Imperial College London
Department of Computing

Argumentation Accelerated Reinforcement Learning

Yang Gao

March 2015

Supervised by Professor Francesca Toni

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work

Abstract

Reinforcement Learning (RL) is a popular statistical Artificial Intelligence (AI) technique for building autonomous agents, but it suffers from the curse of dimensionality: the computational requirement for obtaining the optimal policies grows exponentially with the size of the state space. Integrating heuristics into RL has proven to be an effective approach to combat this curse, but deriving high-quality heuristics from people's (typically conflicting) domain knowledge is challenging, yet it received little research attention. Argumentation theory is a logic-based AI technique well-known for its conflict resolution capability and intuitive appeal. In this thesis, we investigate the integration of argumentation frameworks into RL algorithms, so as to improve the convergence speed of RL algorithms.

In particular, we propose a variant of *Value-based Argumentation Framework* (VAF) to represent domain knowledge and to derive heuristics from this knowledge. We prove that the heuristics derived from this framework can effectively instruct individual learning agents as well as multiple cooperative learning agents. In addition,we propose the *Argumentation Accelerated RL* (AARL) framework to integrate these heuristics into different RL algorithms via *Potential Based Reward Shaping* (PBRS) techniques: we use classical PBRS techniques for *flat* RL (e.g. SARSA(λ)) based AARL, and propose a novel PBRS technique for *MAXQ-0*, a *hierarchical* RL (HRL) algorithm, so as to implement HRL based AARL. We empirically test two AARL implementations — SARSA(λ)-based AARL and MAXQ-based AARL — in multiple application domains, including single-agent and multi-agent learning problems. Empirical results indicate that AARL can improve the convergence speed of RL, and can also be easily used by people that have little background in Argumentation and RL.

Acknowledgements

I consider myself very fortunate to have benefited from the unparallelled knowledge and guidance of my supervisor, Professor Francesca Toni. I am greatly indebted to her patience, continued support and encouragement throughout my PhD studies. Francesca allowed me to follow my own research agenda and to slowly plow my way through research topics I am interested in, while, at the same time, inspiring me with new ideas and providing invaluable instructions whenever I had problems. She spent late nights and early mornings in reading and revising my papers, shared her enthusiasm for my work with her colleagues, and introduced me to the academic community. Without her help, this thesis would not have been possible.

Also, I must give my special thanks to my parents and my girlfriend, Kai Sun, for their consistent support, help and love. Kai cheerily endured my foulest moods and helped me in countless ways, from cooking meals for me during long nights spent in front of my computer, to proofreading drafts of my thesis. The video talk with my parents in each weekend has been my most enjoyable part in each week.

Special thanks also to Xiuyi Fan, who has been an invaluable sounding board giving helpful feedback on many naive thoughts I gave him, a trustworthy 'big brother' providing me with a lot of insights into the problems I encountered, and a humorous friend cheering me up when my self-confidence waned. Without Fan, my life at Imperial would be far less colourful.

In addition, I would like to thank my funding body, China Scholarship Council, for its generous scholarship covering both my tuition and living expenses.

Thanks to everybody in the CLArg group and many others in the department for helping to create an enjoyable working environment. Special thanks to Krysia Broda and Murray Shanahan for their comments on my work, and to Amani El-Kholy, Ann Halford and Teresa Ng for looking after me so well in the department.

Finally, many thanks to Abbas Edalat, Sanjay Modgil and Marc Deisenroth, my examiners, for reading this thesis and giving some very useful comments that have contributed to greatly improve it.

Contents

1 Introduction **11**

1.1 Overview 12
1.2 Proactive Decision Making and Argumentation 14
1.3 Reactive Decision Making and RL 17
1.4 Contribution 19
1.5 Structure of This Dissertation 20
1.6 Publications 21
1.7 Statement of Originality 22

2 Background **23**

2.1 Argumentation Theory 23
2.1.1 A Brief Overview of Argumentation Theory 23
2.1.2 Abstract Argumentation Frameworks 25
2.1.3 Value-Based Argumentation Framework 27
2.1.4 Computation of Argumentation Semantics 28
2.2 Reinforcement Learning (RL) 29
2.2.1 Markov Decision Processes (MDPs) and Semi-MDPs . . . 31
2.2.2 SARSA Learning Algorithm 36
2.2.3 Hierarchical RL (HRL) and MAXQ 42
2.2.4 Eligibility Traces 53
2.2.5 Potential-Based Reward Shaping (PBRS) 56
2.3 Conclusion 58

3 Argumentation Frameworks for Reinforcement Learning **60**

3.1 Motivation 60
3.1.1 The RoboCup Soccer Games 61
3.1.2 Why Incorporate Argumentation into RL 64
3.2 Argumentation Frameworks for Reinforcement Learning 65
3.2.1 Problem Definition 65

3.2.2 Argument for RL . . . 66
3.2.3 Argumentation Frameworks . . . 68
3.2.4 From Extensions to Heuristics . . . 76
3.3 Argumentation Accelerated RL (AARL) . . . 78
3.4 Related Work . . . 81
3.5 Conclusion . . . 83

4 SARSA(λ)-based AARL 84
4.1 SARSA(λ)-based AARL . . . 84
4.2 Experiments in RoboCup Games . . . 87
4.2.1 Performances of Both Sides Using Fixed Strategy . . . 89
4.2.2 Experiments in Keepaway games . . . 91
4.2.3 Experiments in Takeaway games . . . 96
4.3 Experiments in a Wumpus World Game . . . 102
4.4 Related Work . . . 109
4.5 Conclusion . . . 111

5 Potential Based Reward Shaping for Hierarchical RL 112
5.1 MAXQ with PBRS . . . 112
5.1.1 Integrating Potential Values into the MAXQ Decomposition 114
5.1.2 The PBRS-MAXQ-0 Algorithm . . . 120
5.2 Experiments . . . 130
5.2.1 Taxi Problem . . . 130
5.2.2 Stochastic Wumpus World . . . 133
5.3 Related Work . . . 139
5.4 Conclusion . . . 140

6 MAXQ-Based AARL: An Empirical Evaluation 142
6.1 MAXQ-based AARL . . . 143
6.2 The Energy Usage Recommendation System . . . 147
6.3 Why Use MAXQ-Based AARL . . . 148
6.4 A Real-Data-Based Simulated User . . . 149
6.5 Model the Problem as a MDP . . . 152
6.6 Arguments for EURS . . . 157
6.6.1 Arguments Supporting Primitive Actions and Their Values 157
6.6.2 Arguments Supporting Composite Sub-tasks and Their values . . . 164

6.7 Experimental Settings and Results 165
6.7.1 Experimental Settings 166
6.7.2 Results in Learning Phases 168
6.7.3 Results in the Evaluation Phases 169
6.8 Related Works 173
6.9 Conclusion 176

7 Conclusion **178**

List of Tables

2.1 Some types of computation in abstract argumentation frameworks. 29
2.2 Complexity of some types of computation for the preferred and grounded semantics (adjusted from [DW09]). 29

4.1 The performances of 3 keepers playing against 2 takers, both using fixed strategies. 91
4.2 The performances of 4 keepers playing against 3 takers, both using fixed strategies. 91
4.3 State variables in a N-Keepaway game. 93
4.4 Performances of learning keepers playing against random takers after several hours of learning. 96
4.5 Performances of learning keepers playing against always-tackle takers after 40 hours of learning. 97
4.6 Performances of learning keepers playing against argument-based takers after 40 hours of learning. 98
4.7 State variables for learning taker T_1 in a N-Takeaway game. . . . 100
4.8 Performances of learning takers playing against random keepers after 40 hours of learning. 101
4.9 Performances of learning takers playing against random keepers after 40 hours of learning. 101
4.10 Some statistics of the experiments involving students' arguments. . 109

5.1 Some statistics during the learning in the stochastic Wumpus World. 138

6.1 The average rewards and their standard errors of different SARSA(0)-based algorithms in the evaluation phases. 172
6.2 The average rewards and their standard errors of different MAXQ-based algorithms in the evaluation phases. 173
6.3 Pairwise p-values of SARSA-based algorithms in EURS. 174
6.4 Pairwise p-values of MAXQ-based algorithms in EURS. 175

List of Figures

1.1 A scenario in a simple Wumpus World. 13
1.2 An AF for representing the domain knowledge given in Section 1.1. 15
1.3 The interaction between a RL agent and the environment. 18

2.1 A 2×2 Wumpus World. 34
2.2 The first episode of the SARSA-based learning in the 2×2 Wumpus World in Figure 2.1. 37
2.3 The second episode of the SARSA-based learning. 39
2.4 The third episode of the SARSA-based learning. 40
2.5 The task graph for the stochastic Wumpus World problem. 45
2.6 An example scenario in the extended Wumpus World. 47
2.7 The hierarchical execution process of the Wumpus World example shown in Figure 2.6. 49

3.1 An example scenario in RoboCup Soccer Game. The ball is the white circle next to keeper K_1. 62
3.2 The derived AF of SCAFs for the Keepaway game (left) and the Takeaway game (right) in the scenario shown in Figure 3.1. 69
3.3 The simplified argumentation frameworks for the Keepaway game (left) and the Takeaway game (right) in the scenario shown in Figure 3.1. 72
3.4 The simplified argumentation framework for takers in the scenario shown in Figure 3.1, given the new value ranking *QUICK_CLOSE* $>_v$ *QUICK_TAC* $>_v$ *QUICK_MARK*. 76
3.5 The architecture of AARL. 79

4.1 The performances of learning keepers against random takers. . . . 94
4.2 The performances of learning keepers against always-tackle takers. 94

4.3 The performances of learning keepers against argument-based takers. . . . 95
4.4 The performances of learning takers against random keepers. . . . 99
4.5 The performances of learning takers against hand-coded keepers. . 99
4.6 The performances of two Takeaway games by using potential values proposed in [DGK11]. . . . 102
4.7 The GUI-based system of the Wumpus World game. . . . 104
4.8 Performances of SARSA(0)-based AARL and standard SARSA(0) in the Wumpus World game. . . . 108
4.9 A histogram summarising the number of students' AARLs receiving different overall rewards. . . . 109

5.1 Settings of the Taxi problem. . . . 114
5.2 Performances in the Taxi problem. . . . 133
5.3 Performances of R-MAXQ and R-MAXQ in the Taxi Problem. . . 134
5.4 A Wumpus World. . . . 135
5.5 Performances of four RL algorithms in the stochastic Wumpus World game. . . . 137

6.1 The probability distribution of the switching on time of the washing machine. . . . 150
6.2 The probability distribution of the switching on time and working time of the TV set. . . . 151
6.3 The task graph for the energy adviser system. . . . 156
6.4 Settings we use to implement AARL for EURS. . . . 158
6.5 Performances of SARSA(0)-based EURS in the learning phases. . 170
6.6 Performances of MAXQ-based EURS in the learning phases. Lighter colour areas represent 95% confidence intervals. All results are averaged over 100 independent experiments, each consisting of 2000 episodes. . . . 171

7.1 An AF including epistemic arguments. . . . 181

1 Introduction

We take decisions every day, and the ability for making good decisions is an important reflection of people's intelligence. Two decision-making strategies are widely used in our daily lives: the *proactive* decision-making strategy, which predicts possible outcomes of each available option and evaluates these outcomes before making the decision; and the *reactive* strategy, which simply tries options, observes and evaluates their outcomes, and remembers the best option, so that when the same situation is encountered again, the best option can be directly invoked. Intuitively, when the environment is deterministic and the outcomes can be easily predicted, the proactive strategy can be more effective; otherwise, the reactive strategy can be more useful as the proactive strategy cannot work effectively when the environment is highly unpredictable. In many real cases, the environment is stochastic and we cannot precisely predict all possible outcomes; but still, we may have some domain knowledge about the environment or the optimal decisions, based on which we can perform some proactive planning. In these cases, making decisions by jointly using these two strategies is more likely to lead to the optimal decisions than using any of these strategies alone. The general research target of this dissertation is to design intelligent agents that can make decisions by jointly using the proactive and reactive strategies.

Our research falls under the remit of *Artificial Intelligence* (AI), and much AI research has been devoted to building proactive or reactive decision-making agents. For example, *Argumentation Theory* (e.g. as given in [Dun95]), which resolves conflicts between different options using the simple idea that 'the one who holds the last word laughs best', has been widely used for designing proactive agents, e.g. in [KM03, FTMW14]; and *Reinforcement Learning* (RL) [SB98], a *Machine Learning* algorithm based on the idea of 'trial and error', has been widely used in designing reactive, sequential-decision-making agents, e.g. in [NCD$^+$06, SSK05]. Despite their strengths, both face problems when used alone in real applications: for example, a main problem faced by RL is the *curse of dimensionality*, which states the computational requirement for obtaining the optimal policies grows ex-

ponentially with the size of the state space; as for argumentation, it still lacks a mechanism for performing sequential-decision-making under stochastic environments. Given the complementary advantages of these two techniques (RL's advantage of making sequential decisions in stochastic environment, and Argumentation's advantage of proactive reasoning and planning), we believe that their integration can result in a technique, which not only is able to reactively perform sequential decision making in stochastic environments, but also able to proactively make draft action plans based on prior knowledge. A main contribution of this thesis is a methodology for systemically and effectively incorporating these two techniques. To the best of our knowledge, this is the first proposal for integrating these two techniques.

In the remainder of this chapter, we first briefly describe the problems we want to tackle and how we tackle these problems in Section 1.1, and then we give a non-technical introduction of Argumentation Theory and RL in Section 1.2 and 1.3, respectively. We summarise the contribution of this dissertation in Section 1.4, and provide a brief outline of the dissertation structure in Section 1.5. The publications which have resulted from this thesis are presented in Section 1.6, and the statement of originality is in Section 1.7.

1.1 Overview

This dissertation addresses the topic of planning and learning for stochastic, sequential decision making. The term 'stochastic' means that the environment in which decisions are taken has uncertain factors, i.e. the outcome of actions may not be deterministic. The 'sequential' aspect of the decision problem reflects the fact that the immediate costs or benefits of an action only play a small part in determining the true long-term value of this action. In other words, we are not focusing on making a 'one-shot' decision, which maximises the immediate utility; instead, we attempt to make decisions that maximise the long-term rewards. The decision-making problem is framed from the perspective of an agent or a group of agents that are situated in a stochastic environment, and the agents have some sensors to detect certain signals in the environment. Our goal is to develop algorithms that can find these agents' optimal *policy*, which describes a strategy maximising the long-term rewards received by the whole group of agents in this environment,[1] and

[1] In this thesis, we only consider the *cooperative* multi-agent learning problem, in the sense that the goal each learning agent is to maximise the whole team's long-term reward.

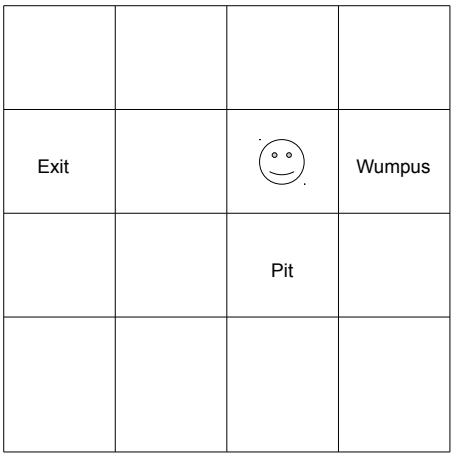


Figure 1.1: A scenario in a simple Wumpus World. The smiling face stands for the agent. Note that the agent does not directly know the location of the Wumpus, the pit or the exit.

these algorithms involve both planning (proactive decision-making) and learning (reactive decision-making).

As a concrete example, consider the decision making problem in a simple *Wumpus World*, a classic application domain in AI [RN09]. The Wumpus World is a grid world, where there are some Wumpuses, some pits and an exit. Each experiment in the Wumpus World consists of a series of *episodes*. At the beginning of each episode, an agent is located in some square in the Wumpus World. The agent has four available actions: *go_up*, *go_down*, *go_left* and *go_right*. These actions move the agent one square towards the intended direction if there are no walls next to the agent in the intended direction; otherwise, the agent will remain in the same square. When the agent steps into a square that has a pit or a Wumpus, the agent dies and the episode ends. When the agent is in a square (non-diagonally) next to a pit or a Wumpus, it detects breeze or stench, respectively. The agent's task is to arrive at the exit as quickly as possible, without being killed by a Wumpus or falling into a pit.

Now suppose the agent is in a square such that it feels stench and breeze, as illustrated in Figure 1.1. Also, we have domain knowledge that, in this scenario, *go_up* and *go_left* do not lead to death. Neither the proactive nor the reactive decision-making strategy can effectively find the best policy in this situation when used alone: by using the proactive planning alone, although the agent can infer

that *go_up* and *go_left* are better than the other available actions (as we will see in the next section), it cannot decide which is the best action; on the other hand, by using the reactive strategy alone, the agent needs to perform all actions before it identifies the best action, although it is unnecessary to try actions *go_right* and *go_down*. So, intuitively, the best solution is to use these two strategies together: first use proactive planning to select some promising actions, and then use the reactive strategy to systematically test their performances and find the best. This example illustrates the motivation of our research. In the following two sections, we will describe, in detail, which proactive and reactive technique we focus on in this thesis, and further motivate why we integrate these techniques.

1.2 Proactive Decision Making and Argumentation

Proactive decision making involves making plans and predictions to avoid a potential disaster or to increase the likelihood of a potential success. Reasoning and planning abilities play important roles in this process. Since people naturally use logic to perform reasoning [Mac01], logic-based approaches are widely used in designing reasoning and planning algorithms in AI (see, e.g. [Min01]). Among the logic-based techniques, *Argumentation Theory* [RS09] has attracted considerable and increasing research attention in recent years, because of its conceptual simplicity as well as strong conflict-resolving capability.

Argumentation is usually traced back to Aristotle's work in formal deductive reasoning and rhetoric [Irw89]. In its classical treatment within philosophy, the study of argumentation is concerned with how assertions are proposed, discussed, and resolved in the context of issues upon which several diverging opinions may be held [BCD07]. The important early motivations that brought Argumentation Theory into use in AI arose from the issue of reasoning and explanation in the presence of incomplete and uncertain information. As classical logic is proved to be inadequate in addressing these issues [Rei80], *non-monotonic* logic has attracted substantial research interest within AI, and formal logics of Argumentation emerged as one style of formalising non-monotonic reasoning (e.g. see discussions in [CML00]). In the non-monotonic logic reasoning formalisms, conclusions drawn may be later withdrawn when additional information is obtained [Ant97], and the idea of Argumentation is that reasoning can be performed by constructing and evaluating arguments, which are composed of a number of reasons for the validity of a claim. Argumentation is considered as a non-monotonic logic because

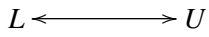


Figure 1.2: An AF for representing the domain knowledge given in Section 1.1.

new information may give rise to new counter-arguments defeating arguments that were originally acceptable. Whether a claim can be accepted, therefore, depends not only on the existence of an argument that supports this claim, but also on the existence of possible counter-arguments, that can then themselves be attacked by counter-arguments, etc.

In this thesis, we focus on computational argumentation frameworks, starting with the influential work of Dung [Dun95]. In Dung's *Abstract Argumentation Frameworks* (AFs), an argumentation framework is only consisting of a set of arguments and a binary *attack* relation. Therefore, AFs are usually represented by directed graphs, where each node stands for an argument and each directed arc stands for an attack. A more comprehensive and detailed review of Argumentation Theory will be given later in Chapter 2.

As an illustration, let us still consider the example we presented in Section 1.1. The domain knowledge can be represented by the AF in Figure 1.2, where L (U) is a shorthand for argument 'recommend the agent to perform action *go_left* (*go_up*) if going left (going up) is deemed to be safe'. Attacks between arguments are represented by directed arcs in Figure 1.2: an arc pointing from argument A to argument B means 'A attacks B', and a two sided arc connecting argument A and B means that A and B mutually attack one another. In Figure 1.2, we can see that two arguments L and U mutually attack each other, because, at each moment, the agent can at most perform one action. Also, we can see that, in this AF, there are no arguments supporting actions *go_down* or *go_right*, because there is no domain knowledge suggesting that these actions are safe; instead, sensory information (i.e. the agent feels stench as well as breeze in the current square) suggests that performing these actions may be dangerous. Since we have no 'applicable' information to encourage the agent to perform *go_right* and *go_down*, no arguments supporting these actions appear in this AF. By comparing the domain knowledge we presented in Section 1.1 and the AF illustrated in Figure 1.2, we can see that the domain knowledge as well as the conflicts therein are concisely and precisely represented by the AF.

To resolve conflicts in AFs and identify the 'winning' arguments, multiple criteria, also known as *semantics*, are proposed [Dun95] and some algorithms are

developed for computing winning arguments according to these criteria [MC09]. In the AF shown in Figure 1.2, for example, different winning arguments may be obtained by using different semantics: according to a 'sceptical' semantics, neither of these two arguments is winning, because no one can 'defeat' the other and thus convincingly win; instead, according to a 'credulous' semantics, both of these arguments win, because each of them can counter-attack all arguments attacking it. Later in Chapter 2, we will revisit this example and introduce some other criteria for selecting winning arguments in greater detail.

Furthermore, based on AFs, new computational argumentation frameworks have been developed, and these new frameworks have even stronger conflict-resolving abilities. For example, *Value-based Argumentation Frameworks* (VAFs) [BCA09] extend AFs with values so that the relative importance of each argument can be ranked. In order to choose the winning arguments in a VAF, we not only need to consider the attack relation between arguments, but also take into account the relative importance of different arguments. In other words, VAFs can take advantage of more information (the relative importance of arguments) to select the winning arguments and, therefore, they have stronger capability in resolving conflicts contained in the domain knowledge (details of VAFs will be presented later in Chapter 2). In addition, a particularly noticeable advantage of VAFs, which is not shared by other extensions of AFs (e.g. *Preference-based Argumentation Frameworks* [AC02]) is that the values in VAFs often represent people's expectations of the outcomes of different arguments, and in some applications (e.g. the energy usage adviser system we introduced later in Chapter 6) the domain expert can correlate the values in VAFs with numerical values straightforwardly. Hence, by using VAFs, people can easily rank the actions according to the rankings of their outcomes; by doing this, people can not only rank arguments, but also provide the reasons for this ranking. This property of VAFs facilitates people to justify their rankings of arguments. Because of the conceptual simplicity as well as strong conflict-resolving capability, AFs and their extensions have been widely used in designing decision support tools in many real applications, e.g. for legal [PS07] and medical [FCS$^+$13] decision-making support. However, to the best of our knowledge, AFs have not been used in stochastic sequential decision-making problems yet, because they lack systemic mechanisms in dealing with uncertainty and time series simultaneously. Considering the strengths and limitations of argumentation frameworks, in this thesis we integrate a variant of VAF into machine learning techniques for tackling stochastic sequential decision problems.

1.3 Reactive Decision Making and RL

Reactive decision making is often used in situations when people cannot or do not want to plan for problems because they have no prior knowledge about how to solve these problems. In reactive decision making, people simply try each option, observe their outcomes and decide which is the best option. This process is expensive, because trying each option is time-consuming and performing some actions may lead to undesirable outcomes. However, by using computer simulations, 'trial and error' processes in reactive decision making can be much cheaper, and this leads to the development of RL algorithms in AI [SB98].[2]

The RL framework is a considerable abstraction of the problem of goal-directed learning from interaction. The interaction between a RL agent and its environment is illustrated in Figure 1.3. We can see that the agent and the environment interact at each of a sequence of discrete time steps[3] $t = 0, 1, 2, \cdots$. We refer to each interaction between the agent and the environment as a *learning step*. At each time step t, the agent receives some representation of the environment: state s_t, and on that basis selects an action a_t. In the next time step, in part as a consequence of action a_t, the agent receives its numerical reward r_{t+1}, and finds itself in a new state s_{t+1}. Note that, in different learning steps, performing the same action in the same state may result in different reward and next state; this represents the *stochastic* nature of the environment. This interaction repeats until the agent reaches a *terminal state*, and the procedure ranging from putting the agent at the initial state to the agent reaching a terminal state is called an *episode*.

A *RL experiment* consists of one or many episodes. People (either the designer of the RL system or the user) need to specify when a RL experiment terminates: for example, an experiment can terminate after learning for ten episodes, or terminate after one hour of learning. In this thesis, we restrict our attention to *episodic* RL problems, i.e. RL problems in which each episode consists of a finite number of learning steps and each experiment consists of a finite number of episodes. The target of RL is to identify mappings from states to actions, which return the actions to be performed in each state, such that by following these mappings, the

[2] In this thesis, we focus on the *model-free* RL algorithms, because they do not learn the dynamic of the environment explicitly and are thus most conceptually close to the idea of 'trial and error' and proactive decision making. We will give greater details of model-free RL techniques later in Chapter 2.

[3] The time sequence can also be continuous in RL, but for simplicity, most RL problems are modelled as having discrete time series [SB98]. In this thesis, we focus on the discrete time RL problems.

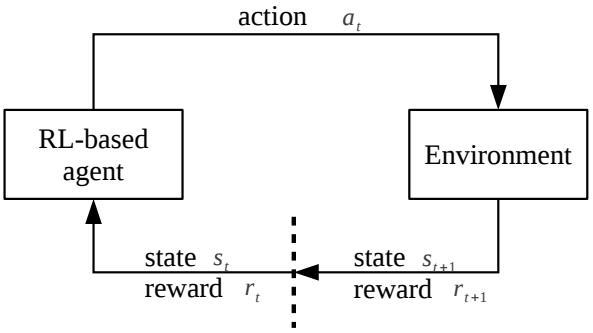


Figure 1.3: The interaction between a RL agent and the environment.

long-term cumulative reward can be maximised. Such mappings are known as *optimal policies* in the RL literature, and RL algorithms amount to any method that is suited to find the optimal policies.

RL can be used to design *multi-agent learning systems*, where there are multiple agents learning independently [BBS08]. In this thesis, we focus on one specific form of multi-agent RL: cooperative multi-agent RL, whose goal is to maximise the whole team's long-term cumulative reward. Single-agent RL problems can be viewed as special cases of cooperative multi-agent RL, where there is only one agent.

Naively, the optimal policies can be found by trying all possible policies, storing the cumulative rewards they lead to, and selecting the policy that leads to the highest long-term rewards. This naive approach is computationally expensive and memory-consuming; as a result, different RL algorithms use different 'tricks' to reduce the time and memory space required to find the optimal policies. However, no matter what tricks RL algorithms use, the computational requirement for obtaining the optimal policies increases exponentially with the size of the state space. This is the famous *curse of dimensionality* faced by all RL algorithms, and it is a main obstacle for applying RL to large-scale and realistic problems, for example in robotics control problems [BS01, SB98].

A widely used idea to combat the curse of dimensionality is to integrate prior knowledge into RL algorithms. By using prior knowledge, RL agents do not need to 'learn from scratch' and can instead take advantage of some existing instructions derived either from people's domain knowledge or other agents' learning experiences. *Potential-Based Reward Shaping* (PBRS) [NHR99] is one of the most widely used techniques for incorporating instructions into RL. It can be applied to

many popular RL algorithms, e.g. Q-Learning [WD92] and SARSA(λ) (a detailed introduction of SARSA(λ) will be given later in Chapter 2). Its idea is to give more promising actions some extra rewards, so that the agent is more encouraged to perform the recommended actions. The most significant advantage of PBRS is that no matter whether the instructions are right or not, the PBRS-augmented RL algorithms are still able to converge to the optimal policy after enough time of learning [NHR99, WCE03]. PBRS techniques have been successfully used in both single-agent (e.g. [WCE03]) and multi-agent (e.g. [GK08]) RL problems. A more detailed introduction of PBRS will be given later in Chapter 2.

Besides using domain knowledge, another popular group of RL techniques to improving the convergence speed is *Hierarchical RL* (HRL). HRL algorithms exploit *temporal abstraction*, where decisions are not required at each step, but rather invoke the execution of temporally extended activities which follow their own policies until termination. Each temporally extended activity can also invoke other activities to solve itself, resulting in a hierarchical reconstruction of the original problem. By doing so, HRL algorithms not only enable quick convergence in complex applications, but also facilitate people to use high-level domain knowledge to instruct learning. On the contrary, all the other non-HRL algorithms are referred to as *flat* RL algorithms. Some popular HRL algorithms like *MAXQ-0* [Die00] have been proved to be able to converge faster than flat RL algorithms in some applications. However, HRL algorithms are also faced with the curse of dimensionality and the classic PBRS techniques cannot be applied to HRL algorithms to accelerate them. As a result, there are still difficulties in applying HRL algorithms to large-scale problems [BM03].

In this thesis, we focus on using prior knowledge to improve both flat and hierarchical RL algorithms' performances, because results of proactive planning can be naturally used as instructions for agents. In particular, we investigate the incorporation of PBRS techniques and HRL algorithms, which can enable the heuristics to be easily integrated into HRL algorithms, without altering its original (hierarchical) optimal policies. We will present the resulting algorithm *PBRS-MAXQ-0* later in Chapter 5.

1.4 Contribution

From the perspective of decision-making research, this thesis contributes a framework for designing cooperative agents that can make decisions jointly using proac-

tive and reactive decision-making strategies; from the perspective of AI research, this thesis contributes a framework for combining logical and statistical AI algorithms; and from a technical perspective, the main contributions of this thesis are presented as follows:

- We propose Argumentation Accelerated RL (AARL), a generic framework for representing domain knowledge by a variant of VAFs, deriving high-quality heuristics from this domain knowledge and incorporating these heuristics into RL algorithms via Potential-Based Reward Shaping (PBRS) techniques. We prove that the heuristics generated by AARL can not only instruct individual learning agent, but also coordinate multiple cooperative independent learning agents.

- We propose an algorithm called *PBRS-MAXQ-0*, which incorporates classical PBRS into a popular HRL algorithm: MAXQ-0 [Die00]. We prove that this algorithm is able to converge to the (hierarchical) optimal policy with arbitrary heuristics (potential values) being used. This algorithm allow for the integration of AARL and HRL algorithms.

- We implement AARL on two widely used RL algorithms: SARSA(λ) and MAXQ-0, and test the resulting algorithms performances in multiple application domains: some are single-agent learning problems, while some others are multi-agent cooperative learning problems. In particular, we use these algorithms to implement a real system called the *Energy Usage Recommendation System* (EURS), and we design a real-data-based simulated user to interact with this system so as to train it and test its effectiveness.

1.5 Structure of This Dissertation

In Chapter 2 we present the background material in Argumentation Theory and Reinforcement Learning. As for Argumentation Theory, we give a detailed and technical introduction of Abstract Argumentation Frameworks and Value-based Argumentation Frameworks. With regard to RL, we introduce the most widely used mathematical models for RL, and present some popular RL algorithms and techniques that are directly used in this thesis.

In Chapter 3, we propose a generic argumentation framework for representing people's domain knowledge, and we prove that by computing the 'winning'

arguments in this framework, heuristics with several desirable properties can be obtained; also, we discuss how to incorporate this framework into RL algorithms to build the Argumentation Accelerated RL (AARL) algorithms.

In Chapter 4, we present our a SARSA(λ)-based AARL, and empirically test its effectiveness in multiple application domains, including singe-agent learning problems (e.g. a Wumpus World problem) and cooperative multi-agent learning problems (e.g. the RoboCup Soccer Takeaway games).

In Chapter 5, we present the PBRS-MAXQ-0 algorithm, which provides a method for integrating recommendations into Hierarchical RL algorithms without altering their original optimal policies. We theoretically prove the convergence property of this algorithm, and empirically test its effectiveness in the *Taxi* problem and a stochastic Wumpus World.

In Chapter 6, we present our MAXQ-based AARL algorithm using the PBRS-MAXQ-0 algorithm given in Chapter 6. Moreover, to comprehensively evaluate the effectiveness of our AARL algorithms, we use both the SARSA(λ)- and MAXQ-based AARL to solve a novel *energy usage recommendation* problem. We use data collected from a real user to simulate this user's behaviour, and use this simulated user to train agents based on these two different RL algorithms, respectively.

In Chapter 7 we conclude this thesis and discuss future works.

1.6 Publications

The work presented in this thesis has resulted in several publications and one paper currently under review:

- (Chapter 3 and 4) Y. Gao, F. Toni and R. Craven. Argumentation-Based Reinforcement Learning for RoboCup Soccer Keepaway. In *Proceedings of 20th European Conference on Artificial Intelligence*, 2012. [GTC12]

- (Chapter 3 and 4) Y. Gao and F. Toni. Argumentation Accelerated Reinforcement Learning for RoboCup Keepaway-Takeaway. In *Theory and Applications of Formal Argumentation, LNCS Volume 8306*, pp 79-94, 2014. [GT13]

- (Chapter 3 and 4) Y. Gao and F. Toni. Argumentation-Based Reinforcement Learning for RoboCup Soccer Takeaway. In *Proceedings of 13th Interna-*

tional Conference on Autonomous Agents and Multiagent Systems, 2014. [GT14b]

- (Chapter 3 and 4) Y. Gao and F. Toni. Argumentation Accelerated Reinforcement Learning for Cooperative Multi-Agent Systems. In *Proceedings of 21st European Conference on Artificial Intelligence*, 2014. [GT14a]
- (Chapter 5 and 6) Y. Gao and F. Toni. Potential-Based Reward Shaping for Hierarchical Reinforcement Learning. *Under review*.

1.7 Statement of Originality

I declare that this thesis was composed by myself and that the work it presents is my own, except where otherwise stated.

2 Background

In the previous chapter, we briefly introduced the basic ideas of Argumentation Theory and Reinforcement Learning (RL). In this chapter, we provide a fuller treatment of these areas. In particular, in Section 2.1, we describe the intuition behind Argumentation Theory in greater detail, give a technical introduction of Abstract Argumentation Frameworks (AFs) and Value-based Argumentation Frameworks (VAFs), and introduce some argumentation *semantics*: criteria used for deciding 'winning arguments'. Then in Section 2.2, we introduce the mathematical models of RL and present some widely used RL algorithms and techniques. At last, we conclude in Section 2.3.

2.1 Argumentation Theory

In this section, we first briefly review Argumentation Theory in Section 2.1.1, so as to obtain further insights into the core idea of Argumentation. Then we introduce AFs, one of the first computational argumentation framework, in Section 2.1.2, and introduce VAFs in Section 2.1.3. Some widely used semantics are also covered in this chapter, and we use the Wumpus World example, which was introduced in Section 1.1, to illustrate how these argumentation frameworks resolve conflicts and obtain the 'winning arguments' by using these semantics. For a more comprehensive introduction of Argumentation Theory, the readers can refer to, e.g. [RS09, CML00, BCD07].

2.1.1 A Brief Overview of Argumentation Theory

Rather than as a paradigm whose study is of independent interest in itself, Argumentation was initially developed as a supportive analytic tool for non-monotonic reasoning [CML00]. However, Argumentation later received attention from researchers in different fields, e.g. law, philosophy, decision making and computer science, and this multi-disciplinary community made important contributions in developing models of argumentation.

Toulmin [Tou58] introduced a conceptual, semi-formal model of argumentation. He considered a representation for legal arguments, in which four elements are distinguished: *claim*, *warrant* (a non-demonstrative reason that allows the claim), *datum* (the evidence needed for using the warrant), and *backing* (the grounds underlying the reason). Counter-arguments are also arguments that may attack any of the four elements of an argument. Toulmin's work is mainly of historical interest now, because it made little progress in formally understanding 'argument acceptability' as we know today.

Pollock's work also had profound influence on the early development of Argumentation Theory [Pol87, Pol91]. Pollock postulated that reasoning operates in terms of *reasons*, and reasons can be assembled to comprise *arguments*. He distinguished two kinds of reasons: *non-defeasible* and *defeasible*. The notion of defeasible reason is defined in terms of a special kind of knowledge called *defeaters*. Defeaters are new reasons that attack the justificatory power that a certain (defeasible) reason has for its conclusion. Pollock refers to two kinds of defeaters, *rebutting* defeaters and *undercutting* defeaters. A rebutting defeater is a reason that attacks a conclusion by supporting the opposite one, while an undercutting defeater is a reason that attacks the connection existing between a reason and a conclusion. The arguments considered in Pollock's work are *structured*: each argument is an inference tree, where the roots of the trees are statements, the leaf nodes are premises, and the paths from leaf nodes to root nodes are aforementioned reasons.

In parallel with this development of the formal logical theory, the early 1990s also saw important uses of argumentation techniques in the computational treatment of legal reasoning (a survey can be found in [BCV97]). The significance of argumentation with regard to non-monotonic logic [SL92, Bre94] and the technical treatment evident in Artificial Intelligence contributions to non-monotonic reasoning and the argumentation-based methodologies offered in the field of legal reasoning found some degree of common ground in the exploitation of logic programming paradigms and knowledge-based systems. It was in this context, building upon the argument-based treatment of 'negation-as-failure' of Kakas, Kowalski and Toni [KKT92], together with Eshghi and Kowalski's work on abductive interpretation [EK89], that the influential contribution of Dung [Dun95] appeared: Dung presented a theory for argumentation whose central notion is the acceptability of arguments. *Abstract Argumentation Frameworks* (AFs) described in [Dun95] are now regarded as providing an important bridge between argumentation theory

as a supporting analytic tool for non-monotonic reasoning and the independent exploitation of argumentation models in wider AI contexts. A fuller treatment of Dung's AFs will be given below in Section 2.1.2.

However, in many contexts, the attack relation between arguments is not the only concern when we decide which arguments are acceptable: arguments also have a force which derives from the value they advance or protect. Thus an argument may be defended not only by counter attacking its attackers, but also by ranking its value more highly than those of its attackers. Moral and legal disagreements should be seen in terms of different preferences for the values which the conflicting arguments defend or promote: this is a major insight of work in jurisprudence such as that of [PB80]. For making decisions in such scenarios, the notion of an argument found in AFs is too abstract, and some extensions of AFs are thus proposed: for example, Modgil [Mod09] integrated meta-level argumentation-based reasoning about preferences into AFs, such that arguments can not only attack arguments, but also can attack attacks; and Bench-Capon [BC03] proposed the *Value-based Argumentation Frameworks* (VAFs), which relate arguments to their supporting values and allow these values to be ranked to reflect the preferences of the audience to which the arguments are addressed. A more detailed description of VAFs will be given below in Section 2.1.3.

2.1.2 Abstract Argumentation Frameworks

An AF [Dun95] is a pair (Arg, Att) where *Arg* is a set of *arguments* and $Att \subseteq Arg \times Arg$ is a binary relation ($(A, B) \in Att$ is read 'A *attacks* B'). Consider $\mathsf{F} = (Arg, Att)$, $S \subseteq Arg$ and $B \in Arg$. S *attacks* B iff some member of S attacks B. S is *conflict-free* iff S attacks none of its members. S *defends* B iff S attacks all arguments attacking B. *Semantics* of AFs are defined as sets of 'winning' arguments, known as *extensions*. For example, S is an *admissible extension* iff S is conflict-free and defends all its elements. We say that an argument is admissible iff it is contained in an admissible extension. Admissible semantics is one of the most widely used semantics, and it captures the idea that a set of 'winning arguments' should be non-contradictory (conflict-free) and be able to protect all its member arguments from external attacks (by defending all its elements). Based on the concept of admissible semantics, other semantics are developed. For example, S is a *preferred extension* iff S is maximally (with respect to $\subseteq$) admissible for F; S is a *complete extension* iff S is conflict-free and $S = \{A | S \text{ defends } A\}$; S

is the *grounded extension* iff S is minimally (with respect to $\subseteq$) complete. Similarly, we say that an argument is preferred, complete or grounded iff it is contained in at least one preferred, complete or grounded extension, respectively. From the definitions of these semantics, we can see that each preferred/complete/grounded extension is also admissible, but not vice versa; each preferred extension is a complete extension, but not vice versa; and the grounded extension is the intersection of all complete extensions (so the grounded extension is unique) [Dun95]. In this thesis, we mainly use the preferred and grounded semantics to select 'winning arguments'. Also note that there can be several preferred extensions 'credulously' accepting arguments that can be coherently defended, whereas the grounded extension is guaranteed to be unique [Dun95], consisting solely of the uncontroversial arguments and being thus 'sceptical'. Both the preferred extensions and the grounded extension can be empty.

As a concrete illustration, let us consider the Wumpus World example introduced in Section 1.1. Credulously, we may think that performing *go_up* and *go_left* are both acceptable, because both these actions are safe, and the other actions are dangerous (*go_right* and *go_down*). Sceptically, however, we may think that neither *go_up* nor *go_left* is 'winning', because neither of them is uncontroversially better than the other. We build an AF for this problem and use different semantics to select the 'winning' arguments. The AF is represented as the graph in Figure 1.2. Formally, this AF can be represented by a pair (Arg_T, Att_T), where

- $Arg_T = \{L, U\}$,
- $Att_T = \{(L, U), (U, L)\}$.

Set $\{L\}$ ($\{U\}$) is an admissible extension, because it is conflict-free and is able to defend all its elements. By definition of the complete and preferred extensions, we can also easily see that $\{L\}$ ($\{U\}$) is a preferred and a complete extension. Also, note that the empty set $\emptyset$ is also an admissible extension and a complete extension. Neither $\{L\}$ nor $\{U\}$ is the grounded extension, because they are not minimally (with respect to $\subseteq$) complete; instead, the grounded extension in the AF is the empty set $\emptyset$. The different extensions obtained by the preferred and grounded semantics reflect the credulous and sceptical nature of these two semantics: when there are multiple conflicting but 'equally good' arguments, preferred extensions will give all of them (maybe in different extensions), whereas the grounded extension will contain none of them. Also, we can see that the winning actions iden-

tified by the preferred and grounded extensions are consistent with our credulous and sceptical reasoning results presented earlier in this subsection, respectively.

2.1.3 Value-Based Argumentation Framework

Value-based Argumentation Frameworks (VAFs) [BC03] are an extension of AFs by incorporating values as well as preferences over them into AFs. The values used in VAFs, in many cases (e.g. in the example follows shortly and in the RoboCup Keepaway/Takeaway games presented later in Chapter 4), can be viewed as people's expectations of the results of some arguments. The key idea is to allow for attacks to succeed or fail, depending on the relative worth of the values promoted by the competing arguments. Given a set V of values, an *audience Valpref* is a strict partial order over V (corresponding to the preferences of an agent), and an *audience-specific VAF* is a tuple $(Arg, Att, V, val, Valpref)$, where (Arg, Att) is an AF and $val : Arg \rightarrow V$ gives the values promoted by arguments. In VAFs, the ordering over values, *Valpref*, is taken into account in the definition of extensions. In *Valpref*, for two values $v_1, v_2 \in V$, 'v_1 is more preferred than v_2' is denoted as $v_1 >_v v_2$. The *simplification* of an audience-specific VAF is the AF (Arg, Att^{-}), where $(A, B) \in Att^{-}$ iff $(A, B) \in Att$ and $val(B)$ is not more preferred than $val(A)$ in *Valpref*, i.e. $val(B) \not>_v val(A)$. $(A, B) \in Att^{-}$ is read '*A defeats B*'. Then, (acceptable) extensions of a VAF are defined as (acceptable) extensions of its simplification (Arg, Att^{-}). We refer to (Arg, Att^{-}) as the *simplified AF derived from* $(Arg, Att, V, val, Valpref)$.

As an illustration, still consider our earlier Wumpus World problem, and suppose that we add a new piece of domain knowledge that 'the exit is in the left-hand side of this agent'. Given this domain knowledge, performing action *go_left* potentially increases the speed to escape from this world and, therefore, promotes a value 'exiting the Wumpus World quickly and safely'. Formally, we extend the earlier illustrative AF with the value set V_T consisting of values:

- v_{exit}: exiting the world quickly and safely,
- v_{safe}: avoiding being killed.

Let val_T be such that *L* promotes v_{exit} (i.e. $val_T(L) = v_{exit}$) and *U* promotes v_{safe}. From the description of each value and the promotion relation, we can see that each value actually corresponds to the (expectation of the) outcome of some argument(s). For example, *L* promotes v_{exit} because we expect that performing

go_left can lead to the results described in v_{exit}. As for the preference over the values, as an example, we let $Valpref_T$ give $v_{exit} >_v v_{safe}$. Then we obtain a VAF $(Arg_T, Att_T, V_T, val_T, Valpref_T)$ and derive the simplified AF (Arg_T, Att_T^-), where $Att_T^- = Att \setminus \{(U,L)\} = \{(L,U)\}$. The attack from *U* to *L* is eliminated because the value v_{safe} promoted by *U* is lower ranked than the value v_{exit} promoted by *L*. Because *U* cannot defend itself from the attack from *L*, the unique preferred and grounded extension for the simplified AF is $\{L\}$, indicating that, when performing *go_left* can lead to quicker exit of this world, *go_left* is the winning action given the preference explained by $Valpref_T$.

However, we note that some requirements in VAFs restrict its usage in knowledge representation in RL problems. For example, in VAFs, the value ranking (i.e. *Valpref*) is required to be a strict partial order; however, in some real applications (e.g. the RoboCup Keepaway/Takeaway games in Chapter 4), the domain expert(s) may rank multiple values with the same preference, indicating that these values are equally preferred. Therefore, in order to use VAFs to represent people's domain knowledge in RL problems, we may need to relax some requirements in standard VAFs and propose a new variant of VAFs.

2.1.4 Computation of Argumentation Semantics

Computing the winning arguments according to different semantics is generally a complex problem. Different types of computation are listed in Table 2.1, and the computational complexity for the grounded and preferred semantics are summarised in Table 2.2. From these two tables we can see that computing preferred extensions is generically more expensive than computing the grounded extension. Note that the simplification in VAFs (see Section 2.1.3) can be accomplished by traversing all attacks once; therefore, for an AF with n arguments, $n \in \mathbb{N}^*$, there can be at most n^2 attacks, so the complexity of the simplification computation is $O(n^2)$.

Standard computational mechanism for computing argumentation semantics, e.g. [DMT07] and [DT09], are defined using trees and disputes, respectively, and only construct relevant parts of extensions. *Answer Set Programming* (ASP) [Lif02] can instead be used to support the full computation of extensions [TS11], and there exist several efficient ASP solvers, including Smodels[1], DLV[2] and clasp[3].

[1] http://www.tcs.hut.fi/Software/smodels
[2] http://www.dlvsystem.com
[3] http://potassco.sourceforge.net

Table 2.1: Some types of computation in abstract argumentation frameworks (adjusted from Table 5.1 in [DW09]). Given an abstract argumentation framework $AF = (Arg, Att)$, $E_{\mathcal{T}}(AF)$ is the set of all extensions of semantics type $\mathcal{T}$.

Problem	Instance	Question
$VER_{\mathcal{T}}$	$AF = (Arg, Att), S \subseteq Arg$	Is $S \in E_{\mathcal{T}}(AF)$?
$CA_{\mathcal{T}}$	$AF = (Arg, Att), x \in Arg$	Is there any x s.t. $x \in S, S \in E_{\mathcal{T}}(AF)$?
$NE_{\mathcal{T}}$	$AF = (Arg, Att)$	Is there any $S \in E_{\mathcal{T}}(AF), S \neq \emptyset$?

Table 2.2: Complexity of some types of computation for the preferred and grounded semantics (adjusted from [DW09]).

Problem	$\mathcal{T}$ = preferred	$\mathcal{T}$ = grounded
$VER_{\mathcal{T}}$	coNP-complete	P
$CA_{\mathcal{T}}$	NP-complete	P
$NE_{\mathcal{T}}$	NP-complete	P

Based on ASP and its solvers, multiple methodologies have been developed to compute different argumentation semantics, e.g. [EGW08, FW09, DGWW11, NCO08, WN09]. *ASPARTIX* [WN09] is a widely used ASP-based tool, and it can compute many widely used argumentation semantics (including all semantics we have mentioned in Section 2.1.2) in both AFs and VAFs. In the remainder of this thesis, we use ASPARTIX, implemented on DLV, to compute the preferred and grounded extensions.

2.2 Reinforcement Learning (RL)

As we have discussed in Section 1.3, the Reinforcement Learning (RL) framework is a considerable abstraction of the problem of goal-directed learning from interaction, and RL algorithms amount to techniques to find the best policy that leads to the goal. In this section, we give a more detailed description of the RL framework from two perspectives: the problems we aim to tackle, and the techniques we use to tackle these problems.

The Wumpus World problem we introduced in Section 1.1 is an illustration of the problems that RL tackles: there is a goal state the agent wants to achieve, but

the agent is currently in a state different from the goal state. Also, the agent has certain actions available in each state, such that by performing different actions, the agent can move to different new states and receive different rewards/punishments. The problem is how to find the best sequence of actions that can lead the agent to the goal state with highest rewards. This problem is known as a *sequential decision-making* problem [Lit96]. *Markov Decision Processes* (MDPs) and Semi-MDPs (SMDPs) provide formal models of these problems: they provide a simple yet powerful method for modelling states, actions, state transitions and rewards received in each transition. As a result, most RL work uses MDPs or SMDPs to model the underlying problems. A detailed introduction of these two models is presented below in Section 2.2.1.

If complete information of a MDP/SMDP is known a priori, the sequential decision-making problem can be effectively solved by using classic planning techniques, e.g. *Dynamic Programming* [Bel56a] or *Teleo-Reactive* [Nil94]. However, in many problems, especially some real problems (e.g. the RoboCup Soccer games that we will introduce later in Section 3.1.1), the agent virtually has no prior knowledge about the problem and it has limited sensory information during its interaction with the environment. Classic planning techniques have difficulties in solving these problems, and RL algorithms are thus developed to combat the 'incomplete knowledge' challenge. RL techniques are based on Dynamic Programming, but they offer two important advantages over Dynamic Programming [SB98, Die00]: **(1)** RL methods are online, i.e. the policies are evaluated and improved simultaneously; therefore, once RL finds that a policy is bad, it will reduce the time used to evaluate this policy but, instead, turn to look for some other better policies. In other words, the online property of RL allows RL to focus only on the policies that are important and spend less time on those less promising ones. **(2)** RL can employ function approximation algorithms (e.g. neural networks [MWS95] or tile coding [SB98]) to represent the agent sensory information, so that the experiences learnt in one state can be used in some other 'close' states as well, and this helps RL to scale better; this property is especially useful in continuous environment. Later in this section, we introduce two widely used RL algorithms in Section 2.2.2 and 2.2.3.

Although RL algorithms can effectively obtain the optimal policies in certain problems, for example a small scale Wumpus World, in large-scale and real applications, standard RL algorithms usually suffer from the *curse of dimensionality*: the number of possible policies grows exponentially with the number of possible

states. To combat this curse, some techniques for accelerating RL are proposed. We introduce two widely used such techniques in Section 2.2.4 and Section 2.2.5.

Since it is impossible and unnecessary to cover all RL algorithms or techniques in this Section, we only cover those that are directly used in the remainder of this thesis. We refer the reader to [SB98] and [BM03] for a more detailed and comprehensive introduction of RL.

2.2.1 Markov Decision Processes (MDPs) and Semi-MDPs

Most RL research is based on the formalism of Markov Decision Processes (MDPs), because they provide a simple framework in which to study basic RL algorithms and their properties. In this thesis, we focus on discrete-time, countable action and finite length episode MDPs, because they are simple yet powerful enough to model the RL problems we aim to tackle. A MDP is a tuple (S, A, P, R, γ), where S is the set of states, A is the set of actions, $P(s'|s, a)$ is the *transition probability* of moving from state s to s' by performing action a, $R(s'|s, a)$ gives the *immediate reward* received when action a is executed in state s, moving to state s', and $\gamma \in [0, 1]$ is the *discount factor*, which is used when computing long-term rewards. The goal of planning in a MDP is to find a policy $\pi : S \rightarrow A$, specifying for each state the action to take, so as to maximise the expected discounted sum of future rewards. We call polices that meet this requirement the *optimal policies*, denoted as π^*. The *value function* $V^\pi(s)$ represents the expected discounted sum of rewards that will be received by following π starting in state s:

$$V^\pi(s) = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots | s_t = s, \pi]$$

where $r_t = R(s_{t+1}|s_t, \pi(s_t))$ is the immediate reward received in time step t, s_t is the state visited in t. Besides the value function, also known as *V-values*, another, arguably more widely used function in RL is the *state-action-value* function, often referred to as *Q-values*. $Q^\pi(s, a)$ is the expected discounted return for executing action a in state s and thereafter following policy π:

$$Q^\pi(s, a) = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots | s_t = s, a_t = a, \pi].$$

where a_t stands for the action performed in time step t. A famous result about value functions is that V-values satisfy the *Bellman equation* for a fixed policy π:

$$V^{\pi}(s) = \sum_{s'} P(s'|s, \pi(s))[R(s'|s, \pi(s)) + \gamma V^{\pi}(s')]. \tag{2.1}$$

From the Bellman's equation, we can see that the value function of a state s (i.e. $V^{\pi}(s)$) can be recursively defined by the value function of the next state (i.e. $V^{\pi}(s')$). By doing this, the problem of obtaining the current state's value function can be solved by obtaining the next state's value function and obtaining the immediate reward (this is trivial, because immediate rewards can be directly observed). Since the next state is 'closer' to the termination state, computing the value function of the next state is slightly simpler than computing the current state's value function. From a computer science perspective, the Bellman's equation breaks the decision problem in MDPs into smaller sub-problems, and suggests that the decision making problems in MDPs have *optimal substructure*, i.e. an optimal solution can be constructed efficiently from optimal solutions of its sub-problems [Bel56b, CLR$^+$01].

From a more mathematical perspective, Bellman's equation guarantees the convergence property of the value function. To be more specific, Equation (2.1) can be rewritten as:

$$T^{\pi}V^{\pi}(s) = V^{\pi}(s)$$

where T^{π} is the *Bellman operator underlying* π such that

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

Note that this is a linear system of equations in V^{π} and T^{π} is an *affine linear operator* [KMA97]; if $0 < \gamma < 1$ then T^{π} is a *maximum-norm contraction* and the fixed-point equation $T^{\pi}V^{\pi}(s) = V^{\pi}(s)$ has a unique solution for any $s \in S$ [Ber95].

For an optimal policy π^*, the value function is:

$$V^*(s) = \sum_a P^{\pi^*}(a|s) \sum_{s'} P(s'|s, a)[R(s'|s, a) + \gamma V^*(s')]$$

where $V^*(s)$ is the V-value at state s by following the optimal policy π^*, and

$P^{\pi^*}(a|s)$ is the probability of performing action a in state s in the optimal policies. The Bellman equation also holds for the Q-values:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma Q^{\pi}(s', \pi(s'))]. \tag{2.2}$$

We denote the Q-value for performing action a at state s and thereafter following policy π^* as $Q^*(s,a)$. The Bellman equation also holds for $Q^*(s,a)$:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma \max_{a'} Q^*(s',a')].$$

Given $Q^*(s,a)$ for all state-action pairs (s,a), the optimal policy π^* at state s can be obtained as follows:

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a).$$

Semi-MDPs (SMDPs) are a generalisation of MDP in which the actions can take variable amounts of time to complete. We restrict our attention to discrete-time SMDP, and let the random variable $\tau \in \mathbb{N}^+$ denote the number of time steps action a takes in state s. Similar to MDPs, a SMDP is also represented as a tuple (S, A, P, R, γ), but functions P and R are dependent on the random variable τ. In particular, the transition probability function P is extended to be the joint distribution of the result state s' and the number of time steps τ when action a is performed in state s: $P(s', \tau|s, a)$. Similarly, the reward function is extended to $R(s', \tau|s, a)$, representing the reward received by performing action a in state s and reaching state s' after τ number of time slots. The value function (V-value) also satisfies the Bellman's equation in SMDP:

$$V^{\pi}(s) = \sum_{s',\tau} P(s', \tau|s, \pi(s))[R(s', \tau|s, \pi(s)) + \gamma^{\tau} V^{\pi}(s')],$$

so does the action function (Q-value):

$$Q^{\pi}(s,a) = \sum_{s',\tau} P(s', \tau|s, a)[R(s', \tau|s, a) + \gamma Q^{\pi}(s', \pi(s'))].$$

For illustrating how to use a MDP to model a RL problem, let us still consider the Wumpus World application introduced in Section 1.1.[4] Recall that the agent's

[4] Here we only illustrate the idea of MDP, namely all actions in this illustration take one and only

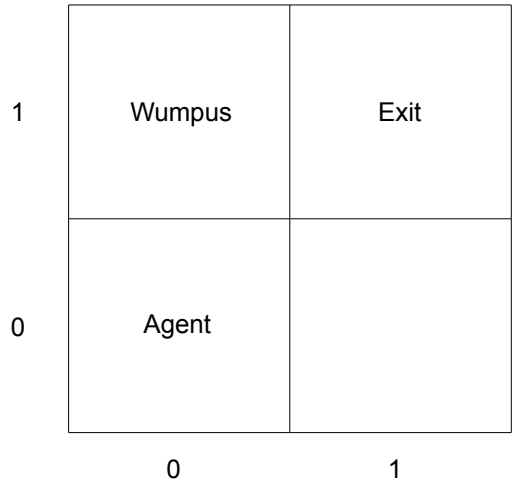


Figure 2.1: A 2×2 Wumpus World.

goal is to arrive at the exit square without visiting any square with a pit or Wumpus. We can use the following MDP to model this problem:

- **State set** S. Note that a state should distinguish the current scenario from other scenarios in this problem, and it should also include available and needed information for decision-making in the current scenario. Given this understanding, we use a vector to represent a state, consisting of the coordinate location of the current square, and two boolean values indicating whether the agent feels breeze and stench in this square, respectively. For example, in the Wumpus World in Figure 2.1, the agent's current state is $< 0, 0, F, T >$, where F and T are shorthands of false and true, respectively. Note that when the current state is $< 0, 1, F, T >$, an episode is terminated, because in this state the agent is eaten by the Wumpus. We call the states that terminate an episode as *terminal states*.

- **Action set** A. As we have described in Section 1.1, four actions are available to the agent in each state: *go_left*, *go_right*, *go_up* and *go_down*. As a result, the action set A consists of these four actions.

- **Transition function** P. The transition function describes, by performing action a in state s, which state s' the agent will reach in the next time slot.

one time step to finish. An illustration of SMDPs will be presented later in Section 2.2.3.

In the Wumpus World problem, an action will move the agent one square towards the intended direction if there is no wall in the intended direction. In other words, the transition function is implicitly defined by the actions. For example, still consider the scenario in Figure 2.1, we can see that $P(< 1, 0, F, F > | < 0, 0, F, T >, \textit{go_right}) = 1$ because performing *go_right* in $< 0, 0, F, T >$ will definitely move the agent to $< 1, 0, F, F >$.

However, in many real problems, for example the *RoboCup Soccer Keepaway/Takeaway* games (introduced later in Chapter 3), the transition function is unknown, i.e. the agent does not know what next state it will reach by performing an action in a state. Some RL algorithms are able to learn the best policy without prior knowledge of the transition function P, including the RL algorithms we will introduce later in this section.

- **Reward function** R. Note that rewards are used to instruct the agent to achieve the tasks of the problem. Given the goal of this problem, we let the agent receive -1 for each action it performs. Also, we let it receive an additional reward +501 when it arrives at the exit, and receive an additional reward -999 when it steps into a square with a Wumpus. The reason that we give a negative reward to each action is to reduce the total number of steps needed to exit the world. Also note that some RL algorithms (e.g. SARSA and MAXQ as we will introduce below in Section 2.2.2 and 2.2.3, respectively) do not require any knowledge of the reward function a priori.

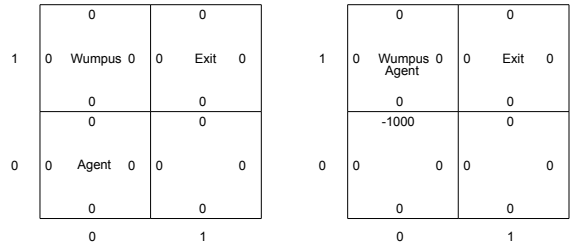
- **Discount factor** γ. Recall that γ defines how 'short-sighted' the agent is: the bigger the γ is, the agent gives more weights to the long-term rewards. In this example problem, we assume that our agent is extremely 'far-sighted' and, therefore, we let $\gamma = 1$.

A naive method to find the optimal policy of a MDP is to perform all possible polices, compare their Q-value (i.e. the long-term cumulative reward) at each state-action pair (s, a), and select the policy that has the highest $Q(s, a)$ in each (s, a) pair as the optimal policy. However, the number of policies grows exponentially with the number of possible states. For example, consider the MDP above for the Wumpus World. Since the agent can perform four actions in each state, there are $4^{|S|}$ policies in total. This is a reflection of the well-known *curse of dimensionality* faced by all RL algorithms for solving (S)MDPs.

2.2.2 SARSA Learning Algorithm

SARSA [RN94] is a simple yet powerful RL algorithm, and it has been used in many application domains, for example the RoboCup Keepaway and Takeaway games, which will be introduced in detail later in Chapter 6. Particularly noticeable features of SARSA include its fast convergence speed and its model-free property [SB98]. Some factors that contribute to the fast convergence property of SARSA include: **(1)** SARSA is an *on-policy* learning method, meaning that the policy being evaluated is also the policy being used by the agent, and, meanwhile, the agent continuously updates the policy according to the latest trajectory. This allows SARSA to select the policy that is optimal with respect to the latest learning experiences. **(2)** SARSA is a *Temporal Difference* (TD) based learning method, meaning that it updates the Q-value of each state-action pair by using the next state-action pairs' Q-value, and this allows SARSA to back-propagate the delayed stimulus information (i.e. rewards) quickly (this will be illustrated shortly in this subsection). Also, SARSA does not require any prior knowledge about the transition function of the underlying MDP, and it does not explicitly reconstruct the transition function (in RL literature, this kind of RL algorithms are referred to as *model-free* algorithms): all it needs to store are the Q-values (illustrated below). This model-free property allows SARSA to be used in applications where the transition function is unknown. Also note that SARSA can be used in either MDP or SMDP problems, i.e. the actions in SARSA can take one or more time slots to finish.

Pseudo code of SARSA is presented in Algorithm 1. A walk-through of this algorithm will be given below. We first describe some learning parameters in this algorithm: $\alpha \in \mathbb{R}, \alpha \in [0, 1]$ is the *learning rate* parameter, which controls how significantly the current Q-value will be changed after each update; γ is the discount factor; in line 4 and 7, ϵ*-greedy* is an *action selection policy*, which chooses the action to perform in the current state: by using ϵ-greedy, the action with highest Q-value is chosen with probability $1 - \epsilon$, and a uniformly random action is chosen with probability ϵ. As the learning proceeds, both ϵ and α values should be gradually decreasing to 0, meaning that less exploration is needed and Q-values should be updated in smaller steps. Note that the purpose of choosing some random actions with probability ϵ is to prevent the learning agent trapped in a local sub-optimal policy, and ϵ-greedy is a method to trade off between *exploration* (i.e. choosing an action that does not have the highest Q-value in the current state) and



(a) Step 1 (b) Step 2

Figure 2.2: The first episode of the SARSA-based learning in the 2×2 Wumpus World in Figure 2.1. Each square has four numbers, each representing the Q-value of the corresponding action in that square.

exploitation (i.e. choosing the action that has the highest Q-value in the current state). Actually, the ϵ-greedy action selection policy can be viewed as an approximation of the *Greedy in the Limit with Infinite Exploration* (GLIE) policy [Thr92], which requires that **(1)** each action is executed infinitely often in every state that is visited infinitely often, and **(2)** in the limit (i.e. after infinitely many episodes of learning), the policy is greedy with respect to the Q-value function with probability 1.

Algorithm 1 The SARSA algorithm (adjusted from [SB98]).

Initialise $Q(s, a)$ for all states s and actions a arbitrarily
while the experiment does not terminate **do**
  Initialise the current state s
  Choose action a in s by using ϵ-greedy
  while s is not a terminal state **do**
    Execute action a, observe the next state s' and immediate reward r
    Choose action a' from s' by using ϵ-greedy
    $Q(s, a) := (1 - \alpha)Q(s, a) + \alpha(r + \gamma Q(s', a'))$
    $s := s'$
    $a := a'$
  end while
end while

We illustrate how this algorithm works by considering the Wumpus World in Figure 2.2. Note that in this Wumpus World game, each episode ends when the agent is killed (either by falling into a pit or being eaten by a Wumpus), and a

new episode immediately starts with the environment reset. The Wumpus is in square $(0, 1)$, the exit is in square $(1, 1)$, and the agent is put in square $(0, 0)$ at the beginning of each episode. An *experiment* consists of multiple episodes, and before the start of the first episode, without loss of generality, we initialise all Q-values to 0 (line 1 in Algorithm 1). Now we consider the first episode. According to line 3, we first initialise the starting state of the agent as $< 0, 0, F, T >$ (note that the first boolean value indicates whether the agent feels breeze in the current state, while the second one indicates whether it feels stench). Then we choose the action to be performed in s, by using ϵ-greedy (line 4). For illustration purpose, we let $\epsilon = 0$, such that the agent always chooses the action that can maximise the Q-value in the current state. Since all actions' Q-values in this state are 0 now, the agent can choose any action according to ϵ-greedy. Suppose the agent chooses to perform *go_up*, and it goes to square $(0, 1)$ and receives a reward of -1000, i.e. $s' =< 0, 1, F, T >$ and $r = -1000$ (line 6). Then the agent chooses the action to be performed in s' (line 7). Once again, according to ϵ-greedy, the agent can select any action because all actions' Q-values in s' are 0. Suppose the agent also chooses *go_up* in s'. Given the current state s, current action a, reward r, next state s' and next action a', we update the Q-value of state-action pair (s, a) (line 8). For simplicity, we let $\alpha = 1$. Since $Q(s, a) = Q(s', a') = 0$, we can easily see that the new $Q(s, a)$ value is -1000. We then update the current state s and the current action a (line 9 and 10), and re-enter the loop between line 5 and line 11. Now $s =< 0, 1, F, T >$, and it is a termination state because there is a Wumpus in this square. Therefore, the algorithm quits the loop and this episode ends. The Q-values until now are given in Fig 2.2(b). We can see that each iteration of the loop between line 5 and line 11 is actually an interaction between the RL agent and the environment; thus, each iteration of this loop is a *learning step* (see Section 1.3). The first episode has only one learning step.

After the first episode finishes, the second episode starts immediately (line 3). The initial state is the same as in the first episode. However, when selecting the action to be performed in s, *go_up* will not be chosen because its Q-value is the lowest among all actions' Q-values in s. Because all the other three actions' Q-values are 0, the agent can randomly select any of those actions. The current situation is illustrated in Figure 2.3(a). Suppose the agent chooses $a =$ *go_right* (line 4); then it will receive reward $r = -1$ and go to a new state $s' =< 1, 0, F, F >$ (line 6). Since s' has not been visited before, all actions' Q-values in s' are 0 and, therefore, the agent will choose a random action a' in s'. Suppose it chooses

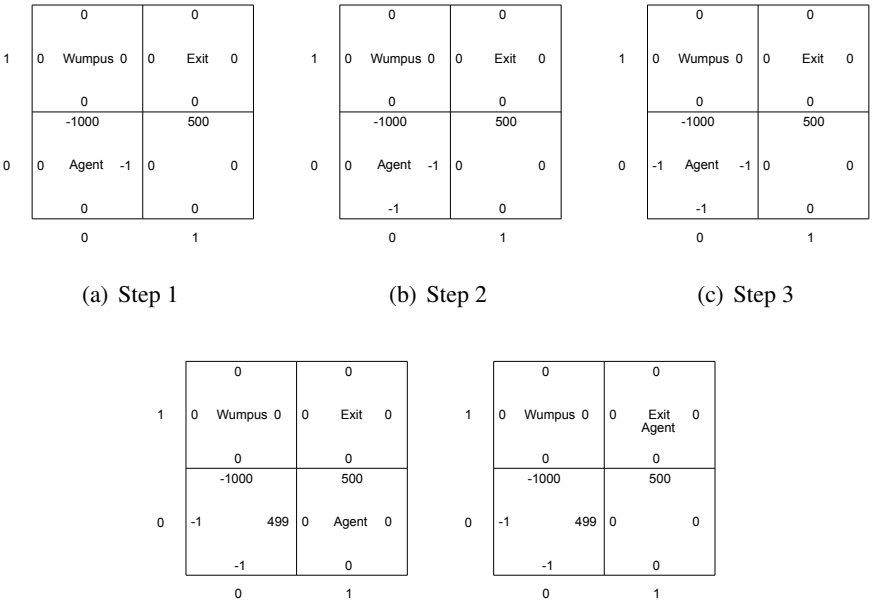


(a) Step 1 (b) Step 2 (c) At the end of step 2

Figure 2.3: The second episode of the SARSA-based learning.

$a' = go_up$. According to line 8, we can easily obtain that the new $Q(s, a)$ value is -1. The algorithm then updates s and a (lines 9 and 10) and re-enters the loop starting from line 5. Note that, in step 2, $a = go_up$ and $s =< 1, 0, F, F >$ (Figure 2.3(b)). By performing a in s, the agent receives reward +500, and moves into a new state $s' =< 1, 1, F, T >$. Once again, since all actions' Q-values in s' are 0, a random action a' is chosen: let us suppose it is go_up. We can easily obtain that $Q(s, a) = +500$. Since the agent arrives at the exit, the second episode ends now (Figure 2.3(c)). Thus the second episode has two learning steps.

Now the agent has its third episode in this experiment. The initial state s is still $< 0, 0, F, T >$. In state s, the agent will first choose action go_left or go_down, because the Q-values of these two actions at s are still 0, whereas the other two actions' Q-value are all negative (Figure 2.4(a)). Suppose the agent chooses $a = go_down$, it will receive $r = -1$ and the new state $s' = s$. Easily we can see that $Q(s, go_down)$ will be updated as -1. In step 2 (Figure 2.4(b)), the agent will choose go_left to perform because it has the highest Q-value in state s, and will receive $r = -1$, remain in the same state (i.e. $s' = s$), and update $Q(s, go_left) = -1$. In step 3 (Figure 2.4(c)), all actions except go_up have the same Q-value, and we assume the agent chooses $a = go_right$. A reward $r = -1$ will be received, and the new state is $s' =< 1, 0, F, F >$. In s', action go_up will be selected (i.e. $a' = go_up$), because its Q-value is +500 while all the other actions' Q-values are 0. So the value of $Q(s, a)$ is updated as follows: $Q(s, a) = -1 + 1 \times [-1 + 1 \times 500 - (-1)] = 499$. In step 4 (Figure 2.4(d)), the agent performs go_up in state $< 1, 0, F, F >$, and this will lead the agent to the exit, which terminates this episode (Figure 2.4(e)). This episode has four learning steps.



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) At the end of step 4

Figure 2.4: The third episode of the SARSA-based learning.

In all episodes afterwards, the agent will first perform *go_right*, and then perform *go_up* to reach the exit. So all episodes afterwards have two steps, and we can see that this is the best policy in this Wumpus World. Also note that, given our specific setting ($\alpha = 1, \gamma = 1, \epsilon = 0$), $Q(s, a)$ for all state-action pairs (s, a) will not change in all following episodes. In other words, in this experiment, SARSA converges to the optimal policy after three episodes of learning. From this illustration, we can have a direct feeling of how SARSA converges.

After illustrating how SARSA learns, we briefly discuss a very important operation in SARSA: the Q value updating, as presented in line 8 in Algorithm 1. Since actions are chosen according to these Q values, the Q value updating plays a key role in SARSA. We can see that given the current state s, current action a, reward r, next state s' and next action a', the value of $Q(s, a)$ is updated by using both the existing $Q(s, a)$ value as well as the new estimation of $Q(s, a)$: $r + \gamma Q(s', a')$. The fact that $r + \gamma Q(s', a')$ is an estimation of $Q(s, a)$ can be seen from Equation (2.2): if the transition function P is deterministic, i.e. performing a at s will lead to one specific state s', then $Q(s, a) = r + \gamma Q(s', a')$. However, in most real applications, P is not deterministic and, therefore, by receiving each r, the SARSA algorithm only changes the old $Q(s, a)$ value by a small step α, such that after many rounds of update, the value of $Q(s, a)$ can asymptotically approach its true value. It has been proved [RN94] that when α asymptotically approaches 0 at certain rates, and when the agent uses a GLIE action selection policy, after long enough time of learning, $Q(s, a)$ of any state-action pair (s, a) will converge to $Q^*(s, a)$ with probability 1.[5]

Also, we have discussed in the first paragraph in this subsection that SARSA's quick convergence is partly due to its TD-based updating, and this allows SARSA to 'back-propagate' delayed information more quickly. We now describe what is the information back-propagation and why SARSA's TD nature helps to accelerate it. Let us revisit step 3 in the third episode in our aforementioned illustration (Figure 2.4(c)). In this step, $s =< 0, 0, F, T >$, $a =$ *go_right*, $r = -1$, $s' =< 1, 0, F, F >$ and $a' =$ *go_up*. We know that performing a' in s' receives a big positive reward $+500$, and this is a piece of important information we want to 'propagate', because our goal is to maximise the long term rewards. Because in TD-based Q-value updating (line 8 in Algorithm 1), $Q(s, a)$ will be updated by using $Q(s', a')$, the information contained in $Q(s', a')$ will be propagated back-

[5] To be more specific, [RN94] proved that when $\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t = \infty, \lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t^2 < \infty$, where α_t is the learning step in the tth episode, the convergence is guaranteed.

ward to $Q(s, a)$. The updated $Q(s, a)$ value is 499, as shown in Figure 2.4(d), and we can see that the reward of reaching the exit (+500) has been successfully propagated to $Q(s, a)$. We can see that by each learning step, the reward can propagate one state back. Intuitively, a more effective back-propagation should be able to back-propagate a reward to all states on the trajectory that leads to the current state. This more effective back-propagation can be achieved by the *Eligibility Traces* technique [SS96], which will be introduced later in Section 2.2.4.

Although RL algorithms like SARSA are widely used and have been proved to be effective in some application domains, they by no means completely circumvent the curse of dimensionality. To further combat this curse, some more advanced RL algorithms are developed, including the *Hierarchical RL* algorithms introduced next.

2.2.3 Hierarchical RL (HRL) and MAXQ

Research in classical planning has shown that hierarchical methods such as hierarchical task networks [CT91], macro actions [FHN72, Kor85] and state abstraction methods [Sac74, Kno90] can provide exponential reductions in the computational cost for obtaining good plans. However, many RL algorithms, e.g. SARSA, are '*flat*' methods in the sense that they treat the state space as a huge flat search space. This means that the paths from the start state to the goal state are very long, and the length of these paths determines the cost of learning and planning, because information about future rewards must be propagated backward along these paths.

Considerable research has been devoted to *Hierarchical Reinforcement Learning* (HRL) techniques [Sin92, PR97]. A feature shared by all HRL techniques is that HRL algorithms exploit *temporal abstraction*, where decisions are not required at each step, but rather invoke the execution of temporally extended activities which follow their own policies until termination [BM03]. Each temporally extended activity can also invoke other activities to solve itself, resulting in a hierarchical reconstruction of the original problem. Among all HRL algorithms, *MAXQ* [Die00] is one of the most widely used, because unlike some other popular HRL algorithms, e.g. *Options* [SPS99] and *Hierarchical Abstract Machines* [PR97] which treat the original problem as a whole, MAXQ decomposes the original problem into multiple sub-problems, and each sub-problem can be solved by invoking solutions of other sub-problems. This allows solutions of multiple sub-problems to be learnt simultaneously. In this thesis, we focus on MAXQ as the

representative of HRL algorithms.

In MAXQ, each temporally extended activity (or sub-problem) is denoted as a *sub-task*. In particular, if this sub-task can invoke other sub-tasks to solve itself, it is referred to as a *composite* sub-task; otherwise, it is referred to as a *primitive* sub-task, or more simply, an action. MAXQ decomposes the overall value function for a policy into a collection of value functions for individual sub-tasks. A core MDP M is decomposed into a set of sub-tasks $H_M = \{M_0, M_1, \cdots, M_n\}$, forming a hierarchy with M_0 the root, i.e. solving M_0 solves M. Actions taken to solve M_0 may be primitive actions or policies that solve other composite sub-tasks, which can in turn invoke primitive actions or policies of other sub-tasks. Throughout this subsection, we let i range over the indices of all sub-tasks in the sub-task set $H_M = \{M_0, M_1, \cdots, M_n\}$, i.e. $i \in \{0, \cdots, n\}$.

For a composite sub-task M_i, the sub-tasks and primitive actions into which M_i is decomposed are called the *children* of M_i, and M_i is called the *parent* of its children sub-tasks. Formally, a sub-task is defined as follows[6]:

Definition 1. Given a MDP $M = (S, A, P, R)$, a sub-task M_i in a MAX decomposition of M is a pair, $< T_i, A_i >$, defined as follows:

1. $T_i \subseteq S$ is the set of *terminal states*, while $S \setminus T_i$ is the set of *active states*, also denoted as S_i. The policy for sub-task M_i can only be executed if the current state s is in S_i. If, at any time that sub-task M_i is being executed, the MDP M enters a state in T_i, then M_i terminates immediately.

2. A_i is a set of 'actions' that can be performed to achieve sub-task M_i. These actions can either be primitive actions from A, or they can be other sub-tasks. We denote members in A_i as *children* of sub-task M_i. No sub-task is allowed to invoke itself recursively either directly or indirectly.

Each primitive action $a \in A$ is a *primitive sub-task* such that a is always executable and always terminates immediately after execution.

From the definition above, we can see that each sub-task M_i corresponds to a SMDP $\mathbf{D}_i = (S_i, A_i, P_i, R_i)$, where S_i and A_i are defined in M_i, $P_i(s', \tau|s, a)$ describes the probability of performing action $a \in A_i$ in state $s \in S_i$ and leading to state $s' \in S$ by taking τ number of time slots, and $R_i(s', \tau|s, a)$ describes the reward received by performing action $a \in A_i$ in state $s \in S_i$ and leading to

[6]Definition 1 is adjusted from Definition 2 in [Die00].

state $s' \in S$ by taking τ number of time slots. Note that most MAXQ-based RL algorithms (e.g. *MAXQ-0*, which will be introduced below in this subsection) do not need any knowledge of functions P_i and R_i a priori.

A MAXQ decomposition is usually represented by an acyclic directed graph called *MAXQ task graph*. The parent-children relations are concisely represented in these graphs. The root of a MAXQ task graph represents the original MDP M, and this means that by solving the root sub-task, the whole problem is solved (i.e. the optimal policies are obtained). As an illustration, we design a MAXQ decomposition for an extended Wumpus World. We extend the Wumpus World application introduced in Section 1.1 by adding some *gold* into this World and five actions for the agent: *shoot_left*, *shoot_right*, *shoot_up*, *shoot_down* and *pickup*. The first four new-added actions will shoot an arrow to the square next to the agent in the intended direction, and if there is a Wumpus in that square, the arrow kills the Wumpus, and this will earn the agent some reward. Note that the agent has an infinite number of arrows, but shooting an arrow without killing any Wumpus will receive negative reward. Action *pickup* will collect a gold if there is a gold in the current square (each square can have at most one gold). The agent can see *glitter* iff there is a gold in the current state, and performing a *pickup* without collecting any gold will receive negative rewards. The ultimate goal of this extended Wumpus World is 'to collect all gold and arrive at the exit, without being killed by a Wumpus or a pit'. Also note that we add two new elements in the state vector (see Section 2.2.1 for the state vector): an integer indicating the number of uncollected gold in the World, and a boolean value indicating whether the agent sees glitter in the current square.

To achieve the ultimate goal, some sub-goals should be achieved first: for example, we can easily identify two sub-goals: to collect all gold safely, and to navigate to the exit safely. These two sub-goals, in turn, involve achieving some other sub-sub-goals: to collect all gold involves navigation, killing Wumpuses and picking up gold (*pickup*); and to exit involves navigation and killing Wumpuses. Navigation involves performing actions *go_left*, *go_right*, *go_up* and *go_down* (we call these four actions *navigation actions* in the Wumpus World), and killing Wumpuses involves performing *shoot_left*, *shoot_right*, *shoot_up* and *shoot_down*(we call these four actions *hunting actions*). We can see that these sub-goals have a hierarchical relation, in the sense that a 'high-level' sub-task can invoke certain 'lower-level' sub-tasks to solve itself.

Given this hierarchy, we design the MAXQ task graph shown in Figure 2.5,

where each rectangular shape represents a sub-task, either composite or primitive. Note that sub-task *Navigate*(x, y) has two arguments, indicating which position (x, y) the agent attempts to navigate to. Actually, a sub-task with argument(s) is a shorthand for many similar sub-tasks: for example, *Navigate*(x, y) actually represents $X \times Y$ sub-tasks, where X and Y are the number of columns and rows in the Wumpus World, respectively.

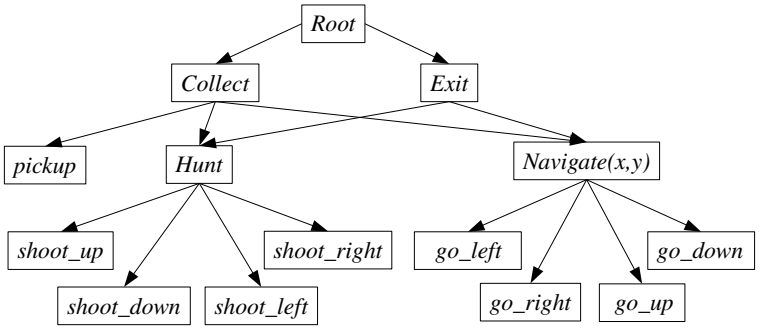


Figure 2.5: The task graph for the stochastic Wumpus World problem.

According to the goal of each sub-task, we can specify its terminal states. In the extended Wumpus World, we let *Root* terminate when all gold are collected and the agent arrives at the exit, or when the agent is killed. In other words, *Root* terminates iff an episode terminates. We let *Collect* terminate when all golds in the map are collected, and let *Exit* terminate when the agent successfully arrives at the exit, because the sub-goals of these two sub-tasks are 'collecting all gold safely' and 'arriving at the exit safely', respectively. Sub-task *Hunt* terminates if the agent cannot feel stench in the current square, *Navigate*(x, y) terminates when the agent arrives at square (x, y). All primitive actions, by Definition 1, terminate immediately. Note that a child sub-task automatically 'inherits' the terminal states of the parent sub-task that is invoking this child sub-task in the current execution. For example, suppose in one execution, *Collect* invokes *Navigate*(x, y); once the agent reaches a terminal state of *Collect*, sub-task *Navigate*(x, y) also terminates immediately.

A MAXQ decomposition defines sub-tasks, particularly the parent-children relation between sub-tasks. However, the execution of children sub-tasks is *unordered*, namely as far as the current state s is in S_i, sub-task M_i can invoke any of its children sub-tasks in any order. To specify the invoking strategy for each sub-task, we define a policy π_i for each sub-task M_i, such that $\pi_i : S_i \rightarrow A_i$ is

a function, and $\pi_i(s)$ indicates which child of M_i is invoked in state s. The set containing all sub-tasks' policies, $\{\pi_0, \cdots, \pi_n\}$, is called a *hierarchical policy* for the MAXQ decomposition H_M, usually denoted as π. So the goal of a MAXQ algorithm is to obtain the optimal hierarchical policy π^* such that each member π_i^* in π^* is the optimal policy for SMDP $\mathbf{D}_i$.

To obtain the optimal hierarchical policy, we need to define the expected cumulative rewards received by each sub-task's policy, so as to select the optimal policy for each sub-task. Let $V^\pi(i, s)$ denote the cumulative reward of performing sub-task M_i in state s following hierarchical policy π, where i is a shorthand for M_i. When i terminates at s, sub-task i is not allowed to be performed in s and, therefore, $V^\pi(i, s) = 0$; otherwise,

$$V^\pi(i,s) = \begin{cases} \sum_{s'} P(s'|s,i)R(s'|s,i) & \text{if } i \text{ is primitive,} \\ Q^\pi(i, s, \pi_i(s)) & \text{if } i \text{ is composite.} \end{cases} \tag{2.3}$$

Note that $P(s'|s,i)$ and $R(s'|s,i)$ are actually shorthands of $P(s', \tau = 1|s,i)$ and $R(s', \tau = 1|s,i)$, respectively, because primitive actions take one and only one time step to finish. When M_i is a primitive action, $V^\pi(i,s)$ is the expected immediate reward of performing action M_i in state s; otherwise, $V^\pi(i,s)$ is $Q^\pi(i, s, \pi_i(s))$, the expected return for M_i of performing sub-task $M_{\pi_i(s)}$ in state s and then following π until M_i terminates:

$$Q^\pi(i, s, \pi_i(s)) = V^\pi(\pi_i(s), s) + C^\pi(i, s, \pi_i(s)) \tag{2.4}$$

where the *completion function* $C^\pi(i, s, \pi_i(s))$ defined as

$$C^\pi(i, s, \pi_i(s)) = \sum_{s', \tau} P_i^\pi(s', \tau|s, \pi_i(s))\gamma^\tau V^\pi(i, s') \tag{2.5}$$

gives the expected return for completing sub-task M_i after sub-task $M_{\pi_i(s)}$ terminates. P_i^π is the transition probability in M_i given a policy π. Equations (2.3), (2.4) and (2.5) provide a recursive way to write the value function given a hierarchical policy.

To illustrate how the cumulative rewards are recursively decomposed into C, V and Q values, we consider a scenario in the extended Wumpus World game shown in Figure 2.6. Rewards for navigation actions are the same as we described above in Section 2.2.1, and killing a Wumpus or collecting a gold receives +100. However, performing a hunting action without killing any Wumpus or perform-

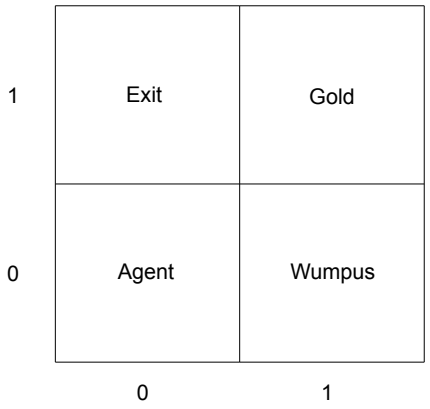


Figure 2.6: An example scenario in the extended Wumpus World.

ing *pickup* without collecting any gold receives -50 and -10, respectively. Now suppose we have obtained the optimal hierarchical policy π^*, and we compute the V^{π^*}, C^{π^*} and Q^{π^*} value for each sub-task in each state. By following the optimal policy, the agent will use five steps to escape from this World: **(1)** perform *shoot_right* to kill the Wumpus; **(2)** perform *go_right* to arrive at square $(1, 0)$; **(3)** perform *go_up* to arrive at square $(1, 1)$; **(4)** perform *pickup* to collect gold; and **(5)** perform *go_left* to arrive at the exit square. Rewards received by the agent in these five steps are 100, -1, -1, 100 and 500, respectively. So by summing up each step's rewards, we can easily obtain that $V^{\pi^*}(\textit{Root}, s_1) = 698$, where $s_1 =< 0, 0, F, T, F >$ (the third boolean value indicates whether the agent sees glitter) is the initial state of the agent.

Now we view this optimal policy from a hierarchical perspective: we can see that within sub-task *Root*, the agent first invokes *Collect* and then invokes *Exit*. In *Collect*, it first invokes *Hunt*, and then invokes *Navigate*$(1, 1)$, and invokes *pickup* at last. Let us look into the execution of *Hunt*. We can see that the agent performs *shoot_right* in *Hunt* and kills a Wumpus. So $V^{\pi^*}(s_1, \textit{shoot_right}) = 100$. After performing *shoot_right*, the stench disappears and *Hunt* also terminates. In other words, *shoot_right* is the last child invoked in *Hunt*, and therefore $C^{\pi^*}(\textit{Hunt}, s_1, \textit{shoot_right}) = 0$. So, according to Equation (2.3), $V^{\pi^*}(\textit{Hunt}, s_1) = Q^{\pi^*}(\textit{Hunt}, s_1, \textit{shoot_right})$; in addition, according to Equation (2.4), we obtain $Q^{\pi^*}(\textit{Hunt}, s_1, \textit{shoot_right}) = V^{\pi^*}(\textit{shoot_right}, s_1) + C^{\pi^*}(\textit{Hunt}, s_1, \textit{shoot_right}) =$

100.

After *Hunt* terminates, the state is $s_2 =< 0, 0, F, F, F >$. Since *Collect* has not terminated yet, it needs to invoke a child sub-task at s_2. We can see that it invokes *Navigate*$(1, 1)$, which, in turn, invokes some navigation actions. We omit the execution details of *Navigate*$(1, 1)$, and directly obtain that V^{π^*}(*Navigate*$(1, 1), s_1) = -2$, because we know the agent takes two steps to move to square $(1, 1)$. After *Navigate*$(1, 1)$ terminates, the state is $s_3 =< 1, 1, F, F, T >$, and *Collect* chooses to perform *pickup* at s_3. The primitive action *pickup* terminates immediately, and *Collect* also terminates because all gold are collected. Therefore, V^{π^*}(*Collect*, s_1) = V^{π^*}(*Hunt*, s_1)+C^{π^*}(*Collect*, s_1, *Hunt*), and the second addend can be expanded as V^{π^*}(*Navigate*$(1, 1), s_2) + C^{\pi^*}$(*Collect*, s_2, *Navigate*$(1, 1)$), where C^{π^*}(*Collect*, s_2, *Navigate*$(1, 1)$) is the rewards received after *Navigate*$(1, 1)$ finishes and before *Collect* terminates. Because only *pickup* is performed after *Navigate*$(1, 1)$ finishes, we can obtain that C^{π^*}(*Collect*, s_2, *Navigate*$(1, 1)) = V^{\pi^*}$(*pickup*, s_3). Since performing *pickup* in state s_3 collects a gold, we can easily see that V^{π^*}(*pickup*, s_3) $= 100$, so we obtain that V^{π^*}(*Collect*, s_1) $= 100 - 2 + 100 = 198$.

After *Collect* terminates, the state is $s_4 =< 1, 1, F, F, F >$, and *Root* selects *Exit*, which in turn invokes *Navigate*$(0, 1)$. We omit the execution of *Exit* and directly obtain that V^{π^*}(*Exit*, s_4) $= 500$. *Root* and *Exit* both terminate when the agent arrives at the exit, and this episode ends. Because *Root* invokes *Collect* and *Exit* during this episode, we have V^{π^*}(*Root*, s_1) $= V^{\pi^*}$(*Collect*, s_1) + V^{π^*}(*Exit*, s_4). By now, we obtain that V^{π^*}(*Root*, s_1) $= 198 + 500 = 698$, and we can see that the value of V^{π^*}(*Root*, s_1) computed by step-by-step reward cumulation equals the value computed by using C and V values. The whole hierarchical execution process is summarised in Figure 2.7.

Based on this recursive definition, Dietterich [Die00] proposes the **MAXQ-0** algorithm to learn the optimal hierarchical policy from sample trajectories. Pseudo code of this algorithm is presented in Algorithm 2. The main part of this algorithm is the **MAXQ-0** function. This function basically does two things: **(1)** updating C and V values by using rewards collected along the sample trajectories, and **(2)** exploiting the current C and V values to produce a policy, which is best with respect to the current C and V values, and using this policy to generate new trajectories.

We now go through the details of function **MAXQ-0**(i, s), where i is a subtask and s is a state. Since **MAXQ-0** recursively invokes itself, we first look at the base case of this function, namely when i is a primitive action (lines 4 to 7

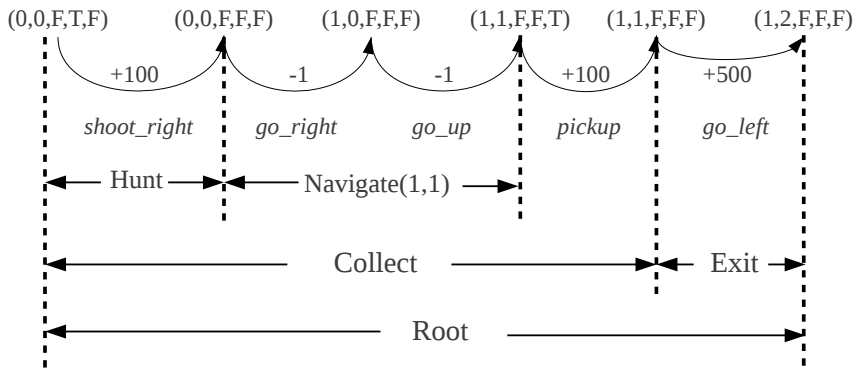


Figure 2.7: The hierarchical execution process of the Wumpus World example shown in Figure 2.6.

in Algorithm 2). First, the algorithm executes action i, observe reward r and new state s' (line 5). Then the algorithm updates the value of $V(i,s)$ according to the rule given in line 6. Note that $\alpha_t(i)$ is the learning rate parameter for sub-task i in the tth episode. $\alpha_t(i) \in \mathbb{R}, \alpha_t(i) \in [0,1]$. This parameter is required to decrease after each episode [Die00], i.e. $\alpha_t(i) > \alpha_{t+1}(i)$, and the purpose of doing this is to ensure that after an infinitely long time of learning, when C values converge to C^{π^*}, they will not be subject to any change. After the V value gets updated, the algorithm then pushes the current s into list seq (line 7), and returns this list so that s can be used to update C values for composite sub-tasks that invoke i.

Then we look into the recursive part of this function (lines 8 to 22). When it does not terminate, the function first chooses a child of i: a, according to an action-selection policy *Ordered GLIE* (OGLIE) (details given shortly in this subsection). Then the function recursively invokes itself, by using a and the current s as the argument (line 11). After **MAXQ-0**(a,s) returns, two tasks are finished: **(1)** all states visited during the execution of a are returned, with the last visited state in the head of the list, and **(2)** all i's descendants' C values and V values are updated. Also, the execution of a at s transits the agent to a new state, s' (line 12). Then the algorithm estimates the value of $V(i,s')$, so as to perform the TD-based updating of C values (see Section 2.2.2 for TD, more detailed descriptions of this updating will be given shortly). For the time being, we skip the details of function **Evaluate**(a,s), and only assume that this function will return the estimated value of $V(i,s')$ by using all i's descendants' C and V values. Details of function **Evaluate** will be given shortly. Now we have the current state s, current action

Algorithm 2 MAXQ-0 with all-states updating.

1: /*Recursive part: function MAXQ-0*/
2: function **MAXQ-0**(Sub-task i, State s)
3: $seq := ()$ /*initialise seq as an empty list*/
4: **if** i is a primitive action **then**
5: execute i, receive reward r and observe next state s'
6: $V_{t+1}(i,s) := (1-\alpha_t(i)) \cdot V_t(i,s) + \alpha_t(i) \cdot r_t$
7: push s onto the beginning of seq
8: **else**
9: **while** sub-task i does not terminate at s **do**
10: choose an action a according to an Ordered GLIE policy
11: $childSeq :=$ **MAXQ-0**(a,s) /* recursive call */
12: observe next state s'
13: $V_t(i,s') :=$ **Evaluate**(i,s')
14: $N := 1$
15: **for** each s in $childSeq$ **do**
16: $C_{t+1}(i,s,a) := (1-\alpha_t(i))C_t(i,s,a) + \alpha_t(i)\gamma^N V_t(i,s')$
17: $N := N+1$
18: **end for**
19: **append** $childSeq$ onto the front of seq
20: $s := s'$
21: **end while**
22: **end if**
23: **return** seq

24: /*Main Programme*/
25: initialise all V and C values arbitrarily
26: **while** the experiment does not terminate **do**
27: **MAXQ-0**(root sub-task 0, starting state s_0)
28: **end while**

a, next state s', and the expected reward to be received within i at the next state s': $V(i,s')$. By using this information, the algorithm will update all visited states' completion functions (lines 15 to 18). This update is TD-based because it uses the next state's V-value: $V(i,s')$ to update the current state' C-value. Note that N (appearing in lines 14, 16 and 17) counts the time gap between the current state and the state being updated, because all states in seq are stored in the last-in-first-out order. When sub-task i terminates in state s, function **MAXQ-0** will return all states visited during the execution of i in s to the parent sub-task j of i (line 23), so that the completion functions $C(j,s,i)$ can be updated in function **MAX-0**(j,s).

Two issues remain unexplained about function **MAXQ-0**: **(1)** how function

Evaluate(i, s) evaluates the value function $V(i, s)$, and **(2)** how the function selects a child sub-task of i (line 10). Now we discuss the first issue. Pseudo code of function **Evaluate**(i, s) is presented in Algorithm 3. Now we look into this function to see how it evaluates the value of $V(i, s)$, where i is a sub-task and s is a state. When i is a primitive action, the function simply returns the value of $V(i, s)$ (line 3). Otherwise, the function constructs $V(i, s)$ by exploiting the recursive decomposition structure of $V(i, s)$ (lines 5 to 9). From Equation (2.3), we can see that $V(i, s)$ can be decomposed into the sum of two addends: $V(a, s)$ and $C(i, s, a)$, where a is the child of i that is performed in s. The $C(i, s, a)$ value is directly stored by the algorithm, and $V(a, s)$ can be computed by recursively invoking the function itself (line 6). By obtaining the estimated value of all $V(a, s)$, the algorithm greedily chooses a child a^* such that a^* can maximise the value of $V(i, s)$ (line 8), and, at last, returns the greedy estimation of $V(i, s)$ (line 9).

Algorithm 3 The function used in MAXQ-0 to greedily evaluate a sub-task.

```
 1: function Evaluate(Sub-task i, State s)
 2: if i is a primitive action then
 3:     return V_t(i, s)
 4: else
 5:     for all child sub-task a of i do
 6:         V_t(a, s) := Evaluate(a, s)
 7:     end for
 8:     a* := argmax_a[V_t(a, s) + C_t(i, s, a)]
 9:     return V_t(a*, s) + C_t(i, s, a*)
10: end if
```

It has been proved that by using an *OGLIE* (Ordered GLIE, in which ties between sub-tasks with the same V-values are broken in a fixed order) action selection policy, MAXQ-0 is guaranteed to converge to the hierarchical optimal policy [Die00]. In practice, people usually use an *ordered ϵ-greedy* action-selection policy to approximate the OGLIE policy. An ordered ϵ-greedy policy is very similar to ordinary ϵ-greedy; the only difference is that ordered ϵ-greedy breaks ties according to a predefined fixed order. Pseudo code of the ordered ϵ-greedy policy is presented in Algorithm 4, whose inputs are a sub-task i, a state s, a real value $\epsilon \in \mathbb{R}, \epsilon \in [0, 1]$ and an order $pref$ over all sub-tasks. We can see that if the input sub-task i is a primitive action, this function simply returns i itself (line 3); otherwise, it computes the value functions of all children of i, also by invoking function **Evaluate** (line 6). After that, it greedily selects children of i and puts them in $list$

(line 9). With a probability of ϵ, a random child of i is returned (line 12); for the other $1-\epsilon$ probability, the function returns the best child of i (lines 14 to 17).

Algorithm 4 The ordered ϵ-greedy sub-task-selection policy.

```
1:  function chooseChild(Sub-task i, State s, Float ε, SubtaskRanking pref)
2:  if i is a primitive action then
3:      return i
4:  else
5:      for all child sub-task a of i do
6:          V_t(a, s) := Evaluate(a, s)
7:      end for
8:      list := ()
9:      insert all argmax_a[V_t(a, s) + C_t(i, s, a)] into list
10:     initialise a random number r ∈ ℝ, r ∈ [0, 1]
11:     if r < ε then
12:         return a random child of i
13:     else
14:         if there are more than one sub-task in list then
15:             return the sub-task in list that is most preferred according to pref
16:         else
17:             return the only sub-task in list
18:         end if
19:     end if
20: end if
```

Another point worth highlighting is the TD-based updating rule of the completion functions C (between lines 15 and 17 in Algorithm 2). We show the similarity between this C-value updating rule and the Q-value updating rule in SARSA (line 8 in Algorithm 1) so as to obtain some insights into how C values are updated in **MAXQ-0**. As we have discussed in Section 2.2.2, Q is updated as follow: $(1-\alpha)Q(s,a) + \alpha(r + \gamma Q(s',a'))$, where $Q(s,a)$ is the old Q value of state-action pair (s,a) and $r+\gamma Q(s',a')$ is a new estimation of the true value of $Q(s,a)$. This structure also applies to the updating rule for C values: according to Equation (2.5), $\gamma^N V(i,s')$ is an estimation of the true value of $C(i,s,a)$. Note that N indicates the time gap between the next state s' and the state being updated: s. Since the first state in seq is the last visited state in sub-task a, its time gap to the next state s' is 1; the earlier a state s is visited, the longer the time gap between s and s'. Because each primitive action takes exactly one time step to finish, the number of state transitions exactly equals the length of the time gap between states.

From Algorithm 2 and Algorithm 3, we can see that the quantities that need to be

stored in the MAXQ-0 learning process are just the C values for all composite sub-tasks and the V values for all primitive actions. All other values will be constructed dynamically by invoking function **Evaluate**, according to the decomposition rule.

Compared with flat RL algorithms, e.g. SARSA, MAXQ-0 has been proved to have better performances in certain applications domains, e.g. the *Taxi* problems [Die00] (details of this application will be given in Chapter 5). Also, because the hierarchical structure itself results from people's domain knowledge, the MAXQ algorithms provide a new approach for domain experts to use their knowledge to instruct the learning procedure. However, because of the introduction of composite sub-tasks, the action space of MAXQ is larger than that of the flat RL algorithms, and this may result in slower convergence speed [BM03]. Furthermore, in each composite sub-task, all children are allowed to be chosen for arbitrary numbers of times. These factors may cause the slow convergence speed of MAXQ algorithms, even in relatively small-scale problems [Die00]. One of the main contributions of this thesis is to introduce a method for tackling this problem, as presented in Chapter 5.

2.2.4 Eligibility Traces

As we have discussed above in Section 2.2.2, the quick convergence property of SARSA is partly attributed to SARSA's TD-based Q-value updating mechanism, which accelerates the reward back-propagation. However, the back-propagation in SARSA is just one step, meaning that in each learning step, a reward can be propagated only one state backward. An ideal reward back-propagation method should be able to achieve the 'whole trajectory' propagation: once the current state-action pair receives its immediate reward, all historical state-action pairs on the trajectory leading to the current state should be able to share a proportion of the current reward, and the proportion is decided by the discount parameter γ as well as their distance to the current state: the longer the distance, the less proportion a historical state-action pair shares, because the reward is discounted by γ per learning step. The *Eligibility Traces technique* (ET) [SS96] is developed to achieve this ideal back-propagation.

An eligibility trace is a temporary record of the occurrence of an event, such as the visiting of a state or the execution of an action. The trace marks the memory parameters associated with the event as eligible for undergoing learning changes.

When a TD error[7] occurs, only the eligible state-action pairs are assigned credit or blame for the error. Thus, eligibility traces help bridge the gap between events and training information. Like TD methods themselves, ET is a basic mechanism for temporal credit assignment. Almost any TD-based RL algorithms can be combined with Eligibility Traces [SB98].

Now we briefly describe how ET helps SARSA to achieve the 'whole trajectory back-propagation'. The ET-augmented version of SARSA is called SARSA(λ) [SS96], where $\lambda \in \mathbb{R}, \lambda \in [0, 1]$ is a parameter used to control how much credit should be delivered back to previous state-action pairs' Q-values. The pseudo code of SARSA(λ) is presented in Algorithm 5. The basic structure of SARSA(λ) is very similar to that of SARSA (Algorithm 1). Here we only highlight the augmented part. Initially, all state-action pairs' eligibility trace are initialised as 0 (line 3). On each state-action pair visit, its corresponding eligibility trace is set to be 1, meaning that this state-action pair has just been visited (line 10). Note that δ in line 9 represents the difference between the new estimation of $Q(s, a)$ and the existing value of $Q(s, a)$. This value is the information (immediate reward plus TD error) we want to propagate backwards to all previous state-action pairs on the trajectory. To this end, we update all *eligible* state-action pairs' Q-values according to the rule given in line 12. A state-action pair (s, a) is eligible iff its corresponding eligibility trace $e(s, a) \neq 0$. In the updating rule in line 12, we can see that the second addend on the right-hand side is a product of three values: the learning step α, the information we want to back-propagate δ, and $e(s, a)$, the value indicating to what extent state-action pair (s, a) is eligible for receiving the latest information. After the update, the eligibility trace of (s, a) is discounted by $\gamma\lambda$, meaning that (s, a) is less eligible for receiving the latest reward in the next update, because the distance (i.e. number of learning steps) from pair (s, a) to the latest pair increases.

To further understand the relation between SARSA(λ) and standard SARSA, we consider a special case of SARSA(λ): SARSA(0). We can see that when $\lambda = 0$, once a new state-action pair (s', a') is obtained, only the previous state-action pair (s, a) is updated, because only its corresponding eligibility trace is non-zero; all earlier pairs' Q-values are not affected, because their eligibility traces are all 0, after updating according to line 13 in Algorithm 5 where $\lambda = 0$. As a result, we can see that SARSA(0) is exactly the same as standard SARSA, and SARSA(λ)

[7] $Q(s', a') - Q(s, a)$, where s, a is the current state-action pair, and s', a' is the next state-action pair, is called a TD error. We can see that in SARSA, Q-values are updated by using this TD error (line 8 in Algorithm 1)

Algorithm 5 The SARSA(λ) algorithm with replacing eligibility traces (adjusted from [SB98]).

1: Initialise $Q(s, a)$ for all state s and action a arbitrarily
2: **while** the experiment does not terminate **do**
3: Initialise $e(s, a) = 0$ for all s and a
4: Initialise the current state s
5: Choose action a from s by using ϵ-greedy
6: **while** s is not a terminal state **do**
7: Execute action a, observe the next state s' and immediate reward r
8: Choose action a' from s' by using ϵ-greedy
9: $\delta := r + \gamma Q(s', a') - Q(s, a)$
10: $e(s, a) := 1$
11: **for** all s and a **do**
12: $Q(s, a) := Q(s, a) + \alpha\delta e(s, a)$
13: $e(s, a) := \gamma\lambda e(s, a)$
14: **end for**
15: $s := s'$
16: $a := a'$
17: **end while**
18: **end while**

is essentially a generalisation of standard SARSA, in the sense that by tuning λ between 0 and 1, we can tune to what extent we want to back-propagate the current information to previous state-action pairs.

To illustrate the advantage of SARSA(λ) over standard SARSA (i.e. SARSA(0)), we consider again the illustrative Wumpus World example we introduced in Section 2.2.2. This time we use SARSA(1) to learn, and suppose that, initially, the agent is in the state shown in Figure 2.3(a). All Q-values are initialised as shown in this figure. So the current state is $s_1 =< 0, 0, F, T >$ (line 4 in Algorithm 5). Suppose the agent randomly selects *go_right* at s_1 (line 5), so $a = a_1 =$ *go_right*. By performing a, the agent moves to $s' = s_2 =< 1, 0, F, F >$, and receives reward $r = -1$ (line 7). Suppose that in s', the agent chooses *go_up*, so $a' = a_2 =$ *go_up* (line 8). Easily, we obtain that $\delta = -1$ (line 9), and we update $e(s, a) = e(s_1, a_1) = 1$ (line 10). For simplicity, we let $\alpha = \gamma = \lambda = 1$. Because all e values except $e(s_1, a_1)$ are 0, only $Q(s_1, a_1)$ is updated in line 12. The new value of $Q(s_1, a_1)$ is -1. After updating $s = s_2$ (line 15) and $a = a_2$ (line 16), the algorithm moves to the next learning step. Until now, the Q-values are the same as the Q-values updated by using standard SARSA, as shown in Figure 2.3(b).

In the second learning step, by performing $a =$ *go_up*, the agent reaches the exit

state $s' = s_3 =< 1, 1, F, T >$, and receives reward $r = 500$ (line 7). Suppose $a' = go_up$ (line 8), so $\delta = 500 + 0 - 0 = 500$ (line 9). Then we update the eligibility trace of the current state-action pair: $e(s, a) = e(s_2, a_2) = 1$ (line 10). Recall that, until now, only two state-action pairs' eligibility traces are non-zero: $e(s_1, a_1) = e(s_2, a_2) = 1$. Given these two non-zero eligibility traces, we can update their corresponding Q-values (line 12): $Q(s_1, a_1) = -1 + 1 \times 500 \times 1 = 499$, and $Q(s_2, a_2) = 0 + 1 \times 500 \times 1 = 500$. Then this episode ends. We can easily see that in all episodes afterwards, the agent will perform the optimal policy, and the Q-values will not change any longer. Compared with standard SARSA, SARSA(1) does not need all learning steps illustrated in Figure 2.4.

2.2.5 Potential-Based Reward Shaping (PBRS)

Reward shaping, as a medium to convey domain knowledge into RL, has been proved to be effective to improve the convergence speed of RL algorithms in both single-agent and multi-agent problems [NHR99, DGK11]. However, despite its effectiveness in many experiments, if used improperly, reward shaping can also mislead the learning process [RA98]. To deal with such problems, *Potential-Based Reward Shaping* (PBRS) is proposed by [NHR99] as the difference of some potential function Φ over the current state s and the next state s'. By integrating PBRS into MDP, the value function in state s following policy π becomes:

$$\hat{V}^{\pi}(s) = E[\sum_{t}^{\infty} \gamma^t (r_t + \gamma\Phi(s_{t+1}) - \Phi(s_t)) | s_t = s, \pi]$$

The augmented part, $\gamma\Phi(s_{t+1}) - \Phi(s_t)$, is called the *potential-based shaping reward*. The purpose of integrating potential values in such a form is to ensure that after a sufficiently long time of learning, V^{π} can be easily recovered from $\hat{V}^{\pi}(s)$. For example, suppose $s_0, s_1, \cdots$ is a trajectory generated by a policy π. Given this trajectory, we can extend $\hat{V}^{\pi}$ as follows:

$$\begin{aligned}
\hat{V}^{\pi}(s) &= [r_0 + \gamma\Phi(s_1) - \Phi(s_0)] + \gamma[r_1 + \gamma\Phi(s_2) - \Phi(s_1)] \cdots \\
&= -\Phi(s_0) + r_0 + \gamma r_1 + \cdots \\
&= V^{\pi}(s) - \Phi(s_0). \qquad (2.6)
\end{aligned}$$

We can see that most potential values cancel each other out, and only the first state's potential value is left. So the original value function V^{π} can be easily re-

constructed from the potential-values-augmented value function $\hat{V}^{\pi}$. So we can see that by integrating potential values into the original MDP in this way, regardless of the potential values being used, the optimal policy of the original MDP is not altered. This 'robustness' property of PBRS is arguably the most significant advantage over the traditional reward shaping techniques [RA98] and other approaches for integrating heuristics into RL, e.g. [BRC08].

However, in PBRS, since the potential values are only based on states, they cannot provide hints on which actions are more promising in some state. To tackle this problem, Wiewiora et al. [WCE03] extended PBRS to the case of shaping functions based on both states and actions: $\Phi(s, a)$, and proposed a method of integrating $\Phi(s, a)$ into RL algorithms: the *look-ahead advice* (LA) technique[8]. They proved that by using look-ahead advices, arbitrary potential values can be incorporated into RL without altering its optimal policy, and they empirically showed that LA can effectively improve the convergence speed of SARSA(λ), by using sophisticated potential values.

For example, by using LA in SARSA(λ) (we call the resulting algorithm *LA-SARSA(λ)*), line 9 in Algorithm 5 is changed as follows:

$$\delta := r + \gamma Q(s', a') - Q(s, a) + \gamma \Phi(s', a') - \Phi(s, a), \tag{2.7}$$

and when greedily choosing actions in state s, the greedy action a^* maximises the sum of the Q-value and the potential value at state s:

$$a^* := \max_a [Q(s, a) + \Phi(s, a)]. \tag{2.8}$$

This rule for selecting actions is referred to as a *biased ϵ-greedy* action selection policy, because the 'greedy' actions are selected to maximise the Q-values biased by a potential value.

As an illustration, let us consider using LA-SARSA(0) in the Wumpus World scenario shown in Figure 2.2(a). Suppose we have the following potential values $\Phi(< 0, 0, F, T >, \textit{go_right}) = \Phi(< 1, 0, F, F >, \textit{go_up}) = 100$, and all other state-action pairs' potential values are 0. These potential values can be viewed as a numerical representation of the domain knowledge that 'in $< 0, 0, F, T >$ ($< 1, 0, F, F >$), performing *go_right* (*go_up*) is more promising'. In episode 1

[8] Wiewiora et al. [WCE03] have also proposed another method for integrating potential values into RL called *look-back advice*. However, they claimed that look-back advice cannot be proved theoretically sound; therefore, we focus on the LA technique in this thesis.

(Figure 2.2(a)), when selecting actions (line 5 in Algorithm 5), by using Equation (2.8), we can see that the agent will choose to perform *go_right* because all actions' Q-values are 0 in this state, and the potential value of *go_right* is much higher than other actions'. Similarly, in episode 2, when the agent is at $< 1, 0, F, F >$, the agent will choose *go_up*. So by using these potential values, it only takes two learning steps to find the optimal policy. This example illustrates how PBRS techniques can use potential values to represent heuristics, and use these potential values to accelerate RL algorithms.

Despite the effectiveness and simplicity of PBRS in many applications, it has also been widely recognised that the performance of PBRS heavily depends on the quality of the potential values, and proposing high quality heuristics to derive instructive potential values can be challenging [Mar07]. However, this problem has received little research attention until now. One of our contributions in this thesis is to introduce a generic framework for deriving high quality heuristics and potential values from people's domain knowledge, even when the knowledge contains conflicts.

2.3 Conclusion

In this chapter, we review two families of Artificial Intelligence (AI) techniques for decision making: Argumentation Theory and Reinforcement Learning (RL). We not only present their technical details, but also illustrate their working processes via the Wumpus World application, so as to provide some further insights into their advantages and limitations. In particular, we show that Argumentation Theory is strong in representing knowledge, resolving the conflicts therein and performing logic reasoning over the knowledge, but has no systemic approach in dealing with sequential decision-making problems; RL, on the other hand, has multiple 'tricks' and techniques in solving sequential decision-making problems, but is faced with the curse of dimensionality. Potential-Based Reward Shaping (PBRS) and Hierarchical RL (HRL) techniques shed some light on how to use heuristics to combat this curse, but they lack a systemic methodology to derive high-quality heuristics from domain knowledge.

To summarise, as representatives of proactive and reactive decision-making AI techniques, both Argumentation Theory and RL have limitations when used alone. As we have discussed in Chapter 1, we believe that integrating proactive and reactive decision-making techniques can result in more powerful decision-making al-

gorithms, because people use these two kinds of decision-making strategies jointly. In the remainder of this thesis, we will investigate the integration of these two techniques and evaluate the resulting algorithms' effectiveness in multiple application domains.

3 Argumentation Frameworks for Reinforcement Learning

In the previous chapters, we have discussed that although integrating heuristics into RL via Potential-Based Reward Shaping (PBRS, see Section 2.2.5) is an effective method to combat the curse of dimensionality faced by the RL algorithms, deriving heuristics from people's domain knowledge may not be an easy task. In this chapter, we propose a generic argumentation framework to tackle this problem. To be more specific, this framework is based on the Value-based Argumentation Frameworks (VAFs, see Section 2.1.3), and it is used to generate heuristics for multiple independent yet cooperative RL agents. Later in Chapters 4 and 6, we will incorporate heuristics generated by this framework into two different kinds of RL algorithms; the resulting algorithms are called *Argumentation Accelerated RL* (AARL) algorithms.

This chapter is organised as follows: firstly, in Section 3.1, we motivate our argumentation framework by illustrating the problems of deriving heuristics from conflicting domain knowledge; then, in Section 3.2, we present the framework and prove some of its properties. After that, we introduce the architecture of AARL in Section 3.3. We review related works in Section 3.4 and, finally, we conclude this chapter and discuss future works in Section 3.5

3.1 Motivation

RoboCup Soccer Keepaway and Takeaway games [SSK05, IE08] are widely used testbeds for evaluating RL algorithms. Because of the high complexity of these games, heuristics are often used to accelerate RL's convergence speed in these games [DGK11]. However, domain knowledge in these problems can be conflicting and, thus, proposing 'good' heuristics for these problems can be very challenging. In this section, we will concretely show some of these challenges in Section 3.1.1 so as to motivate our work. After that, we summarise why we choose to

use Argumentation to derive heuristics for RL from people's domain knowledge, in Section 3.1.2.

3.1.1 The RoboCup Soccer Games

RoboCup Soccer is an international project which aims at providing an experimental framework in which various technologies can be integrated and evaluated[1]. In order to facilitate RL research in this application domain, two simplified tasks have been developed: the *Keepaway* game [SSK05], and the *Takeaway* game [IE08]. The basic settings of these games are the same: $N+1$ ($N \in \mathbb{N}$, $N \geq 1$) *keepers* are competing with N *takers* on a fixed-size field. Keepers attempt to keep possession of the ball within their team for as long as possible, whereas takers attempt to win possession of the ball as quickly as possible. A game scenario involving two keepers and three takers is shown in Figure 3.1. At the start of each episode, the keeper in the top-left corner holds the ball, while all the other keepers are on the right. All takers are initially in the bottom-left corner. An episode ends when the ball goes off the field or any taker gets the ball, and a new episode starts immediately with all the players reset. We call a Keepaway (Takeaway) game consisting of $N+1$ keepers and N takers a N-Keepaway (N-Takeaway, respectively) game.

In Keepaway, only the keeper holding the ball learns; all the other keepers and all takers act in accordance with hand-coded strategies. In Takeaway, on the contrary, all takers learn independently while all keepers play in accordance with hand-coded strategies. So Takeaway games are cooperative multi-agent learning problems, whereas Keepaway games are single-agent learning problems taking place in multi-agent scenarios.

Most research on Keepaway/Takeaway games is performed in the RoboCup Soccer Simulation Platform[2], and agents are assumed to have a perfect knowledge of the environment: they can observe the accurate position of the ball and each agent. In the remainder of this chapter, in all Keepaway and Takeaway examples, we also adopt this assumption that all learning agents have perfect information of the environment. The simulation platform only provides *primitive actions* for each agent, e.g. change the velocity of each wheel[3] to some value. Directly using these primitive actions in RL results in poor performances and, as a result, *macro actions*

[1] See `http://www.robocup.org/` for more information.

[2] For more information about the Simulation platform, please refer to `http://wiki.robocup.org/wiki/Soccer_Simulation_League`.

[3] Each robot is actually a cubic shaped vehicle with two wheels.

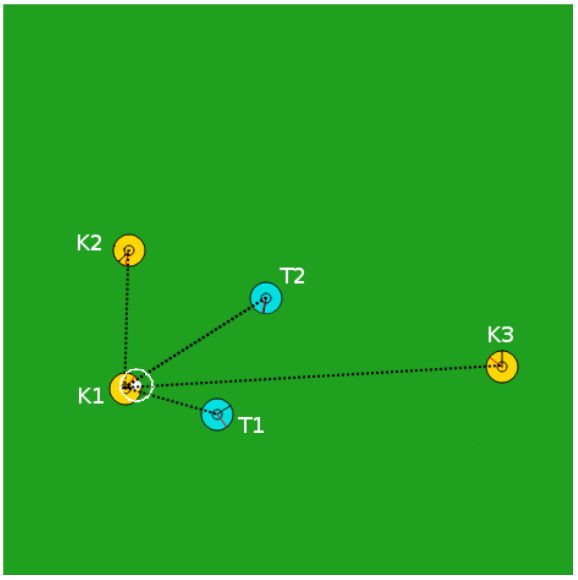


Figure 3.1: An example scenario in RoboCup Soccer Game. The ball is the white circle next to keeper K_1.

have been proposed first in Keepaway games by Stone et al. [SSK05], and then adjusted by Iscen and Erogul [IE08] in Takeaway games. To be specific, there are two macro actions for Keepaway:

HoldBall(): stay still while keeping the ball,
PassBall(p): kick the ball towards keeper K_p,

and two macro actions for Takeaway:

TackleBall(): move towards the ball/K_1 to tackle the ball,
MarkKeeper(p): go to mark keeper K_p, $p \neq 1$,

where $K_p, p \in \{1, \cdots, N+1\}$ represents the pth closest keeper to the ball (so K_1 is the keeper in possession of the ball). Takers are also indexed according to their distance to the ball: T_1 is the closest taker to the ball, while T_N is the farthest taker to the ball. When a taker marks a keeper, the taker stops between the ball holder and that keeper, in the hope of intercepting the ball if it is passed to that keeper. A taker is not allowed to mark the ball holder. Overall, in a N-Keepaway (N-Takeaway) game, there are $N+1$ macro actions available for each learning agent in each state.

SARSA(λ) is the most widely used algorithm for both Keepaway and Takeaway games (e.g. [SSK05, IE08, GTC12, GT14a]). Because these problems are with

continuous state space, to discretise the space, *tile coding function approximation* techniques [SB98] are used together with SARSA(λ). However, even after discretisation, the state space is still big[4] and, therefore, standard SARSA(λ) usually takes a very long time to find the optimal policy, especially in Takeaway games, which involve multiple learning agents (the time that SARSA(λ) needs to find the optimal policies are given later in Chapter 4). Since the game itself is easy to understand, people can easily propose some advise for agents, and the advise can feed into recommendations (heuristics) so as to accelerate RL. As a concrete example, consider the scenario shown in Figure 3.1. We focus on giving recommendations to takers (i.e. view this scenario as a snapshot of a Takeaway game). We first propose the following domain knowledge based on our observation of the game, where $q \in \{1, 2\}$:

1. T_q should tackle the ball if it is closest to the ball holder, because it can tackle the ball most quickly;

2. T_q should mark a keeper if the angle between T_q and this keeper, with vertex at the ball holder, is the smallest among all takers, because then T_q can block the pass to that keeper most quickly;

3. T_q should mark a keeper if T_q is closest to this keeper, because then T_q can approach this keeper most quickly.

Given this domain knowledge and the current state (Figure 3.1), we can give several recommendations to each taker. As for T_1, we can recommend it to tackle the ball according to the first item in the domain knowledge, and recommend it to mark K_3 according to the second. Similarly, we can recommend T_2 to mark K_2 because of items 2 and 3, and recommend T_2 to mark K_3 because of item 3. However, we can see that there are *conflicts* between these recommendations: for example, T_1 is recommended to perform both **TackleBall()** and **MarkKeeper(3)**, but it can perform at most one action at each moment. We call the conflicts between recommendations that recommend the same agent to perform different actions *internal conflicts*. In addition, we can see that both T_1 and T_2 are recommended to mark K_3, but asking multiple takers to mark one keeper is a waste of resources: one taker is able to do the job, so we think there exists a conflict between these two recommendations. This conflict is between recommendations of the same action

[4] The exact number of states depends on the number of agents involved. In a 2-Keepaway game, there are roughly 300 states. More details can be found in [SSK05].

to different agents, and we call conflicts of this kind *external conflicts*. From this example, we can see that to give useful heuristics, we not only need to give domain knowledge, but also need to resolve conflicts arising from this domain knowledge.

3.1.2 Why Incorporate Argumentation into RL

From the example above, we see the potential of using heuristics derived from domain knowledge to accelerate the learning speed of RL. However, in order to obtain high-quality heuristics from domain knowledge, we need to ensure that:

1. The heuristics should be the result of logical reasoning over the state (agents' observation of the current environment) and the domain knowledge. For example, the heuristics we give to T_2 is the result of performing logical reasoning over both the current state (T_2's distance to keeper K_3 is the closest among all takers) and the domain knowledge (the third item of the domain knowledge we provided in Section 3.1.1).

2. The heuristics we give to RL should be self-consistent, i.e. conflicts (either internal or external) should be resolved.

3. The process of deriving heuristics from domain knowledge should be easy to understand, so that people can supervise the deriving process and evaluate the resulting heuristics.

Using argumentation to represent domain knowledge and derive heuristics meets all these requirements since:

1. Argumentation theory has a well-founded logic foundation and is widely used in knowledge representation (see, e.g. [BH08]). Many kinds of logic reasoning can be performed in Argumentation Frameworks (AFs) by using different semantics [Dun95].

2. AFs can naturally model conflicts between recommendations with the attack relation, and several extensions of AFs have been developed to resolve conflicts more efficiently, e.g. Value-based Argumentation Frameworks (VAFs) (see Section 2.1.3).

3. Because of the structural and the conceptual simplicity of AFs, AFs have been recognised as an intuitive methodology for people to understand the reasoning process. This is also supported by recent psychological research,

which indicates that people reason in order to argue [MS11]. In addition, by using VAFs, besides arguments, people can also give reasons behind each argument (i.e. the values, see Section 1.2 and Section 2.1.3) and the ranking of these reasons; this further facilitates people to represent and justify their domain knowledge. Later, in Chapter 4, we perform an experiment involving 128 students to show that arguments can easily used to propose effective heuristics by people with virtually no background in argumentation.

For these reasons, we will use argumentation for deriving heuristics from domain knowledge for RL.

3.2 Argumentation Frameworks for Reinforcement Learning

We first explicitly define the problems we aim to tackle, i.e. the problems that our technique can be applied to (Section 3.2.1). Given the problem definition, we introduce the form of arguments we use (Section 3.2.2), and then define our argumentation frameworks and show that 'rationally acceptable' arguments for these frameworks correspond to 'good' heuristics for the agents (Section 3.2.3). At last, we discuss how to derive heuristics from the 'rationally acceptable' arguments (Section 3.2.4).

3.2.1 Problem Definition

We consider a MDP problem consisting of K ($K \in \mathbb{N}^*$) agents in total. Among them, L ($L \in \mathbb{N}^*, K \geq L$) agents are learning independently yet cooperatively to achieve a shared goal, and each agent has the same set of available actions; the other $K - L$ agents are acting according to *fixed strategies*. A fixed strategy is a function $\sigma : S \times A \to \mathbb{R}$ such that for any state $s \in S$ and action $a \in A$, $\sigma(s, a) \in [0, 1]$ indicates the probability of performing action a at state s, and it is a constant. We can see that function σ specifies the actions to be performed in each state, and their chances to be performed. Also, note that different non-learning agents can use different fixed strategies. We denote these L cooperative learning agents as $Agent_1, \ldots, Agent_L$, and denote the action set as $Act = \{a_1, \ldots, a_M\}$, where $M \in \mathbb{N}, M \geq 2$ is the number of available actions. Note that generally $Act \neq A$, because the available actions for the learning agents can be different from those for the non-learning agents.

To illustrate this problem definition, let us consider the N-Keepaway and N-Takeaway problems (see Section 3.1.1). In a N-Keepaway game, since there are $2N+1$ agents ($N+1$ keepers and N takers) in total, $K = 2N+1$. Among them, since only one agent is learning, namely the ball holder, so $L = 1$. Also, since there are $N+1$ macro actions for the learning keeper, $M = N+1$. In a N-Takeaway game, the total number of agents is the same as that of a N-Keepaway. So we have $K = 2N+1$. However, since all takers are learning independently and cooperatively, $L = N$. Also, because each taker has $N+1$ macro actions, $M = N+1$.

3.2.2 Argument for RL

It is usually easier for domain experts to propose 'action-based' domain knowledge for certain agents [Mar07], i.e. suggesting an agent to perform some action at some state. Thus, we use action-based arguments in the following form:

$$Ag\text{: } Agent_i \text{ performs } con(Ag) \text{ IF } pre(Ag)$$

where Ag is the name of this argument, $con(Ag)$ (the *conclusion* of Ag) is the action that Ag recommends, $pre(Ag)$ (the *premise* of Ag) describes under which conditions argument Ag is *applicable*, and $Agent_i$ indicates which agent this argument *belongs to*, where $i \in \{1, \cdots, L\}$. We say an argument Ag *supports* an action a_j, $j \in \{1, \cdots, M\}$, iff $con(Ag) = a_j$. Throughout this section, unless stated otherwise, i and j will range over learning agents and available actions, respectively, i.e. $i \in \{1, \cdots, L\}$ and $j \in \{1, \cdots, M\}$.

We denote $Agent_i$'s observation of the current state by Sta_i. We denote that an argument A's premise is true according to Sta_i by $Sta_i \models pre(A)$. We say an argument A is *applicable* with respect to $Agent_i$ iff A belongs to $Agent_i$ and $Sta_i \models pre(A)$. Let Arg_j^* be the set of arguments supporting action a_j, i.e. $A \in Arg_j^*$ iff $con(A) = a_j$, then

$$Arg^* = \cup_{j=1}^{M} Arg_j^*$$

is the set of all candidate arguments. We assume that each agent is aware of all arguments in Arg^*.

Example 1. (Candidate arguments in Keepaway game.) Earlier in Section 3.1.1, we have given some domain knowledge for takers. Here, similarly, we give some domain knowledge for the learning keeper as follows:

- The ball should be passed to an 'open' keeper, because by doing this, the risk of the ball being intercepted is lower;
- The ball should be passed to a keeper which is 'far' from all takers, because by doing this, the keepers' team is more likely to have possession of the ball for longer;
- The ball holder should hold the ball, by default, because by doing this, the ball holder can have possession of the ball for longer time.

We explicitly define 'open' and 'far' as follows: a keeper is 'open' if the angle between this keeper and any taker, with vertex at K_1, is greater than 15 degree; a keeper is 'far' if its distances to all takers are bigger than 10 units (the width and height of the field in Figure 3.1 are both 40 units). Since the domain knowledge above is action-based, we can naturally represent it in the form of arguments, as follows:

- **O**(p): K_1 performs **PassBall**(p) IF K_p is open
- **F**(p): K_1 performs **PassBall**(p) IF K_p is far
- **HD**: K_1 performs **HoldBall()** IF *True*

where p ranges over all keepers not holding the ball, i.e. for a N-Keepaway game, $2 \leq p \leq N+1$. Indeed, the ball holder cannot pass the ball to itself but can pass it to the any other keeper. So for a N-Keepaway game, there are $2N+1$ candidate arguments in total. We call the keepers that do not hold the ball the *free keepers*. Subsequently, in all examples (either Keepaway or Takeaway games) in this chapter, unless stated otherwise, p ranges over the free keepers.

Example 2. (Candidate arguments in Takeaway game.) Given the domain knowledge we present in Section 3.1.1, we propose the following arguments:

- T_q**TK**: T_q performs **TackleBall()** IF T_q is closest to the ball
- T_q**SA**(p): T_q performs **MarkKeeper**(p) IF the angle between T_q and K_p, with vertex at K_1, is smallest among all takers
- T_q**SD**(p): T_q performs **MarkKeeper**(p) IF T_q's distance to K_p is shortest among all takers

where $q \in \{1, \cdots, N\}$. In a N-Takeaway game, because there are N free keepers, each taker T_t has one argument T_t**TK**, N arguments of T_t**SA(**p**)**, and N arguments of T_t**SD(**p**)**. Thus we can see that each taker has $2N + 1$ candidate arguments, and the whole takers' team have $2N^2 + N$ candidate arguments. Subsequently, in all examples in this chapter, unless stated otherwise, q ranges over all takers in both N-Takeaway and N-Keepaway games.

3.2.3 Argumentation Frameworks

Arguments as given earlier may have conflicts with one another, and these conflicts can be either internal or external. For example, in the illustrative example we gave in Section 3.1.1, both T_1 and T_2 are recommended to mark K_3, according to different arguments; so, intuitively, these arguments have conflicts with each other (we will revisit this example below in Example 4). To systematically resolve these conflicts, we define argumentation frameworks as follows, to organise the arguments given earlier and represent conflicts between them.

Definition 2. Given $Sta = \langle Sta_1, \cdots, Sta_L \rangle$, where Sta_i is $Agent_i$'s observation of state s, then a *Sta-specific cooperative argumentation framework* is a tuple $SCAF = (Arg, Att)$ s.t.:

1. $Arg = \langle Arg_1, \cdots, Arg_L \rangle$ s.t. $Arg_i \subseteq Arg^*$ and $A \in Arg_i$ iff A is applicable with respect to $Agent_i$
2. $Att \subseteq \cup_{i=1}^{L} Arg_i \times \cup_{i=1}^{L} Arg_i$ s.t. $(A, B) \in Att$ iff for some $g, h \in \{1, \cdots, L\}$:
   a) $con(A) = con(B)$, $A \in Arg_g$, $B \in Arg_h$ and $g \neq h$, or
   b) $con(A) \neq con(B)$ and $A, B \in Arg_g$.

We refer to a Sta-specific cooperative argumentation framework as a SCAF, and we call $(\cup_{i=1}^{L} Arg_i, Att)$ the *AF derived from SCAF*.

Remark. *We follow two steps to build a SCAF: 1. select applicable arguments for all learning agents and 2. build attacks between these applicable arguments. In step 1, we check the premise of each argument that belongs to* $Agent_i$ *(*$i \in \{1, \cdots, L\}$*) to see whether this argument's premise is true according to* Sta_i*, so as to select applicable arguments for* $Agent_i$*. In step 2, we build attacks between any two applicable arguments if and only if: a) these two arguments are applicable with respect to different agents but support the same action (external conflict),*

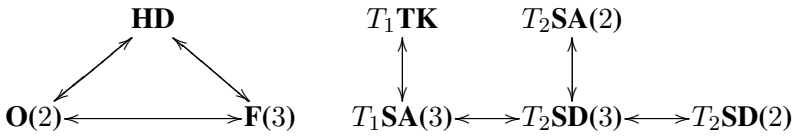


Figure 3.2: The derived AF of SCAFs for the Keepaway game (left) and the Takeaway game (right) in the scenario shown in Figure 3.1.

or b) these two arguments are applicable with respect to the same agent but support different actions (internal conflict). In particular, note that if there is only one learning agent (i.e. $L = 1$*), there can be no external conflicts and attacks are therefore only built according to rule b). Also note that given these rules of building attacks, if argument* A *attacks* B *in a SCAF, then* B *also attacks* A*, namely the attack relation is symmetric.*

Example 3. (Continuation of Example 1.) We build the SCAF for the scenario shown in Figure 3.1. Note that since we are considering the Keepaway game, only the ball holder is learning and we build the SCAF for this learning keeper. First we select applicable arguments by checking all candidate arguments' premises one by one. We first check the applicability of **O**(2) and **O**(3). Note that a keeper is open if the angle between this keeper and all takers, with vertex at K_1, is larger than 15 degree (see Example 1 earlier in Section 3.2.2). Given this criterion, K_2 is open and, thus, **O**(2) is applicable. However, K_3 is not open according to this criterion, so **O**(3) is not applicable. Then we check the applicabilities of arguments **F**(2) and **F**(3). Note that a keeper is far if its distances to all takers are larger than 10. The distance between K_2 and T_2 is not larger than 10; so **F**(2) is not applicable. But the distance between K_3 and any taker is larger than 10, so **F**(3) is applicable. As for **HD**, it is always applicable. So the applicable argument set in this scenario is $\{\mathbf{O}(2), \mathbf{F}(3), \mathbf{HD}\}$.

Then we build attacks between these three applicable arguments. By item b) in Definition 2, we can see that these three arguments mutually attack one another. The derived AF of this *SCAF* is illustrated in Figure 3.2(a).

Example 4. (Continuation of Example 2.) We build the *SCAF* for the Takeaway game in the scenario shown in Figure 3.1. We first select the arguments for T_1. Because T_1 is closest to the ball, T_1**TK** is applicable; its angle with K_2 is not the smallest, but its angle with K_3 is the smallest, so T_1**SA**(2) is not applicable

and T_1**SA**(3) is applicable. Because T_2 is closer to both K_2 and K_3, T_1**SD**(2) and T_1**SD**(3) are not applicable. So the applicable argument set for T_1 is $Arg_1 = \{T_1\textbf{TK}, T_1\textbf{SA}(3)\}$. Similarly, we can obtain $Arg_2 = \{T_2\textbf{SA}(2), T_2\textbf{SD}(2), T_2\textbf{SD}(3)\}$. Then we build attacks between these applicable arguments. We first build internal conflicts according to rule b) in Definition 2. As for T_1, since T_1**TK** and T_1**SA**(3) support different actions, these two arguments attack one another. As for T_2, because argument T_2**SA**(2) and T_2**SD**(2) support the same action, there is no attacks between these two arguments. But these two arguments both are in a mutual attack relation with T_2**SD**(3), which supports a different action. Then we build external conflicts according to a) in Definition 2. Since T_1**SA**(3) and T_2**SD**(3) support the same action to different takers, these two arguments attack one another. So $Att = \{(T_1\textbf{TK}, T_1\textbf{SA}(3)), (T_1\textbf{SA}(3), T_1\textbf{TK}), (T_2\textbf{SD}(2), T_2\textbf{SD}(3)), (T_2\textbf{SD}(3), T_2\textbf{SD}(2)), (T_2\textbf{SA}(2), T_2\textbf{SD}(3)), (T_2\textbf{SD}(3), T_2\textbf{SA}(2)), (T_1\textbf{SA}(3), T_2\textbf{SD}(3)), (T_2\textbf{SD}(3), T_1\textbf{SA}(3))\}$. The derived AF of this *SCAF* is illustrated in Figure 3.2(b). This AF can be viewed as an argumentation-based representation of the heuristics (recommendations) described in Section 3.1.1.

In the remainder of this chapter, we let Arg_{ij} be the set of arguments that are applicable with respect to $Agent_i$ and support action a_j:

$$Arg_{ij} = Arg_i \cap Arg_j^*$$

We prove that each Arg_{ij} is 'rationally acceptable', as follows:

Proposition 1. *Let (Arg, Att) be a SCAF and $AF = (\cup_{i=1}^{L} Arg_i, Att)$ be the AF derived from it. Then, Arg_{ij} is an admissible extension for AF.*

Proof. *By definition of Att, Arg_{ij} is conflict-free because arguments therein have neither external conflicts (they belong to the same agent) nor internal conflicts (they support the same action). Let $A \in Arg_{ij}$ and $B \in Arg \setminus Arg_{ij}$. If $(B, A) \in Att$, then $(A, B) \in Att$ necessarily. So Arg_{ij} can defend all its elements and, thus, is admissible.*

Remark. *Proposition 1 sanctions that, in a SCAF, all actions supported by applicable arguments are 'equally good' for an agent, since their arguments can defend themselves. There may be several such 'equally good' actions for an agent, and different agents may have the same 'equally good' actions: these situations are not desirable in a cooperative MAS of the kind we consider, where one agent can*

only performs one action in each time step and different agents are supposed to perform different actions. To address this problem, we extend the notion of SCAF with values and ranking thereof (as in VAFs, see Section 2.1.3):

Definition 3. Given $Sta = \langle Sta_1, \cdots, Sta_L \rangle$ as in Definition 2, a *value-based SCAF* is a tuple $VSCAF = (SCAF, V, val, Valpref_{Sta})$ s.t.:

1. *SCAF* is a Sta-specific cooperative argumentation framework;
2. *V* is a set (of *values*);
3. $val : Arg^* \rightarrow V$ is a function from Arg^* to *V*; and
4. $Valpref_{Sta}$ is a preorder over *V* under *Sta*.

We denote $val(A) = v$, for $A \in Arg^*$, as $A \mapsto v$, and say that A *promotes* v. Also, we denote 'v_1 if more preferred than v_2 in $Valpref_{Sta}$', for $v_1, v_2 \in V$, as $v_1 \geq_v v_2$. If $v_1 \geq_v v_2$ and $v_2 \geq_v v_1$, we denote $v_1 =_v v_2$. If $(\cup_{i=1}^{L} Arg_i, Att)$ is the AF derived from *SCAF*, then we refer to $(\cup_{i=1}^{L} Arg_i, Att, V, val, Valpref_{Sta}^{-})$ as the *VAF derived from VSCAF*, where $Valpref_{Sta}^{-}$ is a strict partial order $>_v$ such that $\forall v_1, v_2 \in V, v_1 >_v v_2$ if and only if $v_1 \geq_v v_2$ and $v_2 \not\geq_v v_1$. We refer to a value-based SCAF simply as a VSCAF.

Note that, as in standard VAFs, each argument can only promote one value while each value can be promoted by several arguments. $Valpref_{Sta}$ reflects the value preference in one state observation *Sta*. Also note that, unlike in standard VAFs where the value ranking is a strict partial order (i.e. for any two values v_1 and v_2, if $v_1 >_v v_2$ in the value ranking, $v_2 >_v v_1$ is not permitted; see Section 2.1.3), we allow multiple values to be equally preferred in a VSCAF, by using a preorder instead of a strict partial order. By allowing multiple values to be equally ranked, domain experts can represent their knowledge more greater flexibility (see Example 6 below). We assume that agents share the same value set and value preference in each state *Sta*, in line with our assumption that agents are cooperative. Given this value preference, a simplified AF can be derived from the VAF derived from a VSCAF, as in standard VAFs (see Section 2.1.3 for the simplification process). We will refer to

$$AF^{-} = (\cup_{i=1}^{L} Arg_i, Att^{-})$$

as the simplified AF derived from the VAF derived from $(SCAF, V, val, Valpref)$ (with $SCAF = (Arg, Att)$).

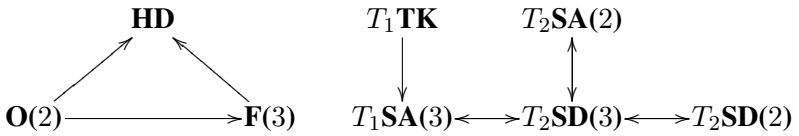


Figure 3.3: The simplified argumentation frameworks for the Keepaway game (left) and the Takeaway game (right) in the scenario shown in Figure 3.1. Note that these two simplified argumentation frameworks are derived from AFs in Figure 3.2(a) and 3.2(b), respectively, by deleting the attacks from arguments promoting lower ranked values to arguments promoting higher ranked values.

Example 5. (Continuation of Example 3.) In order to define values, let us consider again the domain knowledge in Example 1: this not only recommends actions under certain conditions, but also gives the reason behind these recommendations. For example, besides recommending the ball holder to pass the ball to an 'open' keeper, the first piece of domain knowledge also gives the reason for this recommendation: 'the risk of the ball being intercepted is lower'. These reasons for each recommendation can be represented by three values: *LESS_INT*, *TEAM_LONG* and *HOLD_LONG*, standing for 'lower the risk of interception', 'help the keeper's team to have possession of the ball for longer' and 'help the ball holder to have possession of the ball for longer', respectively. As a result, we have $V = \{\mathit{LESS_INT}, \mathit{TEAM_LONG}, \mathit{HOLD_LONG}\}$. Then the promotion relation, *val*, can be naturally obtained from that domain knowledge: $\mathbf{O}(p) \mapsto \mathit{LESS_INT}, \mathbf{F}(p) \mapsto \mathit{TEAM_LONG}, \mathbf{HD} \mapsto \mathit{HOLD_LONG}$. Then, by giving a partial ordering of these values (*Valpref*, we omit the subscript here because the value preference is specifically for the scenario in Figure 3.1), we can derive the simplified argumentation framework AF^{-}.

For example, we can give a value ranking as follows: $\mathit{Valpref} = \{\mathit{LESS_INT} \geq_v \mathit{TEAM_LONG} \geq_v \mathit{HOLD_LONG}\}$. We can obtain that $\mathit{Valpref}^{-} = \mathit{LESS_INT} >_v \mathit{TEAM_LONG} >_v \mathit{HOLD_LONG}$. Given $\mathit{Valpref}^{-}$, we eliminate the attack from **HD** to **F**(3), because the value *HOLD_LONG* promoted by **HD** is (strictly) lower ranked than the value *TEAM_LONG* promoted by **F**(3). We can eliminate all 'unsuccessful' attacks in a similar way and obtain AF^{-} as illustrated in Figure 3.3(a).

Example 6. (Continuation of Example 4.) In order to propose values, we consider the reasons behind each recommendation, which has been included in the domain knowledge listed in Example 2. We can see that the reasons (values) of

those pieces of domain knowledge are 'tackle the ball more quickly', 'mark a keeper more quickly' and 'approach a keeper more quickly', respectively, and we denote these three values as *QUICK_TAC*, *QUICK_MARK*and *QUICK_CLOSE*, respectively. Given these values and the arguments for takers proposed in Example 2, we can see that arguments T_q**TK** promote value *QUICK_TAC*, arguments T_q**SA**(p) promote value *QUICK_MARK* and arguments T_q**SD**(p) promote *QUICK_CLOSE*. Therefore, we have $val = \{T_q\textbf{TK} \mapsto \textit{QUICK_TAC}, T_q\textbf{SA}(p) \mapsto \textit{QUICK_MARK}, T_q\textbf{SD}(p) \mapsto \textit{QUICK_CLOSE}\}$.

We give an example ranking of these values: $\textit{Valpref} = \{\textit{QUICK_TAC} \geq_v \textit{QUICK_MARK} =_v \textit{QUICK_CLOSE}\}$, and then we can obtain that $\textit{Valpref}^{-} = \textit{QUICK_TAC} >_v \textit{QUICK_MARK}$. Given this value ranking, we can eliminate the unsuccessful attacks, and the resulting simplified AF^{-} is illustrated in Figure 3.3(b).

Until now, we have defined three kinds of argumentation frameworks: *SCAF*, *VSCAF* and AF^{-} derived from *VSCAF*. Domain knowledge contained in these frameworks increases progressively: *SCAF* selects applicable arguments from all candidate arguments (Arg^*), and builds the attack relation between the applicable arguments; based on *SCAF*, *VSCAF* includes value-related domain knowledge (the value set V, the promotion relation *val* and the partial order of values *Valpref*); and AF^{-} can be viewed as a concise representation of *VSCAF*, such that all important information (which arguments are applicable, their relation and their relative strength) is included in AF^{-}. In other words, *SCAF* and *VSCAF* can be viewed as 'intermediate products' in our knowledge representation process, and the final product is AF^{-}, which is of simple form (compared with *SCAF* and *VSCAF*) yet contains all domain knowledge we need to extract heuristics. As a result, we use AF^{-} to derive heuristics for RL algorithms.

In order to derive heuristics from AF^{-}, we simply choose the 'winning arguments' in AF^{-}, and derive the 'winning actions' from the 'winning arguments'. This derivation will be discussed below in Section 3.2.4. Here we focus on the selection of the 'winning arguments' themselves and prove that heuristics derived from these winning arguments have some desirable properties and are therefore 'high-quality'. To be more specific, we consider two widely used semantics to select the winning arguments: the preferred and the grounded semantics, as the representatives of the credulous and sceptical approach, respectively (see Section 2.1.2).

Lemma 1. *If G is a non-empty grounded extension for AF^{-}, then $\exists i, j$ s.t. $G \cap$*

$Arg_{ij} \neq \emptyset$.

Proof. *Suppose $A \in G$. According to the definition of arguments (see Section 3.2.2), A must belong to some agent $Agent_g, g \in \{1, \cdots, L\}$ and supports some action $a_h, h \in \{1, \cdots, M\}$. As a result, $A \in Arg_{gh}$ and $G \cap Arg_{gh} \neq \emptyset$.*

Lemma 2. *If P is a preferred extension for AF^{-}, then $\exists i, j$ s.t. $P \cap Arg_{ij} \neq \emptyset$.*

Proof. *We just need to prove that all preferred extensions of AF^{-} are non-empty. In particular, we show that the arguments promoting* highest achievable values *must be in some preferred extension of AF^{-}. Suppose $AF^{-} = (Arg, Att^{-})$, and its set of* achievable values *V_{ach} is defined as follows: $V_{ach} = \{v | v \in V, \exists A \in Arg\ s.t. A \mapsto v\}$. $v^* \in V_{ach}$ is a highest achievable value such that $\forall v \in V_{ach}, v \not\succ_v v^*$. By definition, there must be some argument $A \in Arg$ s.t. $A \mapsto v^*$. Note that Att^{-} is derived from Att (i.e. the attack relation in $SCAF$) by using the simplification rules (see Section 2.1.3), and all attacks in Att are symmetric. As a result, $\forall B \in Arg$, if $(B, A) \in Att^{-}$, $(A, B) \in Att^{-}$, i.e. $\{A\}$ can counter-attack all arguments that attack A. Hence, $\{A\}$ is an admissible extension for AF^{-}. Because a preferred extension is maximally (with respect to $\subseteq$) admissible (see Section 2.1.2), there must exist a non-empty preferred extension in AF^{-}.*

Remark. *Lemma 1 indicates that if the grounded semantics is used to give recommendations, when the grounded extension is non-empty, then at least one agent will receive a recommendation; otherwise, there is no 'winning argument' and, thus, no action is recommended to any agent. On the other hand, Lemma 2 suggests that, when using the preferred semantics to give recommendations, at least one agent will receive a recommendation and, in particular, the 'best available action' (the argument that promotes the highest achievable value) is guaranteed to be recommended to some agent.*

Theorem 1. *If E is a non-empty grounded extension or a preferred extension for AF^{-}, then $\forall i$, if $\exists p, q \in \{1, \cdots, M\}$ s.t. $Arg_{ip} \cap E \neq \emptyset$ and $Arg_{iq} \cap E \neq \emptyset$, then $p = q$.*

Proof. *Necessarily, $\exists A, B \in E$ s.t. $A \in Arg_{ip}, B \in Arg_{iq}$ (Lemma 1 and Lemma 2 for grounded and preferred extensions, respectively). If $A = B$, then the theorem is obviously true. If $A \neq B$, by contradiction, assume $p \neq q$. Then, by definition of Att, $(A, B) \in Att$ and $(B, A) \in Att$. According to the simplification rules in VAFs,*

(A, B) *or* (B, A) *or both are in* Att^-.[5] *Hence,* E *is not conflict-free and so not grounded/preferred: contradiction.*

Remark. *By Theorem 1, we show that the non-empty grounded extension or a preferred extension recommends an agent at most one action. This result is significant because each agent can only perform one action at each time. Note that, although each preferred extension recommends only one action to each agent, different preferred extensions may recommend different actions to the same agent, as shown in Example 7 below.*

Example 7. (Continuation of Example 6.) Consider the simplified argumentation framework AF^- as illustrated in Figure 3.3(b). The grounded extension is $\{T_1\textbf{TK}\}$, so only T_1 gets a recommendation. However, there are two preferred extensions for AF^-: $P_1 = \{T_1\textbf{TK}, T_2\textbf{SA(2)}, T_2\textbf{SD(2)}\}$ and $P_2 = \{T_1\textbf{TK}, T_2\textbf{SD(3)}\}$, and they recommend different actions to T_2. As an agent can only perform one action at a time, we can only select one preferred extension to give recommendations. We will describe how to break ties between multiple preferred extensions below in Section 3.2.4.

Theorem 2. *If* E *is the non-empty grounded extension or a preferred extension for* AF^-, $\forall j$, *if* $\exists p, q \in \{1, \cdots, N\}$ *s.t.* $Arg_{pj} \cap E \neq \emptyset$ *and* $Arg_{qj} \cap E \neq \emptyset$, *then* $p = q$.

Proof. $\exists A, B$ *as in the proof of Theorem 1. Again, if* $A = B$, *the proof is trivial. If* $A \neq B$ *but* $p \neq q$, (A, B) *or* (B, A) *or both are in* Att^- *which contradicts that* E *is grounded/preferred.*

Remark. *By Theorem 2, we prove that the non-empty grounded extension or a preferred extension recommends an action to at most one agent. This result is especially significant in cooperative multi-agent learning problems (e.g. Takeaway games), where it is undesirable for multiple agents to perform the same action. However, the same action could be recommended to different agents by different preferred extensions, as shown in Example 8 below.*

Example 8. (Continuation of Example 6.) Consider again the scenario in Figure 3.1. Let us focus on giving recommendations to takers. We use a new ranking of values $Valpref = \{QUICK_CLOSE \geq_v QUICK_TAC \geq_v QUICK_MARK\}$, and

[5] According to the simplification rules of VAF (see Section 2.1.3), none of the attacks (if any) between two arguments A and B can be eliminated if $val(A) =_v val(B)$.

obtain that *Valpref*$^{-}$ $= \{$*QUICK_CLOSE* $>_v$ *QUICK_TAC* $>_v$ *QUICK_MARK*$\}$. Given this new value ranking, we generate a new simplified argumentation framework. The new *AF*$^{-}$ is illustrated in Figure 3.4. There are two preferred extensions for *AF*$^{-}$: $P_1 = \{T_1\textbf{TK}, T_2\textbf{SD}(3)\}$ and $P_2 = \{T_1\textbf{SA}(3), T_2\textbf{SA}(2), T_2\textbf{SD}(2)\}$. We can see that **MarkKeeper(3)** is recommended to T_2 by P_1, but recommended to T_1 by P_2. Also, P_1 and P_2 recommend the same agent different actions. Similarly to Example 7, this example also suggests that only one preferred extension can be used to give recommendations at each state.

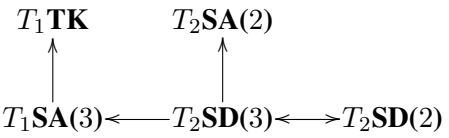


Figure 3.4: The simplified argumentation framework for takers in the scenario shown in Figure 3.1, given the new value ranking *QUICK_CLOSE* $>_v$ *QUICK_TAC* $>_v$ *QUICK_MARK*.

3.2.4 From Extensions to Heuristics

Above, we have introduced *SCAF*, *VSCAF* and *AF*$^{-}$ and proved some properties of the preferred and grounded extensions of *AF*$^{-}$. Recall that our purpose of computing these extensions is to obtain the heuristics, namely the recommended action for each agent. Here we discuss how to obtain heuristics from the extensions.

When the grounded extension is used, there is one and only one grounded extension for *AF*$^{-}$ (see Section 2.1.2). If the grounded extension is non-empty, each agent $Agent_i$ simply needs to find whether there exists any argument in the grounded extension belonging to $Agent_i$ (for definition of 'belong', see Section 3.2.2): if there is an argument A belonging to $Agent_i$, since Theorem 1 proves that each agent can have at most one recommended action, we can just recommend the action supported by A, i.e. $con(A)$ (see Section 3.2.2), to $Agent_i$. The above method also applies to cases when preferred extensions are used and there exists only one preferred extension for *AF*$^{-}$.

When preferred extensions are used and there are more than one preferred extensions, we need to break ties between them and select only one preferred extension to give recommendations: as illustrated above in Examples 7 and 8, although the recommended actions given by the same preferred extension are 'compatible'

with each other, i.e. the same action is not recommended to different agents and the same agent does not receive different recommended actions, actions recommended by different preferred extensions can be 'conflicting'. Recall that when there are multiple preferred extensions for AF^{-}, these preferred extensions are 'equally good' (Section 2.1.2). Based on this understanding, we randomly select a preferred extension and let all cooperative agents use this preferred extension to obtain its own recommended actions. To this end, we select an agent, called the *captain agent*, to perform the random selection; after it selects a preferred extension, it tells other agents which arguments are contained in the selected preferred extension.

Note that when there are multiple preferred extensions, all agents must all use the same preferred extension; otherwise, more than one agent may be recommended the same action: for example, consider Example 8 in Section 3.2.3; if taker T_2 uses preferred extension P_1 to generate heuristics while T_1 uses extension P_2, both these takers will go to perform **MarkKeeper(3)**, which is undesirable. So in these cases, communication between agents are essential. However, some 'tricks' can be used to reduce the communication burden: for example, if the captain agent can afford computing the recommended action for each agent, then it can directly let other agents know their recommended actions, without communicating the whole extension; or when the captain agent cannot afford this, agents can use a universal predefined system to index all candidate arguments, so that the captain agent only needs to let other agents know the indices of the arguments in the selected preferred extension.

The method described above is summarised as a function **getRecActFromExt**, whose pseudo code is given in Algorithm 6. This function has two arguments: a set E containing all extensions of required type (preferred or grounded), and the agent index i. The purpose of this function is to obtain the recommended action for $Agent_i$. If the agent does not have any recommended action, this function returns **null**.

Now we walk through this function. If there is only one extension in E (lines 2 to 8), the function checks every argument in this extension to see whether any argument belongs to $Agent_i$: if there is one or more arguments that belongs to $Agent_i$, the function returns the action supported by these arguments (line 5, note that these arguments are guaranteed to support the same action, proved in Theorem 1); otherwise, it returns **null**, indicating that there is no recommended action for this agent (line 8). If there are multiple extensions in E (lines 10 to 22), when

$Agent_i$ is not the captain agent, this function simply waits for the captain to send the recommended action for $Agent_i$, and then returns this action (line 11); otherwise, the function builds a table to store all agents' recommended actions (line 13), and updates this table by finding each agent's recommended action (line 16). Finally, the function informs all other agents their recommended actions (line 18 to 20), and return $Agent_i$'s recommended action (line 21).

Algorithm 6 Function for obtaining recommended action from extensions.

```
1:  function getRecActFromExt(ExtensionSet E, AgentIndex i)
2:  if there is only one extension S in E then
3:     for argument arg in S do
4:        if arg belongs to Agent_i then
5:           return con(arg)
6:        end if
7:     end for
8:     return null
9:  else
10:    if Agent_i is not the captain then
11:       receive the recommended action a from the captain agent, and return a
12:    else
13:       initialise table actTable, whose keys are all agents' indices, entries are all
          null
14:       randomly selects an extension S from E,
15:       for argument arg in S do
16:          find the owner Agent_j of arg, and have actTable(j) := con(arg)
17:       end for
18:       for all agents indices j ≠ i do
19:          inform Agent_j its recommended action actTable(j)
20:       end for
21:       return actTable(i)
22:    end if
23: end if
```

3.3 Argumentation Accelerated RL (AARL)

Given the argumentation framework we presented above, now we discuss how to integrate this framework into RL algorithms. As we have mentioned in the beginning of this chapter, the resulting algorithms are called *Argumentation Accelerated Reinforcement Learning* (AARL). The architecture of AARL is given in Figure 3.5.

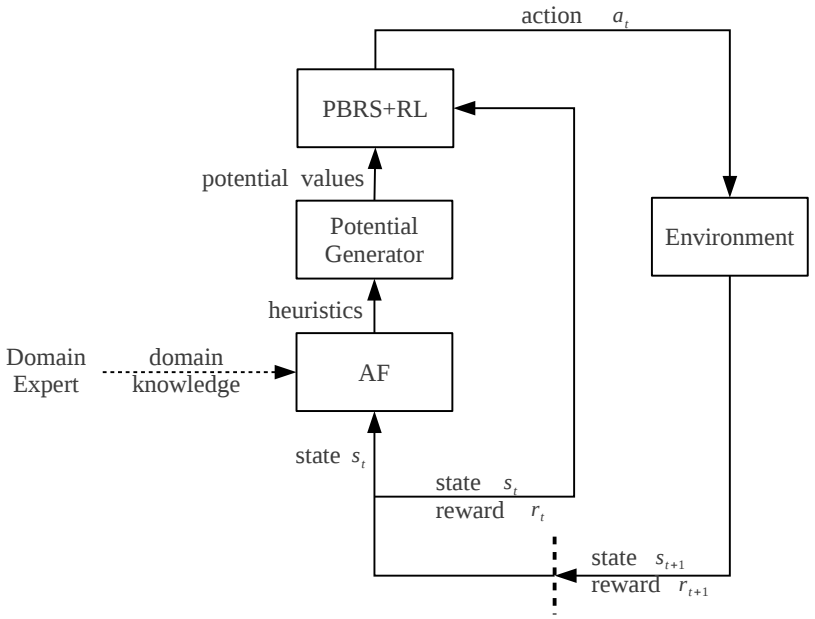


Figure 3.5: The architecture of AARL.

We now discuss the architecture as well as the workflow of AARL. Before the learning starts, domain experts first provide some domain knowledge based on their understanding of the problem, and send this knowledge to module *AF*. Note that this knowledge not only includes each agent's candidate arguments, but also the value set *V*, the value promotion relation *val*, the value ranking $Valpref_{Sta}$ in each state, which kind of extensions (preferred or grounded) should be used to derive heuristics, and the potential value for each state-action pair: $\Phi(s, a)$ (see Section 2.2.5). Note that because there can be infinitely many states or actions, it is unrealistic to ask the domain experts to give potential values for each state-action pair. As a result, we only ask the domain expert to provide a non-negative real number c, and this number will serve as the potential value for every recommended action in any state. Details of how to assign the potential values for each state-action pairs will be presented very shortly within this subsection.

During the learning process (i.e. the interaction between AARL and the environment), this knowledge cannot change and remains the same. In other words, this knowledge delivery from domain experts to module AF is upfront; thus, in Figure 3.5, we use a dashed arrow to connect these two components. Note that all other arrows are continuous, meaning that they are activated during each learning

step (see Section 1.3 for the definition of learning step and Section 2.2.2 for an illustration of it). Because all domain knowledge is provided a priori, if the number of arguments is not huge, the combinations of all possible argumentation frameworks can be easily obtained, and their (either preferred or grounded) extensions can also be computed before the learning starts. Because computing the extensions can be computationally expensive (especially for the preferred semantics, see Section 2.1.4), by doing this, we can avoid the computational overhead introduced by the argumentation frameworks during the learning, and therefore apply AARL to real-time applications, e.g. the RoboCup Keepaway and Takeaway games.

AARL amounts to the combination of three modules in Figure 3.5: the *AF* module, the *Potential Generator* module and the *PBRS+RL* module. In the beginning of each learning step, a learning agent first receives the new state s_t and reward r_t. State s_t is sent to module AF, so as to let the agent obtain the applicable arguments (discussed in detail below). Based on all agents' applicable arguments, module AF then builds *SCAF* (Definition 2), *VSCAF* (Definition 3), derives AF^{-} and obtains heuristics by computing specific type of extensions of AF^{-} (discussed above in Section 3.2.4). These derived heuristics, i.e. recommendations of actions, are then delivered to module *Potential Generator*.

Before we look into module *Potential Generator*, we first discuss some issues about communicating of the applicable arguments. Note that the argumentation frameworks we build (*SCAF*, *VSCAF* and AF^{-}) include all agents' applicable arguments. Because each agent i selects its applicable arguments based on its own observation of the environment Sta_i, to build these argumentation frameworks, an agent may need to communicate with other agents so as to know their applicable arguments.[6] Because all agents know all candidate argument a priori (this is an assumption we made about our argumentation frameworks, see Section 3.2.2), agents can use a universal system to index all candidate arguments, and when an agent needs to let others know its applicable arguments, it only needs to inform the others the indices of its applicable arguments. After knowing each agent's applicable arguments, any agent can build the *SCAF*, *VSCAF* and AF^{-} because all agents use the same rules to build attacks between applicable arguments and they share the same value set *V*, promotion relation *val* and value ranking $Valpref_{Sta}$ in each state. If preferred extensions are used, only the captain agent needs to build the *SCAF*,

[6] If all agents have perfect information of the environment and they all know how other agents select their applicable arguments, communication is not needed, because each agent can obtain all other agents applicable arguments.

VSCAF and *AF*$^-$, and it can inform the other agents their recommended actions (see Section 3.2.4); in this case, all non-captain agents only need to communicate with the captain.

The task of module Potential Generator is to propose potential values (see Section 2.2.5) for actions in state s_t, based on the input heuristics. The basic idea of this module is to give c to the recommended actions, and give zero potential values to the un-recommended actions. The reason that we give zero, not negative potential values, to those un-recommended actions is that the domain knowledge used in *SCAF*, *VSCAF* and *AF*$^-$ is only about which actions are 'good', not about which are 'bad' or 'unknown'. So we use a 'conservative' way to give potential values to those un-recommended actions: giving them zero potential values means that we are not sure about their goodness and thus we give them neither encouragement (positive potential values) nor discouragement (negative potential values).

The potential values generated by module Potential Generator will be fitted into some PBRS-augmented RL algorithms, for example the LA-SARSA(λ) algorithm we introduced in Section 2.2.5, and the *PBRS-MAXQ-0* algorithm, which will be introduced later in Chapter 5. Note that the potential values can directly affect the action-selection process of the PBRS-augmented RL algorithms (in LA-SARSA(λ), this has been illustrated in Section 2.2.5; in PBRS-MAXQ-0, this affection will be illustrated later in Chapter 5), so as to improve the learning speed of RL algorithms.

3.4 Related Work

There has been some research on improving the performance of RL by using high-level domain knowledge. Marthi [Mar07] proposed *abstract MDP* to find an approximation of the true shaping functions by solving a simpler abstract problem under the instructions of prior knowledge. Grzes and Kudenko [GK08] used the high-level STRIPS [FN72] operator knowledge in reward shaping to search for an optimal policy and showed that the STRIPS-based reward shaping converges faster than the abstract MDP approach. However, these approaches have very high and restrictive requirements on the domain knowledge being used: for example, in the abstract MDP approach, people are required to identify which states are 'similar', so as to merge these states into one state and therefore propose the abstract MDP; in Grzes and Kudenko's approach [GK08], people have to provide STRIPS-style domain knowledge, which cannot contain conflicts. We can see that

our argumentation framework based approach is able to handle more flexible domain knowledge: for example, the domain knowledge we provide in this chapter for the Keepaway and Takeaway games does not meet the requirement of their approaches, but can be used in our approach. In addition, their approaches can only be applied to single-agent learning problems, while ours can be used in multi-agent problems also. To summarise, our approach is more generic and flexible than existing techniques for proposing heuristics for RL.

Our work is closely related to work in argumentation-based decision making, because, in each state, our argumentation framework needs to decide which actions should be recommended to which agents. Amgoud [Amg09] proposed a two-phase argumentation-based model for decision making: in the first *inference* step, the model uses a Dung style system in which arguments in favour/against each option (action) are built, then evaluated by using certain semantics; in the second *comparison* phase, pairs of alternative options are compared using a given criterion, which is generally based on the winning arguments computed in the first phase. Note that, in the first phase, two kinds of arguments are used: *practical arguments*, whose conclusions are actions, and *epistemic arguments*, whose conclusions are premises of practical arguments. By introducing the epistemic arguments, the selection of the applicable arguments can also be modelled in an argumentative way. This distinction between practical and epistemic arguments is also present in other approaches to argumentation-based decision making, e.g. [AP09, BCP06]. Note that SCAF and VSCAF only allow practical arguments and the applicability of each argument is decided by comparing its premise with the current state, not by using epistemic arguments. Compared with Amgoud's approach, the argumentation framework we proposed has two immediate advantages: **(a)** by only allowing practical arguments, we reduce the overall number of arguments involved in each learning step, and thus reduce the domain expert's burden to propose arguments; this property is especially useful when the domain experts do not have much expertise in argumentation; and **(b)** the computational overhead of selecting the applicable arguments in our approach is lower than that in Amgoud's approach: in our approach, the programme just needs to go through the premises of each argument, but in Amgound's approach, the programme needs to compute the winning arguments in an argumentation framework, which involves both practical and epistemic arguments, and this computation is generally very expensive (see Section 2.1.4). However, the epistemic arguments allow for more powerful knowledge representation and justification capability. We leave extending our ar-

gumentation frameworks with epistemic arguments as a future work, which will be discussed in greater details later in Chapter 7.

3.5 Conclusion

In this chapter, we first presented the challenges of deriving high-quality heuristics from conflicting domain knowledge, and then proposed a VAF-based argumentation framework to tackle this problem. We proved that the heuristics generated by this framework are suitable for cooperative RL algorithms, because each agent receives at most one recommended action (this property is desirable because each we focused on problems where agent can perform only one action at each time slot) and each action is recommended to at most one agent (this property is desirable because we focused on cooperative RL problems where multiple agents performing the same action is a waste of resources). In addition, we proposed *Argumentation Accelerated RL* (AARL) as an incorporation of our VAF-based argumentation frameworks with RL algorithms. In particular, we outlined the architecture of AARL and described the functionality of each of its modules.

In the next chapter, we will instantiate the AARL framework on different RL algorithms so as to empirically test the effectiveness of AARL. We select SARSA(λ) and MAXQ-0 as the representatives of the flat and hierarchical RL algorithms, respectively, and implement AARL on these two algorithms. The resulting algorithms — SARSA(λ)- and MAXQ-based AARL — and their performances on some application domains will be presented in Chapter 4 and Chapter 6, respectively. In addition, in Chapter 5, we will give the PBRS-augmented MAXQ-0 algorithm: *PBRS-MAXQ-0*, which is essential for the construction of MAXQ-based AARL.

4 SARSA(λ)-based AARL

In Chapter 3, we have proposed a generic argumentation framework for deriving high-quality heuristics from conflicting domain knowledge. In this chapter, we integrate the heuristics generated by the aforementioned argumentation framework into SARSA(λ) (see Section 2.2.4), and propose the resulting algorithm: *SARSA(λ)-based Argumentation Accelerated RL* (SARSA(λ)-based AARL). We choose SARSA(λ) as the RL algorithm to implement AARL because it is a simple and widely used RL algorithm [SB98]; also, it has been integrated with mature PBRS techniques, e.g. *look-ahead advice* (LA, see Section 2.2.5), which have been proved sound and empirically effective.

This chapter is organised as follows: in Section 4.1, we present SARSA(λ)-based AARL; then we empirically test its effectiveness in the RoboCup Keepaway and Takeaway games in Section 4.2, and in a Wumpus World game in Section 4.3. Related works are reviewed in Section 4.4, and we conclude this chapter in Section 4.5.

4.1 SARSA(λ)-based AARL

To propose the SARSA(λ)-based AARL algorithm, let us first revisit the architecture of AARL we illustrated in Figure 3.5 in Chapter 3. We discussed that AARL amounts to the combination of three modules: AF, Potential Generator and PBRS+RL. We choose LA-SARSA(λ) to be the algorithm used in the PBRS+RL module, and this algorithm has been presented in Section 2.2.5.

For the other two modules, we can see that the output of the AF module, i.e. the heuristics, are only used in the Potential Generator module; as a result, for efficiency purposes, we implement one single function with the combined functionality of both these two modules. We name this function **getPotential**(s), and its output is a table containing all actions' potential values in state s. The pseudo code of this function is presented in Algorithm 7.

Function **getPotential** has two arguments: the current state s and the learn-

Algorithm 7 The combined AF module and Heuristics Generator module of SARSA(λ)-based AARL for a learning agent $Agent_i$.

1: function **getPotential**(State s, AgentIndex i)
2: Obtain candidate argument set *Arg**, value set *V*, value promotion relation *val*, value ranking *Valpref*, the argumentation extension type $Type$, and the potential value given to recommended actions: $c \in \mathbb{R}, c > 0$
3: Obtain all agents' applicable argument set
4: Build *SCAF*, *VSCAF*, and derive *AF*$^-$
5: $E :=$ **getExtensions**$(AF^-, Type)$
6: $a_{rec} :=$ **getRecActFromExt**(E, i)
7: Build a $Table$, whose keys are actions, entries are all 0
8: **if** a_{rec} is not **null then**
9: $\quad Table(a_{rec}) := c$
10: **end if**
11: **return** $Table$

ing agent index i. In the beginning (line 2), **getPotential** first obtains all domain knowledge provided by the domain expert. As we have discussed in Section 3.3, this knowledge is 'upfront', i.e. provided before the learning starts and remaining the same throughout the learning. Given this knowledge, each agent can obtain the applicable arguments (line 3, note that this may need communication with other agents, see Section 3.3), and then build *SCAF*, *VSCAF* and derive *AF*$^-$ (line 4). Given *AF*$^-$, the agent can compute the $Type$ extensions of *AF*$^-$ and store these extensions in set E (line 5). Note that $Type$ can be preferred or grounded, as discussed in Chapter 3. Given all extensions, the agent can choose the recommended action for itself, by invoking function **getRecActFromExt** (line 6), as defined in Algorithm 6 in Chapter 3. Note that if this agent does not have any recommended actions, function **getRecActFromExt** returns **null**. After obtaining the recommended action a_{rec}, the agent creates a table, in which the entries corresponding to the recommended actions are c, while the other actions' corresponding entries are 0 (lines 7 to 9, for why these potential values are given to each action, see Section 3.2.4). This table is the output of the function (line 11).

Function **getPotential** is integrated into LA-SARSA(λ) to give the SARSA(λ)-based AARL (in Algorithm 8). Here we highlight some significant differences between this algorithm and LA-SARSA(λ) (see Section 2.2.5):

- Before entering into the first learning step (see Section 1.3 and 2.2.2 for 'learning step'), we compute the potential values in the initial state and store the results in table $CurTable$ (line 5 in Algorithm 8). Note that the current

Algorithm 8 The SARSA(λ)-based AARL for $Agent_i$.

1: Initialise $Q(s, a)$ arbitrarily for all states s and actions a
2: **while** the experiment does not terminate **do**
3: Initialise $e(s, a) := 0$ for all s and a
4: Initialise current state s_t
5: $CurTable :=$ **getPotential**(s_t, i)
6: Choose action a_t from s_t using the biased ϵ-greedy policy
7: **while** s_t is not a terminal state **do**
8: Execute action a_t, observe reward r_t and new state s_{t+1}
9: $NextTable :=$ **getPotential**(s_{t+1}, i)
10: Choose a_{t+1} from s_{t+1} using the biased ϵ-greedy policy
11: $\delta := r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) + \gamma NextTable(s_{t+1}, a_{t+1}) - CurTable(s_t, a_t)$
12: $e(s_t, a_t) := 1$
13: **for** all s, a **do**
14: $Q(s, a) := Q(s, a) + \alpha\delta e(s, a)$
15: $e(s, a) := \gamma\lambda e(s, a)$
16: **end for**
17: $CurTable := NextTable$
18: $s_t := s_{t+1}$; $a_t := a_{t+1}$
19: **end while**
20: **end while**

state's potential values are computed before selecting the action to be performed in the current state (line 6), because in LA-SARSA(λ), actions are selected by using the biased ϵ-greedy policy (see Equation (2.8) in Section 2.2.5), which needs the potential values of all actions.

- After the next state s_{t+1} is observed, this algorithm computes the potential values for all actions in s_{t+1} and store the results in table $NextTable$ (line 9). Also, this potential value computation is performed earlier than selecting the action to be performed in s_{t+1}, which also requires potential values of all actions in s_{t+1}.

- In Q-value updating (line 11), according to Equation (2.8) in Section 2.2.5, potential values of both the current state-action pair (s_t, a_t) and the next state action pair (s_{t+1}, a_{t+1}) are needed. We obtain these two potential values by visiting their corresponding table entries.

- Towards the end of each learning step (line 17), the algorithm updates table $CurTable$ by replacing its entries with entries in $NextTable$. By doing this,

in the next learning step, potential values for s_t do not need to be computed again.

4.2 Experiments in RoboCup Games

We apply SARSA(λ)-based AARL (Algorithm 8) and SARSA(λ) (Algorithm 1) for each learning agent in the RoboCup Soccer Keepaway and Takeaway games to test the effectiveness of AARL. Since Keepaway is a single-agent learning problem and Takeaway is a multi-agent cooperative learning problem (see Section 3.1.1), by using these games as testbeds, we can comprehensively test the performance of SARSA(λ)-based AARL.

Our experiments can be divided into three phases, and the complexities of experiments in these phases grow progressively: in phase 1, both keepers and takers use *fixed strategies* (see Section 3.2.1 for the definition of fixed strategy); so there is no learning involved in this phase, and the purpose is to evaluate the performances of the fixed strategies, because these fixed strategies will play against learning agents in later phases. In phase 2, we perform experiments in two different Keepaway games: 2-Keepaway and 3-Keepaway[1]. In these games, only the ball holder keeper has learning ability and all the other robots, including all free keepers (i.e. keepers not holding the ball, see Section 3.1.1) and all takers, act according to fixed strategies (note that keepers and takers follow different fixed strategies). The purpose of this phase is to test the effectiveness of SARSA(λ)-based AARL in single-agent learning problems. In phase 3, we perform experiments in two different Takeaway games: 2-Takeaway and 3-Takeaway. Here, all takers are learning while all keepers act according to some fixed strategy (note that different keepers use different fixed strategies; these strategies will be described in more detail later in Section 4.2.1). The purpose of this phase is to test the effectiveness of SARSA(λ)-based AARL in multi-agent cooperative learning problems. Note that in all our experiments, the size of the field is 40×40. Note that we do not perform experiments in which both keepers and takers learn, because it has been reported that when both sides learn simultaneously, both sides could fail to converge [BBS08] [2].

[1] In a N-Keepaway or a N-Takeaway game ($N \in \mathbb{N}, N \geq 2$), there are $N+1$ keepers and N takers.

[2] This problem is often referred to as the *non-stationary* problem in multi-agent RL. Note that major RL algorithms, including SARSA(λ) and MAXQ-0, are guaranteed to converge in MDPs where the transition function (see Section 2.2.1) is stationary, i.e. $P(s'|s,a)$ is a fixed value for each states s, s' and action a. Since in multi-agent RL problems, for each agent, all other

In our SARSA(λ)-based AARL, we use both grounded semantics and preferred semantics to give recommendations, and use the *ASPARTIX* library to compute extensions (see Section 2.1.4). Because the number of arguments and the number of value orderings we use is not large (details given below), we can easily list all possible combinations of AF^{-}, compute their grounded extensions and preferred extensions before the learning starts, and store all these results in two tables, one for preferred extensions and the other for the grounded extension, where keys are states and entries are all extensions in these states. The reason that we compute these extensions in such an 'off-line' style is to reduce the action-selection time: the RoboCup Simulation Platform requires each agent to return which action it chooses within 125 milliseconds; our preliminary experiments indicate that when we compute all these extensions 'on-line', ASPARTIX sometimes is not able to finish within this time limit, and this will cause the agent to perform no action.

As for the communication between agents (this is needed when using preferred extensions to derive heuristics, see discussion in Section 3.2.4), because each learning agent is implemented as an independent thread, we let them share a text file, and in each learning step, each thread can write or read this file so as to allow the agents to communicate with one another.

Note that the learning keeper and the learning takers make decisions with different frequencies. The RoboCup Simulator allows each robot to update its action every 125 ms, called a *cycle*. In the keepers' team, the ball holder makes decisions and updates its action once a cycle.[3] However, in the Takeaway games, updating actions once a cycle results in poor performances [IE08]. So we make all the takers make decisions every five cycles.

We use the same learning settings as in most RoboCup Keepaway and Takeaway literature [SSK05, IE08, DGK11]. To be more specific, for all RL-based (including both standard SARSA(λ) and SARSA(λ)-based AARL) agents, the rewards they receive are as follows: in Keepaway games, the rewards received by the holder is the time (in seconds) that the holder possesses the ball; while in the Takeaway

agents are viewed as part of the environment, multi-agent RL problems are non-stationary and, therefore, major RL algorithms are not guaranteed to converge in these problems. Even though Hu and Wellman [HW98] have reported that in general-sum repeated matrix games, when both sides use RL, a Nash equilibrium will be finally reached, since the Keepaway and Takeaway games are far more complex than the problems they considered, their conclusion may not hold here. We have done some preliminary work on investigating the convergence property of RL algorithms in the RoboCup Soccer games where both keepers and takers are learning [GT13].

[3] When no keeper is controlling the ball, learning is suspended and all keepers act according to some hand-coded strategy until some keeper obtains the ball and becomes the ball holder. This hand-coded strategy is provided in the Keepaway platform developed by Stone et al. [SSK05].

game, in every learning step (i.e. every 5 cycles), each taker receives -1. As for the learning parameters, throughout all learning phases' experiments (i.e. the last two phases of experiments), we use the most widely used learning parameters [SSK05, IE08, DGK11] as follows: $\alpha = 0.125, \lambda = 1, \gamma = 1, \epsilon = 0.01$. As for the potential value (c in Algorithm 8), we set it as 2 and keep it fixed throughout the learning. We take $c = 2$ because we want this number to be comparable to the rewards received by either learning takers or learning keepers (note that, from our observation, when keepers are using the hand-coded strategy provided by [SSK05], each holder roughly holds the ball for 2 seconds); by doing this, agents' action-selection process can be effectively influenced by the potential value (see Section 2.2.5 for how potential values influence the action-selection process), while the learning agent can also easily overcome the imperfect instructions given the by potential values. All experiments are performed in RoboCup Simulator V15.1.0[4].

4.2.1 Performances of Both Sides Using Fixed Strategy

We use two fixed strategies for the keepers: the *random* strategy and the *hand-coded* strategy. In the random strategy, the ball holder chooses a random action to perform. The hand-coded strategy we use is the same as the one in Stone et al.'s work [SSK05].[5] The basic idea of this hand-coded strategy is similar to the recommendations we give in Section 3.1.1 for keepers: when all takers are quite 'far' from the ball, hold the ball; otherwise, choose a secure path to pass the ball.

As for takers, we implement three fixed strategies: the *random* strategy, the *always-tackle* strategy and the *argument-based* strategy. In the random strategy, each taker randomly chooses an action periodically; in the always-tackle strategy, all takers perform **TackleBall()** at all time; and in the argument-based strategy, at each state, we build SCAF, VSCAF, AF^{-}, compute the grounded extensions for AF^{-},[6] invoke function **getRecActFromExt** to obtain each taker's recommended action, and let each taker directly perform its recommended action; for those takers that receive no recommended action (i.e. whose recommended action is **null**), we

[4] This simulator can be obtained in `http://sourceforge.net/projects/sserver/?source=navbar`.

[5] Source code of this strategy can be found at `www.cs.utexas.edu/~AustinVilla/sim/keepaway/`, by downloading the file 'keepaway-0.6.tar.gz' and finding the file 'HandCodedAgent.cc'.

[6] We have also implemented an argument-based fixed strategy for takers with preferred extensions, and we find that this strategy's performance is very close to that of using grounded extensions, against both random and hand-coded keepers. As a result, the performances of argument-based takers' strategies can be viewed as the performance for both grounded and preferred semantics.

let them perform **TackleBall()**.

Because all these strategies are fixed, when they play against each other, the resulting performances are very stable, in the sense that in one experiment, the length of each episode is almost the same. So for each fixed strategy combination, we run one experiment consisting of 300 episodes, and compute the average episode duration (in seconds) and its standard errors[7] in this experiment. All results are presented in Table 4.1 and Table 4.2. We can see that under both settings we consider and against all three takers' fixed strategies, the keeper's hand-coded strategy significantly outperforms the random strategy (note that for keepers, longer episode durations indicate better performances). For the takers, when they play against the hand-coded keepers, the argument-based strategy clearly has the best performance (note that for takers, shorter episode durations indicate better performances). However, when playing against random keepers, the intuitively worst strategy — takers' random strategy — has the best performance among all takers fixed strategies. The reason for this seemingly counter-intuitive result is that when keepers are using the random strategy, they are more likely to pass the ball: for example, in games that involve 4 keepers and 3 takers, when keepers use the random strategy, the ball holder has 3/4 chance to pass the ball, regardless of the state in the field; on the contrary, when keepers use hand-coded strategies, the ball holder holds the ball until some taker is close to it. As a result, the random keepers pass the ball very often, so the most effective strategy for takers is to perform **MarkKeeper()** actions more often, because this will increase the chance to intercept the ball. Among all takers' fixed strategies, the random strategy performs **MarkKeeper()** most often and, therefore, has the best performance against random keepers; the argument-based strategy performs fewer **MarkKeeper()**, and its performance is worse than the random strategy's performance; the always-tackle strategy, which performs no **MarkKeeper()** at all, has the worst performance. Similarly, we can explain why when playing against random takers, the hand-coded keepers perform better than the random keepers: random takers are more likely to perform **MarkKeeper()** ($N/(N+1)$ probability for each taker), so keeper's strategies which hold the ball more often have better performances.

Given the analysis above,we gain more insights into the characteristics of each fixed strategy. Among the keeper's fixed strategies, the hand-coded strategy is

[7]The standard error is the standard deviation of the sampling distribution of a statistic. It can be viewed as the standard deviation of the error in the sample mean with respect to the true mean. For more details, please refer to e.g. [Eve02].

Table 4.1: The performances of 3 keepers playing against 2 takers, both using fixed strategies. 'K' and 'T' and shorthands for keepers and takers, respectively. The numbers are average episode durations (in seconds) $\pm$ their standard errors.

	random T	always tackle T	argument-based T
random K	7.89 ± 0.01	9.31 ± 0.01	8.91 ± 0.00
hand-coded K	22.54 ± 0.03	24.69 ± 0.08	12.78 ± 0.02

Table 4.2: The performances of 4 keepers playing against 3 takers, both using fixed strategies. 'K' and 'T' and shorthands for keepers and takers, respectively. The numbers are average episode durations (in seconds) $\pm$ their standard errors.

	random T	always tackle T	argument-based T
random K	7.48 ± 0.01	9.63 ± 0.01	7.88 ± 0.00
hand-coded K	23.58 ± 0.02	28.46 ± 0.06	12.80 ± 0.01

more 'intelligent' in the sense that it makes decisions based on the situation in the field, while the random strategy is more 'unpredictable', because it randomly selects actions to perform. As for the fixed strategies for takers, the always-tackle strategy never marks keepers, while the random strategy has a high chance to mark keepers, and the argument-based strategy is in between these two. To summarise, we can see that there is no clear winner amongst the takers' fixed strategies: each of them has its own strengths as well as weaknesses. In Section 4.2.2 and 4.2.3 below, we will use RL-based learning agents to play against these fixed strategies, to see which learning strategy can develop the best behaviour against its opponents most quickly.

4.2.2 Experiments in Keepaway games

We first give details of the game settings, and then present the performances of AARL[8] and SARSA(λ) against different fixed strategies of takers.

[8] Within this chapter, in cases without ambiguity, we use AARL and SARSA(λ)-AARL interchangeably.

Implementation

As mentioned in Section 3.1.1, at each moment, each agent has 'perfect knowledge' of the environment. However, this knowledge is consisting of coordinate locations, and RL cannot effectively work by directly using these locations as states [SSK05]. Stone et al. [SSK05] proposed *state variables* to represent each state, as shown in Table 4.3. The state variables are proposed not only for describing the situation in the field, but also for facilitating decision making in RL. For example, the distances between takers and the ball holder are state variables, because the holder can use this information to decide when to pass the ball and to whom to pass the ball. As we can see, all state variables are designed from the perspective of the ball holder, because the ball holder is the only learner in Keepaway. We say that these state variables are *holder-oriented*.

The arguments and values we use have been introduced in Example 1 and Example 5, in Chapter 3, respectively. We set the ordering of values (*Valpref*) as follows:

- *HOLD_LONG* $>_v$ *LESS_INT* $>_v$ *TEAM_LONG* iff K_1 is *safe*
- *LESS_INT* $>_v$ *HOLD_LONG* $>_v$ *TEAM_LONG* iff K_1 is *under threat*
- *LESS_INT* $>_v$ *TEAM_LONG* $>_v$ *HOLD_LONG* iff K_1 is *in danger*

When $\min_q dist(K_1, T_q) > 10$, K_1 is *safe*; when $5 < \min_q dist(K_1, T_q) \leqslant 10$, K_1 is *under threat*; when $0 < \min_q dist(K_1, T_q) \leqslant 5$, K_1 is *in danger*, where q ranges over all takers, i.e. $q \in \{1, \cdots, N\}$.

Because the time duration of an action in Keepaway is a variable, e.g. the time duration of action **PassBall(2)** can be different in different states, depending on the distance between the ball holder and keeper K_2, this game is modelled as a SMDP (see Section 2.2.1). Note that both Algorithm 1 and 8 can be directly used in SMDP problems.

Empirical Results in Keepaway

The performances of AARL-based (Algorithm 8) and SARSA(λ)-based (Algorithm 1) keepers against three fixed takers' strategies are shown in Figures 4.1, 4.2 and 4.3, respectively. Note that the length of each experiment under different settings is different: for example, results given in Figure 4.1(a) are averaged over 10 experiments, each lasting for 80 hours, while results shown in Figures 4.2 and 4.3

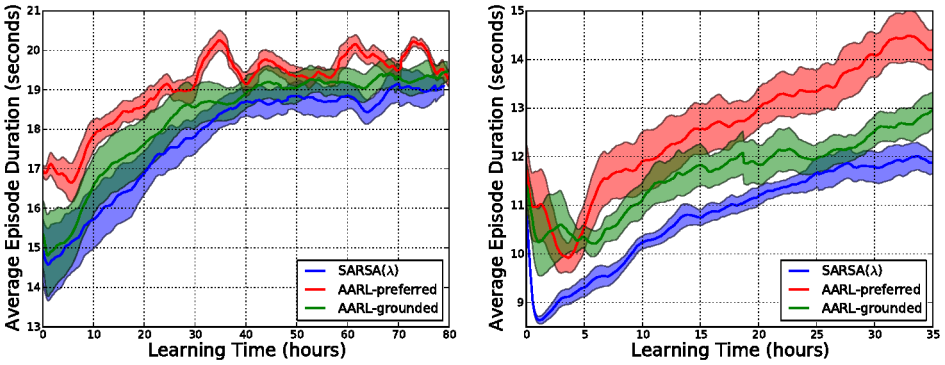
Table 4.3: State variables in a N-Keepaway game.

State Variable(s)	**Description**
$dist(K_p, C)$, $p \in [1, N+1]$	Distance between keepers and the centre of the court.
$dist(T_q, C)$, $q \in [1, N]$	Distance between takers and the centre of the court.
$dist(K_1, K_p)$, $p \in [2, N+1]$	Distance between K_1 and the other keepers.
$dist(K_1, T_q)$, $q \in [1, N]$	Distance between K_1 and the takers.
$\min_{q \in [1,N]} dist(K_p, T_q)$, $p \in [2, N+1]$	Distance between K_p and its closest taker.
$\min_{q \in [1,N]} ang(K_p, T_q)$, $p \in [2, N+1]$	The smallest angle between K_p and the takers with vertex at K_1.

are averaged over 40-hour experiments. The reason is that after a long time of running, the platform becomes unstable and may exit accidentally, so we can hardly make all experiments last for over 80 hours. From these performances, we can see that: at least one AARL algorithm significantly outperforms standard SARSA(λ), regardless of the fixed strategy the takers use and the number of agents involved in the game. However, none of the two different AARL implementations (with grounded extensions or with preferred extensions) is significantly better than the other in all settings. For example, when playing against random takers (Figure 4.1), AARL using preferred extensions outperforms AARL using grounded extensions, while when playing against argument-based takers (Figure 4.3), AARL using grounded extensions outperforms AARL using preferred extensions in 3-Keepaway games. The reason of this mixed result is still unclear and worth further investigation.

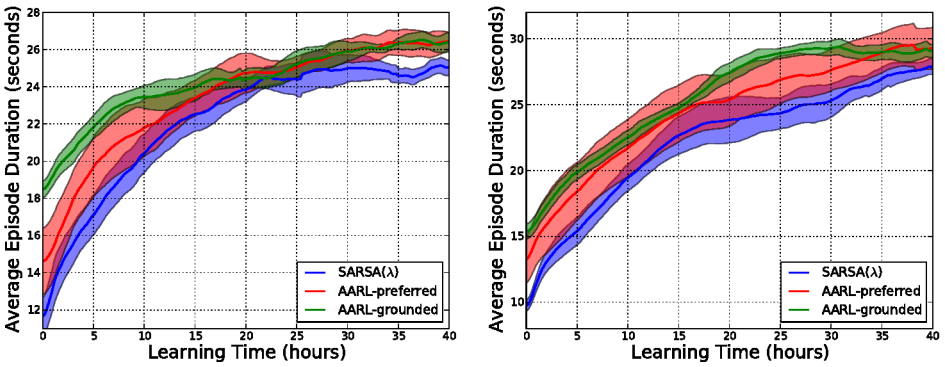
In order to further investigate the learning algorithms' performances after multiple hours of learning, we present 'last' performances of different algorithms against three different takers' fixed strategies in Tables 4.4, 4.5 and 4.6. Note that we do not present the pairwise *p-values*[9] in these tables, but we refer to them

[9] In statistics, the p-value is a function of the observed sample results (a statistic) that is used for testing a statistical hypothesis. If the p-value is equal to or smaller than the significance level (usually 0.05), it suggests that the observed data are inconsistent with the assumption that the *null hypothesis* is true, and thus that hypothesis must be rejected and the alternative hypothesis is accepted as true. All p-values presented in this chapter, unless stated otherwise, are computed by



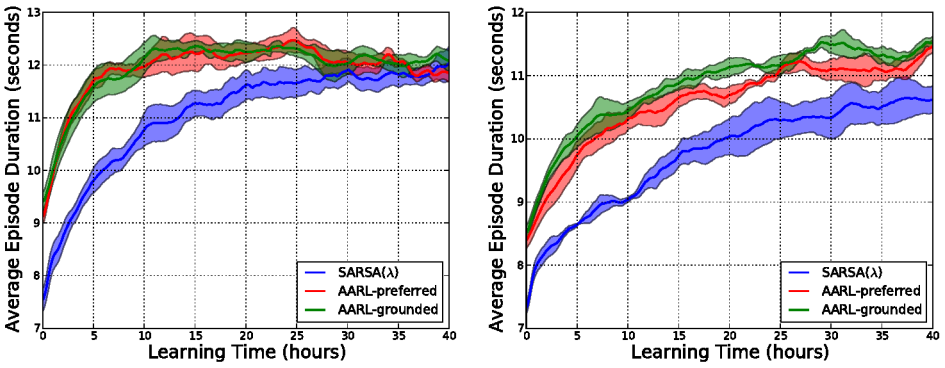
(a) 2-Keepaway (b) 3-Keepaway

Figure 4.1: The performances of learning keepers against random takers. Lighter colour areas represent 95% confidence intervals. All results are averaged over 10 independent experiments.



(a) 2-Keepaway (b) 3-Keepaway

Figure 4.2: The performances of learning keepers against always-tackle takers. Lighter colour areas represent 95% confidence intervals. All results are averaged over 10 independent experiments.



(a) 2-Keepaway (b) 3-Keepaway

Figure 4.3: The performances of learning keepers against argument-based takers. Lighter colour areas represent 95% confidence intervals. All results are averaged over 10 independent experiments.

in our discussion below. The *last* performances are computed by averaging the last 10 episodes' performances in all experiments. From Table 4.4, we can see that in 2-Keepaway games, when playing against random takers, AARL-preferred and AARL-grounded both significantly outperform SARSA(λ) (indeed, p-values are 0.01 and 0.02, respectively), while these two AARL algorithms do not have significant differences in their performances (since p-value is 0.80); in 3-Keepaway, however, AARL-preferred is significantly better than both SARSA(λ) (p-value < 0.01) and AARL-grounded (p-value < 0.01), while AARL-grounded does not have significant advantages over SARSA(λ) (since p-value is 0.83). However, all learning algorithms' performances are significantly worse than that of the hand-coded keeper's strategy (see Tables 4.1 and 4.2). We think the reason is because the takers' strategy is random and therefore unpredictable, and RL-based learning agents are not effective in adapting to such opponents.

From Table 4.5, we see that in 2-Keepaway, when playing against always-tackle takers, AARL-preferred significantly outperforms SARSA(λ) (p-value < 0.01) as well as AARL-grounded (p-value < 0.01), whereas AARL-grounded performs even worse than SARSA(λ) (p-value < 0.01). In 3-Keepaway, however, both AARL-preferred and AARL-grounded's performance are significantly better than that of SARSA(λ) (p-values are smaller than 0.01), while these two AARL algo-

using two-tailed t-test. For more details about p-values and the methods for computing p-values, please refer to e.g. [Eve02].

rithms do not have significant differences (p-value: 0.48). The reason of the mix results for AARL-grounded is still unclear. However, by comparing these learning algorithms' performances with that of hand-coded keeper's strategy (see Tables 4.1 and 4.2), we can see that all learning algorithms' performances are significantly better than that of the hand-coded strategy (p-values are all smaller than 0.01). This result indicates that when playing against some simple and easy-to-predict strategies, RL-based learning algorithms can achieve better performances than sophisticatedly designed hand-coded strategies.

From Table 4.6, we see that in both 2- and 3-Keepaway games, both AARL-preferred and AARL-grounded significantly outperform SARSA(λ) (p-values are all smaller than 0.01), and these two AARL implementations do not have significant difference in their performances (p-value: 0.17). However, by comparing these learning algorithms' performances with that of the hand-coded keeper's strategy (see Tables 4.1 and 4.2), all these learning algorithms perform significantly worse than the hand-coded strategy. This result indicates that when playing against a sophisticatedly design fixed strategies, SARSA(λ)-based learning algorithms still cannot achieve the same performance as people's hand-coded strategies.

Table 4.4: Performances (average episode durations (in second) $\pm$ standard errors) of learning keepers playing against random takers after several hours of learning. All performances are averaged over 100 episodes (10 episodes per experiment, 10 experiments for each algorithm).

Learning algorithms	After 80 hours' learning
2-Keepaway, SARSA(λ)	19.09 ± 0.12
2-Keepaway, AARL-preferred	19.43 ± 0.06
2-Keepaway, AARL-grounded	19.46 ± 0.10

Learning algorithms	After 60 hours' learning
3-Keepaway, SARSA(λ)	13.34 ± 0.05
3-Keepaway, AARL-preferred	14.49 ± 0.06
3-Keepaway, AARL-grounded	13.37 ± 0.13

4.2.3 Experiments in Takeaway games

We first give details of the Takeaway games' settings, and then present the performances of AARL and SARSA(λ) against the fixed keeper's strategies.

Table 4.5: Performances (average episode durations (in second) $\pm$ standard errors) of learning keepers playing against always-tackle takers after 40 hours of learning. All performances are averaged over 100 episodes (10 episodes per experiment, 10 experiments for each algorithm).

Learning algorithms	After 40 hours' learning
2-Keepaway, SARSA(λ)	24.97 ± 0.07
2-Keepaway, AARL-preferred	26.57 ± 0.07
2-Keepaway, AARL-grounded	24.60 ± 0.09
3-Keepaway, SARSA(λ)	27.97 ± 0.08
3-Keepaway, AARL-preferred	31.92 ± 0.09
3-Keepaway, AARL-grounded	32.00 ± 0.07

Implementation

Most existing work on Takeaway uses the holder-oriented state variables (e.g. [IE08, MZCZ08, DGK11]), the same as the state variables used in Keepaway games (see Table 4.3). However, since each taker is learning independently and takers need to cooperate with each other, *self-oriented* state variables, which describe the situation in the field from each taker's perspective, can be more helpful. Therefore, we combine taker's self-oriented state variables and some holder-oriented state variables to build a new state vector for learning takers, as shown in Table 4.7.

Besides the arguments and values we mentioned in Examples 2 and 6, we additionally use another two arguments:

- T_q**O(p)**: T_q performs **MarkKeeper(**p**)** IF K_p is open
- T_q**F(p)**: T_q performs **MarkKeeper(**p**)** IF K_p is far

The definitions of 'open' and 'far' in these two arguments are the same as in the keeper's arguments (Example 1 in Chapter 3). The reason (value) behind these two arguments are just opposite of the values of keeper's arguments **O(**p**)** and **F(**p**)**, respectively. For example, recall that the reason behind **O(**p**)** is 'passing the ball to the open keepers reduces the risk of the ball being intercepted'; so, from the takers' perspective, they should prevent the holder from passing the ball to an open keeper. We denote the values promoted by T_q**O(p)** and T_q**F(p)** as *MARK_OPEN* (standing for 'to mark a keeper that is open, so as to increase the success rate of interception') and *MARK_FAR* (standing for 'to mark a keeper that is far, so as to

Table 4.6: Performances (average episode durations (in second) $\pm$ standard errors) of learning keepers playing against argument-based takers after 40 hours of learning. All performances are averaged over 100 episodes (10 episodes per experiment, 10 experiments for each algorithm).

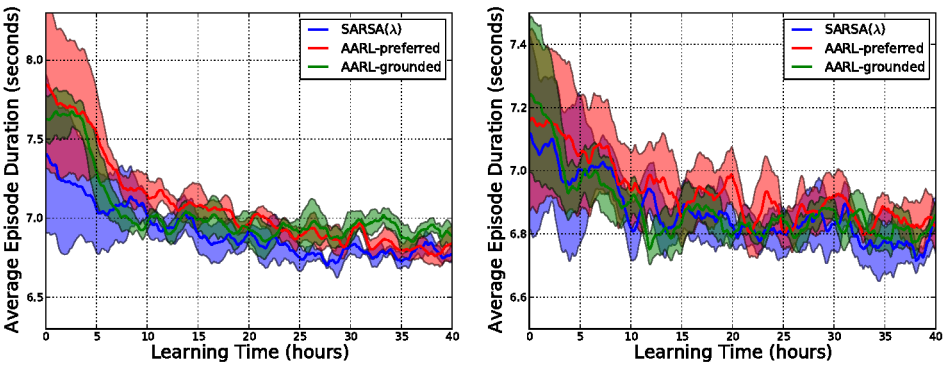
Learning algorithms	After 40 hours' learning
2-Keepaway, SARSA(λ)	11.92 ± 0.03
2-Keepaway, AARL-preferred	12.08 ± 0.02
2-Keepaway, AARL-grounded	12.11 ± 0.03
3-Keepaway, SARSA(λ)	10.74 ± 0.03
3-Keepaway, AARL-preferred	11.45 ± 0.02
3-Keepaway, AARL-grounded	11.50 ± 0.03

reduce the time keepers control the ball'), respectively. We keep the following ranking of values (*Valpref*) fixed throughout our experiments: *QUICK_TAC* $>_v$ *QUICK_MARK* $=_v$ *QUICK_CLOSE* $>_v$ *MARK_OPEN* $>_v$ *MARK_FAR*.

Empirical Results in Takeaway

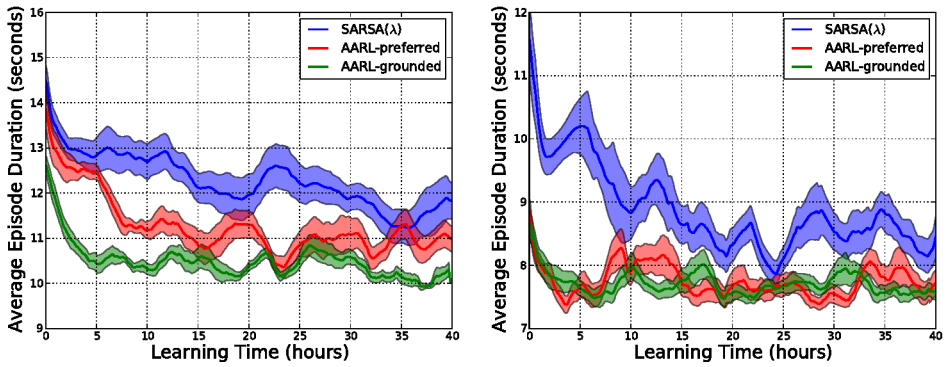
The performances of takers' learning strategies against keeper's random and hand-coded strategies are shown in Figures 4.4 and 4.5, respectively. From Figure 4.4, we can see that when playing against random keepers, all RL algorithms perform similarly (i.e. no algorithm is significantly better or worse than the others); also, we see that the 95% confidence intervals of all RL algorithms are wide and thus have much overlapping, indicating that in cooperative multi-agent learning problems, when playing against a random opponent, RL algorithms need to try many different strategies before they find the optimal policy. From Figure 4.5, we see that both AARL algorithms significantly outperform SARSA(λ) throughout the 40-hour experiments; however, with respect the relative goodness of two AARL implementations, we see that in 2-Takeaway, AARL-grounded significantly outperforms AARL-preferred most of the time, but in 3-Takeaway, no AARL implementation has significant advantages over the other during the learning process.

In order to further investigate the learning algorithms' performances after multiple hours of learning, we present 'last' performances of different takers' RL algorithms against two different keeper's fixed strategies in Tables 4.8 and 4.9. From Table 4.8 we can see that, when playing against random keepers, no algorithm has



(a) 2-Takeaway (b) 3-Takeaway

Figure 4.4: The performances of learning takers against random keepers. Lighter colour areas represent 95% confidence intervals. All results are averaged over 10 independent experiments.



(a) 2-Takeaway (b) 3-Takeaway

Figure 4.5: The performances of learning takers against hand-coded keepers. Lighter colour areas represent 95% confidence intervals. All results are averaged over 30 independent experiments.

Table 4.7: State variables for learning taker T_1 in a N-Takeaway game. State variables of other takers can be obtained similarly. The top three rows describe self-oriented variables, and the others describe variables about the keepers' relative layout.

State Variable(s)	**Description**
$dist(K_p, Me), \quad p \in [1, N+1]$	Distance between keepers and myself.
$dist(T_q, Me), \quad q \in [2, N]$	Distance between other takers and myself.
$ang(K_p, Me), \quad p \in [2, N+1]$	The angle between the free keepers and myself, with vertex at K_1.
$dist(K_p, K_1), \quad p \in [2, N+1]$	Distance between K_1 and the other keepers.
$dist(T_q, K_1), \quad p \in [2, N]$	Distance between K_1 and the other takers.
$\min_{j \in [1,N]} ang(K_p, T_q), \quad p \in [2, N+1]$	The smallest angle between K_p and the takers with vertex at K_1.

significant advantages or disadvantages over the other algorithms (in 2-Takeaway, the p-value between SARSA(λ) and AARL-preferred is 0.44, between SARSA(λ) and AARL-grounded is 0.31, and between the two AARLs is 0.91; in 3-Takeaway, the p-value between SARSA(λ) and AARL-preferred is 0.85, between SARSA(λ) and AARL-grounded is 0.71, and between the two AARLs is 0.40). This result is in line with our observation of the learning curves (Figure 4.4). Also, by comparing the takers' fixed strategies' performances against random keepers (see Tables 4.1 and 4.2), we can see that all learning algorithms' performances are significantly better than any of the takers' fixed strategies (all p-values are smaller than 0.01). This result indicates that when playing against some opponents that are difficult to predict, RL-based multi-agent cooperative learning outperforms people's hand-coded strategies.

From Table 4.9 we can see that when playing against hand-coded keepers, in both 2- and 3-Takeaway, both AARL implementations significantly outperform SARSA(λ) after 40 hours of learning (all p-values are smaller than 0.01). However, in 2-Takeaway, AARL-grounded is significantly better than AARL-preferred (p-value < 0.01), whereas in 3-Takeaway, these two AARL algorithms have no significant difference in their last performances (p-value: 0.17). Also, by comparing the takers' fixed strategies' performances against the hand-coded keepers (see Table 4.1 and 4.2), we can see that all learning algorithms' performances are significantly better than any of the takers' fixed strategies (all p-values are smaller

than 0.01). This result indicates that when playing against some carefully designed hand-coded opponents, RL-based multi-agent cooperative learning outperforms people's hand-coded strategies.

Table 4.8: Performances (average episode durations (in second) $\pm$ standard errors) of learning takers playing against random keepers after 40 hours of learning. All performances are averaged over 100 episodes (10 episodes per experiment, 10 experiments for each algorithm).

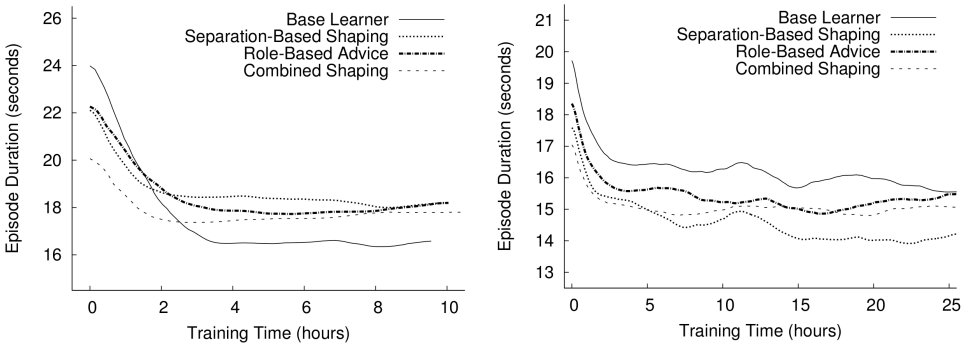
Learning algorithms	Performance after 40 hours' learning
2-Takeaway, SARSA(λ)	6.72 ± 0.07
2-Takeaway, AARL-preferred	6.84 ± 0.14
2-Takeaway, AARL-grounded	6.86 ± 0.12
3-Takeaway, SARSA(λ)	6.80 ± 0.18
3-Takeaway, AARL-preferred	6.76 ± 0.11
3-Takeaway, AARL-grounded	6.87 ± 0.07

Table 4.9: Performances (average episode durations (in second) $\pm$ standard errors) of learning takers playing against random keepers after 40 hours of learning. All performances are averaged over 300 episodes (10 episodes per experiment, 30 experiments for each algorithm).

Learning algorithms	Performances after 40 hours' learning
2-Takeaway, SARSA(λ)	12.55 ± 0.03
2-Takeaway, AARL-preferred	10.70 ± 0.01
2-Takeaway, AARL-grounded	10.09 ± 0.01
3-Takeaway, SARSA(λ)	9.21 ± 0.02
3-Takeaway, AARL-preferred	7.47 ± 0.05
3-Takeaway, AARL-grounded	7.39 ± 0.03

The state-of-the-art heuristics for takeaway games are proposed by Devlin et al. [DGK11]. They also use look-back advice to integrate heuristics into Takeaway, and their strategies' performances in 2- and 3-Takeaway (also on a 40×40 field) are shown in Figure 4.6(a) and 4.6(b), respectively. They also use SARSA(λ) as the standard RL algorithm, and all RL parameters they used are the same as ours[10].

[10] However, the state variables used in [DGK11] are slightly different from ours in that they did not



(a) 2-Takeaway (b) 3-Takeaway

Figure 4.6: The performances of two Takeaway games by using potential values proposed in [DGK11]. Reproduced from [DGK11] with permission.

They used three heuristics: *separation-based shaping* encourages each agent to take actions that increase its distance to other teammates; *role-based shaping* assigns each agent a role (either tackler or marker) a priori and only the tackler is encouraged to tackle; *combined shaping* is the integration of these two heuristics. Their strategies played against the keeper's hand-coded strategy (the same as we use). Their results showed that even though these heuristics successfully improved RL performances in 3-Takeaway (Figure 4.6(b)), they misled RL in 2-Takeaway (Figure 4.6(a)). We believe the reason for these mixed results lies in their lack of a systematic methodology to provide heuristics. Instead, AARL allows to integrate domain knowledge into RL while providing a high-level abstraction method (VAFs) for domain experts to propose domain knowledge. Also, the improvements of their heuristically-instructed strategies over SARSA(λ) are not as significant as with AARL.

4.3 Experiments in a Wumpus World Game

The need to resolve conflicts contained in the domain knowledge is not the only motivation for integrating arguments into RL. As we have pointed out in Section 3.1.2, we also want the heuristics-generation process to be intuitive, so that domain experts can easily check the correctness of the heuristics. Recent psychological and philosophical research has revealed that argumentation is a natural

use the takers' self-oriented state variables. Also, they use RoboCup Simulator v11.1.0. whereas we use v15.1.0. So their baseline performances are different from ours.

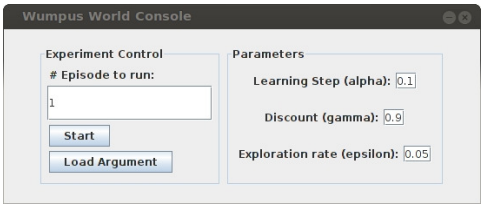
way for people to represent the domain knowledge and perform reasoning over this knowledge [MS11]. To test whether AARL can facilitate people — even non-experts of RL or argumentation — to propose high-quality domain knowledge and to use this knowledge to accelerate RL, we performed an experiment involving 128 students, who attended the course *Introduction to AI* (Course 231) in the Department of Computing, Imperial College London, in 2014. We asked them to propose arguments for the Wumpus World game (the one we introduced in Section 2.2.3, which includes gold and some shooting actions) within a one-hour tutorial session. The Wumpus World instance we consider has three golds, three Wumpuses, three pits, and the size of the world is 5×5. The students were a mix of second-year undergraduates majoring in Computer Science and master students with non-computing first degrees. All students had some background in logic and logic programming, but virtually no background on argumentation. Before this test, we gave them a one-hour lecture about basics of RL. We had informed them about the setting of the game and given them example arguments that represent the domain knowledge as follows: in any state, if the agent smells stench and three out of four navigating actions are safe (OK), then encourage shooting towards the remaining direction. The form of the arguments will be introduced shortly. We asked them to give their own domain knowledge in the form of arguments. Since VAFs were not introduced to them, we did not ask them to give values and told them that all arguments they gave were 'equally important'.

We developed a Graphical User Interface (GUI)[11] for the students to run the Wumpus World games, tune parameters of RL and observe the learning curves. The console of this system is shown in Figure 4.7(a). By clicking the button 'Start' in this window, two new windows pop up: one illustrates the situation of the world (Figure 4.7(b)), the other presents the current state and action of the learning agent (Figure 4.7(c)). When an experiment finishes, the learning curve of this experiment can be generated by clicking the button 'Plot Learning Curve'[12] in window 'Wumpus World Environment' (Figure 4.7(b)). To add arguments, students should first edit a file containing all arguments, click 'Load Arguments' (in the console window) and then select that file before starting an experiment. The form of an argument is as follows:

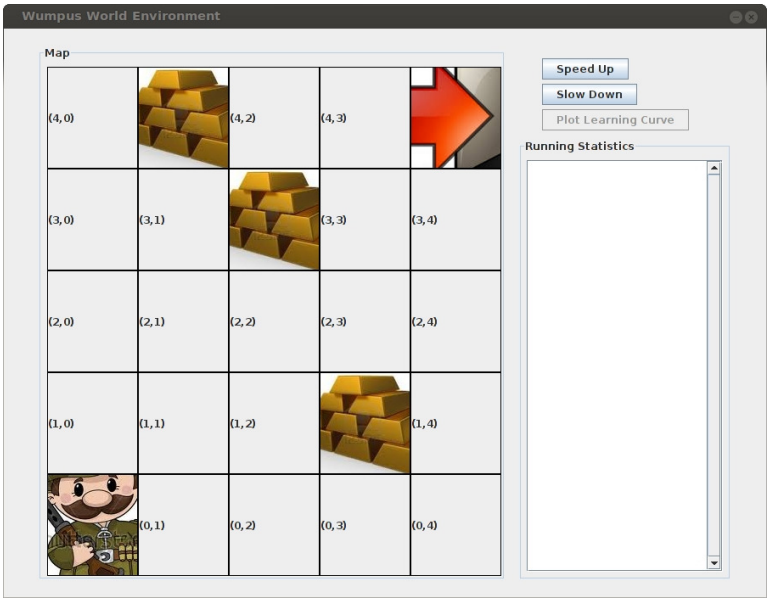
IF

[11] The system and its manual can be found at `www.doc.ic.ac.uk/~yg211/teaching.html`

[12] This button will be activated only after an experiment finishes.



(a) The 'Wumpus World Console' window. Note that once the learning begins (button 'Start' clicked), the button 'Load Argument' is frozen and arguments cannot be changed until this experiment finishes.



(b) The 'Wumpus World Environment' window. By default each learning step takes 500 ms, and this can be changed by clicking buttons 'Speed Up' and 'Slow Down'. After each episode finishes, some run-time statistics (as in Table 4.10) as well as the total reward received in this episode will be shown in the text field 'Running Statistics'.



(c) The 'Agent' window. Note that only part of the state vector is shown in this window: the locations of un-collected golds are used in our RL, but not shown in this window.

Figure 4.7: The GUI-based system of the Wumpus World game.

Condition1
...
ConditionN
THEN
\+ Action1
...
\+ ActionM
DONE

This argument means: when Condition1, $\cdots$, ConditionN are all true, then Action1 is good, $\cdots$, ActionM is good. Note that if there are multiple actions recommended by an argument, we split this argument into two arguments, whose conditions are the same, but support different actions. For each student's input arguments, we do this split, and the resulting arguments constitute the candidate argument set Arg^*. A condition in each argument can be one of the following:

go_left/*go_right*/*go_up*/*go_down* OK/BAD/UNKNOWN
Stench/Breeze/Glitter True/False

Given this form, our example domain knowledge can be represented by four arguments; one of them is:

IF
go_left OK
go_up OK
go_right OK
Stench True
THEN
\+ *shoot_down*
DONE.

Given the candidate argument set, we need to specify their conditions (OK, BAD and UNKNOWN), so that the learning agent can select the applicable arguments in each state. Considering the rewards of this game (see Section 2.2.1 and Section 2.2.3), in any state, if an action's corresponding Q-value is 0.0, we say it is UNKNOWN in this state; if an action's Q-value is (inclusively) larger than -50 and not equal to 0.0, this action is OK in this state; otherwise, we say this action is BAD in this state.

As for the attacks between arguments, since there is only one RL agent in this

problem, the attacks can be built according to rule b) in Definition 2 in Chapter 3. To be more specific, all arguments that support different actions will mutually attack each other in the SCAF.

Because we did not ask the students to provide any values or to rank the arguments they gave, we make all arguments promote the same anonymous value. As for type of extensions, we use preferred extensions to derive heuristics, because this type of 'credulous' semantics can make more use of the knowledge students provided (the grounded extension only recommends the convincingly good actions, whereas preferred extensions recommend all equally good actions, see Section 2.1.2). As for the potential value given to recommended actions (i.e. c in Algorithm 7), we set it to 3 because this value is comparable to the rewards received by the agent; our preliminary experiments indicate that by using this c value, the learning can be effectively influenced by the example arguments. This is all the domain knowledge we give to the learning agent (see line 2 in Algorithm 7).

Note that, since all arguments are equally preferred, and preferred extensions are used to give heuristics, each action that has any applicable arguments supporting it will have equal chance to be recommended to the agent. For example, suppose that in a state s, there are in total three applicable arguments: two arguments A, B supporting *go_right*, and another argument C supporting *go_up*; in this case, there are two preferred extensions: $\{A, B\}$ and $\{C\}$, and each of these extensions supports one action. Therefore, if we randomly choose a preferred extension to recommend an action, *go_right* and *go_up* have the same chance to be recommended, although there are more arguments supporting *go_right*. This is because all arguments supporting *go_right* must be contained in one and only one preferred extension. Formally, this is proved in Proposition 1 as follows:

Proposition 1. *In a single-agent learning problem, where all values in V are equally preferred, i.e.* $\forall v_1, v_2 \in \mathit{Valpref}, v_1 =_v v_2$. *Given the simplified* $\mathit{AF}^- = (\mathit{Arg}, \mathit{Att}^-)$, $\forall A \in \mathit{Arg}$, *there is one and only one preferred extension* E *such that* $A \in E$.

Proof. We first prove that there is at least one preferred extension containing A. Because all values are equally preferred, given the simplification rule of VAFs (see Section 2.1.3), no attacks in *Att* (the attack set of *SCAF*) will be deleted during the simplification. In other words, $\mathit{Att}^- = \mathit{Att}$. Since all attacks are mutual in Att^-, $\{A\}$ is an admissible extension, so there must exists at least one preferred

extension containing A.

Then we prove that there is at most one preferred extension containing A. We prove this by contradiction. Assume that, besides E, there exists another preferred extension P containing A and $\exists B \in Arg, B \in P, B \notin E$. Because A, B are all in preferred extension P, they are conflict-free; also, B can defend itself from all attacks, so $E \cup \{B\}$ is also admissible. This contradicts with the assumption that E is a preferred extension. So $E = P$. □

Each student's arguments were used together with the the provided example arguments to implement a SARSA(0)-based AARL algorithm, and we ran each student's AARL implementation for 100 independent experiments, each consisting of 100 episodes, and obtain the average performance of each student's AARL implementation. Given these averaged performances, we average over them and obtain the average performance of all AARL implementations. In addition, we ran the standard SARSA(0) algorithm without using any arguments also for 100 experiments, each also consisting of 100 episodes. Their performances are given in Figure 4.8. The RL parameters we used in all experiments are as follows: $\alpha = 0.1, \gamma = 1, \lambda = 0, \epsilon = 0$. From Figure 4.8 we can see that between episodes 34 and 48, the averaged performance all students' AARL implementations is significantly better than that of standard SARSA(0). In addition, we can see that, in average, AARL implementations receives the first positive reward at episode 40, whereas standard SARSA(0), in average, receives its first positive reward at episode 45. These facts indicate that students' arguments significantly improve the learning speed of SARSA(0).

In order to further investigate the performance variance among the students' AARL implementations, in Figure 4.9, we summarise the number of AARL implementations receiving different overall rewards[13]. Since the overall rewards received by the standard SARSA(0) is -217335, and each student's AARL implementation receives positive overall rewards (see Figure 4.9), we can clearly see that the arguments provided by the students effectively instruct the RL agents to converge faster.

Furthermore, in order to gain more insights into how the students' arguments accelerate the learning, we compute some statistics during the learning procedures and present them in Table 4.10. Note that all AARL results are averaged over

[13] For each RL implementation, its 'overall rewards' are the rewards received in all 100 independent experiments (i.e. all 10,000 episodes, because each experiment has 100 episodes). Other terms, e.g. 'overall Wumpuses killed', can be interpreted similarly.

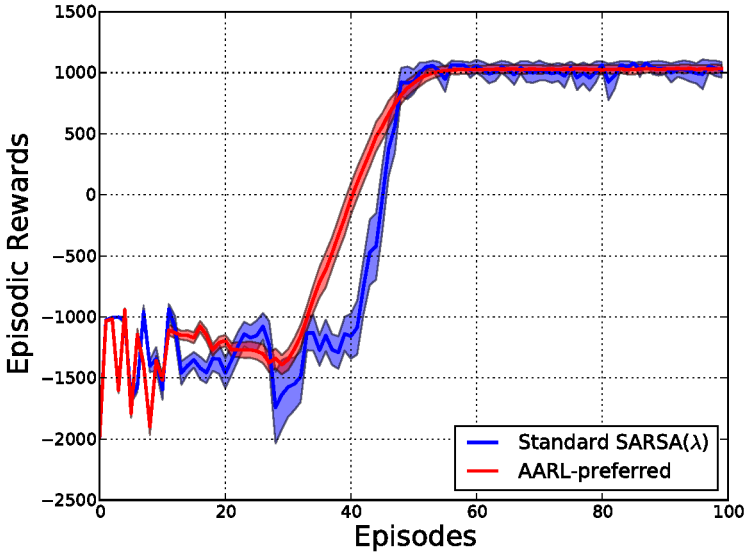


Figure 4.8: Performances of SARSA(0)-based AARL and standard SARSA(0) in the Wumpus World game. Lighter colour areas represent 95% confidence intervals.

all 128 AARL implementations. For example, data in the first row indicate that standard SARSA(0) killed 26008 Wumpuses throughout all 100 independent experiments, while AARL algorithms in average killed 26117 Wumpuses in 100 independent experiments, with a standard error of 6.4545. From data in Table 4.10 we can see that AARL collected significantly more gold, performed significantly fewer *pickup* and shooting actions in the experiments. These results indicate that the arguments provided by students help the learning agents to perform fewer 'useless' *pickup* and shooting actions and, therefore, reduce the negative rewards received by the agents. In particular, we find that most students (88 out of 128, 68.75 %) provided arguments encouraging the agents to perform *pickup* if the agents see glitter; this explains why AARL agents can collect more gold while performing fewer *pickup*. In addition, we can see that AARL agents' averaged probability of being killed (either by falling or being eaten) is significantly lower than that of SARSA(0); this also indicates that arguments provided by students can reduce 'unwise' navigating actions (for example, the example arguments we presented earlier in this subsection can prevent the agents from performing some 'unwise' moves) and, thus, increase the convergence speed of AARL. Considering

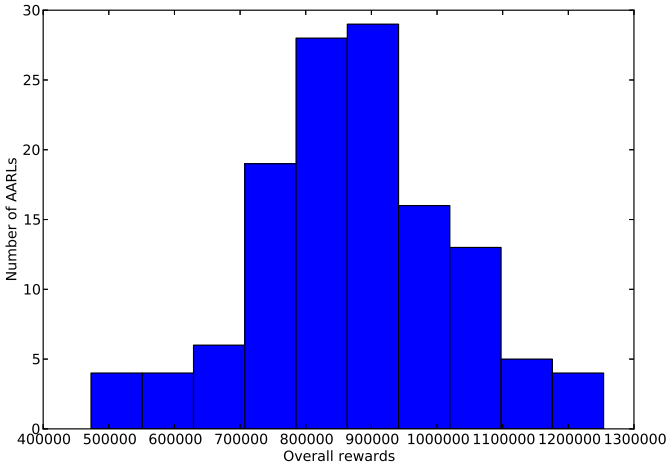


Figure 4.9: A histogram summarising the number of students' AARLs receiving different overall rewards.

that the students were new to RL and Argumentation, and the very limited time they had for editing arguments (the total tutorial time was one hour), these results illustrate that Argumentation can be easily used to represent and manage domain knowledge, even for non-experts, and suggest that AARL is an effective algorithm for integrating this knowledge into RL.

Table 4.10: Some statistics (means $\pm$ standard errors) of the experiments. All p-values are computed by using one-sample t-test.

	AARL	SARSA(0)	p-value
Overall Wumpuses killed	26117 ± 6.4545	26008	0.01
Overall gold collected	22805 ± 6.8081	22043	< 0.01
Overall no. of perf. *pickup*	31962 ± 19.4518	32669	< 0.01
Overall no. of perf. shoot actions	68107 ± 92.0424	74294	< 0.01
Prob. of falling into pit	0.3521 ± 0.0044	0.3806	< 0.01
Prob. of being eaten	0.0577 ± 0.0015	0.0706	< 0.01

4.4 Related Work

There is research on improving machine learning by using argumentation. Mozina et al. [MZB07] proposed argumentation based machine learning, which combines

arguments with the original examples of CN2 algorithm [CN89] to form argumented examples. Arguments can take one of the forms: *Classification because Reasons* or *Classification despite Reasons*. The arguments do not need to hold for the whole domain, which helps experts to articulate more accurate domain knowledge [MZB07]. The use of arguments significantly improves the performance of CN2. However, the relationships between different arguments are not taken into account in their technique, which restricts the effect argumentation has. Also, the machine learning technique considered, CN2, is supervised.

Some research has been devoted to incorporating domain knowledge into RL to improve its performance in MAS. Grzes and Kudenko [GK08] used high-level STRIPS knowledge in combination with reward shaping to search for an optimal policy and showed that the STRIPS-based reward shaping converges faster than the abstract MDP approach. However, their approach requires an explicit goal state and STRIPS-style domain knowledge, which are unavailable in several applications (e.g. Keepaway and Takeaway games). As for cooperative RL, Claus and Boutilier [CB98] distinguished two forms of multi-agent RL: *independent learners* (ILs), which only consider their own Q-values when choosing actions, and *joint action learners* (JALs), which search the exponential joint action space to maximise the sum of all agents' Q-values. They showed that even though JALs have much more information than ILs, they do not perform much differently from ILs. Our agents can be seen as ILs. Guestrin et al. [GLP02] used a *coordination graph* to restrain the coordination relationships so that actions are selected to maximise the sum of Q-values of only related agents. Thus, to obtain the Q-values of all related teammates, an agent has to compute all these Q-values or communicate with other agents. Similar approaches include the *Cooperative Hierarchical RL* proposed by Ghavamzadeh et al. [GMM06], in which coordination is only learnt in predefined *cooperative sub-tasks*, and the approach proposed by Lau et al. [LLH12], which models coordination among agents as *coordination constraints* and uses these to limit the joint action space for exploration. However, in all these cooperative RL approaches, domain knowledge is in the form of hard constraints and the action exploration is strictly constrained. Hence, learning cannot correct errors in the domain knowledge and the performances are highly sensitive to the quality of the domain knowledge.

In addition to PBRS techniques, there also exist other techniques for integrating heuristics into RL. Bianchi et al. [BRC08] proposed the Heuristically Accelerated RL method to integrate heuristic information into RL. However, in their approach,

for a proportion $1-\epsilon$ ($\epsilon \in \mathbb{R}, \epsilon \in [0,1]$), the action that is chosen by the heuristics will be executed, and for the other ϵ proportions, a random action will be chosen. As ϵ is often very small[14], the heuristically chosen actions will be executed very often and, as a result, in order to improve the performance of RL, fine-grained and accurate heuristics are needed. But in complex applications such as Keepaway and Takeaway games, precise heuristics can hardly be obtained, because the domain knowledge used to propose heuristics are usually imperfect.

4.5 Conclusion

In this chapter, we incorporate the VAFs-based argumentation framework we introduced in Chapter 3 into SARSA(λ) via look-ahead advice, and propose the resulting algorithm: *SARSA(λ)-based AARL*. We test this algorithm in three application domains to evaluate its effectiveness in terms of three aspects: in Keepaway games, we test the advantage of SARSA(λ)-based AARL over standard SARSA(λ) in single-agent learning problems; in Takeaway games, we test this advantage in cooperative multi-agent problems; and in a simple Wumpus World game, we test the usability of this algorithm. Experiments in the first two application domains indicate that SARSA(λ)-based AARL outperforms standard SARSA(λ) in terms of convergence speed, even when the domain knowledge used is imperfect. The last experiment, in which 128 students are asked to give argument-based domain knowledge to instruct a RL agent in the Wumpus World, suggests that SARSA(λ)-based AARL can be used by people that have little knowledge in RL or Argumentation to propose high-quality heuristics for RL.

As we have discussed in Section 3.3, AARL is a generic framework and has potential to be implemented in different kinds of RL algorithms, not limited to SARSA(λ). Because Hierarchical RL (HRL) algorithms, e.g. MAXQ, have proven to be more effective than flat RL algorithms, e.g. SARSA(λ), in many application domains [BM03], we believe that implementing a HRL-based AARL can result in more effective learning. However, to the best of our knowledge, there exists no PBRS techniques for HRL algorithms. Below in Chapter 5, we will introduce a PBRS technique for MAXQ-0 (see Section 2.2.3), so as to facilitate the implementation of MAXQ-based AARL, which will be presented in Chapter 6.

[14] Note that with a larger ϵ value, the probability of performing a random action is higher. So using a big ϵ value in HARL will weaken the effectiveness of the heuristics, because then the agent will follow the instruction of the heuristics less often.

5 Potential Based Reward Shaping for Hierarchical RL

In this chapter, we present a PBRS technique for one of the most widely used Hierarchical RL algorithms: *MAXQ* (see Section 2.2.3). To be more specific, we define potential values suitable to be used in MAXQ, and incorporate these potential values into the MAXQ decomposition functions such that the potential-values-augmented functions are still able to hierarchically decompose the whole problem into multiple sub-problems. In addition, based on these new MAXQ decomposition functions, we propose an algorithm called *PBRS-MAXQ-0*, and prove that under certain conditions, this algorithm is guaranteed to converge to the optimal hierarchical policy. Empirically, we also test the effectiveness of PBRS-MAXQ-0 in two application domains. Work in this chapter lays the foundation of MAXQ-based AARL, which will be introduced below in Chapter 6.

In the remainder of this chapter, we first present the main contribution of this chapter, i.e. the PBRS-MAXQ-0 algorithm in Section 5.1. In particular, we will describe how to integrate potential values into MAXQ and prove convergence property of the resulting algorithm. Then, in Section 5.2, we will show the empirical results of PBRS-MAXQ-0 in two application domains. In Section 5.3 we review related works and, finally, in Section 5.4, we conclude this chapter.

5.1 MAXQ with PBRS

In this section, we first present rules for integrating potential values into the original MAXQ decomposition functions, and reveal the relation between the potential-values-augmented MAXQ functions[1] and the original MAXQ functions. Then, we present the *PBRS-MAXQ-0* algorithm, which employs the potential-values-augmented MAXQ functions to find the optimal hierarchical policy (for defini-

[1] Note that we refer to value functions V (Definition 2.3), function Q (Definition 2.4) and completion function C (Definition 2.5) as the MAXQ functions.

tion of optimal hierarchical policy, see Section 2.2.3), and prove its convergence property.

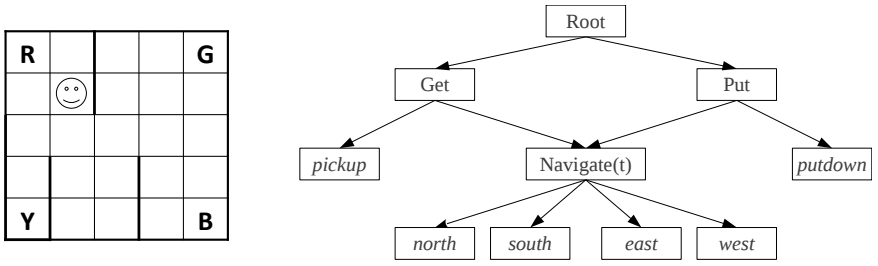
Throughout this section, let $H_M = \{M_0, \cdots, M_n\}$ be a hierarchical decomposition of the core MDP M. In the remainder of this chapter, we sometimes use i as a shorthand of M_i and, unless stated otherwise, we let i range over all sub-tasks in H_M, i.e. $i = 0, \cdots, n$. Before we introduce the integration rules, we first formally define the potential values. Note that, in Section 2.2.5, we discussed that in some classical PBRS techniques, e.g. *look-ahead advice* (LA), a potential value is in the form of $\Phi(s, a)$, indicating the goodness of performing action a at state s. However, in MAXQ, the goodness of performing a sub-task a is not only dependent on the current state s, but also on the parent sub-task that invokes a in state s. As a result, in MAXQ, we extend the potential values so that each potential value represents the goodness of performing sub-task a in state s under a's parent sub-task i: $\Phi(i, s, a)$. Formally:

Definition 4. Given MDP M and its MAXQ decomposition $H_M = \{M_0, \cdots, M_n\}$, let S_M be the set containing all available states in M, and let $C_M \subseteq H_M$ be the set that consists only of all composite sub-tasks in H_M. Let $\Phi_{H_M} : C_M \times S_M \times H_M \to \mathbb{R}$ be the *potential value function for* H_M such that:

- if a composite sub-task i terminates at state s, $\Phi_{H_M}(i, s, a) = 0$, where $a \in H_M$ can be any child sub-task of i;
- otherwise, $\Phi_{H_M}(i, s, a)$ can be any real number.

In the remainder of this chapter, in case of no ambiguity, we ignore the subscript H_M and simply use Φ to denote the potential value function.

To illustrate why, in MAXQ, we need to include parent sub-task as an input of the potential values, let us consider the *Taxi* problem [Die00]. The Taxi problem we consider consists of a 5×5 grid world with four landmarks, labelled (R)ed, (G)reen, (B)lue and (Y)ellow. An example scenario of the Taxi problem is shown in Figure 5.1(a). In the beginning of each episode, the taxi is in a random square, a passenger is in a random landmark and this passenger wants to be transported to one of the landmarks (also randomly chosen). The taxi has to navigate to the passenger's starting position, pick up the passenger, go to the destination and put down the passenger there, and then an episode ends. Note that the passenger's starting position could be the same with its destination; in this case, we still require



(a) Map. (b) Task graph

Figure 5.1: Settings of the Taxi problem. Thick lines in (a) represent walls.

the agent to pick up the passenger and put it down at the destination to finish the task.

There are six primitive actions available for the taxi: i) four navigation actions that move the taxi one square in the indicated direction: *north*, *south*, *east* and *west*; ii) the *pick_up* action, which transfers the passenger into the taxi and iii) the *put_down* action, which puts down the passenger at the square in which the taxi is. There is a reward of -1 for each action and an additional reward of $+20$ for successfully delivering the passenger. Also, there is an additional reward of -10 when the taxi performs *pick_up* and *put_down* illegally, i.e. performing *pick_up* (*put_down*) when there is no passenger in the current square (when the current square is not the destination of the passenger). Hitting a wall is a no-op (i.e. the taxi will remain in its original square) and results in no additional rewards.

We use a widely used MAXQ graph for this problem (used in, e.g. [Die00, JS08]), as shown in Figure 5.1(b). Consider the scenario in Figure 5.1(a), we can see that performing *north* within subtask Navigate(R) is intuitively 'good' in this scenario, while performing it within Navigate(B) is 'bad'. This domain knowledge cannot be represented by using a traditional potential value $\Phi(s, a)$, because it only considers the current state s and current action a, not taking into account the parent sub-task that invokes a at s. This example illustrates the necessity of including parent sub-task as an input of potential values in MAXQ.

5.1.1 Integrating Potential Values into the MAXQ Decomposition

As we discussed in Section 2.2.5, in classical PBRS techniques, potential values are integrated into the updating rules of Q-values (see Equation (2.7)) as well as the action-selection polices (see the biased ϵ-greedy policy in Equation (2.8)) so

as to influence the action-selection process in flat RL algorithms. Similarly, in order to use the potential values to influence the sub-task-selection in MAXQ, we also need to integrate the potential values that we defined in Definition 4 into the MAXQ functions V (Equation (2.3) in Section 2.2.3), Q (Equation (2.4) in Section 2.2.3), and C (Equation (2.5) in Section 2.2.3), and the sub-task-selection process (Algorithm 4 in Section 2.2.3). Here we first focus on the integration of potential values into the MAXQ functions; the integration into sub-task-selection will be discussed below in Section 5.1.2.

The potential-values-augmented counterparts of V, Q and C are $\tilde{V}$, $\tilde{Q}$ and $\tilde{C}$, respectively, and shortly in this subsection we will show that these new functions have close relationship with their corresponding original MAXQ functions. First, we formally define the potential-values-augmented MAXQ-functions as follows:

Definition 5. Given a MDP M, its MAXQ decomposition $H_M = \{M_0, \cdots, M_n\}$ and a hierarchical policy $\pi = \{\pi_0, \cdots, \pi_n\}$, for any sub-task $i \in H_M$ and state s, if i does not terminate at s, i.e. $s \in S_i$,[2] $\tilde{V}^\pi(i, s) = 0$; otherwise:

$$\tilde{V}^\pi(i,s) = \begin{cases} \sum_{s'}[P(s'|s,i)R(s'|s,i)] & \text{if } i \text{ is primitive,} \\ \tilde{Q}^\pi(i,s,\pi_i(s)) & \text{if } i \text{ is composite,} \end{cases} \tag{5.1}$$

where

$$\tilde{Q}^\pi(i,s,a) = \tilde{V}^\pi(a,s) + \tilde{C}^\pi(i,s,a); \tag{5.2}$$

when M_i does not terminate at state s,

$$\begin{aligned} \tilde{C}^\pi(i,s,a) = \sum_{s',\tau} P_i^\pi(s',\tau|s,a) \cdot 1_{S_i}(s') \cdot \gamma^\tau [V^\pi(\pi_i(s'),s') \\ + \tilde{C}^\pi(i,s',\pi_i(s')) + \Phi(i,s',\pi_i(s'))] - \Phi(i,s,a), \end{aligned} \tag{5.3}$$

where V^π is the function defined in Equation (2.3); otherwise (i.e. when M_i terminates at state s), $\tilde{C}^\pi(i,s,a) = 0$, where a can be any child of i. Note that $1_{S_i}(s')$ in Equation (5.3) is an indicator function: if $s' \in S_i$, $1_{S_i}(s') = 1$; otherwise, $1_{S_i}(s') = 0$.

We can see that the potential-values-augmented MAXQ functions are also defined in a hierarchical way, similar to their corresponding MAXQ functions (see

[2] S_i is the set containing all states in which sub-task i does not terminate. See Definition 1 in Section 2.2.3.

Section 2.2.3). In particular, the base case is function $\tilde{V}^{\pi}(i,s)$ when i is a primitive action, and all other functions are defined recursively based on it. Note that the value function V^{π} appearing in Equation (5.3) is the original V^{π} function defined in Equation (2.3) (Section 2.2.3), not the potential-value-augmented MAXQ function $\tilde{V}^{\pi}$.

Now we briefly describe why we integrate potential values into MAXQ functions in this way. We first revisit the integration rules of the PBRS for flat RL algorithms. From Equation (2.6) in Section 2.2.5, we can see that the potential values are integrated in a *temporal difference* (TD) form: each reward is augmented with the difference of the next state's potential value and the current state's potential value. The purpose is to cancel out most potential values when cumulating long-term rewards, and, therefore, to reconstruct the original value function V easily from the potential-values-augmented value function $\hat{V}$ (see Equation (2.6)). Similarly, we can see that potential values are integrated into C values also in a temporal difference way (see Equation (5.3)). Now we show that this way of integration can also facilitate the reconstruction of original completion functions C from their potential-values-augmented counterparts $\tilde{C}$.

Proposition 2. *Given a MAXQ decomposition* $\{\mathrm{M}_0,\cdots,\mathrm{M}_n\}$ *and its fixed hierarchical policy* $\pi=\{\pi_0,\cdots,\pi_n\}$*, for any composite sub-task* M_i*,* $0\leq i\leq n$*, and any state* s*,* $\tilde{C}^{\pi}(i,s,\pi_i(s))=C^{\pi}(i,s,\pi_i(s))-\Phi(i,s,\pi_i(s))$.

Proof. When i terminates in state s, $\tilde{C}^{\pi}(i,s,a)=0=C^{\pi}(i,s,a)-\Phi(i,s,a)$, where a can be any child of i; otherwise suppose that in one sample trajectory, the sequence of children sub-tasks executed within M_i starting from state s is $M_{a_1},M_{a_2},\cdots,M_{a_k}$. For each sub-task $M_{a_j}, j=1,\cdots,k$, we assume that it takes τ_j time steps to finish and terminates at state s_j. Since M_k terminates at state s_k, we can deduce that M_i also terminates at state s_k. Then we expand $\tilde{C}^{\pi}(i,s,a_1)$ as follows:

$$
\begin{aligned}
&\tilde{C}^{\pi}(i,s,a_1)\\
&=\gamma^{\tau_1}[V^{\pi}(a_2,s_1)+\tilde{C}^{\pi}(i,s_1,a_2)+\Phi(i,s_1,a_2)]-\Phi(i,s,a_1)\\
&=\gamma^{\tau_1}V^{\pi}(a_2,s_1)+\gamma^{\tau_1}\tilde{C}^{\pi}(i,s_1,a_2)+\gamma^{\tau_1}\Phi(i,s_1,a_2)-\Phi(i,s,a_1)\\
&=\gamma^{\tau_1}V^{\pi}(a_2,s_1)+\gamma^{\tau_1}\{\gamma^{\tau_2}[V^{\pi}(a_3,s_2)+\tilde{C}^{\pi}(i,s_2,a_3)+\Phi(i,s_2,a_3)]\\
&\quad-\Phi(i,s_1,a_2)\}+\gamma^{\tau_1}\Phi(i,s_1,a_2)-\Phi(i,s,a_1) \qquad (5.4)\\
&=\gamma^{\tau_1}V^{\pi}(a_2,s_1)+\gamma^{\tau_1+\tau_2}V^{\pi}(a_3,s_2)+\gamma^{\tau_1+\tau_2}\tilde{C}^{\pi}(i,s_2,a_3)
\end{aligned}
$$

$$+ \gamma^{\tau_1+\tau_2}\Phi(i, s_2, a_3) - \Phi(i, s, a_1) \tag{5.5}$$

$\dots$

$$= \gamma^{\tau_1}V^{\pi}(a_2, s_1) + \cdots + \gamma^{\tau_1+\cdots+\tau_{k-1}}V^{\pi}(a_k, s_{k-1}) + \gamma^{\tau_1+\cdots+\tau_{k-1}}\tilde{C}^{\pi}(i, s_{k-1}, a_k)$$
$$+ \gamma^{\tau_1+\cdots+\tau_{k-1}}\Phi(i, s_{k-1}, a_k) - \Phi(i, s, a_1) \tag{5.6}$$
$$= \gamma^{\tau_1}V^{\pi}(a_2, s_1) + \cdots + \gamma^{\tau_1+\cdots+\tau_{k-1}}V^{\pi}(a_k, s_{k-1}) - \Phi(i, s, a_1) \tag{5.7}$$
$$= C^{\pi}(i, s, a_1) - \Phi(i, s, a_1)$$

Equation (5.5) holds because $\gamma^{\tau_1}\Phi(i, s_1, a_2)$ and $-\gamma^{\tau_1}\Phi(i, s_1, a_2)$ in Equation (5.4) cancel each other out; Equation (5.6) is equivalent to (5.7) since $\tilde{C}^{\pi}(i, s_{k-1}, a_k) = -\Phi(i, s_{k-1}, a_k)$, proven as follows: since sub-task i terminates in state s_k, the indicator function $1_{S_i}(s_k) = 0$; so according to Equation (5.3), $\tilde{C}^{\pi}(i, s_{k-1}, a_k) = -\Phi(i, s_{k-1}, a_k)$. □

Remark. *Proposition 2 indicates that there exists a constant difference between $\tilde{C}^{\pi}(i, s, a)$ and $C^{\pi}(i, s, a)$, namely $\Phi(i, s, a)$. We can see that the potential values are added to the completion functions in a very similar way as in classical PBRS and look-ahead advice (LA): shaping rewards are added as the difference of some potential values of the current state and the following state (see Section 2.2.5). To prove this proposition, we expand value functions in a 'horizontal' way: we only expand $\tilde{V}^{\pi}(i, s, a)$ to its children sub-tasks' level, ignoring the recursive calling within each child sub-task, if any. By doing this, most of the augmented potential values cancel each other out, as in classical PBRS and LA.*

Example 9. Let us consider the scenario shown in Figure 5.1(a). For simplicity, we let $\gamma = 1$. Suppose that the passenger is now at R and her destination is Y. We call the state shown in Figure 5.1(a) s_0. We can easily see that the agent, following any optimal policy, will first invoke Get within Root at state s_0, and then perform Put. Within Get, the agent first performs Navigate(R), and then performs *pick_up* when it arrives at square R. Navigate(R), in turn, invokes two primitive navigation actions. Suppose Navigate(R) first performs *north*, followed by *west*. We now focus on the value of $C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north})$ and $\tilde{C}^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north})$, to see whether these two functions satisfy the property in Proposition 2.

We first compute $C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north})$. Because after *north* terminates, only action *west* is performed within Navigate(R), according to the meaning of completion functions — $C^{\pi}(i, s, a)$ represents the cumulative rewards received within sub-task i after performing a in state s by following policy π (see Section

2.2.3) — we can intuitively obtain that $C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) = -1$. To be more specific, according to the definition of C^π (Equation (2.5) in Section 2.2.3), $C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) = \gamma[V^{\pi^*}(\textit{west}, s_1) + C^{\pi^*}(\text{Navigate(R)}, s_1, \textit{west})]$, where s_1 is the state when the agent arrives at the square next to square R in the east. We can easily obtain that $V^{\pi^*}(\textit{west}, s_1) = -1$. Now we compute $C^{\pi^*}(\text{Navigate(R)}, s_1, \textit{west})$. Because by performing *west* at s_1, the agent arrives at square R (we denote this state as s_2) and sub-task *Navigate(R)* terminates, by Equation (2.3) and (2.5), $C^{\pi^*}(\text{Navigate(R)}, s_1, \textit{west}) = V^{\pi^*}(\text{Navigate(R)}, s_2) = 0$. Therefore, we obtain that $C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) = -1$.

Now we compute $\tilde{C}^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north})$. According to Equation (5.3), $\tilde{C}^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) = \gamma[V^{\pi^*}(\textit{west}, s_1) + \tilde{C}^{\pi^*}(\text{Navigate(R)}, s_1, \textit{west}) + \Phi(\text{Navigate(R)}, s_1, \textit{west})] - \Phi(\text{Navigate(R)}, s_0, \textit{north})$. Then we compute function $\tilde{C}^{\pi^*}(\text{Navigate(R)}, s_1, \textit{west})$. Note that because s_2 is the next state visited by the agent after s_1, and because Navigate(R) terminates at s_2, from Equation (5.3) we can see that $\tilde{C}^{\pi^*}(\text{Navigate(R)}, s_1, \textit{west}) = 0 - \Phi(\text{Navigate(R)}, s_1, \textit{west})$. So we can obtain that:

$$
\begin{aligned}
&\tilde{C}^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) \\
&= 1 \times [-1 - \Phi(\text{Navigate(R)}, s_1, \textit{west}) + \Phi(\text{Navigate(R)}, s_1, \textit{west})] \\
&\quad - \Phi(\text{Navigate(R)}, s_0, \textit{north}) \\
&= -1 - \Phi(\text{Navigate(R)}, s_0, \textit{north}) \\
&= C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) - \Phi(\text{Navigate(R)}, s_0, \textit{north}).
\end{aligned}
$$

So we can see that $\tilde{C}^{\pi^*}$ and C^{π^*} has the relation suggested by Proposition 2.

Given Proposition 2, we obtain the relation between V^π and $\tilde{V}^\pi$ as follows:

Proposition 3. *Given a MAXQ decomposition* $\{\text{M}_0, \cdots, \text{M}_n\}$ *and its fixed hierarchical policy* $\pi = \{\pi_0, \cdots, \pi_n\}$*, for any sub-task* M_{a^0} *(either primitive or composite),* $0 \leq a^0 \leq n$*, and any state* s*:*

- *if* M_{a^0} *is a primitive action,* $\tilde{V}^\pi(a^0, s) = V^\pi(a^0, s)$*;*
- *if* M_{a^0} *is a composite sub-task, suppose in a sample trajectory following* π*,* $\pi_{a^0}(s) = a^1, \cdots, \pi_{a^{m-1}}(s) = a^m$*, and* a^m *is a primitive action, then* $\tilde{V}^\pi(a^0, s) = V^\pi(a^0, s) - \sum_{j=1}^{m} \Phi(a^{j-1}, s, a^j)$.

Proof. The first item is immediately from the definition of $\tilde{V}^{\pi}$; the second item can be proved based on Proposition 2:

$$
\begin{aligned}
&\tilde{V}^{\pi}(a^0, s) \\
&= \tilde{V}^{\pi}(a^1, s) + \tilde{C}^{\pi}(a^0, s, a^1) \\
&= \tilde{V}^{\pi}(a^2, s) + \tilde{C}^{\pi}(a^1, s, a^2) + \tilde{C}^{\pi}(a^0, s, a^1) \\
&\cdots \\
&= \tilde{V}^{\pi}(a^m, s) + \tilde{C}^{\pi}(a^{m-1}, s, a^m) + \cdots + \tilde{C}^{\pi}(a^0, s, a^1) \\
&= V^{\pi}(a^m, s) + (C^{\pi}(a^{m-1}, s, a^m) - \Phi(a^{m-1}, s, a^m)) \\
&\quad + \cdots + (C^{\pi}(a^0, s, a^1) - \Phi(a^0, s, a^1)) \\
&= V^{\pi}(a^m, s) + C^{\pi}(a^{m-1}, s, a^m) + \cdots + C^{\pi}(a^0, s, a^1) \\
&\quad - (\Phi(a^{m-1}, s, a^m) + \cdots + \Phi(a^0, s, a^1)) \\
&= V^{\pi}(a^{m-1}, s) + C^{\pi}(a^{m-2}, s, a^{m-1}) + \cdots + C^{\pi}(a^0, s, a^1) \\
&\quad - (\Phi(a^{m-1}, s, a^m) + \cdots + \Phi(a^0, s, a^1)) \\
&\cdots \\
&= V^{\pi}(a^0, s) - (\Phi(a^{m-1}, s, a^m) + \cdots + \Phi(a^0, s, a^1))
\end{aligned}
$$

□

Remark. *Unlike Proposition 2, which focuses on the 'horizontal' expansion of the value functions, Proposition 3 goes 'vertically' down until the primitive actions and shows that there is a constant difference between $\tilde{V}^{\pi}(i, s)$ and $V^{\pi}(i, s)$, namely $\Phi(a^{m-1}, s, a^m) + \cdots + \Phi(a^0, s, a^1)$, the sum of of the potential values of* M_i*'s child, grandchild, $\cdots$ until the primitive sub-task in state s.*

Example 10. (Continuation of Example 9.) We compare $V^{\pi^*}(\text{Navigate(R)}, s_0)$ and $\tilde{V}^{\pi^*}(\text{Navigate(R)}, s_0)$ to see their relation. As for $V^{\pi^*}(\text{Navigate(R)}, s_0)$, we have $V^{\pi^*}(\text{Navigate(R)}, s_0) = V^{\pi^*}(\textit{north}, s_0) + C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north})$ according to Equation (2.3), We can easily see that $V^{\pi^*}(\textit{north}, s_0) = -1$, and we have obtained in Example 9 that $C^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) = -1$. As a result, we have $V^{\pi^*}(\text{Navigate(R)}, s_0) = -2$.

Then we compute $\tilde{V}^{\pi^*}(\text{Navigate(R)}, s_0)$. Similarly, based on Equation (5.1), we see that $\tilde{V}^{\pi^*}(\text{Navigate(R)}, s_0) = \tilde{V}^{\pi^*}(\textit{north}, s_0) + \tilde{C}^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north})$. Because *north* is a primitive action, $\tilde{V}^{\pi^*}(\textit{north}, s_0) = V^{\pi^*}(\textit{north}, s_0) = -1$, and we have $\tilde{C}^{\pi^*}(\text{Navigate(R)}, s_0, \textit{north}) = -1 - \Phi(\text{Navigate(R)}, s_0, \textit{north})$. As a

result, $\tilde{V}^{\pi^*}(\text{Navigate(R)}, s_0) = -2 - \Phi(\text{Navigate(R)}, s_0, \textit{north})$.

Until now, we have defined the potential-values-augmented MAXQ functions, and reveal their relations with their corresponding original MAXQ functions. These relations are important because they suggest how to reconstruct the original MAXQ functions' values by using the potential-values-augmented MAXQ functions' values; this reconstruction is important because we need to guarantee that the optimal polices obtained by the *PBRS-MAXQ-0* algorithm — which is introduced in the next subsection and uses the potential-values-augmented MAXQ functions during learning — are the same as those obtained by the classical MAXQ-0 algorithm (Algorithm 2 in Section 2.2.3).

5.1.2 The PBRS-MAXQ-0 Algorithm

In this subsection, we first propose an algorithm called *PBRS-MAXQ-0*, for integrating PBRS into the MAXQ-0 algorithm, and then we prove that PBRS-MAXQ-0 is guaranteed to converge to the optimal policy.

Before we introduce the PBRS-MAXQ-0 algorithm, we first introduce the sub-task-selection policy used in PBRS-MAXQ-0. Recall that in MAXQ-0, by using an OGLIE policy (see Section 2.2.3) on $\mathbf{D}_i$ (the SMDP corresponds to sub-task i, see Section 2.2.3), π_i^* can be found with probability 1. Now we propose the *biased OGLIE* (BOGLIE) policy as follows:

Definition 6. Given a MDP M, its MAXQ decomposition $H_M = \{M_0, \cdots, M_n\}$, its potential value function Φ_{H_M}, and the potential-values-augmented MAXQ functions $\tilde{V}$, $\tilde{Q}$ and $\tilde{C}$. A BOGLIE policy is a sub-task selection policy such that:

- Within each composite sub-task, each of its child is executed infinitely often in every state that is visited infinitely often.
- In the limit (i.e. after an infinitely long time of learning), for any composite sub-task a^0 and state s, the policy selects the *biased-greedy child* of a^0 at state s, namely

$$\underset{a^1 \in A_{a^0}(s)}{\operatorname{argmax}} [\tilde{Q}(a^0, s, a^1) + \sum_{j=1}^{m} \Phi(a^{j-1}, s, a^j)] \tag{5.8}$$

  with probability 1, where $A_{a^0}(s)$ is the set of available child of a^0 at state s, a^j ($j \in \{1, \cdots, m\}$) is the biased-greedy child of a^{j-1}, a^m is a primitive action.

- For each composite sub-task i and state s, there is a fixed order on sub-tasks in $A_i(s)$ such that the policy breaks ties in favour of the sub-task that appears earliest in the order.

By comparing Equation (5.8) and Equation (2.8) in Section 2.2.5, we can see that the BOGLIE policy follows the same spirit as the biased ϵ-greedy policy: these action selection policies choose actions that maximise a 'biased' value function. From Definition 5 and Proposition 3, we can easily see that maximising $\tilde{Q}(a^0, s, a^1) + \sum_{j=1}^{m} \Phi(a^{j-1}, s, a^j)$ is actually maximising $Q(a^0, s, a^1)$. Given this understanding, we can see that the BOGLIE policy essentially provides a method for obtaining the optimal polices with respect to the original MAXQ functions by using the potential-values-augmented MAXQ functions.

Given the BOGLIE policy, we now propose the PBRS-MAXQ-0 algorithm. Pseudo code of this algorithm is presented in Algorithm 9. Similarly to MAXQ-0 (Algorithm 2 in Chapter 2), whose principle component is a function that recursively invokes itself (i.e. the **MAXQ-0** function presented in Algorithm 2), PBRS-MAXQ-0's major component is also a recursively self-invoking function called **PBRS-MAXQ-0**, By comparing function **PBRS-MAXQ-0** and function **MAXQ-0**, we can see that their structures are very similar: between line 4 and 8, these two functions are exactly the same, indicating that when the input sub-task i is a primitive action, they update $V(i, s)$ in the same way; otherwise, when i is a composite sub-task, both these functions choose and perform a child sub-task of i (line 10 in both function, note that the sub-task-selection policy used in these functions are different), and use the resulting trajectory to update completion functions (between line 15 and 18 in both Algorithm 2 and 9).

Now we highlight some differences between MAXQ-0 and PBRS-MAXQ-0:

- The greedy evaluation function of these two algorithms are different, although their structures are similar. In MAXQ-0, function **Evaluate**(i, s) (Algorithm 3 in Chapter 2) is used to greedily evaluate value function $V(i, s)$. As we have discussed in Section 2.2.3, function **Evaluate**(i, s) selects a greedy path from i to a primitive descendant of i, so as to obtain the best estimation of $V(i, s)$. However, in PBRS-MAXQ-0, the values used during learning are the potential-values-augmented completion functions ($\tilde{C}$), so to evaluate $V(i, s)$, the evaluation function needs to reconstruct $V(i, s)$ by using functions $\tilde{C}$. Function **BiasedEvaluate**(i, s) implements this functionality. Its pseudo code is presented in Algorithm 10. Its structure is similar to

Algorithm 9 The PBRS-MAXQ-0 algorithm (adjusted from Algorithm 2).

1: /*Recursive function*/
2: function **PBRS-MAXQ-0**(Sub-task i, State s)
3: $seq :=<>$ /*initialise seq as an empty list*/
4: **if** i is a primitive action **then**
5: execute i, receive reward r and observe next state s'
6: $V_{t+1}(i,s) := (1-\alpha_t(i)) \cdot V_t(i,s) + \alpha_t(i) \cdot r_t$
7: push s onto the beginning of seq
8: **else**
9: **while** i does not terminate at state s **do**
10: choose an action a by using a BOGLIE policy
11: $childSeq :=$ **PBRS-MAXQ-0(**a, s**)**
12: observe next state s'
13: $V_t(a^*, s'), a^* :=$ **BiasedEvaluate**(i, s')
14: $N := 1$
15: **for** each s in $childSeq$ **do**
16: $\tilde{C}_{t+1}(i,s,a) := (1-\alpha_t(i))\tilde{C}_t(i,s,a) + \alpha_t(i)\{\gamma^N[V_t(a^*,s') + \tilde{C}_t(i,s',a^*) + \Phi(i,s',a^*)] - \Phi(i,s,a)\}$
17: $N := N+1$
18: **end for**
19: **append** $childSeq$ onto the front of seq
20: $s := s'$
21: **end while**
22: **end if**
23: **return** seq

24: /*Main Programme*/
25: initialise all V and $\tilde{C}$ values arbitrarily
26: initialise Φ values
27: **PBRS-MAXQ-0**(root sub-task 0, starting state s_0)

that of **Evaluate**: when the input sub-task i is a primitive action, these two functions both return its V^π value; otherwise, both these functions recursively invoke themselves so as to obtain the evaluation of all their children sub-tasks' value functions. However, the major difference between these two functions is that in function **BiasedEvaluate**, to reconstruct $V(i,s)$ by using $\tilde{C}$ values, the function needs to take into account all potential values along the path from i to its primitive descendants, according to Proposition 3. For this reason, in **BiasedEvaluate**, when selecting greedy child (line 8), the children's potential values are also added.

Also, note that the return type of these two evaluate functions are different: function **BiasedEvaluate** (Algorithm 10) returns two values: a sub-task as well as a value, whereas function **Evaluate** (Algorithm 3) only returns the value. This is because in PBRS-MAXQ-0, the selected child of i (i.e. a^*) is used in the updating of $\tilde{C}$ values (line 16 in Algorithm 9).

- The rules for updating completion functions in these two algorithms are different, but are in the same spirit. In line 16 in Algorithm 2, we see that C values are updated by taking into account both the existing C values as well as the new evaluation of the C value (by Equation (2.5), we can see that $V(i, s')$ is actually an evaluation of $C(i, s, a)$; this has been discussed in Section 2.2.3). Similarly, in line 16 in Algorithm 9, $\tilde{C}(i, s, a)$ is updated by taking into account its old value ($(1-\alpha_i t)\tilde{C}_t(i, s, a)$) as well as its new evaluation: from Equation (5.3), we can see that $\gamma^N[V_t(a^*, s') + \tilde{C}_t(i, s', a^*) + \Phi(i, s', a^*)] - \Phi(i, s, a)$ is actually an estimation of $\tilde{C}(i, s, a)$. So we can see that both MAXQ-0 and PBRS-MAXQ-0 uses similar TD-based rules to update their completion functions. Later when we prove the convergence property of PBRS-MAXQ-0 (proof of Theorem 3), we will see that this similarity ensures that PBRS-MAXQ-0 can obtain the same optimal hierarchical policy as MAXQ-0.

Algorithm 10 The function used in PBRS-MAXQ-0 to biased-greedily evaluate the value function of a sub-task in a certain state.

```
1: function BiasedEvaluate(Sub-task i, State s)
2: if i is a primitive action then
3:     return V_t(i, s)
4: else
5:     for all child sub-task a of i do
6:         V_t(a, s) := BiasedEvaluate(a, s)[0]
7:     end for
8:     a* := argmax_a[V_t(a, s) + C̃_t(i, s, a) + Φ(i, s, a)]
9:     return < V_t(a*, s) + C̃_t(i, s, a*) + Φ(i, s, a*), a* >
10: end if
```

Given function **BiasedEvaluate**, we propose the *biased ordered ϵ-greedy* sub-task-selection policy as an approximation of the BOGLIE policy. Pseudo code of this policy is presented in Algorithm 11. We can see that if the input sub-task i is a primitive action, this function simply returns i itself (line 3); otherwise, it computes the value functions of all children of i, also by invoking function **BiasedEvaluate**

(line 6). After that, it greedily selects children of i by taking into account their potential values and put them in $list$ (line 9). For a probability of ϵ, a random child of i is returned (line 12); for the other $1 - \epsilon$ probability, the function returns the best child of i (line 14 to 18).

Example 11. We illustrate the difference between using an OGLIE policy and the original MAXQ functions to select sub-tasks, and using a BOGLIE and potential-values-augmented MAXQ functions to select sub-tasks. We still consider the scenario shown in Figure 5.1(a), and use the same settings as in Example 9. We use these two different strategies to select a good sub-task in state s_0 within sub-task Navigate(R). Before we illustrate these two strategies, without loss of generality, we make some assumptions as follows: assume that $V(k, s) = \tilde{V}(k, s) = 0$ for any primitive action k and any state s, and $C(p, s, q) = \tilde{C}(p, s, q) = 0$ for any composite sub-task p, state s and p's child q. Also, we assume the order of navigation actions is *south* $>$ *east* $>$ *north* $>$ *west*. As for potential values used in BOGLIE, we have $\Phi(\text{Navigate(R)}, s_0, \textit{north}) = \Phi(\text{Navigate(R)}, s_0, \textit{west}) = 1$, and $\Phi(\text{Navigate(R)}, s_0, \textit{south}) = \Phi(\text{Navigate(R)}, s_0, \textit{ease}) = -1$. These potential values represent our domain knowledge that performing *north* and *west* (*east* and *south*) are good (bad) in state s_0 within sub-task Navigate(R). Also, we let $\epsilon = 0$, i.e. we always choose sub-task greedily.

First, we use an OGLIE policy, e.g. ordered ϵ-greedy (see Algorithm 4 in Section 2.2.3), and the original MAXQ functions. The input sub-task is Navigate(R), input state is s_0, input $\epsilon = 0$, and the input ranking $pref = \{\textit{south} > \textit{east} > \textit{north} > \textit{west}\}$. Because Navigate(R) is a primitive sub-task, this function invokes **Evaluate**(Navigate(R),s_0) to obtain the value functions of all children of Navigate(R) (line 6). We skip the execution of **Evaluate**, because we can easily see that all navigation actions' value function at s_0 are 0 (we initialise all primitive actions' value functions as 0). Hence, all navigation actions are added into $list$ (line 9). Since $\epsilon = 0$, this function will choose actions according to ranking $pref$ (line 15). Easily we can see that *south* is returned by this function. However, this action is not the optimal action in this state s_0 within Navigate(R).

Then we use a BOGLIE policy, e.g. the biased ordered ϵ-greedy policy (Algorithm 11) and the potential-values-augmented MAXQ functions to select action. All inputs of function **chooseBiasedChild** are the same as those of **chooseChild**. Again, because Navigate(R) is composite, this function invokes **BiasedEvaluate** to obtain all navigation primitive actions' V-values. Easily we can see that re-

turns of **BiasedEvaluate** are also 0, because all primitive actions' V-values are initialised as 0. So when choosing the greedy child of Navigate(R), potential values of these navigation actions will play decisive roles. By using the selection rule in line 9 in Algorithm 11, we can easily see that *north* and *west* will be chosen, and inserted into $list$. Because *north* is preferred than *west* in the ranking $pref$, *north* will be returned as the greedy child of Navigate(R) in state s_0.

Algorithm 11 The ordered ϵ-greedy sub-task-selection policy.

```
1: chooseBiasedChild(Sub-task i, State s, Float ε, SubtaskRanking pref)
2: if i is a primitive action then
3:    return i
4: else
5:    for all child sub-task a of i do
6:       V_t(a, s) := BiasedEvaluate(a, s)
7:    end for
8:    list := ()
9:    insert all argmax_a[V_t(a, s) + C̃_t(i, s, a) + Φ(i, s, a)] into list
10:   initialise a random number r ∈ ℝ, r ∈ [0, 1]
11:   if r < ε then
12:      return a random child of i
13:   else
14:      if there are more than one sub-task in list then
15:         return the sub-task in list that is most preferred according to pref
16:      else
17:         return the only sub-task in list
18:      end if
19:   end if
20: end if
```

Example 11 illustrates that by using a BOGLIE policy and the potential-values-augmented MAXQ functions, potential values can effectively instruct the agents to select more promising actions more quickly. Furthermore, we want the PBRS-MAXQ-0 with BOGLIE policy to converge to the optimal hierarchical policy regardless of the potential values being used. This property is proved as follows.

Theorem 3. *Given a MDP M, its MAXQ decomposition $H_M = \{\mathrm{M}_0, \cdots, \mathrm{M}_n\}$ and its potential value function Φ_{H_M}, suppose all immediate rewards and all potential values are bounded, and $\tilde{\pi}_i$ is a BOGLIE policy for sub-task i. If $\alpha_t(i) > 0$*

is a sequence of constants for sub-task i such that

$$\lim_{T\to\infty}\sum_{t=1}^{T}\alpha_t(i)=\infty \qquad \textit{and} \qquad \lim_{T\to\infty}\sum_{t=1}^{T}\alpha_t^2(i)<\infty, \tag{5.9}$$

then with probability 1, algorithm PBRS-MAXQ-0 converges to π^*, the unique hierarchical optimal policy for M consistent with H_M.

Proof. Recall that under the assumptions we made in the theorem, MAXQ-0 with OGLIE policies are guaranteed to converge. In this proof, we prove that PBRS-MAXQ-0 also converges to the original optimal policy with probability 1 by showing that the learning dynamics of using a BOGLIE policy in PBRS-MAXQ-0, in which all $\tilde{C}$ values are initialised arbitrarily, is the same as using a OGLIE policy in MAXQ-0, whose C values are initialised the same as the corresponding Φ values.

We define two learning agents, L and $\tilde{L}$. They experience the same sample trajectories throughout their learning. $\tilde{L}$ uses PBRS-MAXQ-0 and a BOGLIE policy, and all its $\tilde{C}$ values are, without loss of generality, initialised as 0; L, on the other hand, uses MAXQ-0 and an OGLIE policy, and all its C values are initialised as the corresponding Φ values: i.e. for any composite sub-task i, state s and child of i: a, $C(i,s,a)=\Phi(i,s,a)$. Also, for any primitive action i and state s, the $V(i,s)$ value for agent L and the $\tilde{V}(i,s)$ value for agent $\tilde{L}$ are initialised as the same, yet arbitrary, real number. Here, without loss of generality, we initialise all these values as 0. $\tilde{C}$ values for agent $\tilde{L}$ are updated according to line 16 in Algorithm 9. By using Proposition 3, we expand this updating rule as follows:

$$\begin{aligned}\tilde{C}_{t+1}(i,s,a) = (1-\alpha_t(i))\cdot\tilde{C}_t(i,s,a) + \alpha_t(i)\{\gamma^N \max_{a^0\in A_i(s')}[\tilde{V}_t(a^m,s') \\ + \sum_{j=1}^{m}(\Phi(a^{j-1},s',a^j)+\tilde{C}(a^{j-1},s',a^j)) + \tilde{C}_t(i,s',a^0) \\ + \Phi(i,s',a^0)] - \Phi(i,s,a)\},\end{aligned} \tag{5.10}$$

where $a^j, j\in\{1,\cdots,m\}$ is the biased greedy child of a^{j-1} at state s', and a^m is a primitive action. As for C values for agent L, its updating rule is presented in

line 16 in Algorithm 2, which can be expanded as follows:

$$C_{t+1}(i,s,a) = (1-\alpha_t(i)) \cdot C_t(i,s,a) + \alpha_t(i)\{\gamma^N \max_{a^0 \in A_i(s')}[V_t(a^m, s') + \sum_{j=1}^{m} C(a^{j-1}, s', a^j) + C_t(i, s', a^0)]\}, \quad (5.11)$$

where a^j is the *current greedy child*[3] of a^{j-1} at state s, $j = \{1, \cdots, m\}$, and a^m is a primitive action. Both these updating rules can be viewed as updating the values with an error term scaled by α, the learning rate. We refer to the error term in Equation (5.10) as $\delta\tilde{C}$, and the error term in Equation (5.11) as δC. We also track the total change in $\tilde{C}$ and C during the learning. The difference between the initial and current values in $\tilde{C}$ and C are referred to as $\Delta\tilde{C}$ and ΔC, respectively. So the current $\tilde{C}$ and C can be represented as follows, for any sub-task i, state s and child sub-task a:

$$\tilde{C}_t(i,s,a) = \Delta\tilde{C}(i,s,a),$$
$$C_t(i,s,a) = \Phi(i,s,a) + \Delta C(i,s,a).$$

Given the terminologies above, we just need to prove that given any learning trajectories, for any sub-task i, state s and i's child a, $\Delta\tilde{C}(i,s,a)$ always equals $\Delta C(i,s,a)$. This is proved by induction, and the base cases is to prove that $\Delta\tilde{C}(i,s,a) = \Delta C(i,s,a)$ when all $\tilde{C}$ and C values have not been updated. Because all $\tilde{C}$ and C equal their initial values, $\Delta\tilde{C}(i,s,a) = \Delta C(i,s,a) = 0$. So the base case holds.

For the inductive case, assume that $\Delta\tilde{C} = \Delta C$ for all entries. We show that by using a trajectory in which sub-task a is performed within sub-task i at state s leading to state s' and taking N time slots, $\delta\tilde{C}(i,s,a) = \delta C(i,s,a)$. For the error term of $\tilde{C}(i,s,a)$:

$$\delta\tilde{C}(i,s,a) = \gamma^N \max_{a^0 \in A_i(s)}[\tilde{V}_t(a^p, s') + \sum_{j=1}^{p}(\Phi(a^{j-1}, s', a^j) + \tilde{C}(a^{j-1}, s', a^j)) + \tilde{C}_t(i, s', a^0) + \Phi(i, s', a^0)] - \Phi(i,s,a), \quad (5.12)$$

where a^j is the current biased-greedy child of a^{j-1}, $j \in \{1, \cdots, p\}$, and a^p is a primitive action, where p is the number of descendents of a^0 in the selected path

[3] For a sub-task i and state s, i's current greedy child at state s is $\text{argmax}_{a^0 \in A_i(s)}[Q_t(i,s,a^0)]$, where t is the current time slot. From line 8 in Algorithm 3 we can see that the V value in MAXQ-0 is computed by selecting the current greedy descendants.

from a^0 to the primitive action a^p. We denote $\tilde{\pi}_i^t(s)$ as the policy that selects the biased-greedy child of i at s according to the current $\tilde{C}$ and $\tilde{V}$ values, where t is the current time slot, i can be any composite sub-task, and s can be any state in S_i (note that S_i is the set containing all active states of sub-task i, see Section 2.2.3). So in Equation (5.12), $a^j = \tilde{\pi}_{a^{j-1}}^t(s')$, for $j = \{1, \cdots, p\}$.

Then we look into the error term of C:

$$\delta C(i,s,a) = \gamma^N \max_{a^0 \in A_i(s)} [V_t(a^q, s') + \sum_{k=1}^{q} C(a^{k-1}, s', a^j) + C_t(i, s', a^0)] \tag{5.13}$$

We denote $\pi_i^t(s)$ as the policy that selects the current greedy child of i at s according to the current C and V values, where t is the current time slot, i can be any composite sub-task, and s can be any state in S_i. So in Equation (5.13), $a^k = \pi_{a^{k-1}}^t(s'), k \in \{1, \cdots, q\}$, and a^q is a primitive action, where q is the number of descendents of a^0 in the selected path from a^0 to the primitive action a^q.

To prove $\delta\tilde{C}(i,s,a) = \delta C(i,s,a)$, we only need to prove that for any descendant composite sub-task e of i, $\tilde{\pi}_e^t(s) = \pi_e^t(s)$, i.e. the biased-greedy child of e at s for agent $\tilde{L}$ is the same with the greedy child of e at s for agent L. We prove this also by induction. The base case is prove this holds when every child of e is primitive. In this case,

$$\tilde{\pi}_e^t(s) = \text{argmax}_{a \in A_e(s)}[\tilde{V}(a,s) + \tilde{C}(e,s,a) + \Phi(e,s,a)],$$
$$\pi_e^t(s) = \text{argmax}_{a \in A_e(s)}[V(a,s) + C(e,s,a)].$$

By the inductive assumption that $\Delta\tilde{C}(e,s,a) = \Delta C(e,s,a)$, we can easily see that $C_t(e,s,a) = \tilde{C}_t(e,s,a) + \Phi(e,s,a)$. Also, because all children of e are primitive, by definition, $\tilde{V}(a,s) = V(a,s)$. Therefore, the base case holds.

For the inductive case, assume that for any e's composite descendant f, $\tilde{\pi}_f^t(s) = \pi_f^t(s)$. Then

$$\begin{aligned}\tilde{\pi}_e^t(s) = \underset{a^0 \in A_e(s)}{\text{argmax}}\{&\tilde{V}(a^p, s) + \sum_{j=1}^{p}[\tilde{C}(a^{j-1}, s, a^j) + \Phi(a^{j-1}, s, a^j)] \\ &+ \tilde{C}(e,s,a^0) + \Phi(e,s,a^0)\},\end{aligned} \tag{5.14}$$

where $a^j = \tilde{\pi}^t_{a^{j-1}}(s), j = \{1, \cdots, p\}$, and a^p is a primitive action, and

$$\pi^t_e(s) = \underset{a^0 \in A_e(s)}{\text{argmax}}[V(a^q, s) + \sum_{k=1}^{q} C(a^{k-1}, s, a^k) + C(e, s, a^0)], \tag{5.15}$$

where $a^k = \pi^t_{a^{k-1}}(s), k = \{1, \cdots, q\}$, and a^q is a primitive action. For each $a^0 \in A_e(s)$, $\tilde{\pi}^t_{a^0}(s) = \pi^t_{a^0}(s)$, so $a^j = a^k$ when $j = k = 1$. By recursively applying this, we can obtain that $a^j = a^k$ iff $j = k$, and $p = q$. By the inductive assumption, we can obtain that $C(i, s, a) = \tilde{C}(i, s, a) + \Phi(i, s, a)$, and therefore by replacing C values in Equation (5.15) by $\tilde{C} + \Phi$, we can obtain that $\tilde{\pi}^t_e(s) = \pi^t_e(s)$. Hence, Equation (5.13) equals Equation (5.12).

By far, we have proved that by using the same learning experiences, $\tilde{C}$ and C experience the same amount of updates, i.e. $\Delta\tilde{C}(i, s, a) = \Delta C(i, s, a)$, for any possible subtask-state-child pair (i, s, a). Since $C(i, s, a)$ is guaranteed to converge to a constant, denoted as $C^{\pi^*}(i, s, a)$, $\tilde{C}(i, s, a)$ is also guaranteed to converge to a constant: $\tilde{C}^{\tilde{\pi}^*}(i, s, a) = C^{\pi^*}(i, s, a) - \Phi(i, s, a)$. As we have proved that when $\Delta\tilde{C} = \Delta C$, for any sub-task i, the biased-greedy child of i in PBRS-MAXQ-0 equals the greedy child of i in MAXQ-0. Hence, we prove that $\tilde{\pi}^*_i(s) = \pi^*_i(s)$ for any sub-task i and state s. □

Remark. *Theorem 3 not only proves that PBRS-MAXQ-0 is guaranteed to converge to the optimal hierarchical policy by using a BOGLIE policy, but also reveals that using PBRS-MAXQ-0 (in which all $\tilde{C}$ are initialised as 0) is equivalent to using MAXQ-0 (in which all C values are initialised as their corresponding potential values). Also, this theorem suggests the 'perfect' potential values: in classical PBRS for MDP (see Section 2.2.5), the 'perfect' potential value in state s is $V^*(s)$, i.e. the value function at s by following the optimal policy. This potential value is regarded as 'perfect' because when $\Phi(s) = V^*(s)$, $\tilde{V}^*(s) = 0$, and this is viewed as a particularly easy value to learn [NHR99]. Similarly, in a MAXQ decomposition, we can see that when $\Phi(i, s, a) = C^{\pi^*}(i, s, a)$, where π^* is an optimal hierarchical policy for this MAXQ, then $\tilde{C}^{\pi^*}(i, s, a) = 0$ and $\tilde{V}^{\pi^*}(i, s) = V^{\pi^*}(a^m, s)$, and we believe that these values are also easy to learn. Note that our intuition on the potential values are also in line with the perfect values for the potential values: the higher the $C^{\pi^*}(i, s, a)$ value, the more performing M_a in M_i at state s is likely to be, and the potential value $\Phi(i, s, a)$ should thus be higher.*

5.2 Experiments

We test the performance of PBRS-MAXQ-0 in two application domains: the *Taxi* problem and a stochastic *Wumpus World* problem. As baseline algorithms, we also implement MAXQ-0 (Algorithm 2 in Section 2.2.3), SARSA(0) (Algorithm 1 in Section 2.2.2) and SARSA(0) with look-ahead advice (LA-SARSA(0), see Section 2.2.5 for these two algorithms).

5.2.1 Taxi Problem

The basic settings of the Taxi problem have been introduced earlier in Section 5.1.1. To make this problem more challenging, we make the navigation actions (i.e. *north*, *south*, *east* and *west*) have some probabilities to fail: each of these actions can move the agent one square in the intended direction with probability 0.8 and in each perpendicular direction with probability 0.1. As for termination states of each sub-task (definition of termination state is given in Definition 1 in Section 2.2.3), we let Root terminate when the passenger is delivered; Get terminate when the taxi picks up the passenger; Put terminate when the taxi delivers the passenger; Navigate(X), where X can be any landmark (R, Y, B and G), terminate when the taxi arrives at square X.

The state representation we use consists of four elements: the taxi's current position, a boolean variable indicating whether the passenger is picked or not, the passenger's starting position and its destination. Both MAXQ-0 and PBRS-MAXQ-0 use the same task graph shown in Figure 5.1(b) in Section 5.1.1. As for learning parameters, for MAXQ-0 and SARSA(0), we have $\gamma = 1, \alpha = 1$ and $\epsilon = 1$ in the beginning of each experiment, and we let α decrease at a rate of 0.999, i.e. $\alpha_t = \alpha_{t-1} \times 0.999$. Similarly, ϵ decreases at a rate of 0.99. For PBRS-MAXQ-0 and LA-SARSA(0), we use the same γ value, initialise α and ϵ as 0.2 and 0.5, respectively, and decrease them at a rate of 0.999 and 0.9, respectively. All these learning parameters are selected based on our preliminary experimental results, so as to minimise each algorithm's convergence speed.

The potential values $\Phi(i, s, a)$ used in PBRS-MAXQ-0 are given as follows:

- When sub-task a is Get, if the passenger has not been picked up, $\Phi(i, s, a) = +20$; else, $\Phi(i, s, a) = -20$. These potential values are proposed based on our domain knowledge that 'when the passenger has not been picked up, Get should be encouraged; otherwise, it should be discouraged'. Note that here

we do not specify i, because when a is Get, its parent sub-task i can only be *Root*, according to the sub-task graph Figure 5.1(b) in Section 5.1. In the following, when we do not specify the parent sub-task i, then i can be any possible parent sub-task of a.

- When sub-task a is Put, if the passenger has been picked up, $\Phi(i, s, a) = +20$; else, $\Phi(i, s, a) = -20$. These potential values are proposed based on our domain knowledge that 'when the passenger has been picked up, Put should be encouraged; otherwise, it should be discouraged'.

- When sub-task a is *pick_up*, if the passenger has not been picked up and the taxi is in the passenger's position, $\Phi(i, s, a) = +10$; else, $\Phi(i, s, a) = -10$. These potential values are proposed to encourage the taxi to perform *pick_up* only when it is at the passenger's location and the passenger has not been picked, and to discourage the taxi to perform *pick_up* in other states.

- When sub-task a is *put_down*, if the passenger has been picked up and the taxi is in the passenger's destination position, $\Phi(i, s, a) = +10$; else, $\Phi(i, s, a) = -10$. These potential values are proposed to encourage the taxi to perform *put_down* only when it takes the passenger to the destination, and to discourage the taxi to perform *put_down* in other states.

- When sub-task a is Navigate(X), $\Phi(i, s) = +10$ if one of the following two requirements satisfies: **(1)** the passenger has not been picked up and X is the passenger's position; or **(2)** the passenger has been picked up and X is the destination. In all other states, $\Phi(i, s, a) = -10$. These potential values are proposed to encourage the taxi to navigate to the 'good' positions (when the passenger has not been picked up, the 'good' position is the passenger's square; otherwise, the 'good' position is the passenger's destination).

- When a is a navigation action, $\Phi(\text{Navigate(X)}, s, a) = +5$ if a reduces the Manhattan distance to the landmark X; otherwise, $\Phi(i, s, a) = -5$. These potential values are proposed to encourage the taxi to choose the navigation actions that reduce the Manhattan distance to the good positions. Note that these potential values are not perfect: when there is wall, the action that reduces the Manhattan distance may not be the best, and these potential values do not consider the stochastic nature of these navigation actions. We deliberately use these imperfect potential values, so as to see the robustness of PBRS-MAXQ-0.

Then we present the potential values used for LA-SARSA(0). Note that potential value functions used in LA only have two inputs: the current state s and the current action a. The potential values $\Phi(s, a)$ used in LA-SARSA(0) are as follows:

- When a is *pick_up*, $\Phi(s, a) = +10$ iff the passenger has not been picked up and the taxi is in the passenger's position; otherwise, $\Phi(s, a) = -10$. These potential values are proposed so as to encourage the taxi to perform *pick_up* only when it is in the passenger's position and the passenger has not been picked up; otherwise, *pick_up* is discouraged.

- When a is *put_down*, $\Phi(s, a) = +10$ iff the passenger has been picked up and the taxi is in the passenger's destination position; otherwise, $\Phi(s, a) = -10$. These potential values are proposed so as to encourage the taxi to perform *put_down* only when it is in the passenger's destination and the passenger has been picked up; otherwise, *put_down* is discouraged.

- When a is a navigation action (*north*, *south*, *east* or *west*), $\Phi(s, a) = +5$ iff:
  - The passenger has not been picked up, and performing a reduces the Manhattan distance to the passenger's position; or
  - The passenger has been picked up, and performing a reduces the Manhattan distance to the passenger's destination position.

  In all other situations, $\Phi(s, a) = -5$. Note that, similar to those potential values for primitive navigation actions that are used in PBRS-MAXQ-0, these potential values are not perfect, in the sense that they may mislead the taxi to perform some non-optimal actions. However, we use these imperfect potential values so as to ensure that the potential values used in PBRS-MAXQ-0 and LA-SARSA(0) are as similar as possible, so as to evaluate these two PBRS techniques effectiveness.

The performances of these four algorithms are given in Figure 5.2. We can see that in both flat and hierarchical RL, PBRS techniques can significantly improve the learning performance. Also, the performance of PBRS-MAXQ-0 is always significantly better than all other three RL algorithms throughout the 500-episode experiments (p-values are all smaller than 0.01). Note that PBRS-MAXQ-0 and LA-SARSA(0) use the same pieces of domain knowledge, but PBRS-MAXQ-0's performance is significantly better. The reason is that **(i)** the foundation RL algorithm

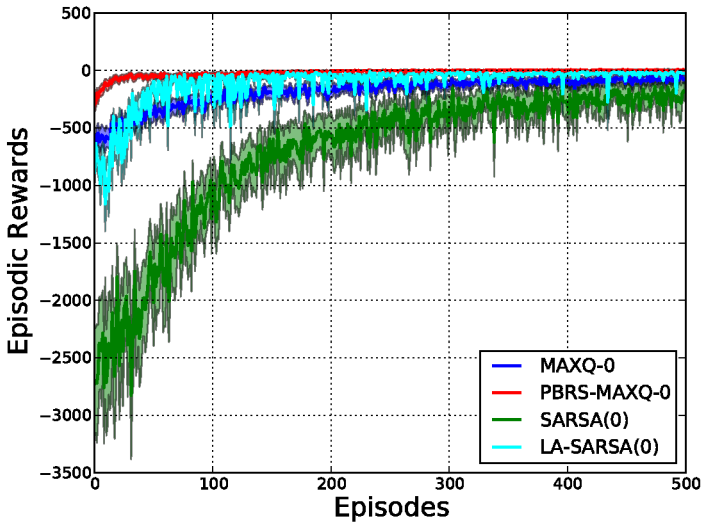


Figure 5.2: Performances in the Taxi problem. Lighter colour areas represent 95% confidence internals. Each experiment consists of 1000 episodes and all results shown are averaged over 100 independent experiments.

of PBRS-MAXQ-0: MAXQ-0, outperforms that of LA-SARSA(0): SARSA(0); and **(ii)** the domain knowledge we use also has a hierarchical structure, so by using PBRS-MAXQ-0, this knowledge can be more directly and effectively integrated into learning. Furthermore, by comparing PBRS-MAXQ-0 with other state-of-the-art MAXQ techniques, e.g. *R-MAXQ* [JS08], which integrates R-MAX [BT03] into MAXQ, we can see that under the same game settings, PBRS-MAXQ-0 converges faster than R-MAXQ and has comparable convergence speed with R-MAX (see Figure 5.3. Note that the game settings we used are the same as theirs).

5.2.2 Stochastic Wumpus World

We extend the Wumpus World we introduced in Section 2.2.3 by allowing the actions to have some probabilities to fail, so as to make this problem more challenging. In particular, we create a 10×10 grid world with 10 pits, 10 golds and 10 Wumpus. The map of this Wumpus World is shown in Figure 5.4. In the beginning of each episode, all Wumpus are alive and the agent is at (0,0). The agent's task is to collect all golds and arrive at the exit (at (9,9)) without being killed (eaten by a Wumpus or falling into a pit). Either successfully finish a task or being killed will end an episode.

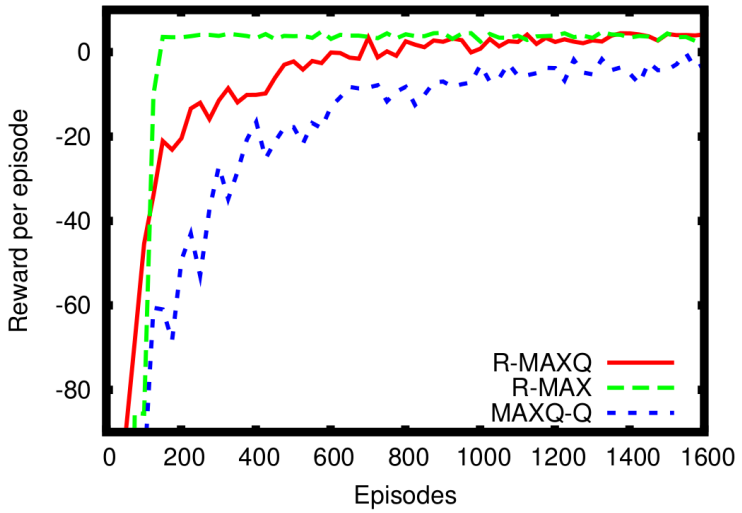


Figure 5.3: Performances of R-MAXQ and R-MAXQ in the Taxi Problem. This figure is reproduced from Figure 2(b) in [JS08] with permission.

Similarly to the previous experiment, we also implement four algorithms on this problem: MAXQ-0, SARSA(0), PBRS-MAXQ-0 and LA-SARSA(0). For each algorithm, we performed 100 independent experiments, each consisting of 5000 episodes. The state representation we use consists of the agent's current location, the exit's location, the number of uncollected gold and their locations, and three boolean variables indicating whether the agent senses glitter, breeze and stench, respectively, For all four learning algorithms, we let $\gamma = 1$ throughout our experiments; α and ϵ are set to be 1 in the first episode of each experiment, and α decreases at a rate of 0.9995. In MAXQ-0 and PBRS-MAXQ-0, we let ϵ decrease at a rate of 0.99, whereas in SARSA(0) and LA-SARSA(0), we let ϵ decrease at a rate of 0.95. These learning parameters are chosen based on our preliminary experimental results, so as to maximise each algorithm's average cumulative reward in each experiment.

As for the potential values used in PBRS-MAXQ-0, $\Phi(i, s, a)$ is given as follows:

- When a is *Collect*, if there still exists uncollected gold, $\Phi(i, s, a) = +100$; else, $\Phi(i, s, a) = -100$. These potential values are to encourage (discourage) the agent to perform *Collect* when there (does not) exists uncollected golds. Note that here we do not specify i, because when a is *Collect*, its parent sub-task i can only be *Root*, according to the sub-task graph Figure

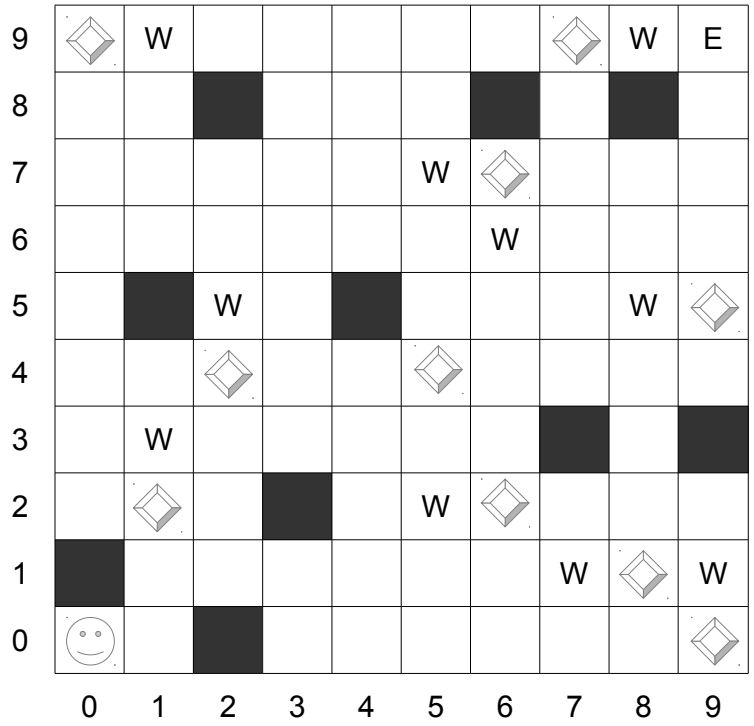


Figure 5.4: A Wumpus World. The smiling face stands for the agent, each black square stands for a pit, each 'W' stands for a Wumpus, the 'E' (in the top-right corner) stands for the exit, and each diamond symbol stands for a gold.

2.5 in Section 2.2.3. In the following, when we do not specify the parent sub-task i, then i can be any possible parent sub-task of a.

- When a is *Exit*, if all gold have been collected, $\Phi(i, s, a) = +100$, where i can only be *Root*; else, $\Phi(i, s, a) = -100$. These potential values are to encourage (discourage) the agent to perform *Exit* when (not) all golds are collected.

- When a is *pickup*, if the agent sees glitter in s, $\Phi(i, s, a) = +50$; else, $\Phi(i, s, a) = -50$. These potential values are to encourage (discourage) the agent to perform *pickup* when there is (no) gold in the current square.

- When a is *Hunt*, if the agent smells stench in s, $\Phi(i, s, a) = +10$; else, $\Phi(i, s, a) = -10$. These potential values are to encourage (discourage) the agent to perform *Hunt* when the agent does (not) feel stench in the current square. Note that the parent sub-task i can be either *Collect* or *Exit*.

- When a is *Navigate(x, y)*,
  - if i is *Collect* and (x, y) is the location of the farthest collected gold towards the exit, $\Phi(i, s, a) = +10$;
  - if i is *Exit* and (x, y) is the location of the exit, $\Phi(i, s, a) = +10$;
  - for all other situations, $\Phi(i, s, a) = -10$.

  These potential values are to encourage the agent to navigate to the farthest uncollected gold if there is any uncollected gold, or encourage the agent to navigate to the exit otherwise. Navigating to any other places are discouraged. Note that the parent sub-task i can be either *Collect* or *Exit*.

- When a is a primitive shoot action, if the agent feels stench in s, $\Phi(i, s, a) = +5$; else, $\Phi(i, s, a) = -5$. These potential values encourage the agent to shoot when it feels stench.

- When a is one of the primitive navigation actions, if the intended direction of a decreases the Manhattan distance to its parent sub-task's target position, $\Phi(i, s, a) = +5$; else, $\Phi(i, s, a) = -5$. These potential values encourage the agent to choose the path that minimising the Manhattan distance during navigation.

Note that the potential values provided here contain some non-perfect instructions: for example, we encourage the agent to collect the farthest gold towards the exit, which may lead the agent to being killed. However, generally, we believe that collecting the farthest gold can reduce the number of steps needed to successfully exit the world. We intentionally use these 'misleading' potential values so as to test the robustness of using PBRS in MAXQ.

As for the potential values $\Phi(s, a)$ (where s can be any state, and a can be any primitive action) used in LA-SARSA(0), we let them be the same with their corresponding potential values used in PBRS-MAXQ-0. For example, when a is *pickup*, if the agent sees glitter in s, $\Phi(s, a) = +50$; otherwise, $\Phi(s, a) = -50$. Note that when a is a navigation primitive action (e.g. *go_left*), we have $\Phi(s, a) = +5$ iff a reduces the Manhattan distance to the farthest gold (if there is

any uncollected gold) or to the exit (otherwise). By using these potential values for LA-SARSA(0), we attempt to ensure that the heuristics we use for both LA-SARSA(0) and PBRS-MAXQ-0 are the same, so as to compare the effectiveness of these two algorithms given the same heuristics.

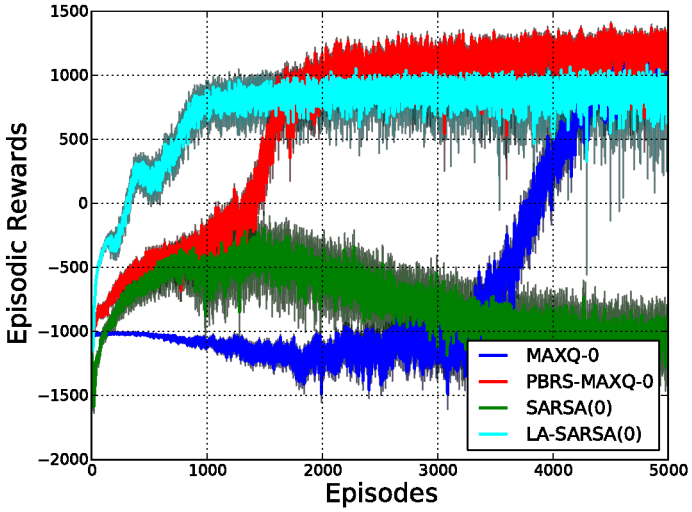


Figure 5.5: Performances of four RL algorithms in the stochastic Wumpus World game. Lighter colour areas represent 95% confidence intervals. All results are averaged over 1000 independent experiments, each consisting of 5000 episodes.

The learning curves of these four algorithms are presented in Figure 5.5. We can see that in the first 1500 episodes, LA-SARSA(0) significantly outperforms PBRS-MAXQ-0 and SARSA(0) significantly outperforms MAXQ-0. After 1500 episodes, PBRS-MAXQ-0 surpasses LA-SARSA(0), and SARSA(0)'s performance begins to drop. After 5000 episodes of learning, the last performance[4] of PBRS-MAXQ-0 is significantly better than that of all other algorithms (actually, the p-value between PBRS-MAXQ-0's and LA-SARSA(0)'s last performance is 0.04, while the p-values between PBRS-MAXQ-0's and the other two algorithms' last performances are both smaller than 0.01), followed by MAXQ-0, LA-SARSA(0) and standard SARSA(0). Note that the advantage of MAXQ-0 over LA-SARSA(0) is not significant (as the p-value is 0.35).

We can see that PBRS-augmented RL algorithms (PBRS-MAXQ-0 and LA-

[4] In this subsection, 'last performance' is referred to as the average rewards received in the last episode (i.e. the 5000th episode), averaged over 1000 experiments.

Table 5.1: Some statistics (mean $\pm$ standard error) during the learning. Percentage values (i.e. the last two rows) are computed by counting all episodes that are ended due to falling into pit (eaten by Wumpus) and dividing this number by the number of episodes (i.e. five million, obtained from 5000 episodes per experiments and 1000 episodes in total). Therefore, no standard errors are provided in the last two rows.

	MAXQ	PBRS-MAXQ	SARSA	LA-SARSA
Step/epi	611.61 ± 7.96	522.01 ± 4.98	1264.97 ± 7.06	794.45 ± 2.96
Rew/step	-0.98 ± 0.02	1.234 ± 0.02	-0.54 ± 0.01	0.93 ± 0.01
Kill/epi	4.30 ± 0.04	8.74 ± 0.01	8.94 ± 0.00	9.34 ± 0.00
Gold/epi	4.25 ± 0.04	7.91 ± 0.01	8.62 ± 0.01	9.27 ± 0.00
#Shoot/epi	5.653 ± 0.04	11.21 ± 0.01	24.23 ± 0.14	12.62 ± 0.01
#Pick/epi	$5.370.04$	9.91 ± 0.01	26.14 ± 0.17	12.32 ± 0.01
Fall%	0.54	0.36	0.31	0.15
Eaten%	0.25	0.01	0.06	0.03

SARSA(0)) significantly outperform their corresponding standard RL algorithms (MAXQ-0 and SARSA(0), respectively). Also, we see that SARSA(0) outperforms MAXQ-0 and LA-SARSA(0) outperforms PBRS-MAXQ-0 in the early learning stage, but after 5000 episodes of learning, the MAXQ-based algorithms outperform their SARSA(0)-based counterpart (indeed, the p-value between the last episode's averaged performances of MAXQ-0 and SARSA(0) is smaller than 0.01, while the p-value between the last episode's averaged performances of PBRS-MAXQ-0 and LA-SARSA(0) is 0.04).

To further investigate the reason behind the performance drop of SARSA(0), we compute some learning statistics during the learning, presented in Table 5.1. In this table, 'epi' and 'Rew' are shorthands for episode and rewards, respectively. From these data we can see that SARSA(0)-based algorithms perform significantly more *pickup* and shooting actions than their MAXQ-based counterparts (p-values are smaller than 0.01). This is because in the early learning stage, performing more *pickup* and shooting actions will result in a higher chance to collect golds and kill Wumpuses and, thus, will earn the agent more rewards; also, performing more *pickup* and shooting actions will also reduce the chance to be killed by falling into the pit or being eaten by a Wumpus. This explains why SARSA(0)-based agents perform better in the early learning stage. However, these rewards will reinforce the SARSA(0)-based learning agents to perform these actions more, which in turn leads to a drop in received rewards. This explains the performance drop

of SARSA(0) after 1500 episodes and why LA-SARSA(0) is not able to reach the same height of PBRS-MAXQ-0 after 5000 episodes of learning. MAXQ-based algorithms avoid this problem because of its hierarchical execution: to execute *pickup* or a shooting action in MAXQ, the learning agent needs to select several levels of sub-tasks, and in each level, there are some randomness in the selection (see Algorithm 4 and 11). As a result, although performing *pickup* and the shooting actions will lead to high rewards in the early stage, they still cannot be selected very frequently in the early stage.

5.3 Related Work

To the best of knowledge, no existing work integrates reward shaping techniques into any of the widely used Hierarchical Reinforcement Learning (HRL) algorithms, e.g. *Options* [SPS99], *Hierarchy of Abstract Machines* [PR97] or MAXQ. However, the *pseudo rewards* used in MAXQ-based algorithms are also for improving the sub-task-selection efficiency. Actually, Dietterich's original work that introduces MAXQ decomposition [Die98] has proposed this technique. Pseudo rewards are some extra rewards used in each sub-task, specifying which states are more desirable to be visited in this sub-task and which are not. In particular, a pseudo reward for a sub-task i is of the form $\sigma_i(s)$, where s is a state. If s is some state that people want the agent to visit more (less) in sub-task i, people can assign $\sigma_i(s)$ a positive (negative) real value. The pseudo rewards are only used in learning, not in MAXQ function values updating, namely pseudo rewards are only used in sub-task-selection, but they will not be directly integrated into the MAXQ functions. We highlight two important differences between the pseudo rewards and our PBRS-MAXQ-0:

- By using pseudo rewards, the original optimal policy is not guaranteed to be found [Die00, BM03]. This is because pseudo rewards are not directly fitted into the MAXQ functions; therefore, if the pseudo rewards are incorrect (e.g. an undesirable state is given a huge positive pseudo reward), the learning process cannot cancel out the negative effect brought by this pseudo reward, no matter for how long the learning proceeds. However, PBRS-MAXQ-0 is guaranteed to converge after a sufficiently long time of learning, because potential values are fitted into the MAXQ functions in a specific form (see Definition 5 earlier in this Chapter), and this integration form guarantees

that after enough long time of learning, potential values can cancel each other out. So, regardless of the correctness of the potential values, PBRS-MAXQ-0 can obtain the optimal policy. In other words, PBRS-MAXQ-0 is more robust than using pseudo rewards in MAXQ-0.

- Pseudo rewards only allow people to provide some 'indirect' instructions, while PBRS-MAXQ-0 allows people to give 'direct' instructions. We can see that pseudo rewards specify which states are more desirable and which are not, but they do not directly instruct the agent how to achieve the desirable states or how to avoid visiting the undesirable states. So pseudo rewards can only provide 'indirect' instructions. On the other hand, the potential values we use in PBRS-MAXQ-0, can directly let the agent know which child sub-task a is more desirable in state s within a's parent sub-task i, because they are of the form $\Phi(i, s, a)$.

5.4 Conclusion

In this chapter, we integrate two popular RL techniques: Potential-Based Reward Shaping (PBRS) and MAXQ. In particular, we prove that, by following our integration rules, arbitrary potential values can be integrated into the MAXQ decomposition without altering the original optimal performance. In addition, based on the integration rules, we propose the PBRS-MAXQ-0 algorithm, and prove that after learning for sufficiently long time, PBRS-MAXQ-0 is guaranteed to converge to the hierarchical optimal policy. Also, we implement MAXQ-0 and PBRS-MAXQ-0, and test their performances in two widely used RL testbeds: the Taxi problem and a stochastic Wumpus World problem. Empirical results suggest that PBRS can effectively accelerate the learning speed of MAXQ-0, even when the domain knowledge is imperfect.

The integration of PBRS and MAXQ provides an approach for using heuristics to improve the convergence speed of MAXQ-0. However, as we discussed in Chapter 1 and Chapter 2, proposing high-quality heuristics for RL algorithms is generally challenging, and this is even more so for MAXQ-0 algorithms, in which the potential values not only need to take into account the state and the child sub-task, but also need to consider which parent sub-task is invoking this child (see Section 5.1). Motivated by the success of using argumentation frameworks to provide heuristics for SARSA(λ) (Chapter 4), below in Chapter 6, we will integrate

the argumentation framework we proposed in Chapter 3 into PBRS-MAXQ-0, and use the resulting MAXQ-based AARL to tackle a real world problem.

6 MAXQ-Based AARL: An Empirical Evaluation

In Chapter 4, we presented SARSA(λ)-based AARL, which uses a classic Potential-Based Reward Shaping (PBRS) technique — look-ahead advice (LA) — to integrate argumentation-generated heuristics into SARSA(λ). Empirical results show that AARL can improve the performance of standard SARSA(λ), and these successful results motivated us to investigate whether AARL can be implemented on some more effective RL algorithms than SARSA(λ), e.g. the MAXQ-0 algorithm, and whether this integration can also improve the learning speed. The main purpose of this chapter is to investigate these problems.

In this chapter, based on the PBRS-MAXQ-0 algorithm we presented in Chapter 5, we propose *MAXQ-based AARL* and empirically test its performance in a novel *Residential Demand Response* (RDR) system called *Energy Usage Recommendation System* (EURS). This system learns the user's usage behaviour of home appliances and gives the user recommendations accordingly so as to help him save electricity expenses. We designed a simulated user based on a large amount of real data, so as to evaluate whether our EURS can give satisfactory recommendations to this simulated user.

The remainder of this chapter is organised as follows: we first present MAXQ-based AARL in Section 6.1, and then give background of RDR systems and present an overview of EURS in Section 6.2. After that, in Section 6.3, we motivate why we choose to use Hierarchical RL (HRL) algorithms to implement EURS, and, in Section 6.4, we describe how we use data collected from a real user to design a simulated user, with which EURS interacts. Afterwards, in Section 6.5, we give the detailed design of our prototype EURS, including how we model this problem as a MDP, and how we design the MAXQ hierarchical decomposition for EURS. In Section 6.6, based on our domain knowledge on this application, we propose some arguments and values, for both SARSA(λ)- and MAXQ-based AARL. At last, we present the experimental settings and performances in Section 6.7, review

related works in Section 6.8 and conclude this chapter in Section 6.9.

6.1 MAXQ-based AARL

As we have discussed in Section 3.3 and 3.5, an AARL amounts to the combination of three modules: the *AF* module, which is responsible for building argumentation frameworks *SCAF*, *VSCAF* and AF^{-}; the *Potential Generator* module, which assigns each action appropriate potential values based on heuristics generated by the AF module; and the *PBRS+RL* module, which integrates potential values into some RL algorithms via some PBRS technique. We have discussed in Section 3.3 and 3.5 that, in AARL, the argumentation framework for generating heuristics is independent of the underlying RL algorithm used. In other words, we can implement different versions of AARL, which share the same argumentation frameworks for generating heuristics, but use different techniques to integrate these heuristics into different RL algorithms. In MAXQ-based AARL, we use the PBRS-MAXQ-0 algorithm introduced in Chapter 5 to serve as the *PBRS+RL* module.

In the remainder of this section, we describe how to implement the other two modules and how to integrate all three modules together so as to construct the whole MAXQ-based AARL algorithm. Note that we only focus on single-agent MAXQ-based AARL in this chapter, because multi-agent learning based on MAXQ algorithms involves considerable problem-specific design issues (this will be discussed in greater detail later in Section 6.9). Since this is the first work in integrating MAXQ with argumentation, we believe that considering only the single agent learning case can help us more focus on the integration itself, not the complex implementation details of multi-agent MAXQ [GMM06].

Firstly, we look into the AF module in MAXQ-based AARL. In SARSA(λ)-based AARL, arguments can attack or be attacked by any other arguments that support different actions for the same agent. This is because, at any state, every action can be selected by the agent; thus, actions are 'competing' with one another. However, in each composite sub-task in MAXQ, only the children of this sub-task can be chosen and, therefore, only sub-tasks that share the same parent are competing with each other. Therefore, in MAXQ-based AARL, two arguments can be in the attack relation if and only if the sub-tasks they support share the same parent sub-task. As an illustration, consider the Taxi problem we introduced in Section 5.1 and its task graph in Figure 5.1(b). An argument supporting sub-task

Get and an argument supporting sub-task *south* should not be in the attack relation, because these two sub-tasks have different parents and, therefore, are not directly competing with one another during the sub-task-selection process in MAXQ. To ensure that only 'competitive' arguments can have attack relation, we propose *sub-task-based SCAF* and *VSCAF*, defined as follows (note, again, that we focus on the single-agent learning case here).

Definition 7. For a MAXQ-based learning agent, let its MAXQ decomposition be $H_M = \{M_0, \cdots, M_n\}$. Given this agent's observation *Sta* of a state s, a *Sta-specific cooperative argumentation framework (SCAF) for sub-task* i, where $i \in H_M$ is a composite sub-task, is an AF (Arg, Att) s.t.:

1. $Arg = \{A | con(A)$ is a child of sub-task i, A is applicable with respect to the learning agent $\}$.
2. $Att \subseteq Arg \times Arg$ s.t. $(A, B) \in Att$ iff $con(A) \neq con(B)$ and $A, B \in Arg$.

We refer to a *SCAF for sub-task* i as $SCAF^i$.

We can see that $SCAF^i$ is an AF such that only applicable arguments supporting children of i can be included in this AF, and two arguments can attack each other in $SCAF^i$ if and only if they support different children of i. So $SCAF^i$ represents and organises domain knowledge relating to the sub-task-selection in i. We then augment $SCAF^i$ with values, and the resulting framework is defined as follows:

Definition 8. For a MAXQ-based learning agent, let its MAXQ decomposition be $H_M = \{M_0, \cdots, M_n\}$. Given this agent's observation *Sta* of a state s, a *value-based Sta-specific cooperative argumentation framework for sub-task* i is a VAF $VSCAF^i = (SCAF^i = (Arg, Att), V, val, Valpref_{Sta})$ s.t.:

1. $SCAF^i$ is the Sta-specific cooperative argumentation framework for composite sub-task i;
2. V is a set (of *values*);
3. $val : Arg \rightarrow V$ is a function from *Arg* to *V*; and
4. $Valpref_{Sta}$ is a preorder over *V*, denoted as $\geq_v$.

We refer to $(Arg, Att, V, val, Valpref^{-}_{Sta})$ as the *VAF derived from VSCAF*, where $Valpref^{-}_{Sta}$ is a strict partial order $>_v$ such that $\forall v_1, v_2 \in V, v_1 >_v v_2$ if and only if $v_1 \geq_v v_2$ and $v_2 \not\geq_v v_1$, and we refer to $AF^{i-} = (Arg, Att^-)$ as the *simplified AF* derived from this VAF by using simplification rules of VAFs (see Section 2.1.3).

As an illustration, let us consider the Wumpus World game we presented in Section 2.2.3. Suppose we have the following arguments:

- **A1**: Collect IF there exist uncollected golds.
- **A2**: *pickup* IF sees glitter in the current square.
- **A3**: Hunt IF feels stench in the current square.

We can see that argument **A2** and **A3** can be contained in one $SCAF^i$ (where i can be any sub-task) and can attack one another, because the sub-task (action) they support — *pickup* and Hunt — share the same parent sub-task (see the task graph shown in Figure 2.5 in Chapter 2). But **A1** and **A2** cannot be contained in one $SCAF^i$, because the sub-task supported by these two arguments – Collect and *pickup* – do not share the same parent. Suppose in one state s, within Root, only Collect is applicable, then we can build $SCAF^{Root} = (Arg^{Root}, Att^{Root})$ where $Arg^{Root} = \{\mathbf{A1}\}$, and $Att^{Root} = \emptyset$. Also in state s, if both **A2** and **A3** are applicable, then $SCAF^{Collect} = (Arg^{Collect}, Att^{Collect})$, where $Arg^{Collect} = \{\mathbf{A2}, \mathbf{A3}\}$, and $Att^{Collect} = \{(\mathbf{A2}, \mathbf{A3}), (\mathbf{A3}, \mathbf{A2})\}$. Also, suppose that we have a value set V consisting of only two values:

- **V1**: collect gold so as to receive more rewards,
- **V2**: kill a Wumpus so as to receive more rewards,

and we have *val* contains both $\mathbf{A2} \mapsto \mathbf{V1}$ and $\mathbf{A3} \mapsto \mathbf{V2}$. In addition, in *Valpref*, we have $\mathbf{V1} \geq_v \mathbf{V2}$. Then we can build $VSCAF^{Collect} = (SCAF^{Collect}, V, val, Valpref)$, and we can easily obtain that the simplified AF derived from $VSCAF^{Collect}$ is $AF^{Collect-} = (Arg^{Collect}, Att^{Collect-})$, where $Att^{Collect-} = \{\mathbf{A2}, \mathbf{A3}\}$. From this illustration, we can see that argumentation frameworks $SCAF^{Collect}$, $VSCAF^{Collect}$ and $AF^{Collect-}$ exclude 'useless' information (arguments) for selecting children of Collect (in this illustrative example, **A1** is useless in children-selection within Collect) and only include the 'relevant' arguments (i.e. **A2** and **A3**).

Also note that AF^{i-} (i can be any composite sub-task) 'inherits' all properties of AF^-. For example, if AF^{i-} has non-empty grounded or preferred extension(s), then at most one child of sub-task i is recommended. This can be proved similarly to Theorem 1, so we skip its proof here.

Secondly, we discuss the *Potential Generator* module. Recall that in SARSA(λ)-based AARL, we require the domain expert to give a upfront real number $c > 0$,

and we give all recommended actions this c value as its potential value (see Section 4.1). In PBRS-MAXQ-0, as suggested by Theorem 3 in Section 5.1.2 (see remark right after the theorem), the ideal value for $\Phi(i, s, a)$ is $C^{\pi^*}(i, s, a)$, where i can be any composite sub-task, s can be any state and a can be any child of i. Note that there can be a big difference between C values for different subtask-state-child pairs (i, s, a): for example, in the illustration we give in Section 2.2.3, $C^{\pi^*}(\textit{Root}, s_1, \textit{Hunt}) = 598$ (this value is the cumulative reward received after performing *Hunt* in state s_1 within *Root*), and $C^{\pi^*}(\textit{Hunt}, s_1, \textit{shoot_right}) = 0$. So the ideal value for $\Phi(\textit{Root}, s_1, \textit{Hunt})$ is 598, and for $\Phi(\textit{Hunt}, s_1, \textit{shoot_right})$ is 0. We can therefore see that giving these two potential values the same value is inappropriate. Hence, in MAXQ-based AARL, we require the domain expert to provide k potential values: $c_1, \cdots, c_k$, where k is the number of composite sub-tasks in the MAXQ decomposition. The idea is that within each composite sub-task i, we give the same potential value to all recommended children of i. Still, this is a simplified setting: an ideal setting is to ask to the domain expert to give each subtask-state-child pair (i, s, a) a specific potential value, but this will require too much human engineering; more importantly, in complex applications, the domain experts may not be able to provide so fine-grained domain knowledge. In both our experiments in Section 5.2, we use this method to give potential values, and empirical results show that the performance is good. We believe this method to give potential values in MAXQ is practical.

Given our discussions on the AF module and Potential Generator module above, we use one function, **getHierarchicalPotential**, to implement these two modules' functionality. Pseudo code of this function is presented in Algorithm 12. We give a detailed explanation of Algorithm 12 as follows: if the input parent sub-task p terminates in input state s, by definition of the potential values (see Definition 4 in Section 5.1.1), $\Phi(p, s, d) = 0$ (line 4 in Algorithm 12); otherwise, the function first obtains the upfront domain knowledge provided by domain expert (line 6). Given this knowledge, sub-task-based argumentation frameworks $SCAF^p$ and $VSCAF^p$ can be built, and the simplified AF^{p-} can be derived (line 7). By computing the set E of required type of extensions for AF^{p-} (line 8) and obtaining the recommended sub-task a_{rec} from the E (line 9, **getRecActFromExt** has been presented in 3.2.4), we can obtain the potential value for child sub-task d: if d equals a_{rec}, $\Phi(p, s, d) = c_p$ (line 11); otherwise, $\Phi(p, s, d) = 0$ (line 13).

In the remainder of this chapter, we use both MAXQ-based and SARSA(λ)-based AARL algorithms to implement *Energy Usage Recommendation System*

Algorithm 12 Function **getHierarchicalPotential** for MAXQ-based AARL

```
1: /*Function for computing potential values*/
2: getHierarchicalPotential(Composite Sub-task p, State s, Sub-task d)
3: if p terminates in state s then
4:    return 0
5: else
6:    Obtain candidate argument set Arg*, value set V, value promotion relation
      val, value ranking Valpref, the argumentation extension type Type, and the
      potential value given to recommended sub-tasks of p: c_p ∈ ℝ, c_p > 0
7:    Build SCAF^p, VSCAF^p, and derive AF^{p−}
8:    E := getExtension(AF^{p−}, Type)
9:    a_rec := getRecActFromExt(E)
10:   if a_rec is not null and d = a_rec then
11:      return c_p
12:   else
13:      return 0
14:   end if
15: end if
```

(EURS), and compare their performances under multiple settings.

6.2 The Energy Usage Recommendation System

Reducing energy consumptions as well as CO_2 emissions have been recognised as an important problem faced by the world, and the demand side management, also known as *demand response* (DR), is regarded as an important technique in tackling this problem [LCL11, OLGM10]. In a nutshell, DR amounts to reschedule users' energy demand so as to reduce the peak load of the whole electricity network as well as to save money for the energy users. DR for large energy users has received intensive investigations and has been implemented in many areas (a comprehensive review of this field can be found in e.g. [AES08]), and an increasing trend of research on *residential DR* (RDR) is also witnessed in recent years [OLGM10].

We implement a RDR system which is able to read and record the power readings of each appliance[1] in a household and, based on these data, give the user advice every thirty minutes, suggesting whether or not to use some selected ap-

[1] We believe this assumption is achievable, because of the increasing popularity of smart meters, the increasing numbers and types of off-the-shelf sensors [KK14], and the rapid development of *non-intrusive load monitoring*, which can derive appliance-level consumptions from the aggregated power consumption [PGWR12].

pliances in the ensuing thirty minutes.[2] The user reads the advice and decides whether to change the original usage plan or not. Note that the user does not need to explicitly reply to the system: the system can detect whether the user follows its advice by reading the power consumption of each appliance. We call this system *Energy Usage Recommendation System* (EURS).

Our contribution in this experiment is threefold:

- we propose a novel RDR system called EURS, and design a real-data-based simulated user to interact with this system;
- we identify the hierarchical structure in energy usage and accordingly propose the first MAXQ-based algorithm for RDR (see next subsection); and
- we novelly use PBRS in designing a RDR system and illustrate that using domain knowledge can effectively improve the performances of both SARSA- and MAXQ-based RDR systems.

6.3 Why Use MAXQ-Based AARL

Predicting the user's usage behaviour plays an important role in designing EURS, because the user is unlikely to accept advice that is very different from the user's planned usage. However, it has been reported that Q-Learning cannot effectively predict the user's usage behaviour, especially when the prediction interval is larger than 30 minutes [GQS10]. We will show that our MAXQ-based EURS is able to give more acceptable advice.

It has been recognised that people's energy usage in the same *type of days* is very similar [TMTT$^+$13], and the reason is that in each day type, certain *types of activities* are performed routinely. For example, each Saturday, my energy usage is very similar, because I usually clean my room in the morning, cook at noon, go out in the afternoon and do some entertainment activities in the evening. Actually, all days that share similar appliances usages can be categorised into one type. For instance, if my usage in Tuesdays is similar to that in Saturdays, they can all categorised as one day type. There exist some algorithms in learning and identifying the type of days, e.g. [TMTT$^+$13]. Each type of activity, in turn, involves using certain appliances: cleaning, for instance, involves using the vacuum cleaner. We

[2] In this system, to discretise time, we divide each day into forty-eight equal-length *time slots*, each covers thirty minutes.

can see that the type of days, activities and appliances naturally form a top-down hierarchical relation, and to predict a user's appliance-level usage, we can first identify the day type and activity type, so as to reduce the number of appliances that may be used and increase the prediction accuracy. The motivation for using MAXQ to design the EURS is to take full use of this hierarchical structure. In this work, for simplicity, we only focus on one type of day: the days when the user uses some selected appliances once a day, and our MAXQ hierarchical decomposition will therefore only consider the activity-appliance hierarchy. More details of the type of day and the hierarchy will be given below in Section 6.4 and 6.5, respectively.

Besides using a hierarchical structure to improve the prediction accuracy, using some prior knowledge about a user's usage pattern can also potentially improve the performance of EURS. Considerable domain knowledge is available about when a certain kind of activity is likely/unlikely to be performed: some of this knowledge is generic (e.g., people usually do not clean their houses at midnight), while some others is personal (e.g., by observing a user's usage for some time, the system can easily identify when he usually cleans his house). However, this knowledge can contain conflicts: for example, some pieces of domain knowledge may advise a user to use a certain appliance at time t because this user usually uses it at time t, while some other pieces of domain knowledge may advise this user to use this appliance later than t, because postponing the usage can save quite some money. And we can see that different recommendations are based on different *values*: for example, still consider the earlier illustrative example, the former piece is based on the value 'avoid disrupting the user', while the latter piece is based on the value 'save money'. Overall, we believe that value-based argumentation framework is a suitable abstraction to represent domain knowledge in this setting so as to obtain high-quality heuristics. And to incorporate these heuristics into RL-based EURS, we use both MAXQ- and SARSA-based AARL.

6.4 A Real-Data-Based Simulated User

Recall that the goal of EURS is to learn the user's usage pattern of some appliances and accordingly give the user recommendations, so as to save money for the user as well as minimising disruption[3]'. So we can see that in this learning problem, the user is the 'environment', and RL-based EURS learns the best actions (in this

[3] 'Disruption' here means the advised action is different from the user's existing planned behaviour.

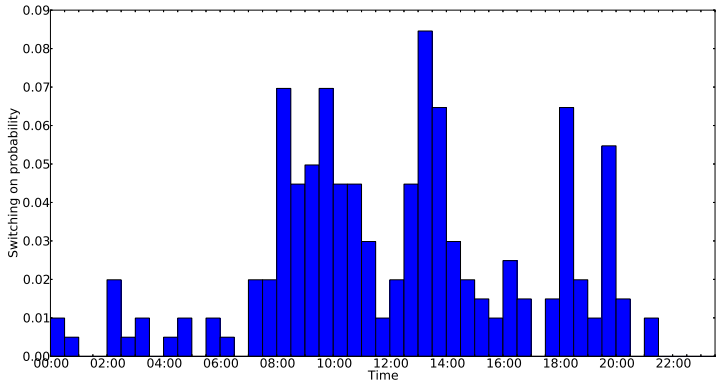


Figure 6.1: The probability distribution of the switching on time of the washing machine. This distribution is obtained by using the user's usage data in 51 selected days.

case, the actions are recommendations) in each time slot of a day by interacting with this environment. Because many episodes of learning may be needed before a good policy can be found (this can be seen from, for example, experiment on the Keepaway and Takeaway games in Section 4.2, experiments on the Taxi problem in Section 5.2.1, and experiments on the stochastic Wumpus World game in Section 5.2.2), it is unrealistic to make the system interact with real users for very long time. As a result, it is essential to design a simulated user to train this system before it is deployed in real households.

To make the simulated user as lifelike as possible, we use a big amount of data collected from a real user to design the simulated user. We use the longest dataset in the *UK-DALE* database [KK14], which contains power readings of 54 appliances over 470 days in one household. We only consider six appliances: the vacuum cleaner (hoover), dishwasher, washing machine, TV, kitchen electronics[4] and PC[5]. We consider these appliances because they consume considerable energy (average ≥ 0.1 kWh/day) and have quite flexible working times (unlike, e.g., the WiFi router, which typically works 24/7). We only consider one specific type of day, in which the simulated user uses these six appliances once a day. In the dataset, 54 days are of this type, and most of them are weekend days (51).

[4] These electronics include a kettle, two toasters, two food mixers and a kitchen aid. According to the description of the UK-DALE database, the data of these electronics are collected by using one sensor, so they are treated as one appliance in our work.

[5] This PC is mainly used for entertaining purposes.

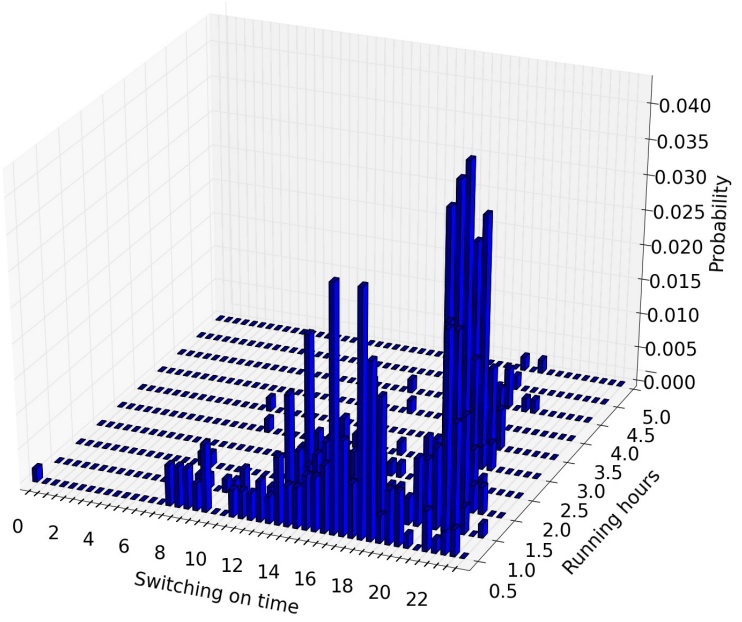


Figure 6.2: The probability distribution of the switching on time and working time of the TV set.

Note that we need to simulate a user in the following two aspects: **(1)** how the user plans his usage of each appliance, namely the original usage pattern of this user, and **(2)** how he responds to different recommendations in different situations. To simulate the first aspect, we assume that, in this selected type of days, the user has a fixed probability to use each appliance in each time slot, and this probability can be simply computed from the user's existing usage data. For example, the probability distribution of the user turning on the washing machine in different time slots is shown in Figure 6.1, and the probability distribution of the user switching on the TV in different time slots and using it for different lengths of time is shown in Figure 6.2. Note that Figure 6.1 is two-dimensional because we do not need to consider the working time of the washing machine: we observe from the data set that, in each usage, the washing machine works for three time slots and automatically turns off. For example, in Figure 6.1, the right-most bar represents that the probability of switching on the washing machine between 21:00 and 21:30 (and using it for the three time slots) is 0.01. Figure 6.2 is, however, three-dimensional, because the working time of each usage is also needed to be taken into account when computing the user's usage pattern. For example, the

bottom left bar in Figure 6.2 represents that the probability of switching on the TV between 00:00 and 00:30 and using it for thirty minutes (one time slot) is 0.002. Three appliances switch off automatically after working for fixed numbers of time slots: the washing machine, dish washer and kitchen electronics. They work for three, two and one time slot(s) before they are turned off, respectively.

More specifically, for each appliance $i, i = 1, \cdots, 6$, we compute $P_i(t_s, t_e)$, the probability of using appliance i from time t_s to time t_e,[6] by using data collected from the specific type of days in the dataset. In the beginning of each day, the simulated user uses these probability distributions to plan when to switch on and off each appliance. For example, given the distribution of the washing machine shown in Figure 6.1, the simulated user has 1% probability to switch on the washing machine between 00:00 and 00:30, and has 8.5% probability to use turn it on between 13:00 and 13:30. For simplicity, we assume that each day, any appliance's usage is independent of other appliance usage, and is also independent of the usage on other days.

As for point **(2)**, we will describe how the user responds to recommendations below in Section 6.5, because this is closely related to the transition function and the rewards function of this problem.

6.5 Model the Problem as a MDP

In Section 6.4 above, we have pointed out that the user is the 'environment' of this learning problem, and the task of EURS is to learn how the environment responds to different recommendations in different situations, so as to give the most 'satisfactory' recommendations to the user in different situations. Given this understanding, we model the simulated user as a MDP, and its four components are as follows (i ranges over all six appliances, i.e. $i = 1, \cdots, 6$):

- **States.** Each day is a learning episode, which is divided into forty-eight equal-length time slots, each covering 30 minutes. Each state is a vector, consisting of the current time index, the status of all appliances (on/off), and when each appliance has been started using in today (if an appliance has not been used until the current time slot of the day, this value is -1). In the remainder of this chapter, unless stated otherwise, we use s to represent a state, $s.time$ is the index of the current time slot in state s, and $s.t_i$ is the

[6]Note that both t_s and t_e here are indies of time slots.

time when appliance i is started used today.

- **Actions**. Each primitive action is a recommendation to the user. Note that, since we consider six appliances, each primitive action consists of six sub-recommendations, or *atomic actions*, one for each appliance. We design five kinds of sub-recommendations for each appliance i (s is the current state, l is the most likely number of time slots the user will use appliance i for; l can be obtained easily from P_i probability distributions):
  - *switch_on*: advises the user to turn on appliance i in the current time slot and keep using it for at least another time slot. To be more specific, this sub-recommendation recommends the user to use appliance i from time $s.time$ to time $s.time+l$. Note that this sub-recommendation can only be recommended to appliance i if i is currently off; when i is on, this atomic action is not available because it is 'useless'.
  - *switch_off*: advises the user to turn off appliance i in the ensuing time slot. To be more specific, this sub-recommendation recommends the user to use appliance i from $s.t_i$ to the current time $s.time$ (note that $s.t_i$ is the time when the user actually starts using appliance i). Note that this sub-recommendation can only be recommended to an appliance that has been turned on and has not finished using; otherwise, it is 'useless'.
  - *keep_on*: advises the user to keep using appliance i for at least another time slot; i.e. recommends the user to use appliance i from $s.t_i$ to the next time slot $s.time+1$. Note that this sub-recommendation can only be recommended to appliance i if i is currently on.
  - *keep_off*: when $s.t_i = t$, i.e. the user plans to use appliance i now, it advises the user to postpone the usage of i for one time slot, i.e. it recommends the user to use appliance i from $s.time + 1$ to $s.time + 1 + l$; otherwise, it simply recommends the user not to use appliance i in the current time slot. Note that this sub-recommendation can only be recommended to appliance i if i is currently off.
  - *on_and_off*: advises the user to turn on and finish using i in the ensuing time slot, i.e. it recommends the user to use appliance i only during the current time $s.time$. Note that this sub-recommendation can only be recommended to appliance i if i is currently off.

For example, a vector $a_1 = (wm$: *keep_off*, hv : *switch_on*, pc : *keep_off*, tv : *keep_on*, kt : *switch_off*, ds : *switch_on*$)$ is a primitive action, which recommends the user to keep off the washing machine (wm), switch on the hoover (hv), keep off the PC (pc), keep on the TV (tv), switch off the kitchen electronics (kt) and switch on the dishwasher (ds). In the remainder of this chapter, we use *recommendation* and *primitive action* interchangeably, and use *sub-recommendation* and *atomic action* interchangeably. Also note that, as the user does not need to turn off dishwasher, kitchen electronics and washing machine (these appliances automatically turn off after they finish their work), *switch_off* and *on_and_off* are not advised to these three appliances.

- **Rewards.** Note that RL-based EURS uses the rewards to evaluate to what extent the user likes or dislikes a recommendation in a state. Because an action is a vector consisting of six sub-recommendations, the reward returned by the user is the sum of six *sub-rewards*, one for each sub-recommendation.

  To be more specific, if the current time slot is not the last time slot of the day, reward $R = \sum_{i=1}^{6} R_i$, where R_i is the sub-reward for appliance i. Sub-reward $R_i = 0$ iff the user accepts the ith sub-recommendation in the recommendation, i.e. the ith sub-recommendation is the same as the action the user actually performs on the corresponding appliance; otherwise, $R_i = pun$ (pun for punishment), where $pun \in \mathbb{R}, pun \leq 0$ is a constant number, and we will specify its value later in our experimental settings (Section 6.7). When the current time slot is the last time slot of the current day, $R = \sum_{i=1}^{6}(R_i + E_i^o - E_i^r)$, where E_i^o and E_i^r are the whole-day expense for appliance i based on the user's original planned usage and his real usage, respectively. Note that E_i^o and E_i^r are computed in the end of each day, when the real usage of each appliances is known. Since the power of all appliances are known a priori (can be read from the original data set), the computation of these two values are therefore quite straightforward and we thus omit them here.

  To understand why we give 0 when the system's advice is accepted, consider a primitive action that is exactly the same as the user's plan. This advice should not receive a positive reward, because it does not save any money, nor should it receive a negative reward, because it does not cause any loss. From the rewarding rules above, we can see that the system only receives positive rewards when it helps the user save money, i.e. $E_i^r < E_i^o$, and by tuning the

rejection punishment pun, we can adjust how much weight the simulated user puts on 'minimising disruption For example, when $pun = 0$, the simulated user does not mind to be disrupted and only cares about money-saving, while when $pun = -\infty$, the user only wants to avoid disruption and does not care about the expense.

- **Transition function**. In this problem, the transition function describes how the user adjusts his original usage after reading advice. Consider a situation where the user's original plan is to use appliance i from t_s to t_e, and the system's sub-recommendation for i is to use it from t_1 to t_2. We make the user accepts this sub-recommendation iff $P_i(t_1, t_2) - P_i(t_s, t_e) \geq Th_1$ (Th for threshold), where $Th_1 \in \mathbb{R}, -1 \leq Th_1 \leq 1$ (we will specify the value of Th_1 later in Section 6.7). We can see that Th_1 is a threshold value to control how willing the simulated user is to change its original planned usage: the bigger the Th_1, the less willing the user is to change his original usage. For example, when $Th_1 > 0$, the user accepts a suggestion iff the advised actions are, according to the user's existing habits, performed more often than the original planned actions.

Now we design the MAXQ hierarchy. Recall that we only consider the activity-appliance hierarchy (because we focus on one specific type of day, see Section 6.4), so the composite sub-tasks in our MAXQ hierarchy correspond to activities, and the primitive actions correspond to appliance usages. Also note that in the execution of MAXQ, at each layer, only one sub-task can be selected. In other words, all activities are mutually exclusive, because at each time only one of them can be recommended to the user. In order to design mutually exclusive activities (sub-tasks), we first divide all six appliances into two groups: *exclusive* appliances, including hoover, kitchen electronics and PC, and *compatible* appliances, including all the other appliances. This division is based on our domain knowledge that, at each time slot, the user is unlikely to use more than one exclusive appliance, but may use one or more compatible appliances. For example, we think it is unlikely for a user to use the hoover and kitchen electronics at the same time, but it is common to use the hoover and dishwasher simultaneously.

Given the above division of appliances, we design five activities (composite sub-tasks): *Cleaning*, *Cooking*, *Relaxing*, *UseCompatible* and *KeepAllOff*. In *Cleaning*, *Cooking* and *Relaxing*, only one exclusive appliance can be advised to be used, namely the hoover, kitchen electronics and PC, respectively, while all compatible

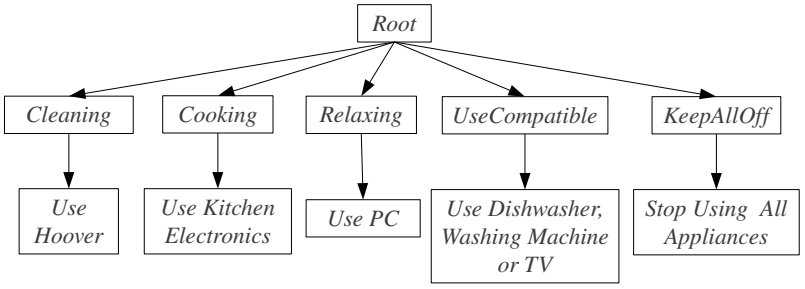


Figure 6.3: The task graph for the energy adviser system.

appliances can also be advised to be used. In *UseCompatible*, one or more compatible appliances can be advised to be used, while no exclusive appliances can be advised to be used. In *KeepAllOff*, all appliances are advised to stop working (i.e. *switch_off* or *keep_off*). The MAXQ graph is illustrated in Fig. 6.3. Each bottom-layer box represents a collection of primitive actions: for example, box *Use Hoover* includes all primitive actions that advise the user to use the hoover. So action $a_1 = (wm : \textit{keep_off}, hv : \textit{switch_on}, pc : \textit{keep_off}, tv : \textit{keep_on}, kt : \textit{switch_off}, ds : \textit{switch_on})$ we mentioned above is in the box *Use Hoover*, whereas action $a_2 = (wm : \textit{keep_off}, hv : \textit{switch_on}, pc : \textit{keep_off}, tv : \textit{keep_on}, kt : \textit{switch_on}, dw : \textit{switch_on})$ is not included in this box. Actually, action a_2 is not included in any bottom layer box in Figure 6.3, because we assume that two exclusive appliances — in this case, the hoover and kitchen electronics — should not be used by the user in the same time slot and, therefore, we do not allow MAXQ-based EURS to give this recommendation. However, in SARSA-based EURS, we still allow action a_2 to be recommended to the user. By doing this, we shrink the action space of MAXQ-based EURS, at the risk of being not able to obtain the optimal policy: if the user indeed performs this action, our MAXQ-based EURS will never be able to give the user this recommendation.

As for the termination predicate, *UseCompatible*, *KeepAllOff* and all primitive actions terminate immediately, *Root* terminates when a day ends, and the other three composite sub-tasks terminate when their corresponding exclusive appliance finished working.

6.6 Arguments for EURS

In this section, we introduce the arguments and values used in both SARSA(0)-based and MAXQ-based AARL for the EURS. In Section 6.6.1, we will introduce the arguments supporting recommendations (primitive actions) and their corresponding values, while in Section 6.6.2, we will introduce arguments and values for composite sub-tasks in MAXQ based AARL.

Figure 6.4 summarises all options considered in a AARL-based implementation of EURS. We use both preferred extensions and grounded extensions (see Section 2.1.2) in implementing AARL algorithms, and in each semantics, we use two different value ranking approaches: one is to have all values equally ranked, and the other is to rank values according to their relative importance, as advised by the domain expert. In particular, when using preferred semantics, we employ two methods in obtaining recommendations from preferred extensions: one method is to randomly select a preferred extension and recommend actions that are supported by arguments in this selected preferred extension. This is the standard method we discussed in Section 3.2.4. The other method is to recommend all actions that are supported by a preferred argument (an argument is preferred iff it is contained in at least one preferred extension, see Section 2.1.2). Compared with the standard approach, this method is somewhat counter-intuitive, since we have shown in Section 3.2.2 that actions recommended by different preferred extensions may be conflicting with one another. In particular, in the single-agent learning case, the same agent may be recommended to perform different actions by different preferred extensions. However, by recommending all preferred-argument-recommended actions to the learning agent, the learning agent can know that these recommended actions are 'equally good', and are better than the other non-recommended actions. As a result, the learning agent are encouraged to explore the recommended actions more.

6.6.1 Arguments Supporting Primitive Actions and Their Values

We propose a systemic method to propose arguments for so many primitive actions, by taking advantage of our assumption that 'the usage of each appliance is independent with usages of other appliances' (this assumption is made in Section 6.5). To be more specific, based on this assumption, when the system evaluates the sub-recommendation for one appliance, it does not need to consider sub-recommendations for other appliance: for example, when we consider whether to

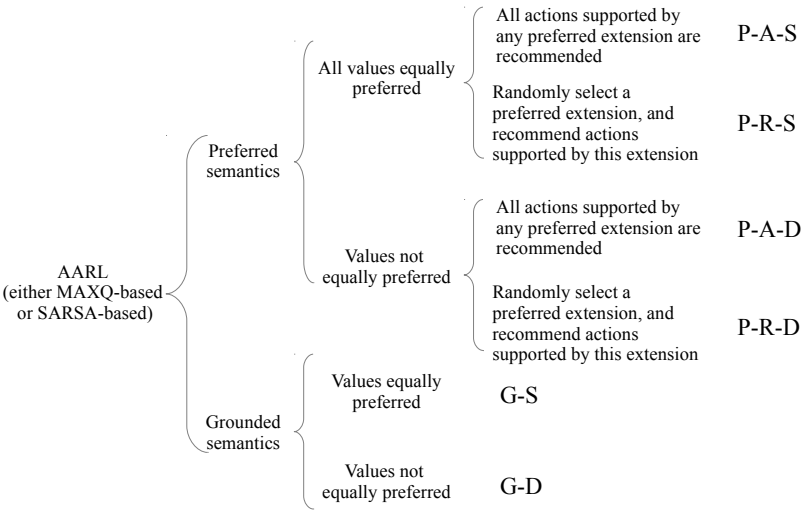


Figure 6.4: Settings we use to implement AARL for EURS. Strings in the right most column are shorthands for setting on its left. For example, 'G-D' is a shorthand for the setting 'grounded extension is used, and values are not equally preferred'.

recommend the user to use the TV in a state s, we do not need to consider whether we should also recommend him to use the dishwasher, because their usages are independent. As a result, when we ask domain expert to give heuristics about whether a recommendation is good or not, we can just ask the domain expert to evaluate the goodness of each sub-recommendation in this recommendation, instead of evaluating the recommendation as a whole.

As an example, consider action $a_1 = (wm : \textit{keep_off}, hv : \textit{switch_on}, pc : \textit{keep_off}, tv : \textit{keep_on}, kt : \textit{switch_off}, dw : \textit{switch_on})$, and suppose we attempt to give some arguments supporting this action. One way to do this is to think about in which states this recommendation is good; this is obviously difficult, because the domain expert has to take into account all sub-recommendations in the recommendation to find the good states to perform this action. Furthermore, since the number of possible actions is big,[7] to consider the good states for each action hence needs

[7] As presented in Section 6.5, each action of EURS is a recommendation consisting of six sub-recommendations. Since we have five kinds of atomic actions, the total number of available actions in each state is quite big: there are $5^6 = 15625$ possible sub-recommendation combinations, but not all of them are legal in this problem: for example, suppose a state in which all appliances are off and have not been used today, then atomic actions *switch_off* and *keep_on* are not available in this state for each appliance, because these two sub-recommendations are 'useless'. Given the legal condition of each sub-recommendation, we can easily see that the TV, PC

lot of human efforts. An alternative way is to ask the domain expert to provide, for each sub-recommendation, in which states this sub-recommendation is good to be recommended to the user; and then by finding the intersection of these good states for each sub-recommendation, the good states for the whole recommendation can be found.

Take a_1 for instance: instead of asking the domain expert to provide in which states this action is good to perform, we can ask the expert to provide in which states it is good to recommend *keep_off* to the washing machine, and in which states it is good to recommend *switch_on* to the hoover, etc. Then, given these good states for each sub-recommendation, the good states for performing action a_1 can be simply obtained by computing the intersection of all these 'good states'.

Now we propose the good states for each sub-recommendation. Recall that we want all sub-recommendations to not only be close to the user's existing usage habit (i.e. to minimise disruption), but also be able save the user money. Given these goals, we list the good states for each primitive action as follows: (i ranges over all appliances, and $Price(t) \in \mathbb{R}$ is the electricity price in time slot t. Th_2 and Th_3 are threshold values, where $Th_2 \in \mathbb{R}, 0 \leq Th_2 \leq 1$ and $Th_3 \in \mathbb{R}, Th_3 > 0$. Their detailed values will be given below in Section 6.7):

1. *switch_on*(i) is good to be recommended to the user in state s if:
   a) the probability of switching on i at $s.time$ and keeping using i for more than one time slot is larger than Th_2 and *switch_on* is available for i in s; or
   b) $Price(s.time) < Th_3$ p/kWh and $Price(s.time + 1) < Th_3$ p/kWh and *switch_on* is available for i in s.
2. *on_and_off*(i) is good to be recommended to the user in state s if:
   a) the probability of switching on i at $s.time$ and use it for only one time slot is larger than Th_2 and *on_and_off* is available for i in s; or
   b) $Price(s.time) < Th_3$ p/kWh and *on_and_off* is available for i in s.
3. *keep_off*(i) is good to be recommended to the user in state s if *keep_off* is available for i in s.

---

and kitchen electronics have three sub-recommendations available, while the other three appliances have two appliances available (because they will turn off automatically, so *on_and_off* is not available for them) in any state; therefore, the total number of legal actions is $3^3 \times 2^3 = 216$.

4. *switch_off*(i) is good to be recommended to the user in state s if *switch_off* is available for i in s.

5. *keep_on*(i) is good to be recommended to the user in states s if the probability of using i from $s.t_i$ to $s.time + 1$ is larger than Th_2 and *keep_on* is available for i in s.

In the remainder of this chapter, we refer to the set of good states for five sub-recommendations as $GS_{switch_on}, GS_{switch_off}, GS_{on_and_off}, GS_{keep_on}$ and GS_{keep_off}, respectively. Note that the above good states are proposed based upon two assumptions: **(1)** we know the probability distribution function P_i (see Section 6.4) for each appliance i a priori; and **(2)** we know the electricity price rate at each time of the day a priori. We think that both these assumptions can be satisfied in real applications: as for assumption **(1)**, we may require the user not to use the system during the first few days after it is installed, so that, during this period, the system can collect some preliminary data about the user's usage and obtain P_i accordingly; as for assumption **(2)**, many tariffs used in our daily lives provide the price rates a priori, for example, most tariffs provided by EDF[8], one of the biggest electricity provider in the UK, have their price rates available online. Also, even if the electricity rates are unknown a priori, they can be predicted by some existing techniques, e.g. [CENC03]. Note that the probabilities in the descriptions of these good states can be computed by the usage probability distribution P_i (see Section 6.4). For example, in the first item in the description of GS_{switch_on}, 'the probability of switching on i at $s.time$ and keep using i for more than one time slot' can be computed as follows: $\sum_{t=1}^{48-s.time} P_i(s.time, s.time + t)$.

We describe the domain knowledge upon which we propose the above five sets of good states:

- GS_{switch_on} is proposed based on the domain knowledge that if the user usually performs *switch_on* on appliance i at the current time slot and uses it for more than one time slot, then sub-recommendation *switch_on*(i) should recommended to the user now, so as to minimise the disruption of the user's existing usage pattern; or if the electricity price of the current and next time slot is quite low (lower than threshold value Th_3), then sub-recommendation *switch_on*(i) should be advised to the user, so as to help him save more money.

[8] `www.edf.co.uk`

- $GS_{on_and_off}$ is proposed based on the domain knowledge that if the user usually switches on appliance i at the current time slot and uses it just one time slot, then sub-recommendation *on_and_off*(i) should recommended to the user now, so as to minimise the disruption of the user's existing usage pattern; or if the electricity price of the current time slot is quite low (lower than threshold value Th_3), then sub-recommendation *on_and_off*(i) should be advised to the user, so as to help him save more money.

- GS_{keep_off} is proposed based on the domain knowledge that whenever sub-recommendation *keep_off* is available for appliance i, *keep_off*(i) should be recommended to the user by default, because it helps the user save money. Also, if the user usually keeps off appliance i in the current time slot, *keep_off*(i) also can minimise the disruption.

- GS_{switch_off} is proposed based on the domain knowledge that whenever sub-recommendation *switch_off* is available for appliance i, *switch_off*(i) should be recommended to the user by default, because it helps the user save money. Also, if the user usually switches off appliance i in the current time slot, *switch_off*(i) also can minimise the disruption.

- GS_{keep_on} is based on the domain knowledge that if the user usually performs *keep_on* on appliance i in the current time slot, then sub-recommendation *keep_on*(i) should be recommended to the user now, so as to minimise the disruption of the user's existing usage pattern.

Given the good states of each sub-recommendations, we define the arguments that supporting primitive action $a = (a_{wm}, a_{hv}, a_{dw}, a_{kt}, a_{tv}, a_{pc})$ as follows (the requirement of the form of arguments used in AARL is given in Section 3.2.2; note that since we focus on the single-agent learning case now, we omit the agent that each argument belongs to):

$$\mathbf{PA}_a\text{: } a \text{ IF } \ s \in GS_{a_{wm}} \bigcap GS_{a_{hv}} \bigcap GS_{a_{dw}} \bigcap GS_{a_{kt}} \bigcap GS_{a_{tv}} \bigcap GS_{a_{pc}}$$

where $a_{wm}, a_{hv}, a_{dw}, a_{kt}, a_{tv}, a_{pc}$ are the sub-recommendations for the washing machine, hoover, dishwasher, kitchen, TV and PC, respectively, and s is the current state. So we can see that an argument $\mathbf{PA}_a$ is applicable in state s iff s is a good state for all sub-recommendations in a. Note that each primitive action has only a single argument supporting it. In the remainder of this chapter, we refer

to arguments of this form as *primitive arguments*, or simply P-arguments, because each of them supports a primitive action.

As an example, let us still consider the primitive action $a_1 = (wm$: *keep_off*, hv : *switch_on*, pc : *keep_off*, tv : *keep_on*, kt : *switch_off*, dw : *switch_on*$)$. Its corresponding argument is $\mathbf{PA}_{a_1}$. This argument is applicable in s iff s is a good state for recommending *keep_off* to the washing machine, for recommending *switch_on* for the hoover, ..., and for recommend *switch_on* to the dish washer.

After proposing the P-arguments, we now consider the values that are promoted by these arguments. We use a similar method to construct the values: we first identify the 'sub-value' of each sub-recommendation, and then build a value by combining the sub-values. For example, consider action $a_1 = (wm$: *keep_off*, hv : *switch_on*, pc : *keep_off*, tv : *keep_on*, kt : *switch_off*, dw : *switch_on*$)$. We know its argument is $\mathbf{PA}_{a_1}$, and we try to identify the value it promotes. We may think that to give sub-recommendation *keep_off* to washing machine is to promote a sub-value 'save money', to give sub-recommendation *switch_on* to hoover is to promote sub-value 'minimise disruption', etc. These sub-values can be combined to form a new value, referred to as *P-value* (because they are promoted by P-arguments), which represent the value of a whole recommendation.

Now we propose the sub-values for each sub-recommendation. We can see that the domain knowledge for proposing good states actually provides the sub-values of each sub-recommendation. For example, from the domain knowledge for G_{switch_on}, we can see that *switch_on* can promote three sub-values under different conditions: if the user usually performs *switch_on* on appliance i at the current time slot, then sub-recommendation *switch_on*(i) promotes the sub-value 'to minimise the disruption of the user's existing usage pattern'; or if the electricity price of the current and next time slot is quite low, then *switch_on*(i) promotes the sub-value 'help the user to save more money'; when both these conditions hold in a state, *switch_on*(i) promotes 'help the user to save money while not disrupting his usage'. Formally, these sub-values are given as follows:

- **SV1**: To avoid disrupting the user's planned usage.
- **SV2**: To save some money for the user.
- **SV3**: To save some money for the user while minimising the disruption.

Given these sub-values, we can see that if a sub-recommendation a is recommended to an appliance i in state s, the sub-value $SV_i^s(a)$ promoted by a is:

- When a is *switch_on*, it promotes **SV1** if the user usually switches on appliance i at the current time slot and uses it for more than one time slot; it promotes **SV2** if the electricity price of the current and next time slot is quite low; and it promotes **SV3** if in state s, both these conditions are satisfied.

- When a is *on_and_off*, it promotes **SV1** if the user usually switches on appliance i at the current time slot and uses it just one time slot; it promotes **SV2** if the electricity price of the current time slot is quite low; and it promotes **SV3** if both these conditions are satisfied.

- When a is *keep_off*, it promotes **SV3** if the user usually keep off i in the current time slot; otherwise, it promotes **SV2**.

- When a is *switch_off*, it promotes **SV3** if the user usually switches off i in the current time slot; otherwise, it promotes **SV2**.

- When a is *keep_on*, it always promotes **SV1**.

As we have discussed above, a P-value is a combination of six sub-values. So we represent a P-value as a vector, consisting of the sub-values promoted by each sub-recommendation. Since *switch_on* and *on_and_off* can promote at most three sub-values each (**SV1**, **SV2** and **SV3**), *keep_off* and *switch_off* can promote at most two sub-values (**SV2** and **SV3**), and *keep_on* can only promote **SV1**, the total number of P-values is $3 \times 3 \times 2 \times 2 = 36$. Now we define the promotion relation from P-arguments to P-values. Given a primitive action $a = (a_{wm}, a_{hv}, a_{pc}, a_{tv}, a_{kt}, a_{dw})$ and its corresponding P-argument A_a, when a is recommended to the user in state s, the P-value promoted by a is

$$< SV_i^s(a_{wm}), SV_i^s(a_{hv}), SV_i^s(a_{pc}), SV_i^s(a_{tv}), SV_i^s(a_{kt}), SV_i^s(a_{dw}) >,$$

and we denote this P-value as $PV^s(a)$.

For example, let us still consider $a_1 = (wm$: *keep_off*$, hv$: *switch_on*$, pc$: *keep_off*$, tv$: *keep_on*$, kt$: *switch_off*$, dw$: *switch_on*$)$. Suppose in one state s, its corresponding argument $\mathbf{PA}_{a_1}$ is applicable (i.e. s is in the intersection of all good states of each sub-recommendation in a_1), and suppose that in state s, *keep_off*(wm) promotes **SV3**, *switch_on*(hv) promotes **SV1**, *keep_off*(pc) promotes **SV2**, *keep_on*(tv) promotes **SV1**, *switch_off*(kt) promotes **SV3** and *switch_on*(dw) promotes **SV2**. Then the value promoted by argument $\mathbf{PA}_{a_1}$ is $PV^s(a_1) =<$ **SV3**, **SV1**, **SV2**, **SV1**, **SV3**, **SV2** $>$.

Then we discuss the ranking of these P-values. To rank them, we need to trade off between the value of money-saving and minimising disruption. We propose a function $F : SV \times AP \times S \rightarrow \mathbb{R}$ to compute the 'score' for sub-value promoted by a sub-recommendation given to appliance i in state s, where $SV = \{\textbf{SV1}, \textbf{SV2}, \textbf{SV3}\}$ is the set of sub-values, AP is the set of all appliances, and S is the set of all states. For example, if *keep_off* is recommended to the TV in a state s, we can see that the sub-value promoted by *keep_off* in this case is $SV^s_{tv}(\textit{keep_off})$, and the 'score' we assign to this sub-value is $F(SV^s_{tv}(\textit{keep_off}), tv, s)$. The 'score' of each P-value is the sum of its component sub-values' score. By doing this, we can obtain a total order over the P-values by comparing their 'scores'.

Recall that the constant value pun (see Section 6.5) represents how much weight the simulated user puts on the value of minimising disruption. So $|pun|$ can be directly used to represent the 'importance' of sub-value **SV1**. As for sub-value **SV2**, its 'importance' is related to how much money can be saved for the user by a sub-recommendation. For a sub-recommendation a, if it is recommended to an appliance i in a state s, the expected money $M^s_i(a)$ saved by a is computed as follows: $M^s_i(a) = E_i(t^{i*}_s, t^{i*}_e) - E_i(t^i_s(a), t^i_e(a))$, where function $E_i(t_s, t_e)$ computes the money costs of using appliance i from time t_s (inclusive) until time t_e (inclusive); $(t^{i*}_s, t^{i*}_e) = \text{argmax}_{t_s,t_e} P_i(t_s, t_e)$, namely (t^{i*}_s, t^{i*}_e) is the most likely start-time-end-time pair for appliance i given the user's existing usage habit; $t^i_s(a)$ and $t^i_e(a)$ are the start and end time advised by atomic action a for appliance i, respectively. Given the 'score' of each sub-values presented above, we let function F assign real numbers to different sub-values as follows: $F(\textbf{SV1}, i, s) = |pun|$, $F(\textbf{SV2}, i, s) = M^s_i(a)$, and $F(\textbf{SV3}), i, s) = M^s_i(a) + |pun|$. And then we overload F to allow it to accept P-values as input, so that for a P-value V, $F(V, i, s) = \sum_{sv} F(sv, i, s)$, where sv are all component sub-values in V. Given this way of ranking values, we can see that the value ranking can be different in different states.

6.6.2 Arguments Supporting Composite Sub-tasks and Their values

In the previous subsection (Section 6.6.1), we introduced the arguments for primitive actions (recommendations) and the values they promote. In this section, we introduce the arguments whose conclusions are composite sub-tasks in MAXQ (see Section 6.5) and values they promote. Note that the arguments we proposed in Section 6.6.1 are also applicable for primitive actions in MAXQ.

First, we propose some arguments that support composite sub-tasks as follows

(s is the current state, $s.time$ is the current time slot, and i ranges over appliances that have not been used until $s.time$ in the current day; also, we let $P_i(t_s) = \sum_{t=1}^{48-t_s} P_i(t_s, t_s + t)$, so $P_i(t_s)$ can be viewed as the marginal probability of switching on appliance i in time t_s):

- **CA1**: *Cleaning* IF $P_{hv}(s.time) > Th_2$ and hoover has not been used today
- **CA2**: *Cooking* IF $P_{kt}(s.time) > Th_2$ and kitchen electronics have not been used today
- **CA3**: *Relaxing* IF PC has not been used today and $P_{pc}(s.time) > Th_2$
- **CA4**: *KeepAllOff* IF $\max_i P_i(s.time) < Th_2$
- **CA5**: *UseCompatible* IF none of the conditions of **CA1**, . . ., **CA4** hold.

We refer to these arguments as *C-arguments*, because they support composite arguments. The domain knowledge behind **CA1**, **CA2** and **CA3** is to advise the user to use the hoover, kitchen electronics and PC, respectively, if these appliances are usually used by the user in the current time slot. As for **CA4**, it represents the domain knowledge of recommending the user to stop using all selected appliances if the user usually uses none of them in the current time slot, and **CA5** is to recommend the user to use only compatible appliances when the user usually does not use any of the exclusive appliances.

From the domain knowledge behind the C-arguments, we can see that unlike P-arguments, which may promote both the value of minimising disruption and money-saving, C-arguments are only based on the idea of 'giving users advises that are close to their existing usage'. This is because the money that can be saved by each composite sub-task is too difficult to be accurately evaluated: within a composite sub-task, unpredictable numbers of primitive actions can be executed in an unpredictable order, and the user's responses to these primitive actions are also unpredictable. As a result, for simplicity, we make all C-arguments promote the same value 'minimising disruption', and we denote this value as **CV**.

6.7 Experimental Settings and Results

To comprehensively evaluate the performances of MAXQ- and SARSA-based AARL, each experiment is divided into two phases: a *learning* phase and an *evaluation* phase. The detailed settings of all our experiments are presented in

Section 6.7.1, and the performances of the learning and evaluation phases are presented in 6.7.2 and 6.7.3, respectively.

6.7.1 Experimental Settings

We use JAVA to implement the simulated environment. We implemented EURS by using four categories of RL algorithms: standard SARSA(0), SARSA(0)-based AARL, standard MAXQ-0 and MAXQ-based AARL. Because we do not use eligibility traces (ET) in MAXQ-0 and PBRS-MAXQ-0 (however, note that ET techniques can be used in these algorithms, because these algorithms are TD-based; see Section 2.2.4), for comparative reasons, we also let $\lambda = 0$ in standard SARSA(λ) and SARSA(λ)-based AARL. Also note that since we have six different implementations of each AARL algorithm (see Figure 6.4).

In all AARL algorithms, we use the *ASPARTIX* library [DGWW11] to compute the preferred and grounded extensions. To reduce computational time, once extensions are computed, we store the current state and the extensions, so that when the same state is encountered again in the future, we can quickly obtain the corresponding extensions without computing them again.

Our experiments are organised as follows: for each RL implementation, we perform 100 independent experiments, each consisting of a *learning* phase. A learning phase consists of 2000 days (episodes). After all experiments finished, we perform the evaluation phase, in which the learning is stopped (i.e. C and V values are not updated, and we let $\epsilon = 0$ so as to stop exploration) and EURS uses the learnt policy to give recommendations to the same simulated user for another 365 days. The purpose of the learning phases is to test initial performances as well as the convergence speed of different algorithms, and the purpose of the evaluation phases is to evaluate the quality of the learnt strategies.

Now we give the constant and threshold values used in the experiments. We use two different values of pun (see Section 6.5): -1 and -3, and two different values of Th_1: 0 and -0.02. For the threshold values Th_2 and Th_3 (see Section 6.6.1), we have $Th_2 = 0.1$ and $Th_3 = 10$. Recall that in MAXQ-based AARL, we require the domain expert to give a positive real number c_i for each composite sub-task i in the MAXQ decomposition (see Section 6.1 and Algorithm 12 in that section). As for the potential value for Root, we set it as $c_1 = |pun|$; and for other composite sub-tasks, we let their potential values be the same: $c_2 = |pun|/6$. In SARSA(0)-based AARL, the potential value for all actions is also set as c_2

(see Algorithm 7 for where this potential value is used). Our preliminary experimental results indicate that the values of c_1 and c_2 have a big influence on the AARL's convergence speed: if these values are too small (e.g. $c_1 < |pun|/10$ and $c_2 < |pun|/20$), the potential values will have little influence on the learning performance, and thus the AARL's performance is almost the same as those of standard RL (i.e. SARSA(0) and MAXQ-0) algorithms; on the other hand, when we use big c_1 and c_2 values (e.g $c_1 \geq |pun| \times 3$ and $c_2 \geq |pun|/2$), the learning performance is mostly determined by the potential values, not the rewards received during the learning, and in these cases, the learning algorithms take a very long time to converge to the optimal policy, although they reach a sub-optimal policy quickly in the beginning of the learning. Also, we find that the smaller the absolute value of pun, the smaller the values of c_1 and c_2 should be, because pun directly influences the amount of rewards EURS can receive (see Section 6.5). Therefore, we let c_1 and c_2 be proportional to pun.

As for the electricity price rates, we use one of the most widely used tariffs in the UK: *Economy7*. In this tariff, the price rate is 5.52 p/kWh between 0:30 and 7:30, and 15.39 p/kWh at other times. In our preliminary experiments, we have also used another popular tariff in the UK: *Economy10*, in which the price rate is 8.98 p/kWh during 4:30 - 7:30, 8:30 - 12:30 and 11:30 - 16:30, and 16.92 p/kWh at other times. However, preliminary results show that the performances of all our implemented algorithms under Economy7 and Economy10 are very similar (i.e. given the same algorithm and under the same parameters, the learning curves in Economy7 and Economy10 looks very similar, while the rewards received in Economy10 is slightly larger than those in Economy7 because Economy10 has more distributed off-peak periods) and, therefore, we only present the learning performances under Economy7 below.

As for the learning parameters, in all four learning algorithms we have implemented, we have $\gamma = 1$, and we use the ϵ-greedy policy as the action selection policy. In SARSA(0) and MAXQ-0, we initialise α and ϵ as 1 in the first episode of each experiment, make α decrease at a rate of 0.999 (i.e. $\alpha_{t+1} = \alpha_t \times 0.999$), and make ϵ decrease at a rate of 0.99. In SARSA(0)-based and MAXQ-based AARL, we have α and ϵ as 0.3 and 0.5, respectively, in the first episode of each experiment, and let them decrease at a rate of 0.999 and 0.9, respectively. Note that, for each learning algorithm, the above parameters are chosen so as to maximise their respective performance in the evaluation phases. We can see that the ϵ values and their decreasing rate are smaller in AARL algorithms than in standard

RL algorithms (i.e. SARSA(0) and MAXQ-0), consistent with our assumption that AARL algorithms need less exploration.

6.7.2 Results in Learning Phases

The performances of SARSA(0)-based RL algorithms are presented in Figure 6.5, while those of the MAXQ-based algorithms are presented in Figure 6.6. We first analyse the performances of SARSA(0)-based algorithms. We can see that, under all settings, throughout the 2000-episode experiments, most of the SARSA(0)-based AARL algorithms' learning curves are above that of SARSA(0) (except when $pun = -1, Th_1 = -0.02$, the performance of P-A-D is almost the same as that of SARSA(0); we will discuss the possible reasons shortly), indicating that, generically, AARL algorithms are significantly better than SARSA(0) in terms of initial performance and convergence speed. With respect to the performances of different AARL implementations, we find that when $pun = -3$ (see the figures on the left-hand side in Figure 6.5), the AARL algorithms that rank values according to the rule we presented in Section 6.6.1 (i.e. P-A-D, P-R-D and G-D) perform significantly better than the other implementations, which rank all values equally (see Section 6.6), in the first 500 episodes of the learning. This result indicates that our value-ranking rules presented in Section 6.6.1 are reasonable and therefore able to help the corresponding algorithms achieve the optimal (or near-optimal) policies more quickly. Also, we find that the G-D implementation of AARL (i.e. the AARL that uses the grounded semantics and ranks values differently, see Figure 6.4) has the (significantly) best performance in the first 500 episodes. The reason of this remains unclear and is worth further investigation, but we assume it can be that the grounded extensions only recommend the 'convincingly good' actions, and by doing this, the risk of being punished can be reduced and hence the rewards received is higher.

However, under the setting $pun = -1$, no AARL has significant advantages over the other AARL implementations. We speculate that the reason may be that the shaping rewards (i.e. the c values, see Section 6.7.1) we give are proportional to the absolute value of pun; so when $pun = -1$, the shaping rewards AARL gives are all small and, hence, their influence on the learning process is weak. Because all AARLs' effects on the learning are weak, neither of them can 'stand out', and their improvements over SARSA(0) are not as big as those under the setting $pun = -3$.

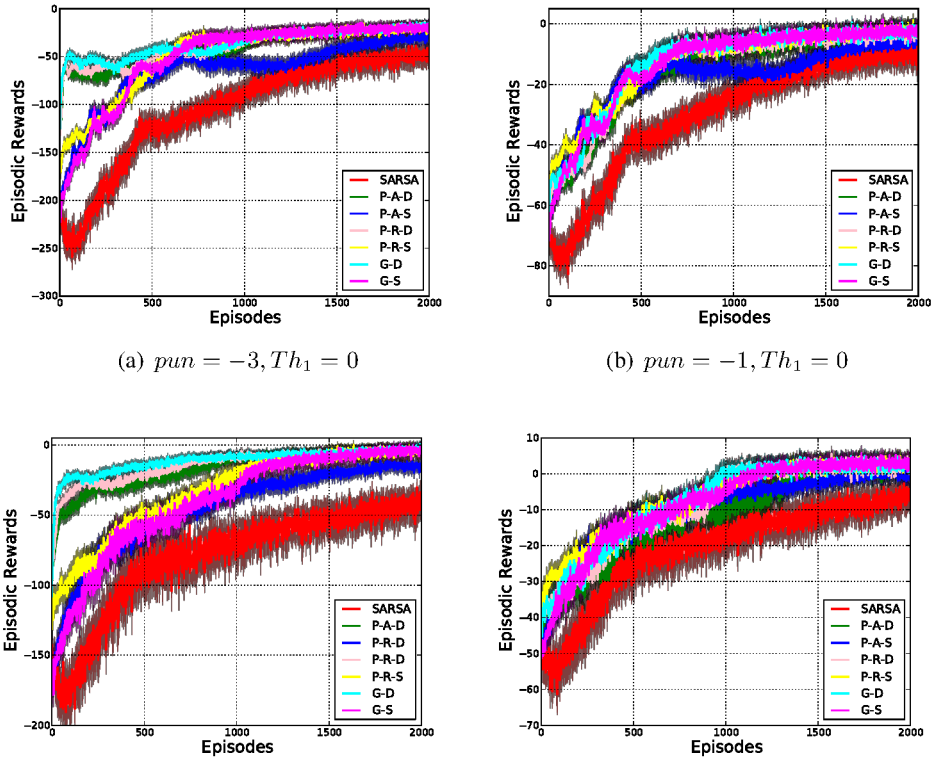
Then we analyse the performances of MAXQ-based algorithms (see Figure 6.6). We can see that under all settings, all AARL algorithms perform significantly better than MAXQ-0, in terms of their initial performances and convergence speed. Also, we find that each MAXQ-based algorithm performs significantly better than their counterpart SARSA-based algorithm, in terms of both initial performance and convergence speed. These results not only suggest that MAXQ-based algorithms can learn more effectively than SARSA-based algorithms, but also indicate that MAXQ-based AARL can also effectively improve the performance of standard MAXQ-0.

However, we find that all AARL implementations have very similar performances: none of them has significant advantages over the other AARL implementations under all settings. We believe that this is due to the fact that all MAXQ-based AARL implementations use the same arguments for composite sub-tasks (see Section 6.6.2); as a result, in the same state, all MAXQ-based AARL implementations recommend the same composite sub-task, and this explains why their performances are similar. From this observation, we can also see that the potential values for composite sub-tasks play an important role in improving the initial performances of AARL, because these potential values can help the system to reduce the scope of primitive actions that need to be considered. This result also indicates the importance of the PBRS-MAXQ-0 algorithm, which allows the designer to give potential values for composite sub-tasks.

6.7.3 Results in the Evaluation Phases

The average rewards and their standard errors of SARSA(0)-based and MAXQ-based algorithms are presented in Tables 6.1 and 6.2, respectively. We first analyse the data for SARSA(0)-based algorithms. We can see that under all settings, the performance of SARSA(0) is significantly worse than all other AARL algorithms (indeed, Tukey's range test [Tuk49] p-values[9] are smaller than 0.01; pairwise p-value comparisons are given in Table 6.3), indicating that even after 2000 episodes of learning, the SARSA(0) algorithm still has not converged. As for the quality of different AARL implementations, we see that no AARL implementation signifi-

[9] In the remaining of this subsection, unless stated otherwise, all p-values are obtained from Tukey's range test after ANOVA tests. We are using this method instead of using two-tailed t-test because doing multiple two-tailed t-tests would result in an increased chance of committing a statistical type I error. For more details about ANOVA test and Tukey's range test, please refer to e.g. [TS$^+$01, Hay86, Tuk49].



(a) $pun = -3, Th_1 = 0$

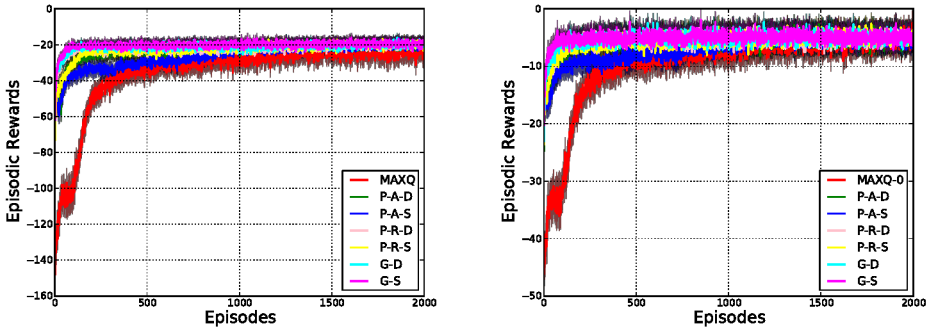
(b) $pun = -1, Th_1 = 0$

(c) $pun = -3, Th_1 = -0.02$

(d) $pun = -1, Th_1 = -0.02$

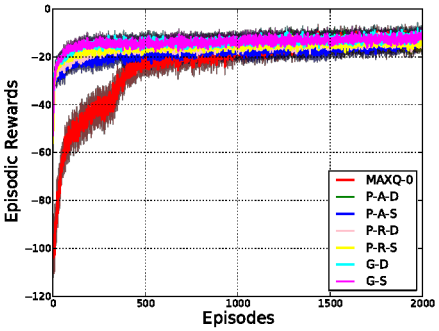
Figure 6.5: Performances of SARSA(0)-based EURS in the learning phases. Lighter colour areas represent 95% confidence intervals. All results are averaged over 100 independent experiments, each consisting of 2000 episodes.

cantly outperforms all other AARL implementations in any setting. For example, when $pun = -3, Th_1 = 0$, although AARL with P-A-D has the best performance, it does not have significant advantages over AARL with P-R-D and P-R-S (as the Tukey's range test p-values are 0.94 and 0.10, respectively). However, we find that under all settings, the performances of P-A-D and P-R-D are either the best or have no significant disadvantages compared with the best. This observation is partly consistent with our observation of the learning curves (see Figure 6.5): actually, in the previous subsection, we have pointed out that AARL algorithms with differently ranked values converge faster. However, we find that when the user is less likely to change his original behaviour (i.e. $Th_1 = 0$), AARL with G-D does not perform well after 2000 episodes of learning, whereas when $Th_1 = -0.02$,

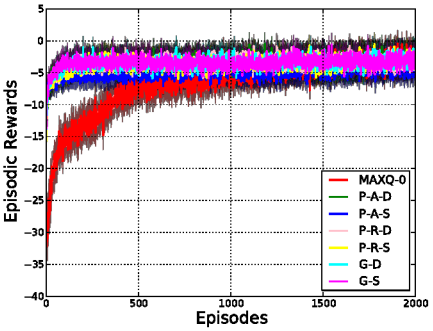


(a) $pun = -3, Th_1 = 0$

(b) $pun = -1, Th_1 = 0$



(c) $pun = -3, Th_1 = -0.02$



(d) $pun = -1, Th_1 = -0.02$

Figure 6.6: Performances of MAXQ-based EURS in the learning phases. Lighter colour areas represent 95% confidence intervals. All results are averaged over 100 independent experiments, each consisting of 2000 episodes.

G-D has no significant disadvantage. The reason behind this is unclear and worth further investigation.

Then we look at the results for MAXQ-based algorithms. The pairwise p-values for these algorithms can be found in Table 6.4. We can see that unlike SARSA-based AARL, which all significantly outperform standard SARSA(0) after 2000 episodes of learning, the MAXQ-based algorithms do not significantly outperform standard MAXQ-0 after 2000 episodes of learning. To be more specific, we can see that when $pun = -1$, MAXQ-0 does not have significant disadvantages compared to the best AARL implementation under the same setting. This observation indicates that MAXQ-0 is able to converge within 2000 episodes when $pun = -1$. However, as for why MAXQ-0 converges slower when $pun = -3$, the reason re-

mains unclear.

As for the relative quality of different MAXQ-based AARL implementations, we find that no AARL implementation is able to significantly outperform all other implementations under each setting. Also, we find that no algorithm is able to have 'good' performance under all settings, where 'good' means either the best performance or no significant disadvantage compared to the best. The reason of this mixed result is worth further investigation, but at least we can draw the conclusion that each MAXQ-based AARL has its strengths and weaknesses, and none of them have significant advantages or disadvantages given all four types of simulated users.

Then we compare the performances between SARSA(0)- and MAXQ-based algorithms. By comparing the counterpart numbers in Table 6.1 and 6.2, we can see that under each setting, MAXQ-0 significantly outperforms SARSA(0) (indeed, all p-values are smaller than 0.01, using two-tailed t-test). In addition, under each setting, we compare the best MAXQ-based algorithm's performance with that of the best SARSA(0)-based algorithm, and find that when $pun = -1, Th_1 = 0$ and $pun = -3, Th_1 = -0.02$, their performances do not have significant differences (p-value are 0.33 and 0.13, respectively, using two-tailed t-test); under the other two settings, however, MAXQ-based AARL's best performance is significantly better than SARSA(0)-based AARL's (p-values are smaller than 0.05, using two-tailed t-test). These results indicate that when $pun = -1, Th_1 = 0$ and $pun = -3, Th_1 = -0.02$, SARSA(0)-based AARL implementations are able to converge faster than in the other two settings. Further investigations into which factors affect the convergence speed in AARL are essential.

Table 6.1: The average rewards and their standard errors of different SARSA(0)-based algorithms in the evaluation phases. All results are averaged over 365 episodes. In each column, the best performance is in boldface, and the performances that do not have a significant disadvantage compared with the best performance are in italic.

pun/Th_1	$-3/0$	$-1/0$	$-3/-0.02$	$-1/-0.02$
SARSA(0)	-58.78 ± 2.21	-17.85 ± 0.93	-46.21 ± 1.74	-13.39 ± 0.62
P-A-D	$\mathbf{-18.94 \pm 0.60}$	*-6.40 ± 0.48*	$\mathbf{-5.22 \pm 0.33}$	*-3.89 ± 0.40*
P-A-S	-23.89 ± 1.26	-8.56 ± 0.34	-16.90 ± 0.65	$\mathbf{-2.61 \pm 0.15}$
P-R-D	*-20.71 ± 0.54*	*-5.78 ± 0.42*	*-7.14 ± 0.20*	*-3.99 ± 0.39*
P-R-S	*-23.43 ± 0.99*	-7.58 ± 0.41	*-7.15 ± 0.76*	*-2.66 ± 0.39*
G-D	-30.83 ± 0.54	-6.97 ± 0.43	*-5.66 ± 0.24*	*-3.89 ± 0.40*
G-S	-26.27 ± 1.12	$\mathbf{-4.69 \pm 0.34}$	-8.97 ± 0.75	-4.43 ± 0.39

Table 6.2: The average rewards and their standard errors of different MAXQ-based algorithms in the evaluation phases. All results are averaged over 365 episodes. In each column, the best performance is in boldface, and the performances that do not have a significant disadvantage compared with the best performance are in italic.

pun/Th_1	$-3/0$	$-1/0$	$-3/-0.02$	$-1/-0.02$
MAXQ-0	-26.73 ± 1.21	$\mathit{-4.72 \pm 0.52}$	-16.11 ± 0.73	$\mathit{-1.54 \pm 0.54}$
P-A-D	-19.43 ± 0.63	-8.23 ± 0.48	-12.38 ± 0.59	$\mathit{-0.29 \pm 0.43}$
P-A-S	-19.23 ± 0.73	$\mathit{-5.30 \pm 0.40}$	$\mathit{-8.62 \pm 0.60}$	-3.75 ± 0.38
P-R-D	$\mathbf{-14.78 \pm 0.73}$	$\mathit{-5.88 \pm 0.41}$	-10.08 ± 0.63	$\mathbf{0.08 \pm 0.51}$
P-R-S	-19.12 ± 0.60	$\mathit{-5.18 \pm 0.37}$	$\mathbf{-6.30 \pm 0.63}$	$\mathit{-0.24 \pm 0.48}$
G-D	$\mathit{-17.83 \pm 0.59}$	$\mathit{-4.57 \pm 0.41}$	$\mathit{-8.86 \pm 0.57}$	-3.70 ± 0.35
G-S	-18.62 ± 0.58	$\mathbf{-4.17 \pm 0.41}$	-9.23 ± 0.51	$\mathit{-0.37 \pm 0.45}$

6.8 Related Works

RL has become a popular technique for designing RDR systems in recent years. O'Neill et al. [OLGM10] proposed a RDR system which can directly control the switch on time of some appliances in a household; when the user tries to switch on an appliances (e.g. by pressing this appliance's starting button), instead of immediately using the appliance, the user essentially makes a 'reservation' of that appliance to the RDR system; the system will decides when to actually switch on that appliance, so as to keep a good balance between maximising money saving and minimising disruption. However, the simulated user they used to interact with their system is simple and unrealistic: they assume that the probability of the user using an appliance in the current time is totally dependent on his usage of this appliance in the previous time slot; in other words, they assume that the user's usage of each appliance is totally Markovian, which is unrealistic as suggested by some research [TMTT+13]. Also, we argue that the assumption that the RDR system can directly control the usage of some appliances is unrealistic and may be unwelcome for residential users.

More RL-based RDR systems use the assumption that each household is able to store some electricity temporally and sell them out to other households or the power station, e.g. [CVRH12, LZL+12, CPG12]. However, the target of these research is to maximise the energy distribution in a community, a specific area or a cluster of devices, whereas our EURS is to maximise the money saving for individual household.

Some other work, although not directly about designing RDR systems, focuses on other approaches for saving money for residential users. Ramchurn et al.

Table 6.3: Pairwise p-values of SARSA-based algorithms in EURS. A, B, C and D stand for settings $pun = -3, Th_1 = 0$, $pun = -1, Th_1 = 0$, $pun = -3, Th_1 = -0.02$ and $pun = -1, Th_1 = -0.02$, respectively. All p-values are computed by using Tukey's range test [Tuk49].

	A	B	C	D
SARSA vs P-A-D	0.00	0.00	0.00	0.00
SARSA vs P-A-S	0.00	0.00	0.00	0.00
SARSA vs P-R-D	0.00	0.00	0.00	0.00
SARSA vs P-R-S	0.00	0.00	0.00	0.00
SARSA vs G-D	0.00	0.00	0.00	0.00
SARSA vs G-S	0.00	0.00	0.00	0.00
P-A-D vs P-A-S	0.04	0.04	0.00	0.29
P-A-D vs P-R-D	0.94	0.98	0.66	1.00
P-A-D vs P-R-S	0.10	0.67	0.65	0.34
P-A-D vs G-D	0.00	0.99	1.00	1.00
P-A-D vs G-S	0.00	0.22	0.02	0.97
P-A-S vs P-R-D	0.47	0.00	0.00	0.21
P-A-S vs P-R-S	1.00	0.83	0.00	1.00
P-A-S vs G-D	0.00	0.30	0.00	0.29
P-A-S vs G-S	0.78	0.00	0.00	0.03
P-R-D vs P-R-S	0.66	0.17	1.00	0.25
P-R-D vs G-D	0.00	0.66	0.87	1.00
P-R-D vs G-S	0.01	0.75	0.71	0.99
P-R-S vs G-D	0.00	0.98	0.86	0.34
P-R-S vs G-S	0.61	0.00	0.71	0.04
G-D vs G-S	0.09	0.03	0.07	0.97

Table 6.4: Pairwise p-values of MAXQ-based algorithms in EURS. A, B, C and D stand for settings $pun = -3, Th_1 = 0$, $pun = -1, Th_1 = 0$, $pun = -3, Th_1 = -0.02$ and $pun = -1, Th_1 = -0.02$, respectively. All p-values are computed by using Tukey's range test [Tuk49].

	A	B	C	D
MAXQ-0 vs P-A-D	0.00	0.00	0.00	0.45
MAXQ-0 vs P-A-S	0.00	0.96	0.00	0.01
MAXQ-0 vs P-R-D	0.00	0.48	0.00	0.15
MAXQ-0 vs P-R-S	0.00	0.99	0.00	0.40
MAXQ-0 vs G-D	0.00	1.00	0.00	0.01
MAXQ-0 vs G-S	0.00	0.97	0.00	0.53
P-A-D vs P-A-S	1.00	0.00	0.00	0.00
P-A-D vs P-R-D	0.00	0.00	0.11	1.00
P-A-D vs P-R-S	1.00	0.00	0.00	1.00
P-A-D vs G-D	0.74	0.00	0.00	0.00
P-A-D vs G-S	0.99	0.00	0.01	1.00
P-A-S vs P-R-D	0.00	0.96	0.62	0.00
P-A-S vs P-R-S	1.00	1.00	0.10	0.00
P-A-S vs G-D	0.85	0.90	1.00	1.00
P-A-S vs G-S	1.00	0.51	0.99	0.00
P-R-D vs P-R-S	0.00	0.91	0.00	1.00
P-R-D vs G-D	0.06	0.32	0.80	0.00
P-R-D vs G-S	0.01	0.08	0.96	0.99
P-R-S vs G-D	0.89	0.95	0.05	0.00
P-R-S vs G-S	1.00	0.65	0.01	1.00
G-D vs G-S	0.99	0.99	1.00	0.00

[ROP+13] proposed a web-based system for recommending tariffs for residential users. They developed novel extensions to Bayesian Quadrature [OGR+12], a machine learning technique, in order to generate predictions of yearly consumption at hourly level. These estimates can then be directly used to select the best tariff available from energy retailers. Also, they gave some advice on which form of recommendations the users prefer to receive. However, we can see that their work is more focusing on predicting the user's usage, whereas our EURS is more focusing on giving advice so as to change the user's usage. Truong et al. [TMTT+13] developed a *graphical model* [Jor99] based approach for predicting a user's energy usage. Their model assumes that people's usage repeats routinely, and in the same *type* of days a user's energy usage is similar. However, their approach is off-line, in the sense that they need to fit considerable data into their model at once, and then the model returns the predicted usage during a certain period of upcoming time. Also, their work does not provide any recommendations to the user.

6.9 Conclusion

In this chapter, we propose the MAXQ-based AARL algorithm and test its effectiveness in a novel real application: *Energy Usage Recommendation System* (EURS). We implement a simulated environment, which includes a real-data-based simulated user and EURS. EURS is implemented by using four different RL algorithms: SARSA(0), SARSA(0)-based AARL, MAXQ-0 and MAXQ-based AARL. In both SARSA(0)- and MAXQ-based AARL, we use different argumentation semantics to select the 'winning' arguments, and also use different methods in deriving heuristics from the winning arguments. We perform detailed analysis on the performances of different approaches so as to obtain deeper insights into the properties of each approach. Our analysis indicates that MAXQ-based AARL outperforms all other three approaches, while SARSA(0)-based AARL outperforms standard SARSA(0).

We focus on only the single-agent MAXQ-based AARL in this chapter. A main reason is that coordinating multiple MAXQ-based learning agents involves considerable application-specific knowledge and much trail and error: in order to coordinate multiple MAXQ agents, the 'depth of cooperation', namely which layers of sub-tasks should to be coordinated between different agents, needs to be considered carefully according to the specific application domain[GMM06]. As this is the first work in proposing MAXQ-based AARL, we choose to only consider

the single-agent learning algorithm, so that we can focus on how to incorporate argumentation frameworks into MAXQ. Nevertheless, we believe the single-agent MAXQ-based AARL algorithm can be potentially extended to multi-agent algorithms, and this is left as an important future work. More details of this extension will be discussed below in Chapter 7.

7 Conclusion

This dissertation addresses the topic of planning and learning for stochastic, sequential decision making. We propose the *Argumentation Accelerated Reinforcement Learning* (AARL) framework, which is an integration of Argumentation Theory and Reinforcement Learning (RL), to address this problem, because the heuristics derived from domain knowledge that is represented in the form of argumentation frameworks can help RL to make sequential decisions more efficiently in stochastic complex environment. This integration not only combats problems faced by both participating techniques — for example the curse of dimensionality faced by RL and a lack of sequential-decision-making capability in Argumentation techniques — but also partly fills some long-standing gaps: in terms of decision-making, this integration partly fills the gap between the proactive and reactive views of decision-making, because it employs Argumentation to plan good decisions proactively, and uses the results to instruct reactive decision-making in RL; in terms of Artificial Intelligence (AI), it partly fills the gap between logic- and statistic-based AI, because Argumentation is a logic-based reasoning technique, while RL learns how to make decisions in stochastic environments by learning probabilistic properties of the environment.

We work on both the Argumentation side and the RL side to fill these gaps: on the Argumentation side, we propose a generic framework for representing domain knowledge by using *Value-based Argumentation Frameworks* (VAFs), derive heuristics from this domain knowledge by selecting the 'winning arguments', and prove that the heuristics generated by our argumentation framework can not only instruct individual learning agents, but also coordinate multiple cooperative independent learning agents. On the RL side, we apply the classical *Potential Based Reward Shaping* (PBRS) techniques to a popular *hierarchical RL* (HRL) algorithm: MAXQ-0, and propose the PBRS-MAXQ-0 algorithm. We prove that under certain conditions, PBRS-MAXQ-0 is guaranteed to converge to the optimal hierarchical policy, regardless of the potential values being used. PBRS-MAXQ-0 facilitates the integration of argumentation-generated heuristics into MAXQ-0.

Both sides' work contribute to the AARL framework, which can be implemented based on both flat RL algorithms (e.g. SARSA(λ)) as well as hierarchical RL (HRL) algorithms (e.g. MAXQ-0).

From the empirical perspective, we perform experiments on multiple application domains. To be more specific, we test the effectiveness of SARSA(λ)-based AARL in RoboCup Soccer Keepaway and Takeaway games (Section 4.2), test the effectiveness of PBRS-MAXQ-0 in the Taxi problem and a stochastic Wumpus World game (Section 5.2), and test the effectiveness of both SARSA(0)- and MAXQ-based AARL in a novel residential demand response system (Chapter 6). In all these experiments, to varying degrees, AARL algorithms outperform their standard RL counterparts in terms of initial performance and convergence speed. These results indicate the effectiveness of AARL. Moreover, besides the experiments mentioned above, we also perform an experiment to evaluate the usability of SARSA(0)-based AARL, in which we ask 128 students to give argument-based domain knowledge to instruct a RL agent in a Wumpus World; results indicate that our argumentation framework used in AARL can effectively help people that have very little background in both Argumentation and RL to integrate domain knowledge into RL.

As for future work, we identify two lines of work worth further investigation: **(1)** how to extend, generalise and refine the techniques presented in this thesis (including the AARL framework, the PBRS-MAXQ-0 algorithm, etc.), so as to make them more effective and to apply them to more challenging application domains; and **(2)** how to propose some paradigms or frameworks, which are based on techniques other than Argumentation or RL, but follow the same spirit of incorporating statistic- and logic-based AI approaches. We will discuss both lines of future work below.

First, we see much potential for improving the techniques presented in this thesis:

- We can improve and generalise our VAF-based argumentation framework (Chapter 3), towards several directions.
  - This argumentation framework can be based on argumentation techniques other than VAFs to represent the preferences over arguments: for example, by allowing arguments to attack attacks [Mod09], the preferences over arguments can be directly represented without using values. An immediate advantage of this replacement is that the domain

experts do not need to provide all value-related domain knowledge (the value set, the promotion relation and the partial order over values), and only need to inform the argumentation framework on which arguments attack which attacks. Other properties of this replacement are unclear, and thus worth further investigation.

– SCAFs and VSCAFs only include arguments whose conclusions are actions (See Chapter 3). In other words, only one kind of information can be represented by these arguments, that is 'some actions are good under certain conditions'. However, in some applications, domain experts may need represent other kinds of information in the argumentation frameworks. For example, in our Wumpus World that involves students, we have asked the students to provide arguments about 'which actions should not be performed under certain conditions', and we find that students can also easily propose this kind of arguments. However, as our current AARL does not allow these 'negative arguments', we do not use these arguments in our experiments. Intuitively, including these negative arguments can prevent the agents from performing some obviously 'stupid' actions and thus improve the convergence speed. As a result, in the future, we will extend AARL to allow for these negative arguments, and investigate the theoretical and empirical influence of this extension.

  Another line of research to extend the arguments in AARL is to introduce *epistemic arguments*, whose conclusions are not actions but, instead, the premises of other arguments (the arguments whose conclusions are actions are known as *practical arguments* [Amg09]). Since the epistemic arguments can argue about the applicability of practical arguments, by introducing epistemic arguments, the applicability of each arguments can also be justified. For example, let us revisit the argumentation framework we presented in Chapter 1 (see Figure 1.2). We can see that no arguments recommend action *go_down*. Let *D* be the argument 'perform *go_down* if going down is deemed safe'. This argument is inapplicable in the setting considered in Section 1.2. However, we did not explicitly justify why this argument is not applicable. By introducing epistemic arguments, we can represent the applicability decision process of this argument within the argumentation framework,

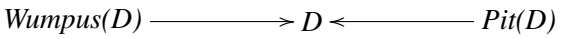


Figure 7.1: An AF including epistemic arguments *Wumpus(D)* and *Pit(D)* to justify the inapplicability of practical argument *D*, given the domain knowledge in Section 1.1.

e.g. as given in Figure 7.1. Here we use two epistemic arguments, *Wumpus(D)* and *Pit(D)*, to justify why argument *D* is inapplicable. Argument *Wumpus(D)* (*Pit(D)*) means 'going down is not safe because the agent feels stench (breeze) and there is no domain knowledge suggesting that the Wumpus (pit) is not on the downside'. We can see that the conclusion of these two arguments are the contrary of the premises of argument *D*; therefore, we build attacks point from these epistemic arguments to *D*. Note that an epistemic argument can also attack or be attacked by other epistemic arguments. From this example, we see that by using epistemic arguments, the justification of each practical arguments can also be represented by argumentation frameworks, and this allows for richer knowledge representation. Therefore, introducing epistemic arguments into our AARL is another interesting topic worth further investigation.

- Since different semantics use different criteria to select the 'winning' arguments (see Chapter 2), given the same argumentation framework, different semantics may give different recommendations, and these recommendations may be suitable for different problems. Therefore, the properties of the heuristics generated by semantics other than the grounded and preferred semantics are also worth investigation.
- In our current argumentation frameworks, the derived heuristics are only suitable for the problems in which 'cooperative agents should perform different sub-tasks (actions)' (see Section 3.2.3). However, in some cooperative multi-agent learning problems, this requirement can be invalid: for example, multiple agents may need to perform the same action at the same time so as to finish some difficult tasks. In these problems, we need an argumentation framework which can allow us to specify the requirements on the heuristics, e.g. which agents are allowed to perform the same action, which actions are allowed to be performed by multiple agents, etc., and this framework can generate

heuristics meeting the requirements we provide. This can be regarded as a generalisation of the current argumentation framework, and (at least intuitively) the new framework would be more flexible and applicable to more problems.

- We may extend this argumentation framework so that it can be used to generate heuristics for non-cooperative or even competitive multi-agent RL problems, e.g. the multi-agent Taxi problem [GMM06], where each agent is selfish but also accepts cooperation with other taxis if the cooperation can help each participating taxi to receive more rewards. We believe that argumentation frameworks can be suitable for these problems as well, because the dialectic nature of argumentation allows us to naturally fit argumentation into the communication between agents. For example, agents can use argumentation to effectively perform certain kinds of dialogue, e.g. information seeking or persuasion [Wal89], so as to quickly reach a cooperative deal such that each side's benefit can be maximised.
- In our current AARL, arguments can provide heuristics and hence influence the learning process, but the learning results cannot affect the argumentation frameworks. In other words, the interaction between argumentation frameworks and RL is 'one-way'. However, as the learning proceeds, we may find that some of the domain knowledge we provided at first is imperfect and, therefore, would like to use the learnt policies to 'revise' the domain knowledge encoded in the argumentation frameworks. For example, in Chapter 4 and Chapter 6, we can see that we sometimes conditionalise the value rankings, but these rankings may not be perfect in some situations. It will be ideal that the value rankings can be dynamically changed if the learning agents find that the value ranking being used lead to poor performance and have found better rankings instead. By doing this, we can close the interaction loop between the argumentation frameworks and the learning agents, and the learning-revised argumentation frameworks can be used by system designers to understand the policy found by the learning algorithm, or be used to facilitate *transfer learning* [TS09], i.e. using one learning agent's learning experiences to help other learning agents to converge faster.

- We plan to conduct more experiments involving people to use argumentation frameworks, similar to the experiments we introduced in Chapter 4, so as to investigate the effectiveness of AARL and, more generally, empirically validate the oft-heard but rarely tested claim that argumentation provides a useful paradigm for integrating human reasoning with computational reasoning [BCD07]. Some immediate future work include inviting the students to give preferences over the arguments they provide and, furthermore, asking them to 'debate' over the preferences they give (e.g. allowing students to give arguments that can attack other attacks [Mod09]).

- As for AARL algorithms (Chapter 4 and Chapter 6), we only select two RL algorithms — SARSA(λ) and MAXQ-0 — to instantiate the AARL algorithm. However, there are many other RL algorithms available, and AARL, in principle, can be implemented on any RL algorithms that have corresponding PBRS techniques (see the architecture of AARL in Chapter 3). We believe that other RL algorithms' implementations of AARL may have some properties that are not revealed by the two versions we have implemented (i.e. SARSA(λ)- and MAXQ-based AARL), and may have better performances in certain applications. For example, *R-MAX* [BT03], a popular *model-based* RL algorithm (for brief description of 'model-based' and 'model-free' RL algorithms, see Section 2.2.2; more detailed description can be found in [SB98]), has been successfully integrated into MAXQ algorithms [JS08] (the resulting algorithm is called *R-MAXQ*), and R-MAXQ can also be augmented with PBRS techniques [ALZ08]; as a result, we believe that PBRS-MAXQ-0 (see Chapter 5) can be applied to R-MAXQ with minor revisions, and the resulting algorithm will lay the foundation of R-MAXQ-based AARL. Since model-based RL algorithms can learn the transition probability of the environment more efficiently than model-free RL algorithms [SB98], we believe that the R-MAXQ-based AARL can be more suitable than model-free-RL-based AARL in some real applications.

  As for MAXQ-based AARL (Chapter 6), extending the MAXQ-based AARL to the multi-agent case is an interesting future work. We have shown in Chapter 3 that AARL can effectively coordinate multiple cooperative learning agents, and we believe that extending MAXQ-based AARL to multi-agent learning problems will also lead to performance improvement over

classic multi-agent MAXQ [GMM06]. We identify two main difficulties of this extension: **i)** how to allow agents with different MAXQ decomposition hierarchies to cooperate, and **ii)** how to decide the '*depth of cooperation*', namely which sub-tasks need to be considered in the coordination process.

- With respect to our Energy Usage Recommendation System (Chapter 6), the current system we use still has some limitations and is built upon some unrealistic assumptions: for example, our current system assumes that all appliances' usages are independent with one another, which is not the case for many appliances: for example, people usually use the dishwasher one or two hours after using kitchen electronics, and people always use the home theatre PC together with their TV sets. These correlations between usages of different appliances provide some more domain knowledge about the user's habits, and by using this knowledge properly, more 'satisfactory' recommendations are more likely to be provided. Also, our current system can only give recommendations in one specific *type of days* (see Section 6.4). In order to combat these limitations of EURS, we need to both improve the learning algorithm as well as the domain knowledge used by the system: on the learning side, we may use other machine learning algorithms to 'dig out' more useful information about the user, e.g. the correlations between appliances, the type of days, etc.; on the domain knowledge side, we may take into account some other information about the user, e.g. the weather of the day, the activities on the calendar of the user, etc.; this information can be represented by some new arguments and also be used to help the system to provide better recommendations.

Finally, besides the extensions of the current works as we discussed above, we may propose other paradigms or algorithms to incorporate logic- and statistic-based AI approaches, because we believe that this integration is an important trend in AI research. Our work is a preliminary step towards this direction, but in our current integration, both sides are working quite independently, and bonded together by using some 'glue' or 'midware' techniques (namely PBRS). Because of the relative independence of each side, this 'loose' integration has its advantage in scalability and flexibility. However, since logic- and statistic-based AI can compensate each other fundamentally (logic-based AI needs statistic-based AI to handle uncertainty, while statistic-based AI needs logic-based AI to handle complexity [Dom06]), we believe that the 'tight' integration of logic-and statistic-based AI

can lead to more effective AI techniques. There has been some research devoted to this 'tight' integration, so as to perform uncertain inferences: for example, *Markov Logic Network* [RD06], integrates first order logic with *Markov networks* [KS80], *Independent Choice Logic* [Poo97] integrates *acyclic logic programming* [AB91] with *Bayesian networks* [Pea88], Despite the successes of these approaches, they both have some limitations: Markov Logic Network is based on first order logic and, thus, cannot perform non-monotonic reasoning (for non-monotonic reasoning, see Chapter 1 and 2); and Independent Choice Logic restricts its representation to acyclic logic programs, which do not allow multiple models (a model can be roughly viewed as a solution of a logic program; more formal descriptions can be found in e.g. [GL88]). Argumentation techniques can perform non-monotonic reasoning, and some of them also allow cyclic logic programming (e.g. *Assumption based Argumentation* (ABA) [DKT09]); these techniques have not been integrated with statistic AI techniques like Bayesian networks or Markov networks,[1] and we believe that their integrations are also worth further investigation.

[1] We note that abstract argumentation frameworks have been integrated with Bayesian networks [Vre05], and ABA has variants in which assumptions and inference rules can have probabilities [Koh03, DT10]. However, we see no research that allow argumentation-based logic reasoning (e.g. ABA) and Bayesian or Markov networks.

Bibliography

[AB91] K. R. Apt and M. Bezem. Acyclic programs. *New generation computing*, 9(3-4):335–363, 1991.

[AC02] L. Amgoud and C. Cayrol. Inferring from inconsistency in preference-based argumentation frameworks. *Journal of Automated Reasoning*, 29(2):125–169, 2002.

[AES08] M. H. Albadi and E. F. El-Saadany. A summary of demand response in electricity markets. *Electric Power Systems Research*, 78:1989–1996, 2008.

[ALZ08] J. Asmuth, M. Littman, and R. Zinkov. Potential-based shaping in model-based reinforcement learning. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2008.

[Amg09] L. Amgoud. Argumentation for decision making. In *Argumentation in artificial intelligence*, pages 301–320. Springer, 2009.

[Ant97] G. Antoniou. *Nonmonotonic Reasoning*. MIT Press, 1997.

[AP09] L. Amgoud and H. Prade. Using arguments for making and explaining decisions. *Artificial Intelligence*, 173(3):413–436, 2009.

[BBS08] L. Buşoniu, R. Babuška, and B. Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38:156–172, 2008.

[BC03] T. J. M. Bench-Capon. Persuasion in practical argument using value-based argumentation frameworks. *Journal of Logic and Computation*, 13(3):429–448, 2003.

[BCA09] T. J. M. Bench-Capon and K. Atkinson. Abstract argumentation and values. In L. Rahwan and G. Simari, editors, *Argumentation in Artificial Intelligence*. Springer, 2009.

[BCD07] T. J. M. Bench-Capon and P. E. Dunne. Argumentation in artificial intelligence. *Artificial Intelligence*, 171:619–641, 2007.

[BCP06] T. J. M. Bench-Capon and H. Prakken. Justifying actions by accruing arguments. *Frontiers in Artificial Intelligence and Applications*, 144:247, 2006.

[BCV97] T. J. M. Bench-Capon and P. R. Visser. Ontologies in legal information systems: The need for explicit specifications of domain conceptualisations. In *Proceedings of International Conference on Artificial Intelligence and Law*, 1997.

[Bel56a] R. Bellman. Dynamic programming and Lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767, 1956.

[Bel56b] R. Bellman. Dynamic programming and Lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767, 1956.

[Ber95] D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, MA, 1995.

[BH08] P. Besnard and A. Hunter. *Elements of argumentation*, volume 47. MIT press Cambridge, 2008.

[BM03] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Applications*, 13:343–379, 2003.

[BRC08] R. Bianchi, C. Ribeiro, and A. Costa. Accelerated autonumous learning by using heuristic selection of actions. *Journal of Heuristics*, 14:135–168, 2008.

[Bre94] G. Brewka. A reconstruction of Rescher's theory of formal disputation based on default logic. In *Proceedings of European Conference on Artificial Intelligence*, pages 366–366. PITMAN, 1994.

[BS01] J. A. Bagnell and J. G. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 2, pages 1615–1620. IEEE, 2001.

[BT03] R. I. Brafman and M. Tennenholtz. R-Max - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 2:213–231, 2003.

[CB98] C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of National Conference on Artificial Intelligence*, 1998.

[CENC03] J. Contreras, R. Espinola, F. J. Nogales, and A. J. Conejo. Arima models to predict next-day electricity prices. *IEEE Transactions on Power Systems*, 18:1014 – 1020, 2003.

[CLR+01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[CML00] C. I. Chesnevar, A.G. Maguitman, and R.P. Loui. Logical models of argument. *ACM Computing Surveys*, 32:337–383, 2000.

[CN89] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[CPG12] M. Carvalho, C. Perez, and A. Granados. An adaptive multi-agent-based approach to smart grids control and optimization. *Energy Systems*, 3(1):61–76, 2012.

[CT91] K. Currie and A. Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[CVRH12] B. J. Claessens, S. Vandael, F. Ruelens, and M. Hommelberg. Self-learning demand side management for a heterogeneous cluster of devices with binary control actions. In *Innovative Smart Grid Technologies, IEEE PES International Conference and Exhibition on*, pages 1–8. IEEE, 2012.

[DGK11] S. Devlin, M. Grzes, and D. Kudenko. An empirical study of potential-based reward shaping and advice in complex, multi-agent systems. *Advances in Complex Systems*, 14:251–278, 2011.

[DGWW11] W. Dvořák, S. A. Gaggl, J. Wallner, and S. Woltran. Making use of advances in answer-set programming for abstract argumentation systems. In *Proceedings of Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management*, 2011.

[Die98] T. G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of International Conference on Machine Learning*, 1998.

[Die00] T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[DKT09] P. M. Dung, R. A. Kowalski, and F. Toni. Assumption-based argumentation. In *Argumentation in Artificial Intelligence*, pages 199–218. Springer, 2009.

[DMT07] P. M. Dung, P. Mancarella, and F. Toni. Computing ideal sceptical argumentation. *Artificial Intelligence*, 171(10):642–674, 2007.

[Dom06] P. Domingos. Whats missing in AI: The interface layer. *Artificial Intelligence: The First Hundred Years. AAAI Press*, 2006.

[DT09] P. M. Dung and P. M. Thang. A unified framework for representation and development of dialectical proof procedures in argumentation. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 746–751, 2009.

[DT10] P. M. Dung and P. M. Thang. Towards (probabilistic) argumentation for jury-based dispute resolution. In *Proceedings of International Conference on Computational Models of Argument*, pages 171–182, 2010.

[Dun95] P. M. Dung. On the acceptability of arugments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.

[DW09] P. E. Dunne and M. Wooldridge. Complexity of abstract argumentation. In *Argumentation in Artificial Intelligence*, pages 85–104. Springer, 2009.

[EGW08] U. Egly, S. A. Gaggl, and S. Woltran. ASPARTIX: Implementing argumentation frameworks using answer-set programming. In *Logic Programming*, pages 734–738. Springer, 2008.

[EK89] K. Eshghi and R. A. Kowalski. Abduction compared with negation as failure. In *Proceedings of Sixth International Conference on Logic Programming*, 1989.

[Eve02] B. S. Everitt. *The Cambridge dictionary of statistics*. Cambridge Unversity Press, 2002.

[FCS+13] X. Fan, R. Craven, R. Singer, F. Toni, and M. Williams. Assumption-based argumentation for decision-making with preferences: A medical case study. In *Proceedings of Fourteenth Workshop on Computational Logic in Multi-Agent Systems*, 2013.

[FHN72] R. Fikes, P. Hart, and N. Nilsson. Some new directions in robot problem solving. *Machine Intelligence*, 7:405–430, 1972.

[FN72] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1972.

[FTMW14] X. Fan, F. Toni, A. Mocanu, and M. Williams. Multi-agent decision making with assumption-based argumentation. In *Proceedings of Thirteenth International Conference on Autonomous Agents and Multiagent Systems*, 2014.

[FW09] W. Faber and S. Woltran. Manifold answer-set programs for meta-reasoning. In *Logic Programming and Nonmonotonic Reasoning*, pages 115–128. Springer, 2009.

[GK08] M. Grzes and D. Kudenko. Plan-based reward shaping for reinforcement learning. In *Proceedings of Fourth IEEE Conference 'Intelligent Systems'*, 2008.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of International Conference on Logic Programming*, volume 88, pages 1070–1080, 1988.

[GLP02] C. Guestrin, M. Lagoudakis, and R. Parr. Coordinated reinforcement learning. In *Proceedings of Nineteenth International Conference on Machine Learning*, 2002.

[GMM06] M. Ghavamzadeh, S. Mahadevan, and R. Makar. Hierarchical multi-agent reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 13(2):197–229, 2006.

[GQS10] J. Gil-Quijano and N. Sabouret. Prediction of humans activity for learning the behaviors of electrical appliances in an intelligent ambient environment. In *Proceedings of IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2010.

[GT13] Y. Gao and F. Toni. Argumentation accelerated reinforcement learning for robucup keepaway-takeaway. In *Proceedings of Second International Workshop on Theory and Applications of Formal Argumentation*, 2013.

[GT14a] Y. Gao and F. Toni. Argumentation accelerated reinforcement learning for cooperative multi-agent systems. In *Proceedings of Twenty-First European Conference on Artificial Intelligence*, 2014.

[GT14b] Y. Gao and F. Toni. Argumentation-based reinforcement learning for robocup soccer takeaway (extended abstract). In *Proceedings of Thirteenth International Conference on Autonomous Agents and Multiagent Systems*, 2014.

[GTC12] Y. Gao, F. Toni, and R. Craven. Argumentation-based reinforcement learning for robocup soccer keepaway. In *Proceedings of Twentith European Conference on Artificial Intelligence*, 2012.

[Hay86] A. J. Hayter. The maximum familywise error rate of Fisher's least significant difference test. *Journal of the American Statistical Association*, 81(396):1000–1004, 1986.

[HW98] J. Hu and M. P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *Proceedings of Fifteenth International Conference on Machine Learning*, 1998.

[IE08] A. Iscen and U. Erogul. A new perspective to the keepaway soccer: The takers (short paper). In *Proceedings of Seventh International Conference on Autonomous Agents and Multiagent Systems*, 2008.

[Irw89] T. Irwin. *Aristotle's first principles*. Oxford University Press, 1989.

[Jor99] M. I. Jordan, editor. *Learning in Graphical Models*. MIT Press, Cambridge, MA, USA, 1999.

[JS08] N. Jong and P. Stone. Hierarchical model-based reinforcement learning: R-Max + MAXQ. In *Proceedings of Twenty-Fifth International Conference on Machine Learning*, 2008.

[KK14] J. Kelly and W. Knottenbelt. 'UK-DALE': A dataset recording uk domestic appliance-level electricity demand and whole-house demand. In *arXiv:1404.0284*, 2014.

[KKT92] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic Computing*, 6:719–770, 1992.

[KM03] A. Kakas and P. Moraïtis. Argumentation based decision making for autonomous agents. In *Proceedings of Second International Conference on Autonomous Agents and Multiagent Systems*, 2003.

[KMA97] A. I. Kostrikin, Y. I. Manin, and M. E. Alferieff. *Linear algebra and geometry*. Gordon and Breach Science Publishers, 1997.

[Kno90] C. A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of Eighth National Conference on Artificial Intelligence*, pages 923–928, 1990.

[Koh03] J. Kohlas. Probabilistic argumentation systems: A new way to combine logic with probability. *Journal of Applied Logic*, 1(3):225–253, 2003.

[Kor85] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[KS80] R. Kindermann and J. L. Snell. *Markov random fields and their applications*, volume 1. American Mathematical Society Providence, RI, 1980.

[LCL11] N. Li, L. Chen, and S. H. Low. Optimal demand response based on utility maximization in power networks. In *Proceedings of IEEE Power and Energy Society General Meeting*, 2011.

[Lif02] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1):39–54, 2002.

[Lit96] M. L. Littman. *Algorithms for sequential decision making*. PhD thesis, Brown University, 1996.

[LLH12] Q. P. Lau, M. L. Lee, and W. Hsu. Coordination guided reinforcement learning. In *Proceedings of Eleventh International Conference on Autonomous Agents and Multiagent Systems*, 2012.

[LZL+12] X. Li, C. Zang, W. Liu, P. Zeng, and H. Yu. Metropolis criterion based fuzzy qlearning energy management for smart grids. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 10(8):1956–1962, 2012.

[Mac01] A. MacIntyre. *Dependent Rational Animals: Why Human Beings Need the Virtues*. Open Court, 2001.

[Mar07] B. Marthi. Automatic shaping and decomposition of reward shaping. In *Proceedings of Twenty-Fourth International Conference on Machine Learning*, pages 601–608, 2007.

[MC09] S. Modgil and M. Caminada. Proof theories and algorithms for abstract argumentation frameworks. In *Argumentation in Artificial Intelligence*, pages 105–129. Springer, 2009.

[Min01] J. Minker, editor. *Logic-Based Artificial Intelligence*. Springer, 2001.

[Mod09] S. Modgil. Reasoning about preferences in argumentation frameworks. *Artificial Intelligence*, 173:901–934, 2009.

[MS11] H. Mercier and D. Sperber. Why do human reason? Arguments for an argumentative theory. *Behavioral and Brain Sciences*, 34:57–111, 2011.

[MWS95] W. T. Miller, P. J. Werbos, and R. S. Sutton. *Neural networks for control*. MIT press, 1995.

[MZB07] M. Mozina, J. Zabkar, and I. Bratko. Argument based machine learning. *Artificial Intelligence*, 171:922–937, 2007.

[MZCZ08] H.-Q. Min, J.-A. Zeng, J. Chen, and J.-H. Zhu. A study of reinforcement learning in a new multiagent domain. In *Proceedings of IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2008.

[NCD+06] A. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Autonomous inverted helicopter flight via reinforcement learning. *Experimental Robotics IX*, pages 363–372, 2006.

[NCO08] J. C. Nieves, U. Cortés, and M. Osorio. Preferred extensions as stable models. *Theory and Practice of Logic Programming*, 8(04):527–543, 2008.

[NHR99] A. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: theory and application to reward shaping. In *Proceedings of Sixteenth International Conference on Machine Learning*, pages 278–287, 1999.

[Nil94] N. J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[OGR+12] M. A. Osborne, R. Garnett, S. J. Roberts, C. Hart, S. Aigrain, and N. Gibson. Bayesian quadrature for ratios. In *Proceedings of International Conference on Artificial Intelligence and Statistics*, pages 832–840, 2012.

[OLGM10] D. O'Neill, M. Levorato, A. Goldsmith, and U. Mitra. Residential demand response using reinforcement learning. In *Proceedings of IEEE International Conference on Smart Grid Communications*, 2010.

[PB80] C. Perelman and H. J. Berman. *Justice, law and argument: essays on moral and legal reasoning*. Reidel, 1980.

[Pea88] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[PGWR12] O. Parson, S. Ghosh, M. Weal, and A. Rogers. Non-intrusive load monitoring using prior models of general appliance types. In *Proceedings of Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[Pol87] J. L. Pollock. Defeasible reasoning. *Cognitive Science*, 11:481–581, 1987.

[Pol91] J. L. Pollock. A theory of defeasible reasoning. *International Journal of Intelligence Systems*, 6:33–54, 1991.

[Poo97] D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1):7–56, 1997.

[PR97] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Proceedings of Neural Information Processing Systems*, 1997.

[PS07] H. Prakken and G. Sartor. A dialectical model of assessing conflicting arguments in legal reasoning. In *Proceedings of the Eleventh International Conference on Articial Intelligence and Law*, 2007.

[RA98] J. Randlov and P. Alstrom. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of Fifteenth International Conference on Machine Learning*, pages 463–471, 1998.

[RD06] M. Richardson and P. Domingos. Markov logic networks. *Machine learning*, 62(1-2):107–136, 2006.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[RN94] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, University of Cambridge, 1994.

[RN09] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[ROP+13] S. D. Ramchurn, M. A. Osborne, O. Parson, T. Rahwan, S. Maleki, S. Reece, T. D. Huynh, M. Alam, J. E. Fischer, T. Rodden, L. Moreau, and S. Roberts. AgentSwitch: Towards smart energy tariff selection. In *Proceedings of International Conference on Autonomous Agents and Multiagent Systems*, 2013.

[RS09] I. Rahwan and G. R. Simari, editors. *Argumentation in Artificial Intelligence*. Springer, 2009.

[Sac74] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[SB98] R. S. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Sin92] S. P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of Tenth National Conference on Artificial Intelligence*, pages 202–207, 1992.

[SL92] G. R. Simari and R. P. Loui. A mathematical treatment of defeasible reasoning and its implementation. *Artificial Intelligence*, 53(2):125–157, 1992.

[SPS99] R. S. Sutton, D. Precup, and S. Singh. Between MDPs and SMDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[SS96] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

[SSK05] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13:165–188, 2005.

[Thr92] S. B. Thrun. Efficient exploration in reinforcement learning. 1992.

[TMTT+13] N. C. Truong, J. McInerney, L. Tran-Thanh, E. Costanza, and S. D. Ramchurn. Forecasting multi-appliance usage for smart home energy management. In *Proceedings of Thirteenth International Joint Conference on Artificial Intelligence*, 2013.

[Tou58] S. Toulmin. *The Uses of Arguments*. Cambridge University Press, 1958.

[TS+01] B. G. Tabachnick, L. Fidell S, et al. *Using multivariate statistics*. Allyn and Bacon Boston, 2001.

[TS09] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685, 2009.

[TS11] F. Toni and M. Sergot. Argumentation and answer set programming. In *Logic programming, knowledge representation, and nonmonotonic reasoning*, pages 164–180. Springer, 2011.

[Tuk49] J. W. Tukey. Comparing individual means in the analysis of variance. *Biometrics*, pages 99–114, 1949.

[Vre05] G. Vreeswijk. Argumentation in Bayesian belief networks. In *Argumentation in Multi-Agent Systems*, pages 111–129. Springer, 2005.

[Wal89] D. N. Walton. Dialogue theory for critical thinking. *Argumentation*, 3(2):169–184, 1989.

[WCE03] E. Wiewiora, G. Cottrell, and C. Elkan. Principled methods for advising reinforcement learning agents. In *Proceedings of Twenth International Conference on Machine Learning*, 2003.

[WD92] C. Watkins and P. Dayan. Q-Learning. *Machine learning*, 8.3-4:279–292, 1992.

[WN09] T. Wakaki and K. Nitta. Computing argumentation semantics in answer set programming. In *New Frontiers in Artificial Intelligence*, pages 254–269. Springer, 2009.