  
  2.7.3 Concurrent Learning . . . . . . . . . . . . . . . . . . . . . . . 26  
  2.7.4 Learning and Communication . . . . . . . . . . . . . . . . . . 29
2.8 Famous Agent-based Models
  2.8.1 Belief-Desire-Intention Agent-based Model
  2.8.2 Swarm Intelligent Agent-based Model
  2.8.3 Hierarchical Multi-agent Learning
  2.8.4 Layered Multi-agent Learning
  2.8.5 Distributed Multi-agent Systems
  2.8.6 Distributed Hunter Prey Problem
  2.8.7 Advantages of Agents-based Models
  2.8.8 Limitations of Agent-based Models

2.9 Learning and Teaching

3 Markov Decision Process
  3.1 Markov Decision Processes
  3.2 Partially Observable Markov Decision Processes
  3.3 Stochastic Games
  3.4 Finite-horizon MDP vs Infinite-horizon MDP
  3.5 Bellman’s Equation
  3.6 Application of Markov Decision Processes
  3.7 Limitations of Markov Decision Processes
    3.7.1 Curse of dimensionality
    3.7.2 Memory Requirement
    3.7.3 Stationary Assumption
  3.8 Large Markov Decision Processes
  3.9 Decomposition Approaches for Markov Decision Process
    3.9.1 Factored Approaches
    3.9.2 Hierarchical Approaches
  3.10 Merging Markov Decision Processes
  3.11 Summary

4 Reinforcement Learning
  4.1 Introduction
  4.2 Reinforcement Learning Model
  4.3 Types of Policy
  4.4 Episodic vs. Continuous Tasks
  4.5 Discounted Reward
  4.6 Applications of Reinforcement Learning
  4.7 The Trade-off Between Exploration and Exploitation
  4.8 Action Selection Policies
  4.9 Temporal Credit Assignment
  4.10 Estimation of Value Function
    4.10.1 Dynamic Programming Methods
    4.10.2 Monte Carlo Methods
    4.10.3 Temporal Difference Learning
  4.11 Q-learning Algorithm
## Final Discussion

9.1 Review of Existing Research
- 9.1.1 Reinforcement Learning
- 9.1.2 Multi-agent Reinforcement Learning
- 9.1.3 Hierarchical Structure of Complex Systems

9.2 Conclusion
- 9.2.1 Research Question 1
- 9.2.2 Research Question 2
- 9.2.3 Research Question 3

9.3 Future Directions
- 9.3.1 Cooperative Dyna Architecture
- 9.3.2 QA-learning for Single-goal Hierarchical Systems
- 9.3.3 Hierarchical Reinforcement Learning algorithms
### List of Figures

2.1 An example of hunter prey problem on a $5 \times 5$ grid. .................. 16  
2.2 Model of an agent based on Russell and Norvig [2003]. .................. 17  
2.3 Mobile agent based on Lange and Oshima [1999] .......................... 18  
2.4 Distributed hunter prey problem composed of two sub-grids. ............ 19  
2.5 The hunter prey problem with a single hunter agent. ..................... 20  
2.6 The hunter prey problem with multiple agents. ........................... 21  
2.7 The hunter prey problem with a heterogeneous team of hunters. .......... 25  
2.8 A team of hunters composed of two disjoint groups of hunters. .......... 26  
2.9 Hierarchical multi-agent systems. ....................................... 31  
2.10 Distributed multi-agent system composed of three subsystems. ........... 32  
2.11 Distributed hunter prey problem. ....................................... 34  
2.12 A game of two players. .................................................. 36  
3.1 A view of an MDP based on Russell and Norvig [2003, Chapter 17]. .... 38  
3.2 An example of hunter prey problem on a $5 \times 5$ grid. .................. 40  
3.3 The visual area of hunter agent $H_3$ with depth 1. ....................... 42  
3.4 An example of hunter prey problem on a $4 \times 4$ grid. .................. 47  
3.5 Distributed hunter prey problem. ....................................... 50  
3.6 Distributed hunter prey problem composed of four sub-grids. ............ 51  
4.1 Reinforcement learning model from Sutton and Barto [1998]. .......... 66  
4.2 The problem formulation used in Dyna based on Sutton [1990]. ......... 81  
4.3 The Actor-critic architecture based on Raicevic [2006]. .................. 82  
5.1 The aggregation architecture based on Jiang and Kamel [2006]. ......... 93  
5.2 A reinforcement learner model based on Liu and Zeng [2006]. .......... 98  
6.1 Distributed RL Hierarchical Model. ...................................... 117  
6.2 Generalisation of DHLM agents. ....................................... 122  
6.3 Example of the two migration procedures. ................................ 124  
6.4 Migration Algorithm. .................................................... 125  
6.5 Partially finished hunting. .............................................. 126  
6.6 Service queue and inactive list. ....................................... 126  
6.7 Redistribution of worker agents. ....................................... 127  
6.8 Task scheduling. ....................................................... 130  
6.9 Sequential scheduling procedure. ....................................... 131
List of Figures

6.10 Priority scheduling procedure. .......................... 132
6.11 Parallel scheduling procedure. .......................... 133
6.12 Combined scheduling procedure. .......................... 134
6.13 Shortest Manhattan distance. .......................... 135
6.14 Specialised hunter agents at system level. .......................... 136
6.15 Roles in a distributed hunter prey problem (system level). .......................... 138
6.16 Ambush algorithm (system level). .......................... 139
6.17 Specialised hunter agents at sub-system level. .......................... 140
6.18 Roles in a distributed hunter prey problem (sub-system level). .......................... 140
6.19 Ambush algorithm (subsystem level). .......................... 141
6.20 Experiment 1: Performance of IDQL vs Q-learning. .......................... 147
6.21 Experiment 2: Performance of IDQL vs Q-learning. .......................... 148

7.1 Nearest emergency exit problem. .......................... 154
7.2 Q-table models for multiple goal systems. .......................... 159
7.3 QA-learning flowchart. .......................... 159
7.4 The second stage of the QA-learning algorithm. .......................... 160
7.5 An example of the Merge procedure of QA-learning. .......................... 162
7.6 Experiment 1: QA-learning with three integration scenarios. .......................... 164
7.7 Experiment 1: Single agent Q-learning. .......................... 165
7.8 Experiment 2: QA-learning with three integration scenarios. .......................... 166
7.9 Experiment 2: Single agent Q-learning. .......................... 167
7.10 Experiment 3: QA-learning with three integration scenarios. .......................... 168
7.11 Experiment 3: Single agent Q-learning. .......................... 169

8.1 Average number of moves per one episode in $5 \times 5$ grid. .......................... 183
8.2 Average number of moves per 10 episodes in $5 \times 5$ grid. .......................... 183
8.3 Average number of moves per 100 episodes in $5 \times 5$ grid. .......................... 184
8.4 Average number of moves per one episode in $10 \times 10$ grid. .......................... 185
8.5 Average number of moves per 10 episodes in $10 \times 10$ grid. .......................... 185
8.6 Average number of moves per 100 episodes in $10 \times 10$ grid. .......................... 186
8.7 Average number of moves per one episode in $20 \times 20$ grid. .......................... 187
8.8 Average number of moves per 10 episodes in $20 \times 20$ grid. .......................... 188
8.9 Average number of moves per 100 episodes in $20 \times 20$ grid. .......................... 189
8.10 Average number of moves per 10 episodes in $5 \times 5$ grid. .......................... 191
8.11 Average number of moves per 100 episodes in $5 \times 5$ grid. .......................... 192
8.12 Average number of moves per 10 episodes in $10 \times 10$ grid. .......................... 193
8.13 Average number of moves per 100 episodes in $10 \times 10$ grid. .......................... 193
8.14 Average number of moves per 10 episodes in $20 \times 20$ grid. .......................... 195
8.15 Average number of moves per 100 episodes in $20 \times 20$ grid. .......................... 196
8.16 Experiment 1: EQA-learning vs Q-learning. .......................... 200
8.17 Experiment 2: EQA-learning vs Q-learning. .......................... 201
8.18 Experiment 3: EQA-learning vs Q-learning .......................... 202
Abstract

Cooperative Reinforcement Learning for Independent Learners

by Bilal Abed-Alguni, BCS, MCS

University of Newcastle
Faculty of Engineering and Built Environment
School of Electrical Engineering and Computer Science

A thesis submitted for the degree of Doctor of Philosophy

Machine learning in multi-agent domains poses several research challenges. One challenge is how to model cooperation between reinforcement learners. Cooperation between independent reinforcement learners is known to accelerate convergence to optimal solutions. In large state space problems, independent reinforcement learners normally cooperate to accelerate the learning process using decomposition techniques or knowledge sharing strategies. This thesis presents two techniques to multi-agent reinforcement learning and a comparison study. The first technique is a formal decomposition model and an algorithm for distributed systems. The second technique is a cooperative Q-learning algorithm for multi-goal decomposable systems. The comparison study compares the performance of some of the best known cooperative Q-learning algorithms for independent learners.

Distributed systems are normally organised into two levels: system and subsystem levels. This thesis presents a formal solution for decomposition of Markov Decision Processes (MDPs) in distributed systems that takes advantage of the organisation of distributed systems and provides support for migration of learners. This is accomplished by two proposals: a Distributed, Hierarchical Learning Model (DHLM) and an Intelligent Distributed Q-Learning algorithm (IDQL) that are based on three specialisations of agents: workers, tutors and consultants. Worker agents are the actual learners and performers of tasks, while tutor agents and consultant agents are coordinators at the subsystem level and the system level, respectively. A main duty of consultant and tutor agents is the assignment of problem space to worker agents. The experimental results in a distributed hunter prey problem suggest that IDQL converges to a solution faster than the single agent Q-learning algorithm. An important feature of DHLM is that it provides a solution for migration of agents.
This feature provides support for the IDQL algorithm where the problem space of each worker agent can change dynamically. Other hierarchical RL models do not cover this issue.

Problems that have multiple goal-states can be decomposed into sub-problems by taking advantage of the loosely-coupled bonds among the goal states. In such problems, each goal state and its problem space form a sub-problem. This thesis introduces Q-learning with Aggregation algorithm (QA-learning), an algorithm for problems with multiple goal-states that is based on two roles: learner and tutor. A learner is an agent that learns and uses the knowledge of its neighbours (tutors) to construct its Q-table. A tutor is a learner that is ready to share its Q-table with its neighbours (learners). These roles are based on the concept of learners reusing tutors’ sub-solutions. This algorithm provides solutions to problems with multiple goal-states. In this algorithm, each learner incorporates its tutors’ knowledge into its own Q-table calculations. A comprehensive solution can then be obtained by combining these partial solutions. The experimental results in an instance of the shortest path problem suggest that the output of QA-learning is comparable to the output of a single Q-learner whose problem space is the whole system. But the QA-learning algorithm converges to a solution faster than a single learner approach.

Cooperative Q-learning algorithms for independent learners accelerate the learning process of individual learners. In this type of Q-learning, independent learners share and update their Q-values by following a sharing strategy after some episodes learning independently. This thesis presents a comparison study of the performance of some famous cooperative Q-learning algorithms (BEST-Q, AVE-Q, PSO-Q, and WSS) as well as an algorithm that aggregates their results. These algorithms are compared in two cases: equal experience and different experiences cases. In the first case, the learners have equal learning time, while in the second case, the learners have different learning times. The comparison study also examines the effects of the frequency of Q-value sharing on the learning speed of independent learners.

The experimental results in the equal experience case indicate that sharing of Q-values is not beneficial and produces similar results to single agent Q-learning. While, the experimental results in the different experiences case suggest that each of the cooperative Q-learning algorithms performs similarly, but better than single agent Q-learning. In both cases, high-frequency sharing of Q-values accelerates the convergence to solutions compared to low-frequency sharing. Low-frequency Q-value
sharing degrades the performance of the cooperative Q-learning algorithms in the equal experience and different experiences cases.
Acknowledgements

This Ph.D thesis would not have seen the light without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

First and foremost I offer my sincerest gratitude to my supervision team: A/Prof. Stephan Chalup, A/Prof. Frans Henskens, A/Prof. Huilin Ye, Dr David Paul, and Dr Yuqing Lin for the continuous support of my Ph.D research. Their patience, advice, encouragement, motivation, and enthusiasm have been of great value to me. Indeed, without their guidance, I would not be able to put the topic together.

I would like to acknowledge the financial, academic and technical support of the University of Newcastle and its staff, particularly in the award of University of Newcastle International Postgraduate Research Scholarship (UNIPRS) and University of Newcastle Research Scholarship Central 5050 that provided the necessary financial support for this research. The library and computer facilities of the university have been indispensable for the completion of my research and writing my doctoral thesis.

I would like to thank everybody who has been involved in the Interdisciplinary Machine Learning Research Group (IMLRG), and the Distributed Computing Research Group (DCRG).

I would like to thank my confirmation committee: A/Prof. Frans Henskens, Dr Yuqing Lin, and Dr Alexandre Mendes. I admit that my research would be much poorer without the constructive criticism and advice that were provided during my confirmation exam.

Finally, and most importantly, I would like to thank my family for their unconditional support, especially my wonderful wife Flordeliza, my lovely daughter Rua, my parents Hashem and my departed mother Sukiba, my sisters Yara Rua Lara Maha, and my brother Tareq. Your encouragement and emotional support helped me to overcome the hard moments that I have been through during my research. I will always owe them a debt of gratitude for believing in me, even when I did not. A smile from baby Rua was enough to encourage me and meant the world to me.
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABM</td>
<td>Agent-Based Model</td>
</tr>
<tr>
<td>AC</td>
<td>Actor-Critic</td>
</tr>
<tr>
<td>AMRLS</td>
<td>Aggregated Multiple Reinforcement Learning System</td>
</tr>
<tr>
<td>AOEs</td>
<td>Areas Of Expertise</td>
</tr>
<tr>
<td>AVE-Q</td>
<td>Average Q-learning</td>
</tr>
<tr>
<td>BA</td>
<td>Boltzman Addition</td>
</tr>
<tr>
<td>BEST-Q</td>
<td>BEST Q-learning</td>
</tr>
<tr>
<td>BM</td>
<td>Boltzman Multiplication</td>
</tr>
<tr>
<td>CBR</td>
<td>Case-Based Reasoning</td>
</tr>
<tr>
<td>CG</td>
<td>Coordination Graph</td>
</tr>
<tr>
<td>CST</td>
<td>Constructing Skill Trees</td>
</tr>
<tr>
<td>DFG</td>
<td>Dissolution and Formation of Groups</td>
</tr>
<tr>
<td>DHLM</td>
<td>Distributed Hierarchical Learning Model</td>
</tr>
<tr>
<td>FMDP</td>
<td>Factored Markov Decision Process</td>
</tr>
<tr>
<td>FMQ</td>
<td>Frequent Maximum Q-values</td>
</tr>
<tr>
<td>FIRL</td>
<td>Feature Construction for Inverse Reinforcement Learning</td>
</tr>
<tr>
<td>MARL</td>
<td>Multi-agent Reinforcement Learning</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi-agent System</td>
</tr>
<tr>
<td>MCLA</td>
<td>Multi-agent Cooperative Learning algorithm</td>
</tr>
<tr>
<td>MCLM</td>
<td>Multi-agent Cooperative Learning Model</td>
</tr>
<tr>
<td>MDP</td>
<td>Markov Decision Process</td>
</tr>
<tr>
<td>MV</td>
<td>Majority Voting</td>
</tr>
<tr>
<td>IDQL</td>
<td>Intelligent Distributed Q-Learning</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>POMDP</td>
<td>Partially Observable Markov Decision Process</td>
</tr>
<tr>
<td>PSO-Q</td>
<td>Particle Swarm Optimisation Q-learning</td>
</tr>
<tr>
<td>QA-learning</td>
<td>Q-learning by Aggregation</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>RV</td>
<td>Rank Voting</td>
</tr>
<tr>
<td>SARSA</td>
<td>State-Action-Reward-State-Action</td>
</tr>
<tr>
<td>SCMARL</td>
<td>State-clusters Multi-agent Reinforcement Learning</td>
</tr>
<tr>
<td>SRL</td>
<td>Selfish Reinforcement Learning</td>
</tr>
<tr>
<td>WSS</td>
<td>Weighted Strategy Sharing</td>
</tr>
<tr>
<td>WMV</td>
<td>Weighted Majority Voting</td>
</tr>
<tr>
<td>2LRL</td>
<td>Two-Level Reinforcement Learning</td>
</tr>
</tbody>
</table>
Dedicated to my family.
Chapter 1

Introduction

In machine learning, an intelligent agent is an autonomous entity designed to use intelligence to learn an assigned task. Normally, an intelligent agent uses sensors to collect sensory data about its environment and actuators to carry out actions that may affect its environment.

An intelligent agent needs a machine learning algorithm to learn how to perform its assigned tasks. One famous machine learning algorithm is Reinforcement Learning (RL). RL is suitable for interactive learning in which the agents learn from interacting with their environment through trial and error [Sutton and Barto, 1998]. In RL, a learner applies actions, and receives rewards for these actions in order to discover the actions that produce the highest reward.

The performance of a reinforcement learner typically degrades with increase of the size of the learning problem. Cooperation between independent reinforcement learners can accelerate the learning process of large problems. Cooperation can be accomplished using decomposition techniques that divide large problems into small problems. Cooperation can be also modelled by sharing information between agents.
1.1 Cooperative Reinforcement Learning for Independent Learners

There are two types of concurrent Reinforcement Learning (RL) systems: heterogeneous and homogeneous. Learners in heterogeneous concurrent RL systems are normally designed to learn a collective learning task simultaneously [Erus and Polat, 2007, Ghavamzadeh et al., 2006, Hansen and Feng, 2000, Ono and Fukumoto, 1997, Tosic and Vilalta, 2010]. A collective learning task is a task that requires more than one learner to learn, with each learner learning a sub-task of the collective learning task. The desired collective task should be decomposed into sub-tasks in order to implement concurrent RL to it. In this case, each learner learns one or more sub-tasks. Then the sub-solutions of the sub-tasks can be combined into a general solution for the collective task.

In homogeneous concurrent RL, the learners learn the same task at the same time. In this case, the learners exchange information such as policies, sensory data, and/or episodes among each other at frequent intervals [Tan, 1993]. Information sharing is known to accelerate the learning process of the independent learners in homogeneous concurrent RL systems [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002, Iima and Kuroe, 2006, 2007, 2008].

Complex learning problems have large state and action spaces. These problems have hierarchical structure that can be decomposed into simpler problems that are loosely coupled [Guestrin and Gordon, 2002]. The hierarchical RL algorithms exploit this structure and use it to decompose complex learning problems into smaller sub-problems that can be processed independently [Erus and Polat, 2007, Ghavamzadeh et al., 2006, Hansen and Feng, 2000, Ono and Fukumoto, 1997, Tosic and Vilalta, 2010]. Distributed systems [Hengst, 2002, 2004] and multi-goal systems [Karlsson, 1997, Sprague and Ballard, 2003] are examples of systems that have hierarchical structure. Distributed systems are organised into system and subsystem levels.
Multi-goal systems are usually decomposed into sub-problems in which each goal and its problem space form a sub-problem.

One of the best known RL algorithms is Q-learning. This algorithm finds mappings from state-action pairs to values by trying to approximate the value iteration algorithm using samples. These values are known as Q-values and are calculated using a utility function called the Q-function. The Q-values are normally stored in a table called the Q-table. The Q-function is an action-value function that returns the expected utility of taking a given action in a given state and following a fixed policy after that [Watkins, 1989, Watkins and Dayan, 1992]. This function is used to rank alternative actions in a given state according to their expected utility to achieving a learning goal. The rule that the learning agent follows in selecting actions in a given state is known as a policy [Sutton and Barto, 1998]. Following a policy enables the learner to accomplish its learning goal.

Sharing of Q-values between concurrent reinforcement learners is known to accelerate the learning process of individual learners without the need for decomposition of the problem space [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002, Iima and Kuroe, 2006, 2007, 2008]. The learning process of cooperative Q-learning algorithms for independent learners normally takes place in two stages. First, the individual learning stage, where each learner learns independently using its own Q-learning algorithm. Second, the learning by interaction stage in which the learners share and update their Q-values by following a Q-value sharing strategy. A sharing strategy determines how each agent should update its Q-table at the learning by interaction stage of cooperative Q-learning. Some of the best known cooperative RL algorithms for independent learners are BEST-Q, AVE-Q, PSO-Q [Iima and Kuroe, 2006, 2007, 2008], and WSS [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002].

The contribution of this dissertation is the development of approaches to improve cooperative Reinforcement Learning for decomposable systems.

One of these approaches allows agents to migrate between subsystems.
of the decomposed problem to improve overall efficiency. The second method decomposes multiple goal-states problems into a set of subsystems that can be learnt independently and then combines these sub-solutions to obtain an overall solution for the system. In addition, this thesis compares some of the best known cooperative Q-learning algorithms for independent learners based on their performance.

The remainder of this chapter is organised as follows: Section 1.2 motivates this research by examining how reinforcement learners can cooperate to learn tasks. In Section 1.3, we specify the goals of our research and decompose the research problem into specific questions. The methodology for addressing these questions is specified in Section 1.4. Section 1.5 concludes the chapter by briefly describing the contributions of this work. Section 1.6 outlines the structure of the remainder of this dissertation.

## 1.2 Motivation

The convergence speed of classical RL algorithms decreases with increase of the size of the state and action spaces. In fact, a large state space cannot be represented as Q-tables because the size of its representation needs large memory to store [Tesauro, 1995]. Additionally, the amount of training data required to learn a large state space problem is normally too large and requires such long training time regardless of the availability of sufficient computational memory to represent it as Q-tables [Asadi and Huber, 2004, Sutton and Barto, 1998].

Applying RL directly to large multi-agent systems is a time consuming process because of the size of the state space of such systems. For example, a team of \( n \) agents in an environment of \( m \) states has a learning space as large as \( m^n \) states. Team learning is a simple approach to cooperative Multi-Agent RL (MARL) in which a single learner applies a single agent RL algorithm to learn a set of behaviours for a group of agents. But, the fact that a single agent learns the behaviours of all
agents in team learning, makes the state space very large. An alternative approach to team learning is concurrent learning where each agent is a reinforcement learner that learns concurrently with the other members in the multi-agent system. The main goal here is to project the joint state space into \( m \) individual spaces. The risk of such an approach is that the existence of multiple concurrent learners affects the stationary attribute of the agents’ environment.

One of the most efficient approaches to concurrent learning is the hierarchical approach [Erus and Polat, 2007, Ghavamzadeh et al., 2006, Ono and Fukumoto, 1997, Tosic and Vilalta, 2010] in which large problems are decomposed into more manageable, simpler components. This approach accelerates the learning process of the reinforcement learners. The loose coupling characteristic is an important condition for the success of hierarchical decomposition. This characteristic means that each one of the system components has or uses little knowledge about other components [Kaye, 2003]. Current research studies have identified two types of decomposition of Markov Decision Processes (MDPs). First, static decomposition that partially or totally requires the implementation designers to define the hierarchy [Parr and Russell, 1997, Sutton et al., 1999]. Second, dynamic decomposition, in which hierarchy components, their positions and abstractions are determined during the simulation process [Dietterich, 2000, Hengst, 2002, Tosic and Vilalta, 2010]. Distributed systems and multi-goal systems have natural decomposable structures that can be decomposed into sub-problems. Each subsystem is a sub-problem in distributed systems, while each single goal and its problem space is a sub-problem in multi-goal systems.

Another efficient approach to concurrent learning is the cooperative RL approach for independent learners. In this approach, the learners alternate between two modes of learning. In the first mode, each learner applies a RL algorithm to the same task. In the second mode, the learners share and update their Q-values based on a sharing strategy. BEST-Q, AVE-Q, PSO-Q, and WSS are examples of the best known cooperative RL algorithms for independent learners. A cooperative RL algorithm is
distinguished from other cooperative RL algorithms based on the sharing strategy each algorithm implements.

Reusability is an important benefit of the hierarchical decomposition and cooperative RL approaches. In the first approach, each concurrent learner in decomposable systems can extend its policy to cover one or more subsystems or construct a general policy for the whole system using the knowledge of the other concurrent learners. In the second approach, the Q-values calculated by the learners during the first mode of learning are reused in the second mode of learning.

1.3 Research Objectives

The focus of this research is on cooperative independent RL systems and algorithms. In cooperative independent RL, learners cooperate among each other to converge faster to an optimal policy.

Applying RL directly to large multi-agent systems is impractical because of the size of the state space of these problems and the coordination required between agents to accomplish their tasks. The first goal of this research is thus:

1. To identify the main problems of applying RL to concurrent multi-agent systems, and discuss the solutions that have been proposed to address these problems.

Current RL decomposition techniques do not allow migration of learners from one problem space to another in distributed systems. Instead they focus on decomposing the state or action space into more manageable parts, and statically assign each learner to one of these parts. By decomposing RL problems into system and subsystem levels, the aim of this component of the research is thus two-fold:
2. To develop a formal system for independent cooperative reinforcement learners in distributed systems.

3. To model dynamic migration of reinforcement learners between subsystems in distributed systems.

In multiple goal-states decomposable systems, each reinforcement learner can learn a particular objective, a group of objectives, or all the objectives. The challenge is how independent learners in different subsystems can cooperate to build their individual policies and a global policy. The aim of this component of the research is also two-fold:

4. To model and evaluate sharing of information among independent reinforcement learners in multiple goal-states systems based on two roles: tutor and learner roles.

5. To find a solution for the problem of slow information sharing among multiple reinforcement learners.

Cooperative Q-learning algorithms for independent learners accelerate the learning process of the individual learners. In cooperative Q-learning, the learners share their Q-values by following a sharing strategy, after some episodes spent learning independently. A sharing strategy determines how each learner should share and update its Q-values. The aim of this component of the research is also two-fold:

6. To compare the performance of some of the best known cooperative Q-learning algorithms (BEST-Q, AVE-Q, PSO-Q, and WSS).

7. To study the effects of the frequency of Q-value sharing on the learning speed of independent learners.

As described above, learning problems that have large state spaces need longer training time and use more memory than smaller problems. Fortunately, many large state
space problems can be decomposed into loosely coupled subsystems that can be processed independently [Barto and Mahadevan, 2003, Daoui et al., 2010, Ghavamzadeh et al., 2006, Hansen and Feng, 2000] such as distributed systems and multi-goal decomposable systems. Current RL decomposition techniques focus on decomposing the state or action space into more manageable parts, and statically assign each learner to one of these parts. A solution model for concurrent reinforcement learners in multiple goal decomposable systems should provide three levels of policies: single goal policy, policy for a group of goals, and general policy for all goals. However, until now, there have been no formal solutions for decomposition of MDPs in multi-goal decomposable systems that provide these three types of policies.

There is no need for decomposition of the problem space in homogeneous concurrent RL algorithms. In these algorithms, the learners share and update their Q-values among each other at frequent intervals using a sharing strategy. Unfortunately, there have been no comparison studies in this field that compare famous cooperative RL algorithms (BEST-Q, AVE-Q, PSO-Q, and WSS), and study the effects of the frequency of Q-value sharing on the learning speed of the independent learners.

**Problem Definition:** How to improve cooperation between independent reinforcement learners in two types of decomposable systems: distributed and multi-goal decomposable systems? What are the best cooperative Q-learning algorithms for independent learners based on the speed of convergence to an optimal solution?

To answer the problem defined above, we decompose the problem into the following research questions:

1. How to model cooperative distributed systems using different types of learners? Given that a system is composed of loosely coupled subsystems, how can a cooperation algorithm provide support for learners’ migration from one subsystem to another?
2. How can reinforcement learners reduce the training time and use less memory for learning problems with large state spaces and multiple goals? How can the reinforcement learners compose one general solution from the solutions of several goal states? What levels of support can be offered to facilitate cooperation between multiple learners?

3. What are the main differences between the best known Q-value sharing strategies that are used to model the second stage of cooperative Q-learning for independent learners? How do these sharing strategies compare based on their performance? What are the effects of the frequency of Q-value sharing on the learning speed of independent learners?

1.4 Research Methodology

This section describes the methodology used for the research presented in this thesis. The methodology comprises five stages:

1. Problem definition. In this stage, the previous research work in single-agent RL and MARL is investigated to identify the main problems in the literature. Of special interest are the main methods that are used to accelerate and enhance the learning process of independent reinforcement learners.

2. Literature review. In this stage, the previous research work in agent-based modelling, single-agent RL, MDP and MARL is investigated to establish a theoretical framework for the research, define main concepts and terminologies, and identify previous models and algorithms that are related to the research presented in the current thesis.

3. Designing of solutions. This involves development of two techniques for cooperation of independent reinforcement learners and a comparison study. The first technique investigates the efficiency of the implementation of the specialisation
design principle to decomposable distributed learning problems. The second technique investigates the efficiency of the implementation of the specialisation design principle to multi-goal decomposable systems. The comparison study compares the performance of some of the best known cooperative RL algorithms for independent learners in two cases: equal experience and different experiences cases.

4. Validation of the solutions. In this stage, the presented techniques in the previous stage are verified and evaluated to determine that they solve the problems. A variation of the hunter-prey problem called distributed hunter prey problem is used to evaluate the performance of the first technique based on the convergence to optimality method. A variation of the shortest path problem called nearest emergency exit problem is used to evaluate the performance of the second technique based on two methods: the convergence to optimality and the Speed of convergence to near optimality. The hunter prey problem is used to evaluate the performance of cooperative RL algorithms for independent learners based on the convergence to optimality method.

5. Evaluation and Conclusions. In this stage, we will look at the benefits, and limitations of the proposed solutions.

1.5 Contributions

This thesis presents two new techniques to MARL and a comparison study between some of the best known cooperative RL algorithms for independent learners. These are achieved by diagnosing the strengths and weaknesses of previous research of MARL and their possible enhancements.

The first technique provides a solution to RL in distributed systems which is accomplished by two proposals. First, a distributed hierarchical learning model (DHLM)
that is based on three specialisations of agents: workers, tutors and consultants. Second, an intelligent distributed Q-learning algorithm (IDQL) that models the problem space of each agent specialisation of DHLM as a separate RL problem.

DHLM provides a solution for migration of worker agents through consultant and tutor agents. A consultant agent can assign the system level problem space to worker agents, while each tutor agent can assign its subsystem problem space to the worker agents that exist in or migrate to its subsystem. These roles provide support for the IDQL algorithm where the problem space of each worker agent can change dynamically. The experimental results in a distributed hunter prey problem suggest that the IDQL algorithm performs better than the single agent Q-learning algorithm.

The second technique is a cooperative Q-learning algorithm for multi-goal decomposable systems. This is accomplished by proposing an algorithm for cooperative policy construction for independent learners, Q-learning with Aggregation (QA-learning), which is based on two roles that an agent can play: learner and tutor. An agent is a learner, if it learns and uses the knowledge of its neighbours (tutors) to construct its Q-table. A tutor is a learner that is ready to share its Q-table with its neighbours (learners). The partial solutions of the learners can be combined to form a general solution for the multi-goal problem. The experimental results in an instance of the shortest path problem suggest that the QA-learning algorithm performs better than the single agent Q-learning algorithm, and the output of QA-learning is comparable to the output of a single Q-learner whose problem space is the whole system.

In cooperative Q-learning algorithms for independent learners, the learners learn for a period of time before sharing their Q-values among each other using a sharing strategy. The main difference between some of the best known cooperative Q-learning algorithms for independent learners (BEST-Q, AVE-Q, PSO-Q, and WSS) is the type of the sharing strategy each one uses. The comparison study compares the
Chapter 1. Introduction

Performance of these algorithms as well as an algorithm that aggregates their results. The study also examines the effects of the frequency of Q-value sharing on the learning speed.

The above cooperative Q-learning algorithms were implemented to the classical hunter prey problem in two cases. In the first case, the learners have equal experience, while in the second case, the learners have different experiences. The obtained results suggest that the cooperative Q-learning algorithms and single agent Q-learning have similar performance in the equal experience case. On the other hand, the results in the different experiences case suggest that each of the cooperative Q-learning algorithms has a similar performance that is better than single agent Q-learning when Q-value sharing is highly frequent. In both cases, high frequency sharing of Q-values accelerates the convergence to optimal solutions compared to low frequency sharing with one exception: BEST-Q. The main disadvantage of BEST-Q is that the optimal Q-values might not be found because the Q-values of all of the agents become the same after each time they share their Q-values among each other. The obtained results suggest that the performance of BEST-Q can be enhanced by allowing the learners to train for sufficient number of episodes before sharing their Q-values.

1.6 Structure of the Thesis

This section describes the structure of the main document:

Chapter 1 “Introduction” describes the motivation, research problem, research questions, objectives, research methodology, structure, and contribution of the thesis.

Chapter 2 “Agent-based Model” gives background information about multi-agent systems. In particular, this chapter gives an overview of the concepts of multi-agent systems and presents the most common multi-agent frameworks.
Chapter 3 “Markov Decision Process” gives background information about MDP. The chapter also introduces previous research about decomposition techniques for solving large MDPs. In particular, factored and hierarchical decomposition approaches.

Chapter 4 “Reinforcement Learning” gives background information that is necessary to understand Reinforcement learning. This includes an overview of the concepts, models and the famous algorithms of Reinforcement learning.

Chapter 5 “Multi-agent Reinforcement Learning” introduces previous research in MARL diagnosing strengths and weaknesses and possible enhancements.

Chapter 6 “Hierarchical Reinforcement Learning Model” presents a distributed hierarchical learning model (DHLM) and an intelligent distributed Q-learning algorithm (IDQL) based on the loosely coupled nature of distributed systems components.

Chapter 7 “Hierarchical Cooperative Policy Construction for Independent Q-Learners” presents and evaluates an algorithm for cooperative policy construction for independent learners in multiple goal-states problems.

Chapter 8 “Cooperative Q-learning Algorithms for Independent Learners” compares some of the best known cooperative Q-learning algorithms for independent Q-learners, and single agent Q-learning against each other. The chapter classifies these algorithms based on their performance, and studies the effects of the frequency of Q-value sharing on the learning speed.

Chapter 9 “Final Discussion” reviews the main findings and results, and discusses future work and possible applications.
Chapter 2

Agent-based Model

Agent-based systems technology is an increasingly used paradigm for conceptualising, designing, and implementing software systems [Jennings, 2000]. Agents are autonomous computer programs that are designed to use intelligence to automatically perform complex tasks. Many computer software applications require the use of multiple software agents to model them. Such systems are known as Multi-agent systems (MASs). A MAS is a system that consists of a number of agents that interact with each other in an environment. This chapter introduces agent-related concepts that help in understanding MASs. This chapter mainly focuses on discussing concepts, types, classifications, learning processes, learning models, and the main challenges of multi-agent systems. The agent-related concepts that will be discussed in this chapter will be used in the later chapters.

2.1 Motivating Problem: Hunter Prey Problem

The hunter prey, or pursuit, problem is considered one of the standard test problems in the field of machine learning [Liu and Zeng, 2006, Pakizeh et al., 2013]. This problem was first introduced by Benda et al. [1986] and, since then, many variations
have been proposed. The hunter prey problem, described in the next paragraph, will be used in this chapter to illustrate agent-related concepts.

In the hunter prey problem, there are normally two types of players, hunter (H) and prey (P), that are randomly positioned in the cells of a grid at the beginning of the game. The players can move in four directions (up, down, right, left) unless there is an obstacle, another agent, or boundary blocking the player’s way. For example, Figure 2.1 shows a classical version of the hunter prey problem, of grid size $5 \times 5$, that involves four players: three hunters and one prey.

![Figure 2.1: An example of hunter prey problem on a $5 \times 5$ grid. Filled squares represent obstacles hunters(H) and prey (P) cannot pass.](image)

In a typical hunter prey game, hunters chase the prey, and the prey tries to escape from the hunters. At any instant, the distance between any two players in a grid is measured using the Manhattan distance [Erus and Polat, 2007].

### 2.2 Intelligent Agents

An intelligent agent is an autonomous entity that is designed to use intelligence to automatically perform an assigned task. In addition, an intelligent agent may learn or use knowledge to perform its task. Normally, the agent perceives sensory data (input) from its environment and carries out actions (output) that may affect its environment [Russell and Norvig, 2003] (Figure 2.2).
In the hunter prey problem, the players can be modelled as intelligent agents. The hunter agent can apply a learning algorithm to learn how to catch the prey, while the prey agent can apply a learning algorithm to learn how to escape from its chasers. Table 2.1 shows the goals, sensor data, and actions of the hunter and prey agents.

<table>
<thead>
<tr>
<th>Hunter Agent</th>
<th>Prey Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goals</td>
<td>Capture the prey</td>
</tr>
<tr>
<td>Sensor Data</td>
<td>Position of the prey, obstacles and position of each other hunter</td>
</tr>
<tr>
<td>Actions</td>
<td>Move left, right, up, down or stay idle</td>
</tr>
</tbody>
</table>

Table 2.1: Hunter and prey agents.

2.3 Agent Types

In process management, process migration is the action of transferring a process between two machines. In terms of process migration, Lange and Oshima [1999] classified software agents into two types:

- Stationary agents are software agents that are bound to the host environments in which they were created. They can communicate with other agents or applications across the network using communication mechanisms such as message passing.
• Mobile agents are software agents that migrate across the network transferring their states and codes to other host environments (Figure 2.3).

Lange and Oshima [1999] listed many advantages that encourage the use of mobile agents:

• Mobile agents act independently and asynchronously of the mother process after being dispatched.

• Mobile agents use the proprietary protocols\(^1\) of their host environment.

• In critical real-time systems, network delay is unacceptable. One solution for this problem is the use of mobile agents, because they can be dispatched from a central controller and execute locally.

• Mobile agents execute locally at the destination environment. Thus, there is no need for communication between the client and the server applications during the execution of the mobile agent which decreases communication across the network.

• Mobile agents are fault tolerant; they can be designed to react dynamically to unfavourable situations. For example, mobile agents can be warned when a\(^1\) A private communication protocol for the host environment.
host system is being shut down, so they can dispatch and continue to work in another host environment.

- Mobile agents are platform-independent.
- Mobile agents act and react autonomously to changes in the host environment.

In the hunter prey domain, if we assume that we have a system of two distributed grids across the network and that the hunting process is a distributed one (Figure 2.4), then the hunter and prey agents that can move from one grid to another are considered mobile agents, while the players that are bounded to their original grids are considered stationary agents. In Figure 2.4, the hunter agent (H$_3$) and the prey agent (P$_1$) are examples of mobile agents that can cross the boundaries of their systems.

**Figure. 2.4:** Distributed hunter prey problem composed of two sub-grids. P$_1$ and H$_3$ move from left sub-grid to right sub-grid.

### 2.4 Single-agent vs. Multi-agent Systems

In general, there are two types of agent systems: centralised single-agent, and multi-agent systems [Stone and Veloso, 2000b]. In a centralised agent system, a single agent is responsible for making all the decisions in the system, while the other agents act as slaves. This section compares single-agent systems and multi-agent systems.
2.4.1 Single-agent Systems

In single-agent systems, the environment’s dynamics are determined by a single agent (central process). If there are other agents in the environment, they are considered as slaves\(^2\).

The hunter side of the hunter prey problem can be modelled using a single-agent approach regardless of the number of hunters in the game. The hunter agent decides the moves of each hunter in the game and considers the prey agent as a part of the environment. Figure 2.5 shows a hunter agent that controls the moves of three hunters. Each action in the action space of the hunter agent is a set of three primitive actions, where each primitive action (move left, right, up, down or stay idle) is performed by one hunter of the three hunters.

![Figure 2.5: The hunter prey problem with a single hunter agent. The hunter agent decides the behaviour of three hunters.](image)

2.4.2 Multi-agent Systems

A Multi-Agent System (MAS) is a system consisting of a number of agents that interact with each other in an environment. The components of the environment (like obstacles and cells in a maze problem) are considered passive agents because they do not have learning goals, while the actors (like a robot or a human in a maze

\(^2\)Slaves are agents that do not have goals and follow the instructions of the central process.
problem) who have goals in the environment are considered active agents [Kubera et al., 2010].

In general, the main difference between single-agent systems and multi-agent systems is that the environment’s dynamics are determined by more than one agent in multi-agent systems, while a central agent determines the environment’s dynamics in single-agent systems [Kabysh et al., 2012, Stone and Veloso, 2000b].

In a typical multi-agent version of the hunter prey problem, there is an agent per hunter. The structure of the agents (learning algorithms, decision systems and communication capabilities) can be heterogeneous or homogeneous.

![Figure 2.6: The hunter prey problem with multiple agents. Each hunter agent decides the behaviour of one hunter.](image)

### 2.5 Multi-agent Systems Agendas

A multi-agent agenda is a term used to define the way we view multi-agent systems. In fact, a MAS can be viewed in different ways depending on the technical perspective of the system viewer. For example, Shoham et al. [2007] presented five distinct Multi-agent agendas:
• **Computational agenda** that views the learning algorithm as a computational problem that requires iterative computations of some defined game properties.

• **Descriptive agenda** that describes the behaviours of the learning algorithm in a way that is near to human behaviours description.

• **Normative agenda** which distinguishes between equilibrium strategies and non-equilibrium strategies. The main goal of this process is to identify rules that bring the system to equilibrium.

• **Perspective cooperative agenda** is an agenda type that describes cooperative MAS models.

• **Non-cooperative descriptive agenda** that describes behavioural interaction of agents without real cooperation between agents. This agenda can be classified as a competitive agenda when some of the agents have opposite goals to the goals of other agents in the same multi-agent system. However, the non-cooperative descriptive agenda can also describe individual learning agents with different non-competitive goals learned at the same multi-agent system.

The research study of Sandholm [2007] is a refinement of the multi-agent agendas presented by Shoham et al. [2007]. The first refinement considers computational perspective; it states that direct learning algorithms are more computationally efficient than Multi-agent learning (MAL) algorithms, however the direct algorithms can only be used when the environmental rules are clearly defined. The second refinement is related to the descriptive agenda; it clarifies that learning rules should govern the learning process among certain entities. Third, the rules and the goals of the prescriptive agenda should be modelled and designed clearly for MASs, but the final design does not need to be exactly like the represented natural entities. Additionally, Sandholm [2007] divided the prescriptive agenda into cooperative, and non-cooperative agendas depending on the nature of the problem description.
2.6 Multi-agent Systems Classifications

Depending on the nature of multi-agent systems, multi-agent systems can be described as cooperative or competitive systems. The goal of agents in cooperative multi-agent systems is to maximise a group utility function, while the goal of competitive agents is to maximise their own utility [Hoen et al., 2005]. However, having different (individual) utility functions does not necessarily imply competition, especially if these functions have no interdependencies. When maximisation of one comes at the expense of maximising some other, then we have real competition.

This chapter is mainly concerned with cooperative multi-agent learning, rather than competitive learning.

2.7 Multi-agent Learning

The ability to learn is a main feature of intelligent agents. Many research studies in artificial intelligence have focused on the learning performed by one agent (e.g., [Rummery and Niranjan, 1994, Wang et al., 2007, Watkins, 1989, Watkins and Dayan, 1992]). In this type of learning, an agent learns to function successfully in an unknown dynamic environment [Shoham and Leyton-Brown, 2008]. Many techniques have been proposed for single-agent learning, however, in a multi-agent setting the learning environment is composed of several learners. A successful learner should take into account the existence of multiple learners in the same environment and the way they affect the dynamics of the environment [Shoham et al., 2007].

2.7.1 Cooperative Multi-agent Learning Techniques

Cooperative multi-agent systems are systems where agents use their interaction to cooperatively solve tasks or maximise some utility functions [Xian-yi et al., 2010].
In multi-agent systems, agents can cooperatively learn how to solve large tasks that are hard for a single agent to learn in a reasonable period of time. Panait and Luke [2005], and Hoen et al. [2005] argued that cooperative multi-agent learning techniques can be divided into two major categories. First, team learning in which a single learner is responsible for learning the behaviours of the entire team. Second, concurrent learning in which a learner for each team member is responsible for learning its behaviour.

The literature of team learning has focused on the heterogeneity of the team members, while the research in team learning has focused in the relationships between concurrent learners [Hoen et al., 2005].

### 2.7.2 Team Learning

Team learning is an approach to model learning in centralised single-agent systems. In team learning, a single learner applies a single-agent learning algorithm to learn a set of behaviours for a group of agents. This technique is a simple approach to cooperative multi-agent learning, but the fact that a single agent learns the behaviours of all agents, makes the state space very large. For example, a team of \( n \) agents in an environment of \( m \) states and \( p \) actions per state has a state space of as many as \( m^n \) states and an action space as many as \( p^n \). Hoen et al. [2005], and Panait and Luke [2005] classified team learning into three branches: homogeneous learning, heterogeneous learning, and hybrid learning.

#### Homogeneous Team Learning

In homogeneous learning, a single learner learns a single behaviour that is adapted from all other team members. This is possible if all agents have the same behaviour. The state space of homogeneous learners is normally small because of the assumption that all agents have the same behaviour [Panait and Luke, 2005].
Homogeneous team learning can be applied to problems where single agents perform well [Balch, 1998], such as problems with a large number of agents (swarm). In addition, problems that do not require decomposition are suitable for homogeneous team learning [Bonabeau et al., 1999], such as the basic hunter prey problem.

In the hunter prey domain, a single hunter agent can learn the behaviour of a team of hunters. Figure 2.5 shows a hunter agent that learns the behaviour of three hunters.

**Heterogeneous Team Learning**

A heterogeneous learner learns a specific behaviour for each member of the team. Normally the state space of the learner is large and grows in complexity with the increase of the number agents.

Heterogeneous team learning can be applied to problems where task specialisation is required, such as robotic soccer [Balch, 1998] or to inherently decomposable problems [Bonabeau et al., 1999], such as distributed decision and air-traffic control systems.

![Figure 2.7](image)

**Figure 2.7:** The hunter prey problem with a heterogeneous team of hunters. A hunter agent learns the blocking behaviour and the chasing behaviour.

Figure 2.7 shows a hunter agent that models two cooperative hunting roles: blocker (B) and chaser (C). The chaser’s role is to pursue the prey wherever it goes, while the blocker’s role is to block escape routes of the prey.
Hybrid Team Learning

Hybrid team learning is a combination of heterogeneous and homogeneous approaches [Hara and Nagao, 1999, Luke et al., 1998]. In this type of learning, each team of agents is divided into disjoint groups. Agents of each group adapt the same behaviour. Figure 2.8 shows a team of hunter agents composed of two disjoint groups. The first group is the group of chaser hunters ($C_1$ and $C_2$) that follow the chaser agent, while the second is the group of blocker hunters ($B_3$ and $B_4$) that follow the blocker agent.

2.7.3 Concurrent Learning

In concurrent learning, each agent is a learner and learns concurrently with the other learners in the multi-agent system. The main goal here is to project the joint state space into $m$ individual spaces. The risk of such an approach is that the existence of multiple learners at the same time may affect the stationary attribute of the agents’ environment [Hoen et al., 2005].

Hoen et al. [2005], and Panait and Luke [2005] argued that there are three main trends in concurrent learning research. First, research on the credit assignment problem that studies the distribution of reward signal among team mates. Second,
research that studies cooperative adaptation of agents. Third, research on how team members can model each other.

Credit Assignment

The credit assignment problem arises when trying to extend single-agent reinforcement learning to the multi-agent systems [Hara and Nagao, 1999, Luke et al., 1998, Rahaie and Beigy, 2009]. The credit assignment problem is the problem of determining how to distribute the reward of some work to a group of agents. The simplest solution for the credit assignment problem is the global reward solution, in which the group’s reward is divided equally among each of the learners. The main disadvantage of the global reward approach is that the agents do not have feedback personalised to their actions, which makes this approach unsuitable for difficult problems [Wolpert and Tumer, 2001]. For example, in the hunter prey domain, a reward signal can be distributed among hunter agents equally regardless of the individual behaviours of hunter agents.

A second solution to the credit assignment problem is the local reward approach where the reward signal is divided among the learners of a group based on the contribution of each one of them in the joint action. One disadvantage of this approach is that the learners of the same group might develop greedy competitive behaviours [Hara and Nagao, 1999, Luke et al., 1998, Rahaie and Beigy, 2009]. However, local rewards do not necessarily lead to better results than global rewards [Balch, 1999]. In fact, the results depend on the nature and the size of the problem. For example, in the hunter prey domain, the reward signal can be distributed among hunter agents based on the distance of each hunter agent from the current position of the prey agent.
Chapter 2. Agent Based Model

Non-stationary Learning Environments

The environments of concurrent learners are normally inherently non-stationary because the learners are free to change their behaviours as they also learn and adapt, which would make the outcome of learning unpredictable [Hara and Nagao, 1999, Luke et al., 1998, Weinberg and Rosenschein, 2004]. In game theory, the concept of Nash equilibrium [Nash, 1951] is used to analyse the interaction of non-cooperative agents. Nash equilibrium defines a stable game, in which each agent has a strategy and knows the strategies of the other agents, and no agent would benefit from changing its strategy unless the other agents also change their strategies. Many of the research studies on concurrent multi-agent learning focus on proving that their solutions converge to Nash equilibrium. However, cooperative concurrent multi-agent learning research focuses on proving that solutions have reached global optimal collaboration or converge to sub-optimal Nash equilibrium [Hara and Nagao, 1999, Luke et al., 1998].

Team Member Modelling

The success of an agent’s actions in concurrent multi-agent environments depends on its knowledge of the environment and on the actions of the other team members. This makes it necessary for each member in a team to model the behaviour of the other members.

A simple approach to team mate modelling in large multi-agent systems is to assume that all agents are identical and follow the same behaviour [Haynes and Sen, 1996, 1998]. For example, the hunter prey problem can be modelled so the hunter agents can follow the same hunting policy without any need for coordination of actions or observing the actions of each other. In small and medium-sized multi-agents systems, team mates can predict each other’s behaviours based on observation. During the execution of a game, each agent observes the actions of the other agents and builds
2.7.4 Learning and Communication

Communication between agents is a key element for the success of multi-agent learning. Agents in MASs communicate for many reasons, such as coordination of actions, developing an accurate model of their environment, exchanging subtask solutions and increasing their performance [Hoen et al., 2005].

There are mainly two approaches of communication:

- **Direct Communication**: In this approach, an external method of communication is employed by agents to share information.

- **Indirect Communication**: Agents communicate implicitly through modifications of the world’s environment.

In the hunter prey domain, examples of direct communication include signalling and message passing [Hoen et al., 2005], and examples of indirect communication include trial pheromones and foot mark methods which were inspired from the behaviour of real ants [Iima et al., 2010].

2.8 Famous Agent-based Models

The terms agent-based model (ABM), MAS, and multi-agent simulation are usually used interchangeably to define a system that is composed of multiple, sophisticated, autonomous agents that interact within an environment to solve problems that are beyond an individual capacities or require sharing of agents’ knowledge. This

2.8.1 Belief-Desire-Intention Agent-based Model

Belief-Desire-Intention (BDI) agent-based model is a well-known model that was proposed to model intelligent agents’ behaviour. It mainly uses belief, desire and intention concepts to model some target problems in the agents’ environment. An advantage of this model is that it provides a clear separation between plan selection and the execution of current plans [Rao and Georgeff, 1991].

BDI has two main advantages [De Silva et al., 2009, Rao and Georgeff, 1991]. First, the BDI modelling process is near to the way humans think about problem planning and solution procedures. Second, the direct result of BDI is a clear problem decomposition to sub-problems that eventually guide the design agents’ subsystems.

However, a complete and precise description of the problem space is needed in order to apply BDI model. Actually, if the design guidelines of BDI are applied in the design of a distributed learning model, many design issues need to be handled from the beginning. Also, communication procedures should be specified between different agents in all distributed system levels. This can be assisted by the use of process modelling software [Hector et al., 2008].

2.8.2 Swarm Intelligent Agent-based Model

Swarm intelligence is the collective behaviour of similar autonomous natural or artificial entities. The concept is based on the collective behaviour showed by biological creatures from the same type that show collective behaviour when acting independently to achieve a certain task. A main advantage of this model is that it is a
decentralised model that allows agents to interact with each other and their environment without a centralised control structure. This model was first proposed by Beni and Wang [1993] as an agent-based model in cellular robotic systems.

Particle swarm optimisation is a computational problem optimisation method that repetitively tries to enhance a nominated solution based on a measure of quality [Kennedy and Eberhart, 1995]. Particles here stand for a community of possible solutions that keep circulating in the search space based on a mathematical formula that is based on the position of the particle and its velocity.

2.8.3 Hierarchical Multi-agent Learning

One of the most common organisational structures of multi-agent systems is the hierarchical structure. Sometimes the learning tasks can be decomposed into a hierarchy of sub-tasks (Figure. 2.9.a), other times the learning agents play different roles to accomplish a task (Figure. 2.9.b).

![Figure 2.9: Hierarchical multi-agent systems.](image)

2.8.4 Layered Multi-agent Learning

In layered multi-agent learning, each individual task is decomposed into a hierarchy of learning layers (subtasks) such that each layer uses the ones beneath it [Stone, 2000, Whiteson and Stone, 2003]. Stone and Veloso [2000a] proposed a layered multi-agent learning that does not include hierarchical task decomposition. In this
approach, each layer is learned by applying a suitable machine learning algorithm then the output is transferred to the upper layer. This means that each learned layer is frozen before learning the next higher layer. Whiteson and Stone [2003] argued that there are situations where layered learning works better, if the lower layers are allowed to keep learning concurrently with the training of subsequent layers.

### 2.8.5 Distributed Multi-agent Systems

Many software applications have distributed architecture, such as air-traffic control or robotic systems [Chaib-Draa et al., 1992]. Sommerville [2011] argued that modern software has decomposable structure. In fact, the main step of architecture design of modern software is decomposing a software system into loosely coupled subsystems.

![Figure 2.10: Distributed multi-agent system composed of three subsystems.](image)

The distributed multi-agent system approach is a suitable one to represent distributed software systems. In this approach, the multi-agent system is decomposed into subsystems in which agents coexist and interact. The agents in distributed multi-agent systems can vary between stationary and mobile agents. In this approach, several issues should be addressed, such as the rules that govern migration...
of agents between subsystems, cooperation at system and subsystem levels, the specialisation of agents, and the distribution of agents.

Figure 2.10 illustrates an example of a distributed multi-agent system composed of three subsystems. The figure shows that an agent migrated from subsystem\(_1\) to subsystem\(_2\) and that the communication among the subsystems is not centralised. However, communication between subsystems, communication between agents and the agent model can differ from one example to another.

### 2.8.6 Distributed Hunter Prey Problem

This section introduces a distributed version of the classical hunter prey problem in order to demonstrate how independent agents can cooperatively learn the policy of a problem of a large state space. The main argument for this design is that the reduction of a state space \(S\) into \(n\) state spaces, \(S \rightarrow \{S_0, S_1, \ldots S_{n-1}, S_n\}\), accelerates convergence to the optimal solutions [Asadi and Huber, 2004, Barto and Mahadevan, 2003, Daoui et al., 2010, Hansen and Feng, 2000].

Figure 2.11 shows a version of the classical \(8 \times 8\) hunter prey problem that has been decomposed into 4 distributed grids of size \(4 \times 4\). We can clearly see from the figure that the bottom left sub-grid has a single prey and no hunter agents and that bottom right sub-grid has a single hunter and no prey agents.

There is a set of rules that governs the learning and interaction of agents in the distributed hunter prey problem. First, an agent, either a hunter or a prey, can be a mobile or a stationary agent. Second, there should be a mechanism to redistribute hunter agents that have finished hunting or that are idle to sub-grids where hunting is still in progress. Third, there are two possible levels of cooperation among hunter agents: cooperation at grid level and cooperation at sub-grid level. Fourth, the distribution of hunter agents to each sub-grid should be relative to the number of prey agents in each sub-grid.
2.8.7 Advantages of Agents-based Models

There are five main advantages for the ABM approach over traditional system modelling methods [Crooks et al., 2008]. First, an agent-based modelling approach is a tool for modelling systems based on their natural structures. For example, in the hunter prey problem, a hunter is an agent with attributes that describe it and model its interaction. Second, the design process of an agent-based model is flexible and efficient in terms of space and time (like layered and hierarchical models). Third, an agent-based approach is suitable for modelling the characteristics of complex systems such as emergent behaviours, self organisation and chaos. Fourth, an agent-based model approach is more near to reality than traditional methods because it is a natural method for describing real world objects. Fifth, an agent-based model approach can be seen as a natural extension of object oriented development since the concept of an agent is similar to the concept of an object. It is even more naturally an extension of component-based software engineering.
2.8.8 Limitations of Agent-based Models

The implementation of the ABM approach to real life applications is curtailed by some limitations [Crooks et al., 2008]. First, the usability of an ABM model is limited by the purposes of its design which is a common problem to all modelling techniques. Second, the modelling process of complex systems that involve human beings is usually very difficult to quantify and justify because of the irrational and complex psychological behaviours exhibited in complex systems. Third, the modelling process of an ABM involves description of agent attributes, agent behaviours, the environment and the agents’ interaction with the environment and among themselves. All these elements should be tested through multiple runs with varying conditions. However, the simulation process becomes computationally difficult with the increase of the number of variables. Finally, agent-based models are subject to surprising behaviours that cannot be found in the real world because of their sensitivity to the initial conditions and the variations in the interaction rules.

2.9 Learning and Teaching

In MASs, learning and teaching are connected; an agent can be a teacher and/or a learner [Shoham and Leyton-Brown, 2008]. An agent is a learner if it implements a learning algorithm or uses the knowledge of other agents. An agent is a teacher if it shares its knowledge with other agents or influences their behaviour. Figure 2.12 shows a game of two players. Each player follows a policy that it learned using its learning algorithm. In this game, there are many ways that a player can learn from and/or teach from the other player depending on the type of the game. For example, if the game is a cooperative game, the players can learn from each other, share knowledge or coordinate their actions. If the game is a competitive game, the players can enhance their policies by learning each other’s behaviour.
In the hunter prey problem with multiple concurrent learners, a hunter agent can play the role of a learner or an expert [Ahmadabadi and Asadpour, 2002, Araabi et al., 2007]. A hunter agent can be a learner or an expert depending on the level of its knowledge. For example, a learner that has spent 100 episodes of learning has more experience than a learner that has spent less time learning. This scenario provides a chance for the hunter agent with less knowledge to take advantage of the knowledge of the expert hunter.
Chapter 3

Markov Decision Process

The theory of Markov Decision Process (MDP) studies discrete time, stochastic, dynamic problems. In practice, many real-life problems in fields such as robotics, automated control, economics, and manufacturing can be modelled as MDPs. However, classical methods for solving large MDPs suffer from Bellman’s curse of dimensionality [Bellman, 1957], and lack of model information. In this chapter, we begin by discussing briefly the basic concepts and the main classifications of MDPs. We discuss, then, the main application fields, and the main limitations of the MDP model. The chapter focuses on decomposition techniques for solving large MDPs, particularly the decomposition techniques used in Reinforcement learning. This chapter gives an overview of collaborative games and MDPs. The zero sum and general sum games are briefly discussed in this chapter because the current thesis focuses on cooperative RL for independent learners (collaborative games). The classical hunter prey-problem (Section 2.1) and the distributed version of it (Section 2.8.6, and Section 3.9.1) will be used in this chapter to illustrate MDP-related concepts.
3.1 Markov Decision Processes

A Markov Decision Process (MDP) is a framework for representing sequential decision problems that allows a decision maker to choose from several possible next states at each decision stage [Mausam and Weld, 2004]. MDP is widely used to represent dynamic control problems, where the parameters of the MDP need to be learned through interaction with the environment [Abbeel and Ng, 2005].

An MDP model is a 4-tuple that contains:

1. A set of possible states, $S = \{s_0, s_1...s_{n-1}\}$.

2. A set of possible actions $A = \{a_0, a_1...a_{m-1}\}$.

3. A reward model $R : S \times A \times S \rightarrow \mathbb{R}$.

4. A transition model $T : S \times A \times S \rightarrow [0, 1]$.

If $(s_i \times a_j \times s_k) \rightarrow 1 \in T$, then it is possible to transition from state $s_i$ to state $s_k$ by performing action $a_j$. The reward received for completing this action is $R(s_i, a_j)$.

Figure 3.1 illustrates a graphical view of the MDP model, where at each time step the system receives an action that might change its state and produces a reward accordingly.

Figure 3.1: A view of an MDP based on Russell and Norvig [2003, Chapter 17].
The main aim of MDPs is to find a policy $\pi$ that can be followed to reach a specific goal (a terminal state). A policy is a mapping between the state set and the action set $\pi : S \rightarrow A$. An optimal policy $\pi^*$ always chooses the action that maximises a specific utility function of the current state. MDP is commonly used to represent optimisation problems that are solved by RL [Sutton and Barto, 1998].

The policy can be calculated following a dynamic programming approach [Singh et al., 1998]. In this approach, the optimal value function is formulated as the solution of Bellman optimality equations:

$$
\forall s \in S, V(s) = \max_{a \in A} \left( \sum_{s' \in S} P^a(s, s')[R^a(s, s') + \gamma V(s')] \right),
$$

where $P^a(s, s')$ is the probability of moving from state $s$ to $s'$ after applying action $a$, $R^a(s, s')$ is the reward received after moving from state $s$ to $s'$ after applying action $a$, and $\gamma$ is the discount factor, $0 < \gamma < 1$, which determines the importance of the future rewards. The policy can be normally determined from $V(s)$ as follows:

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

The hunter prey problem can be modelled as an MDP problem. The following is a possible way to model the hunter prey problem in Figure 3.2 as an MDP problem:

- The hunter prey problem involves two agents: a hunter agent and a prey agent.
- The hunter agent and the prey agent move simultaneously (one step a time).
- The prey agent policy is a fixed stationary policy that maximises its distance from the hunter agent.
- Each cell in the grid represents a state that a hunter agent can be in: $S = (\text{grid}[0][0], \text{grid}[0][1], ..., \text{grid}[4][4])$. The $x$ and $y$ coordinates are respectively the horizontal and vertical components of any position in the grid.
• The hunter agent can move to any of the adjacent cells to its current position:
  \[ A = \text{(move up, move down, move left, move right)}. \]

• The reward that each hunter agent receives is defined as:
  \[
  R_{\text{Hunter}}(s, a, s') = \begin{cases} 
  +100.0 & \text{if it moves to the same cell as the prey} \\
  0 & \text{otherwise}
  \end{cases}
  \]

• The transition model for the hunter agents is:
  \[
  T_{\text{Hunter}}(s, a, s') = \begin{cases} 
  1 & \text{if } s' \text{ is adjacent to } s \text{ in the direction of } a \text{ and is} \\
  & \text{a valid position for the hunter} \\
  0 & \text{otherwise}
  \end{cases}
  \]

![Figure 3.2: An example of hunter prey problem on a 5 × 5 grid.](image)

In the above transition function, there is no need to define the collision case because it is not possible with the existence of one hunter in the above example.

### 3.2 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) is a type of MDP in which the agent cannot determine with full reliability its current state. Instead, it
makes an observation based on the action and resulting state [Kaelbling et al., 1998].

A POMDP model is a tuple $\langle S, A, T, R, O, \Omega \rangle$, where:

- $S, A, T$ and $R$ define an MDP.
- $\Omega$ is a finite set of observations, and finally
- $O : S \times A \rightarrow \prod(\Omega)$ is the observation function that gives for each action and next state, a probability distribution over possible observations $L$, where $L \subseteq \Omega$.

In POMDP, $O(s', a, o)$ is the probability that the agent will make observation $o \in \Omega$ given that it made an action $a \in A$ and moved to next state $s' \in S$. The goal of the agent is still to find a policy $\pi : S \rightarrow A$ that maximises a specific utility function of its current state.

The hunter prey problem can be modelled in a more realistic approach based on two modifications [Senkul and Polat, 2002] (Figure 3.3). First, the grid is no longer homogeneous since it is composed of wall and empty cells. The wall cells block the movement of all agents, but the agents can see through them. Second, the hunter and prey agents can only see the agents inside their visual depth $d$. The area that an agent with visual depth $d$ can see consists of the square with size $(2d+1)$ around the agent [Erus and Polat, 2007].

### 3.3 Stochastic Games

The MDP model is not sufficient to represent the decision process of multiple agents [Partalas et al., 2008]. Hence, a more general model is needed such as stochastic game (SG). SG is a framework for representing multi-level decision problems for multiple agents. A SG is a 5-tuple $=\langle Ag, S, \{A_i\}_{i \in Ag}, R, T \rangle$, where $Ag$ is a set of $n$ agents $\{Ag_0, Ag_1, ..., Ag_{n-1}\}$, $S$ is the set of possible states, $A_i$ is the set of possible...
Chapter 3. Markov Decision Process

Figure 3.3: The visual area of hunter agent $H_3$ with depth 1.

actions of agent $i \in Ag$ ($A$ is the joint action space: $\times_{i \in Ag} A_i$, $R$ is the discounted sum of rewards $\gamma(R_0 + R_1 + \ldots + R_{n-1})$ of all agents where $R_i$ is the reward function for the $i$th agent: $S \times A \to R$ and finally $T$ is the transition model: $S \times A \times S \to [0, 1]$ [Bowling and Veloso, 2000].

SGs can be viewed as a multi-agent extension of MDPs. However, a transition to a new state and the received rewards in a SG depend on the joint action space $A : \times_{i \in Ag}$, while a transition and its associated reward in an MDP depend on a single action.

Learners seek to learn a stochastic policy during the stochastic game, which is a mapping of states to a probability distribution of related actions $P_s : S \to PD(A_i)$. This is the opposite of the deterministic policy of an MDP that specifies a unique action at each state. Unlike MDPs, deterministic policies in SGs can often be exploited by the other agents. A solution to this problem is to implement stochastic policies or mixed policies for stochastic games.

Stochastic games can be classified based on reward function as follows [Lauer and Riedmiller, 2004]:

- Collaborative, or team, games are cooperative games with a single reward
function for \( n \) agents, \( R_1 = R_2 = \ldots = R_{n-1} = R_n \), where \( n \) is the number of agents.

- Zero sum games or fully competitive games are two players games, in which the reward that is received by one player is the additive complement of the other’s reward \( R_1 = -R_2 \).

- General sum games can be viewed as mixed games of collaborative games and competitive games.

The hunter prey problem can be modelled as a collaborative game for the hunter agents’ side. The following is a possible way to model the hunter prey problem in Figure 3.2 as a collaborative game:

- The hunter agents in the grid compose a team of \( n \) hunter agents: \( \text{Ag} = \{H_1, H_2, H_3\} \).

- Each cell in the grid represents a state that a hunter agent can be in: \( S_{\text{Hunter}} = \{\text{grid}[0][0], \text{grid}[0][1], \ldots, \text{grid}[4][4]\} \). The state space \( s \) is any triplet of three states \( \in S_H \). No two hunter agents are allowed to be in the same state at the same time.

- The hunter agents have the same possible actions:
  \( A_i \in \{1, 2, 3\} = \{\text{move up, move down, move left , move right}\} \),
  where \( i \) is the hunter id. The joint action space is \( A_1 \times A_2 \times A_3 \).

- The reward that each hunter agent receives is defined as:
  \[
  R_{\text{Hunter}}(s, a, s') = \begin{cases} 
  +100.0 & \text{if it captures the prey} \\
  0 & \text{otherwise}
  \end{cases}
  \]

- The transition model for the hunter agents is: \( S \times (A_1 \times A_2 \times A_3) \times S \rightarrow [0, 1] \)
3.4 Finite-horizon MDP vs Infinite-horizon MDP

Finite-horizon MDPs are a class of MDPs where the performance (maximising discounted reward) is required to be maximised over a finite time horizon [White, 1995]. This model is normally implemented when the agent life span is known in advance. In general, any discrete-time Markov decision process that has a finite number of decision stages is considered a finite-horizon process. Equation 3.3 shows that the agent maximises its expected reward for the next $N$ steps.

In contrast to finite-horizon MDPs, infinite-horizon MDPs are suitable for continuous tasks where the behaviour continues indefinitely, and the reward can be received at any time. The time horizon of infinite-horizon MDPs tends toward infinity. Equation 3.4 shows that the agent maximises its expected reward without specifying an endpoint ($N$ approaches infinity).

$$V_\pi(S) = \left\{ \sum_{t=1}^{N} r(s_t, a_t) \right\}$$

(3.3)

$$V_\pi(S) = \lim_{N \to \infty} E_{\pi, S} \left\{ \sum_{t=1}^{N} r(s_t, a_t) \right\}$$

(3.4)

Finite-horizon, infinite-horizon RL problems, and their relationships to discounted rewards were discussed in Section 4.5. These concepts are originated from the MDP domain.

3.5 Bellman’s Equation

Bellman’s equation (Dynamic programming equation) [Bellman, 1957] is used in sequential decision problems to calculate the maximum expected sum of discounted rewards for each state (equation 3.5). In MDP, the Bellman’s equation is used to calculate the utility of each state under a given policy. An MDP of $n$ states has a
system of $n$ simultaneous Bellman’s equations, one for each state [Prashanth and Bhatnagar, 2011].

$$U^\pi(s) = R(s) + \gamma \sum_{s'} T(s, a, s') \pi U(s')$$  \hspace{1cm} (3.5)

The Bellman’s equation (equation 3.5) is a linear equation, while the Bellman’s equation for calculating optimal policies ($Bellman$ optimality equation) is a non-linear equation. The max function associated with the Bellman optimality equation makes the function a non-linear one. The max function formulates the process of taking the action with the highest possible return.

$$U^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U^*(s')$$  \hspace{1cm} (3.6)

Many forms of this function are used in RL to formulate the utility functions of various RL algorithms.

### 3.6 Application of Markov Decision Processes

In artificial intelligence, the available data change (model vs. samples) problems can be formalised in the framework of Markov decision processes. These problems include two types: sequential decision-making under uncertainty, and reinforcement learning [Sigaud et al., 2010]. Sequential decision problems are composed of several decision problems arranged in a sequence. For each decision problem of the sequence, an agent needs to choose an action taking into account its effects on the solution of the other decision problems. However, the effects of the agent’s actions are not available immediately for the agent (uncertainty). Finite Markov decision processes are important to the theory of reinforcement learning since RL problems are usually formalised as MDPs. Finite Markov decision processes are the ones with finite state and action spaces.
3.7 Limitations of Markov Decision Processes

There are many issues that limit the use of the MDP model, and complicate its implementation. These limitations are related to time, and space requirements of the implementation of the MDP model. The limitations include: the curse of dimensionality, the memory requirement, and the stationary assumption of the problem model.

3.7.1 Curse of dimensionality

Finding an optimal policy for an MDP is polynomial in the number of states and actions, but the number of states grows exponentially with the number of state variables, which makes it computationally difficult to solve large MDPs [Ermon et al., 2013]. This problem is known as the curse of dimensionality, which is a significant challenge in MDPs. Consider the following example. A hunter prey problem with a grid size of $n \times n$, $m$ hunter agents, and 5 possible actions. Based on these numbers, the number of cells of the grid is $x = n \times n$, the number of states is $x \times x - 1 \ldots \times x - m$, and the number of actions is $5^m$.

3.7.2 Memory Requirement

The modelling of an MDP requires the definition of the state, and action spaces. Each state-action pair needs a transition probability matrix, and a reward function. The required memory increases with the increase of the state, and action spaces. Furthermore, the type of the goal, a stationary or a moving goal, determines the amount of memory required for the solution space. For a moving goal, the agent needs to calculate a policy for each possible state of the goal which requires the position of the prey to be included in the state. Thus, the state space becomes so large. For example, in Figure 3.4, the grid is composed of 16 positions (cell[0][0],...,cell[3][3])
where each cell is a possible position of the prey agent. Thus, the hunter agent needs to develop a policy for each possible position of the prey agent.

![Figure 3.4: An example of hunter prey problem on a 4 × 4 grid.](image)

### 3.7.3 Stationary Assumption

In MDP framework, the transition probabilities and rewards are assumed to be fixed with time, but in fact, they may not stay the same over time. Consider the following two examples of the hunter prey problem. First, the reward that the hunter agent receives after each transition changes with the change of the position of the prey agent. Second, the transition probability of a hunter agent to a new position is 1, if the position is empty or occupied by a prey agent, and 0 if it is occupied by another hunter agent. These non-stationary attributes can be addressed by including time to the state space or using finite horizon MDP model. Another solution is to include the positions of the prey and the other hunters in the state definition.

Multi-agent systems are generally non-stationary environments because many agents affect the environment at the same time [Mahmud and Ramamoorthy, 2013]. This makes the learning policies of the agents non-stationary policies. In addition, the behaviour of an agent is affected by the behaviours of the other agents in the same environment [Hu and Wellman, 2003].
3.8 Large Markov Decision Processes

Reinforcement learning techniques for solving large MDPs suffer from the curse of dimensionality [Bellman, 1957]. This is because the state space of large MDPs grows exponentially with the number of state variables [Asadi and Huber, 2004, Barto and Mahadevan, 2003, Daoui et al., 2010, Hansen and Feng, 2000]. In addition, RL requires knowledge of transition probabilities of the dynamic system from one state to the next, which is not realistic in practice for large systems.

One general solution for tackling large MDPs is decomposition in which large MDPs are decomposed into simpler components. The solutions of these components form the basis of a solution for the original problem. A main reason that encourages the use of this approach is that many real life applications have a decomposable structure such as robotic soccer and traffic control. In addition, the factored approach reduces the complexity of the computation of the solutions of large MDPs.

3.9 Decomposition Approaches for Markov Decision Process

We discuss briefly in this section two decomposition approaches for MDPs: factored, and hierarchical approaches. The latter approach is related to RL problem.

3.9.1 Factored Approaches

A factored MDP (FMDP) is a concept that was first proposed by Boutilier et al. [1995]. A FMDP is an MDP with a state space $S$ that can be specified as a cross-product of a set of state variables ($S = S_0 \times S_1 \times \ldots \times S_{n-1}$). The idea of factored state space is related to the concepts of state abstraction and aggregation. Such a structure is a more natural way to represent state space instead of an $S \times S$
probability matrix per action [Daoui et al., 2010]. In FMDPs, $T_a$ denotes the state transition model for an action $a$. Such a transition model is represented as a Dynamic Bayesian Network (DBN) which is a two-layer directed acyclic graph $G_a$ with the nodes $S = (S_0, S_1, ..., S_{n-1}, S'_0, S'_1, ..., S'_{n-1})$. In $G_a$, the parents of $S'_i$ are noted as the $\text{parents}_{a}(S'_i)$ where these parents are assumed to be a subset of the state space $\text{parents}_{a}(S'_i) \subset S$. This means that there are no synchronous arcs from $S_i$ to $S'_j$.

The reward function $R$ can be decomposed additively $\gamma_1 R_0 + \gamma_2 R_1, ..., + \gamma_{n-1} R_{n-1}$ and the differences of the decomposition do not depend on the state variables [Wu et al., 2014].

There are three main advantages for the factored approach [Mausam and Kolobov, 2012]. First, by implementing the factored approach, similar states can be grouped together which allows the use of the same policy to each group. Second, the value function can be expressed as a function of state variables such as $V(s) = C_1(s) + C_2(s) + ... + C_n(s)$. Third, large MDPs can be modelled using the factored approach as a pre-step for implementing the approximation techniques to them. Although, the factored approach reduces the complexity of the computation of the solutions of large MDPs, it makes large MDP harder to solve [Mausam and Kolobov, 2012].

To illustrate the FMDP framework, consider the distributed hunter prey problem described in Section 2.8.6. In this problem, a version of the classical 8×8 hunter prey problem was decomposed into 4 distributed grids of size 4×4. The state of the grid of such a problem is composed of two variables:

- $G$: The sub-grid number.
- $C$: The cell number inside the grid.

For example, in Figure 3.5, the vector $[G = 1, C = 1]$ represents the state of the hunter agent $H_1$ on the first cell of the top left sub-grid, and the vector $[G = 4, C = 2]$ represents the state of the hunter agent $H_2$ on the second cell of the bottom right sub-grid. An alternative representation of the hunter agents in Fig 3.5 is the agents’
vector in which the vector of each hunter agent is an entry to it $[[G = 1, C = 1], [G = 4, C = 2]]$. In the distributed hunter prey problem, there are special rules that regulate the hunting process and issues that are related to this process. For example, the hunter agents and the prey agents might migrate from one sub-grid to another in one of the following cases:

- The sub-grids are physically distributed across the network. The hunter agents pursue the prey agents from one sub-grid to another until they catch them (both type of agents can migrate from one sub-grid to another), or the hunter agents that caught the prey agents at their original sub-grids migrate to other sub-grids where help is needed.

- The whole grid is located at one environment (computer), but is decomposed into a number sub-grids for the purpose of accelerating the hunting process.

In the above two scenarios of the distributed hunter prey problem, distribution of the hunter agents among grids should be considered. Consider the following example. Hunter agents in some sub-grids might be pursuing prey agents, while the hunter agents in the other sub-grids are idle. Such a situation requires moving the
hunter agents that are idle to the sub-grids where the hunting is still in progress. For example, Figure 3.6 shows that the hunter agents on sub-grid A have finished hunting all the prey agents in sub-grid A, while the hunting is still active in sub-grids B, C and D. The logical action in such a scenario is to provide additional worker hunters for the parts of the grid that are still active.

![Figure 3.6: Distributed hunter prey problem composed of four sub-grids. The hunter agents in the top left sub-grid have finished hunting, while the hunters in the other sub-grids are still active.](image)

In the above example, similar policies can be used in each subgrid because the subgrids have the same size. The above example can be modelled in another way to show how the value function of the hunting task can be decomposed into multiple value functions. Let us assume that the hunting task is a composite task that is composed of two subtasks: blocker task and chaser task. Blocker hunters learn how to block some blocking cells in the corners of the grid while the chaser hunters chase the prey agents that escape outside their subgrids. The composite value function of the hunting task can be expressed as $V_{hunting}(s) = C_{Blocker}(s) + C_{Chaser}(s)$, where $C_{Blocker}(s)$ is the value function of the blocker hunter and $C_{Chaser}(s)$ is the value function of the chaser hunter.

Large weakly coupled FMDPs can be solved by decomposing the state space of
the original MDP into concurrent sub-MDPs (MDP = sub-MDP\(_0\) × sub-MDP\(_1\) × ...
\times sub-MDP\(_{n−1}\)) [Meuleau et al., 1998]. In this approach, the state variables are
assumed to be associated with a specific task and the number of resources allocated
to the individual tasks are regulated by global constraints. These constraints are
the cause of the weak coupling between the sub-MDPs.

Degris et al. [2006] proposed an algorithm to learn the structure of FMDP in RL
problems by the name Structured DYNA (SDYNA). This algorithm is a version of
DYNA algorithm [Sutton, 1991] based on approximation in the form of dynamic
Bayes networks and decision trees. SPITI, an instantiation of SDYNA, increment-
tally builds a decision-tree representation of state transitions. There are two main
disadvantages for SDYNA [Strehl et al., 2007]. First, it uses \(\epsilon\)-greedy action selection
technique, which gives equal weights to the non-greedy actions. Second, SDYNA has
no formal performance guarantees.

Boutilier et al. [1995] proposed an algorithm, called the Structured Policy Iteration
(SPI), that exploits the structure of FMDPs. In this technique, a FMDP is specified
by a set of Dynamic Bayesian Networks (DBNs) per action. SPI constructs DBNs
to extract variable independence and uses structured decision tree representation
of the conditional probabilities matrices to discover further variable independence.
Mainly, SPI generates partitions of the state space. The states of a partition should
have the same estimated value or best action. The authors of SPI mentioned three
main advantages of SPI. First, it computes an optimal solution. Second, it is a
good representation for uncertainty. Third, it can be used with different types of
approximation techniques.

Hansen and Feng [2000] proposed value iteration and policy iteration algorithms
for POMDPs that can be represented as FMDPs. These algorithms are based on
approximate linear and dynamic programming. The algorithms approximate the
value function using a linear combination of basis functions. Each basis function
depends on a small subset of the state variables. The experimental results showed
that a FMDP representation speeds up the dynamic programming in the best case, but it slows it down in the worst case.

Guestrin and Gordon [2002] proposed a distributed planning algorithm in hierarchical factored MDPs that solves large decision problems by decomposing them into subdecision problems (subsystems). The subsystems are organised in a tree structure. Any subsystem has two kinds of variables, internal variables and external variables. Internal variables are the variables that can be used by the value function of the subsystem, while the external variables cannot be used because their dynamics are unknown. Although the algorithm does not converge to the optimal solution, its output plan is equivalent to the output of a centralised decision system. This approach suffers from some limitations. First, although coordination and communication between agents are not centralised, they are restricted by the subsystem tree structure. Second, the definition of a subsystem as an MDP composed of internal and external variables only fits decision problems.

Boutilier et al. [1997] proposed an approach to goal decomposition for multi-feature MDPs. In this approach, the components of the utility function (features) are considered separately. In other words, each component has a utility function that is used to calculate an abstract policy. All the abstract policies are combined together to form an approximately optimal policy for a multi-feature MDP. A main disadvantage of this approach is that it uses prioritised merging to combine the abstract policies which is normally associated with a small error.

Furmston and Barber [2011] proposed a dynamic dual decomposition approach for finite horizon MDPs with stationary\footnote{A policy that is independent of time ($\forall t \in \{1, ..., H\}, \pi(s,a)_t = \pi(s,a)$).} policies. This approach uses Lagrange duality to decouple large MDPs into a sequence of tractable sub-problems. This process is an iterative one that uses dual decomposition and iteratively solves a set of non-stationary MDPs until convergence to a general optimal policy.
Chapter 3. Markov Decision Process

Solving very large MDPs might require sacrificing optimality for the purpose of finding a solution. Very large MDPs could be solved by constructing a hierarchical model and then solving its components. Barry et al. [2010] proposed an algorithm for solving large MDPs that uses state aggregation to construct a hierarchical structure and then solving it. This approach combines two aggregation techniques: aggregating together local states and assigning them the same sub-goal and aggregating together states that behave similarly.

Barry et al. [2011] proposed DetH* algorithm, an algorithm for solving long horizon MDPs, by constructing a temporal hierarchy of smaller shorter horizon MDPs from the original problem. The decomposition process of DetH* is based on the connectivity heuristic of the state space which is used to divide the state space into macro states. All transitions between the macro states are assumed to be deterministic, which makes mapping macro states to macro states much easier. After constructing a policy between the macro states, DetH* finds a policy for each macro state.

The current decomposition techniques do not provide support for learners’ migration from one sub-MDP to another. Instead they focus on decomposing the state space, and statically assign each sub-MDP to a learner. Chapter 6 presents a RL algorithm and a model that provide support for migration of learners.

3.9.2 Hierarchical Approaches

Hierarchical reinforcement learning (HRL) imposes a hierarchical structure on FMDPs. We can identify two common approaches for decomposition of distributed MDPs: static decomposition and dynamic decomposition.

Parr and Russell [1997] proposed a RL approach called HAMQ-learning that combines Q-learning algorithm with Hierarchical abstract machines (HAMs). This approach effectively reduces the size of the state space, since it limits the learning policies to a set of HAMs. However, state decomposition in this form is hard to
apply, since there is no guarantee that it will not affect the modularity of the design or produce HAMs that have large state space.

Dietterich [2000] has shown that MDPs can be decomposed into a hierarchy of smaller MDPs based on the nature of the problem and its flexibility to be decomposed into smaller sub-goals. This research also proposed a MAXQ procedure that decomposes the value function of an MDP into an additive combination of smaller value functions of the smaller MDPs. An important advantage of MAXQ decomposition is that it is a dynamic decomposition while the decomposition of HAMs is a static decomposition [Barto and Mahadevan, 2003].

The MAXQ procedure attempts to decrease the size of large problems into smaller problems, but it does not take into account the probabilistic prior knowledge of the agent about the problem space. This issue can be addressed by incorporating Bayesian reinforcement learning priors on models, value functions or policies [Cao and Ray, 2012]. Cao and Ray [2012] presented an approach that incorporates Bayesian priors in the MAXQ procedure. This approach extends MAXQ by incorporating priors on the primitive environment model and on goal pseudo-rewards. Priors in multi-goal reinforcement learning can be extracted from models or policies of previous learned goals. This approach is a static decomposition approach. In addition, the probabilistic priors should be given in advance in order to incorporate them in the learning process.

Sutton et al. [1999] proposed the concept of option which is a form of knowledge abstraction of MDP. An option can be viewed as a primitive task that is composed of three elements: a learning policy $\pi : S \times A$, where $S$ is the state set and $A$ is the action set; a termination condition $\beta : S^+ \rightarrow [0, 1]$; and an initial set of states $I \subseteq S$. An agent can perform an option if $s_t \subseteq I$, where $s_t$ is the current state of an agent. An agent chooses an option then follows its policy until the policy termination condition becomes valid. In this case the agent can select another option. A main disadvantage of this approach is that the options need to be determined in advance.
In addition, it is difficult to decompose MDPs using this approach because many decomposition elements need to be determined for each option.

Generally, multi-agent cooperation problems can be modelled based on the assumption that the states is a joint state of $n$ agents, where each agent $i$ has access to a partial view $s_i$ from the set of joint state $s = \{s_1, ..., s_{n-1}, s_n\}$. In the same manner, the joint action is modelled as $\{a_1, ..., a_{n-1}, a_n\}$, where each agent $i$ may only have access to partial view $s_i$. One simple approach to model multi-agent coordination is discussed in the survey study of Barto and Mahadevan [2003]. It shows that the concurrency model of joint state and action spaces can be extended to learn task-level coordination through replacing actions with options. However, this approach does not guarantee convergence to the optimal policies since learning low-level policies vary at the same time when learning high-level policies.

The study of Barto and Mahadevan [2003] discussed another MARL cooperation approach. This approach is a hyper approach that combines options [Sutton et al., 1999] and MAXQ decomposition [Dietterich, 2000] together. Simply, an option, $o = \langle I, \pi, \beta \rangle$, is extended to a multi-option $\overrightarrow{o} = \langle o_1, ..., o_n \rangle$, where $o_i$ is the option that is executed by agent $i$. A joint action value of a main task $p$, a state $s$ and a multi-option $\overrightarrow{o}$ is denoted as $Q(p, s, \overrightarrow{o})$. Then the MAXQ decomposition of the Q-function can be extended for the joint action-values.

Hengst [2002] proposed HEXQ, a hierarchical RL algorithm, that automatically decomposes and solves MDPs. Basically, it uses state variables to construct a hierarchy of sub-MDPs, where the maximum number of the hierarchy levels is limited to the number of state variables. Eventually, the results are interlinked small MDPs. As discussed in Hengst [2002], the main limitation of HEXQ is that it must discover nested sub-MDPs and find policies for their exits with probability of 1. This requires that the problem space must have state variables that change on a long time scale.
Tosic and Vilalta [2010] proposed a hierarchical, collaborative RL solution for the problem of agents’ coordination, precisely in dynamic agents’ coalition formation in large-scale problem. The researchers view is based on the argument that an agent has three RL learning levels, individual agent RL level, co-learning among small agents group and Meta learning at the system level. Basically, the learning process of this model is a cooperative process that takes the advantages of learning on three different levels. Furthermore, the model supports dynamic adaptation of coalition among agents based on continuous exploration and adaption of RL in the three layer architecture of the proposed model.

However, the proposed model of Tosic and Vilalta [2010] doesn’t specify any communication scheme among its three RL learning levels. Moreover, the model suffers from the absence of detailed algorithmic specifications on how RL can be implemented in this three layer learning architecture.

Ghavamzadeh et al. [2006] proposed a new Hierarchical Reinforcement (HRL) algorithm named Cooperative HRL that can be implemented to Semi Markov Decision Process (SMDP) model. Cooperative HRL requires the SMDP model to be decomposed into smaller homogeneous SMDPs. The subtasks of each agent are classified into cooperative subtasks and uncooperative subtasks. A cooperative subtask is a subtask that requires coordination of actions among the agents to be accomplished, while an uncooperative subtask is a subtask requiring one agent’s work. The basic idea here is that sharing of information among the cooperative agents is only at the cooperation levels.

The HRL algorithm of Ghavamzadeh et al. [2006] has three main advantages. First, it suits large state space problems. Second, it accelerates joint actions learning without confusion with low level details. Third, it needs less communication than other RL MASs. On the other hand, there are many disadvantages. First, learning optimal policy would not be accomplished if coordination is needed among non cooperative subtasks. Second, optimal policy would be degraded if communication
is costly and finally if agents store only local information, the result would be suboptimal policies.

Lin [1993] has proposed a form of multi-agent RL suitable for parallel executing RL agents. In this algorithm, the RL problem (MDP) is decomposed manually into a set of sub-problems that are learned by all Q-learning agents. The Q-learning agents are controlled by a central Q-learning agent, which learns $Q(s, i)$, where $i$ is the agent to select in state $s$. The selection process is a simple one: When the main Q-learner observes state $s$, each agent $i$ proposes an action $a_i$. Based on these propositions, the main agent selects the action with the highest Q-value to be executed.

Gunady et al. [2014] proposed a RL solution for the problem of territory division on Hide-and-Seek games. The territory division problem is the problem of dividing the search environment between cooperative seekers in order to reach optimal seeking performance. The researchers combined hierarchical RL approach with state aggregation in order to reduce the state space. In state aggregation, similar states are grouped together in two directions: topological aggregation and hiding aggregation. In topological aggregation, the states are divided into regions based on the distribution of obstacles. In hiding aggregation, hiding places are grouped together and treated as the target of aggregation action. A disadvantage of this algorithm is that it requires the model information of the environment to be known in advance.

Jardim et al. [2011] proposed a dynamic decomposition hierarchical RL method. This method is based on the idea that to reach the goal, the learner must pass through closely connected states (subgoals). The subgoals can be detected by intersecting several paths that lead to the goal while the agent is interacting with the environment. Temporal abstractions (options) can be then identified using the subgoals. A drawback of this method is that it requires multiple simulations to define the subgoals. In addition, this method is time consuming and can not be easily applied to large learning problems.
Cuáyahuítl et al. [2014] presented an approach for spoken dialogue policy optimisation that combines hierarchical control and function approximation. In this approach, strict hierarchical control is relaxed by giving human users more control to navigate more flexibly across the dialogue subtasks. In order to accomplish this, each reinforcement learning agent is extended in the hierarchy with a subtask transition function and a dynamic state space. A main drawback of this approach is that it requires human users to continuously monitor and compare each new policy with the previously learned policies in order to improve the agents’ performance. However, the experimental results based on simulations trained from Wizard-of-Oz data and experiments with human users show that the proposed hierarchical approach is promising.

Cai et al. [2013] proposed a combined HRL method for multi-robot cooperation in completely unknown environments. This method is a result of the integration of options with the MAXQ hierarchical reinforcement learning method. The MAXQ method is used to identify the problem hierarchy. The proposed method obtains all the required learning parameters through learning without any need for an explicit environment model. The cooperation strategy is then built based on the learned parameters. In this method, multiple simulations are required to build the problem hierarchy which is known to be a time consuming process.

### 3.10 Merging Markov Decision Processes

The decomposition approach can be viewed as a composition one where general solutions of the original system are composed of the solutions of the components. This approach is known in the literature as the merging approach in which the value function of the whole FMDP is viewed as a merging problem in which the optimal
value function satisfies the following Bellman equation [Singh et al., 1998]:

$$\forall s \in S, V(s) = \max_{a \in A} \sum_{s' \in S} \left\{ \prod_{i=1}^{n} P^{a_i}(s^i, s^{i'}) \left[ \sum_{i=1}^{n} R^{a_i}(s^i, s^{i'}) + \gamma V(s') \right] \right\}, \quad (3.7)$$

where \( n \) is the number of the sub MDPs, and \( i \) is the identity number of the sub MDP. We can see from the above equation that the discount factor \( \gamma \) is the same for all sub MDPs.

The merging problem is concerned with constructing a composite value function for the composite MDP (FMDP) giving the optimal value function for each of its sub MDPs. One approach to finding the optimal composite policy is by using the bounds of sub MDPs to compute the values of the composite states [Singh et al., 1998]. The resulted bounds are incrementally updated and approximated using a special type of value iteration function that allows selective removal of non-competitive actions. The advantages of this approach include: it allows the pruning of actions with upper bounds less than the best known lower bound, and it can be applied to dynamic merging problems.

In practice, applying Q-learning might cause the state space of composite MDP to grow exponentially with the number of MDPs. Ghavamzadeh and Mahadevan [2002] proposed a cooperative RL learning algorithm called MAPLE (MultiAgent Policy LEarning) that uses Q-learning and dynamic merging of MDPs proposed by Singh et al. [1998] to find solutions to the multi-agent problem from its components. In Singh et al. [1998], the dynamic merging problem is that of merging single agent tasks in which each task is modelled as an MDP, however the same concept is used by MAPLE to formulate the multi-agent problem as a multi-agent MDP. The experimental results showed that MAPLE learns faster than the standard Q-learning in multi-agent taxi problem.
Multiple-goal problems can be modelled as composite MDPs. Karlsson [1997] proposed a modular Q-learning approach to solve multiple-goal problems. In this approach, each subMDP of the composite MDP is modelled as a module that is responsible for finding an optimal policy for it. The policy of each module is merged into a composite policy. Each module $i$ applies the standard Q-learning update rule:

$$Q_i(s_i, a) \leftarrow (1 - \alpha)Q_i(s_i, a) + \alpha( r_i + \gamma \max_{a'} Q_i(s'_i, a'))$$  \hspace{1cm} (3.8)$$

where $i$ is the module subscript, $\alpha \in [0, 1]$ is the learning rate, and $\gamma \in [0, 1]$ is the discount factor. The calculated Q-values are then used to select a compromise action for execution. Karlsson [1997] uses the greatest mass method for the selection of compromise actions. This method finds the summation of the Q-values of each state-action for all of the modules, and then chooses the action with the maximum summation to be executed:

$$Q(s, a) = \sum_{i=1}^{n} Q_i(s_i, a)$$  \hspace{1cm} (3.9)$$

One limitation of this approach is that Q-learning is an off-line learning algorithm, which means that it assumes that all the future actions will be chosen optimally for the composite MDP. This is an invalid assumption when using the greatest mass selection method in which the future actions represent a compromise policy that does not guarantee convergence, and the max in equation 3.8 produces Q-values with positive bias. In order to overcome this problem, Sprague and Ballard [2003] proposed the use of SARSA (on-line learning) algorithm instead of Q-learning algorithm. In this approach, the agent takes actions in the environment, and each module $i$ applies SARSA algorithm:

$$Q_i(s_i, a) \leftarrow (1 - \alpha)Q_i(s_i, a) + \alpha( r_i + \gamma Q_i(s'_i, a'))$$  \hspace{1cm} (3.10)$$

The main motivation behind using SARSA instead of Q-learning is that the update
rule of SARSA (equation 3.10) does not produce Q-values with positive bias since the SARSA’s update rule depends on the actually observed actions in the next step. Although this approach does not suffer from the problem of positive bias, there are no convergence proofs for this approach.

Some multiple-goal problems can be modelled as large weakly coupled MDPs in which each goal is modelled as an MDP with an independent utility [Meuleau et al., 1998]. In this approach, the optimal policies of the components of the composite MDP can be collectively used to construct an approximate global solution for the composite MDP using heuristic techniques. For example, a stochastic resource allocation problem can be modelled as a weakly coupled MDP, where each task of the problem is modelled as an independent weakly coupled MDP by resource constraints. The utility function of performing a set of tasks is the summation of the rewards of performing each task independently (additive utility independent). A main limitation of this approach is that the storage and the speed of execution increase exponentially with the number of sub MDPs of the composite MDP [Sprague and Ballard, 2003].

### 3.11 Summary

Discrete time stochastic dynamic problems can be represented using the MDP model. This chapter provides some necessary background information about Markov Decision Process (MDP). An MDP is a framework for representing multi-level decision problems that allows a decision maker to choose from several possible next states at each decision stage. An MDP model is a 4-tuple that contains: a set of possible states, $S = \{s_0, s_1, \ldots, s_{n-1}\}$, a set of possible actions, $A = \{a_0, a_1, \ldots, a_{m-1}\}$, a reward model $S \times A \times S \rightarrow R$, and a transition model $S \times A \times S \rightarrow [0,1]$. MDPs are normally classified into fully observable MDPs, or partially observable MDPs. Both types can be modelled as a single agent game or as a stochastic game.
The Reinforcement learning techniques are used to solve MDPs, but these techniques suffer from Bellman’s curse of dimensionality, and lack of a problem model, when they are implemented to large MDPs. This chapter also examines decomposition approaches for tackling large MDPs in which large MDPs are decomposed into simpler components. The solutions of these components form a solution for the original problem. The two main decomposition approaches that are discussed in the current chapter are the factored, and the hierarchical approaches. The decomposition approach can be viewed as a composition one where general solutions of the original system are composed of the solutions of the components of the system. This approach is known in the literature as the merging approach in which the value function of the whole FMDP is viewed as a merging problem.

The next chapter, Chapter 5, will discuss Multi-agent RL (MARL) approaches that are used to model RL for multi-agent systems. It will be argued that the environment’s dynamics of MASs are determined by more than one agent, which requires MARL algorithms to model such environments. A large part of the next chapter will discuss the benefits and challenges that arise in MARL, and the recent advances in MARL techniques. The decomposition approaches, particularly the hierarchical approach, discussed in the present chapter are important to understand the research work in Chapter 6 and Chapter 7. Chapter 6 will present a distributed hierarchical learning framework (DHLM), and an intelligent distributed Q-learning algorithm (IDQL) for solving large MDPs with distributed structure. Chapter 7 will present an algorithm for cooperative policy construction for independent learners, that is based on two roles: learner and tutor.
Chapter 4

Reinforcement Learning

Machine learning algorithms are usually categorised into three main taxonomies: supervised learning, unsupervised learning and reinforcement learning (RL). RL is a machine learning technique that is based on trial and error. In RL, an agent explores its surrounding environment by applying actions and receiving rewards for these actions. The main target of the agent is to maximise its utility function that is based on rewards. The purpose of this chapter is to provide basic background information about RL as a first step to understand the problem of RL in multi-agent systems and the recent advances in RL. The hunter-prey problem described in section 2.1 will be used in this chapter to illustrate RL-related concepts.

4.1 Introduction

Reinforcement Learning (RL) is a learning paradigm that is based on trial and error [Sutton and Barto, 1998, Wiering and van Otterlo, 2012]. A RL agent learns by applying actions, and receiving rewards for these actions. Unlike most machine learning paradigms, a RL agent tries the actions in order to discover the ones that produce the highest reward.
RL is suitable for interactive learning in which the agents learn from interacting with their environment. In interactive problems, it is hard and impractical to provide the agents with all examples of possible behaviours to which they should react. Thus, supervised learning, which is learning from examples provided by external sources, is not sufficient for learning from interaction [Barto and Dietterich, 2004].

RL has similarities to both supervised and unsupervised learning. It is similar to supervised learning in the case of correct responses since in both methods the learner receives positive feedback about its action. RL is similar to unsupervised learning because the learner in both methods learns from its environment without being provided supervision or training data.

In RL, a learner interacts with its environment in the following way: the learner perceives the state of the environment and selects an action related to the state using its decision making function (policy). The action is then performed, and the agent receives a positive or a negative reward for its action. At the end, information about the reward of the state-action pair is used to update the agent policy (Figure 4.1).

It is important to note that the action does not always change the state of the environment. The goal of the agent is to maximise, at each time-step \( k \), the expected
discounted return

\[ R_k = E(\sum_{j=0}^{\infty} \gamma^j r_{k+j+1}), \text{where } \gamma \in [0, 1) \text{ is the discount rate} \]

and the expectation is taken over the probabilistic state transitions under the actions chosen by the policy of the agent. (4.1)

The goal of the agent is to maximise its sum of rewards \((R_k)\) while it is interacting with the environment.

### 4.2 Reinforcement Learning Model

The RL model is a 4-tuple that is related to the fact that the problem model of RL is normally formulated as an MDP. A typical RL problem consists of [Kober and Peters, 2012]:

- A set of states, \(S = \{s_0, s_1, ..., s_{n-1}\}\).
- A set of actions, \(A = \{a_0, a_1, ..., a_{m-1}\}\).
- A reward model, \(R: S \times A \rightarrow \mathbb{R}\).
- A transition model, \(T: S \times A \times S \rightarrow [0, 1]\).

The behaviour of a reinforcement learner at any instant is determined by a policy that determines the optimal action to be chosen at each state. A policy \(\pi\) is a mapping of environmental states to actions \(\pi: S \rightarrow A\). A RL agent seeks to develop a policy which maximises the sum of its rewards, \(R = r_0 + r_1 + ... + r_n\), for a problem that has a terminal state \(s'\), or a termination condition \(c'\).

The reward signals that an agent receives are related to the learning goal. A reward function is a mapping of state-action pairs to numerical rewards \(S \times A \rightarrow R\).
The value function is a utility function that expresses the benefit of choosing a state over the future. A value of a given state is an estimation of the summation of rewards starting form the given state.

### 4.3 Types of Policy

The policy that the agent follows has a vital role in the agent success to accomplish its task. A policy that is unrelated to time and does not require the agent to possess memory is known as stationary policy, while non-stationary policy requires the agent to possess memory. Policies can be classified based on certainty of actions into two types: deterministic and stochastic policies. A deterministic policy specifies a unique action for each state, while a stochastic policy chooses an action $a$ from a distribution $P^\pi_s$ with a probability $P^\pi_s(a)$ [Sutton and Barto, 1998]. This thesis concentrates on deterministic stationary policy in cooperative independent learners in Discrete-time MDPs. A discrete-time MDP has a finite number of decision stages where its performance is usually required to be maximised over a finite horizon (Section 3.4).

### 4.4 Episodic vs. Continuous Tasks

RL works for episodic, and continuous (non-episodic) tasks. Episodic tasks are a form of problems in which the behaviour can be divided into separate episodes, in which the reward may be received at any step during an episode depending on how utility is computed. An episode normally ends when the learner reaches a terminal state, or fulfils a termination condition. For example, in the hunter-prey problem, an episode is the actions that a hunter performs starting from a particular cell in a grid world until it catches the prey. In contrast, continuous tasks are a form of
problems where the behaviour continues indefinitely and the reward can be received at any time; for example controller balancing a pendulum.

In episodic tasks, the purpose of RL at each episode is to maximise the summation of rewards, which means it is not necessary to discount the reward, so the discount factor is preferred to be $\gamma = 1$. In non-episodic tasks, the discount factor is normally chosen between $0 < \gamma < 1$, since the reward will accumulate indeterminately. Off-line and on-line RL methods can be employed to learn episodic tasks, while on-line methods must be used for non-episodic tasks [Provost, 2008]. In on-line RL, an agent updates its Q-table during trials while, in off-line RL, the agent can not update its Q-table while interacting with the environment but during the simulation process of learning.

### 4.5 Discounted Reward

Determining the influence of the future rewards of the agent on its current behaviour is important for the agent learning to behave optimally. There are three models of optimal behaviour that determine how the rewards that the agent gains in the future affect its current behaviour [Kaelbling et al., 1996].

First, the **finite horizon** model where the performance is required to be maximised over a finite time horizon, which is suitable for episodic tasks. In this model, the agent is required to maximise its expected reward within a limited number of steps (finite horizon).

$$E(\sum_{t=0}^{n} r_t)$$  \hfill (4.2)

Equation 4.2 shows that the agent should maximise its expected reward for the next $n$ steps regardless of what will happen afterwards. The agent that follows this model of behaviour develops a non-stationary policy (a policy that changes with time).
Second, discounted infinite horizon model which is suitable for continuous tasks. In this model, the future rewards are discounted based on a discount factor \( 0 < \gamma < 1 \). A factor of \( \theta \) means that there is no importance for the future rewards, while a factor approaching 1 increases the weight of the future rewards.

\[
E(\sum_{t=0}^{\infty} \gamma^t r_t)
\]  

Equation 4.3 shows that the agent maximises its expected reward without specifying an endpoint.

Third, the average reward model which is a discounted framework that is more appropriate for continuous tasks than the second model. In this model, the agent takes actions that maximise its future average reward in the long run:

\[
\lim_{n \to \infty} E(\frac{1}{n} \sum_{t=0}^{n} \gamma^t r_t)
\]  

A main disadvantage of the last model is that it differentiates between high and low beneficial policies because it uses the average method.

### 4.6 Applications of Reinforcement Learning

RL can be used to solve different interactive problems. It serves as a theoretical tool for studying learning of agents in interactive problems and as a computational tool for constructing systems of independent agents that evolve with experience. Robotics, industrial manufacturing and combinatorial search problems are examples of these applications [Kaelbling et al., 1996].

Two of the main practical applications of RL are [Kaelbling et al., 1996]:

- Game playing: In two-player games, each player tries to maximise its reward against an optimal opponent (minimax game), while in RL, the player tries to
maximise its reward against a stationary environment. However, RL can work for a general class of games [Littman, 1994, Powell, 2000]. In RL, the player learns by playing the game and does not need to follow predetermined rules.

• Control problems. RL has been applied to various control problems such as robotics and manufacturing [Kober et al., 2013, Riedmiller et al., 2009]. RL agents can learn in a simulated environment and with time they will develop good controlling policies.

4.7 The Trade-off Between Exploration and Exploitation

Unlike supervised learning, a reinforcement learner must explicitly explore its environment to learn [Kaelbling et al., 1996]. The trade-off between exploration and exploitation is a main problem that arises in RL [Chen et al., 2009, Louie, 2013, Sutton and Barto, 1998]. The problem is that exploration and exploitation of actions are connected. In the first hand, a RL agent must exploit actions that have been tried and found to be highly rewarded in order to maximise its reward summation. In the second hand, in order to discover highly rewarded actions, an agent must explore new actions. Some of the best known selection policies to solve this problem are described in the next section.

4.8 Action Selection Policies

The most trivial action selection policy is the greedy policy, which selects the action with the highest estimated reward to be executed. This policy is an exploitative policy. But the goal of the action selection policies is to balance between exploration and exploitation. The following action selection policies attempt to do that [Sutton and Barto, 1998]:
• \( \epsilon \)-greedy: an action selection policy that selects most of the time the action with the highest estimated reward. There is a small probability \( \epsilon \) that an action would be selected at random.

• \( \epsilon \)-soft: an action selection policy that selects the best action with probability \( 1 - \epsilon \) and selects a random action the rest of the time.

• Softmax: Softmax assigns a weight to each action according to its action-value estimate. A random action is selected based on its weight, which means that the worst actions are unlikely to be chosen. Normally, Boltzmann distribution [Rawlik et al., 2013] is used in Softmax. Given state \( s \), an agent tries out action \( a \) with a probability

\[
p_s(a_i) = \frac{e^{Q(s,a_i) T}}{\sum_{b=1}^{m-1} e^{Q(s,b) T}}, \quad \text{where } T \text{ is the temperature and}
\]

\[
Q(s,a_i) \text{ is the expected return when starting from state } s \text{ taking the action } a_i.
\]

In equation 4.5, the temperature \( T \) controls the required degree of exploration. Assuming that all Q-values are different, if \( T \) is high, the agent will choose a random action, but if \( T \) is low, the agent will tend to select the action with the highest weight.

4.9 Temporal Credit Assignment

The temporal credit assignment problem is a main problem in RL. This problem is concerned with how an action taken at a particular time step participates in the final outcome [Agogino and Tumer, 2004, Kaelbling et al., 1996]. A direct solution for this problem is to assign rewards to the actions based on the final outcome. However,
this approach requires a great deal of memory, and it is very hard to determine what is the end. An alternative approach is temporal difference learning, which is a prediction technique that updates the value of the state based on the difference of successive predictions of the value function.

4.10 Estimation of Value Function

Value functions are used in RL to calculate the values of the state-action pairs. These functions estimate the utility of a particular action at a given state. Normally, two notations are used in the value functions: \( V^\pi(s) \), which is the value of a state \( s \) under a policy \( \pi \), and \( Q^\pi(s,a) \), which is the value of implementing an action \( a \) in state \( s \) following policy \( \pi \) afterwards.

In RL, the estimated value functions are used for choosing the action that is expected to provide the maximum future total reward. The following section discusses temporal difference methods that are used to learn the value of policies. We start by discussing two important theoretical foundations of TD methods: Dynamic programming (DP), and Monte Carlo (MC) Methods.

4.10.1 Dynamic Programming Methods

Dynamic programming (DP) methods [Chib, 2013] are algorithms that require a complete model of the environment to learn optimal policies. DP algorithms are theoretically important to understand RL, but they have a limited utility in RL for two reasons [Sutton and Barto, 1998]. First, DP algorithms incur a large computational cost. Second, they require a complete model of the environment such as MDP.

Equation 4.6 shows a state-value function \( V^\pi \) for an arbitrary policy, \( \pi \). The expected returned (r) values are assumed to be completely provided by the environment.
Equation 4.7 is a reduced form of equation 4.6 [Sutton and Barto, 1998].

\[
\forall s \in S, \quad V^\pi(s) \leftarrow E_\pi \left\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | s_t = s \right\} \tag{4.6}
\]

\[
\forall s \in S, \quad V^\pi(s_t) \leftarrow E_\pi \left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s \right\} \tag{4.7}
\]

DP methods use an estimate of (4.7) because \(V^t(s_{t+1})\) is used instead of \(V^\pi(s_{t+1})\), which is unknown.

Dynamic programming equation (Bellman equation [Bellman, 1957]) is a necessary condition for the optimality associated with DP methods. In Bellman equation, the value function is expressed as in equation 4.8. The value function gives the long-term value of state \(s\) (equation 4.8).

\[
V(s)^* = \max_a \{ R(s, a) + \gamma \sum_{s'} P(s, s', a) V(s')^* \}, \text{ where } \gamma \text{ is the discount factor.} \tag{4.8}
\]

The basic idea here is to iteratively update the values of the states using Bellman equation (Value Iteration). The Bellman equation is normally arbitrarily initialised and its equation \(V(s)^*\) has a unique solution and value iteration normally converges to it.

### 4.10.2 Monte Carlo Methods

Monte Carlo (MC ) methods are techniques for estimating value functions and discovering optimal policies. Unlike DP methods, MC methods do not require a model of the environment to learn. MC methods learn from experience, which can be on-line or simulated experiences. MC methods can be used to solve the episodic RL problem. A simple form of Monte Carlo methods is the every-visit Monte Carlo method for non-stationary environment. In this form, if a non goal state is visited at time \(t\), the method waits until the reward following the visit is known, then it updates its estimate \(V(s_t)\) based on the returned reward (equation 4.9).
Chapter 4. Reinforcement Learning

\[ V(s_t) \leftarrow V(s_t) + \alpha \left[ R_t - V(s_t) \right] \], \text{ where } \alpha \text{ is the learning rate.} \quad (4.9)

This method is called every-visit method because the value of a state can be updated \( n \) times along the same trajectory (equation 4.10).

\[ V(s_t) \leftarrow V(s_t) + \alpha \left[ r_t + r_{t+1} + r_{t+2} + \ldots + r_n - V(s_t) \right] \text{ where } n \text{ is number of update times.} \quad (4.10)\]

MC methods are considered to be prediction techniques because they use an estimate of equation 4.10. In equation 4.11, the expected return value is not known, thus a sample return is used instead of the real one.

\[ V(s_t) \leftarrow E_\pi \{ R_t \} \quad (4.11)\]

### 4.10.3 Temporal Difference Learning

Temporal difference (TD) learning is a prediction technique that is used in RL to compute optimal policies based on experience. The name TD learning comes from the fact that the update to the value function takes the difference of successive predictions of the value function [Maei and Sutton, 2010]. TD learning combines the principles of Monte Carlo and dynamic programming (DP) [Sutton and Barto, 1998, Uther, 2010].

In TD learning, it is not enough to know the returned reward of visiting a state to update its value function as most of Monte Carlo methods require. In TD(0), the simplest known TD method, if a non goal-state is visited at time \( t \), then TD(0) updates its estimate \( V(s_t) \) based on what happens in the next time step \( t + 1 \).

\[ V(s_t) \leftarrow V(s_t) + \alpha \left[ r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right], \]

where \( 0 \leq \gamma \leq 1 \) is the discount factor. \quad (4.12)
TD target is an estimate because it samples the expected return and uses the current estimate $V^t$ instead of $V^\pi$ in equation 4.12.

**Types of Temporal Difference Methods**

TD methods are classified into two types [Rodrigues et al., 2008, Sutton and Barto, 1998]:

- On-Policy TD methods are used to learn the value of the policy being carried out by the learner (e.g., SARSA algorithm).
- Off-Policy TD methods learn the value of the policy independently of the actions of the agents (e.g., Q-learning algorithm).

### 4.11 Q-learning Algorithm

Q-learning is one of the best known RL algorithms that provides solutions for MDPs. This algorithm uses temporal differences to find mappings from state-action pairs to values. These values are known as Q-values, and are calculated using a utility function, called the Q-function, that returns the expected utility of taking a given action in a given state and following a fixed policy after that [Watkins, 1989, Watkins and Dayan, 1992]. The fact that Q-learning does not require a model of the environment is one of its strengths.

The problem model of the Q-learning algorithm is an MDP model that consists of an agent, a set of states $S$, and a set of actions $A_i$ for each state $s_i \in S$ [Shi et al., 2013, Szepesvári, 2010, Wang et al., 2007].

An agent that applies Q-learning needs a number of learning episodes to find an optimal solution. An episode is a learning period that starts from a selected state and ends when a goal state is reached. During an episode, the agent chooses an
action \( a \) from the set of actions \( A \) of its current state \( s \) based on its selection policy. The learner then perceives the new state of the environment \( s' \), and receives a reward \( R(s, a) \) based on the previously implemented action. The agent then updates its Q-table based on equation 4.13. This procedure repeats until the agent reaches the goal state, which marks the end of the episode.

\[
Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha [R(s, a) + \gamma \max_{a' \in A} Q(s', a')]
\]

Where \( R(s, a) \) is the reward of performing action \( a \) in the current state \( s \), \( a' \) is the action performed in the next state \( s' \), \( \alpha \in [0, 1] \) is the learning rate, and \( \gamma \in [0, 1] \) is the discount factor. SARSA is considered as an on-policy learning algorithm because it learns the Q-values associated with taking the policy it follows itself (equation 4.14).

\[
Q(s, a) \leftarrow Q(s, a) + \alpha [R(s, a) + \gamma Q(s', a') - Q(s, a)]
\]

Where \( R(s, a) \) is the reward of performing action \( a \) in the current state \( s \), \( a' \) is the action performed in the next state \( s' \), \( \alpha \in [0, 1] \) is the learning rate, and \( \gamma \in [0, 1] \) is the discount factor. SARSA is similar to Q-learning, but its Q-function depends on the actually performed action, instead of the value maximising action. SARSA (State-Action-Reward-State-Action) is a RL algorithm that was proposed by Rummery and Niranjan [1994]. SARSA is similar to Q-learning, but its Q-function depends on the actually performed action, instead of the value maximising action. The update equation of SARSA is:
The learning process of SARSA takes place in episodes as Q-learning. The executed policy starts random and gradually becomes more and more greedy.

Wang et al. [2013] proposed backward Q-learning, an algorithm that combines SARSA and Q-learning. In this algorithm, the agent stores the $s, a, s'$ and $r$ for each transition in the memory $M^i$ and updates its Q-function (Equation 4.15) until the end of an episode.

\[
Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[ R(s_i, a_i) + \gamma \max_a Q(s'_i, a_i) - Q(s_i, a_i) \right]
\] (4.15)

After reaching the goal state, the agent backward updates the Q-function from the start state to the goal state based on the information stored in $M^i$ using equation 4.15. The second update affects the selection policy of the agent.

A main disadvantage of backward Q-learning is that it consumes more computational time than SARSA or Q-learning because of the extra computations required to backward update the Q-function in the end of each episode.

### 4.13 Multi-agent Reinforcement Learning

Single agent RL techniques are usually designed to solve stationary environments. In multi-agent framework, several agents affect the environment, and the actions carried out by agents no longer depend just on the environment but also depend on what the other agents are doing [Fang et al., 2014].

In addition, applying a single-agent reinforcement learning approach to large multi-agent systems is inefficient because the state-space and/or the action space of these systems are normally so large. Single agent approach is unsuitable for distributed problems such as air-traffic systems [Agogino and Tumer, 2005].
In multi-agent framework, Q-learning can be applied in a straightforward fashion to each agent in a multi-agent system. This can be accomplished by adding a subscript to identify agents in the Q-function:

\[
Q_i(s, a_i) \leftarrow (1 - \alpha)Q_i(s, a_i) + \alpha[R_i(s, a_i) + \gamma \max_{a_i \in A_i} Q_i(s', a_i)] \quad (4.16)
\]

Where \(R_i(s, a_i)\) is the reward that agent \(i\) receives for performing action \(a_i\) in the current state \(s\), \(a'_i\) is the action performed by agent \(i\) in the next state \(s'\), \(\alpha \in [0, 1]\) is the learning rate, and \(\gamma \in [0, 1]\) is the discount factor.

However, there are two limitations for the above multi-agent Q-learning version. First, it assumes that each agent selects its actions independently from the other agents. Second, the use of the same maxQ function of a single agent version of Q-learning algorithm is invalid to be used for the \(V\) function [Hu and Wellman, 2003].

In recent years, many Multi-agent RL (MARL) approaches were proposed to model RL for multi-agent systems. MARL techniques will be further discussed in Chapter 3 and Chapter 5.

### 4.14 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) imposes a hierarchical structure on complex problems. HRL algorithms employ decomposition techniques to identify the hierarchical structure of big problems [Mahajan, 2014, Rasmussen and Eliasmith, 2014]. Most of the research in RL either discovers the decompositions automatically [Dietterich, 2000], or designs the decomposition by hand [Lin, 1993, Parr and Russell, 1997]. Many HRL algorithms have decomposition functions that decompose the original problem into \(n\) sub-problems (equation 4.17).

\[
\text{Problem} \rightarrow F(\text{sub-problem}_0, \text{sub-problem}_1, ..., \text{sub-problem}_n) \quad (4.17)
\]
If some sub-problems are dependent on each other, they can be solved sequentially, otherwise they can be solved in parallel. The value function of a RL problem can be decomposed into a combination of smaller value functions of the smaller MDPs. For example, Dietterich [2000] decomposes the value function of a RL problem into an additive combination of smaller value functions of the smaller MDPs. HRL techniques will be further discussed in Chapter 3.

4.15 Model-based Learning Methods

This section provides background information about RL model classifications. RL can be classified into two types based on the need of a learning model [Atkeson and Santamaria, 1997, Deisenroth and Rasmussen, 2011, Shteingart and Loewenstein, 2014]:

- Model-free (Direct RL) methods: Learn a controller without explicitly requiring a model of the controller (e.g., Q-learning algorithm). These methods can be used when there is a small number of actions per task, or when the policy can be represented with a small number of the task’s variables.

- Model-based methods: learn a controller and its model simultaneously (e.g., Dyna and prioritised sweeping algorithms). These algorithms can be used with large tasks with a large number of actions. A main advantage of these methods is that they need less training time to achieve good performance compared to direct RL methods. A main disadvantage of these methods is that they require memory to store their models.

4.15.1 Dyna Architecture

Dyna is an AI architecture that combines RL and execution-time planning into a single model [Hwang et al., 2013, Sutton, 1990, Sutton et al., 2008]. In Dyna (Figure
4.2, the agent applies RL at execution time to a real problem (real experience) and also applies RL hypothetically for planning (simulated experience) [Sutton, 1991].

![Diagram of Reinforcement Learning](image)

**Figure.** 4.2: The problem formulation used in Dyna based on Sutton [1990].

The Dyna algorithm requires more computations than the computations of Q-learning per time, because of the additional updates performed for the execution time planning [Kaelbling et al., 1996].

### 4.15.2 Prioritised Sweeping

RL planning in Dyna can be much more efficient if simulated experience is focused on specific state-action pairs rather than continuing to update random state-action pairs when a goal has just been reached or when the agent is stuck in a deadlock [Kaelbling et al., 1996]. Consider the following example of the hunter-prey. At the beginning of the second episode, the state-action pair leading directly into the prey has the only positive value, while the values of all other pairs are still zero. This means that the agent would most likely move from one zero-valued state to another which make backing up these transitions pointless.

Prioritised sweeping was proposed to solve the problem of Dyna [Moore and Atkeson, 1993, Van Seijen and Sutton, 2013]. It is similar to Dyna, but has two differences.
First, the updates are not chosen randomly. Second, the values are associated with states rather than state-action pairs.

In prioritised sweeping, each state has a priority initially set to zero, and each state remembers all the states that have a non-zero transition probability to it. Prioritised sweeping updates \( k \) states with the highest priority at each learning episode [Sequeira et al., 2014].

### 4.16 The Actor-critic Architecture

The actor-critic architecture is a TD framework that has a distinct memory structure to represent the policy independently of the value function. In this architecture, the actor is the policy structure that is used to select actions, and the critic is the estimated value function that is used to criticise the actions of the actor [Pilarski et al., 2011, Raicevic, 2006]. The learning here is an on-policy learning because the critic learns the policy that is being followed by the actor (Figure 4.3).

![Figure 4.3: The Actor-critic architecture based on Raicevic [2006].](image)

The critic calculates a TD error to evaluate the new state \((s_{t+1})\) after each action selection \((a_t)\) done by the actor (equation 4.18). A positive TD error indicates that
the probability of selecting action \( (a_t) \) should be increased, while a negative TD error indicates that the probability of selecting action \( (a_t) \) should be decreased.

\[
\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t),
\]

where \( \gamma \) is the discount factor and \( V \) is the value function. (4.18)

A positive TD error suggests that the tendency to select the recently selected action should be increased for the future, while a negative TD error suggests the tendency to select the action should be reduced.

### 4.17 Performance Evaluation

There exist many RL algorithms that vary in different aspects and produce different or similar policies. The quality of the learned policies of the reinforcement learners are normally evaluated using three methods [Kaelbling et al., 1996].

- **Convergence to optimality.** Many researchers proved that their RL algorithms converged to an optimal solution (like SARSA and Q-learning). However, the speed of convergence to an optimal behaviour is important. For example, in the hunter-prey problem, a hunter agent that quickly catches the prey agent 99% of the time is more preferable than the one that guarantees optimality but needs a longer learning time.

- **Speed of convergence to near optimality.** Measuring the speed of convergence to optimality is impractical because optimality is normally an asymptotic result. Therefore, measuring the speed of convergence to near optimality is a more appropriate measure. However, this measure has no proof that near optimality is sufficient. In addition, some algorithms might be punished while trying to
converge as fast as possible to a near optimal solution, which means that a slower more rewarded result might be better.

- Regret. The difference in performance between that of a tested agent and an agent with an optimal behaviour from the beginning is known as regret. This method is considered better than the above two methods because it compares the performance of the learner from the beginning to the end.

### 4.18 Summary

This chapter provides some necessary background information to understand Reinforcement Learning (RL). RL is a learning paradigm that is based on trial and error. A reinforcement learner learns by applying actions, and receiving rewards for these actions. Unlike most machine learning paradigms, the reinforcement learner tries the actions in order to discover the ones that produce the highest reward. The problem model of a RL problem is normally modelled as an MDP where the reinforcement learner seeks to develop a policy which maximises the sum of its discounted rewards \( R = \gamma(r_0 + r_1 + \ldots + r_n) \).

The chapter also presented the finite horizon discounted model that is suitable for solving episodic tasks, and the discounted infinite horizon model that is applicable for continuous tasks. The chapter introduced different action selection policies the aim to balance between exploration and exploitation of actions such as \( \epsilon \)-greedy, \( \epsilon \)-soft, and Softmax.

The chapter discussed the value functions used in RL to calculate the values of the state-action pairs: Monte Carlo (MC), dynamic programming (DP), and Temporal difference (TD). The last of these is the main technique used in RL for estimating of the value function. TD learning is a prediction technique, that combines the principles of Monte Carlo (MC) and dynamic programming (DP), and is used in RL to compute optimal policies based on experience. In TD learning, the update of the
value function takes the difference of successive predictions of the value function. Q-learning and SARSA are famous TD learning algorithms.

The chapter discussed two types of RL based on the need of a learning model. First, model-free learning methods that learn a controller without explicitly requiring a model of the controller (e.g., Q-learning algorithm). Second, model-based methods that learn a controller and its model simultaneously (e.g., Dyna and prioritised sweeping algorithms). These algorithms can be used with large tasks with a large number of actions.

The most common method for evaluating the performance of the model-free and the model-based RL techniques is the convergence to optimality method, but the best known method is the regret method. The regret method finds the difference in performance between that of a tested agent and an agent with an optimal behaviour from the beginning of the simulation.

The remainder of this thesis will concentrate on RL in multi-agent systems which is the main topic of the current thesis. The next chapter will present the problem model of RL (The Markov Decision Process) with a special concentration on the decomposition approaches used to solve large MDPs by dividing them into simpler components.
Chapter 5

Multi-agent Reinforcement Learning

In recent years, many Multi-agent RL (MARL) approaches have been proposed to model RL for multi-agent systems. The goal of MARL is to enable multiple agents to learn suitable behaviours in a dynamic learning environment using RL. The main difference between single agent RL and MARL is that the environment’s dynamics are determined by more than one agent in MARL systems, while a single reinforcement learner determines the environment’s dynamics in single agent systems. This difference is the source of many advantages and challenges that are specific to MARL systems. This chapter discusses the benefits and challenges that arise in MARL, and some recent advances in MARL techniques.

5.1 Introduction

Applying RL directly to large multi-agent systems is impractical because of the size of the state space of these problems and the coordination required between agents to accomplish their tasks [Boutilier, 1996, Buşoniu et al., 2010]. In addition, several problems arise when extending RL to multi-agent systems. Consider the following
example. In multi-agent framework, Q-learning can be applied in a straightforward fashion to each agent in the multi-agent system. This can be accomplished by adding a subscript to identify agents in the Q-function:

$$Q_i(s, a_i) \leftarrow (1 - \alpha)Q_i(s, a_i) + \alpha[R_i(s, a_i) + \gamma \max_{a_i \in A_i} Q_i(s', a'_i)],$$

(5.1)

Where $R_i(s, a_i)$ is the reward that agent $i$ receives for performing action $a_i$ in the current state $s$, $a'_i$ is the action performed by agent $i$ in the next state $s'$, $\alpha \in [0, 1]$ is the learning rate, and $\gamma \in [0, 1]$ is the discount factor. The above version suffers from three drawbacks according to the research of Hu and Wellman [2003]. First, it assumes that each agent selects its actions independently from the other agents, which is an invalid assumption. Second, the change of the environmental state depends on the agent’s direct actions, which is an assumption that neglects the fact that the other agents are affecting the environment at the same time. Third, it is invalid to use the $\max_{a_i \in A_i} Q_i(s, a_i)$ function of a single agent Q-learning algorithm in the Q-function because one of the other agents may have a state-action pair with a higher Q-value.

In the next two sections, the benefits and the challenges of MARL will be discussed.

5.2 Benefits of Multi-agent Reinforcement Learning

MARL approach has several advantages over a single-agent RL approach. The source of the advantages of MARL is mainly because of the multiplicity of agents. The advantages include: taking advantage of parallel computation, sharing of knowledge between agents, robustness and suitability for distributed learning.
5.2.1 Parallel Computation

Parallel computing can speed up the learning process of MARL algorithms when the agents exploit the decentralised structure of the task [Busoniu et al., 2008]. Cuayáhuítl and Dethlefs [2012] proposed an approach for learning to coordinate verbal and non-verbal behaviours in interactive robots. In this approach, a hierarchy of multi-agent reinforcement learners execute verbal and non-verbal actions in parallel. The research studies of Kaya and Arslan [2001], and Kraemer and Banerjee [2013] investigated the application of parallel and distributed systems to MARL. In this approach, reinforcement learners in different parts of the distributed system execute in parallel.

5.2.2 Sharing of Knowledge

The existence of multiple agents in the same multi-agent system is a chance for information exchange. In MARL, different types of information can be shared: sharing of sensory data, sharing of episodes, and sharing of learned policies [Tan, 1993]. Sharing of sensory data from another agent is beneficial, if the information is relevant and sufficient for learning (like a scouting hunter that sends scouting information for the other hunters). Sharing of episodes means sharing of the Q-values after a number of episodes, while sharing of policies takes place at the end of the learning process. In general, sharing of information accelerates the learning process, if it is used efficiently.

5.2.3 Robustness

MARL is inherently robust because when one or more agents fail, the remaining agents can perform their tasks. Reinforcement learners can be designed to react dynamically to unfavourable situations. For example, reinforcement learners that are scattered in a distributed system can be warned when a host system is being
shut down, so they will dispatch and continue to work in another host environment [Busoniu et al., 2008].

5.2.4 Distributed Learning

The single-agent RL approach is unsuitable for learning distributed problems such as air-traffic systems [Agogino and Tumer, 2005]. Distributed problems have a distributed structure that requires multiple agents to be distributed. MARL has a distributed nature that makes it suitable for distributed learning. The main requirement for modelling distributed problems is that distributed RL learners have to form policies that maximise a global reward for the entire distributed problem [Agogino and Tumer, 2005].

5.3 Challenges in Multi-agent Reinforcement Learning

The size of the state space of large multi-agent systems makes applying RL directly to multi-agent systems difficult and impractical [Boutilier, 1996, Buşoniu et al., 2010], hence, many MARL methods were proposed to address this problem. However, there are two types of challenges in MARL that complicate the learning process: challenges inherited from single agent RL, and new challenges specific to MARL. The inherited challenges include the trade-off between exploration and exploitation, and the curse of dimensionality, while the new MARL challenges include the non-stationarity of the learning problems, the need for coordination between agents, and the difficulty of specifying a good goal.

5.3.1 The Trade-off Between Exploration and Exploitation

The exploration and exploitation trade-off that was discussed in Section 4.7 is more complex in MARL than single agent RL [Busoniu et al., 2008, Chen et al., 2009,
Sutton and Barto, 1998]. In MARL, more problems arise because of the presence of multiple agents in the same environment. Exploration in MARL has dual purposes. First, obtaining information about the environment. Second, gathering information about the other agents. Intensive exploration can easily destabilise the learning dynamics of the other agents, which might make the learning process more difficult for the exploring agent [Busoniu et al., 2008].

5.3.2 Structural Credit Assignment Problem

The structural credit assignment problem is concerned with how to determine the contribution of each agent to the task [Agogino and Tumer, 2004, Yu et al., 2003]. The collective goal of the agents in a multi-agent systems is to maximise a group utility function, which is a function of the joint action of all agents in the system. Instead of directly maximising the group function, each agent tries to maximise its own local utility function. This problem is considered as one of the main problem in MARL algorithms.

For example, robotic soccer is a well known example that exhibits the structural assignment problem. In robotic soccer, the RL algorithm is required to evaluate a specific soccer-agent’s role in achieving a particular goal in the soccer game.

5.3.3 Curse of Dimensionality

RL suffers from the curse of dimensionality, which is the exponential growth of the state-action space with the number of state and action variables [Asadi and Huber, 2004, Daoui et al., 2010]. This growth leads to an exponential growth in the computational cost. In MARL, the increase of the number of agents is another element of complexity. Each agent adds its own variables to the joint state-action space, which makes the space grow exponentially [Busoniu et al., 2008].
5.3.4 Non-stationary Environment

Single agent RL techniques are usually designed to solve stationary environments. In a multi-agent framework, several agents affect the environment, and the actions carried out by agents no longer depend just on the environment but also on what the other agents are doing [Hu and Wellman, 2003].

5.3.5 Coordination of Actions

Coordination of actions is typically required in cooperative MARL because the effect of an agent’s action on the environment depends on the actions carried out by the other agents. Therefore, mutually consistent actions should be performed by the agents in order to achieve the intended effect. In MARL, coordination is normally reduced to consistently breaking up ties between equally good actions or strategies [Busoniu et al., 2008].

5.3.6 Specifying The Goal

When MARL is applied to fully cooperative stochastic games, the agents can jointly maximise their shared return. However, in other cases, the return of each agent depends on the other agents’ returns. So the return of each agent cannot be maximised independently without considering the other agents’ returns. In general, specifying goals in MARL systems is a hard process [Busoniu et al., 2008].

5.4 Recent Advances in Multi-agent Reinforcement Learning

In this section, we will discuss seven research streams in MARL: Section 5.4.1 introduces combinational RL algorithms, Section 5.4.2 discusses some of the best known
swarm RL algorithms, Section 5.4.3 presents some cooperative MARL studies, Section 5.4.4 presents some research studies about the implementation of RL in stochastic games, Section 5.4.5 discusses some recent research work about reducing the joint action space in stochastic cooperative games, Section 5.4.6 discusses two approaches for implementing the teacher-learner model in RL, and Section 5.4.7 presents some recent studies about reward modelling in RL.

5.4.1 Combinational Reinforcement Learning

A combinational RL algorithm is a type of RL algorithm that combines more than one RL algorithms to accelerate the learning process by taking advantage of the strength points of each algorithm (Figure 5.1).

![Figure 5.1: The aggregation architecture based on Jiang and Kamel [2006].](image)

Aggregated Multiple Reinforcement Learning System (AMRLS) that was proposed by Jiang and Kamel [2006] is an example of combinational algorithms. This algorithm aggregates Actor-Critic (AC), Q(\(\lambda\))-learning and SARSA(\(\lambda\)). AMRLS consists of two levels: learning and aggregation levels. The reinforcement learners can follow one of four execution modes while they are learning: synchronous, asynchronous, parallel or serial execution modes. Each learner selects an action for each state it visits, then sends these actions to the aggregation level. In the aggregation level, the
received actions for each state are dynamically aggregated using Majority Voting (WMV) or Weighted Borda Count (WBC) aggregation functions.

The experimental results in the mountain car problem suggest that AMRLS needs less training time and performs better than single agent RL. However, AMRLS was not compared to the best RL algorithms and the size of the experiment was too small.

Wiering and van Hasselt [2008] is another research that studied the use of multiple RL algorithms in single agent. This research, proposed four ensemble methods that combine Q-learning, SARSA, AC, QV-learning and AC learning automaton in a single learner. Each ensemble method was named after the aggregate decision function that it uses (Majority Voting (MV), Rank Voting (RV), Boltzman Multiplication (BM) and Boltzman Addition (BA)). In these algorithms, a learner combines actions or action probabilities of different RL algorithms. The experimental results in five instances of the maze problem of varying complexity show that the BM and MV ensembles outperform the BA, RV and single-agent RL in terms of total learning performance. The BA and RV ensembles do not outperform single-agent RL. A main disadvantage of the ensembles methods is that they require more computations than the single-agent RL approach.

Partalas et al. [2006] studied multiple-classifier systems from a RL perspective. The researchers proposed a method for pruning an ensemble of heterogeneous classifiers based on RL. In this method, an agent explores the state space of the problem and considers future cumulative rewards to learn a policy for selecting classifiers. This is accomplished using the voting method that selects the best policy of choosing classifiers from the agent’s multiple RL algorithms. A major problem with the proposed method is that the state space of the agent grows exponentially with the number of states (classifiers) which increases the complexity of the learning problem.
5.4.2 Swarm Reinforcement Learning

The swarm agent based model can be used to model the learning process of cooperative independent learners. The researchers, Hitoshi Iima and Yasuaki Kuroe, have conducted three studies that investigated the implementation of swarm multi-agent framework with RL [Iima and Kuroe, 2006, 2007, 2008].

Iima and Kuroe [2006] proposed a MARL algorithm that models the learning process of multiple independent learners. In this algorithm, the learning process of independent cooperative reinforcement learners takes place in two stages:

- Independent learning stage: each learner learns independently using its own Q-learning algorithm until the end of each learning episode.
- Q-value sharing stage: learners share their Q-values following a Q-value update procedure (Q-value sharing strategy).

The researchers proposed three interaction procedures for sharing of Q-values among independent learners:

1. Best Q-value update procedure: The best Q-value of each state-action pair for all agents is chosen using the following update rule:

   \[ Q_i(s, a) \leftarrow Q^\text{best}(s, a)(\forall i, s, a), \]

   where \( i \) is the agent identification number. \( (5.2) \)

   The Q-learning algorithm that uses this sharing strategy is called BEST-Q.
2. Average Q-value update procedure: Each learner averages each Q-value in its Q-table with the best Q-value for each state-action pair. The update rule is:

$$Q_i(s, a) \leftarrow \frac{Q_{\text{best}}(s, a) + Q_i(s, a)}{2} \quad (\forall i, s, a),$$

where $i$ is the agent identification number. (5.3)

The Q-learning algorithm that uses this sharing strategy is called AVE-Q (Average Q-learning).

3. Particle Swarm Optimisation update procedure\(^1\). The best Q-values of agent $i$ and the best global Q-values of all agents are used by the Particle Swarm Optimisation (PSQ) search algorithm to find the best Q-values. The Q-learning algorithm that uses this sharing procedure is called Particle Swarm Optimisation Q-learning (PSO-Q).

Single agent Q-learning and the three proposed swarm RL algorithms, BEST-Q, AVE-Q, PSO-Q, were applied to the shortest path problem. The results show that PSO-Q converges faster to an optimal policy compared to BEST-Q and AVE-Q in short term computations time. Meanwhile, BEST-Q converges faster to an optimal policy than the other cooperative algorithms in long term computations time.

In an extension of the research of Iima and Kuroe [2006] on RL Swarm algorithms, Iima and Kuroe [2007] came as a second research sequel that solves the problem of Q-values procedural update among multiple RL agents. This research proposed five selection procedures to determine the proper time to update the Q-values, and to what extent the update should be done. The procedures are as follows:

**First procedure** Update the Q-values for all the state-actions after the end of each learning episode.

\(^1\)Further discussion about PSO-Q is in Section 8.2.3
Second procedure Update each Q-value the same number of times it was updated during the previous episode. Update the Q-values that were not updated by individual learners during the previous episode once.

Third procedure It is the same as the second procedure. The only difference is that the Q-values that were not updated by individual learners are not updated.

Fourth procedure Similar to the second and the third procedures, except that the update procedure of the Q-values is only performed one time.

Fifth procedure Learners update their Q-values whenever an individual learner updates one of its Q-value.

The research study of Iima and Kuroe [2008] uses the same model of swarm RL of Iima and Kuroe [2006, 2007] except that it replaced the Q-learning algorithm with the SARSA algorithm. The main motivation of using SARSA is that it can find optimal policies for problems with large negative rewards. Unlike Q-learning, SARSA updates each Q-value based on the actually performed action, instead of the maximum available action:

\[ Q(s, a) \leftarrow Q(s, a) + \alpha [R(s, a) + \gamma V(s)], \] (5.4)

\[ V(s) \leftarrow Q(s, a), \] where \( s \in S, \) \( a \) is selected by the policy \( \pi, \alpha \in [0, 1] \)

is the learning rate and \( \gamma \in [0, 1] \) is the discount factor. (5.5)

The researchers compared the performance of swarm Q-learning [Iima and Kuroe, 2006, 2007], single agent Q-learning and swarm SARSA in the shortest path problem. The results show that swarm-SARSA has the best performance in problems with large negative rewards according to two comparison criteria: number of actions and number of episodes.
5.4.3 Cooperation and Coordination in MARL

Liu and Zeng [2006] proposed a Multi-agent Cooperative Learning Model (MCLM) that supports agents concurrent learning based on a new RL algorithm called Multi-agent Cooperative Learning algorithm (MCLA). MCLM is composed of four subsystems (Figure 5.2). First, the predictor module that allows each agent to see other agents’ actions and predict their next actions accordingly. Second, the action selector module that allows the agents to select their actions based on the predictor module. Third, the Q-learning module that uses the action selector to update the value for the chosen actions and finally the execution module that executes the selected actions. The experimental results in an instance of the hunter prey problem show that the cooperative hunters acquired cooperative strategy.

The Swarm RL algorithms of Iima and Kuroe [2006, 2007, 2008] discussed in Section 5.4.2 are primarily based on sharing of Q-values among multiple cooperative agents, while the MCLA algorithm is based on observing and predicting of patterns of behaviours (states and actions) of cooperative agents.
Tan et al. [2009] highlighted two challenges that MARL methods face regarding the cooperation and coordination of behaviours of agents. The first challenge is obtaining the optimal equilibrium by regulating the behaviours of agents, and the second challenge is accelerating convergence to an optimal solution. As a solution for these challenges, Tan et al. [2009] proposed a combinational algorithm that combines the ant colony, quantum and Q-learning algorithms. The quantum algorithm entanglement and superposition phenomena regulate and resolve the behaviours of agents to obtain the optimal equilibrium, and the ant colony algorithm footprint depth (agents' past experiences) phenomenon resolves the communication problem of agents to accelerate the learning process.

Jin et al. [2009] proposed a new cooperative multi-agent RL approach based on state-clusters. This new approach, named State-Clusters MARL (SCMARL), uses a new memory structure called state-cluster as a communication media among cooperative agents. In SCMARL, each agent has its own internal Q-learning algorithm. A state-cluster of a given state contains all acyclic trajectories from all other states to the given state. The state-clusters of all states are integrated into a shared state knowledge space (set of state-clusters).

There are two main differences between the MCLA algorithm and the swarm RL algorithms. First, these algorithms differ in the way they model communication among independent learners. In swarm RL algorithms, the individual learners communicate using a Q-value sharing strategy [Iima and Kuroe, 2006], while the state-clusters are used in SCMARL as a communication media between the cooperative agents [Jin et al., 2009]. Second, these algorithms differ in the way they modify their RL algorithms to model concurrent learning among multiple agents. We can see that the Swarm RL algorithms use many update strategies for cooperative learning such as best-Q, average-Q and particle swarm optimisation of Q-values (PSO-Q) [Iima and Kuroe, 2006], while the MCLA uses four cooperative strategies at the concurrent learning level [Liu and Zeng, 2006].
Weiß [1995] is one of the first research in the field of MARL that proposed two innovative algorithms for the concurrency and the coordination of agents. This study provides two solutions for the synchronisation of agents’ actions in cooperative MARL applications. The first proposed algorithm, named action estimation algorithm (ACE), is composed of two steps: competition step and credit assignment step. In competition step, each agent makes a bid to gain control of the actions flow and selection. Eventually the highest bidder wins, and the agents that lose follow the winner actions sequence. In the credit assignment step, positive reward is given not to the current winner, but to the previous winner as a recognition for its proper, current setup of the environment. Then the current winner subtracts the amount of its bid from its credit then each agent receives a reward based on the adjustment estimation equation and its two basic inputs, external environment reward and winning condition. The second algorithm, named Dissolution and Formation of Groups (DFG), is similar to ACE, except that the two steps procedure is applied to groups of agents instead of a single agent.

The researcher mentioned that the main drawback of ACE and DFG is that they require a lot of exploration or training space. Furthermore, he argued that the generality of the two new methods nominate them for further development. However, these two methods are too general to the extent that they don’t discuss any algorithmic details such as synchronisation of actions and tie bids resolution.

In a team game, each member of a team is required to choose one or more agents to cooperate with them. An intelligent team member should select the team members which maximise its future gains and reduce its computational costs. Buccafurri et al. [2004] is a research study that proposed a model for cooperative team. This system is a homogeneous multi-agent system, named SPY, that informs each agent in a team game about the most promising team’s members to cooperate with them. Basically, this model is based on extracting of semantic properties of agents that are classified into two types. First, local properties that represent the agent local view,
which is stored in its local knowledge base (LKB). Second, global properties that reflect the combination of both local properties and global knowledge of the agents’ environment. Generally, the local properties measure the similarity of agents, while the global properties set attractiveness and interest measures of the agents from a global view.

There are two main limitations for the proposed model of Buccafurri et al. [2004]. First, the SPY system can be applied to team games in which the players are homogeneous, but it cannot be applied to team games in which the players are heterogeneous. Second, heavy computations are required to maintain a comprehensive global knowledge of all agents.

The wide use of multi-agent technology in the web, distributed computing and many other intelligent application fields, makes it very difficult to force uniform design standards on the design of agents, thus we have to cope with the heterogeneity of multi-agent systems. By heterogeneity we don’t mean just only the heterogeneity of the agents’ structure, but also the heterogeneity of the learning algorithms. Moreover, the problem of agents’ heterogeneity becomes more complex, when it comes to cooperation among heterogeneous agents. Hence, Kapetanakis and Kudenko [2004] argued that adding a smart agent to heterogeneous MASs might control the learning performance of these MASs. In more details, they suggested the use of Frequent Maximum Q-values (FMQ) technique for the RL of coordination. FMQ is a heuristic technique, proposed by Kapetanakis and Kudenko [2002], that produces a heuristic value for each action. A heuristic value represents the number of times an action produces its maximum corresponding rewards. FMQ is suitable for heterogeneous cooperative MASs because it was originally developed for independent agents that do not communicate or observe one another’s actions. The experimental results show that the cooperation between one FMQ agent and one Q-learner accelerates the convergence to an optimal solution. But, the addition of one FMQ agent to MASs of size greater than one exponentially increased the number of joint actions.
in general number matching game. In general, we can say that the main two disadvantages of the use of the FMQ technique is the significant increase in the number of joint actions and the computation overhead associated with FMQ.

### 5.4.4 Reinforcement Learning for Stochastic Cooperative Multi-agent Systems

A stochastic game is a dynamic game played in a sequence of stages. In this game, one or more players select actions and each one of them receives a reward based on its action and the current state. Then, the state of the game changes randomly. This random transition depends on the previous state and the chosen actions. Stochastic games can be cooperative games or non-cooperative games. Based on these classifications, many RL algorithms have been proposed to solve stochastic games.

The performance of independent learners in stochastic cooperative games degrades because of the lack of information about the actions taken by their teammates. Each agent $i$ knows about its action $a_i$ of the joint action. A value estimation based on this mixes rewards of many joint actions vectors. Lauer and Riedmiller [2004] proposed a modified Q-learning algorithm to solve this problem. In this algorithm, each agent keeps a list for every state in which an individual entry in this list is a single joint action. An entry in a list is a tuple of the form $L_i(s)[l] = <Q, a_i, n>$, where $a_i$ is the $ith$ component of the referenced joint action vector, $Q$ the respective Q-value and $n$ counts the number of occurrences of the pair $(s, a)$, where $a = \{a_1, a_2, ..., a_m\}$ is the joint action of the $m$ agents. Simply, each list is ordered in Q-values decreasing order that supports exploration. At any instant, all the agents select the same joint action. The list solution has two variants:

- **Full list**: a list type that contains an entry for every possible joint action. This type is impractical because of its big size.
Reduced list: a list that contains a reduced number of joint actions. After some training cycles, some of the joint actions that have not been used are deleted to reduce the size of the list.

A main advantage of Lauer and Riedmiller [2004] is that it reduces communication among agents because the communication among agents is implicit. Although the list solution of Lauer and Riedmiller [2004] accelerates convergence to an optimal policy for small size problems, both medium and large size problems were not discussed. This is because the list solution is unsuitable to represent large state space problems. Therefore, Gabel and Riedmiller [2006] proposed a solution for problems with large space size. This solution is mainly based on the basics of Lauer and Riedmiller [2004] and Case-Based Reasoning (CBR). In this solution, a large state space is decomposed into cases (solutions parts). Each agent is responsible of a certain solution part. When a new state emerges, it is matched to the nearest case representation.

Gabel and Riedmiller [2006] mentioned that their RL approach loses its theoretical convergence to an optimal solution because of two reasons. First, medium and large problems have very large state spaces. Second, the CBR method is based on the similarity of cases rather than certainty. Although the experimental results are promising, the researchers mentioned two main issues that need to be taken into consideration. First, the new method should be tested in a larger problem space than the problem of reactive production scheduling. Second, there is a need for more sophisticated CBR management for this algorithm.

Hu and Wellman [2003] proposed a non-cooperative Nash multi-agent version of Q-learning for general-sum stochastic games. This algorithm uses Nash equilibrium as the soul solution for non-cooperative MAS framework for general sum-games. In Nash equilibrium, each agent has clear expectations about the other agents’ behaviours, and acts accordingly. This vision of Nash equilibrium is translated into a model of agents’ Q-values that each agent adapts and updates. Normally, Nash Q-values are learned through repetitive plays in one of two possible ways, either
by updating each agent Q-values through observing patterns of behaviours of other agents, or by observing the rewards flow, if the rewards are visible.

Both an offline and an online versions of the Nash Q-learning were implemented successfully, especially the online version that succeeds to balance between exploration and exploitation. The experimental results show that the algorithm converges to an optimal solution when applied to a unique Q-function equilibrium, but does not converge to an optimal solution, when applied to more than one Q-function equilibrium.

Ono and Fukumoto [1997] proposed an intelligent, cooperative, modular approach for RL that can be applied to cooperative MAS. The proposed approach models each agent as an individual module that applies Q-learning algorithm. Cooperation among agents, shared decisions and actions selection are the responsibilities of a cooperative selector model called the mediator level in which the best Q-values are selected.

The modular Q-learning approach of Ono and Fukumoto [1997] was applied to solve the hunter prey problem. The simulation results show that the action selection procedure of modular Q-learning does not guarantee convergence to an optimal solution. Park et al. [2001a] applied this algorithm to Robot soccer, but to guarantee the optimal convergence, the action selection procedure was modified by taking into account best Q-values of all modules and the states information of the environment.

Erus and Polat [2007] proposed a cooperative RL method called the Two-Level RL with communication (2LRL) method. The proposed method supports cooperative action selection of agents. Learning of tasks in 2LRL takes place in two hierarchical levels: 2LRL-1 and 2LRL-2. In 2LRL-1, the agent decides if it should cooperate with its neighbours in a joint goal. Then the action selection is done in 2LRL-2. For example, in the hunter-prey problem described in Erus and Polat [2007], the hunter agents are required to balance between two tradeoffs: selfish goals (Hunting small preys) and Joint goals (Hunting big preys).
In 2LRL, the Q-values are stored in a shared repository, so the agents can access them at any learning stages. In addition, the learning agents can follow one of two action selection policies. First, an exploitation policy that selects the action that has the highest Q-value to be performed. Second, an exploration policy, that selects the action that has the lowest Q-value to be performed.

Selfish RL (SRL), and multiple versions of 2LRL-2 were experimentally tested in the hunter-prey problem. The results show that 2LRL-2 algorithm converges faster to an optimal solution.

Tosic and Vilalta [2010] proposed a hierarchical, collaborative RL solution for the problem of agents’ coordination, precisely in dynamic agents’ coalition formation in large-scale problems. In this algorithm, learning takes place in three levels: individual agent RL level, cooperative learning among small groups of agents, and meta learning at the system level. This model supports dynamic adaption of coalition among agents based on continuous exploration and adaption of RL in the three learning levels.

Considering the initial design measurements, the unified model of Tosic and Vilalta [2010] is an accepted model. However, the proposed model doesn’t specify any communication scheme among its three RL levels. Moreover, the model suffers from the absence of detailed algorithmic specifications on how RL would be implemented in this three-layered learning architecture.

The environments of single-agent systems are usually assumed to be stationary environments, but this assumption is invalid for MASs. Interactions, concurrency and communication among agents in MASs make their environments non-stationary ones.

Based on this argument, Wang et al. [2007] introduced a two layered multi-agent RL model and algorithm (LMRL) that are based on the communication protocols, the roles and the rules of the layered approach. The first layer, named single-agent RL layer (SRL), represents the interaction of individual agents based on single-agent
Chapter 5. Multi-agent Reinforcement Learning

106

RL (offline training), while the second layer, that is known as interaction layer, represents and stores the accumulative experiences that are continuously extracted and processed from the agents’ community (online layer).

Clearly, LMRL is a logical layering of roles that distinguishes between the stationary knowledge of a single agent and the non-stationary knowledge of multiple agents. In general, the upper layer of LMRL (Interaction layer) has four subsystems: sensors for observing the environment, multi-agent RL (MRL) for observing the current feed from the environment, effector for choosing the best action and revisor for updating rules in the knowledge base. Those subsystems store their data and rules in two dynamic data stores. The first is a history-based data store for storing interaction among all agents and the second is a Knowledge-based data store for storing interaction rules feed from SRL. The bottom layer (SRL) feeds the interaction layer knowledge components. Additionally, LMRL has two type of learning. First, offline learning at SRL and Revisor. Second, online learning at MRL and Revisor.

LMRL has four main advantages. First, using the layered-approach by itself is an advantage, since it has been proven as a successful architectural design method. Second, the two types of learning exist in this model. Third, agents tend to use optimal strategy as a benefit of feedbacks between the two layers and finally LMRL increases the probability of certainty in the RL decision function.

5.4.5 Reducing Joint Action Space in Cooperative Multi-agent Reinforcement Learning

Ishiwaka et al. [2003] studied cooperation among agents in single-goal multi-agent systems. This research, argued that Q-learning algorithm can be used to model the cooperative behaviour of agents using a prediction model. This model was implemented to the hunter prey problem to model cooperative hunting in which each hunter needs two kinds of prediction. The first is the location of the other hunters and target prey, and the second is the movement direction of the target prey.
at step time $t$. These predictions are incorporated in the state of each agent. In addition, the researchers proposed separating the environmental factors that affect the growth of states in order to prevent the state space from growing exponentially.

According to the experimental results of Ishiwaka et al. [2003], there are three advantages for the use of their model. First, the emergent organisation of hunters depends on the relationship between them and the target. Second, the hunters develop heterogeneous abilities during the learning process. Finally, three cooperative behaviours were noted from the experiments, a following hunter, a resting hunter and an ambushing hunter.

Jiang et al. [2007] proposed a new version of the Q-learning algorithm that reduces the size of joint action space in MASs, and prevents the exponential growth of it. In this method, the researchers used a sparse representation of the coordination dependencies between agents to employ roles and context-specific coordination graphs to reduce the joint action space. In this approach, the joint Q-function is decomposed into a number of local Q-functions. Each Local Q-function is related to a small group of agents, and is composed of a set of value rules. The proposed Q-learning algorithm automatically learns the weights of each value rule. In Jiang et al. [2007], mixed of Coordination Graphs (CG), roles, and hierarchical structures of the Q-functions are used to accomplish the goal of reducing the joint action space. CG is a well known technique that models agents as nodes and dependencies between agents as connected edges. The implementation of CG to RL was first proposed by Guestrin et al. [2002].

Another solution to prevent joint actions space from growing exponentially is CGs. In CG, dependent group of agents can be realised in which a local payoff function is assigned to each connected edge. Kok and Vlassis [2006] proposed two techniques to prevent joint actions space from growing exponentially based on CGs. The first is payoff propagation algorithm (max-plus) that finds an approximation of the joint
action in a CG based on the similarity of the best joint action calculated in pre-
determined rewards functions probabilistic graph and the optimal joint action in
CG. The second is sparse cooperative Q-learning approach that uses predetermined
CGs to learn the payoff behaviours of agents. In this approach, there are two levels
of decomposition: decomposition at the level of agents and decomposition at the
level of edges. In agents’ level decomposition, an agent local function is based on
its own actions and the actions of its surrounding neighbours while in edges’ level
decomposition it is based on the two agents who are connected.

Konidaris and Barto [2009] suggested an approach for high-level skills learning in
continuous RL environments using a skill chaining technique. This research argues
that the concept of option\textsuperscript{2} can be extended and applied in continuous domains of
RL applications using a technique for skills discovery named skill chaining. In this
technique, a skill chain represents a solution path for a certain task, an event or
any designated target. The choice of the target neighbourhood size (region) of a
skill chain is important. In the first hand, a small region’s size might make the
target skill chain hard to be achieved. In the other hand, a very large region’s size
might produce many easy reachable regions. Based on the fact that any target may
have more than one solution path, a tree structure with more than one chain was
suggested where each chain resemble an option for the target event.

There are two main advantages for Konidaris and Barto [2009]. First, this approach
is a hierarchical problem solution one that accelerates the learning process. Sec-
ond, it suits complex non-stationary problems where the task’s function can be re-
duced into multiple approximation functions. However, repetitive learning in several
episodes is required which slows the learning process and makes it an expensive one.
To overcome this problem, Konidaris et al. [2010] proposed CST, an algorithm for
constructing skill trees from demonstration trajectories in continuous reinforcement
learning domains.

\textsuperscript{2}An option is a high-level skills structure that is composed of an option policy, an indicator
function and a termination condition, originally from hierarchical RL.
Basically, CST algorithm constructs skill trees from sample solutions that are supplied from a supervised learning method. In details, this algorithm uses MAP (Max A posterior) change point detection method to segment each solution trajectory into skills and then it adds them to their related solution tree. There are many advantages for CST algorithm. First, skill trees can be successively refined over time, which perfectly suits continuous domains. Second, unsuccessful episodes enhance skill trees. Third, each skill is an abstraction which allows it to develop separately from the other skills.

5.4.6 Teacher Learner Model in Reinforcement Learning

We can identify two common approaches for implementing the teacher-learner model in RL: an agent teacher and a human teacher.

Ahmadabadi and Asadpour [2002] proposed a cooperative Q-Learning approach based on two roles: learner and expert. An agent can be a learner or an expert depending on the level of its knowledge. For example, a learner that has spent 100 episodes learning has more experience than a learner that has spent only 10 episodes. In this algorithm, each learner uses a weighted strategy sharing (WSS) method to assign weights to the Q-tables of its team members, and learns from them accordingly. A learner averages its own Q-values with the corresponding weighted Q-values of the expert. The experimental results show that expertness-based Q-learning is better than single-agent Q-learning when learners have different levels of experience, while single-agent Q-learning is better than expertness-based Q-learning when learners have equal levels of experience. The main disadvantage of this approach is the use of averaging-based methods of cooperative learning for agents with different Areas Of Expertise (AOEs) [Araabi et al., 2007]. The output of averaging-based methods might indicate, wrongly, that cooperation in learning is not beneficial for Q-learning agents [Ahmadabadi et al., 2006].
Ahmadabadi et al. [2006] proposed a method to extract AOE of agents for cooperation in learning using their Q-tables in order to enhance the performance of Ahmadabadi and Asadpour [2002]. In this method, an agent that wants to evaluate the AOEs of the other agents follows two steps. First, using a behavioural measure, the agent evaluates itself to find a set of states on which it is an expert in. Second, the agent uses a classification method to extract the AOEs of the other agents using the set of states resulting from step one and a Q-table-based feature such as $\max Q(s) \leftarrow \max_a(s,a)$. This method has two disadvantages. First, knowledge identification depends on a self-evaluation measure. Second, the classification stage needs more advanced Q-based features for increasing the accuracy of AOE extraction.

In real life, it could be argued that teachers are providers of knowledge and students are receivers of that knowledge. Taking advantage of this observation, Saad et al. [2009] incorporated a human teacher in their compact Q-learning algorithm. In this method, a human assists the learning agent by selecting the proper action at each state. This technique is similar to the study of Thomaz and Breazeal [2008], which incorporated a human teacher in their Q-learning algorithm.

Another example of the teacher-learner model is the research of Kuniyoshi et al. Kuniyoshi et al. [1994] which proposed a cooperative learning algorithm. In this algorithm, a learning task is decomposed into a hierarchy of subtasks. A learner learns these subtasks through watching a human (teacher) performing them.

The methods of Kuniyoshi et al. [1994], Thomaz and Breazeal [2008], and Saad et al. [2009] have two main disadvantages [Ahmadabadi and Asadpour, 2002]. First, the teacher does not receive any feedback from the learner to enhance his/her performance which means that the teacher does not learn anything. This is because none of these methods has a self-evaluation method to evaluate the performance of the learners that can be used to enhance the performance of the teacher. Second, these methods cannot be applied to situations where human intervention is not possible.
5.4.7 Rewards Modifications in Reinforcement Learning

Raicevic [2006] proposed a parallel RL method for agents with an actor-critic architecture based on artificial neural networks. In this method, each agent comprises multiple modules, where each module learns a sub-task and receives a separate reward signal. All the modules can learn in parallel. In this method, the critic is a single-layered artificial network that learns the value function of the state space, and the actor is a single-layered artificial network that learns the action policy.

The experimental results show that there are two main advantages for the algorithm of Raicevic [2006]. First, it converges faster to an optimal solution than classical RL. Second, each module of the agent receives a reward under partial satisfaction of a certain environmental aspect (related to the weight of its contribution to action vector) or equal satisfaction (equal states preferences and actions weight). This allows the agent to exhibit an opportunistic behaviour depending on future environmental state change.

Watanabe and Sawa [2010] proposed an instruction approach for RL based on sub-rewards and forgetting mechanism. Sub-rewards are rewards that are giving for accomplishing certain sub-goals related to a certain state-action pair. Sub-rewards play an important role in reducing the number of learning episodes. Forgetting mechanism is a mechanism of resetting the number of sub-rewards received from the surrounding environment to zero.

In Watanabe and Sawa [2010], the concept of instruction is composed of rewards and sub-rewards that are received from the environment. The number of times that any sub-reward can be received from the environment is controlled by the forgetting mechanism. Two versions of this algorithm were developed, the first is based on Q-learning and the second is based on Fuzzy Q-learning. The results of numerical experiments of a grid world task and a mountain car task show that the combined approach of Watanabe and Sawa [2010] reduces the average length of an
episode compared to single agent Q-learning and forgetting mechanism application. However, the results show also very slight improvements of the combined approach compared to sub-rewards application.

Normally, the reward function is determined according to the specifications of the target application (user-supplied features). Levine et al. [2010] proposed an algorithm that constructs the features of the rewards from a large collection of the component features, yielding a reward function. This algorithm is called Feature Construction for Inverse RL (FIRL). FIRL constructs features which are logical conjunctions between the components that are related to the observed examples. Simultaneously, it forms a reward function for which the optimal policy matches the examples.

FIRL explores the motivations behind the observed behaviour (optimal policy traces) of a RL agent in an MDP environment that is represented by that agent reward function. However, FIRL has some limitations. First, FIRL is not suitable for continuous state space applications because FIRL requires full definition of the problem state space. Second, it assumes that the observed behaviour of an object can be always modelled as a reward function, however, this requires capturing of a large set of the component features in addition to the observed behaviour which is not possible all of the time.

5.5 Summary

This chapter provided some necessary background information to understand Multi-agent Reinforcement Learning (MARL). This includes the definition, benefits, challenges, and some recent advances in MARL. The environment of multi-agent systems is normally a dynamic one with a very large state-space and/or action space,
which makes applying single-agent reinforcement learning approach to these systems inefficient. In addition, single-agent RL approach is unsuitable for distributed applications.

MARL algorithms are a class of RL algorithms that are suitable for multi-agent systems. The chapter discussed the main benefits and challenges in MARL. MARL approach is suitable for parallel computation, sharing of knowledge between agents, robustness, and distributed learning. However, there are two types of challenges in MARL that complicate the learning process: challenges inherited from single-agent RL, and new challenges specific to MARL. The inherited challenges include: the trade-off between exploration and exploitation, and the curse of dimensionality, while the new MARL challenges include: the non-stationarity of the learning problems, the need for coordination between agents, and the difficulty of specifying a good goal.

Large part of this chapter introduced some of the major advances in MARL. First, combinational RL approach, in which more than one RL algorithms are combined together in a single framework to accelerate the learning process and/or to take advantage of the strengths of each algorithm. Second, swarm MARL algorithms that are based on the swarm MAS model, and are used to model the learning process of cooperative independent learners. Third, cooperative MARL algorithms that focus on coordination of agents’ actions and agents’ concurrency. Fourth, cooperative MARL algorithms for stochastic games. Fifth, methods for reducing the joint action space in cooperative MARL algorithms, and finally recent research about cooperative RL using the teacher learner model.

The next chapter will build on the information provided in the current chapter to describe a distributed hierarchical learning model, and an intelligent distributed Q-learning algorithm that are based on three specialisations of agents: workers, tutors and consultants. It will be argued that the organisation of distributed systems into
Chapter 5. *Multi-agent Reinforcement Learning*  

Two levels: system level and subsystem level provides an efficient decomposition of large problem spaces of distributed systems into more manageable components.
Chapter 6

Hierarchical Reinforcement Learning Model

The hierarchical organisation of distributed systems provides an efficient decomposition for distributed systems. This chapter presents a distributed hierarchical learning model (DHLM) and an intelligent distributed Q-learning algorithm (IDQL) that are based on three specialisations of agents: workers, tutors and consultants. Worker agents are the actual learners and performers of tasks, while tutor agents and consultant agents are coordinators at the subsystem and the system levels, respectively. A main duty of consultant and tutor agents is the assignment of problem space to worker agents. The problem space of the consultant agent is the whole system and the problem space of each tutor agent is a sub-problem of the system assigned by the consultant agent. The models were experimented with an ambush strategy in a distributed hunter prey problem. This ambush strategy supports two types of non-communicative cooperation strategies among agents: cooperation at sub-system level and cooperation at system level. The experimental results show that IDQL converges faster to a solution than single-agent Q-learning.


6.1 Introduction

RL has proven to be a successful machine learning technique in single-agent applications and is increasingly being used in multi-agent systems. In RL, the environment is usually modelled as a finite Markov Decision Process [Strehl et al., 2009, Thomas and Marcus, 1997]. The problem space of MARL is normally larger than the problem model of single-agent RL because the number of agents, size of the state and action spaces in multi-agent systems cause the problem space to grow exponentially. In addition, the coordination and/or cooperation requirements between the agents complicate the learning process of the individual agents [Digney, 1996, Erus and Polat, 2007, Jong and Stone, 2008, Ono and Fukumoto, 1997, Park et al., 2001b].

One of the most efficient known approaches of MARL to tackle the above problems is the hierarchical approach [Erus and Polat, 2007, Ghavamzadeh et al., 2006, Ono and Fukumoto, 1997, Tosic and Vilalta, 2010]. This approach solves the problem of large MDPs by decomposing them into smaller subMDPs. The main motivation behind it is that the decomposition of the RL problem space into smaller problems accelerates the learning process of RL learners. Efficient decomposition of large-problem spaces require knowledge of the problem models of these spaces.

Current RL decomposition techniques (Section 3.9.1, and Section 3.9.2) do not allow migration of learners from one problem space to another in distributed systems. Instead they focus on decomposing the state or action space into more manageable parts and statically assign each learner to one of these parts.

This chapter describes a Distributed Hierarchical Learning Model (DHLM) and an Intelligent Distributed Q-Learning algorithm (IDQL) that are based on three specialisations of agents, workers, tutors and consultants. Worker agents are the actual learners and performers of tasks, while tutor agents and consultant agents are coordinators at the subsystem level and the system level, respectively (Figure 6.1). A main duty of consultant and tutor agents is the assignment of problem space to
worker agents. Other duties of the consultant agent include the identification of the subsystems and the assignment of a single tutor to each subsystem. An important feature of DHLM is that it provides a solution for migration of agents. This feature provides support for the IDQL algorithm where the problem space of each worker agent can change dynamically.

![Figure 6.1: Distributed RL Hierarchical Model.](image)

In this chapter, the classical hunter prey-problem (Section 2.1) and the distributed version of it (Section 2.8.6, and Section 3.9.1), will be used to illustrate the efficiency of the DHLM and IDQL algorithm in solving distributed RL problems. Jaakkola et al. [1995] argue that partially observable environments can be modelled as fully observable problems with the learners having limited non-overlapping access to the state space. This is normally formalised by forcing access rules for each agent on the state space. The two versions of the hunter prey problems will be treated as fully observable problems as in Erus and Polat [2007]. The distributed hunter prey problem is a problem with a distributed structure (a grid is divided into sub-grids) where the agents can move from one sub-grid to another one based on migration rules and procedures.

The remainder of the chapter is structured as follows: Section 6.2 gives a brief description of related work, Section 6.3 discusses the DHLM, Section 6.4 discusses the
IDQL algorithm design and mathematical model, Section 6.5 discusses the implementation of the DHLM model and the IDQL algorithm to a distributed version of hunter prey problem, and finally Section 6.6 presents the summary of this chapter.

6.2 Related Work

This section is divided into two subsections. Section 6.2.1 provides a brief description of famous approaches of hierarchical decomposition in the RL domain, while Section 6.2.2 discusses cooperative hunting strategies.

6.2.1 Hierarchical Decomposition in the RL Domain

Different approaches to hierarchical decomposition in the RL domain have been discussed in Section 3.9.1 and Section 3.9.2. We can identify two types of decomposition of MDPs. First, static decomposition that partially or totally requires the implementation designers to define the hierarchy [Parr and Russell, 1997, Sutton et al., 1999]. Second, dynamic decomposition, in which hierarchy components, their positions and abstractions are determined during the simulation process [Dietterich, 2000, Hengst, 2002, Tosic and Vilalta, 2010]. Both techniques focus on decomposing the state or action space into more manageable parts, and statically assign each learner to one of these parts. These techniques do not allow migration of learners from one problem space to another in distributed systems.

The distributed, hierarchical learning model described in this chapter is based on the structure of the natural decomposition of modern software as subsystems. There are no restrictions on the structure of the system hierarchy. Additionally, there are two levels of coordination between subsystems: coordination at subsystem level and coordination at system level. The main goal of this model is to model dynamic migration of reinforcement learners between subsystems in distributed systems to increase the learning speed in distributed systems.
6.2.2 Cooperative Hunting Strategy

Yong and Miikkulainen [2009] describes two cooperative hunting strategies that can be implemented in the hunter prey problem. The first is a cooperative hunting strategy for non-communicating teams that involves two different roles of hunters, which are chaser and blocker. In detail, the role of the chaser is to follow the prey movement, while the role of the blocker is to move in a horizontal direction of the prey, staying in the vertical axis of the prey. This allows the blocker to limit the movement of the prey in the direction of prey’s movement. The second strategy is also a cooperative hunting strategy for non-communicating teams, but only involves one role of hunters, which is chaser. In order for two chasers to sandwich the prey, at least two chasers are required to follow the prey in opposite directions to eventually surround the prey agent. This means that at least one chaser must travel the long distance on the grid so that two chasers can be on the opposite sides of the prey. Both hunting strategies were experimentally proved to be successful strategies. One main advantage of these strategies is that no communication is required between hunters. However, both strategies require the prey position to be part of the state definition to provide the chasers and/or blockers with the knowledge of the prey’s position. Additionally, these strategies can be implemented in a single grid level, but cannot be implemented directly in a distributed grid model because this requires redefinition of the chaser and blocker roles.

Lee [2005] proposed a hunting strategy that involves two different roles of hunters, chaser and blocker. In Lee [2005], the roles of hunters are semantically similar to the roles of the hunters of the first strategy of Yong and Miikkulainen [2009]. However, the description of roles is relatively different. In detail, the chasers drive the prey to one of the corners of the grid, while the role of blockers is to surround the prey so it does not have enough space to escape. The hunting is considered successful if the prey is captured. A main difference between blockers in Lee [2005] and Yong and Miikkulainen [2009] is that blockers are required to communicate to surround
the prey agent. This is considered as a disadvantage of the hunting strategy of Lee [2005].

None of the mentioned hunting strategies in this section can be directly implemented to the distributed hunter prey problem described in Section 2.8.6, and Section 3.9.1 because this requires redefinition of the chaser and blocker roles. In addition, the structure of this problem requires a more sophisticated hunting strategy that can be implemented at different levels of the grids. An ambush strategy will be discussed in Section 6.3.6 that is composed of two levels for distributed hunter prey problem. The reasons of using such a strategy will be discussed in Section 6.3.6.

6.3 Distributed Hierarchical Learning Model

In this section, we will discuss the basic concepts of the proposed model using the distributed hunter prey problem described in Sections 2.8.6, and 3.9.1.

6.3.1 Problem Model

The problem model of the distributed hierarchical model is mainly a loosely-coupled FMDP (Section 3.9.1) defined and organised by the implementation designers into two levels: system and subsystem. The loose coupling characteristic of such a system means that each one of its components has or uses little knowledge about other components [Kaye, 2003].

A system is a tuple \([S,A,W,T]\), where \(S\), \(A\), and \(T\) are defined as in an MDP (see Section 3.1) and \(W\) is a set of reward functions \(R : S \times A \rightarrow \mathbb{R}\) for different roles that may be used in the system. A role can then be defined as an MDP \([S,A,R,T]\), where \(R \in W\).
A subsystem is a MDP with a connection set that defines the subsystem’s boundaries with its neighbouring subsystems. More formally, given a role $Role = [S, A, R, T]$, a subsystem is a tuple $Sub = [M, C]$, where:

1. $M = [S_{sub}, A_{sub}, R_{sub}, T_{sub}]$ is a MDP where:
   - (a) $S_{sub} \subseteq S$ is the set of states in the subsystem.
   - (b) $A_{sub} \subseteq A$ is the set of actions in the subsystem.
   - (c) $R_{sub} : S_{sub} \times A_{sub} \rightarrow \mathbb{R}$ is a reward function such that, given $s \in S_{sub}, a \in A_{sub}, r \in \mathbb{R}, R_{sub}(s, a) = r \iff R(s, a) = r$.
   - (d) $T_{sub} : S_{sub} \times A_{sub} \times S_{sub} \rightarrow [0, 1]$ is a transition function such that, given $s_i, s_j \in S_{sub}, a_k \in A_{sub}, t \in [0, 1], T_{sub}(s_i, a_k, s_j) = t \iff T(s_i, a_k, s_j) = t$.

2. $C : S_{sub} \times A \times (S \setminus S_{sub}) \rightarrow [0, 1]$ is a connection set which specifies how $Sub$ connects to other parts of the system such that, given $s_i \in S_{sub}, a \in A, s_j \in S \setminus S_{sub}, t \in [0, 1], C(s_i, a, s_j) = t \iff T(s_i, a, s_j) = t$.

### 6.3.2 Example

The grid in the distributed hunter prey problem represents a system where each sub-grid represents a subsystem. Each subsystem $i$ is composed of:

- An $MDP_i$ that is composed of:
  - A set of possible states, $S_i = \{cell_0, cell_1, ..., cell_{n-1}\}$.
  - A set of possible actions, $A_i = \{\text{move up, move down, move left, move right}\}$.
  - The reward that each hunter agent receives is defined as:

$$R_{Hunter}(s, a, s') = \begin{cases} +100.0 & \text{if it moves to the same cell as the prey} \\ 0 & \text{otherwise} \end{cases}$$
– The transition model for the hunter agents is:

\[
T_{\text{Hunter}}(s, a, s') = \begin{cases} 
1 & \text{if } s' \text{ is adjacent to } s \text{ in the direction of } a \text{ and is } \\
& \text{a valid position for the hunter } \\
0 & \text{otherwise}
\end{cases}
\]

– A connection set \( C_i : S_i \times A \times (S \setminus S_i) \rightarrow [0, 1] \), where \( S \) is the set of all possible states on the distributed hunter prey problem.

### 6.3.3 Design of DHLM

We have applied the specialisation principle on the design of our proposed distributed learning model. This principle supports the separation of duties among agents in distributed MASs. The result is a distributed hierarchical learning model (DHLM) that has three types of agents: workers, tutors and consultants (Figure 6.2).

![Figure. 6.2: Generalisation of DHLM agents.](image)

The following are the basic definitions of the three agent types of the DHLM:

- **Worker agents** are the actual learners and performers of tasks in distributed systems. Worker agents can play different roles.

- **Tutor agents** are coordinators at the subsystem level where each subsystem has one tutor. The main duties of tutors include: learning solutions of subsystems (Section 6.4.1), distributing worker agents (Section 6.3.6), coordinating actions
of worker agents and coordinating with consultants (Section 6.3.4). Moreover, tutor agents arrange the migration of agents with consultants (Section 6.3.4), or coordinate their actions with other tutors in other subsystems. Tutor agents play the roles of teachers for worker agents at the subsystem level (Section 6.3.6).

- Consultant agent is the only coordinator at the system level. Its main duties include: exchanging of information with tutors, granting migration visas and arranging agent migration from one local environment to another (Section 6.3.4), and finally teaching worker agents at the system level (Section 6.3.6).

The problem space of the consultant agent is the whole system while the problem space of a tutor agent is the subsystem assigned to it by the consultant agent. The problem model of each worker agent is assigned by either its tutor or consultant based on the description of the role. The consultant agent keeps track of the tutor agents in the system while the each tutor agent keeps track of the worker agents in its subsystem.

### 6.3.4 Migration of Agents

One main duty of the consultant is to redistribute worker agents from one subsystem to another one based on the distribution of goals (prey). In the context of the distributed hunter prey problem, redistribution of worker hunter can happen for one of the following reasons:

1. Redistribution of worker hunters according to the need of each hunting group.
2. Redistribution of worker hunters according to the need of each local grid.

Migrant agents can follow one of two different migration modes depending on a design time decision.
1. A client server mode (1st Procedure of Figure 6.3).

2. A mobile agent mode (2nd Procedure of Figure 6.3).

**Figure. 6.3:** Example of the two migration procedures. In the figure, worker agent $W_0$ migrated from the right hand subsystem to the left hand subsystem (mobile agent mode). The worker $W_1$ remained on its original subsystem (The right hand subsystem) and a copy of it migrated from the right hand subsystem to the left hand subsystem (client-server mode).

The algorithm that performs the migration is inspired by the research studies of Boyd and Dasgupta [2002], and Vasudevan and Venkatesh [2006] in the field of process migration in operating systems. However, our algorithm is an application-level algorithm that organises the migration process of RL agents between subsystems. The migration algorithm is shown in Figure 6.4. Steps 5 and 10 of the above algorithm are related to the problem space of RL agents. The problem space is related to the MDP model described in Section 3.1.

One main goal of DHLM is to increase the efficiency of worker agents as much as possible. Worker agents in some subsystems might be working to achieve their goals, while worker agents in other subsystems are idle. Such a situation requires redistribution of worker agents to the subsystems that are still active to help in performing the subsystems’ tasks faster. For example, Figure 6.5 shows that the hunter agents on sub-grid $A$ have finished hunting all their prey agents, while the hunting is still active in sub-grids $B, C$ and $D$. The consultant agent tracks the progress of work (hunting) using a monitor program that tracks the number of hunted prey agents and the free prey agents. The logical action in such a scenario is to provide additional worker hunters for the parts of the grid that are still active.
1: A migration request is issued to a consultant through a tutor. After a successful evaluation based on the availability of free worker agents on other subsystems and the need of worker agents on other subsystems, a migration visa is granted.

2: A migrant agent is declared to be in a migrating state at subsystem level and at system level.

3: If the migrant agent is a part of a stochastic game:
   1. Number of agents of stochastic game in the source subsystem = Number of agents of stochastic game in the source subsystem – 1.

4: If running in client server mode:
   1. Duplicate worker agent.
   2. Maintain a communication channel between the agent and its copy for the rest of the steps.
   3. Go to step 7.

5: Write the problem space of the migrated agent to the tutor of the source-subsystem.

6: Terminate the migrant agent process or thread.

7: Relocate the migrant agent to a new subsystem.

8: Inform the destination-subsystem’s tutor of the migrant’s new location.

9: Resume the agent thread.

10: Allocate problem space to the migrant agent.

11: If the migrant agent is a part of a stochastic game:
   1. Number of agents of stochastic game in the destination subsystem = Number of agents of stochastic game in the destination subsystem + 1.

12: Resume the execution of migrant agent.

13: If a worker agent finishes execution and it is running in mobile agent mode:
   1. Terminate the migrant agent process or thread
   2. Deallocate memory and data of the migrant agent.
   3. Relocate the migrant agent to its original subsystem.
Redistribution of worker hunters is one of the responsibilities of consultant agents. A consultant agent has a monitor program that uses two data structures to monitor tutor’s activities. First, a service queue that is used to register the tutors that are still active. An active tutor is the one whose worker agents are still performing or learning their assigned tasks. Second, an activity list that is used to register inactive tutors (inactive list). Figure 6.6 shows the contents of the service queue and inactive list for the case shown in Figure 6.5.
1: LinkedList Inactive\(<\text{tutor}\>) 
2: Queue Service\(<\text{tutor}\>) 
3: while true do 
4: \hspace{1em} for each tutor do 
5: \hspace{2em} if ! tutor.finished == true then 
6: \hspace{3em} Service.enqueue(tutor.number) 
7: \hspace{2em} end if 
8: \hspace{1em} end for 
9: \hspace{1em} for each tutor do 
10: \hspace{2em} if tutor.number-of-goals== 0 then 
11: \hspace{3em} tutor.finished = true 
12: \hspace{3em} Inactive.add(tutor.id) 
13: \hspace{2em} for each worker \in tutor.group do 
14: \hspace{3em} \hspace{1em} worker.state-flag= available 
15: \hspace{2em} \hspace{1em} end for 
16: \hspace{2em} end if 
17: \hspace{1em} end for 
18: \hspace{1em} if ! Service.empty() then 
19: \hspace{2em} for each entry of Service do 
20: \hspace{3em} if entry.finished == true then 
21: \hspace{4em} Service.remove(entry) 
22: \hspace{4em} Inactive.add(entry.id) 
23: \hspace{3em} end if 
24: \hspace{2em} end for 
25: \hspace{1em} end if 
26: \hspace{1em} target = Service.dequeue() 
27: \hspace{1em} if target.number-of-goals! = 0 and Inactive.size()! = 0 then 
28: \hspace{2em} for each worker \in Inactive.getfirst() do 
29: \hspace{3em} worker.migrate(target) 
30: \hspace{2em} end for 
31: \hspace{2em} Inactive.removeFirst() 
32: \hspace{2em} end if 
33: \hspace{1em} end while 

Figure 6.7: Redistribution of worker agents.
Consultant agents use the algorithm in Figure 6.7 to monitor the progress of work in each subsystem, and redistribute worker hunters when they are needed in other subsystems. The monitor program simply keeps track of the progress of each tutor by registering active tutors in the service queue. The monitor program also initiates agents’ migration when it is required. The monitor program works as follows:

1. Register each tutor that is active and working to achieve its goals in the service queue (Lines 4 to 8).

2. If any tutor finishes processing its goals, register it in the inactive list, and flag the state of its workers as available (Lines 9 to 17).

3. If any tutor in the service queue finishes processing its goals, delete it from the service queue, and register it in the inactive list (Lines 18 to 25).

4. Set the environment of the first tutor in the service queue as the target environment for migration (Line 26).

5. Apply the migration procedure for all worker agents that follow the tutor that is registered first in the inactive list (Lines 27 to 30).

6. Delete the first entry in the inactive list (Line 31).

7. Go to step 1.

### 6.3.5 Task Scheduling

The implementation of MARL in a multiple goals MAS requires cooperative work and tasks scheduling. Each tutor agent as a team leader is responsible of its team plan. In other words, each team leader is responsible of choosing the task scheduling policy that can be sequential, parallel or priority policy. The choice of a policy depends on a number of factors: the number of agents, the number of goals, the relative positions of agents to goals, the distance between agents and goals, the
distance between each goal and the other goals, and the movement of each goal. The state space of a tutor agent includes the states of its subsystem combined with the position of the prey.

One of the ways for calculating the distance between two points is the Manhattan distance [Perlibakas, 2004]. The Manhattan distance between two points is the sum of the absolute differences of their corresponding component. Formally, the Manhattan distance \((D_m)\) between two points, \(a = (a_1, a_2, ..., a_n)\) and \(b = (b_1, b_2, ..., b_n)\), is:

\[
D_m = \sum_{i=1}^{n} |a_i - b_i|, \quad \text{where } n \text{ is the number of components, } a_i \text{ and } b_i \text{ are the values of the } i_{th} \text{ component of the points } a \text{ and } b.
\]

A disadvantage of Manhattan distance is that it is not the best method to be used to measure the distance of objects that can move horizontally.

In a 2D space like the hunter prey problem, each cell has a horizontal and a vertical components. Thus, the Manhattan distance in a grid \(G\) between a cell \(G[i_1, j_1]\) and another cell \(G[i_2, j_2]\) is:

\[
D_m = |i_1 - i_2| + |j_1 - j_2|, \quad \text{where the } i_{th} \text{ component is the row number and the } j_{th} \text{ component is the column number of the two cells.}
\]

In order to calculate the Manhattan distance between two agents in a grid, the cells of the grid must be arranged in a rectangular shape, the grid should have fixed dimensions and the coordinates of each agent should be known.

In DHLM, consultant and tutor agents use sequential, parallel, and/or priority scheduling procedures to distributed worker agents among goals. In the implementation of the DHLM, the main scheduling procedure (Figure 6.8) selects the scheduling type after receiving one parameter \((\text{region})\) which can be a system, a subsystem or a region in a subsystem. This procedure shows that the agents can be distributed to the goals based on three scheduling policies: sequential, parallel or priority scheduling (Line 6). The choice of the scheduling procedure depends on the understanding of the implementation designer of the learning problem. For example, the designer
may decide that the nearest prey to each hunter agent should be targeted first to reduce the number of moves required to hunt all the pre agents. The identity and the \( x \) and \( y \) coordinates of each goal and agent of \( \text{region} \) are respectively stored in \( \text{Goals} \) and \( \text{Agents} \) lists (Lines 3 to 9). The \( \text{Goals} \) and the \( \text{Agents} \) lists are then passed as actual parameters to the sequential scheduling procedure (Line 12), the parallel scheduling procedure (Line 14) or the priority scheduling procedure (Line 16).

```
1: Procedure scheduler(<type>[ ][ ]region)
2: begin
3: Struct Point{
4:  int Id, xCoordinate, yCoordinate
5: }
6: Set taskSchedulingMode to parallel, sequential or priority based on the nature of the learning problem
7: LinkedList Goals < Point>, Agents < Point>
8: Scan \( \text{region} \) to identify the \( x \) and \( y \) coordinates of each goal in \( \text{region} \). Store the identified \( x \) and \( y \) coordinates of each goal in \( \text{Goals} \)
9: Store the identified \( x \) and \( y \) coordinates of each agent of \( \text{region} \) in \( \text{Agents} \)
10: if Goals & Agents are not empty then
11:  if taskSchedulingMode == sequential then
12:    Call sequentialScheduler(Goals,Agents)
13:  else if taskSchedulingMode == parallel then
14:    Call parallelScheduler(Goals,Agents)
15:  else
16:    Call priorityScheduler(Goals,Agents)
17: end if
18: end if
19: end
```

**Figure. 6.8:** Task scheduling.

The sequential scheduling procedure (Figure 6.9) guarantees that only one goal is learned at a time, and that all agents are learning this goal. Once the agents of \( \text{Agents} \) list are assigned to learn a goal of \( \text{Goals} \) list (Lines 8 and 9), the procedure continues to wait until the learning of the current goal is finished (Lines 10 to 12). The procedure resumes when the current goal is learned, and then moves to the next goal in sequence (Line 10). If the learning goal is a moving goal, its location
must be included in the state definition to determine the finishing point of learning. Priority scheduling means that the number of agents assigned to each goal should

reflect its importance represented by an importance number. Figure 6.10 shows a priority scheduling procedure. In the beginning, each goal is assigned an importance number based on its importance (Line 13). Simultaneously, the importance number and the identity of each goal is stored in the linked list goalsProrities (Line 14). The goals in goalsProrities are then ordered in ascending order (Line 17). Finally, the agents are distributed to the goals in goalsProrities one by one starting from the first position until all the agents are distributed (Line 20). For our better understanding, let us consider the following example: A hunter prey problem of size 10 * 10 with 10 hunter agents and 3 prioritised prey agents \( (P = \{P_1 = 2, P_2 = 1, P_3 = 2\}) \). The order of the goals in goalsProrities after ordering it is \( (goalsProrities = \{P_2 = 1, P_3 = 2, P_1 = 2\}) \). Thus, the number of agents assigned to \( P_1 \) is 4, \( P_2 \) is 3 and \( P_3 \) is 3.

The parallel scheduling procedure allows parallel learning of goals. This procedure

### Procedure sequentialScheduler \( \text{LinkedList} < \text{Point} > \text{Goals, LinkedList} < \text{Point} > \text{Agents} \)

1. **begin**
2. if Goals.empty != false then
3. Point currentGoal=Goals.first()
4. Goals.remove(Goals.first())
5. **end if**
6. **while** Goals is not null do
7. Assign each agent \( \in \text{Agents} \) to currentGoal
8. Start learning how to perform currentGoal
9. **while** learning is not finished do
10. wait()
11. **end while**
12. currentGoal=Goals.next()
13. Goals.remove(Goals.first())
14. **end while**
15. **end**

Figure 6.9: Sequential scheduling procedure.
Chapter 6. Hierarchical Reinforcement Learning Model

(Figure 6.11) visits one agent at a time from the Agents list (Lines 3, 4, 17 and 18) to assign it to the goal that has the minimum Manhattan distance to it (Line 7 and lines 9 to 16). An important factor for the success of this procedure is that the agents should be uniformly distributed on the grid. If the agents are not uniformly

```
1: Procedure priorityScheduler(LinkedList < Point > Goals,  
   LinkedList < Point > Agents)
2: begin
3: Struct goalPriority{  
4:   int Id, importance-number, numberOfAgents=0
5: }
6: goalsPriorities LinkedList
7: int sumOfPriorities=0
8: if Goals.empty!= false then
9:   Point currentGoal=Goals.first()
10:   Goals.remove(Goals.first())
11: end if
12: while Goals is not null do
13:   Assign an importance-number to currentGoal based on its importance
14:   Store the importance-number and the identity of the goal in goalsPriorities
15:   currentGoal=Goals.next()
16: end while
17: Order goalPriorities in descending order based on goalsPriorities.importance-number
18: int count=Agents.size()
19: currentPriorityGoal=goalsPriorities.first()
20: while count ≥ 0 do
21:   while goalsPriorities is not null and count ≥ 0 do
22:     currentPriorityGoal.numberOfAgents=
23:     currentPriorityGoal.numberOfAgents+1
24:     currentPriorityGoal=currentPriorityGoal.next()
25:     count=count-1
26:   end while
27:   currentPriorityGoal=goalsPriorities.first()
28: end while
29: while Goals is not null do
30:   Start learning how to perform currentGoal
31:   currentGoal=Goals.next()
32: end while
33: end
```

Figure 6.10: Priority scheduling procedure.
1: Procedure parallelScheduler(LinkedList<point> Goals, LinkedList<Point> Agents)
2: begin
3: Point currentGoal, chosenGoal
4: Point currentAgent=Agents.first()
5: while Goals is not null do
6: currentGoal=Goals.first()
7: Calculate the Manhattan distance from currentGoal to currentAgent
8: chosenGoal=currentGoal
9: while Goals is not null do
10: Calculate the Manhattan distance from currentGoal to currentAgent
11: if Manhattan distance from currentGoal of currentAgent ≤ Manhattan
distance of chosenGoal to currentAgent then
12: chosenGoal=currentGoal
13: end if
14: currentGoal=Goals.next()
15: end while
16: Assign chosenGoal to currentAgent
17: currentAgent=Agents.next()
18: end while
19: end

Figure. 6.11: Parallel scheduling procedure.

distributed on the grid, the procedure might assign all the agents to one or few goals
leaving the rest of the goals without agents to learn or perform them.

Let us consider the following example: A hunter prey problem of size 3 * 4 with five
hunter agents and two prey agents (Figure 6.13a). If we assumed that hunting a
prey agent is a goal, then we can define two goals that are individually classified as
tasks (T1: hunting prey P1, T2: hunting prey P2). Figure 6.13b shows a table of the
Manhattan distance of the hunter agents, H = \{H1, H2, H3, H4, H5\}, and the prey
agents, P = \{P1, P2\}, to each prey agent P = \{P1, P2\}. The last column of the table
shows that each hunter agent is assigned to the prey agent that has the minimum
distance to it.

The proposed scheduling procedures are inadequate in some cases. Sequential does
not take advantage of the many agents, priority ignores distances and parallel ignores
the importance of each goal. A better procedure is a combination of the priority and parallel procedures. In this procedure, the priority procedure determines the number of agents per goal (Lines 3 to 17 of Figure 6.12), then a variation of the parallel procedure assigns agents to goals (Lines 18 to 29 of Figure 6.12).

```
1: Procedure combinedScheduler(LinkedList<Point> Goals, LinkedList<Point> Agents)
2: begin
3: Struct goalPriority
4: int Id, importance-number, numberOfAgents=0
5: }
6: goalsPriorities LinkedList
7: int sumOfPriorities=0
8: if Goals.empty!= false then
9: Point currentGoal=Goals.first()
10: Goals.remove(Goals.first())
11: end if
12: while Goals is not null do
13: Assign an importance-number to currentGoal based on its importance
14: Store the importance-number and the identity of the goal in goalsPriorities
15: currentGoal=Goals.next()
16: end while
17: Order goalPriority in descending order based on goalsPriorities.importance-number
18: int count=Agents.size()
19: Point currentAgent=Agents.first()
20: Point currentPriorityGoal=goalsPriorities.first()
21: while count≥ 0 and Goals is not null do
22: while goalsPriorities is not null do
23: Assign currentAgent to currentPriorityGoal and start learning
24: currentAgent=Agents.next()
25: currentPriorityGoal=currentPriorityGoal.next()
26: count=count-1
27: end while
28: currentPriorityGoal=goalsPriorities.first()
29: end while
```

Figure. 6.12: Combined scheduling procedure.
6.3.6 Roles of Worker Agents at Different Levels of Distributed Systems

Worker agents play different roles to perform tasks at system and subsystem levels. As coordinators, consultant agents and tutor agents are responsible for assigning these roles to worker agents.

We will discuss in this section cooperative strategies at subsystem level and system level.

Cooperative Roles at System Level

Worker agents might be required to work cooperatively or independently to learn and solve problems outside their local systems. For example, a prey agent in a distributed hunter prey problem might migrate (escape) from one sub-grid to another one. In such a case, hunters from different sub-grids need to cooperate to catch the escaping prey. In order to provide a solution for this problem, we propose an ambush strategy at system level that is inspired from the research of Boesch [2002] about the cooperative hunting roles among Tai chimpanzees:
• Chaser hunters are worker hunters that chase prey agents that escape outside their local systems.

• Blocker hunters are worker hunters that are responsible for blocking the way of the escaped prey agents. The blocker learns how to block some blocking cells in the corners of the grid.

• Ambusher hunters are worker hunters that attack suddenly and without warning from a concealed place. The ambusher learns how to capture the prey agent in the ambush area which is the area between the blocking cells.

Figure 6.14 shows that the chaser hunters inherit the problem space and the Q-learning algorithm of the system level, while the ambusher hunters and the blocker hunters inherit the problem space and the Q-learning algorithms of one or more tutors at subsystem level (Section 6.4).

Figure 6.15 shows an example of the distributed hunter prey problem, in which an ambush strategy is applied at system level. In the figure, the chaser hunters (H) are located at the same grid of the target prey, while the the blocker (H) and the ambush hunters (H) are located at the neighbouring grids of the target prey. The ambush area is composed of four cells (← . . . . →). The blocking cells and the ambush cells are fixed. There are two blocker cells in each corner of the neighbouring
sub grid of the target prey. The ambush cells are in the centre of the neighbouring edge, which the ambush hunter targets.

Consultant agents use the ambush algorithm (Figure 6.16) to choose these hunter agents. In the beginning, the algorithm distributes the available hunter agents over the sub grids based on the number of prey agents on each sub grid (Lines 1 to 15). The algorithm then identifies the prey agents of each grid that are most likely to escape (target prey agents) to other grids (Line 17). The algorithm then chooses a specific number of worker hunters to chase each target prey agent (Lines 19 and 20). The chaser hunters of each target prey should be the nearest non-specialised worker hunters to each target prey. These chaser agents inherit the problem space of consultant agents (Line 21). The algorithm then continues, and chooses a specific number of worker hunters on each neighbouring grid of each target prey agent to be blocker hunters. The blocker hunters of each target prey should be the nearest non-specialised worker hunters to each target prey. (Lines 23 and 24). These blocker hunters inherit the problem space of the tutor hunters of the neighbouring grid and the target prey of the blocker hunters (Line 24). At the end of the algorithm, an ambusher hunter is chosen from each group of blocker hunters for each target prey (Line 25). The ambusher is a blocker hunter that has the maximum distance to its target prey agent compared to other blockers within its group.

The success of the ambush algorithm depends on the distribution of the hunters on the grids, which should be uniformly distributed, and the availability of a sufficient number of worker hunters to play different roles in the hunting process.

**Cooperative Roles at Subsystem Level**

The idea of the ambush can be implemented at subsystem level, where we assume that prey agents cannot migrate to other subsystems. The following is the ambush strategy at sub-grid level:
Figure 6.15: Blocker, chaser and ambusher agents in a distributed hunter prey problem. The number of chaser agents in the target-prey’s sub-grid is \( n = 2 \). Each adjoining sub-grid to the target-prey’s sub-grid has 4 blocker agents \( (m = 4) \) and one ambusher agent \( (o = 1) \).

- Chaser hunters chase prey agents inside their sub-grid. They inherit the problem space of tutor agents.

- Blocker hunters are responsible for blocking the way of the escaped prey agents to direct them to one of the ambush areas. The problem space is determined by tutor agents.

- Ambusher hunters attack suddenly and without warning from a concealed place (around the corner). The problem space is determined by tutor agents.

Figure 6.17 shows that all the hunter specialisations inherit the problem space and the Q-learning algorithm of their tutor (Section 6.4). Figure 6.18 shows a sub-grid in which ambusher hunters \( (A) \) hide around the corner waiting for the prey \( (P) \), two chaser hunters \( (C) \) chase \( P \) and three blocker hunters \( (B) \) block the way of \( P \) to push it to one of the ambush areas.
1: `LinkedList prioritiesList < subgrid >`
2: Store the subgrids in prioritiesList
3: Order prioritiesList in ascending order based on the number of prey agents
   on each subgrid.
4: `count=totalNumberOfAgents`
5: `currentSubgoal=prioritiesList.first()` 
6: `currentAgent=Agents.first()` 
7: `while count > 0 do`
8:   `while prioritiesList is not null do`
9:     Assign `currentAgent` to `currentSubgoal`
10:    `currentAgent=Agents.next()` 
11:    `currentPriorityGoal=currentPriorityGoal.next()` 
12:    `count=count-1`
13:  `end while`
14: `currentSubgoal=prioritiesList.first()` 
15: `end while`
16: `for each sub-grid ∈ grid do`
17:   Choose the prey agents that have the minimum Manhattan distance to
   the boundaries of each neighbouring grid to be the target prey agents.
18:  `for each target prey do`
19:   Choose $n$ number of worker hunters from the same grid as the prey to
   be chaser hunters that satisfy the following conditions:
20:     • They are not blocker, chaser or ambusher hunters.
21:        • They have the minimum Manhattan distance to the target prey
22:   `inheriting the problem space of consultant agent to each chaser hunter.
23:  `for each neighbouring grid do`
24:   Choose $m$ worker hunters from the current neighbouring grid that
   satisfy the conditions of Line 20 to be blocker hunters.
25:   `Inheriting the problem spaces of the tutor hunters of the neighbouring
26:   grid and the target prey to the blocker hunters.
27: `Choose $o$ blocker hunters that are the most far from the target prey
28:   agent to be ambusher hunters.``

**Figure. 6.16:** Ambush algorithm (system level).
Chapter 6. *Hierarchical Reinforcement Learning Model*

**Figure. 6.17:** Specialised hunter agents at sub-system level. The chaser, blocker, and ambusher agents inherit the problem space of a tutor agent.

**Figure. 6.18:** Blocker, chaser and ambusher agents at grid level of a distributed hunter prey problem.

The ambush algorithm (Figure 6.19) is used by the tutor agents at sub-grid level to choose the worker hunters that are going to participate in each sub-grid ambush. In the beginning, the algorithm scans each sub-grid counting the prey agents of each one (Lines 2 and 3). The algorithm then chooses some non-specialised worker hunters to play the roles of ambush hunters in the following order: chaser hunters, ambusher hunters, and finally blocker hunters. The algorithm assigns each specialised worker hunter to learn the problem space of its role (chaser: Lines 5 to 7, ambusher: Lines 8 and 9, and blocker: Lines 10 and 11).
1: Let \( m \) be the number of sub-grids.
2: for \( g = 1 \rightarrow m \) do
3: \( n = \text{count}(\text{prey})_{\text{of grid}}[g] \).
4: for \( p = 1 \rightarrow n \) do
5: Choose \( x \) number of worker hunters to be chaser hunters that satisfy the following conditions:
6: • They are not blocker, chaser or ambusher hunters.
7: • They have the minimum manhattan distance to \( \text{prey}[p] \).
8: Inherit the problem space of tutor\([g]\) to each chaser hunter.
9: Choose \( y \) worker hunters that satisfy the first condition of Line 6 to be ambusher hunters to the nearest ambush area.
10: Assign the problem space of each ambush area to the nearest ambusher hunters to it.
11: Choose \( z \) worker hunters that satisfy the first condition of Line 6 to be blocker hunters to the nearest blocking area.
12: Assign the problem space of each blocking area to the nearest blocker hunters to it.
13: end for

Figure 6.19: Ambush algorithm (subsystem level).

6.4 Intelligent Distributed Q-learning Algorithm

This section discusses a new RL algorithm called intelligent distributed Q-learning algorithm (IDQL). IDQL proposes that each problem space of each agent specialisation of DHLM can be separately modelled. The problem space model is an MDP for independent learners (Section 6.4.1).

6.4.1 Independent Learners

IDQL proposes that the problem space of each tutor agent and each consultant agent can be modelled as a separate MDP as follows based on Section 6.3.1:

Let \( sys \) be a system composed of \( m \) subsystems. Then,

• A subsystem \( s_i \) of \( sys \) is a problem model of tutor agent \( i \).
• The problem space of each consultant agent is sys.

In order to understand the above two divisions, let us consider the following example. In the distributed hunter prey problem, the state space of a worker hunter that inherits the problem space of its tutor is limited to positions within its local grid, while the state space of a worker hunter that inherits the problem space of the consultant agent covers all positions within the distributed grid. Each chaser hunter that inherits the problem space of its tutor learns how to catch a prey agent inside its sub-grid (Section 6.3.6). In the other hand, each chaser hunter that inherits the problem space of the consultant learns to catch a prey agent at system level (Section 6.3.6). The action space of the chasers contains four possible actions: move up, move down, move right, and move left.

6.4.2 The Q-Functions of IDQL

IDQL is represented by two types of Q-functions. The first one is the subsystem related Q-function:

\[ Q_i(s, a) \leftarrow (1 - \alpha) Q_i(s, a) + \alpha [R_i(s, a) + \gamma \max_{a' \in A_i} Q_i(s', a')], \]

where

- \( i \) is the subsystem identity,
- \( s \in S_i \) is a state from the state space of \( i \),
- \( \alpha \in [0, 1] \) is the learning rate and \( \gamma \in [0, 1] \) is the discount factor. \( (6.1) \)

The second Q-function is the system related Q-function:

\[ Q_{sys}(s, a) \leftarrow (1 - \delta) Q_{sys}(s, a) + \delta [R_{sys}(s, a) + \theta \max_{a' \in A_{sys}} Q_i(s', a')], \]

where

- \( \delta \in [0, 1] \) is the learning rate and \( \theta \in [0, 1] \) is the discount factor. \( (6.2) \)
Chapter 6. *Hierarchical Reinforcement Learning Model*  

The state space of each subsystem is unique but the subsystem’s reward-function and the action space are the same for each subsystem. The union of the state spaces of all subsystems is the state space of the system.

The discount rates, $\gamma$ and $\theta$, should be chosen near to 1 which will make the agents seek for long-term high rewards. The study of Even-Dar and Mansour [2004] has proven that choosing high discount rates approaching 1 result in faster convergence to an optimal solution.

### 6.4.3 Modifications of Policy

Normally, a RL agent seeks to develop a policy, $\pi : S \rightarrow A$, which maximises the sum of its rewards, $R = \gamma(r_0 + r_1 + \ldots + r_n)$, for an MDP that has a terminal state $s'$, or a termination condition $c$. The outputs of the IDQL algorithm are two policies that can be used by the worker agents:

- Let $M$ be a set of tutors. A policy for tutor $t_i$ of $M$ is $\pi_{t_i} : S_{i} \rightarrow A$, where $S_{i}$ is the state space of subsystem $i$.

- A policy for consultant agents is $\pi_{\text{consultant}} : S \rightarrow A$, where $S$ is the state space of the system and $A$ is the action space of the system.

The action space of any tutor and the action space of the consultant are the same in the distributed hunter prey problem.

### 6.4.4 Limitations vs Benefits

The decomposition process of DHLM is the duty of the implementation designers. It goes hand in hand with the process of the design of distributed systems. This means that all decompositions need to be predefined and have to be compatible with the distributed organisation of distributed systems. However, this decomposition process is considered an easy one when implemented to distributed structure problems.
DHLM has two advantages over dynamic decomposition algorithms. First, the decomposition process of DHLM is a static one which means that it does not require multiple simulations to define the decompositions. Second, the decomposition process of DHLM can be implemented to large learning problems provided that these problems have distributed structures.

### 6.5 Experiments

#### 6.5.1 Setup

We implemented IDQL and Q-learning to two instances of the distributed hunter prey problem with different sizes. In the first experiment, the problem is composed of 4 grids (arranged in two rows of two columns) where each grid is of size 10 × 10. In the second experiment, the problem is composed of 8 grids (arranged in four rows of two columns) where each grid is of size 20 × 20. Each grid involved 4 hunter agents and 2 prey agents. A worker hunter is allowed to migrate from one subgrid to another one when it finishes learning and there is a need for it in another subgrid.

IDQL was implemented based on two scenarios:

1. We assumed in the first scenario that all prey agents are not moving. Thus, we implemented the ambush learning at subsystem level (Section 6.3.6).

2. We assumed in the second scenario that the prey agents are mobile agents that can migrate to other local grids. Thus, we implemented ambush learning at system level (Section 6.3.6).

The location of each prey agent is determined before learning begins, and the hunter agents are randomly positioned in each subgrid for each learning episode. The hunter agents can move in four directions (up, down, right, left) unless there is an obstacle or boundary blocking the learner’s way. There are two blocker cells in each corner
of the neighbouring sub grid of the target prey, which the blocker hunters target, and three ambush cells in the centre of the neighbouring edge, which the ambush hunter targets.

The learning parameters are set as follows:

- The learning rates $\alpha = \delta = 0.4$, and the discount factors $\gamma = \theta = 0.9$ as in Arai and Sycara [2000], Jiang et al. [2007].

The reward that each chaser hunter receives is defined as:

$$R_{\text{chaser}}(s, a, s') = \begin{cases} +100.0 & \text{if it moves to the same cell as the prey} \\ 0 & \text{otherwise} \end{cases}$$

The reward that each ambusher hunter receives is defined as:

$$R_{\text{ambusher}}(s, a) = \begin{cases} +100.0 & \text{if it moves to the same cell as the prey on the ambush cells} \\ 0 & \text{otherwise} \end{cases}$$

The reward that each blocker hunter receives is defined as:

$$R_{\text{blocker}}(s, a) = \begin{cases} +100.0 & \text{if it is moving within its blocking cells} \\ 0 & \text{otherwise} \end{cases}$$

If two or more hunter agents move at the same time to the same cell where the prey agent resides, the reward is assigned randomly to one of them.

The following characteristics of the prey agents were adapted from Erus and Polat [2007]:

1. Nonadaptive: the prey agents follow a predefined action selection algorithm that maximises their distance from their chasing hunters.
2. Non cooperative: the prey agents do not cooperate to escape from hunters.

Prey agents can cooperate to escape from the hunter agents, if one or more prey agents distract the hunter agents while the other prey agents escape. This kind of cooperation requires intelligent algorithms that are not discussed or modelled in the current thesis.

The prey agents are dummy agents that move randomly. The location of each prey is determined randomly in the beginning of each episode.

Q-learning was implemented in two different modes:

1. Parallel Q-learning where the goals are learned in parallel (i.e., Q-learning is implemented to learn how to hunt all the prey at the same time).

2. Sequential Q-learning where the goals are learned in sequential order (i.e., Q-learning is implemented to learn how to hunt one prey before learning how to hunt the next).

In our experiments, the IDQL algorithm used the parallel scheduling algorithm (Figure 6.11). A learning episode ends when the hunter agents capture all the prey agents, or after 5000 moves without capturing all the prey agents.

The experiments were conducted using an Intel Xeon 3.4 GHz CPU with 16 GB RAM running 64-bit Windows. The running time of each of the algorithms for the two experiments is insignificant (less than 1 second).

6.5.2 Results and Discussion

This section compares the performance of IDQL against single agent Q-learning in the distributed hunter prey problem. In our experiments, an algorithm is said to have converged when all the prey agents are captured and the average number of moves in its policy improves by less than one move over 200 consecutive episodes.
Figure 6.20: Experiment 1: Performance of IDQL vs Q-learning in a distributed hunter prey problem composed of 4 grids. Each grid is 10 × 10.

Figure 6.20 shows the average number of moves to capture the prey agents over 1000 learning episodes in a distributed hunter prey problem composed of 4 (10 × 10) grids. We can see from the plot that the performance of IDQL algorithm at system level is similar to the performance of IDQL algorithm at subsystem level. In both cases, the IDQL algorithm converges after around 225 episodes. Parallel Q-learning converges after around 400 episodes while sequential Q-learning converges after around 780 episodes. This indicates that the performance of IDQL algorithm is better than parallel Q-learning and sequential Q-learning. As expected, parallel Q-learning converges much faster to a solution than sequential Q-learning.
Figure 6.21 shows the average number of moves to capture the prey agents over 3000 learning episodes in a distributed hunter prey problem composed of 8 (20 \times 20) grids. The results suggest that the performance of IDQL algorithm at system level is similar to the performance of IDQL algorithm at subsystem level. In both cases, IDQL converges after around 500 episodes. Parallel Q-learning converges after around 600 episodes while sequential Q-learning converges after around 2000 episodes.

The overall results of the two experiments suggest that IDQL performs better than Q-learning because each hunter agent learns a smaller subset of the overall problem, and IDQL supports migration of hunter agents. Parallel Q-learning performs better than sequential Q-learning.
6.6 Summary

This chapter described a distributed hierarchical learning model (DHLM), and an intelligent distributed Q-learning algorithm (IDQL) for solving problems with a distributed or decomposable structure. The DHLM and IDQL are based on the argument that the organisation of distributed systems into two levels (system level and subsystem level) provides an efficient decomposition of large problem spaces of distributed systems into more manageable components. The DHLM and IDQL are based on three specialisations of agents, workers, tutors and consultants. Worker agents are the actual learners and performers of tasks, while tutor agents and consultant agents are coordinators at the subsystem level and the system level, respectively.

The chapter defined four concepts related to the problem model of DHLM. First, subsystem which is an MDP that is connected to its neighbouring subsystems through connection sets. Second, connection set which is a component of subsystem that shows how a subsystem connects to another subsystem. Third, system which is a collection of loosely coupled subsystems. The loose coupling characteristic of a system means that each one of its components has or uses little knowledge about other components [Kaye, 2003]. Fourth, nested system which is a system composed of a set of systems and other nested systems.

The chapter discussed important features of the DHLM model. The model supports migration of worker agents from one problem space to another through coordination between tutors and consultants. It is also showed that the model supports division of work among worker agents, and distribution and redistribution of worker agents based on the distribution of tasks.

In DHLM model, worker agents can cooperate to accomplish a task at system and/or subsystem level. This is demonstrated using an ambush strategy in a distributed hunter prey problem. This ambush strategy supports two types of cooperation that
do not require communication among cooperative agents: cooperation at sub-system level, and cooperation at system level.

IDQL is a MARL algorithm for the DHLM model. IDQL proposes that the problem space of each tutor agent and each consultant agent can be modelled as a separate MDP: each subsystem is a problem model of one tutor agent, and the system is the problem model of consultant agents. In the experimental section of this chapter, the performance of the IDQL algorithm is compared against the performance of single agent Q-learning. The results suggest that IDQL performs faster than single agent Q-learning.

The next chapter will introduce Q-learning with Aggregation (QA-learning), an algorithm for cooperative policy construction for independent learners, based on a similar problem model to the problem model introduced in this chapter. It will be argued that each independent learner in multiple goals decomposable systems can extend its policy to cover one or more subsystems or construct a general policy for the whole system based on two cooperative roles: learner and tutor. These roles are based on the concept of learners reusing tutors’ sub-solutions.

The distributed hunter problem was used in the current chapter to evaluate the DHLM model and the IDQL algorithm. In Section 8.4, we will combine QA-learning, DHLM and IDQL in one algorithm called Enhanced QA-learning (EQA-learning). The EQA-learning algorithm will be implemented to the nearest emergency problem which is the problem of finding the nearest exit in a multiple storey building. The purpose of the EQA-learning algorithm is to show that DHLM and IDQL can be generalised to solve different learning problems.
Chapter 7

Hierarchical Cooperative Policy Construction for Independent Q-Learners

Learning problems that have large state spaces require longer training time and use more memory than smaller problems. This chapter presents Q-learning with Aggregation (QA-learning), an algorithm for cooperative policy construction for independent learners that is based on two roles: learner and tutor. This algorithm provides a solution for large problems that have multiple goal states. In this algorithm, each learner incorporates its tutors’ Q-tables into its own Q-table calculations. A comprehensive solution can then be obtained by combining these partial solutions. Experiments on an instance of the shortest path problem show that QA-learning converges to solutions faster than single-agent Q-learning.
Chapter 7. Hierarchical Cooperative Policy Construction for Independent Q-Learner

7.1 Introduction

The performance of classical RL algorithms such as Q-learning [Watkins, 1989, Watkins and Dayan, 1992] typically degrades with the increase of the size of the state space [Asadi and Huber, 2004, Sutton and Barto, 1998]. RL converges slower to an optimal solution in large state space problems compared to smaller state space problems. Fortunately, many large state space problems can be decomposed into loosely-coupled subsystems that can be processed independently [Barto and Mahadevan, 2003, Daoui et al., 2010, Ghavamzadeh et al., 2006, Hansen and Feng, 2000].

This chapter argues that each independent learner in multiple goal decomposable systems can extend its policy to cover one or more subsystems, or construct a general policy for the whole system, based on two cooperative roles: learner and tutor. A learner is an agent that learns and uses the knowledge of its neighbours (tutors) to construct its Q-table. A tutor is a learner that is ready to share its Q-table with its neighbours (learners). These roles are based on the concept of learners reusing tutors’ sub-solutions.

This chapter introduces the Q-learning with Aggregation algorithm (QA-learning), an algorithm for multiple goal problems that is based on the two roles of learner and tutor. In this algorithm, each learner incorporates its tutors’ knowledge into its own Q-table calculations. A comprehensive solution can then be obtained by combining these partial solutions.

In the experimental section of this chapter, the performance of QA-learning and single-agent Q-learning are compared in an instance of the shortest path problem (the nearest emergency exit problem). The shortest path problem is the problem of finding the shortest path between a start cell and a goal cell in a two dimensional grid. Jaakkola et al. [1995] argue that partial observable environment can be modelled as fully observable problems with the learners having partial access to the state space.
Chapter 7. Hierarchical Cooperative Policy Construction for Independent Q-Learner

The nearest emergency exit problem is treated as a fully observable problem as in Iima and Kuroe [2006, 2007, 2008]. The nearest emergency exit problem is a problem with a multiple goal-states decomposable structure where the problem space of each level of the building is a separate problem. The obtained results suggest that the output of QA-learning is comparable to the output of a single Q-learner whose problem space is the whole system. But the QA-learning algorithm converges faster to a solution than a single learner approach.

The remainder of the chapter is structured as follows: Section 7.2 discusses related work, Section 7.3 discusses a motivating example, Section 7.4 discusses the QA-learning algorithm, Section 7.5 discusses the implementation of the QA-learning algorithm to the nearest emergency exit problem, and finally Section 7.6 presents the summary of this chapter.

### 7.2 Related Work

In Section 5.4.6, we identified two common approaches for implementing the teacher-learner model in RL: an agent teacher and a human teacher. In the present chapter, we propose and implement a teacher-learner model for solving loosely coupled FMDPs. In our approach, a learner includes the knowledge of its tutors in its own Q-table calculations without changing them. The previous related studies did not apply a teacher-learner model in this context.
Chapter 7. Hierarchical Cooperative Policy Construction for Independent Q-Learner

7.3 Motivating Example: Nearest Emergency Exit Problem

The nearest emergency exit problem is the shortest path problem [Iima and Kuroe, 2007] of finding a nearest emergency exit (E) in a multiple level building. Figure 7.1 shows an abstract view of a four level building\(^1\). In the figure, the \(x\) and \(y\) coordinates are respectively the horizontal and vertical components of any room. An agent at any storey can choose its actions from the set of actions \(A = \{\text{move up, move down, move left, move right}\}\) where any action that belongs to \(A\) is a deterministic action. A deterministic action transition leads to a new state \((T : S \times A \rightarrow S)\). We consider each level of the building as a separate problem model. However, each two adjoining stories are connected by a stairway (D). If we

---

\(^1\)The example in Figure 7.1 is smaller than each of the three examples implemented in the experiments. In Section 7.5, the building in the first example is composed of 4 storeys where each storey is composed of 100 rooms while the building in the second and third examples are composed of 8 and 12 storeys, respectively, where each storey is composed of 200 rooms.
apply Q-learning to the problem model of each storey, we will be able to find the nearest exit for each storey. However, the only way to find the nearest exit in the building is by having a full map of the four storeys. For example, if an agent at level four is positioned at room [1][0], its local policy after applying Q-learning to the problem space of level four would be eight instructions: \{up,7 right\}, \{7 right,up\}, \{6 right,up,right\},..., or \{2 right,up,5 right\}. However, if the same agent has a full map of the building, its policy could be \{move down one storey\} because it will know then that the nearest exit is on the third level. Finding the nearest exit in the building is a multiple goal-states problem. In this problem, finding an exit is a single goal that can be achieved in four different ways (multiple goal-states). In this kind of problem, learning the goal requires one reward function compared to multi-goal problems that have a reward function for each goal based on its description. We will look at this problem as a distributed learning problem with a large state space, rather than a partially observable problem.

7.4 QA-learning Algorithm

We propose a new variation of the Q-learning algorithm called Q-learning by Aggregation (QA-algorithm). The motivating example described in the previous section will be used to demonstrate the basic concepts of the QA-learning algorithm.

7.4.1 Two Cooperative Roles

In the QA-learning algorithm, an independent learner can play two roles:

- A tutor, which is a provider of knowledge. The knowledge of an agent is its Q-table.

- A learner, which is an agent that incorporates its tutors’ knowledge into its own Q-table calculations, or uses that knowledge for its policy construction.


7.4.2 Problem Model

The problem model used in this chapter is the same as the problem model described in Section 6.3.1. The following is a reminder of the basic definitions of the model supported by examples.

A system is a tuple $[S, A, W, T]$, where $S$, $A$, and $T$ are defined as in an MDP (see Section 3.1) and $W$ is a set of reward functions $R : S \times A \rightarrow \mathbb{R}$ for different roles that may be used in the system. A role can then be defined as an MDP $[S, A, R, T]$, where $R \in W$.

A subsystem is a MDP with a connection set that defines the subsystem’s boundaries with its neighbouring subsystems. More formally, given a role $Role = [S, A, R, T]$, a subsystem is a tuple $Sub = [M, C]$, where:

1. $M = [S_{sub}, A_{sub}, R_{sub}, T_{sub}]$ is a MDP where:
   
   (a) $S_{sub} \subseteq S$ is the set of states in the subsystem.

   (b) $A_{sub} \subseteq A$ is the set of actions in the subsystem.

   (c) $R_{sub} : S_{sub} \times A_{sub} \rightarrow \mathbb{R}$ is a reward function such that, given $s \in S_{sub}, a \in A_{sub}, r \in \mathbb{R}, R_{sub}(s, a) = r \iff R(s, a) = r$.

   (d) $T_{sub} : S_{sub} \times A_{sub} \times S_{sub} \rightarrow [0, 1]$ is a transition function such that, given $s_i, s_j \in S_{sub}, a_k \in A_{sub}, t \in [0, 1], T_{sub}(s_i, a_k, s_j) = t \iff T(s_i, a_k, s_j) = t$.

2. $C : S_{sub} \times A \times (S \setminus S_{sub}) \rightarrow [0, 1]$ is a connection set which specifies how $Sub$ connects to other parts of the system such that, given $s_i \in S_{sub}, a \in A, s_j \in S \setminus S_{sub}, t \in [0, 1], C(s_i, a, s_j) = t \iff T(s_i, a, s_j) = t$.

Example 7.1. Each storey of the four storey building in the nearest emergency exit problem (Figure 7.1) represents a subsystem that is composed of:
1. An MDP, that is composed of:

(a) A set of possible states, \( S_i = \{\text{room}_0, \text{room}_1, ..., \text{room}_{n-1}\} \).

(b) A set of possible actions, \( A_i = \{\text{move up}, \text{move down}, \text{move left}, \text{move right}\} \).

(c) The reward that each agent receives is defined as:

\[
R(s, a, s') = \begin{cases} 
+100.0 & \text{if it reaches an exit state} \\
-1 & \text{if it goes down or up one level} \\
0 & \text{otherwise}
\end{cases}
\]

The goal is to find the nearest exit in a building with multiple exits. Thus, there is a single reward function because we have a single goal with multiple states.

(d) The transition model is:

\[
T(s, a, s) = \begin{cases} 
1 & \text{if } s' \text{ is adjacent to } s \text{ and is a valid position for the agent} \\
0 & \text{otherwise}
\end{cases}
\]

In the above model, collision is not modelled because multiple learners in the nearest emergency exit problem learn how to find the nearest exit for a single-agent. Thus, collision is not possible.

(e) The connection set is the rooms directly connected to each other through stairways. The set of connection actions that connect a storey to its adjoining storeys is \( \{\text{go up one level}, \text{go down one level}\} \).

2. A connection set \( C_i : S_i \times A \times (S \setminus S_i) \rightarrow [0, 1] \), where \( S \) is the set of all possible states on the whole building.

For our better understanding of the concept of connection set, let us consider the following situation in the nearest emergency exit problem. If we assume that each
agent has applied Q-learning to its subsystem (Figure 7.1), then each learner can start another round of learning in which it incorporates its tutors’ knowledge into its own Q-table calculations. For example, Learner$_1$ at the third level can start another round of learning in which it can use the knowledge of the tutors at the second and the fourth levels. Learner$_1$ can connect the Q-tables of Tutor$_1$ and Tutor$_2$ to its Q-table using the following connection sets:

- Learner$_1$ can connect the Q-table of Tutor$_1$ using the following instructions:
  
  \[ \text{(cell}[2][0], \text{go up one storey)} \rightarrow \text{cell}[1][0], \text{ (cell}[1][0], \text{go down one storey)} \rightarrow \text{cell}[2][0]. \]

- Learner$_1$ can connect the Q-table of Tutor$_2$ using the following instructions:
  
  \[ \text{(cell}[3][0], \text{go down one level)} \rightarrow \text{cell}[4][0], \text{ (cell}[4][0], \text{go up one level)} \rightarrow \text{cell}[3][0]. \]

The model information of the building should be available in order for any learner to connect its Q-table with its tutors’ Q-tables. In the above example, the Q-table of Learner$_1$ is triple its original size after including the Q-tables of Tutor$_1$ and Tutor$_2$.

### 7.4.3 QA-learning Algorithm

The QA-learning algorithm is composed of two learning stages. First, learning of each subsystem individually. Second, learning by aggregation between subsystems.

We can argue that there are two approaches to modelling the learning of multiple goals. First, a single Q-table for all the goals (Figure 7.2a). Second, a Q-table for each goal in the system (Figure 7.2b). Both approaches require a long training time because their problem model is the whole system.
Our approach to this problem is to learn one goal at a time at subsystem level, then reuse these solutions to compose a general solution or append the solution of each of a subsystem’s neighbours to its own solution. The two learning stages of QA-learning are:

- **First Learning Stage**: Q-learning is applied to each subsystem individually. Figure 7.3 shows that the inputs for the first learning stage are \( n \) subsystems and the outputs are \( n \) Q-tables.

- **Second Learning Stage**: This stage takes place after the end of the first stage. In this stage, each agent incorporates the Q-tables of its tutors into...
its new Q-table calculations. Figure 7.3 shows that the inputs for the second learning stage are \(n\) Q-tables (outputs of the first learning stage), and \(n\) subsystems, and the outputs are \(n\) revised Q-tables, and a derived general Q-table.

Figure 7.4 shows the aggregative algorithm implemented by each learner in the second learning stage (Knowledge Based Q-learning of Figure 7.3). This algorithm is a modified Q-learning algorithm that uses knowledge of surrounding subsystems. In this algorithm, the Q-tables of the learner's tutors are merged into the learner's Q-table (Line 1). After this, the learner applies the Q-learning algorithm for \(\epsilon\) episodes for each state that belongs to its connection sets (Lines 13 to 14). The purpose is to connect the learner Q-table to its tutors’ Q-tables through few exploratory learning episodes.

1: Append knowledge of the Q-table of each tutor connected via the learner’s connection set to the learner’s Q-table.
2: for each \(s\) in the connection set of the learner do
3:   for episode= 1 to \(\epsilon\) number of episodes do
4:     while \(s\) is not a terminal state do
5:       Choose an action \(a\) from the action set of \(s\) \((A)\) or the set of connection actions \((C_a)\) using a policy derived from \(Q\)
6:       Take action \(a\)
7:       Receive \(r, s'\)
8:       Update the Q-table as follows:
9:       \[Q_L(s, a) \leftarrow (1 - \beta)Q_L(s, a) + \beta[R_L(s, a) + \gamma V_L(s)],\]
10:      \[V_L(s) \leftarrow \max_{a\in A}Q_L(s, a), \text{ where } s \in S, a \in A \text{ or } a \in C_a, \text{ the learning rate } \beta \in [0, 1] \text{ and the discount factor } \gamma \in [0, 1].\]
11:     \(s \leftarrow s'\)
12:   end while
13: end for
14: end for

Figure 7.4: The second stage of the QA-learning algorithm.
Constructing a General Policy for the Whole System

Following the two learning stages of QA-learning allows each learner to incorporate the knowledge of its neighbouring subsystems into its own Q-table calculations. However, this does not produce a general solution for the whole system. All the solutions to the sub-systems should be combined to give a solution to the whole system.

If we combine all the subsystems at once (system integration), the instruction “Append the connection sets of the tutors to the learner’s subsystem” should be inserted after the first line of the QA-learning algorithm (Figure 7.4) to allow the learner to connect to all the subsystems, even the ones that are not neighbouring it. For example, the solutions of level1 and level3 on the nearest emergency exit problem can be connected through the solution of level2, and the solutions of level2 and level4 can be connected through the solution of level3. If we combine the subsystems incrementally (incremental integration), the same instruction should be inserted after line fourteen of the QA-learning algorithm (Figure 7.4) with the resulting solution inheriting the connection sets of the merged sub-solutions. For example, when solution1 and solution2 are integrated, the resulting solution should inherit the connection sets of the merged solutions to allow connection to solution3.

Figure 7.5 shows an example of the merge procedure that can be used to combine the Q-tables of all of the agents. This procedure is an interpretation of Line 1 of Figure 7.4 for an instance of the nearest emergency problem composed of $n$ storeys ($n = 4$). In the beginning, we defined a new Q-table for the learner (learnerNewQTable) that can fit the whole system (Line 1), then we copied the Q-values of the old Q-table of the learner (learnerOldQTable) into learnerNewQTable (Line 6). Lines 7 to 9 show three calls for the function combineQtables. This function takes the parameters tutoriQtable, learnerNewQTable and the connection point to learnerQTable. The function combineQtables copies the Q-values of each tutoriQtable to learnerNewQTable (Lines 11 to 17).
**Chapter 7. Hierarchical Cooperative Policy Construction for Independent Q-Learner**

1. learnerNewQTable = new double[subSystem0.length*n][System0[length]
2. tutor1Qtable = new double[subSystem1.length][subSystem1[0].length]
3. tutor2Qtable = new double[subSystem2.length][subSystem2[0].length]
4. tutor3Qtable = new double[subSystem3.length][subSystem3[0].length]
5. Begin
6. combineQtables(learnerNewQTable, learnerOldQtable, 0, 0)
7. combineQtables(learnerNewQTable, tutor1Qtable, subSystem1.length, 0)
8. combineQtables(learnerNewQTable, tutor2Qtable, 2*subSystem2.length, 0)
9. combineQtables(learnerNewQTable, tutor3Qtable, 3*subSystem3.length, 0)
10. End

| Private void combineQtables(double[][] learnerNewQTable, double[][] tutorQtable, int iB, int jB) |
| Begin |
| for (int i1 = 0, i2 = iB; i1 < subQtable.length; i1++, i2++) do |
| for (int j1 = 0, j2 = jB; j1 < subQtable.length; j1++, j2++) do |
| learnerNewQTable[i2][j2] = tutorQtable[i1][j1] |
| end for |
| end for |
| End |

**Figure. 7.5:** An example of the Merge procedure of QA-learning.

### 7.5 Experiments

#### 7.5.1 Setup

The QA-learning and Q-learning algorithms were applied to three instances of the nearest emergency exit problem (Section 7.3) with different sizes. In the first experiment, the building is composed of 4 storeys where each storey has 100 rooms. In
the second experiment, the building is composed of 8 storeys where each storey has 200 rooms. In the third experiment, the building is composed of 12 storeys where each storey has 200 rooms. The doors and exits have the same positions as shown in Figure 7.1, i.e. the exits are only in the first and the last column, but the floors are longer (instance one: 10 rows of 10 rooms per storey; experiments two and three: 10 rows of 20 rooms per storey).

In these experiments, the learning rate $\alpha$ and $\beta$ are set to 0.4 and the discount factor $\gamma$ is 0.9 for both the Q-learning algorithm and the QA-learning algorithm as in [Arai and Sycara, 2000, Jiang et al., 2007]. We chose a high discount factor for the two stages of the QA-learning algorithm because it allows the learners to target high future rewards and converge to optimal solutions [Sutton and Barto, 1998]. The selection policy of actions for both algorithms is a random policy.

An agent that goes down or up one level using the stairs is punished because climbing stairs requires more time than moving between the rooms. The reward function of each agent was defined earlier in Section 7.1.

### 7.5.2 Results and Discussion

This section compares the performance of the QA-learning algorithm against the Q-learning algorithm. QA-learning was implemented for the nearest emergency exit problem using the three proposed integration scenarios described in Section 7.4.3. While Q-learning was implemented to the nearest emergency exit problem for the whole building. In our experiments, an algorithm is said to have converged when the average number of moves in its policy improves by less than one move over 200 consecutive episodes.
Figure 7.6: Experiment 1: QA-learning with three integration scenarios. The building is a 4 storey building where each storey is composed of 100 rooms.

Figure 7.6 shows the average number of moves per 25 episodes of 4 QA-learners required to reach the nearest exit over 500 episodes in a 4-storey building. In this experiment, each QA-learner converges to a solution for its storey after around 100 episodes of learning, which marks the end of the first learning stage of QA-learning. The three integration scenarios were implemented after 120 episodes. The results show that QA-learning converges to a solution after around 250 episodes regardless of the implemented integration scenarios.

Figure 7.7 shows the average number of moves per 25 episodes of a single Q-learner required to reach the nearest exit over 1000 episodes in a 4-storey building. We can see from the figure that Q-learning converges to a solution after around 800 episodes spread uniformly all over the building. This is more than the training episodes required in QA-learning. QA-learning performs faster than Q-learning because QA-learning allows multiple learners to learn in parallel then combines their solutions.
through few learning episodes. Even if we do not have parallel learning, the total number of learning episodes\(^2\) required to converge to a solution \((120 \times 4 + 130 = 610)\), is still less 76% than the number of episodes required for single-agent Q-learning \((800)\). Further, since the learners in the QA-learning scenario are learning smaller subsets of the overall problem, their individual learning episodes are quicker.

\[\text{Episodes} \quad \text{Average number of moves} \]

\[\begin{array}{c}
0 & 0 \\
200 & 200 \\
400 & 150 \\
600 & 100 \\
800 & 50 \\
1000 & 0 \\
\end{array}\]

**Figure. 7.7:** Experiment 1: Single agent Q-learning implemented for the nearest emergency exit problem for the whole building. The building is a 4 storey building where each storey is composed of 100 rooms.

\(^{2}\text{Duration of the first stage of QA-learning} \times \text{the number of the learners} + \text{duration of the second learning stage until convergence to a solution}\)
Figure 7.8: Experiment 2: QA-learning with three integration scenarios. The building is an 8 storey building where each storey is composed of 200 rooms.

Figure 7.8 shows the average number of moves per 25 episodes of 8 QA-learners required to reach the nearest exit over 700 episodes in an 8 storey building. In Figure 7.8, the first learning stage of QA-learning ends at around 140 episodes. The three integration scenarios were implemented after 150 episodes. The results show that QA-learning converges to a solution after around 480 episodes. Each of the integration scenarios produces similar results to each other when implemented with QA-learning.
Figure 7.9: Experiment 2: Single agent Q-learning implemented for the nearest emergency exit problem for the whole building. The building is an 8 storey building where each storey is composed of 200 rooms.

Figure 7.9 shows the average number of moves per 25 episodes of a single Q-learner required to reach the nearest exit over 3000 episodes in an 8 storey building. The figure shows that single-agent Q-learning converges to a solution after around 2200 episodes. This is more than the training time required for QA-learning in Figure 7.8. If QA-learning in Figure 7.8 does not support parallel execution of learners, its total number of learning episodes required to converge to a solution is $150 \times 8 + 330 = 1530$. This number is still less 70% than the number of episodes required for single-agent Q-learning (2200).

Figure 7.10 shows the average number of moves per 25 episodes of 12 QA-learners required to reach the nearest exit over 1000 episodes in a 12 storey building. In this experiment, QA-learning converges to a solution after around 600 episodes (stage
one: 150, stage two: 450) with any of the three integration procedures. If the learners at the first stage execute sequentially, the total number of episodes required to converge to a solution is $150 \times 12 + 450 = 2250$.

![Figure 7.10: Experiment 3: QA-learning with three integration scenarios. The building is a 12 storey building where each storey is composed of 200 rooms.](image)

The results in Figure 7.11 shows that Q-learning performs worse than QA-learning in Figure 7.10. The Q-learning algorithm converges to a solution after around 3000 episodes which is worse 75% than the results of sequential QA-learning.
Chapter 7. Hierarchical Cooperative Policy Construction for Independent Q-Learner

Figure 7.11: Experiment 3: Single agent Q-learning implemented for the nearest emergency exit problem for the whole building. The building is a 12 storey building where each storey is composed of 200 rooms.

<table>
<thead>
<tr>
<th></th>
<th>Experiment 1</th>
<th>Experiment 2</th>
<th>Experiment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel QA-learning</td>
<td>31%</td>
<td>22%</td>
<td>20%</td>
</tr>
<tr>
<td>Sequential QA-learning</td>
<td>76%</td>
<td>70%</td>
<td>75%</td>
</tr>
</tbody>
</table>

Table 7.1: The ratio of the number of episodes in QA-learning to the number of episodes in Q-learning.

The overall results of the experiments suggest that QA-learning performs faster than single agent Q-learning even if we do not have parallel learning. When we increased the task size (Table 7.1), the performance difference in the parallel execution case of QA-learning became even larger (see row 2 of Table 7.1); in Experiment 1 QA-learning required 31% of the episodes of single agent Q-learning, in Experiment 2 QA-learning required only 22% of the episodes of single agent Q-learning, and in Experiment 3 QA-learning required only 20% of the episodes of single agent Q-learning. Sequential QA-learning reduces the required learning episodes of single
agent Q-learning because the learners in the first stage of QA-learning learn smaller subsets of the original problem. This also means that the average length of an episode in QA-learning is shorter than the average length of an episode in single agent Q-learning. QA-learning produces similar results when implemented following any of the three integration scenarios.

<table>
<thead>
<tr>
<th></th>
<th>Experiment 1</th>
<th>Experiment 2</th>
<th>Experiment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q-learning</td>
<td>3021</td>
<td>39400</td>
<td>65336</td>
</tr>
<tr>
<td>QA-learning</td>
<td>833</td>
<td>1233</td>
<td>1423</td>
</tr>
</tbody>
</table>

Table 7.2: The running time of Q-learning vs the running time of QA-learning in milliseconds.

Table 7.2 shows the running time of Q-learning vs the running time of QA-learning for the three experiments in milliseconds. Row 1 of Table 7.2 shows that the running time of Q-learning increases with increase of the size of the problem, while row 2 of Table 7.2 shows that the running time of QA-learning is much faster for each of the three experiments. This is because the learners in the QA-learning scenario are learning in parallel and each of them is learning a smaller subset of the overall problem.

<table>
<thead>
<tr>
<th></th>
<th>Experiment 1</th>
<th>Experiment 2</th>
<th>Experiment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1 of QA-learning</td>
<td>11.9</td>
<td>18.6</td>
<td>20.5</td>
</tr>
<tr>
<td>Stage 2 of QA-learning</td>
<td>8.25</td>
<td>14.3</td>
<td>14.2</td>
</tr>
</tbody>
</table>

Table 7.3: The average number of moves after applying Stage 1 of QA-learning vs the average number of moves after applying Stage 2 of QA-learning.

Table 7.3 shows the average number of moves after applying Stage 1 of QA-learning vs the average number of moves after applying Stage 2 of QA-learning. Row 2 of Table 7.3 shows that the policies that resulted after applying Stage 2 of QA-learning are much better than the ones resulted after applying Stage 1 of QA-learning (Row 1). This is because the learners in Stage 2 of QA-learning discovered that the nearest exits for some of their rooms are in their tutors’ storeys.
In our experiments, the problem instances are conveniently chosen, so that the global solution differs little from the concatenation of the local ones. In fact, only those states near the connecting stairways will be affected in terms of Q-values and policy. However, this fact does not contradict with the main purpose of the design of the QA-learning algorithm which is reducing the convergence time of large state-space problems by decomposing them into small state-space problems.

### 7.6 Summary

This chapter described the QA-learning, an algorithm for cooperative policy construction for independent learners that is based on the two roles of learner and tutor. A learner is an agent that learns and uses the knowledge of its neighbours (tutors) to construct its Q-table. A tutor is a learner that is ready to share its Q-table with its neighbours (learners). These roles are based on the concept of learners reusing tutors’ sub-solutions.

The chapter demonstrated that in QA-learning, each agent incorporates its tutors’ Q-tables into its own Q-table calculations. A comprehensive solution can then be obtained by combining these partial solutions. Informing the learner of the existence of other goals around it, and the way it should use them, increases the accuracy of each learner’s Q-table.

The chapter also described the problem model of the QA-learning algorithm which is the same as the problem model described in the previous chapter which represents large state problems using hierarchical decomposition into subsystems. The chapter presented two solution scenarios for the QA-learning algorithm. First, each learner incorporates the knowledge of its neighbouring subsystems into its own Q-table calculations. Second, all the solutions to the sub-systems are combined to give a solution to the whole system.
In the experimental section of this chapter, we implemented the QA-learning and the single agent Q-learning algorithms to three instances of the shortest path problem. The experimental results suggest that QA-learning performs faster than conventional Q-learning, even if we do not have parallel learning. Further, the average length of an episode in QA-learning is shorter than the average length of an episode in single-agent Q-learning. QA-learning converges to a solution regardless of the choice of the integration procedure (system or incremental integrations).

The Q-learning and the QA-learning algorithms should be provided with the model information of the system that shows how each subsystem connects to its adjoining subsystems. Our purpose of implementing the QA-learning algorithm is to reduce the running time required to converge to a policy by reducing the problem space into smaller manageable subsystems. In the end of the learning process, Q-learning and QA-learning converge to similar policies, but the running time of QA-learning is much faster than the running time of Q-learning. A disadvantage of QA-learning is that it requires the decomposition of the problem space to be provided by the implementation designer. However, many learning problems have clear naturally decomposable structures that can be modelled for the QA-learning algorithm.

In cooperative Q-learning for independent learners, sharing of Q-values among independent learners takes place after the single agent learning stage. In the single agent learning stage, each learner learns independently from the other learners implementing its own Q-learning algorithm. The next chapter will compare the performance of some of the best known cooperative Q-learning algorithms (BEST-Q, AVE-Q, POS-Q, and WWS), and single agent Q-learning. The main purposes of the next chapter is to classify these cooperative algorithms based on their performance and study the effects of the frequency of Q-values sharing on the speed of convergence to optimal solutions. The QA-learning algorithm will not be compared with the cooperative Q-learning algorithms in the next chapter because QA-learning does not implement
a sharing strategy. The next chapter will present an approach that combines IDQL, DHLM, QA-learning and BEST-Q in one algorithm.
Chapter 8

Cooperative Q-learning

Algorithms for Independent Learners

Cooperative reinforcement learning algorithms such as BEST-Q, AVE-Q, PSO-Q [Iima and Kuroe, 2006, 2007, 2008], and WSS [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002] algorithms that use Q-value sharing strategies between reinforcement learners are known to accelerate the learning process of individual learners [Eshgh and Ahmadabadi, 2002, Galindo-Serrano et al., 2010, Iima et al., 2011]. This chapter presents a comparison study of the performance of these cooperative algorithms as well as an algorithm that aggregates their results. In addition, this chapter studies the effects of the frequency of Q-value sharing on the learning speed. This chapter also presents an approach that combines IDQL, DHLM, QA-learning and BEST-Q in one algorithm.
8.1 Introduction

In cooperative Q-learning, multiple independent learners learn the same task over the entire state space. The learning process of cooperative Q-learning normally takes place in two stages. First, the individual learning stage, where each learner independently uses its own Q-learning algorithm to improve its solution. Second, the learning by interaction stage, in which a Q-value sharing strategy is implemented. A Q-value sharing strategy allows the independent learners to share their Q-values and use this information to obtain new Q-tables. Sharing of Q-values between reinforcement learners accelerates the learning process of individual learners.

Some of the best known cooperative Q-learning algorithms are BEST-Q, AVE-Q, PSO-Q [Iima and Kuroe, 2006, 2007, 2008], and WSS [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002]. Until now there have been no studies that compare the performance of these algorithms and study the effect of the frequency of Q-value sharing on the learning speed. This chapter presents a comparison study of these algorithms as well as an algorithm that aggregates their results. The study compares the results in two cases. In case one, the learners have equal experience, and in case two, the learners have different experiences. In addition, this chapter studies the effects of the frequency of Q-value sharing on the learning speed. For example, does sharing of Q-values after each learning episode have benefits over sharing after each hundred learning episodes?.

The QA-learning algorithm that was proposed in the previous chapter will not be compared with the mentioned above cooperative Q-learning algorithms. This is because QA-learning does not have a Q-value sharing strategy and our main purpose in the current chapter is to compare some famous Q-value sharing strategies. In the end of the current chapter, we will present an approach that combines IDQL, DHLM, QA-learning and BEST-Q in one algorithm.
Chapter 8. *Cooperative Q-learning Algorithms for Independent Learners*

The remainder of the chapter is structured as follows: Section 8.2 discusses four of the best known cooperative Q-learning algorithms, Section 8.3 discusses the implementation of these algorithms to the hunter prey problem, Section 8.4 presents an approach that combines IDQL, DHLM, QA-learning and BEST-Q in one algorithm, and Section 8.6 is the summary of the chapter.

### 8.2 Q-value Sharing Strategies

In this section, we will discuss the second learning stage (learning by interaction) of the following cooperative Q-learning algorithms that were discussed in Section 5.4.2 and Section 5.4.6: BEST-Q, AVE-Q, PSO-Q, WSS, and an algorithm that aggregates their results.

#### 8.2.1 BEST-Q

In the second learning stage of BEST-Q [Iima and Kuroe, 2006, 2007, 2008], the best Q-value for each state-action pair is selected from the Q-tables of all of the learners. Then, each learner updates its Q-table by replacing each one of its Q-values with the corresponding best Q-value:

\[ Q_i(s, a) \leftarrow Q_{best}(s, a) \quad (\forall i, s, a), \]

where \( i \) is the agent identification number. (8.1)

A disadvantage of this approach is that the optimal Q-values may not be found because the Q-values of all of the agents become the same after each update [Iima and Kuroe, 2006].
8.2.2 AVE-Q

AVE-Q [Iima and Kuroe, 2006, 2007, 2008] is similar to BEST-Q except that each learner updates its Q-values by averaging its current Q-value and the best Q-value for each state-action pair from the Q-tables of all of the learners:

\[ Q_i(s,a) \leftarrow \frac{Q^{\text{best}}(s,a) + Q_i(s,a)}{2} \quad (\forall i, s, a), \]

where \( i \) is the agent identification number. \( (8.2) \)

8.2.3 PSO-Q

The PSO-Q [Iima and Kuroe, 2006, 2007, 2008] algorithm uses the Particle Swarm Optimisation (PSO) method to find a global optimal solution. PSO is an optimisation method that repetitively improves a candidate solution regarding a qualitative measure [Niu et al., 2013]. PSO solves decision problems that have multiple decision variables. In PSO, a swarm is a collection of particles (candidate solutions) [Iima and Kuroe, 2007]. Let \( D \) be a decision problem of \( n \) decision variables \( y = \{y_1, y_2, \ldots, y_n\} \) that minimise an objective function \( f(y) \), and the size of the swarm for \( D \) be composed of \( m \) particles \( \{p^1, p^2, \ldots, p^m\} \), then particle \( p^i \) of the swarm at iteration \( t \) is \( y^i(t) = (y^i_1(t), y^i_2(t), \ldots, y^i_n(t)) \). The following two functions determine the candidate solution of \( p^i \) at the next iteration \( t + 1 \):

\[
v^i_n(t + 1) \leftarrow wv^i_n(t) + c_1 r_1 [p^i_n(t) - y^i_n(t)] + c_2 r_2 [g_n(t) - y^i_n(t)] \quad (8.3)
\]

\[
y^i_n(t + 1) \leftarrow y^i_n(t) + v^i_n(t + 1) \quad (8.4)
\]

Where \( v^i(t) \) is the velocity vector of \( p^i \) at iteration \( t \), \( w, c_1, \) and \( c_2 \) are weight parameters, \( r_1 \), and \( r_2 \) are random numbers between 0 to 1, \( p^i \) is the personal best solution of particle \( i \), and \( g(t) \) is the best solution found by all particles.
In PSO-Q, the RL problem is modelled as an optimisation problem in which the solution candidate and qualitative measure are respectively the Q-values and an evaluation function of the Q-values (Q-function). In PSO-Q, the best Q-values of each learner and the best global Q-values of all of the learners are used by each learner to update its Q-table.

8.2.4 WSS

In WSS [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002], each learner assigns a weight to the Q-tables of each other learner based on each learner’s relative expertness. Then, each learner uses the weighted average of all of the Q-table values to update its own Q-table:

\[
Q_i(s, a) \leftarrow \sum_{j=1}^{n} (W_{ij}Q_j(s, a)), \text{ where } W_{ij} \text{ is the weight learner } i \text{ assigns to learner } j’s \text{ expertness.} \quad (8.5)
\]

In the above function the weight \(W_{ij}\) is a function of the agents’ relative expertness. There are several expertness measures that have been shown to have similar outcomes when used in the WSS [Ahmadabadi and Asadpour, 2002]. One of these measures is the Normal measure (Nrm) which is defined as the sum of the rewards that a learner receives during the individual learning stage [Ahmadabadi and Asadpour, 2002]:

\[
e_{i}^{Nrm} = \sum_{t=1}^{now} r_i(t) \quad (8.6)
\]
where \( r_i(t) \) is the reward of agent \( i \) at instant \( t \). The formula for assigning a weight to agent \( j \)'s knowledge by learner \( i \), when using the knowledge of all agents is\(^1\):

\[
W_{ij} \leftarrow \frac{e_i}{n} \sum_{k=1}^{n} e_k
\]  

(8.7)

where \( n \) is the number of agents, and \( e_k \) is the expertness amount of agent \( k \).

### 8.2.5 Aggregate Sharing Strategy

The cooperative Q-learning algorithms BEST-Q, AVE-Q, PSO-Q, and WSS exhibit different behaviour depending on their sharing strategies. This motivated us to combine the sharing strategies of BEST-Q, AVE-Q, PSO-Q, and WSS in a single sharing strategy. In the rest of this chapter, we will refer to the Q-learning algorithm that uses this strategy as average aggregation Q-learning.

In average aggregation Q-learning, each agent updates its Q-values by averaging the Q-values from each of the BEST-Q, AVE-Q, PSO-Q, and WSS algorithms for each state-action pair:

\[
Q_i(s,a) \leftarrow \frac{Q_{best}^{\text{best}}(s,a) + Q_{average}^{\text{average}}(s,a) + Q_{WSS}^{WSS}(s,a) + Q_{PSO}^{PSO-Q}(s,a)}{4} \quad (\forall i, s, a),
\]  

(8.8)

where \( i \) is the agent identification number, and \( 4 \) is the number of the sharing strategies.

### 8.3 Experiments 1

In this section, the classical hunter prey problem (Section 2.1) will be used to compare the performance of four cooperative Q-learning algorithms. In the hunter-prey

\(^1\)Several weight assigning mechanisms can be found in Ahmadabadi et al. [2006], Ahmadabadi and Asadpour [2002].
Chapter 8. *Cooperative Q-learning Algorithms for Independent Learners*

problem, there are normally two types of agents, hunter (H) and prey (P), that can move in four directions (up, down, right, left) unless there is an obstacle, another agent, or boundary blocking the agent’s way. In this game, hunters chase the prey, and the prey try to escape from the hunters.

The cooperative Q-learning algorithms described in Section 8.2 and the single agent Q-learning algorithm were applied to three instances of the hunter-prey that have different grid sizes: $5 \times 5$, $10 \times 10$, and $20 \times 20$. Each instance involved four players: three hunters and one prey.

The performance of these algorithms were compared against each other following experimental models similar to the models of Ahmadabadi and Asadpour [2002].

There were two variations of the experiments. In the first variation, the learners had the same experience at each learning by interaction stage, which was accomplished by having each learner complete the same number of learning episodes in the first stage of cooperative Q-learning. In the second variation, the learners had different levels of experience, which was modelled by having each learner complete a different number of episodes in each first stage of cooperative Q-learning. For example, a learner that has spent 100 episodes learning has more experience than a learner that has only spent 50.

The reward that each hunter receives is defined as:

$$R_{\text{Learner}}(s, a) = \begin{cases} +5.0 & \text{if it captures the prey} \\ -0.01 & \text{otherwise} \end{cases}$$

The selection policy of actions is the Softmax selection policy (Section 4.8). The values of the learning parameters for the experiments are as follows:

- The learning rate $\alpha = 0.01$, and the discount factor $\gamma = 0.9$ in the Q-learning algorithm as in Crook and Hayes [2003].
• The position of each hunter agent is chosen randomly at the beginning of each episode. A learning episode ends when a hunter agent is in the same position with the prey agent, or after 1000 moves without capturing the prey.

• The temperature $T = 0.4$ in the softmax selection policy as in Ahmadabadi and Asadpour [2002].

• The expertness measure of WSS is the Nrm measure (Section 8.2.4).

• The weights$^2$ in PSO-Q: $W=0$, $C_1=C_2=1$.

In our experiments, an algorithm is said to have converged when the average number of moves in the hunter’s policy until termination of the episode improves by less than one move over 200 consecutive episodes. The experiments were conducted using an Intel Xeon 3.4 GHz CPU with 16 GB RAM running 64-bit Windows.

8.3.1 Equal Levels of Experience

In this set of experiments, three hunter agents learn for the same number of learning episodes before sharing of their Q-values. The effects of the frequency of information sharing on the performance of cooperative learners are tested in three cases. In the first case, the learners share their Q-values after each learning episode. In the second case, the learners share their Q-values after each 10 episodes, and, in the third case, the learners share their Q-values after each 100 learning episodes.

Equal Experience in $5 \times 5$ Grid

Figures 8.1, 8.2, and 8.3 show the average number of steps to hunt the prey agent over 500 learning episodes in a $5 \times 5$ grid. Figure 8.1 shows that each of the cooperative Q-learning algorithms has similar performance to the performance of single agent Q-learning, when sharing of Q-values takes place after each learning episode.

$^2$The weights: $W=0$, $C_1=C_2=2.2$ have been used in [Iima and Kuroe, 2006]. These values caused poor performance of PSO-Q when we implemented PSO-Q in our experiments.
The results in Figure 8.2 shows that sharing of Q-values after each 10 learning has little effect on the learning speed. These results are similar to Figure 8.1. The results
shown in Figure 8.3 (sharing of information after 100 episodes) cannot be used to compare the performance of the cooperative Q-learning algorithms because most of these algorithms converge to solutions after around 80 episodes.

Although the size of the grid ($5 \times 5$) is small, we can conclude from the figures that each of the cooperative Q-learning algorithms has little effect on the learning outcome when compared to single-agent Q-learning (no sharing case). This indicates that sharing of Q-values is inefficient in the equal experience case. Furthermore, sharing of Q-values among different learners should take place before they converge to solutions in order to compare their performances.

**Equal Experience in $10 \times 10$ Grid**

The figures in this section show the average number of steps to capture the prey agent over 500 learning episodes in a $10 \times 10$ grid. Sharing of Q-values takes place after three periods: after each learning episode (Figure 8.4), after each 10 episodes (Figure 8.5), and finally after each 100 episodes (Figure 8.6).
Figure 8.4: Average number of moves per one episode in 10 × 10 grid.

Figure 8.5: Average number of moves per 10 episodes in 10 × 10 grid.
Each of the algorithms in Figures 8.4 and 8.5 converges after around 180 episodes to a solution except BEST-Q. BEST-Q requires around 420 episodes in Figure 8.4 and around 190 episodes in Figure 8.5 to converge to a solution.

In Figure 8.6, AVE-Q, PSO-Q, and WSS algorithms converge after around 340 to solutions, while average aggregation, BEST-Q, and single agent Q-learning converge after around 200 episodes to solutions. This is because AVE-Q at each learning by interaction stage only moves halfway between its current value and the best value found so far, PSO-Q relies heavily on its initial parameters and may not search outside its global best neighbourhood, and WSS sometimes leads to wrong moves, if the Q-values of the learners vary widely.

As expected, the results of this experiment suggest that sharing of Q-values in equal experience case does not speed up the learning process of individual learners. Also, high-frequency sharing of Q-values accelerates the convergence to solutions compared to less frequent sharing except in BEST-Q. In BEST-Q, better results can
be obtained by allowing the individual learners to train for sufficient period of time before sharing their Q-values.

**Equal Experience in 20 × 20 Grid**

The figures in this section show the average number of steps to capture the prey agent over 1000 learning episodes in a 20 × 20 grid.

From Figure 8.7, we can see that most of the algorithms show similar performance when the learners share their Q-values after each episode. Each algorithm converges to a solution at around 300 episodes. The only exception is BEST-Q which requires more training to converge to a solution because the optimal Q-values may not be found because the Q-values of all of the agents become the same after each time the Q-values are shared [Iima and Kuroe, 2006].

![Figure 8.7: Average number of moves per one episode in 20 × 20 grid.](image)
The results in Figure 8.8 are similar to the results in Figure 8.7. Each algorithm converges to a solution at around 320 episodes except BEST-Q which converges at around 420 episodes. But BEST-Q in Figure 8.8 performs much better when sharing is after each 10 episodes of independent learning (less frequent sharing).

We can see from Figure 8.9 that each algorithm converges to a solution at around 400 episodes\(^3\). AVE-Q initially learns slower than the other shown cooperative algorithms because at each learning by interaction stage it only moves halfway between its current value and the best value found so far. PSO-Q, WSS, and average aggregation also learn more slowly than BEST-Q or single agent learning. This is because PSO-Q relies on its initial parameters and may not search outside its global best neighbourhood, and WSS and average aggregation sometimes lead to wrong moves if the Q-values of the learners vary widely.

\(^3\)The x axis in this figure was trimmed after 600 episodes because the performance had stabilised after this point.
The overall results suggest that sharing of Q-values is not efficient in the equal experience case. In addition, sharing of Q-values in BEST-Q should take place after a sufficient number of episodes so its performance would not degrade.

**Results Summary of Equal Experience Case**

The overall results of all the experiments in the equal experience case suggest that sharing of Q-values has little effect on the learning speed of individual learners, and the cooperative Q-learning algorithms perform similarly to each other. Also, high-frequency sharing of Q-values accelerates the convergence to a solution compared to less frequent sharing except in BEST-Q. A problem with BEST-Q is that the Q-values of all of the agents become the same after each sharing session which means that the optimal Q-values might not be found [Iima and Kuroe, 2006]. This problem is more extreme when the learners share their Q-values so frequently that the individual learners have little chance to improve their own values. But the
performance of BEST-Q can be enhanced by allowing the learners to train for a sufficient number of episodes before sharing their Q-values.

8.3.2 Different Experiences

In this set of experiments, each of the three hunter agents learns for a different number of learning episodes in each first stage of cooperative Q-learning. The frequency of Q-value sharing is tested in two cases. In case one, the first, the second, and the third hunter agents learn for 10, 5 and 1 episodes, respectively before sharing. In case two, the first, the second, and the third hunter agents learn for 100, 50 and 25 episodes, respectively. The total number of learning episodes is 1000 and the cooperation time is after the individual learning stage.

Different Experiences in $5 \times 5$ Grid

Figures 8.10 and 8.11 show the average number of steps to hunt the prey agent over 500 learning episodes in a $5 \times 5$ grid. Figure 8.10 shows the results of case one of Q-value sharing while Figure 8.11 shows the results of case two of sharing.

Each of the cooperative learners in Figure 8.10 converges faster to a solution than single-agent Q-learning. However, Best-Q needs more training episodes to converge to a solution than the other cooperative algorithms.
In Figure 8.11, each of the cooperative learners converges faster to a solution than single-agent Q-learning. But each of the cooperative algorithms in Figure 8.11 performs worse compared to the earlier results in Figure 8.10. This indicates that less frequent sharing of Q-values affects the performance of the cooperative learners contrastingly.
The overall results in Figures 8.10 and 8.11 indicate that each of the cooperative Q-learning algorithms converges faster to a solution than single agent Q-learning. Furthermore, sharing of Q-values more frequently in the different experiences case is better than sharing less frequently. But there is one exception: BEST-Q. The performance of BEST-Q is enhanced with an increase of the training time before the sharing of Q-values.

**Different Experiences in 10 $\times$ 10 Grid**

The figures in this section show the average number of steps to hunt the prey agent over 500 learning episodes in a 10 $\times$ 10 grid. Figure 8.12 shows the results of case one of the different experience cases. While, Figure 8.13 shows the results of case two of the different experience cases.

Figure 8.12 shows that each of the cooperative Q-learning algorithms performs similarly, but better than single-agent Q-learning. Figure 8.13 shows that single-agent
Figure 8.12: Average number of moves per 10 episodes in $10 \times 10$ grid.

Figure 8.13: Average number of moves per 100 episodes in $10 \times 10$ grid.
Q-learning and each of the cooperative Q-learning algorithms have similar performance except AVE-Q. AVE-Q learns slower than the other cooperative algorithms. This is because at each learning by interaction stage AVE-Q only moves halfway between its current value and the best value found so far which may lead to wrong moves. The other averaged based methods, WSS, PSO-Q, and Average aggregation, also perform worse than the other methods. This is because high-frequency sharing of Q-values make the individual learner’s Q-values vary widely at each learning by interaction stage compared to low-frequency sharing.

The overall results of Figures 8.12 and 8.13 suggest that each of the cooperative Q-learning algorithms performs better than single agent Q-learning when Q-value sharing is highly frequent. The performance of each of the cooperative algorithms degrades in the beginning of the learning process when information is shared less frequently (Figure 8.13).

**Different Experiences in $20 \times 20$ Grid**

In Figure 8.14, the first, the second, and the third hunter agents learn how to hunt in a $20 \times 20$ grid for 10, 5 and 1 episodes, respectively before sharing. We can see from the figure that each of the cooperative Q-learning algorithms has similar performance and converges faster to a solution than single-agent Q-learning.
In Figure 8.15, the first hunter agent learns for 100 episodes, the second learns for 50 episodes, and the third learns for 25 episodes before they share their Q-values. The figure shows that each of the cooperative Q-learning algorithms has similar performance and converges faster to a solution than single-agent Q-learning. However, the performance of the PSO-Q algorithm during the first 600 learning episodes is slower than the other cooperative Q-learning algorithms. This is because PSO-Q relies heavily on its parameters and may not search outside the neighbourhood of the best global Q-value. The algorithms that rely on averaging (i.e. WSS, AVE-Q, and Average aggregation) also perform worse than the other methods. This is because the less frequent sharing of Q-values make the individual learners’ values considerably different at each learning by interaction stage than compared to when sharing is more frequent (as in Figure 8.14).


Results Summary of Different Experiences Case

The overall results of all the experiments in the different experiences case suggest that each of the cooperative Q-learning algorithms converges faster to a solution than single-agent Q-learning when Q-value sharing is highly frequent. Sharing of Q-values after long intervals of training (low-frequency sharing) has a negative effect on the performance of some cooperative Q-learning algorithms. In this situation, the performance of each of the cooperative algorithms degrades especially the average based sharing algorithms.
8.4 Combined approach of IDQL, QA-learning and BEST-Q

IDQL (Chapter 6) is an algorithm for distributed RL systems that is based on three specialisations of agents: workers, tutors and consultants, while QA-learning (Chapter 7) is an algorithm for multiple goal-states problems that is based on two roles: learner and tutor. The problem structure of IDQL and QA-learning is similar, which encourages us to combine QA-learning, IDQL and BEST-Q in one algorithm called Enhanced QA-learning (EQA-learning). The BEST-Q will be used to aggregate the Q-tables of multiple worker agents into their tutor’s Q-table. The EQA-learning algorithm will be implemented to the nearest emergency problem.

The two learning stages of EQA-learning are:

- **First Learning Stage:**
  - The consultant agent has a Q-table that represents the problem space of the whole system.
  - Each tutor agent has a Q-table that represents the problem space of its assigned subsystem.
  - Each tutor has a number of worker agents that are responsible of learning its problem space. Each worker agent inherits the Q-table of its tutor.
  - The worker agents of each tutor start learning at the same time (equal experiences). After each period of individual learning, the worker agents of a tutor agent aggregate their Q-tables into the Q-table of their tutor using the BEST-Q aggregate strategy.
  - The first stage of EQA-learning ends when each tutor converges to a solution for its subsystem, i.e. when each tutor’s goal is accomplished and the average number of moves in the tutor’s policy improves by less than $m$ moves over $n$ consecutive episodes.
• Second Learning Stage:
  – The tutor agents combine their Q-tables into the Q-table of the consultant
    agent using the connection sets between each subsystem and the other
    adjoining subsystems (Figure 7.5).
  – The consultant agent applies the algorithm of the second stage of QA-
    learning (Figure 7.4) after combining the Q-tables of all of the tutors into
    the consultant agent’s Q-table.

8.5 Experiments 2

8.5.1 Setup

The EQA-learning and Q-learning algorithms were applied to three instances of the
nearest emergency exit problem (Section 7.3) with different sizes. In the first exper-
iment, the building is composed of 4 storeys where each storey has $25 \times 10^2$ rooms.
In the second experiment, the building is composed of 4 storeys where each storey
has $625 \times 10^2$ rooms. In the third experiment, the building is composed of 4 storeys
where each storey has $25 \times 10^4$ rooms. The doors and exits have the same positions
as shown in Figure 7.1, i.e. the exits are only in the first and the last column, but
the floors are longer (instance one: 50 rows of 50 rooms per storey; instant two: 250
rows of 250 rooms per storey; instant three: 500 rows of 500 rooms per storey ).

The learning parameters were set as follows:

• The learning rates $\alpha$ and $\beta$ equal 0.4 and the discount factor $\gamma = 0.9$ for
  both the Q-learning algorithm and the EQA-learning algorithm as in [Arai
  and Sycara, 2000, Jiang et al., 2007].

• The selection policy of actions for both algorithms was the Softmax selection
  policy where the value of the temperature $T=0.7$. 
Each tutor was assigned 4 worker agents. The Q-tables of the worker agents of each tutor were aggregated into their tutor’s Q-table after each 25 episodes of individual learning using BEST-Q. The consultant agent aggregated the Q-table of a tutor into its Q-table as soon as the tutor converged to a solution.

The reward that each worker agent receives was defined as:

\[
R(s, a, s') = \begin{cases} 
+100.0 & \text{if it reaches an exit state} \\
0 & \text{otherwise}
\end{cases}
\]

The reward that the consultant agent receives when implementing the second stage of EQA-learning was defined as:

The reward that each agent receives was defined as:

\[
R(s, a, s') = \begin{cases} 
+100.0 & \text{if it reaches an exit state} \\
-1 & \text{if it goes down or up one level} \\
0 & \text{otherwise}
\end{cases}
\]

8.5.2 Results and Discussion

In our experiments, an algorithm is said to have converged when an exit is reached and the average number of moves in its policy improves by less than one move over 100 consecutive episodes. The learning curves of the algorithms in all the figures were cut after the algorithms converged to solutions.

Figure 8.16 shows the average number of moves per 25 episodes required to reach the nearest exit in a 4-storey building where each storey is composed of \(25 \times 10^2\) rooms. In this experiment, each tutor converges to a solution for its storey after around 150 episodes of learning, which marks the end of the first learning stage of EQA-learning. The consultant converges to a solution after 125 episodes. Thus,
the total number of episodes required in EQA-learning to converge to a solution is 275 episodes when the tutors are allowed to learn in parallel and 725 episodes when the tutors are allowed to learn sequentially. In both cases, the number of episodes required to converge to a solution is less than the number of episodes of Q-learning as shown in the plot. Q-learning converges after 900 episodes to a solution. EQA-learning outperformed Q-learning because EQA-learning allows multiple learners to learn in parallel smaller subsets of the overall problem.

Figure 8.16: Experiment 1: EQA-learning vs Q-learning. The building is a 4 storey building where each storey is composed of $25 \times 10^2$ rooms.

Figure 8.17 shows the average number of moves per 25 episodes required to reach the nearest exit in a 4 storey building where each storey is composed of $625 \times 10^2$ rooms. In Figure 8.17, the parallel tutors converge to solutions after 175 episodes while the consultant converges to a solution for the whole building after 75 episodes. The total number of episodes required in EQA-learning to converge to a solution is
250 episodes which is much less than the number of episodes required to converge to a solution in Q-learning (1950 episodes).

Figure 8.17: Experiment 2: EQA-learning vs Q-learning. The building is a 4 storey building where each storey is composed of $625 \times 10^2$ rooms.

Figure 8.18 shows the average number of moves per 25 episodes required to reach the nearest exit in a 4 storey building where each storey is composed of $25 \times 10^4$ rooms. In Figure 8.18, the first learning stage of EQA-learning ends at episode 175 while the second learning stage of EQA-learning ends after 75 episodes of learning. This means that EQA-learning converges to a solution after 250 episodes which is less than the convergence time of Q-learning which is 1475 episodes.
Figure. 8.18: Experiment 3: EQA-learning vs Q-learning. The building is a 4-storey building where each storey is composed of $25 \times 10^4$ rooms.

The overall results of the experiments suggest that EQA-learning performs faster than single agent Q-learning even if we do not have parallel learning. This is because EQA-learning allows multiple learners to learn in parallel smaller subsets of the overall problem. This also means that the average length of an episode in EQA-learning is shorter than the average length of an episode in single-agent Q-learning.

<table>
<thead>
<tr>
<th></th>
<th>Experiment 1</th>
<th>Experiment 2</th>
<th>Experiment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q-learning</td>
<td>81</td>
<td>9360</td>
<td>18423</td>
</tr>
<tr>
<td>EQA-learning</td>
<td>28</td>
<td>1394</td>
<td>4532</td>
</tr>
</tbody>
</table>

Table 8.1: The running time of Q-learning vs the running time of EQA-learning in seconds.
Table 8.1 shows the running time of Q-learning vs the running time of EQA-learning for the three experiments in seconds. Row 1 of Table 8.1 shows that the running time of Q-learning increases with increase of the size of the problem, while row 2 of Table 8.1 shows that the running time of EQA-learning is much faster for each of the three experiments than single-agent Q-learning. This is because the learners in the EQA-learning scenario are learning in parallel and each of them is learning a smaller subset of the overall problem.

8.6 Summary

This chapter studies and compares various cooperative Q-learning algorithms for independent reinforcement learners. The learning process of cooperative Q-learning takes place in two stages: independent learning and learning by interaction. In the first stage, each learner learns independently using its own Q-learning algorithm. In the second stage, all of the independent learners share their Q-values based on a sharing strategy.

The chapter discussed five cooperative Q-learning algorithms: BEST-Q, AVE-Q, PSO-Q, WSS, and a cooperative Q-learning algorithm with an aggregate sharing strategy. The first learning stage of these algorithms is the same in which each learner learns independently using Q-learning, while the second learning stage (learning by interaction) is different for each algorithm because each algorithm uses a different sharing strategy. A sharing strategy determines how each agent should update its Q-table at the learning by interaction stage of cooperative Q-learning. In BEST-Q, the best Q-values are copied to their corresponding Q-values of each learner. AVE-Q is similar to BEST-Q except that the best Q-value of each state-action pair of all of the learners is averaged with its corresponding Q-value of each learner. In PSO-Q, each learner uses its personal best and the global best to update the equations of the PSO algorithm. In WSS, each learner assigns weights to the Q-tables of the other learners depending on their relative expertness. Then, each learner takes the
weighted average of the other learners’ Q-tables and uses it as its Q-table. Each agent that uses the aggregate cooperative Q-learning algorithm updates its Q-values by averaging the results of BEST-Q, AVE-Q, PSO-Q, and WSS for each state-action.

In the experimental section of this chapter, we implemented the above cooperative Q-learning algorithms and the single agent Q-learning algorithm for three instances of the hunter prey problem of different grid sizes: $5 \times 5$, $10 \times 10$, and $20 \times 20$. There are two sets of the experiments. In the first set, the learners have equal experience, and in the second set, the learners have different experiences.

The overall results of the experiments in the equal experience case indicate that sharing of Q-values is not beneficial and produces similar results to single-agent Q-learning. In addition, high-frequency sharing of Q-values accelerates the convergence to solutions compared to low-frequency sharing except for BEST-Q. In BEST-Q, the Q-values of all of the agents become the same after they share their Q-values. This means that the optimal Q-values might not be found [Iima and Kuroe, 2006]. However, the performance of BEST-Q can be enhanced by allowing the learners to train for a sufficient number of episodes before sharing their Q-values. The overall results in the different experiences case suggest that the cooperative Q-learning algorithms perform better than single-agent Q-learning, when Q-value sharing is highly frequent. Low-frequency Q-value sharing degrades the performance of the cooperative Q-learning algorithms in the equal experience and different experiences cases especially the average-based methods.

This chapter also described the EQA-learning, an algorithm for cooperative policy construction for independent learners. EQA-learning combines QA-learning, IDQL and BEST-Q in one algorithm. We implemented the EQA-learning and the single-agent Q-learning algorithms to three instances of the nearest emergency exit problem. The experimental results suggest that EQA-learning converges to a solution faster than single-agent Q-learning, even if we do not have parallel learning. In addition, the running time of EQA-learning is much faster than the running time
of Q-learning for each of the three experiments. This is because the learners in EQA-learning are learning in parallel smaller subsets of the overall problem. This means that the average length of an episode in EQA-learning is much shorter than Q-learning.
Cooperation between independent reinforcement learners is often important to learn big tasks. Classical RL algorithms such as Q-learning converge slower to optimal solutions in large state space problems than small state space problems [Asadi and Huber, 2004, Sutton and Barto, 1998]. In large state space problems, independent learners normally cooperate to accelerate the learning process using decomposition techniques or sharing of knowledge procedures.

The hierarchical approach is an efficient approach for learning large state space problems [Erus and Polat, 2007, Ghavamzadeh et al., 2006, Ono and Fukumoto, 1997, Tosic and Vilalta, 2010]. In this approach, the learning problem is viewed as a FMDP that can be decomposed into smaller sub-MDPs. Learning the smaller sub-MDPs is normally much faster than learning the whole problem. We can identify two common approaches for decomposition of large MDPs. First, static decomposition that requires the implementation designers to partially or totally define the hierarchy of the desired problem [Parr and Russell, 1997, Sutton et al., 1999]. Second, dynamic decomposition, in which the hierarchy components, their positions and abstractions are determined during the simulation process [Dietterich, 2000, Hengst, 2002, Tosic and Vilalta, 2010].
Current RL decomposition techniques do not allow migration of learners from one problem space to another in distributed systems. Instead they focus on decomposing the state or action space into more manageable parts, and statically assign each learner to one of these parts. This thesis presents a Distributed Hierarchical Learning Model (DHLM) and an Intelligent Distributed Q-learning (IDQL) algorithm that are based on three specialisations of agents, workers, tutors and consultants. Worker agents are the actual learners and performers of tasks, while tutor agents and consultant agents are coordinators at the subsystem level and the system level, respectively. A main duty of consultant and tutor agents is the assignment of problem space to worker agents. An important feature of the IDQL algorithm is that it can change the problem space of any worker agent dynamically using the migration procedures provided by the DHLM.

The performance of classical RL algorithms such as Q-learning degrades when implemented for large multiple goal problems with increase of the size of the state space. Multiple goal problems that have loosely-coupled goals can be decomposed into subproblems in which each goal and its problem space is a sub-problem. This thesis introduces Q-learning with Aggregation (QA-learning), an algorithm for multiple goal-states problems that is based on two roles: learner and tutor. A learner is an agent that learns and uses the knowledge of its neighbours (tutors) to construct its Q-table. A tutor is a learner that is ready to share its Q-table with its neighbours (learners). These roles are based on the concept of learners reusing tutors’ sub-solutions. This algorithm provides solutions to multiple goal-states problems. In this algorithm, each learner incorporates its tutors’ knowledge into its own Q-table calculations. A comprehensive solution can then be obtained by combining these partial solutions.

Various cooperative Q-learning algorithms have been proposed in the last two decades to model sharing of Q-values between independent reinforcement learners [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002, Iima and Kuroe, 2006, 2007,
2008]. In this type of algorithms, independent Q-learners share their Q-values by following a sharing strategy after spending some episodes learning independently. This thesis compares the performance of some famous cooperative Q-learning algorithms (BEST-Q, AVE-Q, PSO-Q, and WSS), and single-agent Q-learning against each other in the case the learners have equal experience and in the case they have different experiences. The thesis also studies the effects of the frequency of Q-value sharing on the learning speed of these cooperative Q-learning algorithms.

The remainder of this chapter concludes the work by first reviewing existing literature. This followed by revisiting the research questions asked in Chapter 1. Finally, a discussion of possible future research directions is presented in Section 9.3.

9.1 Review of Existing Research

9.1.1 Reinforcement Learning

In machine learning, an agent is an autonomous entity that is designed to use intelligence to automatically learn how to perform assigned tasks. The system that is modelled using multiple agents is known as Multi-Agent System (MAS). Chapter 2 introduced agent-related concepts that help in understanding MASs. This chapter focused on discussing concepts, types, classifications, learning processes, learning models, and main challenges of MASs. The agent-related concepts that have been discussed in this chapter, were also used in the discussion of Chapters 3-9 of the thesis.

Finite Markov decision processes are important to the theory of reinforcement learning since RL problems are usually formalised as MDPs. Finite Markov decision processes are the ones with finite state and action spaces. Chapter 3 explained the following topics: basic concepts of MDP, main classifications of MDP, main application fields, and main limitations of MDP model. This chapter focused on
decomposition techniques for solving large MDPs, particularly the decomposition techniques used in Reinforcement learning.

Afterwards, Chapter 4 introduced RL. RL is a learning paradigm that is based on trial and error. A RL agent learns by applying actions and receiving rewards for these actions. Chapter 4 provided basic background information about RL as first step to understand the problem of RL in multi-agent systems. It also discussed recent advances in RL.

**9.1.2 Multi-agent Reinforcement Learning**

The main difference between single-agent systems and multi-agent systems is that several agents determine the environment’s dynamics in multi-agent systems, while a central agent determines the environment’s dynamics in single-agent systems [Stone and Veloso, 2000b]. This fact makes applying single-agent RL to multi-agent systems impractical and time consuming. In recent years, many MARL approaches have been designed to model RL for multi-agent systems to overcome the problems of existing methods. The main goal of such approaches is to enable multiple agents to learn suitable behaviours in a dynamic learning environment with the use of RL. Furthermore, sharing of knowledge among cooperative reinforcement learners is an important trend in the literature. Chapter 5 examined these issues and showed how cooperative independent learners can cooperate among each other using various techniques such as swarm RL techniques. The loose coupling of multi-agent systems also makes it much easier for the learners to cooperate and share knowledge among each other without requiring the detailed knowledge of each others’ problem models and Q-tables.

MARL approach has several advantages over a single-agent RL approach. The source of the advantages of MARL is mainly because of the multiplicity of agents. The advantages include: taking advantage of parallel computation, sharing of knowledge
between agents, robustness and suitability for distributed learning. However, there are two types of challenges in MARL that complicate the learning process: challenges inherited from single agent RL, and new challenges specific to MARL. The inherited challenges include: the trade-off between exploration and exploitation, and the curse of dimensionality, while the new MARL challenges include: the non-stationary of the learning problems, the need for coordination between agents, and the difficulty of specifying a good goal.

A large part of Chapter 5 introduced some of the major advances in MARL. First, combinational RL approach, in which more than one RL algorithms are combined together in a single framework to accelerate the learning process and/or to take advantage of the strength points of each algorithm. Second, swarm MARL algorithms that are based on the swarm MAS model, and are used to model the learning process of cooperative independent learners. Third, cooperative MARL algorithms that focus on coordination of agents’ actions and agents’ concurrency. Fourth, cooperative MARL algorithms for stochastic games. Fifth, methods for reducing the joint action space in cooperative MARL algorithms, and finally recent research about cooperative RL using the teacher learner model.

9.1.3 Hierarchical Structure of Complex Systems

The coupling degree describes the dependency degree to which each subsystem relies on the other subsystems [Kaye, 2003]. Loosely-coupled systems (i.e., systems with low coupling degree) are easier to process and decompose into smaller subsystems than tightly-coupled systems (i.e., systems with high-coupling degree). Distributed systems and multi-goal systems are complex systems that have hierarchical structure that can be decomposed into simpler subsystems that are loosely coupled.

Chapters 6 and 7 affirm that the hierarchical organisation of decomposable systems provides an efficient decomposition into more manageable components. The problem
model described in Section 6.3.1 and Section 7.4.1 is used to model the distributed RL models and algorithms proposed in Chapters 6 and 7.

There are three concepts that form the proposed distributed problem model. First, subsystem which is an MDP with connection sets that defines the MDP’s boundaries with its neighbouring subsystems. Second, connection set which is a component of subsystem that shows how a subsystem connects to another subsystem. Finally, system which is a collection of loosely-coupled subsystems.

In conclusion, the problem model provides support for the hierarchical organisation of distributed systems and multi-goal systems. In this model, the coupling between subsystems are modelled as connection sets. The implementation details of the problem model such as the description of the subsystems should be determined by the implementation designers.

9.2 Conclusion

This section returns to the research questions presented in Chapter 1, examining how each has been handled in the work presented in this thesis.

9.2.1 Research Question 1

- How to model cooperative distributed systems using different types of learners? Given that a system is composed of loosely coupled subsystems, how can a cooperation algorithm provide support for learners’ migration from one subsystem to another?

Chapter 6 introduced a distributed hierarchical learning model (DHLM) and an intelligent distributed Q-learning algorithm (IDQL) that are based on the problem model in Section 6.3.1.
DHLM and IDQL are based on three specialisations of agents: workers, tutors and consultants. Worker agents are the actual learners and performers of tasks, while tutor agents and consultant agents are coordinators at the subsystem and the system levels, respectively. IDQL models each problem space of tutor agents and consultant agents as a separate problem space. A main duty of consultant and tutor agents is the assignment of problem space to worker agents based on the description of the learning tasks.

An important feature of DHLM is that it provides a solution for migration of agents. In DHLM, migrant agents can follow one of two different migration modes: client server or mobile agent. In the client server mode, the migrant agent duplicates itself then the copy migrates to another environment. On the other hand, in the mobile agent mode, the migrant agent terminates its execution in its current environment and then migrates to another environment.

The migration feature of DHLM provides support for IDQL algorithm where the problem space of each worker agent can change dynamically. The assigned problem space of a worker agent is not fixed through all of the learning process. Consultant agents monitor the progress of work in each subsystem, and redistribute worker agents when they are needed in other subsystems. The consultant agents simply keep track of the progress of each tutor, and initiate agents’ migration when it is required. The existing hierarchical RL models do not provide solution for agents’ migration.

We experimented with an ambush strategy using our models in a distributed hunter-prey problem. This ambush strategy supports two types of cooperation that do not require communication among cooperative agents: cooperation at subsystem level and cooperation at system level. The experimental results suggest that IDQL converges faster to a solution than parallel or sequential single-agent Q-learning. This occurs because each hunter agent learns a smaller sub-grid of the overall grid, and IDQL supports migration of hunter agents.
The roles and tasks of DHLM and IDQL agents need to be defined by the implementation designers. An advantage of DHLM is that it introduced a hierarchical model of learning that supports migration of learners.

### 9.2.2 Research Question 2

- How can reinforcement learners reduce the training time and use less memory for learning problems with large state spaces and multiple goal-states? How can the reinforcement learners compose one general solution from the solutions of the subsystems with multiple goal-states? What levels of support can be offered to facilitate cooperation between multiple learners?

The speed of convergence to optimal solutions of classical RL algorithms such as Q-learning degrades with increase of the size of the state space [Asadi and Huber, 2004, Sutton and Barto, 1998]. A possible solution is the decomposition approach in which large state space problems are decomposed into loosely coupled subsystems that can be processed independently [Barto and Mahadevan, 2003, Daoui et al., 2010, Ghavamzadeh et al., 2006, Hansen and Feng, 2000]. Chapter 7 examined this issue and introduced QA-learning, an algorithm for decomposable multi-goal problems that is based on the problem model described in Section 7.4.1. QA-learning is an algorithm for cooperative policy construction for independent learners that is based on two roles: learner and tutor. A learner is an agent that learns and uses the knowledge of its neighbours (tutors) to construct its Q-table. A tutor is a learner that is ready to share its Q-table with its neighbours (learners). These roles are based on the concept of learners reusing tutors’ sub-solutions. In this algorithm, each agent incorporates its tutors’ Q-tables into its own Q-table calculations. Informing the learner of the existence of other goals around it, and the way it should use them, increases the accuracy of each learner’s Q-table. A comprehensive general solution for the whole system can be calculated using the sub-solutions of all agents.
All the solutions to the sub-systems should be combined to produce a solution to the whole system. In the QA-learning algorithm, each solution of a subsystem is integrated with the solutions of the adjoining subsystems. Integration approaches such as incremental, whole system, or sequential approaches produce a general solution for the whole system.

The QA-learning and Q-learning algorithms were implemented to three instances of the nearest emergency exit problem (Section 7.3) with different sizes. In Experiment 1, the building is composed of 4 storeys where each storey has 100 rooms. In Experiment 2 and Experiment 3, the building is composed of 8 and 12 storeys respectively, where each storey has 200 rooms.

The QA-learning was implemented based on three integration scenarios (Section 7.4.3): whole system integration, incremental integration of size one, and incremental integration of size two. The overall experimental results suggest that QA-learning performs similarly with any of the integration procedures (system or incremental integrations).

The overall experimental results suggest that QA-learning performs better compared to single-agent Q-learning because QA-learning allows multiple learners to learn in parallel then combines their solutions through few learning episodes. Even if we do not have parallel learning, the total number of learning episodes required to converge to a solution, is still less than the number of episodes required for single-agent Q-learning. When we increased the task size, the performance difference in the parallel execution case of QA-learning became even larger; in Experiment 1 QA-learning required 31% of the episodes of single-agent Q-learning, in Experiment 2 QA-learning required only 22% of the episodes of single-agent Q-learning, and in Experiment 3 QA-learning required only 20% of the episodes of single-agent Q-learning. Sequential QA-learning reduces the required learning episodes of single-agent Q-learning because the learners in the first stage of QA-learning learn smaller subsets of the original problem. This also means that the average length of an
episode in QA-learning is shorter than the average length of an episode in single-agent Q-learning. QA-learning produces similar results when implemented following system or incremental integration.

The overall experimental results also suggest that the running time of Q-learning increases with the increase of the size of the problem, while the running time of QA-learning is much faster for each of the three experiments. This is because the learners in QA-learning are learning in parallel and each of them is learning a smaller subset of the overall problem.

The QA-learning algorithm has many advantages. First, the problem model of the QA-learning algorithm is a loosely-coupled FMDP. This model reduces the complexity of large state space problems by taking advantage of the decomposable nature of the system itself.

Second, the QA-learning algorithm supports the reuse of sub-solutions. When two subsystems are compatible, a Q-table constructed for one subsystem can be reused in the other.

Third, unlike the learning methods of Kuniyoshi et al. [1994], Thomaz and Breazeal [2008], and Saad et al. [2009], the learning process of QA-learning is a bidirectional process in which an agent can be both a learner and a tutor.

Fourth, the communication between agents in the QA-learning algorithm is decentralised and there is no implementation constraint on how the subsystem MDPs should be implemented.

Finally, the experimental results suggest that QA-learning performs better than single agent-Q-learning on problems of different sizes.
9.2.3 Research Question 3

- What are the main differences between the best-known Q-value sharing strategies that are used to model the second stage of cooperative Q-learning for independent learners? How do these sharing strategies compare based on their performance? What are the effects of the frequency of Q-value sharing on the learning speed of independent learners?

In cooperative Q-learning for independent Q-learners, each Q-learner spends some time learning independently using its own Q-learning algorithm. Then, the learners share and update their Q-values following a Q-value sharing strategy. The main difference between the most famous cooperative Q-learning algorithms for independent learners (e.g., BEST-Q, AVE-Q, PSO-Q [Iima and Kuroe, 2006, 2007, 2008], and WSS [Ahmadabadi et al., 2006, Ahmadabadi and Asadpour, 2002]) is the type of the sharing strategy each one uses.

Chapter 8 introduced a comparison study of the performance of BEST-Q, AVE-Q, PSO-Q, WSS, and an algorithm that aggregates their results. The results of these algorithms were compared in two cases: when each of the learners had equal levels of expertness, and when they had different levels of expertness. Furthermore, the study examined the effects of the frequency of Q-value sharing on the learning speed.

The above algorithms were implemented to three instances of the classical hunter prey problem that have different grid sizes: $5 \times 5$, $10 \times 10$, and $20 \times 20$. Each instance involved four players: three hunters and one prey.

The obtained results of all the experiments in the equal experience case suggest that sharing of Q-values is not beneficial and produces similar results to single-agent Q-learning. Also, high-frequency sharing of Q-values accelerates the convergence to a solution compared to less frequent sharing except in BEST-Q. A problem with BEST-Q is that the Q-values of all of the agents become the same after each sharing session which means that the optimal Q-values might not be found [Iima and Kuroe,
This problem is more extreme when the learners share their Q-values so frequently that the individual learners have little chance to improve their own values. But, the performance of BEST-Q can be enhanced by allowing the learners to train for a sufficient number of episodes before sharing their Q-values.

The overall results of all the experiments in the different experiences case suggest that each of the cooperative Q-learning algorithms converges faster to a solution than single-agent Q-learning when Q-value sharing is highly frequent. Low-frequency sharing of Q-values has a negative effect on the performance of some cooperative Q-learning algorithms. In this situation, the performance of each of the cooperative algorithms degrades especially the average-based sharing algorithms.

In both cases, high-frequency sharing of Q-values accelerates the convergence to solutions of most of the cooperative Q-learning algorithms compared to low-frequency sharing. Low-frequency Q-value sharing degrades the performance of the cooperative Q-learning algorithms in the equal experience and different experiences cases. When Q-value sharing is low, the cooperative Q-learning algorithms can be classified as follows:

- BEST-Q performs better than the other cooperative Q-learning algorithms. This is because the performance of BEST-Q enhances with increase of the number of the learning episodes of the individual learning stage.

- The average based algorithms (i.e. WSS, AVE-Q, and Average aggregation) perform worse than BEST-Q. This is because the low-frequency sharing of Q-values make the individual learners’ Q-values vary widely at each learning by interaction stage compared to high-frequency sharing.

- The performance of PSO-Q is slower than the other cooperative Q-learning algorithms. This is because PSO-Q relies heavily on its parameters and may not search outside the neighbourhood of the best global Q-value.
Chapter 8 also described an algorithm for cooperative policy construction for independent learners (EQA-learning). This algorithm combines QA-learning, IDQL and BEST-Q in one algorithm. The EQA-learning and the single-agent Q-learning algorithms were implemented to three instances of the nearest emergency exit problem. The experimental results indicate that EQA-learning converges to a policy faster than single-agent Q-learning in parallel or sequential execution modes. In addition, the running time of EQA-learning is much faster than the running time of Q-learning for each of the three experiments. This is because the learners in EQA-learning are learning in parallel smaller subsets of the overall problem. Therefore, their individual learning episodes are typically of shorter duration.

9.3 Future Directions

This section presents possible enhancements and expansion of the techniques described in this thesis.

9.3.1 Cooperative Dyna Architecture

Dyna is an AI architecture that combines RL and execution time planning into a single model (Section 4.15.1). In the future, we will work to develop a cooperative Dyna architecture for independent learners that combines multiple RL algorithms and aggregation techniques in the classical Dyna model.

In the new architecture, a learner applies multiple RL algorithms for execution time learning and other RL algorithms for execution time planning. At the end of each learning session, the learner uses a voting method to aggregate the resulted Q-values during execution time learning, and a voting method to aggregate resulted Q-values during execution time planning. Examples of the voting methods are maximum, minimum, and median voting. The aggregate RL algorithms in Jiang and Kamel
[2006] and Wiering and van Hasselt [2008] implement some of these voting methods to combine multiple RL algorithms that execute in parallel. This architecture will realise the benefits of the methods described all throughout this thesis.

### 9.3.2 QA-learning for Single-goal Hierarchical Systems

Implementing QA-learning in single-goal hierarchical systems and automating the identification process of subsystems is one of our future research plans. Our next target is to study the applicability of QA-learning in partially observable environments.

### 9.3.3 Hierarchical Reinforcement Learning algorithms

The research studies of Jardim et al. [2011], Cao and Ray [2012], Cai et al. [2013], Gunady et al. [2014] and Cuáyahuítl et al. [2014] have recently proposed new hierarchical reinforcement learning algorithms and approaches. In future work, we will experimentally compare the performance of these algorithms and approaches in the hunter prey problem.

The QA-learning algorithm requires the hierarchy components of the learning problem to be defined before the beginning of the learning process. Hence, in the future, we will combine the MAX-Q method [Dietterich, 2000] with the QA-learning algorithm to automatically identify the hierarchy of the learning problem.
Bibliography


Tosic, P. T., Vilalta, R., 2010. A unified framework for reinforcement learning, co-
learning and meta-learning how to coordinate in collaborative multi-agent systems.
Procedia Computer Science 1 (1), 2217–2226.

Uther, W., 2010. Temporal difference learning. In: Encyclopedia of Machine Learn-

Van Seijen, H., Sutton, R. S., 2013. Planning by prioritized sweeping with small

Vasudevan, N., Venkatesh, P., August 2006. Design and implementation of a process
migration system for the Linux environment. In: 3rd International Conference on
Neural, Parallel and Scientific Computations. Atlanta, Georgia, USA. pp. 1–8.

two-layered multi-agent reinforcement learning model and algorithm. Journal of

sarsa algorithm and q-learning. Engineering Applications of Artificial Intelligence
26 (9), 2184 – 2193.

Watanabe, T., Sawa, T., July 2010. Instruction for reinforcement learning agent
based on sub-rewards and forgetting. In: 2010 IEEE International Conference on


279–292.

Weinberg, M., Rosenschein, J. S., 2004. Best-response multiagent learning in non-
stationary environments. In: Jennings, N., Tambe, M., Sierra, C., Sonenberg, L.
(Eds.), Proceedings of the Third International Joint Conference on Autonomous


