
Contents

1	Introduction	1
1.1	Motivation and Research Questions	7
1.2	Research Contributions	10
1.3	Thesis Outline	12
2	Related Work	15
2.1	Reinforcement Learning for Computer Game AI	16
2.1.1	Q-Learning	18
2.1.2	Sarsa	20
2.2	Case-Based Reasoning and Hybrid Approaches	21
2.3	Hierarchical Approaches and Layered Learning	27
2.4	StarCraft as a Testbed for AI Research	30
2.5	Summary	32
3	Background	33
3.1	Real-time Strategy Games and StarCraft as Testbeds for AI Research	33
3.1.1	Characteristic Traits of RTS games and their Relevance to AI Research	34
3.1.2	RTS Games as Testbeds for AI Research	37
3.1.3	StarCraft as a Domain for AI Research	41
3.2	RTS Game Bot Architectures	46
3.3	Reinforcement Learning	48
3.3.1	Origins of Reinforcement Learning	48
3.3.2	Reinforcement Learning Algorithms	49
3.4	Case-Based Reasoning	54
3.4.1	Histogram-Based Similarity Computation for IMs	57
3.4.2	Hausdorff Distance for Similarity Computation	58
3.5	Layered and Hierarchical Learning	60

4	Reinforcement Learning for Strategy Game Unit Micromanagement	63
4.1	Reinforcement Learning Model	64
4.1.1	Reinforcement Learning States	64
4.1.2	Reinforcement Learning Actions	65
4.1.3	Transition Probabilities	66
4.1.4	Reinforcement Learning Reward Signal	66
4.2	Algorithm	66
4.3	Empirical Evaluation and Results	68
4.3.1	Experimental Setup	68
4.3.2	Results and Discussion	69
4.4	Conclusions on the Future Use of RL for Micromanagement in RTS Games	75
5	Combining Reinforcement Learning and Case-Based Reasoning for Strategy Game Unit Micromanagement	77
5.1	CBR/RL Agent Architecture	78
5.1.1	Case-Based Reasoning Component	78
5.1.2	Reinforcement Learning Component	81
5.2	Model	81
5.3	Empirical Evaluation and Results	83
5.3.1	Experimental Setup and Parameter Optimization	84
5.3.2	Performance	86
5.3.3	Case Base Development and State-Action Space Exploration	87
5.4	Discussion	88
5.5	Conclusion and Influence on Hierarchical Approach	90
6	A Hybrid Hierarchical CBR/RL Architecture for RTS Game Micromanagement	93
6.1	Modeling a Hierarchical CBR/RL Architecture in a RTS Game	94
6.2	Evaluating the Hierarchical Architecture	98
6.3	Unit Mapping	100
6.4	Summary	102
7	Architecture Level Three: Unit Pathfinding using Hybrid CBR/RL	103
7.1	CBR/RL Integration and Model	105
7.1.1	Navigation States: Case Description	106
7.1.2	Navigation Actions	108
7.1.3	Navigation Reward Signal	108
7.2	Similarity Computation and Navigation Module Logic	109
7.3	Empirical Evaluation and Results	111

7.4	Navigation Discussion	114
7.5	Training the Navigation Case-Base	116
8	Architecture Level Two: Squad-Level Coordination	119
8.1	Unit Formations	120
8.1.1	Unit Formations in StarCraft	121
8.1.2	Formation Solution Case-Base	124
8.1.3	Unit Formation Model	126
8.1.4	Formation Evaluation and Training	129
8.1.5	Formation Results	136
8.1.6	Formation Discussion	139
8.1.7	Effects of Using a Solution Case-Base	140
8.2	Unit Attack	142
8.2.1	Unit Attack Model	143
8.2.2	Attack Evaluation and Training	146
8.2.3	Initial Attack Results and Discussion	150
8.2.4	Additional Attack Training Scenarios	151
8.3	Unit Retreat	153
8.4	Summary	155
9	Architecture Level One: Tactical Decision Making	157
9.1	Tactical Decision Making Model	159
9.1.1	States	160
9.1.2	Tactical Case Similarity	161
9.1.3	Actions	163
9.1.4	Reward Signal	164
9.1.5	State Transitions	165
9.2	Overall Hierarchical CBR/RL Algorithm	165
9.3	Tactical Decision Making Evaluation	168
9.4	Results	172
9.5	Discussion	176
9.6	Knowledge Transfer between Scenarios	180
10	Discussion and Future Work	185
11	Conclusion	197
A	Database Diagrams	201

Contents

B Munkres Assignment Algorithm	205
C Algorithm Parameter Optimization	207

List of Symbols

Name	Symbol	Description
Discount Rate	γ	The discount rate decides how important projected future rewards are in the computation of state- or state-action values in a temporal difference reinforcement learning algorithm
Exploration Rate	ϵ	The probably of choosing a random action in ϵ -greedy exploration policies.
Learning Rate	α	The learning rate decides to what degree newly gained knowledge on a state- or state-action value replaces old knowledge.
Optimal Action-Value Function	Q^*	The optimal action-value function for a given reinforcement learning algorithm.
Optimal Policy	π^*	The optimal policy for a given reinforcement learning algorithm.
Similarity Threshold	ψ	The similarity threshold that is used to determine when new cases in the case-base are created when using a CBR approach.
Trace Decay Parameter	λ	The parameter determines how far rewards propagate back through a series of states and actions

List of Acronyms

ABL A Behavior Language.

AI Artificial Intelligence.

BWAPI Broodwar API.

CBP Case-Based Planning.

CBR Case-Based Reasoning.

CBR Trace-Based Reasoning.

FPS First-Person Shooter.

GA Genetic Algorithm.

GDA Goal-Driven Autonomy.

GOAP Goal Oriented Action Planning.

GP Genetic Programming.

HRL Hierarchical Reinforcement Learning.

HTN Hierarchical Task Network.

IM Influence Map.

kNN k-Nearest Neighbour.

LGDA Learning Goal-Driven Autonomy.

LL Layered Learning.

MCTS Monte-Carlo Tree Search.

List of Acronyms

MDP Markov Decision Process.

ML Machine Learning.

NEAT NeuroEvolution of Augmented Topologies.

NN Neural Network.

ORTS Open-Source RTS Environment.

PF Potential Field.

RDBMS Relational Database Management System.

RETALIATE Reinforced Tactic Learning in Agent-Team Environments.

RL Reinforcement Learning.

RPG Role-Playing Game.

RTS Real-Time Strategy.

SMDP Semi-Markov Decision Process.

TD Temporal Difference.

TIELT Testbed for Integrating and Evaluating Learning Techniques.

UCT Upper Confidence Bounds Applied to Trees.

List of Tables

1.1	Number of Publications on RTS Game AI ordered by Tasks and Techniques Lara-Cabrera et al. (2013)	3
3.1	Bot Architecture AI Techniques	47
4.1	Reinforcement Learning Evaluation Parameters	69
5.1	Evaluation Parameters	85
5.2	Case Base Statistics	88
6.1	Unit Assignment Example	101
7.1	Navigation Case-Base Summary	111
7.2	Navigation Evaluation Parameters	112
7.3	Additional Pathfinding Training Scenarios	117
8.1	Number of Possible Solutions for Assigning n Units to n Formation Slots	123
8.2	Solution Similarities	125
8.3	Formation State Case Description	127
8.4	Example Formation Slot Assignments	128
8.5	Formation Evaluation and Training Scenarios	134
8.6	Formation Evaluation Parameters	135
8.7	Number of Recorded Solutions vs Number of Possible Solutions by Scenario and Unit Number	141
8.8	Attack State Case Description	145
8.9	Attack Evaluation Parameters	150
8.10	Attack Training Scenario Parameters	151
8.11	Attack Case-Base after Training	152
9.1	Possible Tactical Solutions for $n = 5$ Units	159
9.2	Tactical State Case Description	160
9.3	Tactical Unit Attribute Similarity Computation	162

List of Tables

9.4	Tactical Decision Making Evaluation Parameters	170
9.5	Tactical Decision Making Evaluation Scenarios	170
9.6	Tactical Decision Making Evaluation Scenario A	172
9.7	Tactical Decision Making Evaluation Scenario B	172
9.8	Tactical Actions and Duration for all Scenarios	175
9.9	Cases per Agent and Opponent Unit Numbers for <i>Scenario E</i> for $\psi = 80\%$.	181
9.10	Cases per Unit Numbers for <i>Scenario C</i>	181
9.11	Knowledge Transfer Evaluation Scenarios	182

List of Figures

1.1	Number of Annual Publications mentioning ‘Starcraft Game AI’ compared to those mentioning ‘Real Time Strategy Game AI’	3
3.1	RTS Game Layers and Tasks	36
3.2	The CBR Cycle	56
3.3	Case Retrieval	56
3.4	Computing the non-directed Hausdorff Distance between Sets A and B	59
4.1	StarCraft RL Algorithm Integration Initial Run	67
4.2	StarCraft RL Algorithm Integration General Run	67
4.3	Initial unit positioning for the experimental evaluation	68
4.4	Development of Total Reward for 1,000 Episodes Played	70
4.5	Percentage of Games Won for 1,000 Episodes Played	70
4.6	Development of Total Reward for 500 Episodes Played	71
4.7	Percentage of Games Won for 500 Episodes Played	71
4.8	Standard Deviation for 1,000 Episodes Played	72
4.9	Standard Deviation for 500 Episodes Played	72
5.1	Case Retrieval Process	79
5.2	Excerpt of the Influence Map	79
5.3	Logical Structure of Cases and the Information they contain	80
5.4	Results for 50 Game Runs	86
5.5	Results for 500 Game Runs	87
5.6	Results for 1000 Game Runs	87
6.1	RTS Micromanagement Tasks	95
6.2	Levels, Information Abstraction and Action Scope of our Architecture	96
6.3	Hierarchical Structure of the Case-Bases	97
7.1	Navigation in the Context of RTS Tasks	104
7.2	Game Situation with Influence Map Overlay	107

List of Figures

7.3	Possible Movements for a Unit	108
7.4	Example of IM Field Similarity Computation	110
7.5	Results for the Damage Avoidance Scenario	113
7.6	Results for the Target Approximation Scenario	113
7.7	Results for the Combined Navigation Scenario	114
8.1	Fixed Formation Example	121
8.2	Dynamic Formation Example	122
8.3	Agent Formation Layout and Parametrisation	122
8.4	Formation Solution Similarity Example	125
8.5	Formation Unit Positions	128
8.6	Randomised Formation Scenario	131
8.7	Example Desired Behaviour for Scenario B	132
8.8	Example of Desired Behaviour in Scenario C	133
8.9	Example Desired Behavior for Scenario D	133
8.10	Results for Scenario A	136
8.11	Results for Scenario B	137
8.12	Results for Scenario C	138
8.13	Results for Scenario D	138
8.14	Average Distance from Opponent Units to Attackers	145
8.15	Randomised Attack Scenario	148
8.16	Results for Attack Scenario A	150
8.17	Retreat Destination Computation Based on IM Values	154
9.1	Action Selection using Hierarchical CBR/RL for Unit Micromanagement	166
9.2	Reward Computation using Hierarchical CBR/RL for Unit Micromanagement	167
9.3	Performance Results for Scenario A for different ψ	173
9.4	Performance Results for Scenario B for different ψ	174
9.5	Performance Results for all Scenarios	175
9.6	Performance Results for <i>Scenario F</i> without and with existing knowledge	183
A.1	DB Diagram Level 3	202
A.2	DB Diagram Level 2	203
A.3	DB Diagram Level 1	204
B.1	Munkres Assignment Algorithm Steps	205

List of Algorithms

1	A Simple TD Algorithm for Estimating V^π	51
2	Pseudocode for One-Step Q-Learning	51
3	Pseudocode for One-Step Sarsa	52
4	Pseudocode for TD(λ)	53
5	Pseudocode for Sarsa(λ)	54
6	Pseudocode for Watkins's Q(λ)	55

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 4 is extracted from "Wender, S., & Watson, I. (2012). Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game StarCraft:Broodwar. In 2012 IEEE Symposium on Computational Intelligence and Games (CIG2012)"

Nature of contribution by PhD candidate

Extent of contribution by PhD candidate (%)

CO-AUTHORS

Name	Nature of Contribution
Ian Watson	Supervision + editing

Certification by Co-Authors

The undersigned hereby certify that:

- ❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- ❖ in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

Name	Signature	Date
Ian Watson		Click here
		Click here
		Click here
		Click here
		Click here
		Click here

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 5 is extracted from "Wender, S., & Watson, I. (2014). Integrating Case-Based Reasoning with Reinforcement Learning for Real-Time Strategy Game Micromanagement. In PRICAI 2014: Trends in Artificial Intelligence (pp. 64–76). Springer"

Nature of contribution by PhD candidate

Work & writing

Extent of contribution by PhD candidate (%)

90

CO-AUTHORS

Name	Nature of Contribution
Ian Watson	Supervision + editing

Certification by Co-Authors

The undersigned hereby certify that:

- ❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- ❖ in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

Name	Signature	Date
Ian Watson		Click here
		Click here
		Click here
		Click here
		Click here
		Click here

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter 7 is extracted from "Wender, S., & Watson, I. (2014). Combining Case-Based Reasoning and Reinforcement Learning for Unit Navigation in Real-Time Strategy Game AI. In L. Lamontagne & E. Plaza (Eds.), Case-Based Reasoning Research and Development (Vol. 8765, p. 511-525). Springer International Publishing."

Nature of contribution by PhD candidate

Work & writing

Extent of contribution by PhD candidate (%)

90

CO-AUTHORS

Name	Nature of Contribution
Ian Watson	Supervision + editing

Certification by Co-Authors

The undersigned hereby certify that:

- ❖ the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- ❖ in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

Name	Signature	Date
Ian Watson		Click here
		Click here
		Click here
		Click here
		Click here
		Click here

The goal of creating an artificial human-level intelligence has always triggered lots of interest in the area of artificial intelligence (AI) research from both novices and experts alike. However, the reality is that the research community is still far removed from the elusive human-like artificial general intelligence (AGI). AI is still focused on narrow areas of specialised application. Although some landmark developments and products such as Apple's *Siri* (Aron, 2011) or IBM's *Watson* (Ferrucci et al., 2010) have made their way into everyday use and trigger the imagination of the public, a closer examination leads to a quick realisation that while these are often groundbreaking applications, there is still only the illusion of human intelligence behind the approaches. However, despite not having general human-level intelligence, these applications still advance the overall field through creating fascinating real-world applications by applying advanced AI research.

An area that has always been at the forefront of interesting AI utilization is games. Even in the early days of the field, games provided a tangible, fascinating area that allowed the application of existing AI techniques and led to the development of advanced algorithms and methods (Samuel, 1959). Games provide a fertile breeding ground for new approaches and an interesting and palpable test area for existing ones. And as games such as checkers and chess are devised as high-level abstractions of mechanisms and processes in the real world, creating AI that works in these games can eventually lead to AI that solves real-world problems. With the advent of personal computers, games have evolved from their highly abstracted board- and card-game origins into increasingly complex simulations that move ever closer to the thing they were originally created to abstract, the real world.

One of the most popular genres of computer video games is real-time strategy (RTS). RTS is a genre of computer video games in which players perform simultaneous actions while competing against each other using combat units. Often, RTS games include elements of base building, resource gathering and technological developments and players have to carefully balance expenses and high-level strategies with lower-level tactical reasoning. RTS games incorporate many different elements and are related to areas such as robotics and military simulations. RTS games can be very complex and, especially given the real-time

aspect, hard to master for human players (see Section 3.1). Since they bear such a close resemblance to many real-world problems, creating powerful AI in an RTS game can lead to significant benefits in addressing those related real-world tasks.

A long-term objective in many avenues of game AI research, just as in other areas of AI, is the creation of agents that play human-like, both in terms of raw performance and in terms of in-game strategies and behaviour. The seminal paper by Laird & van Lent (2001), which started a lot of interest in the area of interactive computer game AI research, specifically mentioned the creation of **human-level AI** as the goal in that environment, and there are constant attempts at creating game-playing agents that resemble human players in many ways. The *BotPrize* targets human-level AI in first-person shooter (FPS) games. This is a competition that seeks to create AI bots that are indistinguishable in style of play from human players (Cothran & Champanand, 2009), basically a Turing Test for FPS games. FPS games are the most popular genre of games among players (ESA, 2011) and research in the area has yielded impressive results (Cothran & Champanand, 2009). However, RTS game AI research has also made some great strides over the last decade and grown into one of the most prominent areas of simulation-based AI research. In particular, the creation of the *Broodwar API (BWAPI)* interface that gave full access to the popular commercial RTS game *StarCraft* led to a surge in RTS game AI research (BWAPI, 2009). *StarCraft* proved a booster to the whole area and became the most-used testbed within a short time-frame. This development is reflected in Figure 1.1, which gives an indication of both the rise of popularity of research in RTS game AI and research using *StarCraft* by looking at publications using the respective key terms¹.

The topic of this thesis is the creation of an agent that focuses on the tactical and reactive tasks in RTS games, the so-called ‘micromanagement’. While tasks related to RTS game micromanagement such as navigation, pathfinding and creating unit formations have been the focus of AI research for a long time, research involving micromanagement in RTS games only recently received more attention. Lara-Cabrera et al. (2013) reviewed research in RTS game AI according to problem area and the algorithmic approaches that are used. The authors’ findings on publications per task area, summarized in Table 1.1, show that there is still a considerably larger body of work that focuses on high-level **Planning** tasks when compared to **Unit Maneuvering**. Additionally, when looking at the publication dates, the

¹ The diagram was created by retrieving the number of publications listed by *Google Scholar* for each of the years shown. The graphic only serves as an indicator of the general trend, not as a list of the absolute numbers of publications in which the respective environments are used as a testbed. BWAPI was only created in 2009 and a *StarCraft* competition at a research conference first happened in 2010, which led to a wider adoption of *StarCraft* as a testbed for AI research. Therefore, the large number of mentions before 2009 is mostly due to mentions of *StarCraft* as a groundbreaking RTS game.

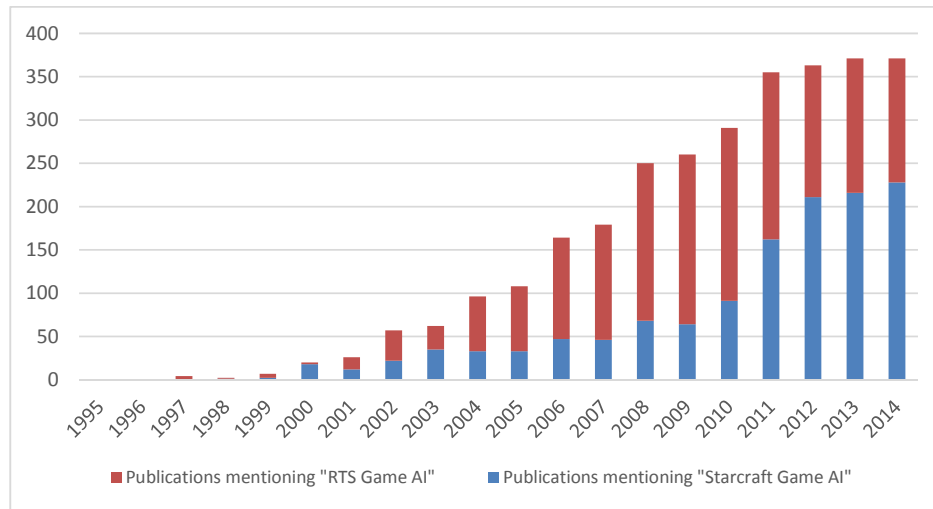


Figure 1.1: Number of Annual Publications mentioning ‘Starcraft Game AI’ compared to those mentioning ‘Real Time Strategy Game AI’

average date for publications falling into the former category is early 2007 compared to late 2009 for publications on **Unit Maneuvering**. While Lara-Cabrera et al. (2013) do not examine every publication on RTS game AI, this is indicative of the aforementioned shift in research focus, or rather an extension of research interests to also include lower-level tasks.

	Planning	Unit Maneuvering	Opponent Matching	Partial Observability	Plan Recognition	Procedural Content Generation	Difficulty Adjustment
CBR and/or RL	IIIIIII (8)	II (2)			III (3)		
AI Planning	IIIIIII (8)						
Influence and Potential Maps		IIIIIII (9)		I (1)	I (1)		
Evolutionary	IIIIIIIII (11)	IIIIIII (7)	I (1)			IIIII (6)	II (2)
Simulations	III (4)			I (1)			
Dynamic Scripting	III (4)						
ANN	II (2)	I (1)	II (2)				I (1)
Fuzzy/Bayesian Models		I (1)			III (4)		

Table 1.1: Number of Publications on RTS Game AI ordered by Tasks and Techniques Lara-Cabrera et al. (2013)

One reason why strategy-level approaches were more interesting in the early stages of RTS game AI research is the different requirements in terms of accessibility of the testbed. Strategic tasks work with high levels of abstraction, both in terms of knowledge representation and the time actions take to execute and take effect. In contrast, tactical and especially reactive reasoning require a considerably higher level of precision for action execution and knowledge representation. The real-time constraint plays a much larger role on the reactive layer of the game. The required level of precision is not always provided by the testbed interfaces. Even BWAPI, as the interface to the potentially most popular testbed for game AI research, does not have unlimited capabilities in terms of allowing access to the game engine's functionality. Churchill et al. (2012) described the creation of their *SparCraft* environment, an abstract StarCraft combat simulator. They created this abstract environment since the original BWAPI interface could not provide the necessary performance for their *Monte-Carlo Tree Search (MCTS)* approach.

Similar to other areas of AI and AI in games, there is large interest in creating human-like behaviour in RTS games. In fact, many of the StarCraft bots that play entire games using *Machine Learning (ML)* approaches are created with the expressed goal of creating human-like behaviour. As currently the performance of AI agents in the game is still significantly inferior to that of expert players (Huang, 2011), creating 'human-like' behaviour refers to the attempt at recreating the level that humans perform at. In the future, once RTS game AI performance has surpassed that of human players as it has done in classical board games such as chess (Campbell et al., 2002) and checkers (Schaeffer et al., 2007), the focus could shift towards creating agents which adjust their levels of skill to that of human players in order to create an enjoyable experience for players of any skill level. While adjusting their level of skill is important, this also requires the creation of AI players that play games in a 'human way' for the games to be enjoyable for users who want to perceive their opponents as equals and play according to the same rules. The primary focus in the development of commercial game AI lies in creating enjoyment for players - something which currently, more often than not, involves the creation of a static, beatable AI (Davis, 1999; Johnson, 2008; Robertson & Watson, 2014). However, professional game developers also acknowledge that users are looking for an experience that feels like playing against other human players, a desire which explains the meteoric rise of online games where humans compete against others. Thus, once the capabilities of RTS game AI players reach a point where they can compete with human players as equals, the creation of human-like behaviour traits becomes an important target.

The creation of powerful AI agents that perform well in computer games is made considerably harder by the enormous complexity these games exhibit. The complexity of any board game or computer game is defined by the size of its state- and decision space. A state in chess is defined by the position of all pieces on the board while the possible actions at a certain point are all possible moves for these pieces. Shannon (1950) estimated the number of possible states in chess as 10^{43} . The number of possible states in RTS games is vastly bigger. D. Aha et al. (2005) estimated the decision-complexity of the *Wargus* RTS game (i.e. the number of possible actions in a given state) to be in the 1,000s even for simple scenarios that involve only a small number of units. StarCraft is even more complex than Wargus, with a larger number of different unit types and larger combat scenarios on bigger maps, leading to more possible actions. Weber (2012) estimated the number of possible states in StarCraft, defined through hundreds of possible units for each player on maps that can have maximum dimension of 256×256 tiles, to be in excess of 10^{11500} . In comparison, chess has a decision complexity of about 30.

Human players are good at abstracting these vast state- and decision spaces by automatically excluding nonsensical actions such as scouting with worker units or building their base away from their resources (Weber, 2012). Furthermore, when human players control units by issuing orders such as movement actions, they also automatically extrapolate, over the near future, how these orders are executed, i.e. they predict the movement path. Unless specifically provided with this information, AI agents do not have these capabilities. Therefore, considerable effort has to be made in modelling the game environment so the AI agent can work with it to a suitable degree. Creating usable abstractions is a major challenge encountered throughout this thesis. The three distinct phases of the approach developed as part of this thesis all address the micromanagement problem at different levels of abstraction and, as a result, can solve tasks with varying levels of complexity. The agent developed as part of the third and final step presented in Chapter 6 is designed to address the entire range of reactive and tactical tasks. Thus the approach designed in Chapters 6 is built to most closely resemble human-like capabilities.

In order to achieve ‘human-like’ reasoning for AI agents in RTS games, there first has to be a careful consideration of what constitutes ‘human-like’ performance. Human players are very good at figuring out loopholes and shortcomings in their opponents and subsequently exploiting these. In fact, commercial games are often designed with purpose-built exploitable loopholes that allow players to find a perfect winning strategy, thus giving them a sense of achievement (Johnson, 2008; Robertson & Watson, 2014). Human players are adept at devising new solutions by using the tools at their disposal. Human learning and problem solving are often based on the ability to analyse a problem from all angles and subsequently solving this problem by pursuing a trial-and-error method that attempts different solutions.

This is the essence of *reinforcement learning* (*RL*), the ML technique that is used throughout this thesis (R. S. Sutton & Barto, 1998). This characteristic way of solving problems through trial-and-error builds on the ability to identify the smallest possible way to adjust the solution, something that humans are particularly good at.

Human players also often perform certain actions with a high-level goal in mind, such as surrounding enemy units in a combat, winning a skirmish or achieving control over certain parts of the map. Part of human problem-solving skills is attempting a high-level solution to a problem made up of numerous actions, dissecting it into parts that worked and parts that did not and re-attempting an adapted version of this solution. Given a certain set of available actions, players trial combinations of these actions to find out which combination works best. In this way, human players create complex solutions by concatenating atomic actions in order to achieve high-level goals.

When creating a model for an AI agent that is built to reproduce a similar performance, there are two possibilities in terms of abstraction. The first possibility is to analyse human performance at a comparably high level of abstraction and recreate the high-level actions by hand-coding the low-level atomic actions involved in a **top-down** fashion. However, given the high complexity of RTS games such as StarCraft, there is a very large number of high-level actions that can be created through permuting low-level actions. Even if expert knowledge is used to identify relevant actions of human players this still leads to a very large number of possible high-level actions for the agent to choose from. This method is often used to prove the viability of ML approaches at certain sub-tasks in RTS game micromanagement (Uriarte & Ontañón, 2012; Zhen & Watson, 2013; Micić et al., 2011) or in order to hand-code certain aspects of agents that play the entire game (Ontañón et al., 2013). Often, such an approach then only focuses on a selected subset of high-level actions that are identified as relevant, which means that AI agent actions risk being limited and repetitive when compared to human players.

Alternatively, a lower level of abstraction can be used in the game world model. This leads to a smaller number of lower-level actions that can be combined to create more elaborate high-level actions in a **bottom-up** fashion. The challenge then lies in identifying which of these combinations are ‘good’ or ‘human-like’. The approach pursued in this thesis is inspired by both the **top-down** and the **bottom-up** methods and eventually leads to a combination of both. In Chapter 4, which evaluates the viability of RL algorithms for use in micromanagement, the agent only has a very limited number of high-level actions at its disposal, which it uses to learn a specific behaviour of a single unit: *kiting*. Chapter 5 serves as a proof-of-concept for the use of a combination of RL and *case-based reasoning* (*CBR*) for micromanagement. It pursues a **bottom-up** method by providing the agent with atomic StarCraft *Move* actions that allow it to control multiple units individually but still uses a

high-level *Attack* method. The AI agent successfully manages to learn a *kiting* behaviour for its units while also coordinating them in a non-centralized fashion. This model is limited on the one hand by using very low-level movement that leads to a high degree of complexity even for basic movement patterns. The hand-coded attack on the other hand, limits the AI agent in learning more variable attack patterns. Given a distinctively abstract state representation in addition, the agent using the model defined in Section 5.2 would not be able to learn functionality that is much more complex than what the evaluation in Section 5.3 shows; certainly nothing that resembles the flexibility and dynamics of human players.

The hierarchical approach presented in Chapter 6 addresses these concerns through a combination of **top-down** and **bottom-up** methods. The entire problem is subdivided into several layers with the lowest layer using atomic actions in the game environment and higher-level, more abstract layers building on those lower layers. The aim is to create an agent that has the ability to create effective, human-like actions by learning how to effectively combine actions on the three defined levels of abstractions. One major criterion for the success of the approach is an evaluation of its performance in terms of eliminating an opponent. However, another interesting aspect is the hierarchical agent's ability to learn novel and interesting strategies in situations where there is, unlike the evaluations in Chapters 4 and 5, not an optimal solution. This less-tangible capability would be indicative of a more human-like ability to create novel solutions by reasoning across multiple layers of abstraction.

1.1 Motivation and Research Questions

The goal of this thesis is the creation of an homogeneous machine learning approach that learns how to solve large parts of the tasks involved in the reactive and tactical levels of RTS games. While Chapters 2 and 3 go into more detail of the reasons for choosing this particular domain and using the selected machine learning techniques, this section gives an overview of the underlying motivation of this thesis.

Creating a learning agent that can acquire knowledge on how to play a game on its own can lead to more challenging, dynamic and generally diverse game play. Additionally, having agents that acquire gameplay knowledge themselves makes them easier to create and, provided the agent uses a continuously adapting learning process, easier to maintain. Agents that manage to adjust to changing conditions and situations also appear more human-like (Hsieh & Sun, 2008; Weber et al., 2011).

While RTS game AI research in general and research using the StarCraft RTS game in particular have seen a great increase in popularity in recent years, much of the research still focuses on isolated approaches for the numerous tasks involved in these problem areas. Agent architectures that address the entire gameplay are patchworks of different approaches

for sub-problems (see Section 3.2), often using hard-coded domain knowledge instead of adaptive components for large parts of the problem space. Few attempts have been made to solve the problems in a sub-divided yet holistic way, using an homogeneous machine learning approach for multiple layers of game-play abstraction, and none of them in a way that acquired all necessary knowledge online while playing the game. Using such a unified dynamic approach as proposed in this thesis can lead to hierarchical architecture that addresses vast areas of the RTS game problem space in dynamic fashion while remaining modular enough to be easily extensible for additional tasks and customizable enough to allow adapted solutions for particular problems.

There are two main research questions addressed in this thesis. These main questions in turn lead to several low-level questions.

I *Can machine learning techniques be used to create an AI agent that is able to address reactive and tactical components in a RTS game by acquiring necessary knowledge through online human-like learning processes within the game environment?*

Provided such an approach proves feasible, this leads to the subsequent question on underlying details.

II *For an AI agent that is able to address reactive and tactical components, what are the requirements in terms of architecture and knowledge modelling?*

Attempting to answer these questions leads to an iterative process that asks smaller, more technical questions. Answering the smaller questions then allows identification of potential solutions.

1. *Given its similarity to the human learning process and its ability to find optimal policies in unknown environments, can reinforcement learning achieve the desired results of learning how to perform in the domain of RTS micromanagement?* Answering this question provides an evaluation of the potential of applying reinforcement learning in the RTS domain and also provides an indication of which RL algorithms work best. It furthermore suggests what kinds of adjustments have to be made to enable efficient and effective learning for this task. Additionally, investigating RL algorithms in the context of RTS game micromanagement points out the limitations of the technique in this domain, which in turn leads to suggestions as to how these limitations can be overcome. This question is addressed in Chapter 4.
2. *If RL is not able to solve the problem on its own, does the creation of a hybrid approach that uses another ML technique to improve the agent's capabilities lead to a solution?*

This question builds on findings from (1.) which include a number of limitations for the proposed technique. Discovering these limitations in turn lead to suggested potential improvements. In order to offset the shortcomings and get closer to addressing the entire micromanagement problem, CBR is integrated with the RL approach and used as the ‘memory’ of the agent while also serving as a generalization technique. This question is addressed in Chapter 5.

3. *When using a combination of RL and CBR, how can the problem as well as the relevant elements of the simulation environment be represented so that a ML agent can adequately address the micromanagement tasks using this hybrid technique?* This question is based on results from (2.) which indicate requirements for using a hybrid CBR/RL approach for large-scale problems. The identified requirements and the initial research question lead to an investigation of the sub-problems involved in RTS game micromanagement in order to answer this question. This investigation not only focuses on the specific RTS game that is used as a testbed throughout this thesis, but also expresses the problem in a way general enough to also encompass other RTS and combat simulations and even different but related areas of research, such as robotics. Subsequently, a hierarchical architecture is created that addresses the relevant sub-problems found during this investigation in individual, inter-connected modules. This question is addressed in Chapter 6.
4. *In a hierarchical modular architecture that addresses reactive and tactical tasks which are part of RTS games, what are the challenges involved in addressing relevant sub-tasks through hybrid CBR/RL?* As part of the answer to this question, the specific sub-tasks identified in (3.) are converted into individual CBR/RL modules. Each module addresses a particular sub-task relevant to the overall micromanagement problem. Each of the sub-tasks that is identified as relevant to the overall problem comes with its own set of requirements and features. While all of these modules use similar combinations of hybrid CBR/RL to acquire and manage the relevant knowledge, there are distinct differences in the solutions depending on the particular task. This question is addressed in Chapters 7 and 8.
5. *What interfaces are required in a hierarchical hybrid CBR/RL architecture and what are the potential effects, positive and negative, of using a homogeneous ML approach for significant parts of reasoning spanning multiple layers of abstraction in an RTS game?* To answer this question, the individual modules created to answer (4.) are integrated into a holistic approach that addresses the entire micromanagement problem. In addition to an analysis of the overall performance, a meta-level analysis of the interaction between the different components is performed. This meta-level analysis of execution

frequencies is then used to draw conclusions on how well the overall approach leads the agent to exhibit intelligent, human-like activity. Potential shortcomings and limitations are explained in the subsequent discussion. This question is answered in Chapters 9 and 10.

6. *How successful is knowledge transfer across multiple levels of hierarchically interconnected modules?* During the evaluation of the integrated approach created in (5.) previously acquired knowledge in lower-level modules has to be re-used across several layers of reasoning. The evaluation thus leads to findings on how successful this re-use is. The aim of the hierarchical CBR/RL approach is the acquisition of knowledge of how to play parts of an RTS game. An additional sub-question is therefore: Given the complexity of the domain through varying map environments and army compositions, how well does acquired knowledge transfer from one scenario to another. To answer this question knowledge acquired in one scenario is used to perform in different yet related scenarios. This question is answered in Sections 9.4 and 9.6.

1.2 Research Contributions

The research presented in this thesis contributes to the field of artificial intelligence and its machine learning branch, more specifically to game AI and the areas of reinforcement learning and case-based reasoning. The progress of the research is categorized into roughly three iterative steps that are geared towards solving the research questions listed in the previous section. The eventual goal is to answer the initial research questions through the creation of a machine learning approach that acquires the knowledge to address the micromanagement problem, a complex sub-problem of real-time strategy games.

1. Reinforcement Learning is one of the central ML techniques that are used in this research. Before using RL as part of a hybrid approach for micromanagement in RTS games, an evaluation of the potential of RL techniques on their own is performed. This involves an experimental evaluation of common RL algorithms to identify the most suitable algorithms, exploration techniques and model for an application to the micromanagement problem as well as an analysis of their performance. This work has resulted in the following publication:

Wender, S., Watson, I.: *Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game StarCraft:Broodwar*. In: Proceedings of the IEEE Conference on Computational Intelligence and Games 2012 (CIG 2012).

2. Micromanagement is a complex problem and as such is defined by a very large state- and action space. Using the knowledge gained in the previous RL approach, a hybrid

CBR/RL agent is created and applied to a larger subset of the micromanagement problem in an RTS game. This also includes an optimisation of parameters for the domain and an analysis of performance, bottlenecks and potential of the application of CBR in the problem domain in general in conjunction with RL for the given task of micromanagement in particular. This work has resulted in the following publication:
Wender, S., Watson, I.: *Integrating Case-Based Reasoning with Reinforcement Learning for Real-Time Strategy Game Micromanagement*. Proceedings of the 13th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2014).

3. Based on previous findings on the application of hybrid CBR/RL to the micromanagement problem and an analysis of the structure and components of the problem domain, my research eventually resulted in a modular architecture that addresses the entire micromanagement problem in an RTS game. This architecture was inspired by the layered learning methodology as well as other related approaches and adapted towards using hybrid CBR/RL in a hierarchical manner. The contribution is fourfold. First, there is the actual modular architecture. The architecture is designed to not just be specific to the chosen testbed (the StarCraft RTS game) but to be easily adaptable for both other RTS games and related problem areas which contain similar structures of tactical and reactive tasks in terms of managing individual units and coordinating squads of units. As part of this hierarchical architecture, several distinct levels of reasoning within the problem domain are addressed, where each level handles specific sub-problems. Crucial aspects of RTS game micromanagement are defined as individual modules in the architecture and modules for additional tasks can easily be added. In the eventual evaluation of the individual modules, the hierarchical architecture is shown to enable knowledge transfer between problem scenarios on all three layers.
 - a) The second part of the contribution is the module that makes up the lowest level of the architecture; a component that learns how to manage unit navigation and pathfinding using goals provided by higher-up levels in the hierarchical architecture. The CBR/RL approach to unit navigation that was devised for this component resulted in this publication:
Wender, S., Watson, I.: *Combining Case-based Reasoning and Reinforcement Learning for Unit Navigation in Real-Time Strategy Game AI*. In: Proceedings of the 22nd International Conference on Case-Based Reasoning (ICCBR 2014).
 - b) Third, for the mid-tier levels of the architecture, two CBR/RL components are created that learn how to coordinate between squads of units to create *Formations* and to focus *Attacks*. These two tasks were identified as the two core tasks that require coordination among units and which make up the mid-level logic in tactical

micromanagement in RTS games and combat simulations. For each of the two tasks, a sophisticated CBR/RL module is created and trained for re-use as part of the overall architecture. In addition to using CBR for memory management and generalisation, the *Formation* module also uses a solution case-base that allows use of a complex solution definition based on unit-slot permutations.

- c) The fourth and final part of the contribution is the overall tactical component, which integrates the previously-created individual modules into a homogeneous CBR/RL approach that addresses all identified reactive and tactical tasks.
4. Extensive knowledge modelling is performed to abstract the game world into a format that the five different CBR/RL modules which were developed during the course of this thesis and the RL module can work with. Throughout the development of these modules, this leads to an incrementally more complex approach of modelling RTS game environments for utilization in AI agents. One of the principal concerns of this part of the thesis was a focus on an innovative use of influence maps for encoding spatial information to form parts of case descriptions in a CBR component. Given the resulting high-dimensional case definitions, appropriate similarity metrics are developed and tested in order to compare large number of high-dimensional cases (see Sections 3.4.1 and 3.4.2).

1.3 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 surveys the related work in areas relevant to the content of this thesis. It starts with an overview of research concerning reinforcement learning and case-based reasoning, the two main ML techniques used in this thesis. This is followed by relevant research into applying hierarchical architectures and layered learning to any of the algorithms or to the problem domain. Finally, a summary of significant work on RTS game AI in general and research in the most popular testbed, the commercial RTS game StarCraft in particular, is explained.

Chapter 3 enlarges on the previous chapter by providing background both on the problem domain, RTS game micromanagement, and the machine learning techniques and algorithms that are used. First, background on RTS games as research testbeds is presented. This is followed by a section on example RTS game agent architectures. Then, details on RL and relevant RL algorithms are listed. The concept of CBR and relevant CBR similarity metrics are explained. Finally, important principles and concepts behind hierarchical architectures and layered learning are explained.

Chapter 4 evaluates the capabilities of different reinforcement learning algorithms for the purpose of learning micromanagement in a RTS game. The aim is to decide on the suitability of RL for creating a learning agent in StarCraft. In order to do this, two prominent *temporal-difference* (*TD*) learning algorithms are used, both in their simple one-step versions and in their more complex versions that use eligibility traces. A StarCraft agent is implemented and the performance for the different RL algorithms in a simple micromanagement scenario is evaluated.

Chapter 5 describes the concept of a hybrid RL and CBR approach to managing a group of combat units in StarCraft. Both methods are combined into an AI agent that is evaluated by using a more complex scenario than the one for the RL-only agent in Chapter 4. Combining CBR with RL helps to offset shortcomings of the simple RL agent while retaining key features of its performance. The hybrid agent uses CBR for its memory management and state-space generalization and RL for the adaption of solution fitness values. This chapter also contains an optimization of the algorithmic parameters for both RL and CBR components using a combination of experimental evaluation and ML to find the best possible settings. Subsequently, as part of an experimental evaluation, the agent is tested in different scenarios using these optimized algorithm parameters.

Chapter 6 describes the concept of a hierarchical architecture of CBR/RL modules. First, the considerations behind modeling the micromanagement problem in a hierarchical manner are explained. The procedure for evaluating the architecture and the reasoning behind it in the context of acquiring knowledge on how to play the game are next described. Finally, the process of mapping in-game unit entities to knowledge-base unit entities is explained, since this is a recurring step that is crucial for the effective use of RL.

Chapter 7 describes the creation of the CBR/RL module for pathfinding and navigation. This module makes up the lowest level in the architecture. This and subsequently created modules are based on the previous approaches described in Chapters 4 and 5. The agent acquires the knowledge it needs through interacting with the StarCraft game environment and subsequently uses that knowledge to navigate in micromanagement game scenarios.

Chapter 8 describes the different squad-level components on the second level of the hierarchical architecture in detail. This includes the two main modules, *Formation* and *Attack*, as well as a shorter section explaining the deterministic *Retreat* functionality. Both the *Formation* and the *Attack* modules are evaluated in scenarios that are designed to showcase the relevant competencies. After a discussion of the results, the modules are trained for re-use by the *Level One* module.

Chapter 1. Introduction

Chapter 9 outlines the high-level *Tactical Unit Selection* module. This module makes up the highest layer, *Level One* of the architecture. The results obtained during the evaluation of this module in appropriate game scenarios are discussed in detail. Furthermore, an evaluation of the capabilities of the *Tactical Unit Selection* module in terms of knowledge transfer is performed.

Chapter 10 presents wider implications of the results in the individual evaluations in previous chapters. In particular, the evaluation of the integrated hierarchical agent is discussed, as are the knowledge acquisition capabilities of the agent, knowledge transfer between scenarios and contributions of individual modules and the overall architecture. The discussion examines in general the relevance of research results in terms of contributions as well as the positioning of the contributions both in terms of existing research and in terms of the initial research questions. Furthermore, limitations and possible ways to offset these limitations are discussed for each chapter.

Chapter 11 concludes the thesis by summing up the overall findings and contributions.

This chapter looks at the related work in the main areas of research that this thesis touches upon. While other problem areas are mentioned, a large part of the related work focuses on game AI and learning in games. The examined related work is relevant and influential to areas such as RL, CBR, layered learning, hierarchical architectures and RTS game AI as well as StarCraft. Additional related work that relates only to specific parts of this thesis, such as navigation and pathfinding (see Section 7) or unit formations (see Section 8.1), is presented in the relevant sections.

The content of this chapter is arranged to roughly follow the structure of the main contributions of the thesis in Chapters 4 to 9. Thus, the sections for specific areas of related work examine reinforcement learning, case-based reasoning, hierarchical architectures and layered learning in sequence. Furthermore, combinations of these approaches as well as applications in the area of game AI research are listed, all in the context of how the research presented in this thesis improves upon the techniques currently available. For each of the three main parts (RL, hybrid RL/CBR and hierarchical RL/CBR) related work leading up to the research done in the relevant section is presented. Additionally, for the earliest - Chapter 4 - subsequent related work following the publication of Wender & Watson (2012) is listed.

Interesting aspects of AI research that uses RTS games in general and the commercial RTS game StarCraft in particular are also explored. The huge increase in popularity of research in computer game AI over the past decade has led to a large increase in publications in this area. Limiting the review to research related to RTS game AI or even only to research that uses StarCraft still produces too many publications to survey each of them. Another indication of this rise in the number of publications is the number of surveys on computer game AI (Fürnkranz, 2001, 2007; Galway et al., 2008; Muñoz-Avila et al., 2013), RTS game AI or, in the past few years, surveys focusing specifically on StarCraft game AI (Lara-Cabrera et al., 2013; Ontañón et al., 2013; Robertson & Watson, 2014). Therefore, this section is limited to the more significant publications which are closely related to the work presented in this thesis.

2.1 Reinforcement Learning for Computer Game AI

Large parts of this thesis are concerned with the application of RL to the task of micro-management in real-time strategy games, often in connection with other machine learning techniques. The use of RL for micromanagement was inspired by previous work on using the technique for city-site selection in the turn-based strategy game *Civilization IV* (Wender, 2009). Parts of this Related Work section regarding RL are inspired by content created for that thesis due to the underlying similarities. The first part of this thesis, described in Chapter 4 and published in Wender & Watson (2012), describes the creation of an agent that learns how to manage single units in combat situations in StarCraft using a number of different RL algorithms. Section 3.3 provides more information on the general background for RL. This includes background on its origins and on the basic RL algorithms. The related work presented in this section focuses on the application of RL to RTS game AI.

Together with the increased interest in computer game AI in general, the application of RL algorithms in computer game AI has seen a big increase in popularity within the past decade, as the technique is well suited to complex game environments. RL is very effective in computer games where perfect behavioural strategies are unknown to the agent, the environment is complex and knowledge about working solutions is usually hard to obtain.

Szita (2012) surveyed the use of reinforcement learning both in classic board games and in computer games. The author found that RL is a technique that has been used extensively in board games with impressive results, such as Tesauro's *TD Gammon* (Tesauro, 1992) and, more recently, the search-based approaches to the game of *Go* (Gelly et al., 2006).

Additionally, RL is still an area of active research that produces new discoveries in terms of algorithms and findings on the underlying theoretical background (Brafman & Tennenholtz, 2003; R. Sutton et al., 2009; Maei et al., 2010). Recently, the *UCT* algorithm (*Upper Confidence Bounds applied to Trees*) (Kocsis & Szepesvári, 2006), an algorithm based on Monte-Carlo Tree Search (MCTS), has lead to impressive results when applied to games. MCTS in general and UCT in particular are closely related to RL which is partially based on Monte-Carlo methods.

Balla & Fern (2009) applied UCT to planning problems using simple Wargus scenarios as a problem domain. UCT differs from other MC algorithms in that it does not require significant expert domain knowledge. The authors focused on the tactical planning for battle scenarios with similar unit configurations and omitted the resource-gathering and base-building parts of the game. In their experiments, the authors showed that the UCT planner performs very well in all tested scenarios when compared to baselines and even human players. Wargus is an open source RTS game built on top of the *Stratagus* RTS game engine and modelled according to StarCraft's predecessor, *Warcraft II*. As the similarity in problem domain suggests, the use

of MCTS in general and UCT in particular have seen some promising results when applied to StarCraft, especially in the micromanagement part of the game.

This is in spite of the fact that techniques which are based on game tree search translate badly to StarCraft. The main reason for the usually bad performance is the limited access to the game source. Even using the powerful BWAPI interface, there are many parts of the game core that cannot be manipulated. This means there is uncertainty about a number of in-game mechanics and game states can only be manipulated by using in-game simulation. This also requires computationally expensive calculations such as collision control and unit vision. In approaches based on game tree search, where extremely large numbers of minimally different states have to be checked in every search cycle, this makes a real-time computation all but impossible. Churchill et al. (2012) overcame this and described the use of heuristic search to simulate combat outcomes and control units accordingly. Because of the aforementioned lack in speed and precision of the StarCraft game environment, the authors first created their own simulator, *SparCraft*, to evaluate their approach. SparCraft emulates the original game and also allows skipping game states that are ‘uninteresting’ for the search algorithm, thus tremendously speeding up the simulation. Using this simulator and a modified version of alpha-beta pruning that also takes into account the duration of actions, they looked for the best moves for a given unit configuration. Search times were limited to stay within ‘real-time’ conditions. Despite these limitations, the agent still managed to win 92% of the games versus the scripted opponents it is tested against. However, these results did not translate back from the SparCraft simulator into the actual game perfectly, as the simulator simplifies the original problem and does not take into account unit collisions and acceleration rates. The win rate dropped to 82%, despite a prediction of 100% by the agent (Churchill et al., 2012; Churchill & Buro, 2012).

In Churchill & Buro (2013), the authors extend their previous research and introduce both a variation of UCT search, *UCT Considering Durations*, and a greedy-search-based technique called *Portfolio Greedy Search*. Both approaches are evaluated against each other with *Portfolio Greedy Search* performing the strongest. The authors then translate these combat simulations from the SparCraft simulator back into the actual StarCraft game by including the simulator as part of their *UAlbertaBot* which plays entire games of StarCraft. *UAlbertaBot* was the winner of the 2013 instance of the StarCraft bot competitions at *AIIDE* Ontañón et al. (2013).

Bowen et al. (2013) also create a micromanagement component which uses UCT to search for the best orders to assign to tactical groups. The authors use a very high level of abstraction and, contrary to Churchill et al. (2012) with their SparCraft simulator, work within the original game. The high level of abstraction leads to only two possible actions for each unit, *join* and *attack*. The micromanagement module is trained offline using UCT and then

integrated with the authors' game-playing *Adjutant Bot*. This approach is then compared against built-in AI, fixed scripted attack patterns and a manually-tuned micromanagement behaviour. UCT is found to compare favourably, being second only to the expert-tuned behaviour patterns. This is an impressive result given the high level of abstraction that is employed.

All the approaches listed so far have addressed low-level micromanagement tasks. The main reason is that in search-based approaches the branching factor is integral to the success of an algorithm. RTS games have large branching factors even at low levels of a hierarchical problem decomposition and these branching factors exponentially increase the higher in the problem space one goes (Ontañón et al., 2013). In spite of this, Uriarte & Ontañón (2014) apply MCTS and alpha-beta search for game tree search over high-level game states. The authors do this by creating a very high-level abstraction of the game states that only includes combat units in state descriptions. The entire game state is represented as a matrix with one row per unit type and spatial region with few attribute columns. Each group of units addressed in such a way can then be assigned high-level orders of types *move*, *attack* or *idle*. In an experimental evaluation, the two search algorithms and an optimized scripted AI are compared against the built-in AI. To account for search times, the game is stopped when a new strategy is searched every 400 frames. MCTS is shown to perform better than alpha-beta search, while the highly-optimized scripted AI still achieves better results than both search-based approaches.

Apart from MCTS and UCT however, few of the new theoretical discoveries in RL have made it into game AI research. This is mostly due to the large differences between the theoretical definitions of an algorithm or exploration technique with promising characteristics such as the one described by R. Sutton et al. (2009) and the significant boundaries for adapting such an algorithm to real-world problems. Most research in computer game AI, including this thesis, works with the well-tested temporal difference (TD) RL algorithms. However, this could change with some of the more recent discoveries looking promising for an application in RTS game AI.

Subsequently, the related work concerning RL in computer games is loosely divided by the specific TD algorithms used. The TD algorithms applied in this thesis are described in detail in Section 3.3. Modular and hierarchical RL approaches are explained as part of Section 3.5.

2.1.1 Q-Learning

An important breakthrough in reinforcement learning was the development of Q-learning (Watkins, 1989). Q-learning integrates different branches of previous research such as dynamic programming and trial-and-error learning into RL.

Andrade et al. (2005) used a Q-learning algorithm to create an adaptive agent for a fighting game, *Knock'em*. The agent was initially trained offline to be able to adapt quickly in an online environment. During the game, it adjusted its play to the other player's level of skill. The game AI thus always presented a challenge but remained beatable regardless of the player's level of skill.

M. Smith et al. (2007) introduced *RETALIATE* (**R**einforced **T**actic **L**earning in **A**gent-**T**eam **E**nvironments), an online Q-learning technique that creates strategies for teams of computer agents in the commercial FPS game *Unreal Tournament*. The bots controlled by RETALIATE were able to adapt and devise a winning strategy against the built-in game AI in most cases, using just a single game for learning. At the risk of getting stuck in local maxima, future rewards were **not** discounted. This led to an extremely fast convergence and the performance showed that the overall approach was successful.

Auslander et al. (2008) extended the RETALIATE algorithm developed by M. Smith et al. (2007). They introduced CBR elements in order for the agent to adapt faster to a change in the strategy of its opponent. The resulting technique, *CBRetaliante*, tried to obtain a better matching case whenever the collected input reading showed that the opponent was outperforming it. The Q-table of the previously computed policy was part of the stored/retrieved cases together with other recorded data. As a result of the extension, the CBRetaliante agent was shown to significantly outperform the RETALIATE agent when it came to sudden changes in the opponent's strategy.

Whiteson & Stone (2006) used a combination of Q-learning and the *NEAT* (**N**euro**E**volution of **A**ugmented **T**opologies) algorithm to perform online evolutionary function approximation. They tested this method with several standard RL benchmarks and concluded that evolutionary function approximation can significantly improve standard TD learning.

Jaidee & Muñoz-Avila (2012) created *Class Q-learning* ($CLASS_{Q-L}$), a Q-learning agent which plays entire games of Wargus. Given the high complexity of the game and the limitations of simple table-based Q-learning, learning how to play the game would normally not be possible. They overcame this by learning separate action-value functions for each of twelve selected unit classes. Each class received its own model with appropriate state descriptions and actions. During the learning process, an entire game was played and Q-tables were subsequently updated, effectively resulting in offline learning. In an experimental evaluation, the $CLASS_{Q-L}$ agent was trained and subsequently shown to perform well against a number of scripted opponents in small-scale scenarios.

2.1.2 Sarsa

Graepel et al. (2004) used the Sarsa algorithm to compute strategies in the commercial fighting game *Tao Feng* to create a challenging computer opponent. Q-learning could not be used because of the lack of information about possible actions in any given state. Sarsa does not require this information.

Stone et al. (2005) used the game *RoboCup Soccer Keepaway* as a testbed. In the game, a number of agents (the keepers) try to keep the ball in their possession within a limited area while their opponents (the takers) try to capture the ball. The episodic semi-Markov decision process (SMDP) version of a Sarsa(λ) algorithm was used to compute the best strategy for the adaptive keepers against a set of predefined static takers. Empirical studies showed that the time it took the takers to acquire the ball rose in line with the number of episodes that were played.

McPartland & Gallagher (2008) used a tabular Sarsa(λ) algorithm to control a bot in a first-person-shooter game. Different techniques were combined with RL (finite state machines (FSMs), rule based and hierarchical RL) to control the bot and the different approaches were compared against each other. Hierarchical and rule-based RL proved to be the most efficient combinations.

Closely related to the agent in Section 4 is the approach presented by Micić et al. (2011). The authors also developed an adaptive agent for low-level combat in *StarCraft*. They implemented a custom A* algorithm to handle the units' pathfinding and used heatmaps to easily cluster information on military power. Both pathfinding information and heatmaps were eventually used in game state descriptions by the *Information Manager*-component of the proposed architecture. The other major component, the *Unit Manager* was designed to control individual unit's decisions and actions using simple FSMs with the actions *Fight*, *Retreat*, *Regroup* and *Idle*. No grouping behaviour was implemented at the time, as this would have had to be handled by a higher-level manager. The authors used the Sarsa algorithm to train the intelligent agent in varying scenarios using different types and numbers of units. Empirical evaluation showed how the agent managed to improve its performance to win far more than half of the games in most scenarios. However, the chosen state representation was highly tailored towards the experiments that were run and the size of the state space severely limited the efficiency of the learning agent. An extension of the managed units beyond those used in the experiments does not seem possible without entirely redesigning the RL model.

Another approach closely related to Section 4 is Shantia et al. (2011). The authors used neural networks (NNs) to approach the problem of state space complexity that is inherent in RTS games such as *StarCraft*. The NN received the game state as input and then approximated the state-action value function $Q(s_t, a_t)$ to be used for the Sarsa RL algorithm, thus creating neural-fitted Sarsa (NFS). With sufficient training, this technique was shown

to significantly outperform the built-in game AI in combat scenarios between a small number (3vs3) of similar units. However, in a larger (6vs6) scenario the NFS agent did not manage to outperform the built-in game AI. Furthermore, the authors also pointed out that the gains of NFS over normal Sarsa were small. Scalability was another problem as the learning process remained slow in the 6vs6 scenario, despite an acceleration by transferring knowledge from the 3vs3 scenario.

The listed approaches have several things in common that in turn point to problems inherent in using RL in this problem domain. The authors created models that are tailored very strictly to the environment that the agents perform in, often only to a very particular scenario. Furthermore, the particular algorithm that the authors used was chosen without an evaluation of possible alternatives. The first step in this thesis, presented in Chapter 4, is an evaluation of a RL agent that manages units in StarCraft micromanagement scenarios using a number of different TD algorithms. Additionally, the model created for that particular agent is designed to serve as a basis for an agent that addresses problems more complex than controlling a single unit.

2.2 Case-Based Reasoning and Hybrid Approaches

Using only RL for learning diverse actions in a complex environment quickly becomes infeasible due to the curse of dimensionality (R. E. Bellman, 1961). Therefore, additional modifications such as ways of inserting domain knowledge or combining RL with other techniques to offset its shortcomings are necessary. As part of this thesis, RL is combined with case-based reasoning (CBR) (Aamodt & Plaza, 1994). CBR is a methodology that re-uses solutions of previously-encountered problems to solve current problems. Section 3.4 gives more background information for the CBR methodology.

Combining CBR with RL has been identified as a rewarding hybrid approach (Bridge, 2005) and has been done in different ways for various problems. The combination of CBR and RL that is described in the second part of this thesis is performed in order to enable the agent to address more complex problems by using CBR as an abstraction- and generalisation-technique. Chapters 5 and 6 - 9 elaborate on the details of the created hybrid algorithms and modules. This section shows relevant research that uses variations of CBR, including hybrid CBR/RL, in game AI.

One of the most common problem areas addressed by CBR-based approaches in RTS game AI is high-level strategic planning. Case-based planning (CBP) in particular has been used extensively for high-level planning of strategies in computer games. CBP combines planning with lazy learning through re-use of experience (Hammond, 1989; Cox et al., 2005). Similar

to the general CBR methodology, case-based planning techniques re-use previously-acquired knowledge in the shape of plans for new situations instead of creating plans from scratch.

M. Ponsen (2004) used offline learning to improve *dynamic scripting* (Spronck et al., 2003), a technique inspired by RL, in Wargus. In contrast to other approaches, Ponsen addressed not a low-level sub-problem but worked with plans for strategies to win the overall game. Ponsen successfully created an AI agent that utilised dynamic scripting to play entire games superior to static opponents. Additionally, an evolutionary algorithm was introduced that further optimised the planning of the agent to beat even the algorithms improved by simple dynamic scripting. D. Aha et al. (2005) extended this approach by creating an agent that is capable of transferring knowledge between tasks. They introduced a *Case-based Tactician (CaT)* that uses knowledge learned while playing against one opponent to quickly adapt strategies when playing against a different opponent. The authors utilised the *Testbed for Integrating and Evaluating Learning Techniques (TIELT)* (D. W. Aha & Molineaux, 2004), an attempt to standardise the learning environment, to integrate their AI agent into Wargus. The results showed an agent that manages to quickly adapt to new environments and promised even better adaptability with future improvements.

Ontañón et al. (2007) applied CBP to the task of playing entire games using knowledge gained from human-annotated game logs for Wargus. The authors created a framework that allowed planning and execution in real time and was based on the concepts of ideas of behaviours, goals, and alive-conditions from *A Behavior Language (ABL)* (Mateas et al., 2002). Their planner kept track of remaining open goals in the current plan and retrieved the most adequate behaviour for each open goal from the case-base, according to the current game state. The case-base was generated by observing humans while they played the game, recording their respective goals at a certain point. Mishra et al. (2008) re-used this CBP framework, called *Darmok*, and further improved it with a situation assessment algorithm based on a decision tree model that improved retrieval times. As the transfer of knowledge from one scenario to another with differences in map layout or size was problematic in the original implementation, the authors extended *Darmok* by introducing this refined case-selection process. The process generated models reducing the size of the possible case-base in an additional step before the actual case selection. The execution of this step returned a set of high-level features which could be used in the case selection.

Cadena & Garrido (2011) combined CBR with fuzzy sets for managing tactical reasoning in StarCraft. They also implemented a discrete CBR-based solution for managing strategic reasoning and split the whole game into four parts. This hierarchical decomposition is similar to the one that is done as part of this thesis and separates the overall task into strategic reasoning, resource management, tactical reasoning and micromanagement. Micromanagement and resource management were solved using hard-coded expert knowledge. For strategic rea-

oning, a standard CBR approach was used. Case descriptions consisted of the numbers of all units and buildings that existed at a certain point in time. Solution plans were sequences of the next five build actions. For tactical reasoning, which includes the deployment and movement of units across the map, the authors used fuzzy CBR. Each region in the map was regarded separately for a case. A case description consisted of a number of fuzzy, i.e. non-discrete variables. Solutions were high-level tactical decisions, such as moving to or attacking certain regions of the game map. For their evaluation, the authors created a case-base from a single replay of a human playing against the inbuilt AI on the same map. Subsequently their agent managed to win about 60% of all games. However, it remained unclear how well the agent would do against a different opponent or on a different map.

Hsieh & Sun (2008) analysed a large number of StarCraft replays and created a case-base for the build orders encountered in the games. They then used this knowledge to predict players' strategies and, based on that, the general performance of said player. This was possible as they counted wins and losses associated with certain build orders and then used this to evaluate which build order performed best in which game situation.

Weber & Mateas (2009a) used CBR to learn build orders in Wargus. They created a CBR component with annotated cases to determine build orders for their agent. A case was determined by a fixed set of features such as workers, fighters and construction buildings. During the retrieval phase certain criteria - depending on which kind of build decision had to be made - were *generalised*, i.e. became irrelevant for the retrieval. All other criteria had to match exactly. This was retrieval with so-called *conceptual neighborhoods* (McCoy & Mateas, 2008). The adapted CBR algorithm was used in the strategy-selection part of an agent that was integrated with Wargus through the reactive planning language ABL. In several experiments where the enemy used different scripted strategies on different maps the newly created CBR algorithm managed to outperform random selection and simple kNN algorithmic selection. This is most noticeable in environments with imperfect information.

The authors extended their previous use of ABL in Weber, Mateas, & Jhala (2010a), where they built an autonomous agent that uses the reactive planning language. They applied the concept of *Goal-driven autonomy (GDA)* to the problem. *GDA* is a framework developed by Molineaux et al. (2010) based on the work by Cox (2007). *GDA* is based on the concept of planning agents that are able to reason about the plans they develop and about inconsistencies between those plans and the actual game state. Before its first application to the StarCraft problem domain, *GDA* had previously been used in a team shooter game where it was combined with a *hierarchical task network (HTN)* planner to compete in a domination game (DOM) (Muñoz-Avila et al., 2010). The agent developed in Weber, Mateas, & Jhala (2010a) consisted of multiple components: a discrepancy detector, an explanation generator, a goal formulator and a goal manager. The evaluation showed that the agent (EISBot) won

73% of all games across different maps and against different races. As the first (and to date only) StarCraft AI bot, EISbot was also entered into the *Ladder*, an online competition for human StarCraft players. In games against competitive human players, EISBot achieved an impressive 37% win rate, outperforming 48% of all players in the competition.

Jaidee et al. (2011) extended the standard *GDA* algorithm presented by Muñoz-Avila et al. (2010) into *Learning GDA (LGDA)*. LGDA analyses the resolution of discrepancies between expected states and actual states and generates expectation cases, which map state-action pairs to a distribution over expected states, and goal formulation cases, which map goal-discrepancy pairs to a distribution of expected values over discrepancy-resolution goals. LGDA was created by integrating CBR with RL, i.e. the agent tried to choose the best goal, based on the expected reward. While the integration of CBR and RL differs from the approach pursued in this thesis, the online acquisition of knowledge using a CBR/RL approach is similar to the technique presented in Chapter 5. The aim is to create a LGDA which performs as well as the standard GDA that employs expert knowledge. The authors used a team domination game in a FPS as testbed for their research. The results of these experiments showed that LGDA managed to significantly outperform most hand-coded opponents and performed nearly as well as the non-learning GDA that used expert domain knowledge.

Weber, Mateas, & Jhala (2010b) coined the term *case-based goal formulation* to describe their formulation of goals for game-playing agents based on traces i.e. sets of cases. StarCraft replays were used as traces while states were described as sets of feature vectors with numbers representing certain units (workers, buildings, fighters). Goals are states as well, meaning that there is a current state and a goal state with the planner deducting the necessary actions to get from one to the other. Goal reformulation happens in a number of events like attacks or when a previous goal has been reached. The difference between the previous and the new goal is computed and the plan is updated accordingly. States are ‘windows’ of actions, i.e. one state consists of a certain number of actions. A variation of this number allows one to switch between short- and long-term planning. The authors evaluated their system using opponent modelling in StarCraft, trying to predict their opponent’s strategy. They successfully managed to outperform other statistical classifiers such as *NULL*, *IB1* and *AdaBoost*. The authors also integrated these predictions into their *EISBot* StarCraft AI agent to perform planning according to the given cases.

In Weber & Ontanón (2010) the authors built a case-base for the Goal Oriented Action Planning (GOAP) framework *Darmok 2* (Ontanón et al., 2009) described above, by automatically annotating StarCraft replays. They did this by defining a ‘Goals-to-win-StarCraft’ ontology and automatically breaking up replays into cases by splitting them according to the actions happening. Different types of actions were grouped into the shape of a *Petri net* by the goals these actions were supposedly trying to achieve (e.g. set up resource infrastruc-

ture, strategic decision, tactical decision) and based on the time at which they happened. A label was applied according to the state the replay was in when the first action happened. The authors then created a case-base using this method. Subsequently, using the Darmok 2 framework to play the game, their agent gathered resources but ultimately did not manage to defeat the built-in StarCraft AI consistently.

Szczepański (2010) directly applied *trace-based reasoning* (TBR) (Mille, 2006), a form of CBR in which cases are made from series of temporally connected data, to micromanagement in StarCraft. The author created a ML agent to micromanage combat units based on stored knowledge. Traces of unit attributes were used in the case descriptions. However, the development process did not involve the analysis of game replays or a review of recorded agent behaviour. Attribute selection was only based on expert knowledge.

Molineaux et al. (2008) described the integration of CBR and RL in a continuous environment. Both state- and action-space were continuous and the agent had to learn effective movement strategies for units in a RTS game. This approach was unique in that other approaches discretize these spaces to enable machine learning. As a trade-off for working with a non-discretized model, the authors only looked at the movement component of the game from a meta-level perspective where orders are given to groups of units instead of individuals and no orders concerning attacks are given. The authors used two separate case-bases for state-action pairs and state-value pairs. Updates only happened to the state-value case-base and reward is distributed among several cases in the value case-base according to how similar cases are to newly encountered problems. The algorithm used a standard RL value update based on previous state, previous chosen action, reward obtained and new state. Empirical evaluation in *MadRTS*, a commercial RTS game engine, showed that the algorithm outperformed both random agents as well as agents using discrete state/action spaces significantly, both in terms of efficiency and won scenarios. The solution case-base used for unit formations in Section 8.1 is partially inspired by the state-value case-base used here. The publication is also interesting due to its innovative abstraction of the large-scale state- and action-spaces into a working model.

A lot of research using CBR is based on either observing humans play directly or on analysing replays of human performances. The availability of large amounts of expert gameplay data in the form of replays is another factor that makes StarCraft such a popular testbed. Game replays in StarCraft are not only used for CBR but serve as a source for expert knowledge for a variety of other ML approaches. Dereszynski et al. (2011) used hidden Markov models to model opponents in StarCraft. They analysed a large number of logs of expert human games to discover players' strategies. Strategies were modelled as a sequence of hidden states, each state containing information on a player's preferred buildings and units. The

model also contained transition probabilities between the states, thus allowing the prediction of a player's likely strategies and finding common strategies in general. Furthermore, as the model contained information on the time of the actions as well, it enabled the authors to find important decision points in a game.

Weber & Mateas (2009b) analysed a large number of StarCraft replays in order to predict an opponents strategy based on their build order. A major issue when using game replays as data source is the limited amount of information that can be obtained from them. The information contained in replays is limited to build-actions and -times and does not include any information on players' current resources or remaining active units. The authors built one feature vector for each player that contained all possible features (units/buildings/upgrades) of that player. Replays were also labelled, based on their strategy defined by the first advanced building that is created, e.g. for a focus on air units or a focus on cloaked units. The feature vectors were then analysed by machine learning algorithms such as *J48*, *kNN*, *NNge* and *LogitBoost* in order to find common strategies. The aim was to recognize common strategies and to predict an opponent's moves based on the current information. The algorithms managed to predict build timing well with varying degrees of accuracy depending on the type of building. Synnaeve & Bessiere (2011a) pursued a similar approach of analysing game logs to create a simple Bayesian model for predicting the buildings a player uses. They made use of the same set of replays as Weber & Mateas (2009b). Their aim was to create an agent that used this model to infer its opponent's strategy from the incomplete information that is available. The authors' approach focused on buildings, since these are more visible for the agent when playing the game. Whereas units usually vanish in the *fog of war*, buildings can serve as more prominent indicators of an opponent's strategy. The authors succeeded in creating a robust model that could predict up to four buildings in the future reliably while remaining relatively immune to the noise.

While the knowledge contained in the cases in a CBR system for RTS games often comes from demonstrations by an expert player, this is not always the case. An example of an approach which obtains knowledge directly from the environment is Baumgarten et al. (2008). They used an iterative learning process that is similar to RL and employed that process and a set of pre-defined metrics to measure and grade the quality of newly-acquired knowledge while performing in the RTS game *DEFCON*. Their system learned which moves it could make at the same time as learning which move the opponent was likely to make. The system used CBR to store information and generalise by grouping similar cases through a decision tree algorithm, based on how successful a particular game is that the case came from. Success is measured in terms of game score and several additional metrics. Using a high-scoring decision tree results in a game plan.

Similar to this approach, the aim in this thesis and the CBR modules created as part of it is to acquire knowledge directly through interaction with the game. The learning process is controlled by RL which works well in this type of unknown environment without previous examples of desired outcomes. CBR is then used for managing the acquired knowledge and generalising over the problem space. Another aspect demonstrated by the related work concerning CBR approaches is the limited scope in terms of problem space that these approaches are usually applied to. Most publications describe not the creation of a CBR-based game playing agent but of an approach to predicting strategies or build orders. Even for approaches that do play the game, the problem is usually very limited, most often in the strategy layer of RTS games, occasionally either in the tactical or reactive layer, but never across several layers. As such, the creation of a hierarchical CBR agent that addresses several layers of complexity at the same time can be a valuable contribution.

2.3 Hierarchical Approaches and Layered Learning

Combining several ML techniques, such as CBR and RL, into hybrid approaches leads to more powerful techniques that can be used to address more complex problems. However, problems such as those simulated by commercial RTS games with many actors in diverse environments still need significant abstraction in order for agents to successfully solve the problems they are confronted with. A common representation of the problems that are part of RTS games is in a hierarchical architecture (Ontañón et al., 2013). This section examines related work in terms of hierarchical approaches to problem solving, both related to RTS games in general and related to RL and CBR. This includes hierarchical case-based reasoning, hierarchical reinforcement learning as well as the layered learning (LL) paradigm which was introduced for problems in the robotic soccer domain, an area that is closely related to RTS games (Stone, 1998), and has found wide-spread application there. Layered learning is remarkably suitable for a holistic CBR/RL approach to strategy game AI as the one that is created in this thesis. LL characteristics and relevant features are explained in detail in Section 3.5.

Holistic approaches that attempt to address AI problems by only using a single unified method for only a single overall problem are not very common as decomposing the problem into more manageable smaller, separate problems often leads to better results (Ontañón et al., 2013). One holistic approach is the previously mentioned Darmok system (Ontañón et al., 2010) which uses a combination of CBP and learning from demonstration to create plans to play the Wargus RTS game. The system learns from observing human players and storing their behaviour in case-based plans which are annotated by experts. The system then re-uses this experience and extends the observed plan to fit the newly encountered situations.

Humphrys (1996) evaluated different approaches to action selection in RL, among them also modular hierarchical Q-learning. Marthi et al. (2005) developed *Concurrent ALisp*, a language that is based on *ALisp* and has constructs that allow dynamically assigning subagents to tasks that are part of a superordinate overall task. The agents each learned a partial representation of the overall state-value function for their particular subtask. At run time they then combined their knowledge to pick the best overall action. This version of hierarchical reinforcement learning (HRL) was tested in a problem domain where multiple taxis were coordinated to pick up and drop off passengers. The domain is significantly less complex than a RTS game. However, the authors' approach showcased a way to extend Q-learning, an algorithm with strong limitations in terms of problem space, to a more complex domain by hierarchically subdividing the problem. Hanna et al. (2010) used a modular approach to apply RL to games. They created a simple test game based on the classic fungus eater experiment from cognitive science (Pfeifer, 1996) where an autonomous agent tried to balance multiple conflicting goals. Each module in their approach handled a different part of the input and computed a value for a particular state-action entry. Subsequently, all partial values were used to select the next action. The authors showed that, compared to a standard single-module approach, the modular approach resulted in a much better performance for the given task.

Instead of using a single, holistic approach for often hierarchically subdivided problems in RTS game AI, many approaches address subproblems separately, using a diverse range of techniques. By decomposing the overall problem into more manageable subproblems these can also be ordered into the different scales of reasoning that are part of the overall problem. Subsequently, interfaces are created between these hierarchically connected layers where higher layers represent a higher level of abstraction. Architectures structured in this way lead to problems if the distinction between the different layers is not clear or when one level of reasoning has to communicate with another (Weber, Mawhorter, et al., 2010). Another problem arising from a layered hierarchical architecture that is used for micromanagement comes from units being directly controlled on several different levels. If several levels have direct access to a unit control, complex interfaces and coordination are required and abstraction-breaking inter-layer messaging might be necessary (Weber, Mawhorter, et al., 2010). Given these limitations, using a homogeneous holistic approach as presented in this thesis, where each layer has clearly confined responsibilities, is a promising approach for a capable hierarchical architecture.

An early application of hierarchical reasoning in RTS games was done by Chung et al. (2005), who divided the planning tasks in RTS games into a hierarchy of three different layers of abstraction. This is similar to the structure identified in Section 3.1.1, with separate layers for unit micromanagement, tactical planning in combat situations and high-level stra-

tegic planning. The authors used *MCPlan*, a search/simulation based Monte Carlo planning algorithm, to address the problem of high-level strategic planning. The testbed was the *Open Source RTS Environment (ORTS)*, developed by Buro (2003a) in an effort to create a unified interface for RTS games that can be used in AI research. In a capture-the-flag scenario, the agent using *MCPlan* easily beat simple scripted AI. Unit micromanagement and tactical reasoning were managed by scripted behaviour for both agent and opponent.

Andersen et al. (2009) used hierarchical RL to learn how to perform in a simple custom-made RTS game *Tank Commander*. The authors found that the state-action space is too big to use standard RL algorithms and instead split the problem into separate modules on different levels of abstraction. Using both Q-learning and Sarsa, the authors' modular approach mostly outperformed a scripted AI in several scenarios. However, the modules each represented very specific actions ('Attack Opponent Headquarters', 'Attack Opponent Resources') instead of a generalised action such as *Attack*, leading to questions on the overall generalisability beyond the tested scenarios.

Layered learning (LL) was devised for computer robot soccer, an area of research that pursues similar goals as RTS games and can be regarded as a simplified version of these combat simulations (Stone, 1998). The main differences between the two are the less complex domain and less diverse types of actors in computer soccer. Additionally, computer soccer agents often compute their actions autonomously while RTS game agents orchestrate actions between large numbers of objects (Buro, 2003b). Because of the many similarities, LL makes an excellent, though as of now mostly unexplored, paradigm for a machine learning approach to RTS game AI.

The LL paradigm, presented in more detail in Section 3.5, has been extended and used in a number of ways. Originally it was used in the context of *RoboCup* soccer (Kitano et al., 1997), an initiative to encourage research in robotics, AI and related disciplines, by providing a standardised problem domain with popular appeal. More specifically, LL was used for simulated robotic soccer. The problem space was split into five different interconnected layers (Stone, 1998). Three of these layers are modelled and implemented for robotic soccer sub-tasks at different levels in a hierarchical problem decomposition. The three tasks that are evaluated are *Ball Interception*, an individual skill on the first layer, *Pass Evaluation*, a multi-agent behaviour on the second layer and *Pass Selection*, a collaborative and adversarial team behaviour on the third layer. *Ball Interception* on layer one is addressed by using a neural network. *Pass Evaluation* on layer two is addressed through a decision tree. *Pass Selection* on layer three is addressed through *TPOT-RL*, a multi-agent RL method based on Q-learning. All three modules are trained offline. Each layer is individually evaluated in empirical experiments. Furthermore, an evaluation of the integrated layers in several tournaments against other simulated robotic soccer agents leads to very favourable results

with the LL agent winning the final simulation competition in '98 when all components were functional.

Whiteson & Stone (2003) extended the paradigm by introducing a concurrent learning process instead of training layers individually. The authors hypothesized that there are situations where a concurrent approach is possible and beneficial. They then went on to demonstrate this hypothesis using the example of a simulated robotic soccer keepaway task which they subdivided into two layers and learn using neuro-evolution. However, the authors also concluded that the conditions in which such a concurrent learning process is possible and beneficial were limited. Further evidence supporting this observation is found in the later parts of this thesis during the evaluation of the hierarchical architecture.

MacAlpine et al. (2015) combine both original and concurrent LL to create *overlapping layered learning* for tasks in the simulated robotic soccer domain. The original paradigm froze components once they had acquired learning for their tasks. The concurrent paradigm purposely kept them open during learning subsequent layers. Overlapping LL attempted to find a middle ground between freezing each layer once learning is complete and always leaving previously learned layers open. This is done by creating a large number (19) of atomic robotics tasks related to the basic tasks of movement and kicking the ball. The behaviours such as *GetUp_Front_Behavior*, *GetUp_Back_Behavior*, *Kick_Long_Primitive* and *Kick_Low_Primitive* were then put in a six-layer hierarchy and learned offline. During the learning phase, there was then the option of either passing learned parameters on as fixed values or as basis of another subsequent learning process. Additionally, learned values from superordinate layers could be passed back for re-learning. The agent that was created using this paradigm and model managed to outperform all other participants in the *2014 RoboCup 3D simulation competition*.

Layered learning has been combined with other machine learning techniques as well. Gustafson & Hsu (2001) combined it with genetic programming (GP) to create Layered learning GP (LLGP). This approach to solving multi-agent system problems was then further extended and evaluated by Hsu & Gustafson (2002). GP was used to optimise the robotic soccer keepaway task which was decomposed into two layers. The results showed that the decomposition led to individual agents that performed better and could be trained faster.

2.4 StarCraft as a Testbed for AI Research

Due to its enormous popularity and the well-documented and well-maintained BWAPI interface (see Section 3.1.3), StarCraft has recently seen a large increase in its use as a testbed for AI research, as indicated by the large number of publications listed in this chapter that use the game as a testbed. Another reason for StarCraft's popularity is the fact that it exhibits

all the characteristics identified by Buro & Furtak (2004) as interesting AI problem areas in RTS environments. Section 3.1.3 explores these interesting characteristics that qualify RTS games as a simulation environment for AI research in detail.

Even before the creation of a suitable interface enabled game-playing AI agents, StarCraft was of interest as a testbed for research. Its popularity made it interesting not just for AI research but also for other topics. This included such diverse topics as the exploration of its network traffic models (Dainotti et al., 2005), user identification through mouse movement patterns (Kaminsky et al., 2008) and procedural content generation to automatically generate playable StarCraft maps (Togelius et al., 2010).

Due to its large scope and complex problem domain, StarCraft is also an ideal background for research in planning algorithms. In fact, large parts of the AI research using StarCraft as a testbed is concerned with different aspects of automated planning (Lara-Cabrera et al., 2013). Peikidis (2010) used GOAP in his planner *StarPlanner*, which made high-level and mid-level decisions in StarCraft. Safadi & Ernst (2010) created a planning agent capable of playing complete games by splitting the decision-making process into several interconnected parts. Weber, Mateas, & Jhala (2010a) built an autonomous reactive planning agent using ABL.

A major incentive for using StarCraft as a research testbed is the large amount of game traces in the form of game replays that are available online. The large number of CBR-based approaches that re-use expert knowledge from game replays illustrated this.

Lewis et al. (2011) analysed a large corpus of StarCraft replays in the context of cognitive research. The authors tried to find a correlation between actions that are observable in the replays and performing successfully in the games. Their results showed that winning games was directly related to the number of actions that a player performed.

Replays have also been used to analyse which build order performs best against which other build order (Kim et al., 2010). As mentioned previously, Weber & Mateas (2009b) examined a large number of StarCraft replays in order to predict an opponent's strategy through their build order and Synnaeve & Bessiere (2011a) pursued a similar approach of analysing game logs to create a simple Bayesian model for predicting the buildings a player uses. Synnaeve & Bessiere (2011b) showed another possible area of application for a Bayesian model using StarCraft as the problem domain. They created a model in order to maximise the use of individual units by controlling their micromanagement through Bayesian learning. The distributed sensory-motor model the authors introduced also took into account the need for hierarchy, i.e. the control of units on different levels of grouping, a concept that plays a crucial role in this thesis. An empirical evaluation of their model showed that the units

controlled by it significantly outperformed the built-in AI while not requiring any high-level goals from managers above the level of single units. While each unit acted individually, there were four distinct action types, pre-defined using expert knowledge, which also created inter-unit effects such as flocking.

2.5 Summary

As this chapter has outlined, there is considerable interest in using RTS games in general and StarCraft in particular as testbeds for AI research. These games exhibit interesting AI problems and present complex, polished simulation environments. The examination of related work concerning individual ML approaches such as RL, CBR and combinations of both showed that applications of these techniques in RTS game AI have led to interesting results. However, the applications were usually restricted to smaller, clearly delimited sub-problems, unlike the holistic approach proposed in this thesis which is aimed at a large part of the entire RTS game problem.

A hierarchical decomposition of RTS game problems in order to reduce the significant complexity inherent in the domain is common. However, there is no widely accepted or standardised architecture for this. This will be further emphasised in Section 3.2, which analyses example StarCraft bot architectures. Furthermore, game-playing bots were found to often be patchworks of different approaches for different components instead of addressing sets of problems in a homogeneous way. In particular, approaches that address micromanagement through ML often look at it as an isolated component. The LL paradigm has been used extensively in computer robot soccer, an area of research that pursues similar goals to RTS games. Because of the interesting similarities and the use of LL for hierarchical problem decompositions, LL is examined in more detail in the next chapter, which also explores background related to the algorithmic techniques and RTS game characteristics more comprehensively.

This chapter gives an overview of the general background of topics which are relevant in this thesis, both in terms of algorithms and techniques which are used throughout this thesis and in terms of the problem domain. First, an examination of the reasons for using RTS games as a testbed for AI research is performed. As the eventual result in the final step of this thesis is an adaptive StarCraft bot architecture that learns how to play parts of the game, common StarCraft bot architectures and the ML approaches they use are also examined. Then, a general background for the two major machine learning techniques which are part of this thesis, reinforcement learning and case-based reasoning, is provided. Finally, layered learning and hierarchical architectures, are explained in detail.

3.1 Real-time Strategy Games and StarCraft as Testbeds for AI Research

RTS games can serve as complex simulators of real-world problems which offer a large assortment of diverse problems. RTS environments allow researchers to develop new approaches which can be tested without requiring expensive or complex hardware. Continuously evolving computing hardware further reinforces the capabilities of ML approaches to solve the diverse problems incorporated in these virtual simulators. This section describes the considerations behind using RTS games in general and StarCraft in particular as testbeds for AI research. This includes an itemisation of prominent AI research problems which are part of many RTS games. Furthermore, the considerations behind choosing a commercial RTS game over a custom-developed simulation environment are explained. Finally, an overview of the RTS game StarCraft is given and the reasons for its prominence as an AI research environment are shown.

3.1.1 Characteristic Traits of RTS games and their Relevance to AI Research

This section describes the common traits of RTS in terms of gameplay and how these traits make RTS games of interest to AI researchers.

In order to be classified as RTS, a video game usually exhibits a number of certain traits in its gameplay.

- **Economic Aspects:** RTS games require players to build some form of economy to sustain their armies. This usually happens by collecting one or more natural resources like gold, wood or iron by means of units and/or buildings.
- **Base Building:** Players create bases to conscript their army and build their economy.
- **Warring Factions:** There are several factions which battle each other in a contest for victory. Players create combat units and use them in that battle. Achieving victory through combat might not necessarily be the only way but it is a possibility in most games.
- **Technology Tree:** Players advance their units and/or buildings by researching techniques, going along a so-called ‘Technology Tree’. Later research can only be conducted if certain requirements, such as prior research, are met.

Depending on the focus of a game, the different aspects can come in varying degrees of distinctiveness. Broadly speaking, there is a distinction between ‘classic’ RTS, where the focus is on the combat-component of the game (*Command & Conquer* (Westwood Studios, 1995), *Warcraft* (Blizzard Entertainment, 1994), *StarCraft* (Blizzard Entertainment, 1998)) and RTS games where the focus lies on simulating complex economies. Games like *The Settlers* (Blue Byte, 1993) and *Anno 1602* (Max Design, 1998) place more emphasis on setting up bases and economic cycles than on building armies. *Age of Empires* (Ensemble Studios, 1997) gives both aspects equal weight.

In the two decades since the inception of the RTS genre, there has been a plethora of RTS games that balance military and economic aspects in different ways. While there are the two common combinations which, whilst retaining the other major facets, place most weight on either economic or combat aspects, there also exist games that cover basically any possible combination of RTS characteristics.

Buro & Furtak (2004) wrote a seminal paper on the possible role of RTS games as environments for AI research. They identified a significant lack of AI performance in RTS games compared to classic board games. While the paper was written more than a decade ago, similar problems still exist and the reasons then identified still remain the same. The only reason given by the authors that has become nearly obsolete is the limitation of computing power.

Most of a computer's resources are still used for graphical and sometimes physical simulations. However, the massive increase in processing power together with the move towards multi-core architectures means there are now ample resources for AI computations.

As part of their observations, Buro & Furtak (2004) identified a number of fundamental AI research problems which can be studied in RTS environments.

- **Adversarial Planning:** RTS games involve very complex environments in which several agents with opposing goals operate. Due to the large number of available objects and actions, a layer of abstraction has to be introduced to make planning and searching through possible strategies feasible.
- **Incomplete Information:** The 'Fog of War' prevents the players initially from seeing their opponents' actions. Agents have to gather intelligence to make plausible decisions on available information, however limited it may be.
- **Learning and Opponent Modelling:** While human players quickly adapt to new situations, current game AI systems do not adapt at all or only very slowly and in limited fashion to previously unknown situations. To make computer game agents viable opponents, they have to be able to adapt and learn quickly.
- **Spatial and Temporal Reasoning:** Understanding map layout and features and their relationship to actions and results are important for game agents. Even more crucial is the comprehension of temporal correlations, i.e. the problem of assigning rewards correctly to the appropriate actions.
- **Resource Management:** Often entire games are based on gathering and using resources. Therefore, finding a balance between investments in economy, military and technology is an important part of these games.
- **Multi-Agent Systems:** RTS games are inherently multi-agent environments. At the highest level, this can be the cooperation between several game playing agents controlling their own armies. In this case, agents have to be able to rationally coordinate their actions with or against another player.
- **Pathfinding:** Finding their way through 2D and 3D terrain remains a predominant problem in RTS games, mostly because of the great number of moving and stationary objects. However, this is also the area in which academic research, originally in the form of the A* algorithm and more recently using more varied techniques such as those described in Van Den Berg et al. (2008) and Van Den Berg et al. (2011) has made the most impact on commercial RTS games.

When looking at these concrete tasks involved in playing a RTS game, they logically fall into distinct categories such as strategy, economy, tactics or reactive manoeuvres. The categories can in turn be grouped into hierarchical layers (Weber et al., 2011). For the creation of an AI that plays a RTS game, these tasks and layers are used to subdivide the overall problem into areas of responsibility for certain parts of an agent (or ‘bot’) architecture.

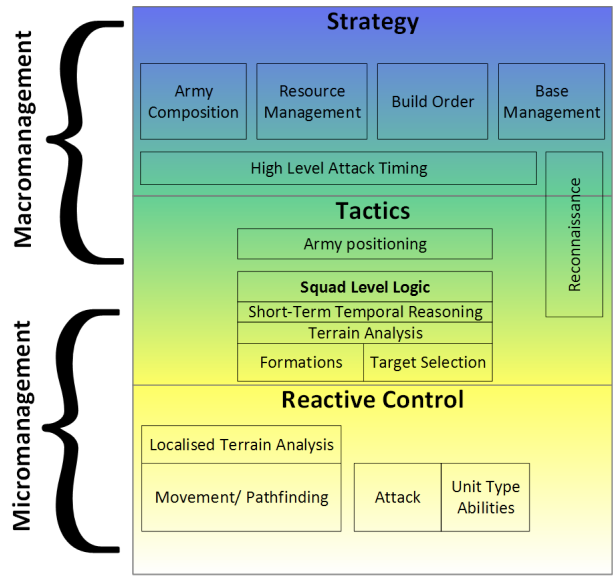


Figure 3.1: RTS Game Layers and Tasks

Figure 3.1 shows the subdivision of an RTS into tasks and layers involved in playing the game. The strategic layer contains tasks related to the game economy, such as base building, technology tree management (build order) and resource management. High-level strategic decisions also include army composition and high-level attack timing.

The tactical layer mostly involves the coordination among different numbers of units. As such, it is concerned with army positioning and squad actions while also taking into account the map terrain. Squad actions involve the coordination among groups of units. This can be in terms of movement or in terms of other actions such as attacking. Tactical decisions also have a temporal component but it is only very short-term. Strategy-level decisions in contrast involve long-term planning. A cross-level task is reconnaissance or scouting, i.e. discovering an opponent’s base, economy, army composition and unit movement. RTS games such as StarCraft have incomplete information due to the *fog of war*, an effect which hides any area on the map that is not in proximity to a players units or buildings. This makes

scouting crucial for strategic decisions as well as for tactical unit movement to avoid being overwhelmed by a surprise attack.

On the lowest level, RTS games are about reactive control of single units. This involves terrain analysis for movement or attack actions.

Layering in RTS games leads to most RTS agents being inherently hierarchical. Agents usually have a top-down approach when it comes to giving orders or executing actions and plans (Ontañón et al., 2013). A high-level planner will decide on an action, which will subsequently be passed to subordinate managers who in turn might pass on parts of these actions to managers who are even further down in the hierarchy, potentially as far as a per-unit level.

3.1.2 RTS Games as Testbeds for AI Research

This section explores the historic development as well as more recent instances of the use of RTS games, both commercial and other, as testbeds for AI research. It first investigates the general considerations behind choosing a commercial or non-commercial RTS game as a testbed. These considerations extend beyond RTS and apply to other game genres as well. In the second part of this section, the reasoning behind choosing the RTS game *StarCraft* as testbed is explained. This includes an analysis of how the AI problems contained in RTS games identified in Section 3.1.1 manifest in *StarCraft*.

The relationship between commercial games and academic game AI research is a complex topic and still evolving (Muñoz-Avila et al., 2013). While there are techniques from AI research such as A* (Hart et al., 1968), which have made their way into commercial games, this is the exception to the rule in an area of software development that mostly relies on older and simpler methods (Robertson & Watson, 2014). Commercial games have the advantage of being played and thus tested by a large community of players. After having been developed by a company in a thoroughly vetted process, players, as paying customers, have an interest in receiving an error-free product that is maintained even after release. Furthermore, those players form communities which produce new content for games in the form of maps and add-ons. Additionally, players can be engaged as testers for academic AI research that integrates into their game. Players are also the source of game traces in the form of recorded games against other players.

Commercial games are often considerably more complex than custom-built AI testbeds which frequently have only very limited and narrowly defined functionality. This is due to the different requirements for the two types of simulation environments and also due to the amount of (financial) effort invested in their development. Commercial games must have a minimum level of complexity in order to be enjoyable for a long time. Even comparat-

ively simple commercial games usually offer more elaborate gameplay than games developed as academic AI testbeds. Furthermore, large amounts of time and money are invested in fine-tuning usability in commercial games. In academic testbeds, usability comes second to internal logic and functionality in terms of the specific simulation purpose. When they are used for academic objectives, the superior level of complexity in commercial games on the one hand gives a wide range of interesting problems to solve. On the other hand, the large complexity also adds a lot of noise, i.e. unimportant information, which has to be excluded during experimental evaluation.

Games which are custom-made for research only exhibit features that were explicitly included, leading to no unforeseen behaviour, except outright errors. Further, the game can be specifically optimized for the chosen approach. However, for significant modifications to the approach, the game will usually have to be modified as well. Even though using a custom environment eliminates unwanted noise from experiments, real-world problems often also contain noise. Thus, commercial games exhibit a more realistic representation of many problems than custom-made test environments do.

A major limiting factor for the suitability of a commercial game as testbed for AI research is the availability of a suitable interface. Nearly all commercial games are closed-source and therefore do not offer access to most of their core functionality. For these games, only the built-in editors which often come with the games can be used. More recently, creators of commercial games have added powerful editing tools which players can use to create their own content and modifications ('mods'). Examples of commercial games that come with powerful content creation tools include first-person shooters (FPSs) such as the games in the *Halo* series (Järvinen, 2002) and role-playing games (RPGs) such as *The Elder Scrolls: Skyrim* (Bethesda Game Studios, 2011). More recent RTS games such as Blizzard Entertainment's *Warcraft III* and *StarCraft II*, also offer their own powerful scripting language which enables users to build add-ons and modifications, without interfering with the game's core game elements. On some occasions, these editors can be powerful enough to enable the games to be used as testbeds (Szczepański & Aamodt, 2009). However, the editor usually severely limits the possible interactions with an environment. The major advantage of using scripting and modification ('modding') tools that are developed specifically to enable the users of a games to create new content is usability. The tools are created according to the same standards as the multi-million dollar games which they are a part of.

Another way to interface with a commercial game is the direct manipulation of its source code. Several developers of commercial games have released the source code of their games for enthusiasts to work with. *id Software* routinely release the source code of their game engines as open-source, thus making available the engines of popular games such as the *Doom* series and the *Quake* series (id Software, 2011). *Firaxis Games*, maker of the the

popular turn-based strategy game *Civilization IV*, released large parts of the game's code base as open-source, including that part which controls the AI (Firaxis Games, 2007). The effort required to use this complex low-level code is much higher when compared to mapping and modding tools since source code often comes without any manual, let alone a customized user interface. However, for researchers who need complete control over their test-environment, these games offer very interesting solutions.

Developing custom games has become easier and more convenient with the availability of open-source game engines such as those used for the *id Software* games from the *Quake* and *Doom* series. More recently, game development tools such as the *Unity* game engine offer even more integrated capabilities such as physics and pre-defined in-game logic, thus further simplifying game development (Craighead et al., 2007). However, developing a game environment still requires a huge amount of effort while not doing any actual research. Possible errors will also have to be found and resolved by the researchers themselves as they cannot rely on a large community or a supportive game developer.

Another problem with custom-developed games is that, due to the very specific characteristics of the game, the team that develops the game will usually remain the only one to use it. Other researchers who want to use games as testbeds are likely to shy away from the effort needed to adapt the game to their own purposes. The effort required to adopt a custom research game could potentially be similar to adapting a more complex and powerful commercial game to their needs or even to the effort required to create a custom game of their own.

The *Open-Source RTS Environment (ORTS)* developed by Buro (2003a) is one notable effort to solve this problem of creating a unified interface for RTS games which can be used in AI research. It was created as a hack-free open-source game engine with which AI approaches can be tested in a secure, standardised environment (Buro & Furtak, 2005). In contrast to the client-side implementation of popular commercial RTS games, the ORTS architecture was created as a server-side RTS simulation to which people can connect any software client they like. The game information is processed only by the server, which prevents cheating. Over the years, different kinds of AI approaches have been integrated into the environment and the functionality was expanded (Buro & Furtak, 2004; Chung et al., 2005). ORTS competitions, where researchers could compete with their clients against others, were held at *AIIDE* conferences for a number of years. Naveed et al. (2011) used ORTS to empirically evaluate their path finding technique which is based on a combination of Monte-Carlo tree search with the randomized exploration capabilities of rapidly exploring random tree (RRT) planning.

Chapter 3. Background

Stoykov (2008) used ORTS to show how a competitive approach to AI design can improve the quality of AI in military simulators. The author used existing ORTS clients in a competition and subsequently evolved these clients by changing their underlying algorithm in order to improve their competitiveness.

Hagelbäck & Johansson (2008) proposed the use of multi-agent potential fields for navigating units in RTS. They evaluated their approach in ORTS using predefined *Tankbattle* and *Tactical combat* scenarios. In the 2007 ORTS competition the bot was in the bottom half of the entries, managing to win about one third of all its games.

The *Testbed for Integrating and Evaluating Learning Techniques (TIELT)* (D. W. Aha & Molineaux, 2004) is another environment conceived in an attempt to create a standard for AI research. Specifically, TIELT was designed to function as testbed middleware that enables ML research to use simulation environments such as RTS games. TIELT was developed to provide composable interfaces to game engines and reasoning systems and has been used together with a number of game environments.

Karpov et al. (May 2006) used the *NeuroEvolution of Augmented Topologies (NEAT)* algorithm (K. Stanley et al., Dec. 2005) to generate strategies for bots that play the FPS game *Unreal Tournament*. The authors performed their experiments using TIELT as an integration platform and also included an evaluation of the time and effort required to adapt a commercial game for use as testbed in AI research. As a test-case, the task of navigating through a level in *Unreal Tournament* was used.

Both Souto (2007) and Gundevia (2006) described the integration of the commercial Civilization game *Call To Power 2 (CTP2)* with TIELT. Their overall aim was to transfer learning between different scenarios, but the test environment could also be used as an integrated testbed for future research using CTP2. TIELT integrated with CTP2 was used by Sánchez-Ruiz et al. (2007) as a testbed for adaptive game AI. In order to communicate with the game, an ontology for the domain was developed and case-based planning in combination with CBR was used to create an adaptive AI.

One of the first attempts to create a simulation environment for AI research in RTS games was *Stratagus* (Wen, 2004). *Stratagus* is a community project that serves as a RTS engine on which games with open and modifiable source code could be created. On top of the *Stratagus* engine, *Wargus* was created, a game replicating the gameplay and appearance of the popular RTS game *Warcraft II* (The Wargus Team, 2004). *Wargus* has been used for research in numerous publications on game AI (D. Aha et al., 2005; McCoy & Mateas, 2008; M. Ponsen, 2004; Jaidee & Muñoz-Avila, 2013). Until BWAPI was released and the use of *StarCraft* as a testbed for AI research acquired a certain level of popularity, *Wargus* was probably the

most popular RTS environment to be used as testbed for AI research. This was due to its open-source engine, its relatively large complexity, the fact that it exhibited all the standard characteristics of a RTS game and that it closely resembled the popular Warcraft II game.

3.1.3 StarCraft as a Domain for AI Research

StarCraft is a 17-year old RTS game that, despite its age, remains popular to this day. In spite of the outdated graphics and strong competition from newly-created RTS games, even in the form of its own official successor, StarCraft remains the best-selling RTS game of all time and still has an ardent player base. With the development of a programming interface, the *Broodwar API (BWAPI)* from 2009 onwards (BWAPI, 2009), StarCraft became one of the first commercial RTS games with a completely modifiable core functionality. This has since led to StarCraft becoming a de-facto standard simulation environment for many aspects of RTS game AI research. This section examines how the characteristic RTS game problems feature in StarCraft. The section also gives a more in-depth description of BWAPI, its workings and its effects on the research community.

General Gameplay

StarCraft is an RTS set in a science-fiction scenario in which the player assumes the role of a military commander in attempting to eliminate all opponents in a complex strategy simulation. A major feature that sets StarCraft apart from other RTS games are the three playable factions of the game. *Terrans*, *Protoss* and *Zerg* all have completely different units, buildings and, most importantly, very different styles of play. Despite these differences, the three factions were very well balanced. Playing StarCraft requires actions and considerations on several different levels of reasoning. These tasks include building an economy that rapidly acquires resources and then using these resources to set up a base. In parallel, additional technologies are researched to unlock units and techniques on the *technology tree* and units are produced to form an army that is then employed to attack opponent units and bases. The scope and content of these tasks, both in general and in their specific implementation in StarCraft, are explained in detail in subsequent sections.

This thesis focuses on the micromanagement aspect in a 1vs1 environment, i.e. one player against a single opponent. StarCraft also allows teams of players to compete against each other, which again increases the complexity involved in the game. The limitation to micromanagement excludes higher-level aspects such as base-building, developing an economy and researching new technologies. Instead the agent is provided with a set number of units to control.

In general, the action- and decision space in StarCraft are very big. Weber (2012) estimates the complexity based on the possible numbers of units for all players (1700), unit types (9) and map sizes (256*256 plots) at $(100 * 256 * 256)^{1700}$, or roughly $10^{11,500}$. This also counts illegal game states such as overlapping units, however ignores many unit actions and is orders of magnitude above the complexity of classic games such as chess, which is estimated to have a complexity of 10^{43} by Shannon (1950).

Characteristic RTS Features in StarCraft

StarCraft features all characteristic RTS problems of interest to AI research listed in Section 3.1.1. Below is a list of these AI problems, how they manifest within the game and thus enable potential contributions from AI research.

- **Adversarial Planning:** There are three entirely different factions, each with a number of distinct units, buildings and technologies. This results in a very large number of possible high-level strategies and an even larger number of low-level micromanagement actions. Furthermore, each player controls large groups of units and there can be up to twelve players in a game. As a result, the complexity of StarCraft is immense and even basic applications of AI techniques require a high level of abstraction.
- **Incomplete Information:** StarCraft implements a ‘Fog of War’ which initially covers the whole map. Once an area has been scouted, landscape features as well as buildings of enemy players remain visible, even if the area drops out of the visibility range again. However, this only shows the state of that part of the map at the time of the scouting. Any possible changes have to be scouted again, thus requiring constant intelligence-gathering to enable the agent to make informed decisions.
- **Learning and Opponent Modelling:** The built-in game AI uses deterministic methods to decide its actions. No learning is involved. Fundamental flaws in the AI which have been found to date had to be fixed by patches after launch (teamliquid, 2011). StarCraft is thus an ideal testbed for learning and adapting AI techniques as its built-in AI poses a noteworthy yet deterministic opponent for any newly developed agent.
- **Spatial and Temporal Reasoning:** StarCraft maps are generally 2D but contain up to three predefined layers of height. Connections between these layers (so called ‘Choke’ points) form an integral part of many strategies. Higher layers are not visible from the below. StarCraft games are, much like other RTS games, usually split into the three (colloquially named) phases ‘Early Game’, ‘Mid Game’ and ‘Late Game’. These refer to the time that has been played and are also used to indicate what types of units are used, i.e. how much of the tech-tree is accessible to the players and how far their

economy is advanced. Agents have to be able to relate the time that has passed since the game started to the possible state their opponent's army is in as well as to their own building possibilities.

- **Resource Management:** StarCraft requires a constantly updated weighting in terms of extending resources gathering capabilities. As the game is based on gathering and using resources, finding a balance between investments in economy, military and technology is an important part of the game.
- **Multiple Agents:** Most commonly, StarCraft is played by one player against one opponent. However, there can be up to eight players playing at the same time in teams of different sizes or in a free-for-all type of play. Teams can consist of both human and AI players, however the built in AI does not cooperate well with human players. There are large numbers of individual units for each player, each of them potentially an individual agent.
- **Pathfinding:** StarCraft uses a variation of the common A* algorithm for pathfinding. However, the pathfinding runs into problems frequently. The reason is that while the algorithm works well with static objects like buildings and landscape objects, it has problems taking into account moving units. StarCraft II, the sequel to StarCraft, performs much better in terms of pathfinding as it combines A* with a number of other techniques such as flocking and floating fields (teamliquid, 2010).

The Broodwar API

Probably the most important factor in enabling research with StarCraft as testbed is BWAPI, the Broodwar API (BWAPI, 2009). BWAPI is an interface that allows to access internal functions and information in the StarCraft source. Several people in the StarCraft community were not satisfied with the capabilities of the original map editor that came with the game. Initial development on a community project that reverse-engineered the StarCraft memory management started in 2008. This resulted in an API that directly manipulates the StarCraft program memory through C++ functions, effectively simulating game logic.

Eventually, this interface enabled the control of most aspects of the game. BWAPI allows programmers to retrieve information on units and players in the game. It furthermore allows players to issue commands to units, thus enabling the development of a wide variety of AI modules. By default, only visible areas are shown to a player, i.e. imperfect information is maintained. However, it is possible to obtain all existing information for an agent to use. Furthermore, the interface allows to speed up games or even to disable the GUI part entirely, leaving only the computation running in the background. Using this method, large

numbers of experiments can be easily run pitching different or similar bots against each other. Despite these capabilities, BWAPI is still only a plug-in for the actual StarCraft game code. There are certain aspects that cannot easily be manipulated, especially in terms of high-level functionality, since only functions predefined in BWAPI can be used.

The first research conference competition in which AI agents using a version of BWAPI could compete against each other was announced in 2009 and played in 2010 (Expressive Intelligence Studio, 2010). Competitions between StarCraft bots have since become regular events at multiple annual conferences (Ontañón et al., 2013). More recent instances of these competitions require participants to publish the code of their agents and thus enable re-use and advancement of the overall approaches. The level of skill in the bots that participate in these competitions is still well below even medium-level human players, but is improving each year (Ontañón et al., 2013). Section 3.2 provides an overview of the ML approaches used in some of the bots involved in these competitions.

Micromanagement in StarCraft

The decomposition of tasks in a RTS game presented in Section 3.1.1 and displayed in detail in Figure 3.1 shows that all sub-problems that are identified as being part of an RTS game can be grouped into two high-level categories: *Macromanagement* and *Micromanagement*. Macromanagement includes mostly strategic tasks such as base-building and general build trees, whereas micromanagement focuses on low-level tactical and reactive tasks. There are a number of surveys on ML in RTS games that analyse which techniques are more prevalent for which problem area. Lara-Cabrera et al. (2013) find that the majority of research using StarCraft as a problem domain focuses on high-level planning tasks and opponent modelling as mentioned in Chapter 1.

The dominance of high-level tasks in research in RTS game AI is confirmed by Robertson & Watson (2014), who also list case-based and evolutionary methods as popular approaches in this area. Both surveys find that RL and RL-related search-based methods such as MCTS are more common for low-level logic in the micromanagement bracket.

Ontañón et al. (2013) survey StarCraft bot architectures for participants in the annual competitions at the CIG and AIIDE conferences in 2012. All bots must be able to play the entire game and thus must have modules that address all problems listed in Figure 3.1. However, for bots presented in that survey and examined in detail in Section 3.2, the majority that uses ML approaches for any of the sub-problems employs these approaches for high-level logic rather than low-level reactive reasoning. Low-level tactical and reactive components are commonly controlled either by scripted behaviour or by simple deterministic approaches such as FSMs and decision trees.

There are a number of attributes that make micromanagement in StarCraft an interesting problem domain. An important reason is the overall complexity of the problem. Micromanagement requires a large number of actions over a short amount of time. It requires very exact and prompt reactions to changes in the game environment. The problem involves tasks like damage avoidance, target selection and, on a higher, more tactical level, squad-level actions and unit formations. All of these tasks involve, depending on the particular RTS environment in use, the coordination of large numbers of very different units.

In terms of the interesting AI problems mentioned by Buro (2003b), micromanagement can include incomplete information, machine learning, spatial and, to a lesser degree, temporal reasoning and pathfinding. Any micromanagement problem that involves more than a single unit is inherently a multi-agent environment. Micromanagement is also central to expert gameplay (Churchill & Buro, 2013).

All of the requirements are made more difficult by the use of a commercial game that, while enabling complete access to its functionality, not always reacts in the most exact and prompt fashion. As stated in Section 3.1.3, while the BWAPI interface theoretically gives access to most of the game functionality, it is not comparable to an open-source engine. Given the overall complexity of the problem, it is quite logical that the built-in AI in StarCraft is hard-coded, non-adaptable and simple. Its skill level is comparably low and even amateur human players can easily defeat it. As such, the built-in AI can serve as a base-line comparison for ML in the domain, that should be beatable through adaptive techniques.

Together with the previously mentioned factors, this contributes to making micromanagement in StarCraft a challenging yet rewarding domain for ML approaches.

3.2 RTS Game Bot Architectures

This section examines a selection of StarCraft bot architectures as primary example of the application of hierarchical architectures that are used in ML approaches for RTS game AI. Often, these agents are accumulations of different modules that are interconnected. Because of the complexity of the problem domain, these connections are across multiple layers in hierarchical structures. Many of the different approaches to creating a game-playing agent in RTS games do not necessarily involve adaptive ML techniques. Even agents that use ML techniques to solve some parts of the overall problem will usually use scripted knowledge in other parts. This section compares several bot architectures that employ machine learning approaches as part of their problem solving modules. The architectures are from bots that have been analysed by Ontañón et al. (2013). These are bots that participated in the two StarCraft AI competitions at the *CIG* and *AIIDE* conferences in 2012. While Ontañón et al. (2013) look mostly at performance and low-level implementation, the focus in this section lies on decomposing the approaches into the task-solving modules according to the RTS game model shown in Section 3.1. For each of the three layers for strategy, tactics and reactive control, the technical approaches are identified for the respective bots. This identification serves to point out trends and prevalent techniques in bots that address the entire game.

Table 3.1 lists the approaches by bot, distinguished by which of the three distinct hierarchical layers they fall into (Ontañón et al., 2013). As the entries show, there is a noticeable prevalence of script-based components. Similar to commercial game AI (Robertson & Watson, 2014), even researchers that build agents to test new ML approaches resort in many instances to scripted behaviour. This might be at least partially due to performance reasons. This selection of bots, is originally based on participants in 2012 StarCraft competitions. The winner of the competitions at both the *AIIDE* conference and the *CIG* conference was *Skynet*, which consists of only deterministic scripts based on expert knowledge (Ontañón et al., 2013). However, this does not mean that adaptive approaches can not achieve similar performance. In 2013 *UAlbertaBot*, a bot that uses heuristic search for both build-order selection and combat simulation, managed to win the *AIIDE* competition. *Skynet* retained its title for *CIG* since the conference did not allow bots to store knowledge on persistent memory, a central requirement for *UAlbertaBot*'s performance (Churchill & Buro, 2013).

The two most relevant bot architectures for the approach pursued as part of this thesis, also indicated through the coloured cells, are *UAlbertaBot* and *BroodwarBotQ*. Both of these use similar adaptive approaches across more than one layer of the problem domain. *UAlbertaBot* is able to adapt dynamically online by using techniques based on game tree search to produce impressive results in both strategic and tactical tasks (Churchill & Buro, 2011, 2013). While *BroodwarBotQ* is mostly trained offline, it is closer related to the approach in this thesis since

Bot	Layer and Approaches		
	Strategy	Tactical	Reactive
<i>AIUR</i> (Richoux, 2014)	Random selection of one of six pre-defined ‘Moods’ that influence build order scripts; A <i>SpendManager</i> can slightly adapt build orders according to perceived game states.	Moods also influence pre-defined tactical scripts.	Scripted reactive behaviour.
<i>BroodwarBotQ</i> (Synnaeve & Bessiere, 2011b)(Synnaeve et al., 2012)	Scripted build-orders; Selection based on Bayesian predictions. Arbitrator for resource bidding.	Bayesian decision maker trained offline using replay data.	Bayesian unit controller subsuming potential fields. Hand-specified parameter settings.
<i>BTHAI</i> (Hagelbäck, 2012)	Scripted build-orders.	Scripted squad formations.	Individual unit control is based on a combination of different potential fields and A*.
<i>Nova</i> (Uriarte & Ontañón, 2012) (Pérez, 2011)	Blackboard architecture and multi-agent system. FSM strategy manager, makes decision according to opponent prediction.	Scripted tactical behaviour, uses flocking, which also extends to the reactive layer.	Influence maps for kiting.
<i>SPAR</i> (PLANIART Lab, 2012)	Believe-based system that selects scripts according to plans developed through perception of the game state.If-Then implementation.	Hard-coded group behaviour.	FSMs/statecharts.
<i>Skynet</i> (Q. Smith, 2012)	Scripted build order manager.	Sophisticated, hard-coded scripts.	Hard-coded reactive behaviour; sophisticated kiting and manoeuvring out of area-of-effect (AOE) damage.
<i>UAlbertaBot</i> (Churchill & Buro, 2011)(Churchill & Buro, 2013)	Search-based build order manager.	UCT-based simulation of combat outcome.	Low-level action scripts executed according to tree-search results.

Table 3.1: Bot Architecture AI Techniques. Green Cell = Online Adaptive AI. Light Green Cell = Partially Adaptive AI. White Cell = Static AI.

it uses a ML approach and addresses both hierarchical and tactical tasks directly. However, neither of these two nor any of the other approaches attempts to address these tasks by using an integrated holistic approach as proposed in this thesis. In addition, none of the architectures or modules presented here solves their problems using online acquisition of knowledge, thus making it an interesting and untested problem at the scale it is proposed.

3.3 Reinforcement Learning

The use of RL for micromanagement in RTS games was inspired by previous work on using RL for city-site selection in the turn-based strategy game Civilization IV (Wender, 2009). This similarity leads to parts of this section being influenced by content created for that work. The part of the section that details the early origins of RL is partially based on Sutton’s and Barto’s seminal survey on RL (R. S. Sutton & Barto, 1998), as are the detailed descriptions of the specific RL algorithms.

3.3.1 Origins of Reinforcement Learning

The modern field of reinforcement learning was introduced in the late 1980s and evolved mainly from two different branches of research: optimal control, and learning by trial and error. A third branch, which also contributed to the development to a smaller degree, is *temporal-difference* (*TD*) learning.

Optimal control is a task which is used to describe the search for a controller that minimises a specific variable in a dynamic system over time. One of the most prominent approaches to this problem was developed by Richard Bellman in the 1950s, the so-called *dynamic programming* (R. Bellman, 1957a). To this day, this technique is still used to solve RL problems. R. Bellman (1957b) also developed the notion of the *Markovian Decision Process* (*MDP*) which is the discrete stochastic version of the optimal control problem. Both dynamic programming and MDPs form a vital part of what is today known as RL.

The second big branch of RL, learning by trial and error, is based on the *Law of Effect* in cognitive sciences: If an action is followed by a positive reward, it is subsequently remembered in this context and thus more likely to be performed again in the same situation. An action followed by a negative reward is remembered in a negative context and will be avoided (Thorndike, 1911). The *Law of Effect* is selective, i.e. it tests numerous different options and chooses based on consequences. The law is also associative in that a trialled option will be associated with a particular situation. One of the first attempts at teaching a computer program through trial-and-error learning was made by Farley & Clark (Sep 1954).

The third, more recent branch which of modern RL is TD learning. The first application of a form of TD learning, by analysing the changes of a certain attribute between time-steps, was done by Samuel (1959). Witten (1977) was the first to integrate optimal control and trial-and-error learning. He also made a significant contribution to TD learning, an area which had received little attention since its initial discovery in the late 1950s.

The final step to create modern day RL was the development of Q-learning by Watkins (1989). His work integrated all three previously described branches and extended the field of RL in general. One of the first successful and, to this day, one of the most noteworthy applications of these techniques was performed by Tesauro (1992), who brought further attention to the emerging field of RL with his backgammon player *TD-Gammon*.

3.3.2 Reinforcement Learning Algorithms

This section describes the general algorithms and concepts which are used throughout this thesis in detail. First, the notions of the *Markov property* and *Markov decision processes* are explained. These are important since they are prerequisites for the efficient use of RL techniques. The idea behind TD learning is described subsequently. The four TD algorithms that are evaluated in Chapter 4 are based on this technique. The two specific underlying algorithms, *Q-learning* and *Sarsa*, are explained in detail afterwards. After explaining the simple, *one-step* versions of these algorithms, the more complex variants which include *eligibility traces* are shown in detail.

The Markov Property

The agent in a RL framework makes its decision based on the information it gets about the environment at a certain point in time, the so called *state*. If this state signal contains all the information of present and past sensations it is said to have the *Markov Property*. This does not mean that single actions leading up to the present state have to be observable, only the important parts for the current state have to be present. Mathematically, the complete probability distribution for the response of an environment at time $t + 1$ to an action taken at time t is

$$Pr \left\{ s_{t+1} = s', r_{t+1} = r | s_t, a_t \right\}.$$

If an environment has the Markov property, this also means that, given the current state and action, it is possible to predict the next state and reward. Through iteration this allows to predict all future states. Furthermore, choosing a policy based on a Markov state is just as effective as choosing a policy when knowing the complete history until that state. In RL, Markov states are important since decisions are only made based on the current state. Even if

a RL environment does not have the Markov property, the standard algorithms which assume this property can still be applied. However, these algorithms will usually only be as effective as far as the state signal resembles a Markov state.

Markov Decision Processes

If a reinforcement learning task presents the Markov property it is said to be a *Markov Decision Process (MDP)*. If the task has a finite number of states and actions it is called a *finite MDP*. A specific finite MDP is defined by a quadruple $(S, A, \mathcal{P}_{ss'}^a, \mathcal{R}_{ss'}^a)$. In this quadruple

- S represents the state space,
- A represents the action space,
- $\mathcal{P}_{ss'}^a = Pr \left\{ s_{t+1} = s' \mid s_t = s, a_t = a \right\}$ represents the transition probabilities, i.e. the probability of reaching state s' from s after taking action a .
- $\mathcal{R}_{ss'}^a = Pr \left\{ s_{t+1} = s' \mid s_t = s, a_t = a \right\}$ represents the expected reward, given a current state s , an action a and the next state s' .

The MDP is used to maximize a cumulative reward by deriving an optimal policy π^* according to the given environment.

Temporal-Difference Learning

TD learning is a RL method which combines ideas from dynamic programming with Monte-Carlo methods (R. S. Sutton, 1988). Similar to Monte-Carlo methods, it samples the environment to observe the current state without the need of a complete model of this environment. Both methods update their estimated value V of a visited state s_t based on the return after visiting the state while following a policy π . Similar to dynamic programming, the estimation is thus based on previous estimates, the so-called ‘bootstrapping’. The pseudocode for the simple TD algorithm is shown in Algorithm 1.

The Q-Learning Algorithm

Q-learning is an off-policy TD algorithm. Off-policy means, it calculates the values of policies not only based on experimental results but also based on estimates about values of hypothetical actions, i.e. actions which have not actually been tried.

The value update formula for simple *one-step Q-learning* is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \quad (3.1)$$

```

Initialise  $V(s)$  arbitrarily and  $\pi$  to the policy to be evaluated
for (each episode) do
  Initialise  $s$ ;
  repeat for each step of episode
     $a \leftarrow$  action given by  $\pi$  for  $s$ ;
    Take action  $a$ ; observe reward,  $r$ , and next state,  $s'$ ;
     $V(s) \leftarrow V(s) + \alpha [r + \gamma V(s') - V(s)]$ ;
     $s \leftarrow s'$ ;
  until  $s$  is terminal;

```

Algorithm 1: A Simple TD Algorithm for Estimating V^π

Since Q-learning works independent of the policy being followed, the learned action-value Qfunction directly approximates the optimal action-value function Q^* . In contrast to simple TD, it does not assign a value to states, but to state-action pairs. The only requirement it has to guarantee the discovery of the optimal policy π^* is that all states are visited infinite times, which is a basic theoretical requirement for all reinforcement learning methods that are guaranteed to find the optimal behaviour.

The procedural form of the Q-learning algorithm can be seen in Algorithm 2.

```

Initialize  $Q(s, a)$  arbitrarily;
for (each episode) do
  Initialise  $s$ ;
  repeat for each step of episode
    Choose  $a$  from  $s$  using the policy derived from  $Q$ ;
    Take action  $a$ , observe  $r, s'$ ;
     $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$ ;
     $s \leftarrow s'$ ;
  until  $s$  is terminal;

```

Algorithm 2: Pseudocode for One-Step Q-Learning

The Sarsa Algorithm

Sarsa is an on-policy TD learning algorithm very similar to Q-learning (Rummery & Niranjan, 1994). The main difference is the selection of the action for the subsequent state. While Q-learning always chooses the action with the biggest reward, Sarsa always selects the action for the next state according to the same policy that led to the present state.

Chapter 3. Background

Sarsa stands for **State-Action-Reward-State-Action**, more specifically the quintuple referred to here is $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. This implies that, to compute the update for a Q-value, the following information is needed:

- The current state s_t .
- The chosen action a_t according to a policy π .
- The reward r_{t+1} gained from executing this action.
- The resulting new state s_{t+1} .
- The next a_{t+1} action that is chosen the policy π .

Sarsa was developed as an alternative to the off-policy Q-learning which always chooses the action yielding the maximum reward. Sarsa allows for a more controlled trade-off between exploitation (taking the highest yielding action) and exploration (picking random/unknown actions). The update function for Q-values is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \quad (3.2)$$

This function makes use of every element in the quintuple described above. In its general form Sarsa looks as shown in Algorithm 3.

```
Initialise  $Q(s, a)$  arbitrarily;
for (each episode) do
  Initialize  $s$ ;
  Choose  $a$  from  $s$  using the policy derived from  $Q$ ;
  repeat for each step of episode
    Take action  $a$ , observe  $r, s'$ ;
    Choose  $a'$  from  $s'$  using the policy derived from  $Q$ ;
     $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$ ;
     $s \leftarrow s'$ ;
     $a \leftarrow a'$ ;
  until  $s$  is terminal;
```

Algorithm 3: Pseudocode for One-Step Sarsa

Eligibility Traces

Eligibility traces are a basic mechanism of RL that is used to assign temporal credit. Using eligibility traces means that not only the value for the most recently visited state or state-action pair is updated. The value for states or state-action pairs that have been visited within a limited time in the immediate past are also updated. The technique can be combined with

any TD technique and speeds up the learning process. Assigning temporal credit is an important issue in any RL problem.

TD(λ) is a popular TD algorithm which uses eligibility traces and which was developed by Sutton (R. S. Sutton & Barto, 1998). It was used by Tesauro (Tesauro, 1992) for his famous backgammon agent which learned to play on the same level as human players. The λ in TD(λ) is the so-called *trace-decay* parameter, i.e. the parameter which determines how far rewards propagate back through a series of states and actions. This parameter is used to compute the eligibility trace which is for state s at time t

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t. \end{cases}$$

To guarantee convergence towards the optimal solution, the limits are $0 < \lambda < 1$. In other words, lambda has to decay in order to make future rewards less important than present rewards. The pseudocode for TD(λ) can be seen in Algorithm 4.

```

Initialise  $V(s)$  arbitrarily;
for (each episode) do
  Initialise  $e(s) = 0$  for all  $s \in S$ ;
  Initialise  $s$ ;
  repeat for each step of episode
     $a \leftarrow$  action given by  $\pi$  for  $s$ ;
    Take action  $a$ ; observe reward,  $r$ , and next state,  $s'$ ;
     $\delta \leftarrow r + \gamma V(s') - V(s)$ ;
     $e(s) \leftarrow e(s) + 1$ ;
    forall the  $s$  do
       $V(s) \leftarrow V(s) + \alpha\delta e(s)$ ;
       $e(s) \leftarrow \gamma\lambda e(s)$ ;
     $s \leftarrow s'$ ;
  until  $s$  is terminal;

```

Algorithm 4: Pseudocode for TD(λ)

For both Q-learning and Sarsa, eligibility traces are not used to learn state values ($V_t(s)$) but rather values for state-action pairs ($Q_t(s, a)$), just as in the one-step versions of these algorithms. The pseudocode for the eligibility trace version of Sarsa(λ) can be seen in the figure for Algorithm 5.

There are two different popular methods that combine Q-learning and eligibility traces. They are called Peng's Q(λ) (Peng & Williams, 1994) and Watkins's Q(λ) (Watkins, 1989) after the people who first proposed them. Empirically, it has been shown that Peng's Q(λ) usually performs better and nearly as good as Sarsa(λ) (R. S. Sutton & Barto, 1998). However

```
Initialise  $Q(s, a)$  arbitrarily for all  $s \in S, a \in A(s)$ ;  
for (each episode) do  
  Initialise  $e(s, a) = 0$  for all  $s \in S, a \in A(s)$ ;  
  Initialise  $s, a$ ;  
  repeat for each step of episode  
     $a \leftarrow$  action given by  $\pi$  for  $s$ ;  
    Take action  $a$ ; observe reward,  $r$ , and next state,  $s'$ ;  
    Choose  $a'$  from  $s'$  using policy derived from  $Q$ ;  
     $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$ ;  
     $e(s, a) \leftarrow e(s, a) + 1$ ;  
    forall the  $s, a$  do  
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$ ;  
       $e(s, a) \leftarrow \gamma \lambda e(s, a)$ ;  
     $s \leftarrow s'; a \leftarrow a'$ ;  
  until  $s$  is terminal;
```

Algorithm 5: Pseudocode for Sarsa(λ)

Peng's $Q(\lambda)$ is far more complex to implement than Watkins's $Q(\lambda)$, has not yet been proven to converge to the optimal function Q^* , and is basically a hybrid between Watkins's $Q(\lambda)$ and Sarsa(λ). Therefore, only Watkins's $Q(\lambda)$ is used as Q-learning algorithm with eligibility traces in Chapter 4. The pseudocode for Watkins's $Q(\lambda)$ can be seen in Algorithm 6.

All four of the listed algorithms are evaluated for their use in RTS game micromangement in Chapter 4. Subsequently, Watkins's $Q(\lambda)$ is used in the AI module developed in Chapter 5, while Chapters 7 to 9 use one-step Q-learning because of its easier implementation and nearly identical performance when compared to Watkins's $Q(\lambda)$.

3.4 Case-Based Reasoning

This section gives a brief overview of the standard cycle of the case-based reasoning (CBR) methodology. The section is mostly based on the original definition of the CBR cycle in Aamodt & Plaza (1994). While the approach described here represents this originally defined standard application, the hybrid approach presented in Chapter 4 and in the different modules developed in Chapter 6 use versions of CBR that are altered to fit a specific purpose. These customised applications of CBR are described in the respective sections and chapters. Furthermore, the next Section 3.5 gives a more detailed description of the structure of hierarchical CBR.

CBR is a methodology in which a reasoner remembers previous situations similar to the current one and re-uses these experiences to solve new problems. The reasoner maintains experiences in a *case-base*, where each previous experience is stored as a case in that case-

```

Initialise  $Q(s, a)$  arbitrarily for all  $s \in S, a \in A(s)$ ;
for (each episode) do
  Initialise  $e(s, a) = 0$  for all  $s \in S, a \in A(s)$ ;
  Initialise  $s, a$ ;
  repeat for each step of episode
    Take action  $a$ ; observe reward,  $r$ , and next state,  $s'$ ;
    Choose  $a'$  from  $s'$  using policy derived from  $Q$ ;
     $a^* \leftarrow \operatorname{argmax}_b Q(s', b)$  (if  $a'$  then  $a^* \leftarrow a'$ );
     $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$ ;
     $e(s, a) \leftarrow e(s, a) + 1$ ;
    forall the  $s, a$  do
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$ ;
      If  $a' = a^*$ , then  $e(s, a) \leftarrow \gamma \lambda e(s, a)$ ;
      else  $e(s, a) \leftarrow 0$ ;
     $s \leftarrow s'; a \leftarrow a'$ ;
  until  $s$  is terminal;

```

Algorithm 6: Pseudocode for Watkins's $Q(\lambda)$

base. CBR has been formalised into a multi-step process displayed in Figure 3.2 (Aamodt & Plaza, 1994).

1. **Retrieve:** Given a target problem, a suitably similar case is retrieved from the case memory. A case usually consists of a case description and a case solution.
2. **Reuse:** The solution-part of the retrieved case is mapped to the new target problem, potentially involving some adaptation.
3. **Revise:** The new, adapted solution is tested in the real world and, depending on the outcome, revised to reflect the necessary changes.
4. **Retain:** After successfully updating the solution for the current target problem, the new case, consisting of original problem description and newly adapted solution, is retained in the case-base.

In many instances in this thesis where RL is used, the case description can be seen as equivalent to the RL-state and the solution can be seen as equivalent to the RL-action. Additionally, in the context the hybrid CBR/RL modules, the **Revision** is done through RL and equivalent to the update of the state-action values of a given action or solution.

The selection of a case, given a certain problem description, and the re-use of the solution of that case is based on the assumption that similar problem descriptions lead to similar solutions. During the **Retrieval** phase, the most similar case according to a specified similarity

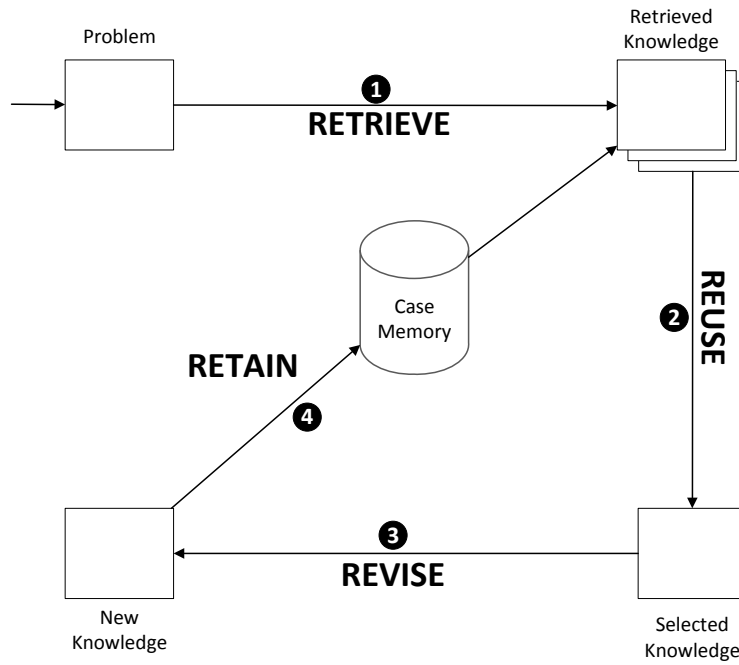


Figure 3.2: The CBR Cycle

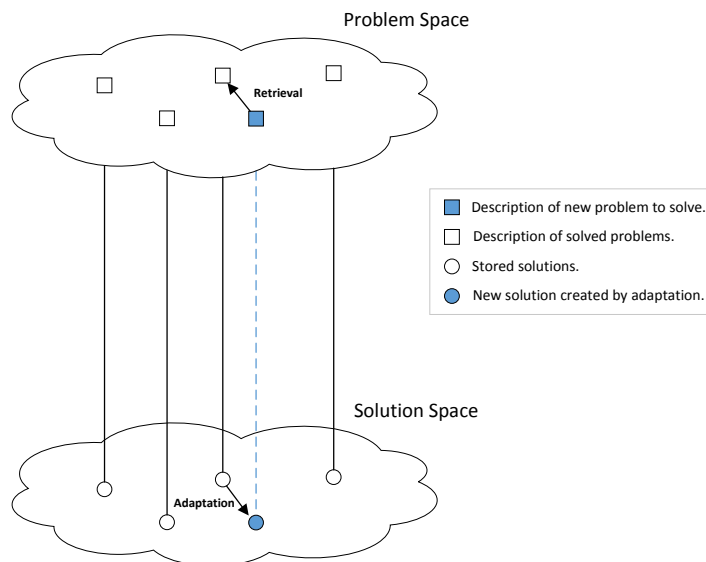


Figure 3.3: Case Retrieval

metric is retrieved from the case-base (see Figure 3.3). The specification of the similarity metric and the resulting retrieval mechanics are crucial aspects of the overall CBR component and decisive for its performance. The retrieval method that is used throughout the thesis is k-Nearest Neighbor (kNN) (Russell & Norvig, 1995). This retrieval is done in its simplest form with $k = 1$, i.e. only taking the closest neighbour into account.

There are many different similarity metrics to compute case similarity that have been developed for the various case representations (Liao et al., 1998; Cunningham, 2009). In general, cases in the CBR modules that are created as part of this thesis are represented by a set of features which translate to numerical values. A major challenge is the design of suitable normalisation functions for the features these values represent. A precise similarity computation using individually customised metrics for each attribute in a case description case is performed where it is computationally feasible, i.e. if there are only a very limited number of cases. These computations are described in the respective chapters. However, in certain situations the number of cases in the case-base is too big to compare them attribute by attribute. In other situations, the case descriptions have too many features to compare them each individually. This is where a more abstract way of comparing case similarities is required. The following two sections give brief summaries of the two abstraction methods that were used in parts of this thesis where case description dimensionality and case numbers required a modified similarity computation.

3.4.1 Histogram-Based Similarity Computation for IMs

Histograms serve as graphical representations and abstractions of the distribution of numerical data (Pearson, 1894). Each value in a distribution is put into a ‘bin’, an interval of the overall range of values of a certain size. The size of the intervals can be freely chosen and determines the accuracy. A particularly popular application of histograms is the use for colours in images (Jain, 1989). A *colour histogram* represents a digital image by plotting the number of pixels for each particular colour. These histograms are flexible in that they can work with different colour spaces and can also be adjusted in how wide each colour bin is. Comparing colour histograms can be very useful for object recognition as they are relatively invariant towards translation and rotation around an image.

Davoust et al. (2008) use fuzzy histograms to abstract spatial objects that robotic soccer players perceive in their environments while trying to navigate these environments. The environment is represented as a two-dimensional grid on which objects that the agents’ vision picks up are placed. The objects are placed in certain grid cells only to a certain degree, hence the fuzziness. Situations that are perceived like this are stored in memory as cases in a CBR system. For interchangeable objects in the given scenario, grid cells are thus simply defined through the number of objects contained in them. The authors can

then use a histogram to with each bin representing a certain object count. As there are non-interchangeable object such as players and the ball the authors use one histogram for each object class. The experimental results of using this scene representation against related approaches in a robotic soccer scenario leads to better results while requiring much less computational effort for scene comparison.

The CBR component in Chapter 5 works with case descriptions that consist only of large-scale influence maps and is thus well suited for this type of similarity metric. The number of grid cells in a single IM and thus the number of features in a case description can be very large, potentially 256×256 . The approach by Davoust et al. (2008) for abstracting the scene representation in the robotic soccer domain translates well to the problem of abstracting spatial information in the form of influence maps. Combined with the concept of colour histograms, different influence maps for the same spatial area can be regarded as different colour channels in an image. This results in a single histogram that encodes all spatial information in multiple IMs.

In order to directly compare two histograms that form case descriptions, there are a number of possibilities. Davoust et al. (2008) use a computationally complex method based on the *Jaccard coefficient* and histogram intersection. The approach presented in Chapter 5 uses the more straightforward *Correlation* metric. Both approaches are standard histogram comparison metrics in state-of-the-art image processing libraries (Bradski, 2000).

3.4.2 Hausdorff Distance for Similarity Computation

As described in the previous section, histograms can be suitable for abstracting case descriptions consisting of high-dimensional equally weighted features. Therefore, translating a case description based entirely on IMs like the one for high-level cases in Chapter 5 works well. A single IM or a set of IMs consist of a large number of low-dimensional data points with the same characteristics.

In Chapter 6, the case representation and the model it is part of lead to a different type of problem. Cases are represented by fewer yet more diverse features. The cases are based on collections of units which are in turn defined through sets of unit attributes. The main issue is not the dimensionality of the case feature vectors but the large number of cases in the case-base resulting from the complex problem that is addressed through the component. Therefore, histograms for abstraction are unsuitable. A method had to be found to compare large numbers of high-dimensional data points quickly.

Another method from the image processing domain turned out to be suitable, the *Hausdorff distance* (Huttenlocher et al., 1993). This metric is used to select similar cases from the large case-base. Applying this metric for fast similarity computation was inspired by an abstraction of unit descriptions as normalised n -dimensional feature vectors in n -dimensional space that

form polygons. Polygon comparison with control for simple geometric operations such as translation and rotation is a very common problem in image processing (Jain, 1989). Instead of individually maximising unit-to-unit similarity through a complex and computationally expensive procedure as described in Section 6.3 that requires an in-memory table, this allows a fast computation of case similarity in a single pass with low memory requirements.

The Hausdorff distance measures the degree to which each point in a set A lies close to *some* point in a set B . By comparing each point in both sets, this leads to a measurement of how far these two sets are removed from each other. In image processing, this measurement is used to compare objects from two images and thus measure the resemblance (Huttenlocher et al., 1993). In the problem addressed in this thesis, the Hausdorff distance serves to determine the resemblance between two groups of units where each unit is defined by a normalised feature vector.

Given two sets of units A and B , the Hausdorff distance in a metric space between them is defined as

$$h(A, B) = \max_{a \in A} \{ \min_{b \in B} \{ d(a, b) \} \}.$$

Figure 3.4 shows an example computation. As the figure displays, the distance between b_3 and a_1 is the general, non-directed Hausdorff distance between the two sets.

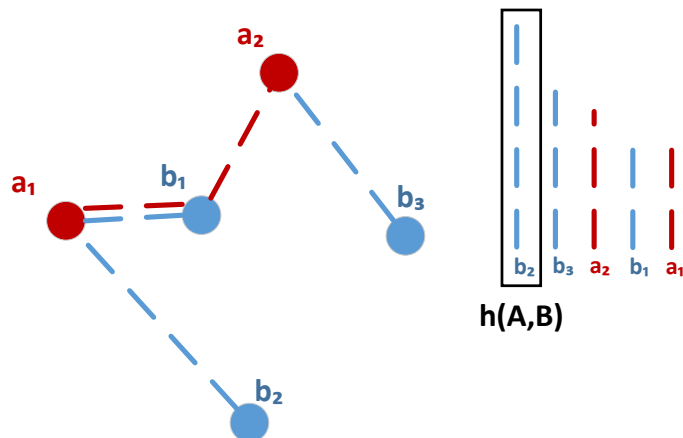


Figure 3.4: Computing the non-directed Hausdorff Distance between Sets A and B

As this example also shows, the two compared sets do not have to have the same number of elements. While the current implementation described in Chapters 6 - 9 requires an equal

number of units for the two cases that are compared, the theoretical ability of this metric to compare unevenly matched sets of units is interesting for potential future work.

3.5 Layered and Hierarchical Learning

Decomposing larger tasks into hierarchies of smaller, more manageable tasks can lead to promising results in addressing large-scale problems and is a popular approach to reduce complexity Ontañón et al. (2013). This section gives more detailed background on the layered learning paradigm and other hierarchical techniques.

Hierarchical approaches that use only RL have seen some applications, although limited through the inherent characteristics of RL. Publications such as Marthi et al. (2005) and Hanna et al. (2010) successfully use variations of modular and hierarchical RL in tasks with small-to-medium-sized state-action complexity. This illustrates that while subdividing a problem makes it more manageable and enables learning where it was not possible in a monolithic problem, RL-only approaches are still limited in terms of how complex a problem can be that they successfully address.

Hierarchical CBR is a sub-genre of CBR where there are several case-bases organised in an interlinked order. Smyth & Cunningham (1992) coin the term HCBR to describe a CBR system for software design. The authors use a blackboard-based architecture that controls reasoning with a hierarchical case-base. The case-base is structured as a partonomic hierarchy where one case is split into several layers. Finding a solution to a software design problem requires several iterations where later iterations are used to fill lower levels of the case. Eventually, one solution (i.e. one program) is found. The approach presented in this thesis is similar, though slightly more complex. The case-bases of the modules in Chapters 6 to 9 are not strictly partonomic, since lower-level case-bases do not only use solutions from higher level states as input for their case descriptions. Furthermore, the modules in this thesis are more autonomous in that they run on different schedules which can result in differing numbers of CBR-cycle executions for different case-bases.

While the approach presented in this thesis is related to HCBR and, to a lesser degree, HRL, the strongest conceptual influence comes from layered learning (LL). The LL paradigm was introduced by Stone (1998) as an architectural concept for an AI agent that manages virtual robots in a robotic soccer simulator. While being less complex, this domain is remarkably similar to RTS games in that is an inherently multi-agent adversarial environment.

There are four major principles in LL. These key principles according to (Stone, 1998) are

1. A mapping directly from inputs to outputs is not tractably learnable.
2. A bottom-up, hierarchical task decomposition is given.
3. Machine learning exploits data to train and/or adapt. Learning occurs separately at each level.
4. The output of learning in one layer feeds into the next layer.

All of these principles also apply to the hierarchical architecture for RTS game tactical and reactive tasks that is designed as part of this thesis. **Point 1** results from the high level of complexity that RTS games exhibit as a problem domain. **Point 2** is satisfied both on a high level through the many hierarchical RTS problem decompositions that are used in AI agents and in detail, through the customised task decomposition for the approach presented in Chapter 6. **Point 3** is decided through the choice of ML algorithms for the individual layers, in this case hybrid CBR/RL. The second part that requires levels to be learned individually is defined in the original publication by Stone (1998) but, as it basically limits the learning process, has since seen attempts at being modified (Whiteson & Stone, 2003; MacAlpine et al., 2015). **Point 4** is again decided through the design of the hierarchical architecture in Chapter 6.

These high-level principles match the problem addressed in this thesis very well. However, the detailed definition and resulting formalism presented in Stone & Veloso (2000) make the paradigm slightly less suitable. According to these definitions, layered learning is strictly a bottom-up approach where underlying layers feed feature vectors into higher ones. While this is partially the case in the approach presented in this thesis in that CBR/RL modules on higher levels re-use experience on lower ones, there is also an additional behaviour of higher layers feeding problem description parameters into lower-level case-bases. Furthermore, the formalism defining layered learning restricts the layers to working with concrete state-feature-vectors, both in terms of input and output and, more importantly, works on a distinct set of training samples. While case definitions in a CBR approach could be expressed as feature vectors, RL, unlike other ML approaches, does not work with training examples.

As these discrepancies show, the original formal definition of the LL does not account for all aspects of the approach presented in this thesis. As such, this approach cannot be formally expressed in the LL model. However, the work presented in this thesis can serve to highlight possible extensions to the LL paradigm and opportunities that arise from applying the paradigm the RTS game domain.

Reinforcement Learning for Strategy Game Unit Micromanagement

¹ This chapter evaluates the capabilities of different reinforcement learning algorithms (R. S. Sutton & Barto, 1998) for the purpose of learning micromanagement in a RTS game. The aim is to determine on the suitability of RL, as a ML technique that enables adaptive AI, for creating a learning agent in StarCraft. To do this, two prominent TD learning algorithms are used, both in their simple one-step versions and in their more complex versions that use eligibility traces. The background in Section 3.3 gave a more detailed account of these algorithms and RL in general.

Existing approaches such as the ones by Micić et al. (2011) and Shantia et al. (2011) define models that are specific to the tested scenarios. More importantly, there is no evaluation of different RL algorithms for those approaches, which is a central aspect described in this chapter. The task chosen for this evaluation is managing a combat unit in a small scale combat scenario in StarCraft. Subsequent chapters describe the development of a larger hybrid AI solution that addresses the entire StarCraft micromanagement problem, where the agent manages the different layers of problems in a complex commercial RTS game (Buro, 2003b). The overall goal is for the technique to be easily scalable for different types of units and eventually flexible enough to be transferred to different problem domains such as bigger scenarios, other games or even different sets of problems as described in (Laird & van Lent, 2001).

The chapter is structured as follows. First, the model that abstracts the in-game information and enables the agent to use RL for learning the micromanagement task is explained in detail. The overall algorithm that uses StarCraft as its simulation environment and RL as its learning technique is then explained. Subsequently, the setup of the empirical evaluation and the results from that evaluation are elaborated on. Finally, those results are discussed and their meaning for further developments is detailed.

¹ The contents of this chapter are based on a paper presented at and published in the conference proceedings for CIG 2012 (Wender & Watson, 2012).

4.1 Reinforcement Learning Model

The agent in a reinforcement learning framework makes its decisions based on the state s in which an environment is at any one time. If this state signal contains all the information of present and past sensations it is said to have the *Markov property*. If a RL task presents the Markov property, it is said to be a *Markov decision process (MDP)*. A specific MDP is defined by a quadruple $(S, A, \mathcal{P}_{ss'}^a, \mathcal{R}_{ss'}^a)$. RL problems are commonly modelled as MDPs, where S is the state space, A is the action space, $\mathcal{P}_{ss'}^a$ are the transition probabilities and $\mathcal{R}_{ss'}^a$ represents the expected reward, given a current state s , an action a and the next state s' . The MDP is used to maximise a cumulative reward by deriving an optimal policy π^* according to the given environment.

In order to adapt the RL technique to the given task of micromanaging combat units, a suitable representation has to be chosen. This is especially important since StarCraft is a very complex environment that requires extensive abstraction of the available information in order to enable meaningful learning and prevent an exponential growth of the state- or action space. This would make any meaningful learning within a reasonable time impossible.

4.1.1 Reinforcement Learning States

States are specific to a given unit at a given time and are designed as a quadruple of values consisting of the following.

- *Weapon Cooldown*: Is the weapon of this unit currently in cooldown? (i.e. 2 Possible Values)
- *Distance to Closest Enemy*: Distance to the closest enemy as a percentage of the possible movement distance of this unit within the near future. Distances are grouped into four different groups ranging between $\leq 25\%$ and $> 120\%$ (i.e. 4 Possible Values)
- *Number of Enemy Units in Range*: Number of enemy units within range of this unit's weapon.
- *Health*: Remaining health of this unit, classified into one four different classes of 25% each (i.e. 0%-25% etc.).(i.e. 4 Possible Values)

This leads to a possible size of the state space

$$|S| = 32 * (|U_{enemy}| + 1) \quad (4.1)$$

where U_{enemy} is the set of all enemy units.

4.1.2 Reinforcement Learning Actions

There are two possible actions for the units in this model: *Fight* or *Retreat*. This set of actions was chosen for several reasons. While there are many more possible actions for an agent playing a complete game of StarCraft, the focus for this task, and therefore for this model, lay on unit control in small scale combat. This excludes many high-level planning actions, such as actions given to groups of units. Many units in StarCraft have abilities reaching beyond standard fighting, so-called *special abilities* that can be decisive in battles but which are ignored in this setup. However, the core of any army and therefore the core of any fight will always consist of ranged or melee units that have no special abilities beyond delivering normal damage.

Using only *Fight* and *Retreat* as actions also omits more general actions such as exploring the surroundings or simply remaining idle. Exploration is a crucial part of any StarCraft game but involves many different aspects not directly related to combat and will therefore eventually be handled by a completely separate component of the AI agent. The *Idle* action was originally part of the model but experimental empirical evaluation showed that this only meant that the built-in unit micromanagement took over, which is actually a mix of both existing actions, *Fight* and *Retreat*. Therefore, the learning process was severely impeded by the *Idle* action receiving a reward signal that was basically a mix of the reward signal of the other two actions.

Fight Action The *Fight* action is handled by a very simple combat manager. The combat manager determines all enemy units within the agent's unit's range and selects the opponent with the lowest health which can thus be eliminated the fastest. Should no enemy unit be within weapon range when the action is triggered, nothing will happen until the next action has to be chosen.

Retreat Action The *Retreat* action aims to get the agent's unit away from the source of danger, i.e. enemy units, as quickly as possible. A weighted vector is computed that leads away from all enemy units within the range they would be able to travel within the next RL time frame. The influence of one enemy unit on the direction of the vector is determined by the amount of damage it could do to the agent's unit.

To avoid being trapped in a corner, on the edge of a map or against an insurmountable obstacle on the map (StarCraft maps can have up to three different levels of height, separated by cliffs) a repulsive force is assigned to these areas. This force is included in the computation of the vector. If an agent's unit that tries to retreat is surrounded by equally threatening and thus repulsive forces, no or next to no movement will take place.

4.1.3 Transition Probabilities

The game logic in StarCraft is almost entirely deterministic. However, because of the high level of abstraction in the definition of the states S and the actions A , the transition rules are stochastic. This means that, depending on the behaviour of the units involved, different subsequent states s_{t+1} can be reached after taking the same action a_t in the same state s_t at different times in the game.

4.1.4 Reinforcement Learning Reward Signal

The reward signal is a crucial element of the MDP since the RL agent bases its action selection policies on the reward it received for choosing previous actions. This means that the reward has to reflect all potential goals of the model in order to enable the agent to learn how to achieve these goals. In terms of small-scale combat, the two goals are the elimination of opponents and the preservation of one's own unit's health. Therefore, the reward signal is based on the difference in health of both the enemy's and the player's units between two states. The reward is computed as the difference in health of the enemy's units (i.e. the damage done by the agent) minus the difference in the RL agent's unit (i.e. the damage received by the agent).

$$\begin{aligned} \text{reward}_{t+1} = \sum_{i=1}^m \text{enemy_unit_health}_{i_t} - \text{enemy_unit_health}_{i_{t+1}} \\ - (\text{agent_unit_health}_t - \text{agent_unit_health}_{t+1}) \quad (4.2) \end{aligned}$$

As a result, the agent will measure its success in the amount of damage it is able to deliver while trying to retain as much of its own health as possible.

4.2 Algorithm

StarCraft, like most other games in the RTS game category, emulates 'real-time' gameplay. This means that while the game is still computed internally on a turn-by turn basis, these turns (or *frames*) happen so quickly that human players perceive the game as being continuous. On the one hand, this limits the computing power that can be used by any algorithms since the player still has to perceive the environment as real-time. This is especially important for an online learning technique such as RL, which runs updates every time frame. On the other hand, RL techniques are usually designed to work with concrete time steps. One possible translation of the RL model into StarCraft would have been to use a fixed number of StarCraft frames as one RL time step. However, this proved problematic as the *Fight* action

Chapter 4. Reinforcement Learning for Strategy Game Unit Micromanagement

varies in duration, depending on the unit that executes it and also depending on the physical positions of units on the map. Therefore, it was decided to define one RL time step simply as the time it takes for that action to be finished for the *Attack* action. For the *Retreat* action this meant that units would retreat in the chosen direction at maximum speed for a fixed time. A high-level view of the steps involved in integrating the agent into StarCraft, both in initial and subsequent runs, can be seen in Figures 4.1 and 4.2.

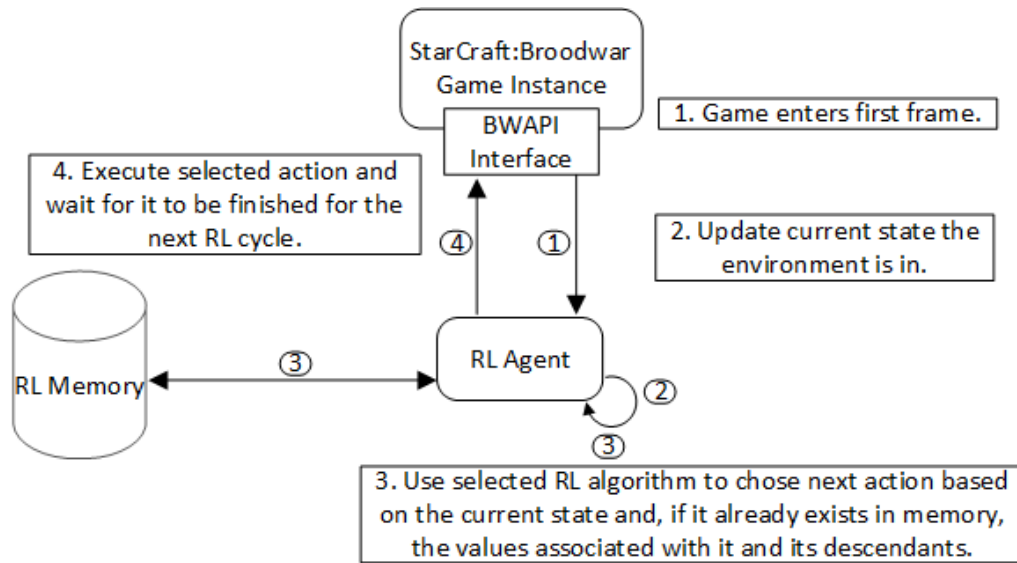


Figure 4.1: StarCraft RL Algorithm Integration Initial Run

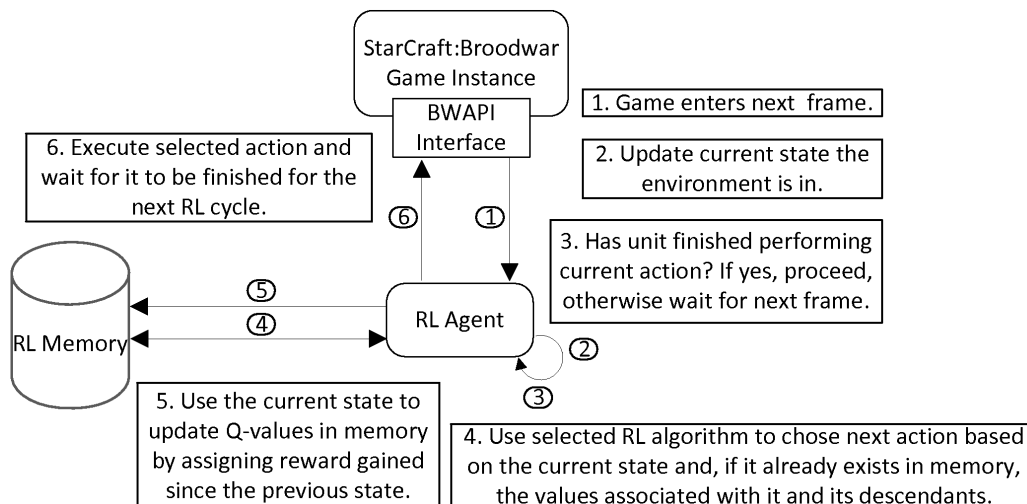


Figure 4.2: StarCraft RL Algorithm Integration General Run

4.3 Empirical Evaluation and Results

For the evaluation of the performance of the selected RL algorithms, a small scale combat scenario was designed in StarCraft to allow the RL agents to show their ability to learn in an unsupervised environment. The scenario consists of one combat unit controlled by the RL agent fighting a group of enemy units spread out around the starting location of the RL agent as can be seen in Figure 4.3. The RL agent unit has the advantage of superior speed, superior range and - by a small margin - superior firepower in comparison to a single enemy unit. However, when fighting more than one enemy unit, the agent's single unit easily loses. Preliminary tests showed, that using only the built-in game AI, the single unit quickly loses in this scenario every time. The aim for the RL agent is to learn to exploit its advantages in terms of speed and range by adopting so-called 'kiting', a form of hit-and-run: getting and staying out of range of enemy units while firing at them from a safe distance. An episode in the experiment concludes when either the agent's unit or all the enemy's units have been eliminated.



Figure 4.3: Initial unit positioning for the experimental evaluation

4.3.1 Experimental Setup

Each RL algorithm was run for 1,000 episodes during which the memory of the agent was not reset, i.e. after 1,000 games the memory was wiped and the agent started learning from scratch. These 1,000 episodes were repeated 100 times for each algorithm in order to gain conclusive insights into their performance. For all algorithms, the agent followed an ϵ -greedy policy. However, the value of ϵ was set to decline from its initial value of 0.9 to reach zero at

Parameter	Values
Algorithm	One-Step Q-learning, Watkins's $Q(\lambda)$, One-Step Sarsa, Sarsa(λ)
Number of Games	1000, 500
Learning Rate α	0.05
Discount Factor γ	0.9
Eligibility Trace Decay Rate λ	0.9
Exploration Rate ϵ	0.9 - 0

Table 4.1: Reinforcement Learning Evaluation Parameters

the end and thus make the action selection process more and more greedy towards the end. This enables the RL agent to always follow the best known policy Q^* at the end.

After the 1,000-episode runs, the process was repeated with shorter 500-episode runs, again with diminishing ϵ -greedy policy. This was done to gain a better understanding of algorithm performance in shorter terms and their speed of convergence towards an optimal policy. Altogether, each RL algorithm played 100,000 games for the 1,000-episode runs and 50,000 games for the 500-episode runs. The values chosen for the experimental setup are $\alpha = 0.05$, $\gamma = 0.9$ and, for those algorithms using eligibility traces, $\lambda = 0.9$. These values were chosen based on commonly-used parameter settings (R. S. Sutton & Barto, 1998) and preliminary experimental runs. The settings result in a slow learning process which hugely discounts possible future rewards. For algorithms using eligibility traces, the slow decay of these traces results in future rewards playing a big role for previously visited states. Table 4.1 summarises the algorithmic parameter settings.

4.3.2 Results and Discussion

Figures 4.4 to 4.9 show the results of the experimental evaluation. The diagrams show averaged values where each data point represents the average of ten values from the 500-game or 1,000-game runs, leading to 50 or 100 data points respectively.

The development of the overall reward gained by the RL agent during a single game was measured (Figures 4.4 and 4.6). The overall reward is the sum of all rewards the RL agent achieves in a single game. An optimal performance, i.e. defeating all enemy units without sustaining any damage to the agent's own unit, would result in a total reward of 300. Since the final aim for the agent is to win an entire game, the percentage of games the agent is able to win was also recorded, i.e. the amount of games where the agent eliminates all opponents without losing its unit (Figures 4.5 and 4.7). This value is closely tied to the overall reward but not entirely similar, as shown in the differences between Figures 4.6 and 4.7. Furthermore, the standard deviation as a measurement of the variance in results was computed (Figures 4.8 and 4.9).

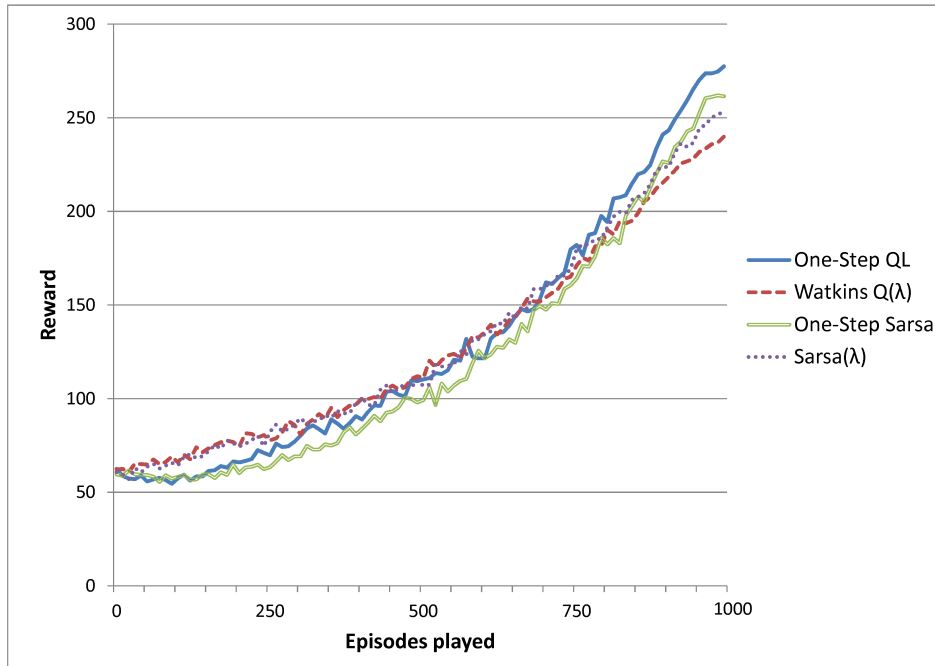


Figure 4.4: Development of Total Reward for 1,000 Episodes Played

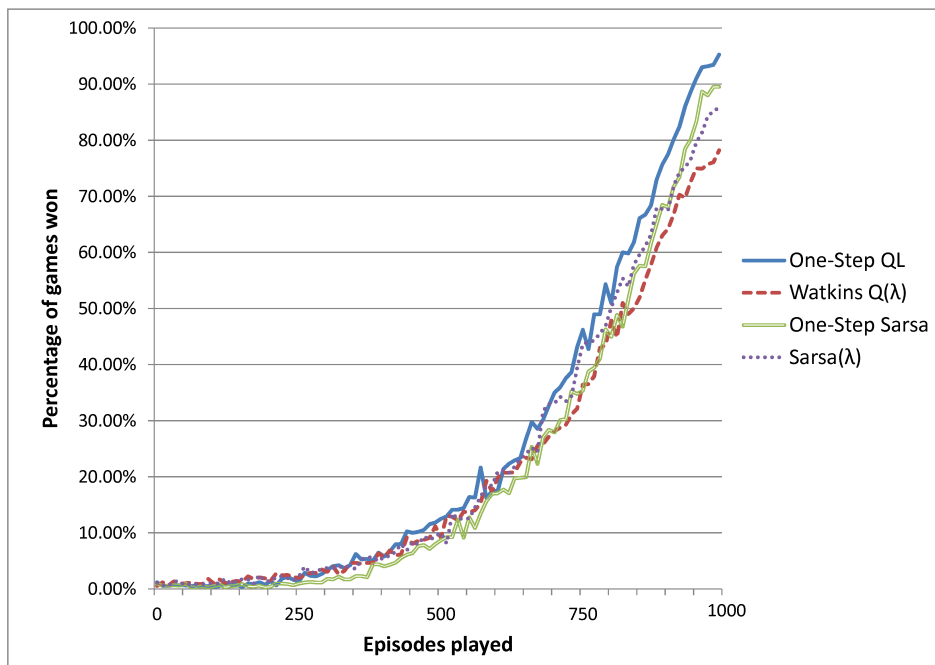


Figure 4.5: Percentage of Games Won for 1,000 Episodes Played

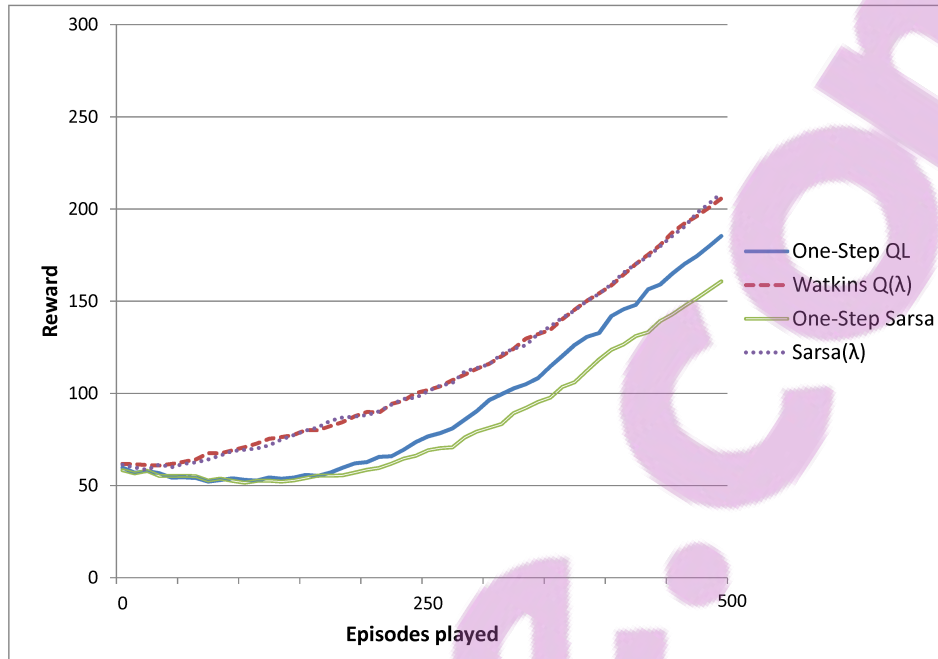


Figure 4.6: Development of Total Reward for 500 Episodes Played

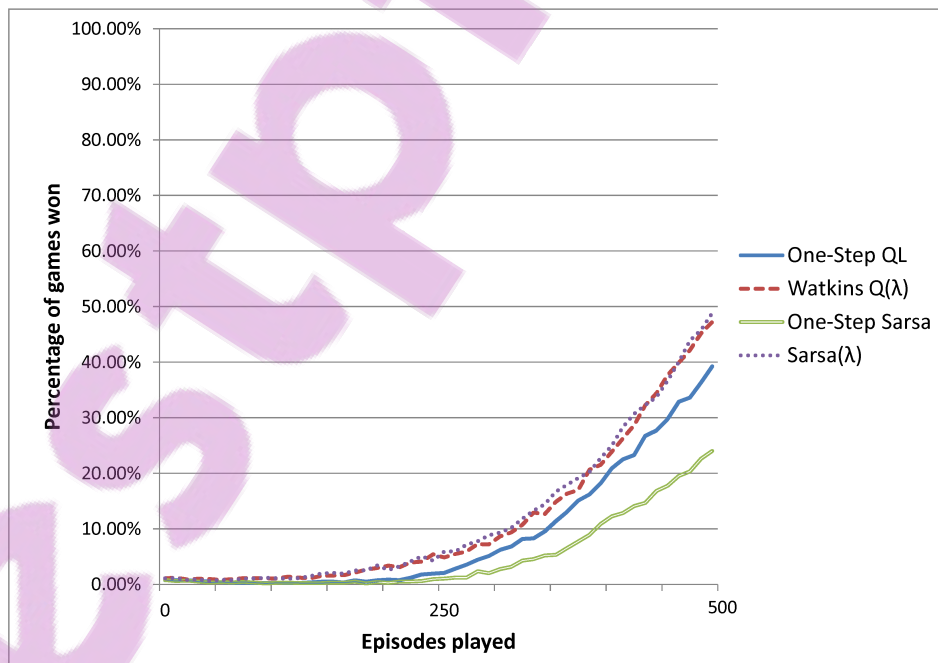


Figure 4.7: Percentage of Games Won for 500 Episodes Played

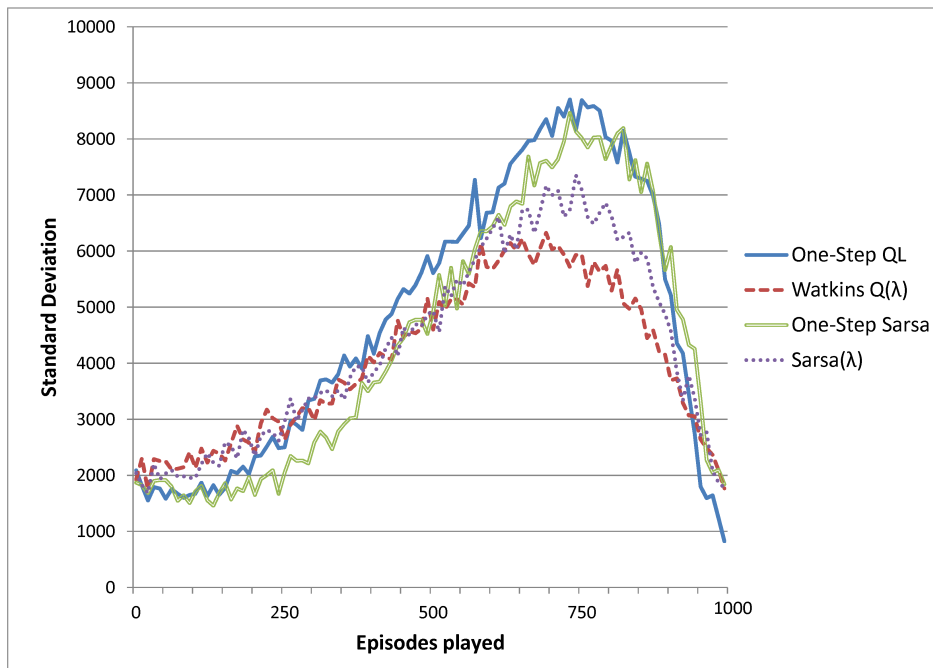


Figure 4.8: Standard Deviation for 1,000 Episodes Played

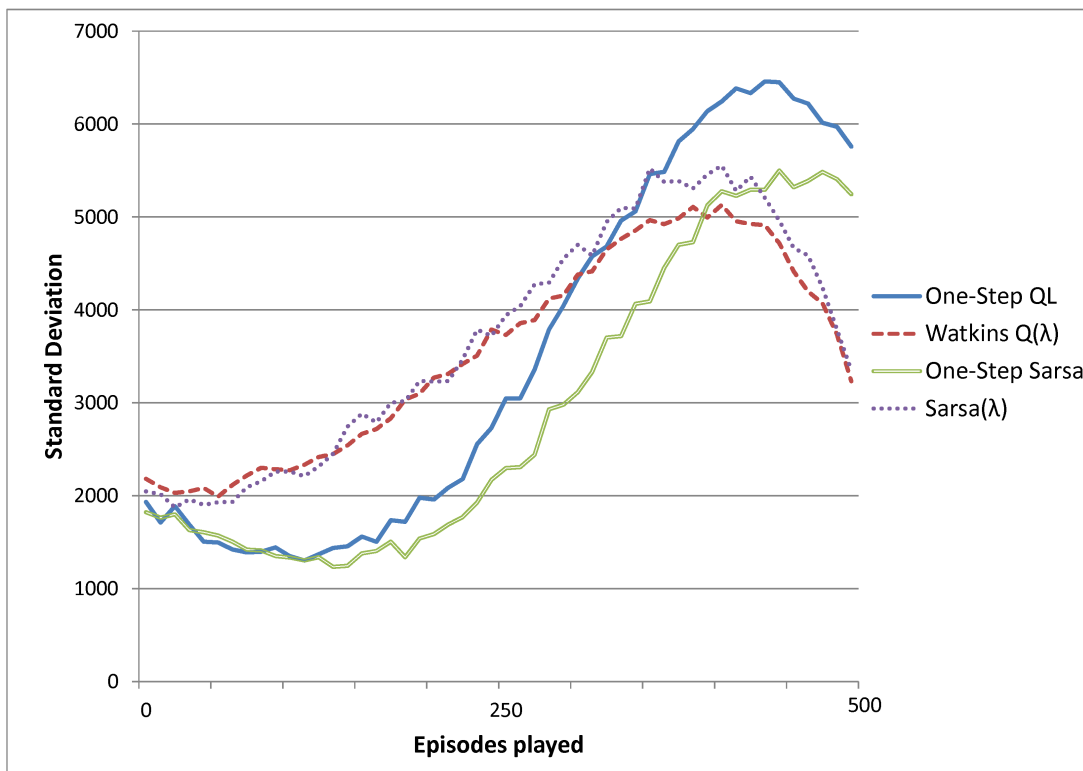


Figure 4.9: Standard Deviation for 500 Episodes Played

Chapter 4. Reinforcement Learning for Strategy Game Unit Micromanagement

After 1,000 episodes of learning, all four tested algorithms manage to achieve easy wins when following their most greedy learned policy. The total reward is close to the maximum value of 300 for all algorithms towards the end. This means that all algorithms manage to learn a policy close to the optimal policy π^* in 1,000 episodes. Furthermore, even the speed of learning seems to differ little between the different algorithms. The reward for the one-step versions of the algorithms slightly declines at first, however starts to grow equally quickly to that of Watkins's $Q(\lambda)$ and $Sarsa(\lambda)$ after about one-third of the experimental run. Eventually, both one-step Q-learning and one-step Sarsa manage to gain more reward than their counterparts which are based on eligibility traces. Using the best possible learned policy at the very end, one-step Q-learning achieves an average reward of about 275, slightly outperforming one-step Sarsa (about 262), $Sarsa(\lambda)$ (about 252) and, a little behind, Watkins's $Q(\lambda)$ (about 237). The results also show that this is the biggest difference between the best and the worst performing algorithm at any point throughout the whole 1,000 episodes.

The behaviour of the standard deviations for 1,000 episodes, depicted in Figure 4.8, differs more between algorithms than the results for total reward and games won. Given the behaviour of RL, the expected development of the standard deviation is that there is a large standard deviation at the start, when a good policy is still unknown. This initial deviation then diminishes towards the end when the agent uses more and more greedy selections which finally lead to only following their approximation of the optimal policy π^* .

The results show the same general behaviour for all algorithms. The initial standard deviation rises to a peak in the later stages of the 1,000-episode run and finally sharply drops below even the initial value when approaching a purely greedy policy. The reason for this behaviour is the greater variance of possible results once the agent has acquired a certain amount of knowledge. It therefore sometimes selects very good policies while on the other hand still following a partially exploratory policy that can lead to low rewards. The initial standard deviation results from the near-random policy that the agent follows at the start of an experimental run. Towards the end, the agent quickly approaches the point where it always follows the same policy. The remaining variance is a result of the non-deterministic state transitions resulting in turn from the complexity of the game.

The standard deviation for the one-step versions of the algorithms marginally declines at the start of the evaluation run. After about one-third of the run, the standard deviation for one-step Q-learning and one-step Sarsa starts to grow rapidly, quickly surpassing that of Watkins's $Q(\lambda)$ and $Sarsa(\lambda)$, which have been growing from the start but less rapidly. This behaviour bears a similarity to that of the reward signal, thus indicating the connection between acquired knowledge and a high standard deviation mentioned in the previous paragraph.

Among the algorithms, one-step Q-learning shows the smallest standard deviation towards the end while having the largest deviation of all algorithms - although not by far - at its peak.

Chapter 4. Reinforcement Learning for Strategy Game Unit Micromanagement

This means that one-step Q-learning gets closest to repeatedly following an optimal policy at the end, something which is in line with what Figure 4.4 shows. The standard deviation for one-step Sarsa shows similar behaviour to that of one-step Q-learning with a slightly less-pronounced peak. The lowest maximum for the standard deviation is shown by Watkins's $Q(\lambda)$; otherwise it behaves similarly to the other algorithms. The randomness in exploration is the same for all algorithms, therefore the difference must lie in the best learned policy. As the development of its total reward already shows that Watkins's $Q(\lambda)$ performs slightly worse than the other algorithms, the behaviour of the standard deviation now gives a hint where the problem lies. The behaviour of the standard deviation is nearly identical for all algorithms in the first half of the experimental run and the standard deviation of Watkins's $Q(\lambda)$ only starts to grow slower in comparison to the other algorithms after about 500 runs. Therefore, this must be the point where Watkins's $Q(\lambda)$ starts to learn less than the other algorithms. Further experiments will have to show what exactly makes the difference here.

The performance in terms of reward gained and winning games over 1,000 episodes is very similar among all algorithms, only showing minimal differences in their effectiveness. Therefore, the results of the 500-episode runs are important to compare the learning capabilities of the different algorithms in terms of speed.

Most notable is that, while a learning process definitely took place, none of the algorithms managed to achieve the optimal results as seen in the 1,000-episode runs. Both of the algorithms that use eligibility traces, Watkins's $Q(\lambda)$ and Sarsa(λ), show a similar, comparably strong, performance. At the end of the 500-episode runs, following their best learned policy, they achieve an average reward of about 200, or about two-thirds of the maximum possible reward. Compared to this, one-step Q-learning achieves about 10% less reward overall and one-step Sarsa about 25% less than both best performing algorithms. Furthermore, there is a distinct difference in the learning curves between the algorithms that use eligibility traces and their one-step pendants. Both one-step Q-learning and one-step Sarsa show a slightly declining curve at first, very similar to that exhibited by all algorithms in the longer 1,000-episode runs. Subsequently, that curve rises again, until it is nearly linear with one-step Q-learning, changing more quickly than one-step Sarsa from decline to growth. This growth eventually results in the difference in overall reward between one-step Q-learning and one-step Sarsa at the end.

The figures for the algorithms that use eligibility traces indicate a faster learning process compared to the one-step algorithms. While it takes one-step Q-learning and one-step Sarsa more than 100 episodes to start increasing the obtained reward, Watkins's $Q(\lambda)$ and Sarsa(λ) nearly instantly (there is a minimal decrease in reward at first) start increasing their total reward. This is the same behaviour that is apparent in the longer runs of 1,000 episodes, only it is more distinctive due to larger differences in values and lower overall rewards. The

instant growth of reward for Watkins’s $Q(\lambda)$ and Sarsa(λ) basically shows the effectiveness of eligibility traces in terms of speeding up the learning process.

Figure 4.9 shows how the behaviour of the standard deviation is quite different among the algorithms. The standard deviation for all four algorithms is initially identical to that in Figure 4.8. The standard deviation for one-step Q-learning and one-step Sarsa shows an initial decline before growing rapidly and eventually surpassing that of Watkins’s $Q(\lambda)$ and Sarsa(λ). However, as compared to the 1,000-episode runs, it is different towards the end when the policy gets more and more greedy for all algorithms. Watkins’s $Q(\lambda)$ and Sarsa(λ) show a drop similar - although less distinct - to that of 1,000-episode runs. The one-step versions of the algorithms however, show much less of a drop and start to decline much later. The standard deviation for Sarsa(λ) even seems to stay linear before declining a little in the last few episodes. These results indicate the lack of a sufficiently advanced policy because the agent so far has been unable to learn which are the best states/actions and therefore is stuck with a sub-par policy. The development of the overall reward for these two algorithms confirms this conclusion.

4.4 Conclusions on the Future Use of RL for Micromanagement in RTS Games

The empirical evaluation shows that all RL algorithms are suitable to learn the selected task. One-step Q-learning demonstrates the strongest performance in terms of accuracy, given enough episodes to learn the required task. One-step Sarsa is a close second. Watkins’s $Q(\lambda)$ and Sarsa(λ) also display good results in terms of accuracy, although their strength seems to lie in learning fast and, following from this, speed of convergence towards the optimal policy π^* . While eligibility traces help to speed up the convergence towards an optimal policy as expected, this happened at a trade-off of diminishing overall reward.

The values chosen for α , γ and λ in the experimental setup are based on common values for RL experiments and some early experimental runs (R. S. Sutton & Barto, 1998) but might not be optimal for the specific problem. A more sophisticated parameter selection through ML can help to optimise these settings for use in future problems and algorithms.

Furthermore, the approach presented in this chapter is intended to be the first step towards of a larger RL-based agent which is able to address the entire micromanagement component of StarCraft. Due to the problem’s high complexity, this is not possible with a RL-only agent but has to be handled by a hybrid approach that also further abstracts the problem into different layers of logic. This is a similar approach to handling the overall game through different agents that address specific sub-problems (Safadi & Ernst, 2010; Weber, Mawhorter, et al., 2010). Simple RL will not be enough to handle all the different tasks in StarCraft

Chapter 4. Reinforcement Learning for Strategy Game Unit Micromanagement

micromanagement. Therefore, the addition of other ML techniques such as CBR to create hybrid approaches (Weber & Ontanón, 2010) for the different problems inherent in StarCraft micromanagement is a logical next step.

Overall, the empirical evaluation over 1,000 episodes showed that an AI agent using RL is able to learn a strategy that beats the built-in game AI in the chosen small-scale combat scenario almost 100% of the time. This impressive win rate is achieved by all evaluated RL algorithms and hints at the extensive possibilities of using RL for micromanagement in StarCraft. Since the aim is to evaluate the usability of RL in StarCraft as a representative commercial RTS, the efficiency of the algorithms and thus the time it takes the agent to develop a usable strategy also has to be taken into account. This is very important as, should these types of ML algorithms ever be considered for application in commercial games, the learning process has to be both quick and be able to avoid major drops in performance.

The experiment over half the number of episodes showed that a quicker learning curve is indeed possible but currently comes at the price of a decrease in obtained reward. In the case of the best-performing one-step Q-learning this shortening of the learning phase even leads to a performance drop of nearly 40%. Further research is clearly needed before it will be feasible to create a powerful standard game AI in a commercial game such as StarCraft using RL. However, the results are very promising in terms of overall performance for solving a simplified version of the micromanagement problem. Furthermore, those results also indicate many possible ways of easily reusing the current findings to advance the development of a RL agent which is able to learn how to solve more complex micromanagement problems in StarCraft.

Based on these findings, Chapter 5 introduces the next step: an agent that addresses a more complex version of the micromanagement problem by using a hybrid ML approach to control several units.

Combining Reinforcement Learning and Case-Based Reasoning for Strategy Game Unit Micromanagement

¹ This chapter describes the concept of a hybrid RL and CBR approach to managing a group of combat units in StarCraft. Both methods are combined into an AI agent that is evaluated by using a more complex scenario than the one for the RL-only agent in Chapter 4. While StarCraft offers a very large and complex set of problems, that RL-only agent focuses on a small subset of the overall problem space and is limited to one simple scenario. In this limited area of application, the simple RL approach is very successful in learning how to beat the built-in game AI. The hybrid CBR/RL agent presented in this chapter uses a more generalised model which can handle a broader set of problems. Combining CBR with RL helps to offset the shortcomings of the simple RL agent, while retaining key features of its performance. The hybrid agent uses CBR for its memory management, RL for fitness adaptation and an influence map(IM)/potential field (Khatib, 1986) for the abstraction of spatial information. This is not the final increment of this approach as the eventual aim of this approach is an AI agent that has actions and information at its disposal that are close to what human players use when performing micromanagement actions.

This chapter also contains an optimisation of the algorithmic parameters for both RL and CBR components, using a combination of experimental evaluation and ML to find the best possible settings. Subsequently, as part of an experimental evaluation, the agent is tested in different scenarios using these optimised algorithm parameters. The integration of CBR for memory management is shown to improve the speed of convergence to an optimal policy, while also enabling the agent to address a larger variety of problems when compared to simple RL. The agent manages to beat the built-in game AI and also outperforms a simple RL-only agent. An analysis of the evolution of the case-base shows how scenarios and algorithmic

¹ The contents of this chapter are based on a paper presented at and published in the conference proceedings for PRICAI 2014 (Wender & Watson, 2014b).

parameters influence agent performance and will serve as a foundation for future improvement to the hybrid CBR/RL approach and its use for the eventual hybrid hierarchical algorithm.

This chapter is structured as follows. First, the CBR/RL architecture is explained in detail. This includes an explanation of the integration of the CBR cycle into the StarCraft RTS game as well as an explanation of the RL model that abstracts the game and enables the agent to use RL for learning the micromanagement task. The setup of the empirical evaluation is then explained. This is followed by a section explaining how the algorithmic parameters for both RL and CBR are optimised. Then, the actual evaluation using optimised parameters and the results of that evaluation are described. Finally, those results are discussed, both in terms of learning performance and in terms of case-base behaviour and their meaning for further developments.

5.1 CBR/RL Agent Architecture

The agent uses a CBR-based memory which utilises RL to learn the fitness of its case solutions. The model of the game world is based on two different case-bases for different levels of abstraction of the current game state. RL is used to update the value of unit actions. Those unit actions represent the case solutions.

5.1.1 Case-Based Reasoning Component

The CBR component reflects a general CBR cycle (Aamodt & Plaza, 1994) but does not use all of the cycle's phases. Two separate case-bases are used to reflect two different levels of granularity within the game. The first case-base contains information on the overall game state at a certain point. This includes information on the number and types of unit in the game as well as information on the state of the surroundings in the form of an IM. The second case-base stores and administers per-unit information. This type of abstraction is a first attempt at reflecting the multi-tiered nature of RTS games (see Section 3.1.1) through an appropriate representation.

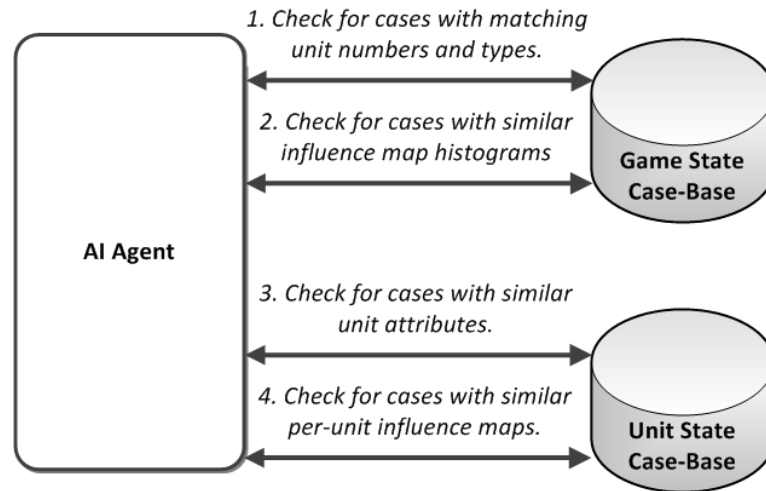


Figure 5.1: Case Retrieval Process

Retrieval is thus a multi-tiered process, shown in Figure 5.1. At first, cases matching the overall current game in terms of agent and opponent units exactly and the overall game environment state to a certain degree of similarity are retrieved. The game environment state is encoded as an IM that reflects the areas of influence of the AI agent units and of enemy units. Figure 5.2 shows an excerpt of the field representing enemy unit influence.



Figure 5.2: Excerpt of the Influence Map

This influence is based on the damage potential of the units. Since an IM can theoretically be as big as a whole game map (up to $256 \times 256 = 65536$ fields), a suitable abstraction has to be found to compare similarities. It was decided to use a histogram-based comparison (Davoust et al., 2008). This is possible if an IM is regarded as a picture with multiple colour channels for multiple IMs and colour values representing the influence values. After converting IMs into a histogram, correlation can be used to determine similarity. A nearest neighbour (NN) retrieval of cases in the case-base is done based on this similarity. Histogram-based similarity computation is described in more detail in Section 3.4.1.

Chapter 5. Combining Reinforcement Learning and Case-Based Reasoning for Strategy Game Unit Micromanagement

Cases from the game state case-base are linked to cases from the unit state case-base, i.e. each unit case is secondary to a game case (Figure 5.3). This results in a tree-like structure where each game state case branches out into a subset of unit state cases. As there is currently no re-use between unit state cases that are under different game state cases, this architecture requires careful specification of the similarity metrics that determine whether a new overall game state case is created. Each new game state case results in a whole new problem space which has to be explored by the agent. Therefore, a new game case should only be created when a sufficiently novel game state is encountered.

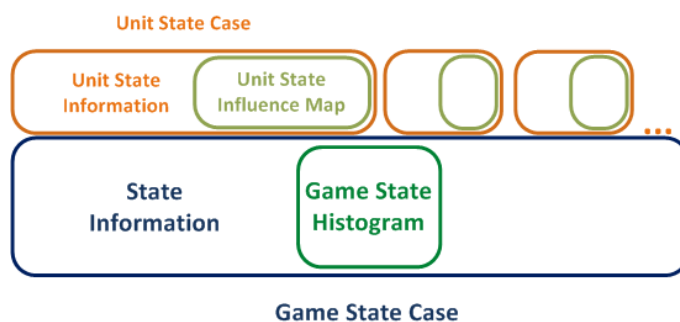


Figure 5.3: Logical Structure of Cases and the Information they contain

Unit cases contain general information on units as well as a local IM, i.e. an influence map for the immediate surroundings of the unit. As this is the only precise spatial information the unit has, these unit-specific IMs cannot be converted into histograms since that would abstract too much information they hold on positioning. Both general information and IM are used to compose a similarity score for comparison with the problem unit case.

The final step in the case retrieval process returns actions with assigned fitness values. These actions can then be **Reused** to solve the current problem case.

Currently, there is no **Revision** of case descriptions stored in the case-bases. Once a case has been stored, the case descriptions consisting of the general attributes and the IM that describe a game state or unit state are not changed further. The case *solution* that consists of actions and associated fitness values however, are adjusted each time they are selected and executed. These updates are done using RL (see Section 5.1.2).

Retention is part of the retrieval process. In the first step of that process (see Figure 5.1), if there is no sufficiently similar overall case, a new case is generated from the current environment state. Likewise, when searching the database for unit cases does not lead to any sufficiently similar results, a new unit case is created from the current problem.

5.1.2 Reinforcement Learning Component

The RL component is used to learn fitness values for case solutions. This part of the agent is based on the approach presented in Chapter 4 where Q-learning was identified as the most suitable TD RL algorithm. While the complexity of the model for the problem addressed by hybrid CBR/RL is higher, the problem domain remains the same. Because of this similarity, it was decided to also use a Q-learning algorithm for policy evaluation. One-step Q-learning and Watkins' $Q(\lambda)$, which uses eligibility traces, showed by a small margin, in this particular setting, the best performance when compared to other tested algorithms.

Q-learning as an off-policy temporal-difference algorithm does not assign a value to states, but to state-action pairs (Watkins, 1989) (See Section 3.3). Since Q-learning works independently of the policy being followed, the learned action value Q function directly approximates the optimal action-value function Q^* . The value update function for One-Step Q-learning is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \quad (5.1)$$

The Q value in this context is used as a fitness value. Function 5.1 is used to update the value of taking an action a_t in state s_t resulting in state s_{t+1} , given the learning rate α and a discount factor for future reward γ .

5.2 Model

States: States are represented by the cases of the CBR component minus the case solutions. Case architecture was explained in detail in Section 5.1.1. A state is mostly specific to a given unit at a certain time, but also contains some information on the general game state. The unit-specific part of a state contains general information such as *Unit Health*, *Unit Type* and *Weapon Cooldown*. Both the general part and the unit-specific part contain an IM for spatial information. In the general game state part, this IM is abstracted into histograms. The local, unit-specific IM is made up of 7×7 tiles of the general IM and not abstracted.

Actions: There are nine possible actions for the units in this model, one *Attack* action and eight *Move* actions. The model of the environment used in Section 4 is geared towards RL in a simple scenario. Due to this specialisation, it has a number of shortcomings that make it badly suited for the current approach. In the simple model, there are only two possible actions for units: *Attack* and *Retreat*. Being this limited in terms of actions means that the translation of RL actions into in-game actions has to contain a lot of expert knowledge in order to allow complex manoeuvres inside the game.

Ideally, the CBR/RL agent would have a continuous state- and action-space such as that used in Molineaux et al. (2008). However, the aim for the agent presented in this section

is not only to focus on the movement of units but also to allow them to function inside the StarCraft game environment, which has a considerably higher complexity and level of detail than the environment used by Molineaux et al. (2008). Therefore, unit movement is abstracted to eight discrete directions, one for each 45° .

The *Attack* action is handled by a very simple combat manager. This combat manager determines all enemy units within the agent's unit's range and selects the opponent with the lowest health and which can thus be eliminated the fastest. Should no enemy unit be within weapon range while the action is triggered, no attack will happen. This simple logic is only a very minor but effective improvement over a random attack. If, on the other hand, the selection of combat targets were to be optimized using ML methods, this would be a complex problem in its own right, as presented by Gunnerud (2009). The final hybrid approach designed in Chapter 6 addresses this problem in Section 8.2.

It was also considered to include a *Regroup* action based on a flocking algorithm, to facilitate several units to create formations. After several runs of evaluation studies, it was decided to remove this action, essentially for the same reasons for which the *Retreat* action from Chapter 4 was discarded: *Regroup* based on flocking contained too much expert knowledge that the agent should rather be able to learn using RL/CBR, given a proper representation of the game environment and an appropriate evaluation algorithm. Because of their complex internal logic, such specialised actions can also introduce a level of randomness that complicates replicable learning.

During action execution, there are no fixed time intervals pre-selected for action durations. Each unit starts and finishes actions in a potentially different game frame. A unit transitions to a new state once its current action is finished.

Transition Probabilities: While StarCraft has only few non-deterministic components, its overall complexity and the abstraction through the model mean that the transition rules between states are stochastic. As a result, different subsequent states s_{t+1} can be reached when taking the same action a_t in the same state s_t at different times in the game.

Reward Signal: Rewards, similar to cases, are computed on a per-unit basis. Rewards reflect the fitness value of an action that a certain unit took in a certain state. The reward signal is based on the difference in health of the agent unit in question, as well as the difference in health for enemy units it attacks, i.e. the damage that the agent unit is able to deliver.

While this is a rather simplistic view - the behaviour of other units is not taken into account at all - there are several reasons for this. As long as a unit chooses only its own actions, taking into account the effects of any action that another unit chose, would only dilute the reward signal and mislead in the terms of fitness. Actions could end up being regarded as much better than they actually are, simply because other allied units chose good actions. In the

long run, these effects can potentially even out so that only actually good actions receive high scores. However, those effects would prolong the learning time.

Equation 5.2 shows the resulting formula for the reward signal.

$$\text{reward}_{t+1} = \text{damage_done}_t - \text{damage_received}_t \quad (5.2)$$

In other words, this definition of the reward signal means that the agent measures its success in the amount of damage it is able to deliver while trying to retain as much of its own health as possible. This is similar to the reward signal of the RL-only agent as the task has remained the same, except the reward obtained for damage dealt to opponents is now assigned to specific units only.

5.3 Empirical Evaluation and Results

There are three parts to the empirical evaluation. The first part, supplementary to the overall approach, is an optimisation of algorithmic parameters for the CBR and RL components. The second and most important part is evaluating the performance of the approach in different scenarios, compared both against the built-in game AI and the previous simple RL algorithm. In the third part, the behaviour of the case-bases that serve as the agent's memory during the performance evaluation is evaluated in detail. The motivation behind this is to identify trends pointing to bottle-necks in the approach and to find interesting behaviour and points which require improvement when the agent's abilities are extended in the future. This case-base evaluation is done by recording and analysing additional metadata for CBR. This metadata includes the time that experimental runs take, as well as several metrics relating to action selection and case-base size. In order for RL to approximate an optimal policy π^* , there has to be a sufficient state- or state-action space coverage (R. S. Sutton & Barto, 1998). The amount of coverage that is sufficient in this context can be determined by evaluating at what point the agent is able to reliably achieve optimal results in terms of reward.

Both the integration of CBR with the RL component and the model created to facilitate this integration intend to allow the CBR/RL agent to perform in a similar fashion as the RL-only agent in a more complex problem domain. This is despite a different model which allows for a higher problem complexity with multiple agent-controlled units. The increase in the complexity of the state-action space comes from an increased number of actions (from two to nine) as well as an increase in the amount of information a state is defined by. The local influence map of a unit alone consists of 7×7 tiles, each of which can have any positive integer assigned to it.

5.3.1 Experimental Setup and Parameter Optimization

The agent is evaluated in two different scenarios. In *Scenario A*, the agent controls one fast, weak combat unit against six slower, stronger enemy units. This scenario is identical to that used for the simple RL agent for 500- and 1,000-game runs in Chapter 4. I.e. here the performance of the CBR/RL approach can be directly compared to simple RL.

Scenario B is a variation of the first scenario that uses the same types of units. However, there are now three agent units and eight opponents. This scenario is targeted at evaluating both the effects and interactions of multiple units as well as a generally bigger scenario in comparison to the first, simplified one. Since the simple RL agent in Chapter 4 can control only one unit at a time, results for *Scenario B* cannot be compared to this, only against the built-in game AI.

To achieve the best performance possible, a first step is to optimise the parameters involved in both algorithmic components of the hybrid CBR/RL approach. This optimisation is omitted in other RL-based approaches that use RTS game micromanagement as testbeds (Shantia et al., 2011; Micić et al., 2011) and also in Chapter 4, where the emphasis is on finding the differences between temporal-difference RL algorithms.

A number of parameters are already decided upon through previous design decisions. Following the results in Chapter 4, it was decided to use Q-learning as the RL algorithm. In those previous experiments, the differences in performance between one-step Q-learning and Watkins' $Q(\lambda)$, which uses eligibility traces, were only marginal. Therefore, both algorithms are used in this evaluation. Watkins' $Q(\lambda)$, as the better performing one during the optimisation, was eventually chosen for the performance evaluation. The retrieval method in the CBR component is a NN algorithm that uses histogram-based similarity. The similarity threshold during retrieval is not fixed and a set of different values is tested.

Each set of parameters was tested 10 times in both scenarios with an empty case-base, first for 50 games, then for 500 games. Due to the relative simplicity of the scenarios that are evaluated, a good solution should be obtainable within a short run, but might be even better if the agent were allowed a longer time to learn. To compare it with the CBR/RL agent, a baseline with random action selection was added. Further, the results from the previous chapter using only RL for the simpler problem are compared in *Scenario A* where the agent controls only a single unit. Since the RL-only agent is not capable of controlling several units, this is the only scenario where a comparison is possible.

One change that was suggested by the several initial test runs was a change to the similarity metric used for the game state cases. These initial runs showed only very little learning success since the correlation similarity threshold that was used resulted in too many game states. As there is no reuse of unit cases that are grouped below different game cases, this leads to a state space that is too big to be sufficiently explored within a reasonable time. For this

reason, the similarity metric for the game state was adjusted to the point where game states are purely dependent on unit numbers.

The agent uses a declining ϵ -greedy exploration policy which starts with $\epsilon = 0.9$. This means that the agent initially selects actions 90% random and 10% greedy, i.e. choosing the best known action. The exploration rate declines linearly towards 0 over the course of the 50, 500 or 1,000 games respectively.

Parameter	Values
Number of Games	50, 500, 1000
Scenario	A(1vs6), B(3vs8)
Algorithm	Q-learning, Watkins' Q(λ)
CBR Unit Similarity Threshold ψ	60%, 80%,
RL Learning Rate α	0.2 , 0.3, 0.4
RL Discount Factor γ	0.6, 0.75, 0.9
Trace Decay Rate (for Q(λ) only)	0.6, 0.9
RL Exploration Rate ϵ	0.9 - 0

Table 5.1: Evaluation Parameters

The values in Table 5.1 result in 216 possible parameter combinations. For each combination, 10 runs of 50 games and 10 runs of 500 games were completed, more than one million games altogether. Experiments over 1,000 games are only used in the performance evaluation and not for parameter optimization due to the large number of experiments and the resulting requirements in terms of computational effort. After each 50- or 500-game run, a final 10 games were played where the agent acted completely greedy, using its best known policy at that point. The average performance in these final 10 games was used as an indicator of the fitness of the respective combinations. To determine which set of parameters was optimal overall, linear regression was used to build a model, at first for single algorithms and scenarios, eventually taking into account all combinations. Appendix C shows an excerpt of both the input file and of the resulting model that is produced by the WEKA machine learning software (Hall et al., 2009).

The optimal set of parameters according to this model is marked bold in Table 5.1. Results for the discount factor γ were inconclusive (it was not taken into account in the final regression model), therefore it was decided to use the discount rate applied in Chapter 4. This was the case for the similarity threshold ψ , which was part of the model in some of the sub-problems but not in the final model. Since one of the main aims of this paper is to evaluate the effects of integrating CBR with RL, it was decided to retain both similarity threshold values for subsequent experiments.

5.3.2 Performance

Having identified the optimal parameters, the next step is to run experiments using these parameters to analyse the agent's performance as a benchmark of the viability of the CBR/RL approach. In this step, experiments of 1,000 games length are also run in addition to games of length 50 and 500. This was not possible for all previous parameter combinations due to time and resource constraints. (See section 5.3.3 for details on the effort involved.) The longer runs are used to find out if there is any additional benefit in learning in a larger number of games and, if so, how the additional benefit scales compared to the additional effort. Furthermore, the number of runs for each configuration is doubled from 10 to 20. Figures 5.4, 5.5 and 5.6 show the results of running experiments on both scenarios with 50-game runs, 500-game runs and 1,000-game runs. Since there was no statistically significant difference between results for the 60% and 80% similarity thresholds, the diagrams display the results for 60% only, to improve readability. The average reward values are normalized to a range of 0% to 100% of the achievable score in a scenario, so the two scenarios can be compared. Achieving 100% of the possible reward means, that the agent played a perfect game in which it destroyed all opposing units without sustaining any damage itself. The figures also show the results of random action selection, which serves as a baseline for comparison. Furthermore, the results for *Scenario A* are compared with results of the simple reinforcement learner from Chapter 4. Since that learner uses a different model, there is a different initial minimum value for the reward. Reward development and overall reward are, however, comparable. The diagrams also include error bars that give a 95% confidence interval for the results.

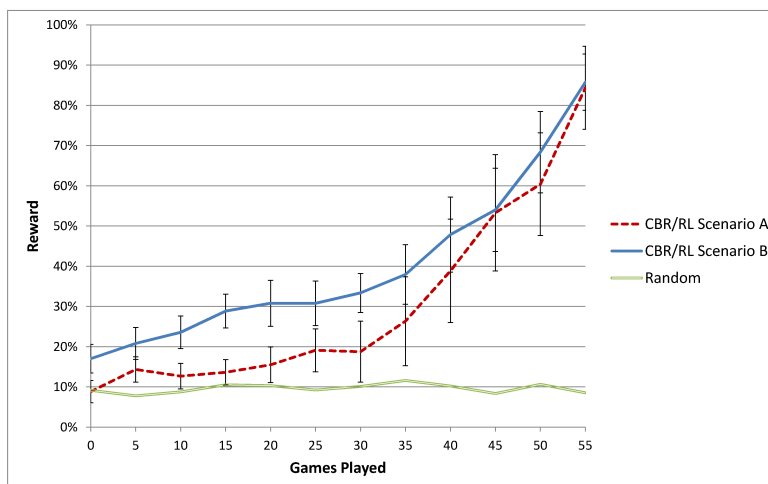


Figure 5.4: Results for 50 Game Runs

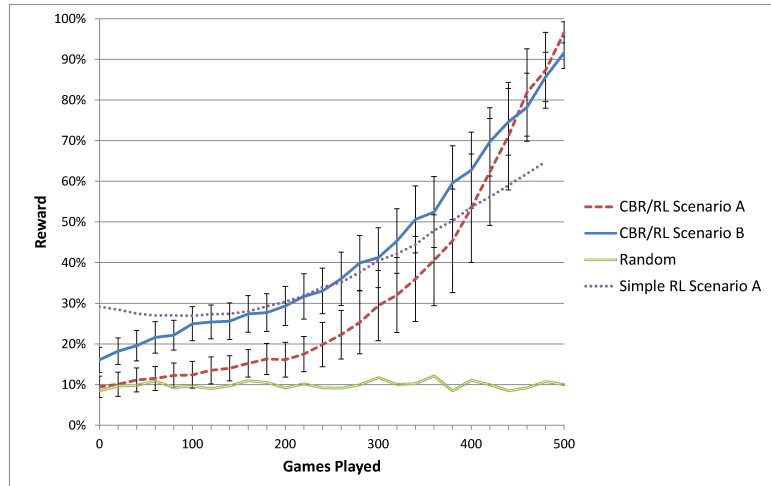


Figure 5.5: Results for 500 Game Runs

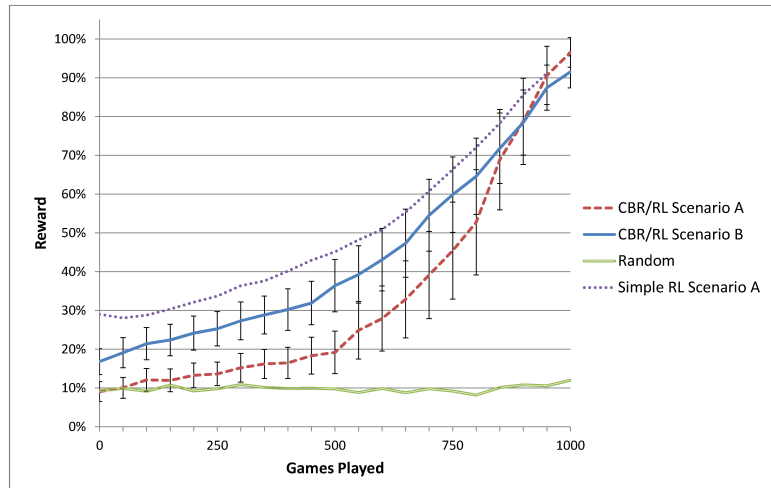


Figure 5.6: Results for 1000 Game Runs

5.3.3 Case Base Development and State-Action Space Exploration

Table 5.2 shows the metadata describing the development of the unit case-base for both scenarios. Since the number of game state cases is directly tied to the number of agent units, these cases are not separately listed. *Scenario A* has only one possible game state; *Scenario B* has three.

The table also contains data from additional runs in *Scenario B* using a similarity threshold $\psi = 95\%$. This threshold is introduced to determine performance implications for the current model and algorithm when working with larger case-bases. The runs are not listed in the performance charts so as not to overload the diagrams. However, the obtained reward is considerably worse compared to lower similarity thresholds: only about 40% of the maximum

Similarity Threshold ψ	<i>Scenario A</i>		<i>Scenario B</i>		
	60%	80%	60%	80%	95%
Number of Unit Cases:50 Games	2.26	7.3	6.63	101.8	1117.4
Number of Unit Cases:500 Games	2.3	11.1	7.8	204.67	2545.75
Number of Unit Cases:1000 Games	2.8	13.15	16.42	243.5	-
Visits per S/A Pair:50 Games	150.10	44.78	112.49	6.92	0.45
Visits per S/A Pair:500 Games	1264.20	252.15	806.93	32.83	0.04
Visits per S/A Pair:1000 Games	2041.49	441.34	1540.33	57.29	-
Unexplored S/A Pairs:50 Games	4.13%	26.33%	5.91%	52.90%	87.59%
Unexplored S/A Pairs:500 Games	2.90%	29.18%	2.35%	24.45%	85.24%
Unexplored S/A Pairs:1000 Games	2.58%	28.90%	1.69%	17.19%	-
Time Per Game in ms:50 Games	2716.11	2773.32	4218.20	7983.05	17559.11
Time Per Game in ms:500 Games	2678.18	2758.58	4506.65	7486.93	115486.5
Time Per Game in ms:1000 Games	2689.96	2726.64	4619.32	7729.05	-

Table 5.2: Case Base Statistics

reward in the 50 game runs and 70% for 500 game runs. The experimental run of 1,000 games with a 95% similarity threshold could not be finished because, after about 600 games, a single game takes up to ten minutes to complete on a normal desktop computer due to the amount of case comparisons required during each step.

5.4 Discussion

The figures show that the agent performs very well in both scenarios and is able to learn an optimal or near-optimal policy. The performance is far better than the random action selection baseline. Furthermore, the CBR/RL agent outperforms simple RL in 500-game runs in *Scenario A* where both can be compared. This suggests a faster speed of convergence to an optimal policy for the CBR/RL approach. In general, the highest increase in reward obtained for both scenarios is in the second half of a run when exploration is slowly cut back and knowledge has already been obtained. This is underlined by the shrinking 95% confidence interval that reflects a higher consistency in terms of performance because of an increasingly greedy action selection towards the end. The larger confidence intervals in Figure 5.4 show that the development of the average reward in the experiments with only 50 games is slightly more volatile than those with 500 or 1,000 games. Apart from this slightly more volatile development, which is also shown by wider error bars, the policies learned within 50 games are basically on the same level as those of longer runs in terms of reward. Generally, confidence intervals start out narrow, widen during the middle part of a run and shrink again when the policy becomes greedier again.

The average reward obtained by the CBR/RL agent over the course of one episode of 50, 500 or 1,000 games follows a similar pattern for both scenarios. For *Scenario A*, the

initial values are slightly higher than for *Scenario B*. This is potentially because three units that share the same memory are faster at obtaining a basic level of knowledge than a single unit. Eventually, the average reward for *Scenario A* becomes similar to that for *Scenario B*. When compared to the simple RL results, it is noticeable that the initial average reward for CBR/RL is lower. This is because of the different models used, especially the differences in action spaces. The model presented in this chapter has a much higher complexity than that used in Chapter 4. The initial average reward for simple RL is at the same level as random action selection in that model.

The data in Table 5.2 shows that, for 60% and 80% similarity thresholds, the number of unit cases in the case-base is relatively stable for experiments of different lengths, but vary considerably between different thresholds. The visits to a state-action pair scale linearly with the number of games played and are directly linked to the number of unit state cases, since one unit state has nine possible actions. The similarity in performance also means that the number of unit cases in the case-base, and thus the average number of visits to a state-action pair, have a very wide range in which optimal results can be achieved. For *Scenario B*, the number of cases for different unit situations can range from just below 6 to more than 250. The number of visits for an action behaves accordingly and reaches from about 1500 times that an action is executed on average, down to only about 7.

An important metric for the performance of RL is the percentage of actions which are never explored. Since the RL methodology requires an infinite number of visits to each possible action or state-action pair to guarantee convergence, a large number of unexplored actions points to a potential problem. In this case, the performance mostly seemed not to be influenced in a negative way, despite some experiments having up to 50% unexplored actions. The performance only dropped significantly for runs with a 95% similarity threshold which resulted in more than 80% unexplored actions. However, even for experiments with a similarity threshold of $\psi = 80\%$ where the performance is good, the amount of unexplored actions means that a large number of potentially good policies are never explored.

While the performance remains stable for similarity thresholds within a certain range, computational effort does not. The increased number of comparisons for retrieval resulting from the larger case-bases leads to longer run times. For thresholds of $\psi = 80\%$ and especially $\psi = 95\%$, running a larger number of games results in many more unit cases. With lower thresholds the agent stops adding new unit cases at some point and only explores the existing ones. For higher thresholds it keeps adding new unit states and new unexplored actions for much longer, potentially until the end of a run. This shows bad scaling and similarity thresholds that are too high for the given scenario, since cases are still being added despite greedy action selection. The behaviour for a 60% similarity threshold, which is close to constant in terms of unit case numbers and time required, is ideal. For $\psi = 80\%$, this is still

partially true, however the number of unit cases does increase for longer runs, even though not in an extensive way.

The ideal similarity threshold depends on the complexity of the scenario. Both scenarios used in this evaluation are comparatively simple, *Scenario B* slightly less than *Scenario A*. While it was not experienced in these experiments, it is to be expected that there is, depending on scenario complexity, a lower boundary for the similarity threshold, beyond which cases are too general and learning is no longer possible. For these reasons, using a full scale combat scenario will require a careful selection of the similarity threshold.

The results for *Scenario B* show that the agent controlling multiple units performs well. An optimal solution where all three units remain undamaged is obtained less often than the optimal solution in Scenario A. However, besides the fact that more actions are needed for a perfect game in Scenario B (about 120 vs 55) this is probably mostly because the information on other units is basically limited to the game case the current unit case is listed under (see Figure 5.3).

5.5 Conclusion and Influence on Hierarchical Approach

This chapter describes the integration of CBR and RL in an agent that controls units in combat situations in the RTS game StarCraft. The empirical evaluation demonstrates that the model which is developed reflects the environment well, and that the agent scales appropriately when used with a larger scenario where the agent controls multiple units instead of just one. The optimised algorithm parameters manage to obtain good results. The hybrid CBR/RL approach performs as well as or even better than the previous simple RL approach in the smaller scenarios where they are comparable, and manages to converge significantly faster towards a near-optimal policy.

As the next step is the creation of an agent that can handle the entire multi-level micromanagement problem, the analysis of the recorded meta-data gives valuable insights into the behaviour of the current architecture of the CBR component and its potential for future developments. The current experiments work best with a low similarity threshold which still lead to good results in terms of performance. Very high thresholds can lead to significant increases in terms of run-time and case-base size, while at the same time decreasing the agent's overall performance. However, since the number of cases also increases for more complex scenarios, a better handling of large scale case-bases is a problem that will have to be addressed in the future. When using the current approach in larger scenarios and eventually for the full micromanagement problem, higher similarity thresholds might also be required to distinctively distinguish between cases.

Having shown the viability of hybrid CBR/RL, the next step is to increase the complexity of the problem space by enabling the agent's access to more StarCraft unit types in even larger scenarios. The subsequent and final step also includes the ability to manage groups of units at squad level. In this chapter, the behaviour of allied units is not taken into account at all in the reward computation to avoid diluting the effect of the units' own actions. The hierarchical CBR/RL architecture presented in the next chapter enables learning more team-oriented strategies. The use of a two-level approach to abstract the simulation environment in this chapter turned out to be problematic. During the empirical evaluation, this layered architecture resulted in too many parallel cases to be evaluated within a reasonable time. The performance in the evaluation showed that using a single layer turned out to be sufficient for the scenarios addressed here. However, for the even larger scenarios presented in the next few chapters, the problems arising from using RL in a hierarchical approach will have to be addressed.

A Hybrid Hierarchical CBR/RL Architecture for RTS Game Micromanagement

This chapter describes the concept of a hierarchical architecture of CBR/RL modules that is used to create an agent which addresses the micromanagement problem in RTS games. The architecture and its constituent separate modules are based on previous approaches described in Chapters 4 and 5. Using insights gained through creating those modules as well as related work on layered learning and hierarchical agent architectures, a number of layered CBR/RL components are devised and, in subsequent chapters, implemented in order to subdivide the unit micromanagement problem. Since there are several interacting components compared to just the single one used in Chapters 4 and 5, the hierarchical approach leads to additional general considerations which are explained in this chapter. The hierarchical CBR/RL agent is designed to be able to coordinate within and between squads of units as well as navigating the game world with individual units. The agent acquires the knowledge it needs to solve the sub-problems in each layer and subsequently uses that acquired knowledge to win micromanagement scenarios.

In order to map hierarchical CBR/RL to the micromanagement task, the overall problem is subdivided into a number of interconnected layers that each addresses sub-problems at different levels of reasoning. Integrating these layers then allows the solution of the micromanagement problem in its entirety. Subdividing the problem enables a more efficient solution than when addressing the problem on a single level of abstraction. In the context of CBR/RL, using a single case-base and associated actions/solutions would either result in case representations which are too complex to be used for learning in reasonable time, or that require such a high level of abstraction that it prevents any meaningful learning process. The CBR/RL approach presented in Chapter 5 was based on two layered case-bases. However, initial evaluation results pointed to issues with the case representation on the top layer. This effectively resulted in only a single layer being used in the eventual evaluation. This problem is addressed in the modelling process for the different modules as well as in changes to the overall evaluation as described in Section 6.2.

The chapter is structured in the following way. First, the considerations behind modelling the micromanagement problem in the way that is chosen are explained. This includes an enumeration of the tasks involved and their inherent layering as part of an RTS game. Then, the procedure for evaluating the architecture and the reasoning behind it in the context of acquiring knowledge on how to play the game are described. This also shows the connection to the LL paradigm. Finally, the process of mapping in-game unit entities to case-base unit entities is explained, since this is a recurring step that is crucial for the effective use of RL.

6.1 Modeling a Hierarchical CBR/RL Architecture in a RTS Game

An important initial step in creating a ML agent that learns how to manage groups of combat units in real-time is the creation of an abstract model. While representing the environment in a representation that a computer agent can work with was part of Chapters 4 and 5 as well, this step is even more complex and important for a hierarchical architecture. The complexity and importance increase, since the larger scope of the actions leads to higher requirements for the knowledge representation, both in terms of precision and the amount of information which has to be abstracted.

To summarise, this step should lead to a representation of the problem for the agent that fulfils a number of criteria.

- The representation provides the agent with appropriate actions that allow it to perform the desired task (i.e. fighting opposing combat units).
- The representation reflects the game world in sufficient detail to provide enough information for all required actions. This means that the representation must allow the agent to distinguish between states that are inherently ‘different’ in terms of the problem.
- The representation allows an agent to either directly control a large number of units or is easily extensible to allow the control of a large number of units. This is in contrast to the scenarios in Chapters 4 and 5, where only a single unit or a very limited number of units was managed.

The first step for the creation of such a representation is to identify the core problems inherent in the StarCraft RTS game that have to be addressed with this architecture. Section 3.1 explained these problems inherent in RTS games in general and in StarCraft in particular. The layering in RTS games, shown in Figure 3.1, leads to most RTS agents being hierarchical. As elaborated in Section 3.2, agents usually have a top-down approach when it comes to giving orders or executing actions and plans. A high-level planner will decide on an action, which will subsequently be passed down to subordinate managers. These lower-level managers in turn

might pass on parts of that action to managers which are even further down the hierarchy. This hierarchy extends down to a per-unit level.

The architecture presented in this chapter covers the micromanagement component that includes part of the general RTS game problems. More specifically, the architecture addresses the tasks enclosed by the solid red square shown in Figure 6.1. Reconnaissance is currently not part of the framework, as the CBR/RL agent only works with units which are already visible. While reconnaissance includes tactical elements, it also ties into the strategic layer. As strategic decisions, which are not part of the hierarchical approach, are a major influence, reconnaissance actions are outside the scope of this thesis.

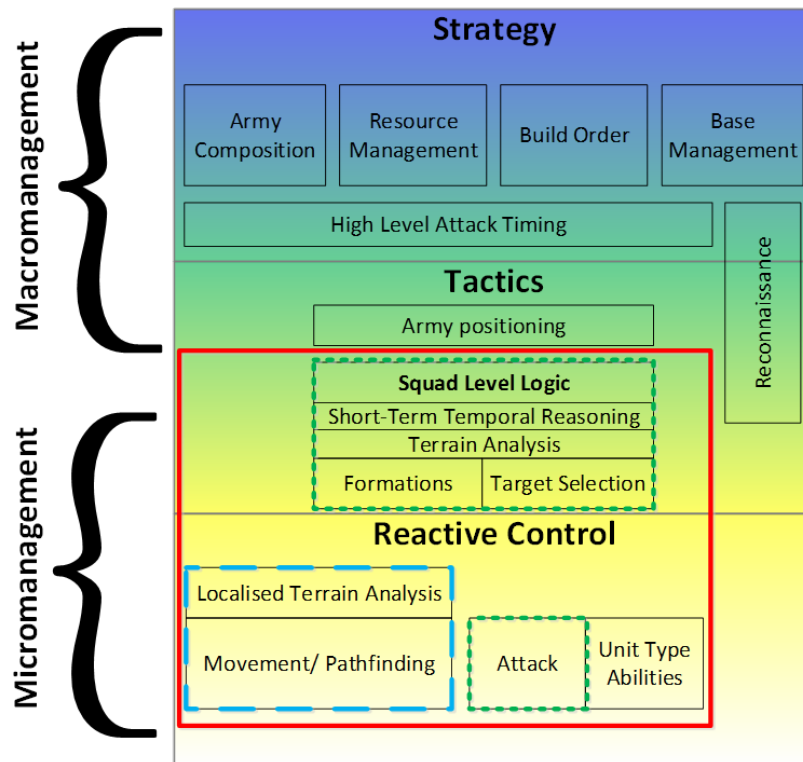


Figure 6.1: RTS Micromanagement Tasks

Based on this task decomposition, three distinct organisational layers are identified. The *Tactical Level (Level One)* is the highest organisational level and represents the entire world the agent has to address, i.e. the entire battlefield and the entire solid red square in the

figure. The *Squad Level (Level Two)* reflects the *Squad Level Logic* and tasks attached to it in the context of a RTS, represented by the dotted green square. Sub-tasks represented here concern groups of units, potentially spread over the entire battlefield. In the implementation of the hierarchical agent architecture developed in subsequent chapters, the reactive *Attack* action is also included, since it directly translates to an in-game action without any further logic being required. Finally, the *Unit Level (Level Three)* is the bottommost layer. This layer covers pathfinding, works on a per-unit basis and is denoted by the dashed blue square in the diagram. Since only single units are concerned, the spatial information handled on this layer has the smallest scope. However, the information used here is also the most precise and the least abstract. There is an inverse correlation between the precision of game information that is used at a certain level (lowest precision/highest abstraction at the highest level) and the scope of an action (smallest scope at the unit level). Figure 6.2 depicts the three distinct organizational levels that were identified based on the task decomposition in Figure 6.1.



Figure 6.2: Levels, Information Abstraction and Action Scope of our Architecture

Translating this layered problem representation into a CBR/RL architecture is done through a number of hierarchically interconnected case-bases. Section 3.5 gave detailed background on general hierarchical CBR and layered learning, two areas that influenced the creation of this hierarchical model. The approach to HCBR here is strongly inspired by that of Smyth & Cunningham (1992) who built an HCBR system for software design.

One major difference between the approach described here and the one in (Smyth & Cunningham, 1992) is that the use of RL for updating fitness values in the hierarchically interconnected case-bases means that each case-base has its own *Adaptation*-part of the CBR cycle (Aamodt & Plaza, 1994). Furthermore, in terms of running the modules on different levels of the model, several lower-level CBR cycles can be executed during one high-level CBR cycle. There are also strong connections between the hierarchical architecture presented here and the layered learning paradigm. Most of the LL principles presented in Section 3.5 apply and the evaluation concept presented in Section 6.2 further emphasises the similarities.

Figure 6.3 shows the case-bases resulting from modelling the problem in this hierarchical fashion. Both the tactical level and the unit level are represented by a single case-base. In terms of action scope, the unit level is only responsible for *Navigation*. The intermediate squad level has one case-base for two possible actions on that level, *Attack* and *Formation*. Each case-base is part of a distinct CBR/RL module. Higher levels can then use the lower level components to interpret their solutions. As a result, higher levels base their learning process on the knowledge previously acquired on lower levels.

Case descriptions on any level consist of abstracted information on the units that are involved and that make up the game state at that particular point in time. The level of abstraction depends on where in the architecture the respective case-base is located. Higher-level case descriptions, which cover more in-game units and environment, also have a different level of abstraction since they require different information to make their decisions. At the single-unit-level, both the spatial information and the unit information are closest to the actual in-game data.

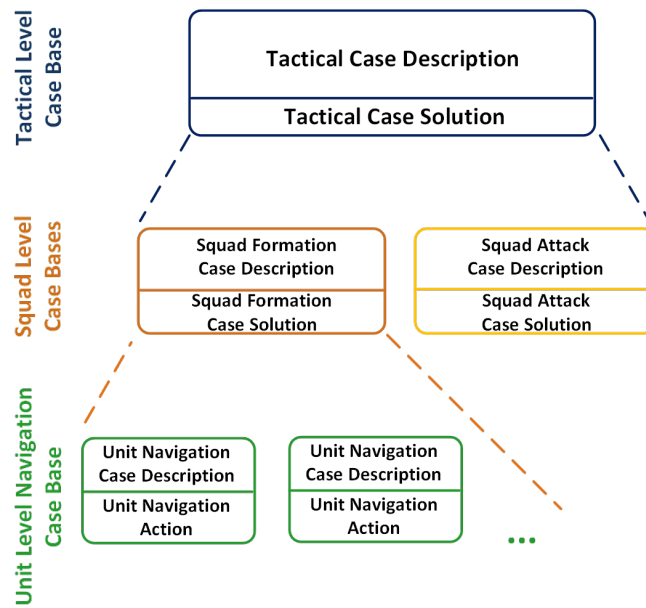


Figure 6.3: Hierarchical Structure of the Case-Bases

All tasks identified in Figure 6.1 as belonging to the micromanagement part of a RTS game are either addressed directly by a CBR/RL component or through a function that is executed by one of those components. Chapters 7, 8 and 9 describe the considerations and implementation behind each of the layers and individual modules in detail. As part of the chapter that describes *Level One*, Section 9.2 also contains Figures 9.1 and 9.2, a representation of the overall architecture logic both in terms of action selection and reward propagation.

6.2 Evaluating the Hierarchical Architecture

Ideally, acquiring knowledge should be done while running the entire system in an integrated fashion, similarly to the evaluations in Chapters 4 and 5. This would lead to each of the four case-bases for the overall system, for *Navigation*, *Attack*, *Formation* and *Tactical Unit Selection*, to be trained at the same time.

However, due to the layered nature of the architecture and the learning behaviour of RL, this would be difficult to achieve within a reasonable time. On higher levels, the hierarchical agent bases its case selection policy on knowledge previously acquired on lower levels. This means that when a certain solution for an *Attack* case on *Level Two* of the architecture is selected, it is assumed that the performance of the underlying *Navigation* component on *Level Three* is consistent and allows replication of prior actions. RL requires this assurance in order not to constantly have to relearn the values of state-action pairs on higher levels because the underlying levels gained new knowledge. If all levels were to learn at the same time, this would also mean that there are constant changes to the best known actions on each level at all times. While values for state-action pairs for higher levels would eventually stabilise, this could only happen after the lower levels learned optimal policies for their respective tasks and did not change fitness values. Finally, *Principle 3* of the layered learning paradigm (see Section 3.5) requires learning in individual layers to happen sequentially. While there has been research on concurrent LL (Whiteson & Stone, 2003), this was only done in a very narrow problem domain and the authors concluded that the approach was not easily generalizable.

Therefore, it was decided to evaluate and train the system in a sequential bottom-up fashion. Lower level components are evaluated first and the acquired knowledge for the respective tasks is retained in the appropriate case-bases. Subsequently, the next-higher level is evaluated and trained using the lower-level cases as a foundation. In order to avoid diluting the learning- and evaluation process of higher levels, cases in lower-level case-bases are not changed once they are reused by a higher-level evaluation.

This procedure results in the following sequence of actions:

1. Evaluate and train *Level Three* of the architecture, the pathfinding component (see Chapter 7).
2. Evaluate and train *Level Two* of the architecture, while re-using knowledge previously acquired from the pathfinding component for navigation.
 - Evaluate and train the *Formation* component (see Section 8.1).
 - Evaluate and train the *Attack* component (see Section 8.2).
3. Evaluate and train *Level One* of the architecture, the *Tactical Unit Selection* component, while re-using the knowledge previously acquired in the case-bases at *Levels Three* and *Two* (see Chapter 9).

The evaluation of the hierarchical architecture is thus done in a multi-step process. Chapter 7 describes the creation, evaluation and training of the lowest level of the architecture, the *Navigation* component. Chapter 8 describes the second level of the architecture, the creation, evaluation and training of components for *Attack* and *Formation*. This chapter re-uses knowledge stored in the lower *Level Three* case-base for *Navigation*. Finally, Chapter 9 integrates all lower case-bases into an agent that makes *Tactical Unit Selection* decisions and re-uses the knowledge previously acquired on *Levels Two* and *Three*.

In order to avoid noise that comes from updating fitness values through RL, the case-bases for lower levels are no longer modified when they are re-used on higher levels: While components on *Level Two* are evaluated and trained, the *Navigation* case-base on *Level Three* is re-used but no longer updated, thus avoiding noise in the learning process. This means that the *Navigation* case-base and other lower-level case-bases that are re-used by higher-level components must contain cases and solutions that cover **all** potential situations in order to enable a good performance for the higher-level components. This requirement necessitates training scenarios for all components to learn how to act in these potential situations. The training scenarios are described in the relevant sections for each component. Re-using lower-level knowledge for different scenarios also automatically evaluates the knowledge transfer capabilities of those lower-level modules.

This evaluation procedure is not ideal since it partially negates the online learning characteristic of the CBR/RL agent. However, the alternative is a very noisy learning process that would seriously complicate the use of RL. Since the sequential learning process is also such an inherent characteristic of the underlying LL paradigm, it was decided that an iterative learning procedure is to be preferred and the development of a technique that enables concurrent reinforcement learning on multiple layers of a hierarchical architecture is left for future work.

6.3 Unit Mapping

A recurrent issue in all of the components described in this thesis is the translation of the game world into a representation that the AI agent can work with as explained in Section 6.1. For the simplified micromanagement problems described in Chapters 4 and 5, strong abstractions are used which reduce the actual game state to a limited, manageable number of feature values.

The hierarchical architecture described in this chapter uses a more precise representation of the game state to allow the agent to distinguish between a larger number of more complex game situations. The most important part of the game state representation is the state of the units, both for the agent and the opponents, at a certain point in time. Also, in addition to serving as an integral part of a game state description that is used to retrieve similar cases from the case-base, units are part of the case solutions.

For the sub-problems of *Formation*, *Attack* and the overall assignment of units to *Tactical Unit Selection* tasks as described in subsequent chapters, the case solutions consist of assigning units certain commands, for example:

- Unit A - Assign to *Retreat* action.
- Unit B - Assign to *Formation* action, (*Formation Solution: Slot 1*).
- Units C, D, E - Assign to *Attack* action, (*Attack Solution: Target Opponent 3*).

Every time an assignment problem emerges in any of the modules for *Tactical Unit Selection*, *Attack* and *Formation*, the number of possible solutions is based on permuting the available units among available solution ‘slots’. This means that it is crucial which particular unit is assigned to which particular slot. In order to re-use prior experiences in the game, there has to be a consistent mapping between units in the problem descriptions and -solutions stored in the case-bases and the units in the current game state which represents the new problem. In order to create a meaningful connection between these two sets of entities, the overall unit-to-unit similarity has to be optimised. This is a combinatorial problem, since each unit in the current game state has to be compared with each unit in the stored case that is being examined. In order to solve this combinatorial optimisation problem in polynomial time, the *Kuhn-Munkres algorithm* or *Munkres assignment algorithm* ((Kuhn, 1955): see Appendix B for algorithm steps) is used. This algorithm solves the problem in $O(n^3)$ for n units. This as an acceptable running time since the number of units remains comparatively small and only the units from the single case that has already been determined as being similar to the current game state are examined.

As a first step, a case is selected that is ‘similar enough’ to the current game state. The selection of suitable similarity thresholds and similarity metrics is described in detail in

Table 6.1: Unit Assignment Example

	$UnitG_1$	$UnitG_2$	$UnitG_3$	$UnitG_4$	$UnitG_5$
$UnitDB_1$	0.5	0.8	0.2	0.9	0.5
$UnitDB_2$	1	0.9	0.3	0.9	0.3
$UnitDB_3$	0.7	0.8	0.7	0.8	0.2
$UnitDB_4$	0.8	0.3	1	0.1	0.5
$UnitDB_5$	0.1	0.2	1	0.2	1

the sections for the respective CBR/RL modules. Once such a case has been selected, the similarities between each of the n units in the game state and each of the n units in the case description are computed and entered into an $n * n$ matrix. The Munkres algorithm is then used to compute the optimal assignment of game state units to case units. Table 6.1 shows an example assignment. Unit similarities between 0 and 1 are computed by comparing a number of unit attributes such as *unit type*, *unit health* and *unit position*, always depending on the sub-problem the particular unit assignment is used for. The selected unit associations between the units currently active in the game ($UnitG_{1..n}$) and those associated with a certain case ($UnitDB_{1..n}$) are shown in bold.

The problem of maximising unit-to-unit similarity for two sets of units from a case in the case-base and the in-game units is closely related to the problem of computing case-to-case similarity. However, case descriptions often contain more information than only unit data. More importantly, the effort involved in computing the maximum unit-to-unit similarity for two sets of units is too big to use it for several thousand cases while making real-time decisions. Therefore, general case-to-case similarity is computed using more high-level, less precise methods. The unit-to-unit similarity is only optimised once a similar case has been found. In Section 5.1.1 a histogram-based comparison was used to compare cases that rely heavily on high-dimensional spatial stored in multiple IMs. While this worked quite well for the given problem, it was decided to use a less abstract approach in Chapter 9. The problem there is similar in that the particular case representations in that module require an abstract similarity metric to speed up case-to-case comparisons in real-time. However, the case descriptions are less focused on fewer cases where the description consists of a large number of low-dimensional data points. Instead, there is a vast number of cases with case descriptions made up of unit attribute vectors, effectively high-dimensional data points. Additionally, the requirements for precision are higher in this multi-level architecture. This is because a hierarchical architecture propagates errors from lower to higher levels, potentially multiplying these errors. Given these considerations, the Hausdorff distance was used for case similarity computation. Section 3.4.2 described the details of using the Hausdorff distance for similarity computation.

6.4 Summary

This chapter presented several considerations and mechanisms that are supplementary to creating a hierarchical CBR/RL architecture. The main aspect is the high-level modelling described in Section 6.1. Through the analysis of tasks involved in RTS games, a decomposition into suitable modules and an arrangement of those modules into a hierarchical structure, a suitable representation of the problem domain, is created. Subsequently, an evaluation procedure is designed that takes into account the unique properties of the structure. Finally, a solution to the crucial issue of mapping unit representations that are stored in memory to newly occurring problems is presented. Having introduced these preparatory concepts, the next three chapters are concerned with the creation of the relevant modules that Section 6.1 identifies as essential for micromanagement in RTS games.

Architecture Level Three: Unit Pathfinding using Hybrid CBR/RL

¹This chapter describes the module created for the lowest layer of the hierarchical architecture. Since higher levels in the architecture re-use the modules on lower levels, the modules are described in reverse order, with lower-level components being described first. To begin with, the CBR/RL module for pathfinding and navigation is presented.

The module created for *Level Three* addresses the *Navigation* task that is part of the lowest layer of tasks involved in a RTS game, the *Reactive Control Layer* (see Figure 7.1).

Some of the actions that fall within this layer can be directly executed through the game itself and do not need any further learning process. Specifically, the reactive part of the *Attack* action is combined into the *Attack* CBR/RL module that handles target selection and is described in detail in Section 8.2. Special unit abilities are actions specific to a single type of StarCraft unit. These abilities are powerful and very diverse, basically necessitating an extensive learning process (and thus a separate module) for each in order to use them effectively. Because of this requirement and their limitation to StarCraft only, it was decided to not use them and limit the agent to standard melee and ranged attacks only.

The remaining tasks to be addressed in the reactive layer are unit navigation and movement, a core component of any RTS game. The task of navigating the game world at this level requires even more exact and prompt reactions to changes in the game environment than micromanagement in general. There are numerous influences a unit has to take into account when navigating the game world. This includes static surroundings, the agent's own and opposing units and other dynamic influences. Navigation and pathfinding are not problems unique to video games, but are a topic that is of great importance in other areas of research, such as autonomous robotic navigation. Unit navigation in an RTS game is closely related to autonomous robotic navigation, especially when looking at robotic navigation in a simulator without the added difficulties of managing external sensor input (Laue et al., 2006).

¹ The contents of this section are based on Wender & Watson (2014a), a paper that has been published in the conference proceedings for ICCBR2014.

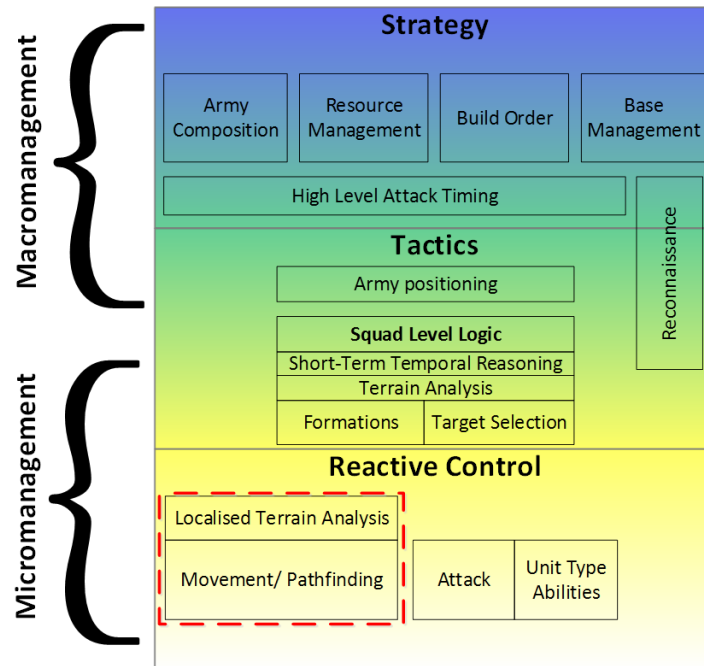


Figure 7.1: Navigation in the Context of RTS Tasks

CBR, sometimes combined with RL, has been used in various approaches for autonomous robotic navigation. Ram & Santamaria (1997) described the *self-improving robotic navigation system (SINS)* which operated in a continuous environment. One aim of SINS, similar to the approach presented here, was to avoid hand-coded, high-level domain knowledge. Instead, the system learned how to navigate in a continuous environment by using reinforcement learning to update navigation case fitness. Kruusmaa (2003) developed a CBR system to choose the least risky routes in a grid-based map, similar to the influence map (IM) abstraction used in this thesis. The CBR system also found solutions that lead to a goal position faster. However, the approach only worked properly when there were few large obstacles. Additionally, the state space was not very big when compared to RTS game environments such as StarCraft.

Both these approaches were non-adversarial, however, and thus do not account for a central feature of RTS games that raise the complexity of the problem space considerably. As part of this component, the presence of opposing units is addressed by using IMs to abstract the battlefield which can contain dozens of units on either side at any one time.

The A* algorithm (Hart et al., 1968) and its many variations are the most common exponents of search-based pathfinding algorithms used in video games while also being successfully applied in other areas. However, the performance of A* depends heavily on a suitable heuristic and is also very computation-intensive as it frequently requires complex pre-computation to enable real-time performance. A* variations, such as the real-time heuristic search algorithm kNN-LRTA* (Bulitko et al., 2010), which uses CBR to store and retrieve sub-goals to avoid revisiting known states too often, have been used extensively in commercial game AI and game AI research (Bourg & Seemann, 2004; Hagelback, 2012; Sturtevant, 2012).

The appeal of researching navigation in video games in general and in StarCraft in particular lies in having a complex environment with various different influences that can be precisely controlled. Navigation is also integrated as a crucial supplementary task in the much broader problem of performing well at defeating other players. As part of more high-level modules presented in later chapters, the CBR/RL navigation module can be used and evaluated in numerous ways.

7.1 CBR/RL Integration and Model

The core part of the *Navigation* component is a case-base with case-descriptions that contain data on the environment surrounding a single unit. Navigation cases use both information from the game world and the target position given by a higher-level reasoner on the architecture layer above. Case descriptions are mostly abstracted excerpts of the overall state of the game environment. Case solutions are movement actions of a unit. Units acting on this level are completely autonomous, i.e. there is only one unit per case and no coordination with other units. This means that the information taken from the game world is not concerned with the entire battlefield, but only with the immediate surroundings of the unit in question.

The RL component is used to learn fitness values for case solutions. This part of the architecture is based on previous research described in Chapters 4 and 5. While the complexity of the model here is much higher, the problem domain remains the same, which is why one-step Q-learning is used for policy evaluation. In this specific setting, Q-learning showed the best performance when compared to other tested algorithms by a small margin, with Watkins's $Q(\lambda)$ performing minimally better than one-step Q-learning. However, the one-step version was used instead of Watkins's $Q(\lambda)$ since the implementation of the one-step version is considerably easier and evaluations using one-step algorithms are computationally less expensive when compared to algorithms using eligibility traces.

In order to use RL to update the fitness of case solutions, the *Navigation* problem is expressed as a *Markov Decision Process*. As explained in the background on RL in Section 3.3, a specific MDP is defined by a quadruple $(S, A, \mathcal{P}_{ss}^a, \mathcal{R}_{ss}^a)$. In this case, the state space S

is defined through the case descriptions. The action space A is the case solutions. \mathcal{P}_{ss}^a are the transition probabilities. While StarCraft has only few minor non-deterministic components, its overall complexity means that the transition rules between states are stochastic. As a result, subsequent states s_{t+1} can be reached when taking the same action a_t in the same state s_t at different times. \mathcal{R}_{ss}^a represents the expected reward, given a current state s , an action a and the next state s' .

7.1.1 Navigation States: Case Description

The case description consists of an aggregate of the following information:

- Static information about the environment;
- Dynamic information about opposing units;
- Dynamic information about allied units;
- Dynamic information about the unit itself, and
- Target specification from the level above.

Most of this information is encoded in the form of influence maps (IMs) (Khatib, 1986). IMs are one of the main techniques that is used throughout this thesis to create representations of the game environment that the AI agent can work with.

Influence maps or *potential fields* (PFs) for the abstraction of spatial information have been used in a number of domains, including game AI, mostly for navigation purposes. Initially IMs were developed for robotics Khatib (1986). Uriarte & Ontañón (2012) used IMs to create ‘kiting’ behaviour in StarCraft units. Kiting is a hit-and-run movement that is similar to movement patterns that the AI agent created in Chapter 4 learns for micromanagement. However, the authors’ focus was solely on the hit-and-run action and not on general manoeuvrability and the potential to combine pathfinding and tactical reasoning into a higher-level component.

Hagelbäck (2012) described the creation of a multi-agent bot that was able to play entire games of StarCraft using a number of different artificial potential fields. However, large parts of the bot used non-potential field techniques such as rule-based systems and scripted behaviour, highlighting the common approach to combine different ML techniques to address larger parts of the problem space in an RTS game. The bot architecture created by Hagelbäck (2012) is further examined in Section 3.2.

For the pathfinding component devised in this chapter, there are three distinct influence maps: one for the influence of allied units, one for the influence of enemy units and one for the influence of static map properties such as cliffs and other impassable terrain. This influence

Chapter 7. Architecture Level Three: Unit Pathfinding using Hybrid CBR/RL

map for static obstacles also contains information on unit positions, as collision detection means that a unit cannot walk through other units.

While higher levels of reasoning in the architecture use spatial information that can cover the entire map, the relevant information for a single unit navigating the game environment on a micromanagement level is contained in its immediate surroundings. Therefore, the perception of the unit is limited to a fixed 7×7 cutout of the overall IM directly surrounding the unit. An example of the different IMs for varying levels of granularity and other information contained in the environment can be seen in Figure 7.2.

The red numbers denote influence values. The influence values are based on the damage potential of units. This particular figure shows only the enemy influence in order to not clutter the view. Any square in the two influence maps for enemy and allied units has assigned to it the damage potential for adjacent units. The IM containing information on passable terrain only contains a true/false value for each field. In Figure 7.2, impassable fields are greyed out.

The 'local' IM that represents the perception of a single unit it surrounds, is marked by the green squares. These green squares form a sub-selection of the overall yellow IM. Only values from this excerpt are used in the case descriptions.



Figure 7.2: Game Situation with Influence Map Overlay

Chapter 7. Architecture Level Three: Unit Pathfinding using Hybrid CBR/RL

In addition to the spatial information contained in the IMs, three other values are part of a case description: *Unit Type*, *Previous Action* and *Target Location*. The *Target Location* indicates one of the 7×7 fields surrounding a unit and is the result of a decision on the higher levels. In Figure 7.2, the current target location is marked by the blue **X**. The *Unit Type* is indicative of a number of related values such as speed, maximum health and ability to fly. The *Previous Action* is relevant as units are able to carry momentum over from one movement action to the next.

7.1.2 Navigation Actions

The case solutions are concrete game actions. There currently are four *Move* actions for the four different cardinal directions, i.e. one for every 90° (see Figure 7.3).

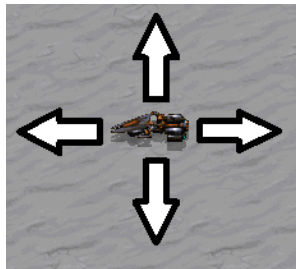


Figure 7.3: Possible Movements for a Unit

7.1.3 Navigation Reward Signal

Reward is computed on a per-unit and per-action basis, similar to the simple RL agent presented in Chapter 4. The reward signal is crucial to enable the learning process of the agent and has to be carefully selected. As the agent tries to achieve a number of different goals, depending on the given scenario, there is a composite reward signal. For damage avoidance, the reward signal includes the difference in health h_{unit} of the agent unit in question between time t and time $t + 1$. The reward signal also includes a negative element that is based on the amount of time t_a it takes to finish action a . This is to encourage rapid movement towards the goal position. To encourage target approximation, the other central requirement besides damage avoidance, there is a feedback (positive or negative) for the change in distance d_{target} between the unit's current position and the target position. Finally, to account for inaccessible fields and obstacles, there is a penalty if a unit chooses an action that would access a non-accessible field. This penalty is not part of the regularly computed reward after finishing an action but is attributed whenever such an action is chosen and leads to immediate re-selection.

The resulting compound reward that is computed after finishing an action is

$$\mathcal{R}_{ss'}^a = \Delta h_{unit} - t_a + \Delta d_{target}.$$

When looking at scenarios that evaluate the performance of the two subgoals *Target Approximation* and *Damage Avoidance*, only the parts of the reward signal that are relevant for the particular subgoal are used.

7.2 Similarity Computation and Navigation Module Logic

The CBR component in the pathfinding module reflects a general CBR cycle (Aamodt & Plaza, 1994) but does not use all of the cycle’s phases. Similar to the other CBR modules created in this thesis, during the **Retrieval** step, the best-matching case is found using a simple kNN algorithm where $k = 1$, given the current state of the environment. If a similar state is found, its solution, made up of all possible movement actions and associated fitness values, is returned. These movement actions can then be **Reused** by executing them. There is no further adaptation of the solution apart from RL changing fitness values.

Currently, there is no **Revision** of case descriptions stored in any of the case-bases. Once a case has been stored, the case descriptions, consisting of the abstracted game information and of the solutions from upper levels, are not further changed. However, the fitness values associated with case solutions are adjusted each time a case is selected. These updates are done using RL.

Retention is part of the retrieval process. Whenever a case retrieval does not return a sufficiently similar case, a new one is generated from the current game environment state.

The similarity measure that is used to retrieve the cases plays a central role, both in enabling the retrieval of a similar case and in deciding when new cases have to be created if no sufficiently similar case exists in the case-base. Part of the novelty of the approach presented here is the use of IMs as a central part of the case descriptions. These IMs are then used in the similarity metric when looking for matching cases.

In all three types of influence maps that are used, the similarity between a map in a problem case and a map stored in the case-base is an aggregate of the similarities of the individual map fields. Comparing the IM that contains information on the accessibility of single map plots is relatively easy since each field is identified by a Boolean value, which means that similarity between the plot of IMs in a problem case and in a case in the case-base is either 0% or 100%. The similarity of a single field describing the damage potential of enemy or the agent’s own units, is decided by the difference in damage to its counterpart in the stored case. I.e. if one field has the damage potential 20 and the counterpart in the stored case has the damage potential 30, the similarity is decided by the difference of 10. Currently, all

differences are mapped into one of four intervals between 0% and 100%. This is also subject to if a unit influence exists in both cases and to how big the damage potential difference is. An example similarity computation is shown in Figure 7.4.

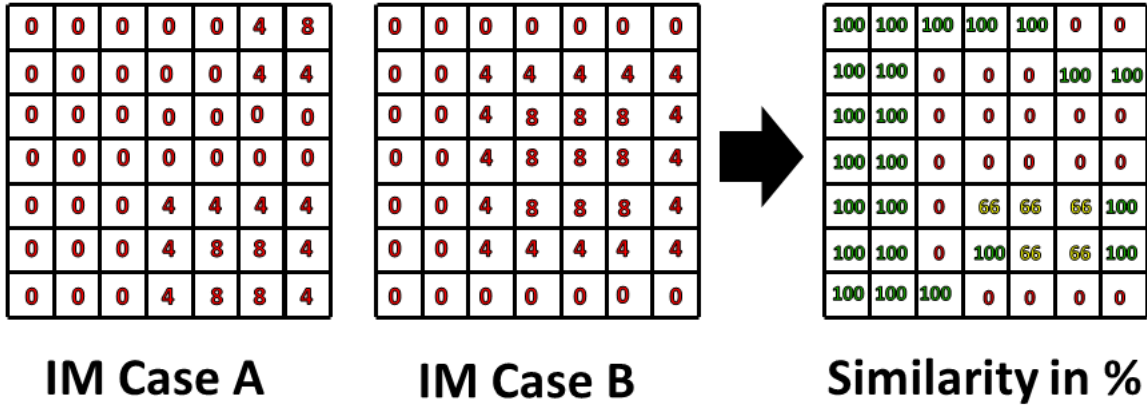


Figure 7.4: Example of IM Field Similarity Computation

It was decided not to abstract the information stored in the IM fields any further and do a direct field-to-field comparison. However, as a large case-base could slow the search considerably, future improvements could include modifications such as accounting for geometric transformations on the IMs in order to reduce the number of cases. In higher levels of the architecture (see Chapters 8 and 9), more abstract ways of representing and comparing game state information are used.

The similarity between *Last Unit Actions* is determined by how similar the chosen movement directions are, i.e. same direction means a similarity of 1, orthogonal movement 0.5 and opposite direction means 0. *Unit Type* similarity is taken from a table that is based on expert knowledge as it takes a large number of absolute (health, energy, speed) and relative (can the unit fly, special abilities) attributes into account. The similarity between *Target Positions* is computed as a distance between target positions in cases in relation to the maximum possible distance (dimension of the local IM field).

Attribute	Description	Similarity Measure
Agent Unit IM	Map with 7x7 fields containing the damage potential of adjacent allied units.	Normalised aggregate of field similarity.
Enemy Unit IM	Map with 7x7 fields containing the damage potential of adjacent allied units.	Normalised aggregate of field similarity.
Accessibility IM	Map with 7x7 fields containing true/-false values about the accessibility.	Normalised aggregate of field similarity.
Unit Type	Type of a unit.	Table of unit similarities between 0 and 1 based on expert knowledge.
Last Unit Action	The last movement action taken.	Value between 0 and 1 depending on the potential to keep momentum between actions.
Target Position	Target position within the local 7x7 map.	Normalised distance.

Table 7.1: Navigation Case-Base Summary

Table 7.1 summarises the case description and related similarity metrics.

7.3 Empirical Evaluation and Results

A core parameter to determine in any CBR approach is, given a chosen case representation and similarity metric, a suitable similarity threshold ψ that enables the best possible performance for the learning process. ‘Suitable’ means on the one hand that enough cases are created to distinguish between inherently different situations, while on the other hand the case-base is not inundated with unnecessary cases. Keeping the number of cases to a minimum is especially important when using RL. In order for RL to approximate an optimal policy π^* , there has to be a sufficient state- or state-action space coverage (R. S. Sutton & Barto, 1998). If the case-base contains too many cases, not only will retrieval speeds reduce, but a lack of state-action space coverage will diminish the performance or even prevent meaningful learning altogether.

The model described in Section 7.1 is an aggregate of influences for several subgoals. For this reason, it was decided to split the empirical evaluation into two parts. First, the two main parts of the approach, *Target Approximation* and *Damage Avoidance*, are evaluated separately. This yields suitable similarity thresholds for both parts that can then be integrated for an evaluation of the overall navigation component.

For each step, suitable scenarios in the StarCraft RTS game were created. The first capability that is evaluated is *Damage Avoidance*. The test scenario for *Damage Avoidance* contains a large number of enemy units that are arbitrarily spread across a map. The scenario contains

Chapter 7. Architecture Level Three: Unit Pathfinding using Hybrid CBR/RL

both mobile and stationary opponents, essentially creating a dynamic maze that the agent has to navigate through. The agent’s aim is to stay alive as long as possible.

In order to test the agent’s ability to find the target position, it controls a unit in a limited-time scenario with no enemies and randomly generated target positions. The core performance metric here is the number of target positions an agent can reach within the given time. Accessibility penalties, *Unit Type* and *Previous Action* computation are part of both scenarios.

Based on initial test runs and experience gained in the optimisation in Chapter 5, a learning rate $\alpha = 0.1$ and a discount rate $\gamma = 0.8$ for the Q-learning algorithm are used. α and γ are slightly lower than the previous settings from the optimised parameters to produce a slightly slower, consistent learning process. The agent uses a declining ϵ -greedy exploration policy that starts with $\epsilon = 0.9$. This means that the agent initially selects actions 90% random and 10% greedy, i.e. choosing the best known action. The exploration rate declines linearly towards 0 over the course of the 1,000 games for the *Damage Avoidance* scenario and 100 games for *Target Approximation* respectively. After each full run of 1,000 or 100 games, there are another 30 games where the agent uses only greedy selection. The performance in this final phase therefore showcases the best policy the agent learned. The difference in the length of experiments (1,000 vs 100 games) is due to the vastly different state-space sizes between the two scenarios. When looking at *Target Approximation*, the target can be any one of the fields in a unit’s local IM, i.e. one of 49 fields. As a result, there can be a maximum of 49 different states even with $\psi_{approx} = 100\%$ similarity threshold. On the other hand, there are n^{49} possible states based on the damage potential of units, where n is the number of different damage values. This also illustrates why the generalisation that CBR provides during case retrieval is essential to enable any learning effect at all. Without CBR, the state-action space coverage would remain negligibly small even for low n -values. Table 7.2 sums up the evaluation parameters.

Parameter	Values
Scenario	<i>Damage Avoidance (A1)</i> , <i>Target Approximation (A2)</i>
Number of Games	1000(A1), 100(A2)
Algorithm	One-Step Q-learning
Damage Avoidance Similarity Threshold ψ_{dam}	35% - 75%
Target Approximation Similarity Threshold ψ_{approx}	55% - 95%
Overall Navigation Similarity Threshold ψ_{nav}	80%
RL Learning Rate α	0.1
RL Discount Factor γ	0.8
RL Exploration Rate ϵ	0.9 - 0

Table 7.2: Navigation Evaluation Parameters

Chapter 7. Architecture Level Three: Unit Pathfinding using Hybrid CBR/RL

The results for both variants are shown in Figure 7.5 and Figure 7.6.

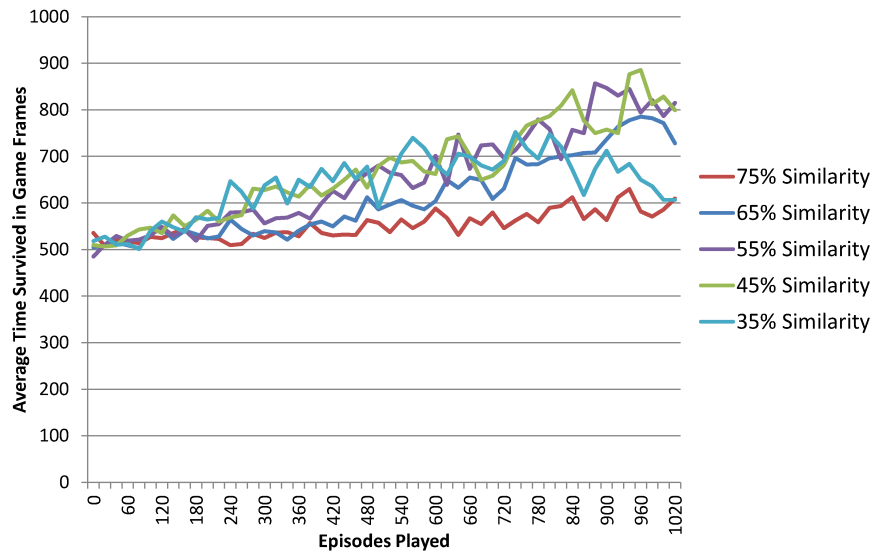


Figure 7.5: Results for the Damage Avoidance Scenario

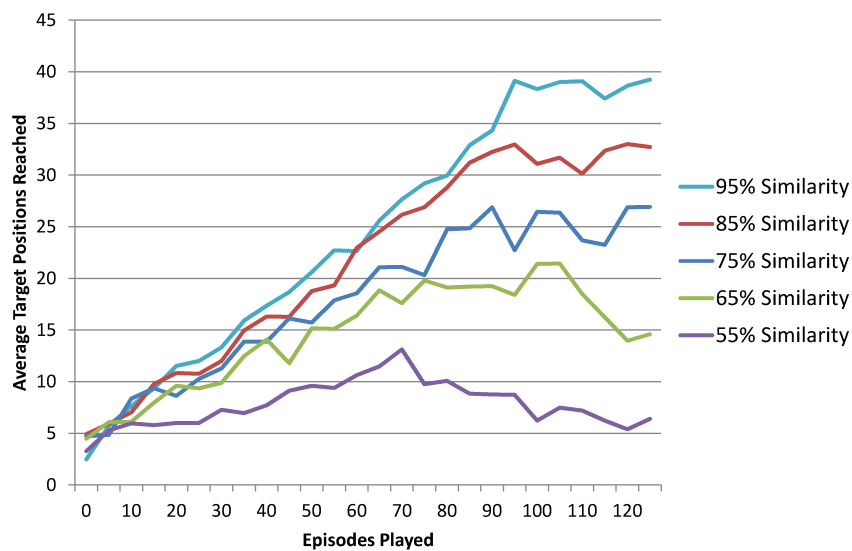


Figure 7.6: Results for the Target Approximation Scenario

The results from the two sub-problem evaluation scenarios were used as a guideline to decide on a similarity threshold $\psi_{nav} = 80\%$ to evaluate the overall algorithm (see Section 7.4 for more details). Both subgoals were equally weighted. The scenario for the evaluation of the integrated approach is similar to that used for the *Damage Avoidance* sub-goal. The target metric is no longer the survival time, however, but the number of randomly-generated positions the agent can reach before it is destroyed. The run length for the experiment is increased to 1,600 games to account for the heightened complexity. The resulting performance in the overall scenario can be seen in Figure 7.7. The figure also shows a random action selection baseline, as well as the outcome for using only *Target Approximation* without any attempt to avoid damage.

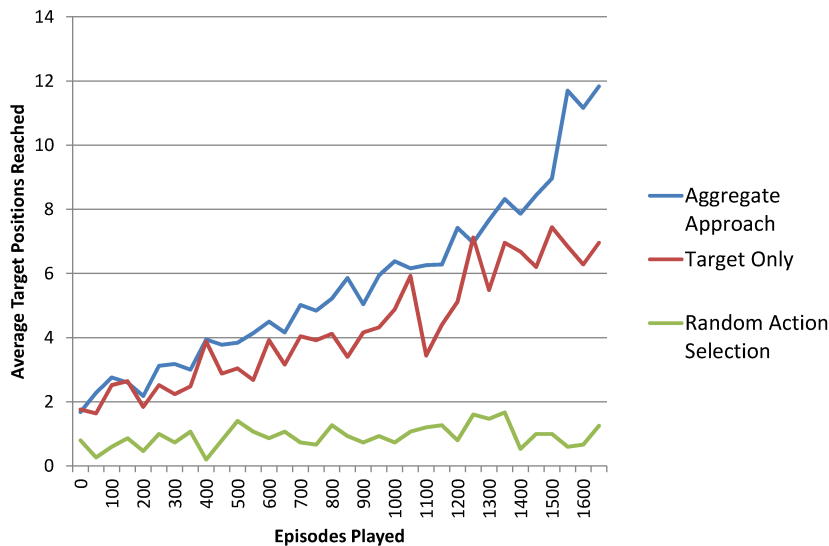


Figure 7.7: Results for the Combined Navigation Scenario

7.4 Navigation Discussion

The evaluation for the *Damage Avoidance* subgoal indicates that there is only a narrow window in which learning an effective solution is actually achieved in the given 1,000 episodes: With a similarity threshold of $\psi_{dam} = 35\%$ there is not enough differentiation between inherently different cases/situations and the performance drops significantly towards the end. At the other end of the spectrum, the worst results with practically no improvement in the overall score is achieved for a similarity threshold $\psi_{dam} = 75\%$. This means that with such a high threshold, the state-action space coverage drops to a level where learning a

‘good’ strategy is no longer guaranteed: For a similarity threshold of 75% there are about 7,000 cases in the case-base after 1,000 episodes played and less than 1/2 of all state-action pairs have been explored at least once.

Target Approximation is slightly different, in that an effective solution is achieved for any similarity threshold $\psi_{approx} \geq 65\%$. This is because even at 95% similarity there are still only $49 * 4 = 196$ state-action pairs to explore. The number of cases in the case-base varies, however. At 95% similarity threshold, there are 49 different cases, i.e. the maximum possible. At 85% similarity there exist about 20 different cases and at 75% only about 9 cases. This is important when combining target approximation with damage avoidance for the overall approach since this also combines the state-space of the two subgoals. Therefore, twice as many cases for only a $\sim 20\%$ higher reward as for the 75% compared to the 85% threshold or the 85% compared to the 95% threshold is a bad trade-off. For this reason, it was decided to use a 75% similarity threshold for the *Target Approximation* subgoal.

Both subgoal approaches show a performance improvement in their chosen metrics over the run of their experiments. Since the *Target Approximation* algorithm manages to exhaustively explore the state-action space, the performance improvement is more visible in this scenario. This also means that the best possible solution in this scenario has been found, as a 95% similarity threshold with all possible states and a fully-explored state-action space guarantee that the optimal policy π^* has been found. For the *Damage Avoidance* scenario on the other hand, there is still a potential for improvement, since significant parts of the state-action space remain unexplored: Even for a 55% threshold, only about $\frac{3}{4}$ of all possible actions are executed at least once.

The results for the overall algorithm in Figure 7.7 show that the agent manages to successfully learn how to navigate towards target positions. The performance of the agent far outperforms the random action selection. However, the results also show that the combined algorithm initially only has a small advantage over the *Target Approximation*-only agent and only towards the end performs better. This is far less an improvement than expected, especially given the fact that these experiments reached a sufficiently high state-action coverage of about $\frac{2}{3}$. This indicates that the chosen overall similarity threshold of $\psi_{overall} = 80\%$ is too low and there is not enough differentiation between inherently different cases/situations to find the optimal policy.

In general, the results show that the important information was identified and the model that was designed encompasses the different influences that are important for navigation and pathfinding in RTS games. The empirical evaluation showed the approach successfully learnt how to achieve partial goals of the navigation problem. Integrating the findings from the partial evaluation and running an evaluation for the entire approach including all relevant

influences and goals worked well. This overall evaluation shows that the model and approach manage to successfully learn how to navigate a complex game scenario that tests all different sub-goals.

7.5 Training the Navigation Case-Base

The final step is the preparation of the *Navigation* module for re-use by higher-level components. This means that the navigation module must contain knowledge in the shape of cases that can address any potentially arising game situations. Given the attributes that make up a case description (see Table 7.1), scenarios based on different static map layouts (cliffs, walls etc.) have to be learned as well as scenarios that contain different opposing unit types. Other aspects such as opponent and agent IMs as well as previous actions and target positions are varied automatically within a single scenario throughout the learning process.

Using the insights gained in the initial evaluation, the similarity threshold $\psi_{overall}$ is raised from 80% to 85% in order to achieve more consistent results while keeping the increase in training time reasonable. To account for the additional effort to find the best policy π^* , the number of training episodes is increased from 1,600 games to 2,500 games. Three additional scenarios were created in order to cover a maximum of possible cases. Table 7.3 lists the scenarios, their layout in terms of impassable terrain as well as the unit numbers and types involved. Each scenario is run with a number of opponent unit configurations, mainly grouped by the different attack types. The agent starts off with five units from each category and subsequently loses one after the other as they are eliminated by opponents: thus experimenting also covered different agent IM values.

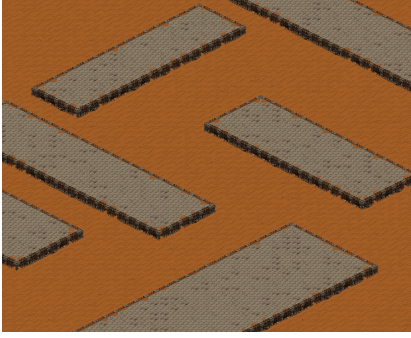

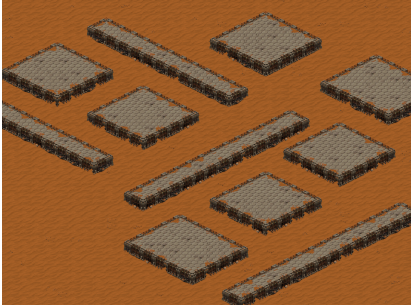
Scenario Map	Opponent Unit Types and Numbers
	<p>2 Melee and 2 Ranged 3 Melee and 3 Ranged 5 Melee and 5 Ranged 3 Melee and 0 Ranged 0 Melee and 3 Ranged</p>
	<p>2 Melee and 2 Ranged 3 Melee and 3 Ranged 5 Melee and 5 Ranged 3 Melee and 0 Ranged 0 Melee and 3 Ranged</p>
	<p>2 Melee and 2 Ranged 3 Melee and 3 Ranged 5 Melee and 5 Ranged 3 Melee and 0 Ranged 0 Melee and 3 Ranged</p>

Table 7.3: Additional Pathfinding Training Scenarios

After running each scenario in each configuration for five times, the training is complete and the resulting case-base is ‘locked’, i.e. no further changes are made during reuse by higher-level modules. This is according to the training procedure described in Section 6.2. The final case-base contains 11,394 cases. Of the resulting 45,576 state-action pairs, 73.6% have been explored repeatedly during the evaluation and training.

Subsequent higher-level components use this pre-trained *Navigation* case-base for the pathfinding of the units they control. The next chapter describes the creation of the relevant modules on *Level Two* of the hierarchical architecture.

Architecture Level Two: Squad-Level Coordination

This chapter describes the modules of the second level of the hierarchical architecture introduced in Chapter 6. Having created, evaluated and successfully trained a navigation component for *Level Three* of the hierarchical architecture, the next step is to create the modules that manage squads of combat units. This chapter includes the creation of the two main modules on *Level Two*, *Formation* and *Attack*, as well as a shorter section explaining the deterministic *Retreat* functionality.

The squad-based tasks that the model in Section 6.1 puts on this level are commonly classified as *tactical* in the literature that examines the structure of tasks involved in RTS game environments (Weber, Mawhorter, et al., 2010; Safadi & Ernst, 2010). Ontañón et al. (2013) examined StarCraft bot architectures as well as the general underlying RTS tasks involved and define *Tactics* as

[...] the implementation of the current strategy. It implies army and building positioning, movements, timing, and so on. Tactics concerns a group of units.

Both *Level Two* and *Level One* fall into this definition of *Tactics*. The modules on *Level Two* define how actions that coordinate groups of units are executed while *Level One* defines how units are distributed among these squad-based modules.

These modules are created for the particular tasks of *Attack* and *Formation*. The model created in Section 6.1 defined *Retreat* as the third possible action on *Level Two*, however *Retreat* only affects single units and thus does not require any reasoning or case-base for inter-unit coordination. The *Retreat* action is thus only important as a possible action to assign units to during the unit allocations that happen on *Level One* of the architecture.

The following sections explain the components that handle unit formations and unit attacks in detail. Each of the CBR/RL modules is developed in a similar process as the one that is used for the development of the hybrid CBR/RL agent described in Chapter 5. The considerations behind the *Formation* and *Attack* actions are shown, as are the RL model and

the architecture of the CBR components for these parts. Finally, the modules are evaluated in a number of scenarios. Following the example of the *Navigation* component described in the previous chapter, the case-bases for *Formation* and *Attack* are subsequently trained through a number of test scenarios in order to be re-used without modification by the tactical reasoner on *Level One*.

8.1 Unit Formations

Tactical formations are an important component in RTS games, which often resemble a form of military simulator and are heavily inspired by real-life combat strategy and tactics. Numerous tactical formations have been used throughout the phases of human history and the ensuing military conflicts, from ancient Greek phalanx formations to formations employed by tribes such as the New Zealand Maori (Gudgeon, 1907). Many historic formations such as a shield wall or a wedge are still used in today's military operations (Rabin, 2002). Since the importance of the placement of units relative to each other in military combat is very significant, *Formation* was identified as one of the three possible categories on *Level Two* of the hierarchical architecture presented in this chapter.

There are two possible ways in which units can coordinate their movements or, in fact, their overall actions. Individual units can choose actions or movements while taking into account actions or movements of other surrounding units. In terms of movement, *flocking* (Reynolds, 1987) is a behaviour that is based on this principle of autonomous agents making individual decisions that complemented each other to produce seemingly coordinated behaviour. The approach described in Chapter 5 falls within this category since it includes several agent-controlled units that act autonomously while taking into account the presence of allied units through the use of IMs.

The second way to coordinate unit movement is by deciding at a higher level that a certain set of units should move together, also called *Formation Motion* (Millington & Funge, 2009). *Formation Motions* describe the movement of a group of units or characters so that they retain some group organization. In terms of games, besides being used in RTS games, formation motion is also used in sports and driving games as well as FPS games. Formations also play an important role in other research areas such as robotics (Balch & Arkin, 1998).

Unit formations are generally concerned with the placement of units relative to a number of environmental influences. The most important of these influences are:

- Allied Units;
- Opposing Units, and
- Environmental Elements.

These influences that play a role for unit formations are also crucial for the general movement of units as shown in the model of the *Navigation* component in Section 7.1. However, if a high-level reasoner is used to coordinate a number of units, it is much easier for that agent to create an organized pattern of movement with a single goal for all units.

8.1.1 Unit Formations in StarCraft

Given the importance and diversity of unit formations as described above, unit placement in formations plays a major role both in RTS games in general and in a combat-oriented game like StarCraft in particular. The built-in AI provides very little support in this regard and thus leaves it to the players themselves to choose unit formations.

There are two general types of unit formations. On the one hand, there are **fixed formations** that use pre-defined geometric shapes like wedges, circles or squares which often take inspiration from historic military formations. Figure 8.1 displays a selection of common formation shapes. Fixed formations use pre-defined patterns that available units are slotted into. On the other hand, there are **emergent formations**, that are defined freely and are supposed to adapt unit positioning within a formation according to the current game situation (Lin & Ting, 2011). **Dynamic formations** are a combination of fixed and emergent formations (Van Der Heijden et al., 2008). They use predefined shapes, but adapt certain parameters about such shapes to dynamically adjust the formation to the situation (see Figure 8.2).

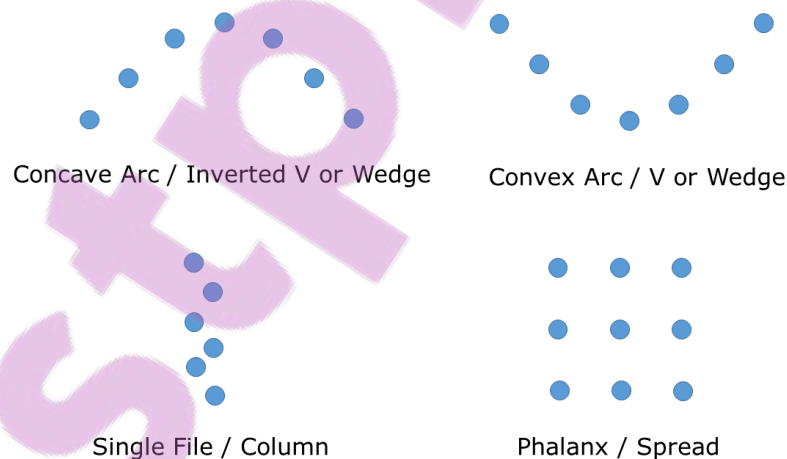


Figure 8.1: Fixed Formation Example

The most flexible, adaptable and thus the most desirable solution for the *Formation* action in the hierarchical architecture would be to use fully emergent formations that adapt to current situations. However, this type of formation creation requires a large computational

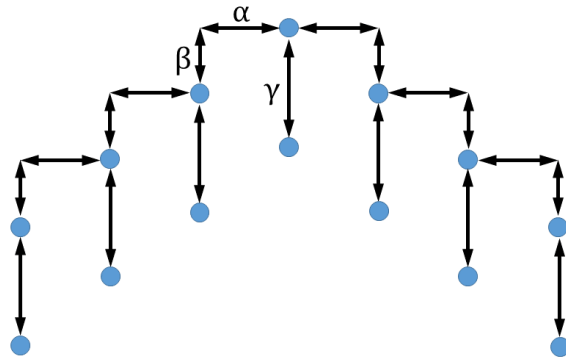


Figure 8.2: Dynamic Formation Example

effort and does not always guarantee the creation of optimal solutions (Lin & Ting, 2011). Furthermore, using this approach leads to a number of additional parameters that have to be optimized for a problem that is already complex. Adding more parameters to the problem descriptions thus further complicates the use of CBR for the retention of game states.

However, dynamic formations are also very powerful, depending on the number of parameters assigned to a certain formation. Since fully fixed formations on the other hand are inflexible, it was decided to use a reduced-complexity variant of dynamic formations. In this variant, the general shape of the formation is fixed, certain parameters are automatically adapted to the game state, and a third set of parameters is flexible and learned through CBR/RL. Figure 8.3 shows the general layout of the formations created in this way.

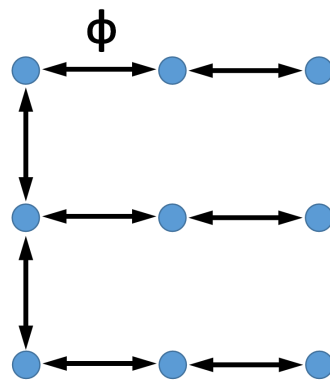


Figure 8.3: Agent Formation Layout and Parametrisation

As the figure shows, the formation layout is based on a simplified dynamic formation. The main distance parameter ϕ defines the distance between units both in horizontal and in vertical direction. The general layout is a phalanx or square, with the length of the sides s being dependent on the number of units n that are part of the formation:

$$s = \lceil \sqrt{n} \rceil.$$

ϕ is not part of case descriptions or case solutions and is not learned. Instead, it is directly tied to the range of *splash damage* of the opposing units that are in the immediate vicinity of the formation units. Units that deal this type of damage do not just deal damage to a single opponent but instead to an entire area of the map. ϕ is set to a value that is large enough to ensure no more than a single agent unit can be hit by splash damage effects. If there is no splash damage at all, ϕ is set to a minimum distance to ensure units do not get stuck to each other.

The main values to be learned through CBR/RL are the unit-slot associations, i.e. which slot in the formation a certain unit is assigned to. Theoretically, all possible unit-slot permutations can be solutions. This means that for n units and n slots, there are $n!$ possible unit-slot permutations and therefore possible solutions. (See Table 8.1)

Unit Number n	Solution Number $n!$
1	1
2	2
3	6
4	24
5	120
6	720
...	...

Table 8.1: Number of Possible Solutions for Assigning n Units to n Formation Slots

The resulting large number of solutions shows that even for medium-sized unit squads that should be arranged into a formation, there are too many solutions to effectively evaluate them all repeatedly and find the best one possible. On the other hand, repeatedly testing all possible solutions is a requirement for RL to find the best policy to solve a given problem. In theory there is a requirement to visit each possible solution infinitely often to guarantee convergence to the optimal policy π^* . Additionally, each solution can exist for each possible state, i.e. the number of possible solutions has to be multiplied by the number of overall cases in the case-base. If, for example, there are 100 cases for formations with six units in the case-base, there would be 72,000 possible solutions that have to be repeatedly explored.

While the number of cases is controlled by an appropriate definition of the case descriptions and the similarity thresholds (see Section 8.1.3), it is also necessary to find a way to reduce the number of possible solutions, especially for larger numbers of units in formations. Therefore it was decided to use a *Solution Case-Base*. This case-base reduces the number of solutions that have to be examined by using CBR to generalise over this solution space.

8.1.2 Formation Solution Case-Base

This concept of using a separate case-base for case solutions was inspired by Molineaux et al. (2008) who used a solution case-base for their agent *CASSL (Continuous Action and Space Learner)* to generalise across a continuous action space. Their initial case-base has case descriptions for combat scenarios where groups of combat units have to be assigned movement actions in a continuous action space. The solutions to these initial states form cases for a second case-base that describes the actual movement in a continuous actions space defined by movement direction, movement distance, the groups' sizes as well as the type of group selection method.

The given problem here is simpler in that the solution space is not continuous but a finite set of unit-slot permutations. The generalisation technique is based on reusing existing solutions if a sufficiently similar solution has already been evaluated and retained in the solution case-base. The similarity between solution cases is defined as the sum of similarities between individual slots.

Slot similarity in turn is based on two attributes.

XY Similarity How close is the slot to the original spot in terms of x - and y -coordinates?

Anything with the same x - **and** y -coordinates has *XY Similarity* 1; anything which is removed by one row or column has *XY Similarity* 0.5 and anything else has *XY Similarity* 0.

Border Similarity A factor of major importance for a position in a formation is the proximity of that particular position to the edge of the formation. This results from positions at the outer edges being attacked first. The similarity measure compares the number of outer edges two different slots have (between zero and four) and assigns a similarity accordingly (between 0 and 1 in 0.25 intervals).

Figure 8.4 shows an example comparison between two formation solutions for 5 units based on these similarity metrics.

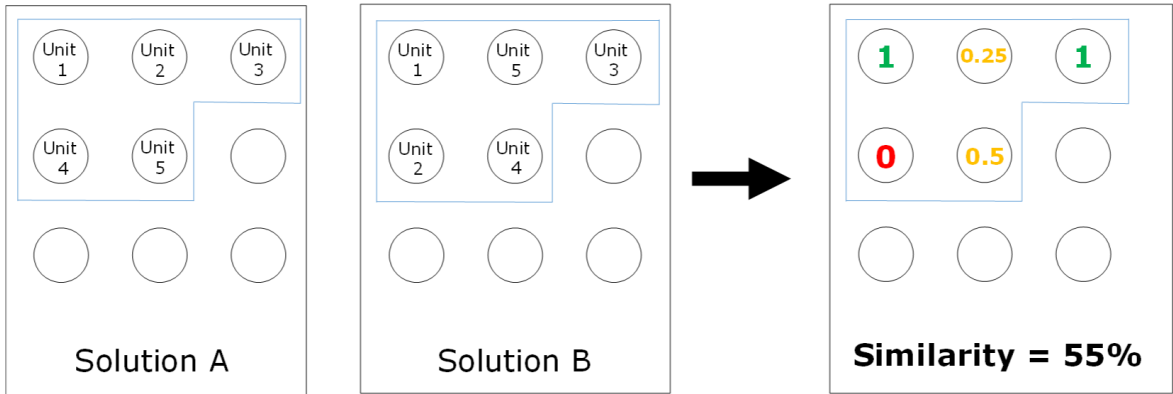


Figure 8.4: Formation Solution Similarity Example

As the diagram shows, the five units result in a 3×3 formation layout. The overall similarity between solutions A and B is computed as the sum of individual slot assignments, with each slot contributing equally.

The similarity between two solutions does not depend on the game state, but always remains the same for these two sets of unit-slot associations. Therefore, it is possible to compute these similarities offline and simply retrieve them once a certain solution is used. The offline computation requires a considerable effort, since for a given number of units each possible solution has to be compared to each other possible solution. As an indication, Table 8.2 shows the resulting numbers of similarities for formations including up to seven units that were tested within the scope of this thesis.

Unit Number n	Solution Number $n!$	Solution Similarities $n! \cdot (n! - 1)$
1	1	0
2	2	2
3	6	30
4	24	552
5	120	14,280
6	720	515,680
7	5040	25,396,560
...

Table 8.2: Solution Similarities

While the overall number of similarity values that are computed for a certain number of units is very large, for any one solution for n units that is selected, only $n! - 1$ entries have to be retrieved and checked. If a solution sufficiently similar to the chosen solution, given a similarity threshold ψ , has already been used, that solution can be re-used. If this is not

the case, a new solution entry is created in the case-base and that solution is executed. The evaluation of how well the solution case-base reduces the number of stored solutions is part of the overall performance evaluation of the *Formation* actions in Section 8.1.4.

8.1.3 Unit Formation Model

Similar to other problems described throughout this thesis, the *Formation* problem is expressed in terms of a MDP (see Section 3.3) to enable RL to learn how to use formations effectively. However, unit formations are inherently indirect in their merits in terms of RTS game successes. *Attack* actions result in a direct change in opponent unit health and potentially opponent unit numbers while *Retreat* actions serve to simply maintain the status quo and keep units alive. *Formation* actions, on the other hand, are supposed to prepare the scenario for an improved performance in those other, subsequent actions. This means that the problem of delayed reward that is inherent in TD RL algorithms is further exacerbated. Another aspect that contributes to this effect is the requirement to individually evaluate the performance of the *Formation* action as well as those of the other actions. Evaluating a formation individually would mean that there can be no other actions whose reward can trickle down and be at least partially attributed to the *Formation* that made them possible.

For these reasons, the RL model has to be carefully designed with a specific desired outcome and features of the performance in mind. Subsequently, an evaluation of the performance of the agent in terms of these desired features can be run. When putting units into a formation, the ‘quality’ of this formation has to be defined, both in terms of being part of a larger tactical task and on its own.

After considering the desired effects of formations, two main criteria were chosen as a benchmark for the performance of a unit formation resulting from two aims that are central to the performance of a unit formation. The first aim, once units have been assigned to formation slots, is to create the eventual formation quickly. This is even more important in the context of the overall hierarchical architecture, where higher level reasoners wait for lower level actions to be finished. This means that no further high-level tactical decisions can be made until lower-level *Formation*, *Attack* and *Retreat* actions are all finished.

The second aim is to maximise the potential of units that make up a formation. In terms of combat units in an RTS game such as StarCraft, this means that the potential damage that units can effect should be maximised over time. For example, this means that melee units are kept towards the outside of a formation where they can still deal damage, while ranged units can just as well deal damage while acting behind other allied units in the formation.

The first criterion is straightforward, both in terms of how it can be integrated into the model and in terms of how to evaluate it. The second criterion is more complex, which makes it more difficult to create a model that allows the agent to learn this criterion and to

evaluate successful performance. This difficulty reflects the difficulty of evaluating the merits of *Formation* actions through RL as been mentioned above.

Formation States

Formation states are descriptions of cases stored in the formation case-base. These descriptions include all relevant information at the point in time when a new *Formation* action is to be decided.

Category	Attribute	Type
Index	Units Agent	Integer
Unit	Type	Enum
	Health	Integer
	Position	Integer
Opponent	Attacking Damage 1-8	Integer

Table 8.3: Formation State Case Description

Table 8.3 lists the attributes that make up a formation state description, ordered by categories. The initial index is the number of units that are assigned to a formation. Since solutions are strict unit-to-slot assignments with no contingency for missing or additional units, there is no reuse between cases for different numbers of units.

Unit entries in terms of the *Formation* action are defined by their position relative to the centre of the future formation and their unit type, as well as their health as divided into one of four intervals. The centre of the future formation is computed as the centre of the polygon that all formation units form. In order to abstract the unit positions into a format that is easier to compare, the map is split into four separate areas with the polygon centre forming the centre. Each unit is then assigned a quadrant 1 – 4 it falls into (see Figure 8.5).

Opponents are abstracted in a similar way. Each opponent is assigned to one of eight different directions/slices, relative to the formation centre. For each opponent, its damage potential is added to the slice it falls into. All information on opponent units that is stored for a certain formation case are the values for those eight slices. This level of abstraction is a trade-off between recording as much information as possible about opponent influence and abstracting available information to the point where the number of cases is kept to a reasonable value. The reasoning behind this particular abstraction is that the two most important aspects of an opponent in terms of formations are the direction from which it comes and the damage it can deal.

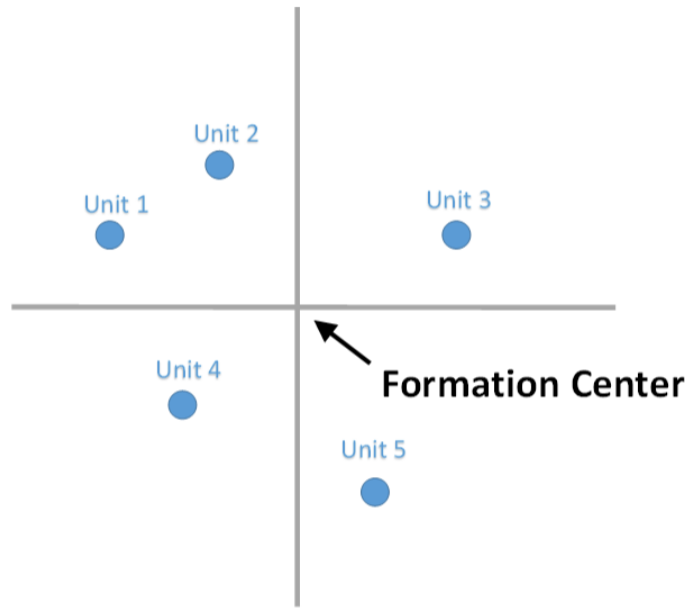


Figure 8.5: Formation Unit Positions

Formation Actions

The case solutions for formation cases are, as mentioned above, unit-to-slot assignments. Table 8.4 lists some example solutions for $n = 5$ units.

	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5
Solution A	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5
Solution B	<i>Slot 2</i>	<i>Slot 1</i>	Slot 3	Slot 4	Slot 5
Solution C	Slot 2	<i>Slot 3</i>	<i>Slot 1</i>	Slot 4	Slot 5
Solution D	Slot 2	Slot 3	<i>Slot 4</i>	<i>Slot 1</i>	Slot 5
...

Table 8.4: Example Formation Slot Assignments

As described in Section 8.1.2, not every solution will be used for each case. If a sufficiently similar solution, for a given minimum similarity threshold ψ , has been attempted before, that solution is used instead. The evaluation in Section 8.1.4 illustrates the resulting reduction in solution space.

Formation State Transition Probabilities

Given the high level of abstraction, especially in terms of unit health and positions as well as the abstraction of opponent influence, transition probabilities are stochastic. This means that executing the solution a_i in the same initial state s_t at two different times can lead to two different states s_{t+1} .

Formation Reward

Similar to the reward signal that was used for unit navigation in Chapter 7, the reward signal for the *Formation* action consists of two different influences. The RL reward signal is built to entice the agent to learn how to fulfil the two main aims that have been identified in terms of unit formations: Enter a formation quickly and maximize the damage potential of the units that create a formation.

The component of the reward signal that reflects the time taken to get into position t_{form} is negative. On the other hand, there is a positive reward component d_{avg} for the average damage that can be delivered to all opposing units once a formation has been created. This part is computed for every frame of the evaluation and adds up the damage that can be dealt to any opposing unit at that particular time. These values are then added up and averaged over the number of frames. Both the negative and the positive component are normalized. Since using a unit's full potential is more important than how quickly the units can get into position, the eventual overall reward r_{form} is weighted.

$$r_{form} = 1.5 * d_{avg} - t_{form}.$$

Defining the reward signal in this particular fashion means that the agent is trying to optimize the attack range for all controlled units to cover a maximum number of opponents. Implied is also a penalty for any unit lost while being put into position, since that unit's damage potential is no longer part of the reward signal. This implied penalty should result in formations that keep high-damage ranged units in positions where they can cover lots of ground, while being shielded by low-damage and melee units that have higher resilience.

As a result, the agent is compelled to find formations into which units can move easily and at the same time prevent those units from taking unnecessary damage.

8.1.4 Formation Evaluation and Training

The evaluation of the *Formation* task is more complex than the evaluation of previous agent implementations and the evaluation of the *Attack* task due to the more indirect merits of formations. Formations serve the purpose of getting units into positions that allow them to perform better at other tasks such as *Attack* and *Retreat*. This means that units manage to stay alive longer and deliver more damage if they successfully apply formations. As explained in the previous section, the two core performance indices of successfully using formations are

- time taken to form the formation, and
- damage potential that can be delivered at any one point once the formation has been formed.

Chapter 8. Architecture Level Two: Squad-Level Coordination

In previous empirical evaluations, the agent performed in a number of combat scenarios where its goal was to eliminate opponents (Chapters 4 and 5) or to reach certain positions while under attack (Chapter 7). The variations in model parameters that are required to ensure that the agent does not simply learn a specific scenario but instead the overall task were created through the interaction between agent unit(s) and opponents as well as through free movement.

Learning formations is considerably harder. The first problem occurs in changing from one formation state to another. Once a certain formation has been reached, learning about the effects of a different formation is not easy, especially when the formation is under attack. StarCraft uses collision control, i.e. walking through other units is not possible. If only formations are evaluated - which is the case for this evaluation - opposing units will strictly focus their attack on formation units only and introduce a large amount of noise into the learning process since their irregular movement patterns are not attributed for in the RL model defined in Section 8.1.3. Having this non-reproducible effect as part of an evaluation is a major hindrance for effective RL.

Therefore, since transitioning between formations during the learning phase is not possible without introducing noise, it was decided to run scenarios that evaluate a single formation per game/episode. Subsequently, the knowledge gained in this way can be re-used (though not modified) at the higher, tactical level (see Chapter 9) in traditional combat scenarios.

There are two immediate effects of this learning strategy. Since only one possible solution is explored per game (compared to about 40 in the evaluation scenarios in Chapter 7 and even more in the evaluation scenarios in Chapters 4 and 5) the number of learning episodes that are required to find an optimal policy π^* is significantly larger than in those previous evaluations. Instead of 50 to 1,000 episodes, several thousand to tens of thousands of episodes are required. Furthermore, only one solution is picked at the beginning of a game and subsequently executed and evaluated before the scenario is restarted. If this is a standard game scenario like those used in previous evaluations, this would mean that the initial state is always the same, which in turn leads to the agent only being able to learn a single case over and over again. In order to acquire relevant knowledge for different game situations, a large number of different scenarios would have to be run. StarCraft has only limited functionality in terms of randomisation, but it is possible to create a scenario that randomly places opposing units on the map (see Figure 8.6). This scenario enables the agent to explore all possible settings of agent attack patterns. In order to explore different numbers of agent units, more than one evaluation scenario is still required.

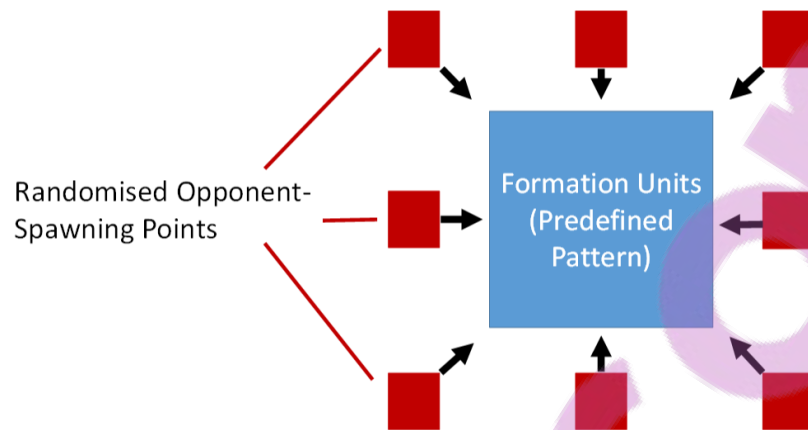


Figure 8.6: Randomised Formation Scenario

Apart from varying attack patterns and different agent unit placements and agent unit types, the last attribute that should be explored is the handling of injured units. Injured units are similar to ranged units in that they should be protected from opponent attacks. However, due to their low health they are even more fragile and therefore should take precedence over simple ranged units.

All of these factors were considered for the creation of the evaluation scenarios. To account for all of these factors, four different types of scenario are evaluated. *Scenario A* is used to evaluate the effectiveness of the model in terms of enabling the agent to learn how to reduce the time it takes to move into formation. This is a very straightforward task, since the agent would simply have to assign each unit to the geographically closest slot in the formation. Therefore, the scenario was merely to evaluate whether the model represents the game in sufficient detail for this to be possible.

Scenario B trains and evaluates the agent's ability to protect ranged units from direct damage, thus enabling them to live longer while still being able to deal damage. Figure 8.7 shows both the units involved and an example of the behaviour the agent should ideally learn. The agent's units consist of five melee units and one ranged unit. As the diagram shows, for this particular scenario, the opposing units are spawned at five of the eight possible spawning points, leaving the bottom half blank. As a result, the ideal formation for the agent, i.e. the one that enables it to deal the most damage for the longest time, is to shield the ranged unit from all outside edges. This way, the ranged unit can deal damage to any opposing units that attack the fringe of the formation while being shielded by the agent's own melee units.

Scenario C is similar to *Scenario B*, except that the focus is on the placement of injured units instead of ranged ones. Both *Scenario B* and *Scenario C* only use the average damage that can be dealt (i.e. the second component in the composite reward signal) as their reward signal, thus ignoring the time it takes to create a formation. Figure 8.8 shows both the units involved and an example of the behaviour the agent should learn. The agent units consist

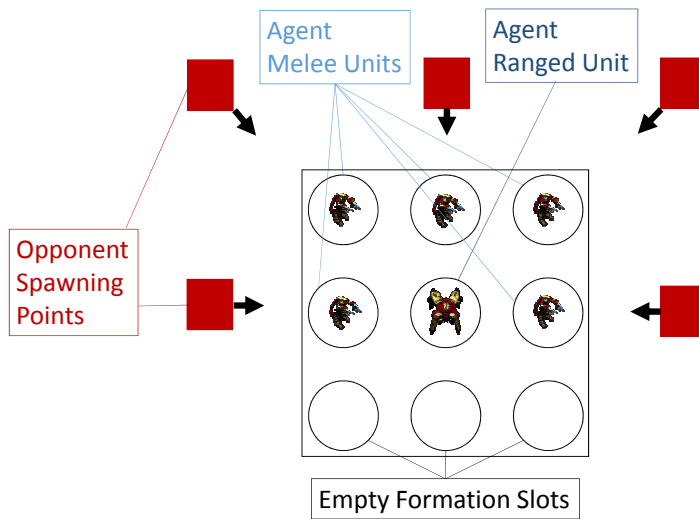


Figure 8.7: Example Desired Behaviour for Scenario B

entirely of ranged units. The health of a specific units is shown as a health bar beneath that unit. As the diagram shows, all but the single injured unit have full health. In this particular example, the opposing units were spawned at five of the eight possible spawning points, leaving the bottom half blank, similar to the example diagram for *Scenario B*. As a result, the ideal formation for the agent is also similar to that presented in Figure 8.7. For *Scenario C*, the ideal formation enables the agent to deal the most damage for the longest time by protecting the injured unit as long as possible. Therefore, the best spot for the injured unit is in the middle of the formation, shielded by the other, healthy units.

Finally, *Scenario D* combines all tasks from *Scenarios A-C* and also uses both components of the reward signal. Figure 8.9 shows both the units involved and an example of the behaviour the agent should learn. In this example of *Scenario D*, the agent controls three melee units and three ranged units. Two of the ranged units are injured. In the particular example shown, opposing units attack from left, right, top-left and top-right. This means that the ideal positions for the two injured ranged units are in the middle column.

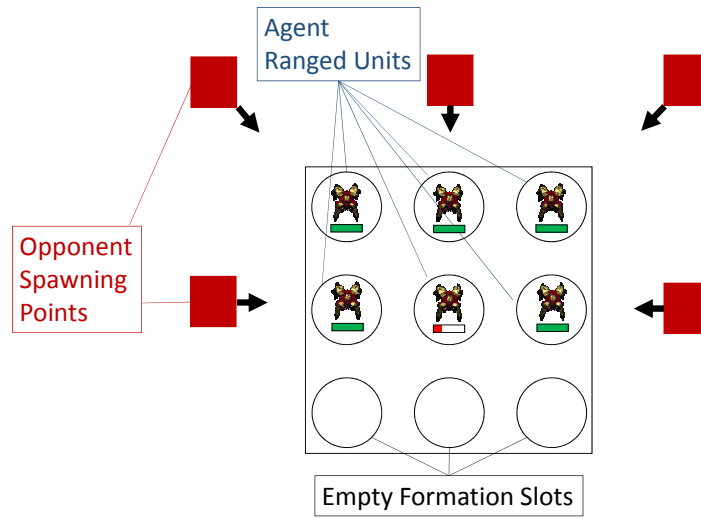


Figure 8.8: Example of Desired Behaviour in Scenario C

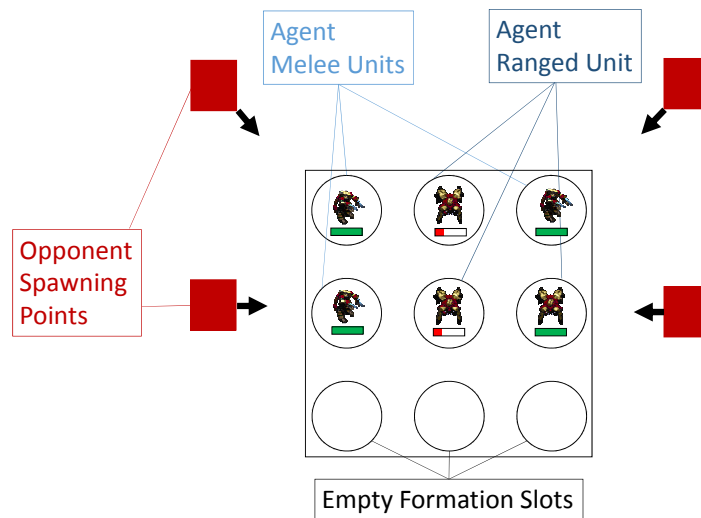


Figure 8.9: Example Desired Behavior for Scenario D

All three examples shown in Figures 8.7 to 8.9 represent ideal situations with at least one ideal solution. This means that there is an ideal slot for each of the ranged and/or injured units that are protected from direct attacks by opponents. However, in a typical game this

might not be the case. In 50% and more of all situations, opposing units spawn in a way that there are no or too few protected spots for all units that should be protected. As a result, it is to be expected that there is a wide variance in scores that the agent obtains in the evaluation. Some situations, like the shown examples, have a huge potential for improvement due to an existing ideal solution. In other situations, for example when opponents attack from all eight possible directions, there is little potential for improvement over a random action selection. This situation becomes even worse if the agent controls four or fewer units. Four or fewer units means the units are arranged in a 2×2 formation layout with no protected inner slot. Given this kind of layout, the agent can only learn how to place vulnerable units away from locations that spawn opponents.

Table 8.5 sums up the different scenarios and the sub-problems they evaluate.

Scenario	Unit Number	Ranged	Melee	Injured	Reward
A	6		X		Formation Time
B	6	X	X		Damage Potential
C	6	X		X	Damage Potential
D	2 - 6	X	X	X	Formation Time & Damage Potential

Table 8.5: Formation Evaluation and Training Scenarios

These are the main configuration parameters for the evaluation scenarios. Since the second task, besides evaluating the performance of the *Formation* component, is to gain knowledge that can subsequently be used by the *Tactical Unit Selection* task on *Level One*, it is also important to cover the entire possible formation case-space. In the context of these scenario settings, this meant that *Scenario D* was also run with lower numbers of units $2 \leq n \leq 6$.

The maximum number of units, 6, was selected as a compromise between the huge computational effort and storage required to pre-compute distribution similarities (see Section 8.1.2) and the requirement to have a sufficiently complex formation layout. Sufficiently complex in this case means, that the formation the units form has a core, potentially ranged or injured units that need protection, and an outer layer of units that shield the core. Given the square layout of the formations, the lowest possible number of units that would try to form a square with a core would be 5: Four or fewer units would try to form a 2×2 square where every unit is exposed to the outside, five or more units use a 3×3 layout, thus providing some kind of protection to certain units.

All test scenarios were run using a learning rate $\alpha = 0.1$ and a discount rate $\gamma = 0.8$ for the Q-learning algorithm, similar to the *Navigation* module evaluation. Furthermore, one-step Q-learning was again used as the RL algorithm that updates solution fitness.

The agent uses a declining ϵ -greedy exploration policy that starts with $\epsilon = 0.8$. This means that the agent initially selects actions 80% random and 20% greedy, i.e. choosing the best known action. The exploration rate declines linearly towards 0% over the course of the experiments. After the exploration rate has reached 0%, there is another number of games where the agent uses only greedy selection. The performance in this final phase thus showcases the best policy the agent could learn. Each experiment was run five times and the results were averaged over these runs. The length of each experiment was decided based on the observed coverage of the state-action space. If there were a large number of unexplored or infrequently explored actions, more episodes had to be run. Since only a single solution/action was explored during each episode, the number of episodes was larger than in previous evaluation scenarios. This is especially noticeable for *Scenario D*, which combines the reward signal for both the time taken to establish the formation and the average damage that can be dealt by a certain formation. *Scenarios A, B and C* only used single-component reward signals and learned in 2,000 to 3,000 episodes. *Scenario D* uses a composite reward signal made from both components and thus needs much longer to learn the trade-off between the two components. *Scenario D* requires 30,000 episodes to sufficiently explore the state-action space.

The similarity threshold ψ_{form} that decides when new cases in the *Formation* case-base are created was set to 75% after several initial experimental runs. The similarity threshold $\psi_{formSol}$ that decides when new cases in the *Formation solution* case-base are created is set to 80%. Table 8.6 sums up the parameters of the evaluation.

Parameter	Values
Scenario	A(6vs8), B(6vs8), C(6vs8), D(6vs8)
Number of Games	5,000 (A), 2,000 (B), 2,500 (C), 30,000 (D)
Algorithm	One-Step Q-learning
Formation Case-Base Similarity Threshold ψ_{form}	75%
Formation Solution Case-Base Similarity Threshold $\psi_{formSol}$	80%
RL Learning Rate α	0.1
RL Discount Factor γ	0.8
RL Exploration Rate ϵ	0.8 - 0

Table 8.6: Formation Evaluation Parameters

8.1.5 Formation Results

This section presents the results for the experimental evaluation combat scenarios listed in Table 8.6. Aside from the displayed results, additional training using *Scenario D* was run with fewer formation units $2 \leq n \leq 6$ in order to cover the entire case-space. Cases that can address any potentially arising situation are required for the evaluation of the *Tactical Unit Selection* component on *Level One* where no further changes are made to any of the *Level Two* knowledge bases. Each diagram in this section compares the CBR/RL agent with a random action selection baseline policy.

Figure 8.10 shows the results for *Scenario A*. This scenario requires the agent to form a formation with six melee units in the shortest possible time. The diagram shows that the average time a formation takes to form is reduced from the initial 75 frames, which is also the average for random action selection, to about 35 frames. This is a reduction of the required time of more than 50% over the course of 5,000 episodes. The most noticeable decrease happens in the middle part of the experiment, with more level initial and final phases.

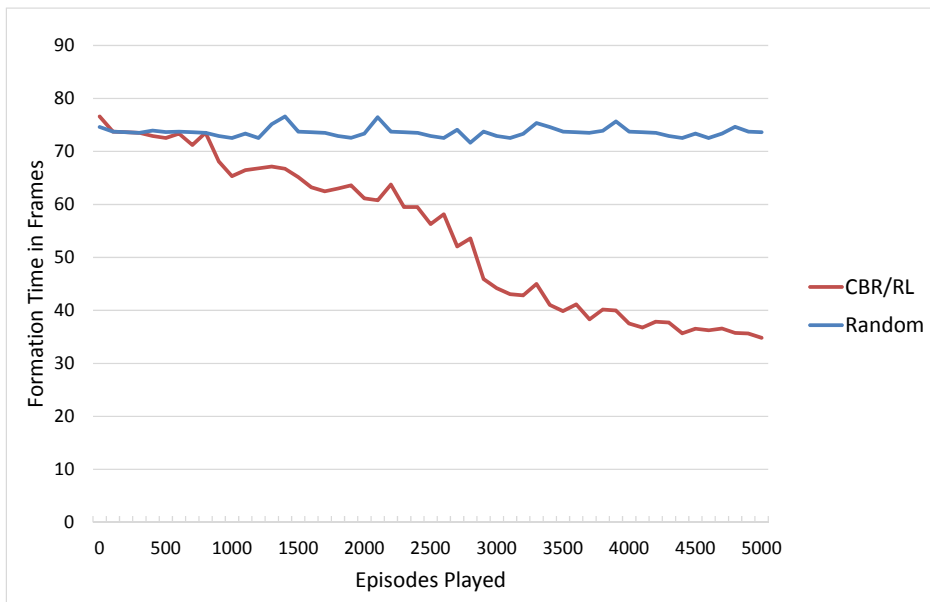


Figure 8.10: Results for Scenario A

Figure 8.11 shows the results for *Scenario B*. This scenario requires the agent to form a formation that is able to deal the highest possible average damage per frame to opposing units. The agent controls five melee units and one ranged unit. Over the course of 2,000 episodes, the average damage that can be dealt to opposing units increases from about 70

to about 90, an increase of roughly 30%. The increase is linear. The average damage for random action selection remains at the initial 70.

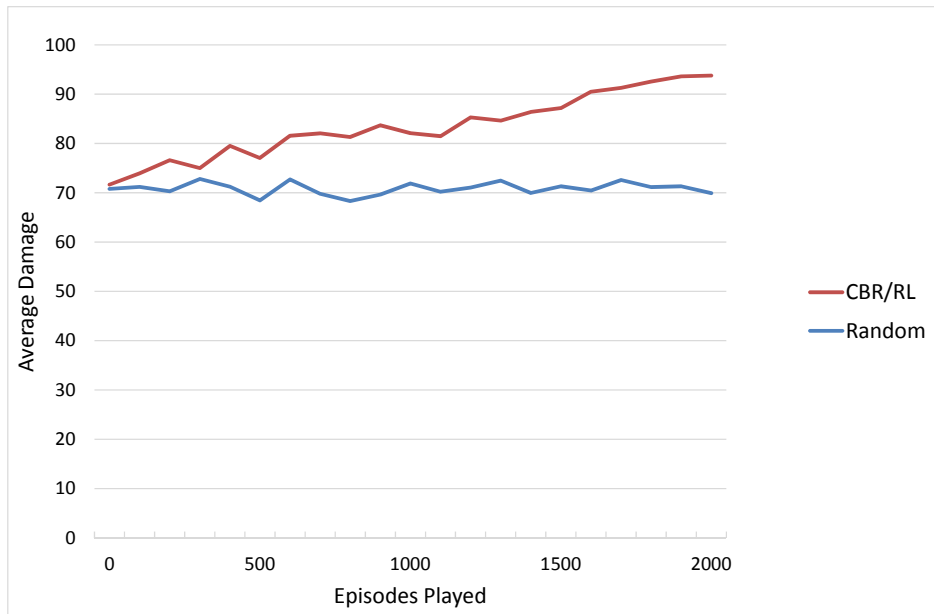


Figure 8.11: Results for Scenario B

Figure 8.12 shows the results for *Scenario C*. This scenario requires the agent to create a formation that is able to deal the highest possible average damage per frame to opposing units, similar to *Scenario B*. The agent controls six ranged units, one of them injured. Over the course of 2,500 episodes, average damage increases from about 65 to just below 90. The average damage per frame for random action selection is about 65. This increase of about 40% occurs linearly.

Figure 8.13 shows the results for *Scenario D*. This scenario requires the agent to create a formation in the fastest possible way. Additionally, the formation should also be able to deal the highest possible average damage per frame to opposing units, given the provided units. The agent controls six units, three melee units and three ranged units, with one of the ranged units injured. Over the course of 30,000 episodes, the average reward obtained increases from about 135 to 180. The reward signal is a combination of negative reward for the time used to get into position, and positive reward for the average damage per frame the agent units can deal. This increase of about 33% occurs linearly and plateaus towards the end.

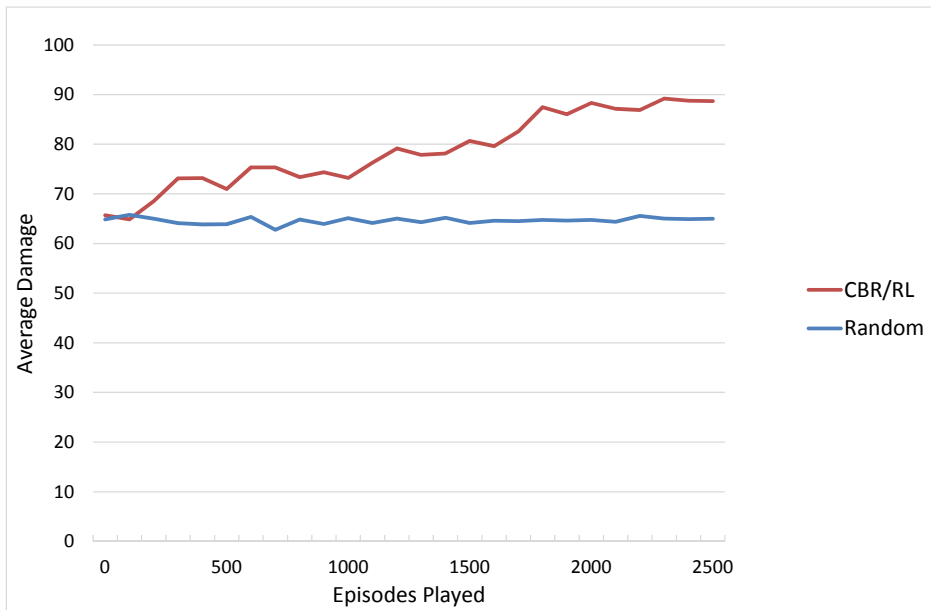


Figure 8.12: Results for Scenario C

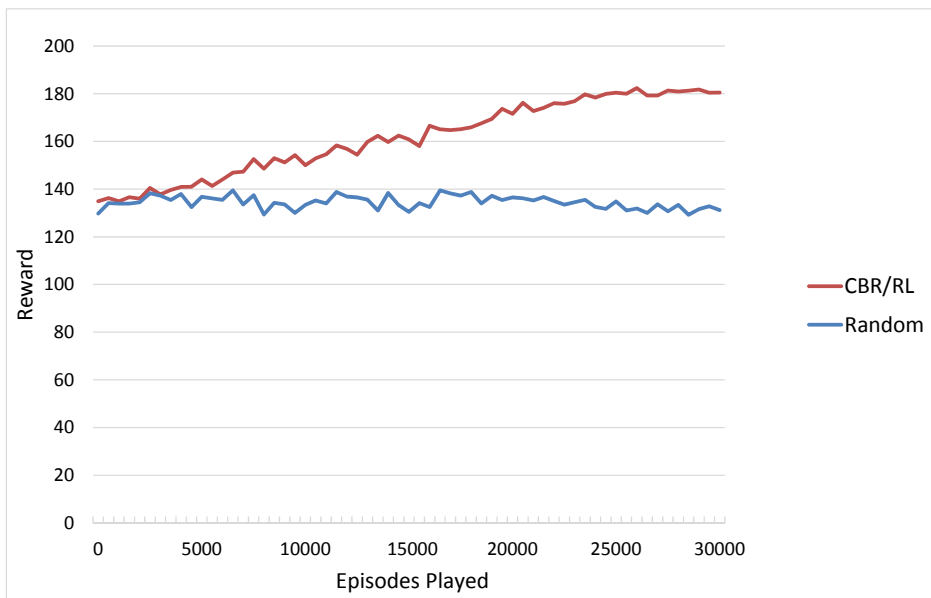


Figure 8.13: Results for Scenario D

8.1.6 Formation Discussion

As the results show, the agent manages to learn a good policy in each of the evaluation scenarios. Being able to find these policies indicates that the model in Section 8.1.3 represents the game world sufficiently well to enable the agent to learn the desired behaviour. A closer, in-depth examination of the optimal solutions for cases also shows that the agent manages to learn the ideal behaviours that are exhibited in the example diagrams in Figures 8.7 to 8.9 for *Scenarios B, C* and *D*.

The significant reduction in the average time that it takes for units to reach their ideal positions shows that this is the easiest task to learn. This was to be expected, since a hard-coded script that simply picks the closes slot for each unit could have achieved the same result. However, a script could not have been easily combined with other reward signal components to allow the agent to consider trade-offs among the different components, e.g. picking a spot further away if that leads to a much higher average damage over time. For this reason, it is also important to compare the results for *Scenarios A* and *D* where this trade-off should be learned through the composite reward signal.

The results for *Scenario B* in Figure 8.11 show that there is only a difference of about 33% between the random action selection policy and the average optimal solution that was found after 2,000 episodes. This can be explained by the previously mentioned observation (see Section 8.1.4) that while some situations have a huge potential for improvement, many other situations have only little or no potential for improvement over a random action selection. This leads to this rather low increase in average obtained reward when compared to previous experiments. Those experiments in previous chapters also use different models for their respective problems, therefore results are not directly comparable. The situations without an optimal solution have the effect of diluting the overall result. These ‘noise-cases’ lead to an overall average reward that is closer to the random action selection, as shown in the rewards diagrams. Initially, it was considered to filter cases that could not be decisively won. However, even for cases that describe these game situations there are usually solutions that lead to higher than random rewards. The increase might only be minimal, however. The only case with absolutely no possible improvement over a random action selection would be for a scenario where opponent units attack from every direction and there is no protected ‘core’ in a formation since there are too few units.

Scenario C produces similar results as *Scenario B*, both in terms of optimal policies that are found and in terms of the number of episodes required to reach these optimal solutions. Furthermore, the absolute amount of reward obtained is similar, even though *Scenario C* has a slightly lower starting point and also obtains slightly lower rewards towards the end. This can be explained through the injured unit that is part of *Scenario C* and which has a shorter survival time than its healthy equivalent in *Scenario B*.

Scenario D combines the reward signals for both the negative component for the time it takes to create the formation (in frames) and the positive component for the average damage the formation can deliver to attacking opponents at any one time. As a result, the learning process also takes considerably longer to evaluate the finer nuances of the resulting composite reward signal. Using the best learned policies, the agent manages to obtain an average reward that is 33% higher than that for random action selection. This is similar to the increase obtained for *Scenarios B* and *C* although less than that obtained for *Scenario A*. This shows that in the current composition of the reward signal where average reward is weighted at 150% and formation time at 100%, the effects of the average damage over time outweigh that of the time it takes to form a formation. Since the agent manages to learn the desired behaviour, this is considered acceptable.

8.1.7 Effects of Using a Solution Case-Base

As Table 8.1 shows, the number of possible slot-to-unit associations and thus the number of possible solutions per case grows exponentially and very quickly becomes too large to be explored within reasonable time even for small numbers of cases and units. For this reason, a solution case-base is used (see Section 8.1.2). The aim is, despite the use of this solution case-base and thus a generalization over the solution space, still to be able to perform the formation task in the way the model was designed, i.e. to account for changes in unit attributes such as ranged and melee units and for different levels of unit health. Furthermore, the formations are supposed to be created quickly. As the results in Section 8.1.5 shows, these goals for the performance of the *Formation* module are achieved.

This section analyses the effects of using a solution case-base on the number of solutions to be explored in the experimental evaluation. The number of solution cases was recorded during the experimental evaluation for all scenarios. Table 8.7 shows the number of solutions per case averaged both over the different scenarios and over all cases in a scenario. These results were obtained using a similarity threshold $\psi_{formSol} = 80\%$.

Averaging among all scenarios is possible since the number of possible solutions is only tied to the number of units that create a formation irrespective of the scenario.

Unit Number	Possible Number of Solutions	Actual Average Number of Solutions	Reduction
2	2	2	0%
3	6	5.83	2.83%
4	24	21.70	9.58%
5	120	59.38	50.52%
6	720	282.67	60.74%

Table 8.7: Number of Recorded Solutions vs Number of Possible Solutions by Scenario and Unit Number

The table shows that for low unit numbers, which in turn lead to low numbers of possible solutions, there is little to no difference between the actual number of solutions and the possible number of solutions. This is similar to other case-bases with small case spaces as there is little potential for abstraction and also little need for it, since the number of solutions to be examined is low and easily managed in reasonable time.

With higher numbers of units and thus higher numbers of potential solutions, the benefit of using a solution case-base becomes apparent. For six units, the highest number of units used, using the solution case-base leads to a reduction of over 60% in the number of solutions to be tested. This leads in turn to a significant reduction of computational effort and running time required when looking for the best possible solutions.

An adjustment of the similarity threshold $\psi_{formSol}$ to lower values would lead to a further reduction in the number of solutions being examined. However, this would have to be carefully weighted against potential decreases in performance. Currently, for a similarity threshold of $\psi_{formSol} = 80\%$, the use of a solution case-base has the desired effect in that it leads to a significant decrease in the number of solutions while enabling a performance that still gets very close to the optimal solutions possible.

8.2 Unit Attack

The *Attack* action is a central component in all combat-oriented RTS games. The main distinction here is between the standard attack that units perform and potential special attacks that only certain unit types can perform. Furthermore, there is often a distinction between units that can only attack certain groups of units. One common way to create unit types that can only attack or be attacked by certain other unit types is to group units into *ground units* and *air units*. A specific unit type can then be classified as being able to either attack *ground*, *flying* or both types of units. Such a distinction between unit types adds an additional strategic element to be considered in combat. Another big distinction is between units with melee range and units that can attack opponents from further away.

As part of this *Attack* component, it was decided to focus only on one of the three mentioned attack characteristics: the differences in attack range. The usage of special attacks was excluded since each special attack effectively is a complex learning task of its own. Every single special ability has distinct movement and usage patterns that maximize its optimal application. Due to this complexity, bots that play the entire game mostly either use hard-scripted behaviours for special abilities or ignore them entirely (Ontañón et al., 2013). Additionally, special attacks are specific to a particular game, e.g. StarCraft, and one focus of this hierarchical architecture is to create an approach that is generalisable beyond a particular simulation environment.

The usage of flying units was excluded due to their technical implementation inside the StarCraft game. The management of flying units is very similar to that of ground units. However, since flying units can ignore both collisions with other units and with fixed map objects such as cliffs, learning the ideal usage of flying units can be daunting. Several parts of the model of the *Navigation* module, such as the entire influence map that encodes map features, do not apply. Therefore, flying units are not part of the attack module evaluation either, even though the model does theoretically include them through the *UnitType* feature (see Section 8.2.1). Given a future extension of the *Navigation* module, the ability to attack with flying units could easily be added.

In general, the aim of the *Attack* action can be summed up as finding the most efficient use of the firepower available to the player or agent. While other components on the same or lower levels in the hierarchy of the architecture, such as *Retreat*, *Formation* and *Navigation* try to manoeuvre units into position or keep them alive for as long as possible, an *Attack* action simply tries to deliver the most damage possible. The goal of using attacking units in the most efficient way can be summed up under the often-used term *Focus Fire*. *Focus Fire* means focusing on a specific opponent unit in order to eliminate it and, as a result, also eliminate the potential damage it can do to agent units. Using exactly the right amount of

damage (i.e. not committing too many units to attack an opponent) is crucial in order to get the most out of the agent's units. The *Focus Fire* behaviour is also being learned as part of the architecture described in Chapter 5. The *Attack* module re-uses parts of the knowledge gained in that chapter but also modifies that approach to better suit into the hierarchical architecture.

Another RTS game unit behaviour that could potentially be categorized into the *Attack* action is *Kiting*. In order to learn *kiting*, which was also the aim of the agent described in Chapter 4, the scenarios have to give the agent a number of faster, weaker units that confront slower (potentially stronger) opponents. However, kiting requires the use not only of *Attack* actions but also of *Retreat* in order to remove units from danger. For this reason, the aim in this section is only to learn appropriate behaviour for *Focus Fire*.

As mentioned in the previous chapter, unit attacks are a crucial part in a combat-oriented RTS game such as StarCraft. All other actions serve to support the *Attack* action, to keep units alive and manoeuvre them into good positions to perform their attacks. In terms of the reward signal that is used to guide the learning process, the *Attack* action is the only action that results in positive reward by reducing the health of opposing units (see Section 8.2.1).

The procedure for creating a RL model, evaluating its effectiveness to represent the game world suitable for the CBR/RL *Attack* module and subsequently training the case-base for the *Attack* action is similar to the procedure presented in Section 8.1 for the *Formation* component.

8.2.1 Unit Attack Model

The model that is used for the *Attack* action is very similar to that used for the *Formation* action described in Section 8.1. The model describes the problem in terms of an MDP. Since the *Attack* problem is more direct and the reward less convoluted for the reasons described in the previous section, the model presented here is also less complex.

The aim of an *Attack* action is to deliver the most damage in the shortest possible time. It was decided to simplify attacks by having all units that are assigned to an *Attack* action at a certain time focus on the same target unit. This is done to keep the number of possible solutions as low as possible. If it is possible to assign units among different targets, the number of possible solutions grows exponentially with both the number of attacking units and the number of targets. If all units are assigned to the same target, this means that for n opponents there are n possible solutions, regardless of the number a of agent units. If, on the other hand, attacking agent units can be freely assigned to any opponent, this can be described as the combinatorial problem of distributing a labelled balls among n labelled urns (R. P. Stanley, 1986) which results in n^a possible solutions. Even in small scenarios with, for example, $a = 5$ agent units and $n = 4$ opponent units (i.e. possible targets) this results in

the huge number of $n^a = 1024$ possible solutions for every single case. Such a large number of solutions would require a solution case-base similar to the one used for the *Formation* action (see Section 8.1.2) or another way of abstraction to make the approach viable. For this reason, it was decided to simplify the module by having only a single attack target for each attack action and to leave the option to use different targets for future work.

The damage dealt to the opponent can be maximised by choosing a target that has enough hitpoints so that all attacking units can deliver their maximum damage. Additionally, there is a bonus for completely eliminating an opponent unit and thus removing its damage potential from the game. Completing the action in the shortest possible time is achieved by choosing targets that are close and easily reachable for all attacking units. This means that it is necessary to also take into account potential collisions with the agent's own or opposing units. These collisions are handled by the *Navigation* component on *Level Three*. Examining the trade-off among these three influences through trial-and-error, the CBR/RL *Attack* module attempts to find the best possible *Attack* policy. The model that is described in this section is designed to enable the agent to find this optimal policy.

Attack States

Attack states are the case descriptions that include all relevant information when a new *Attack* action is triggered. One simplification that is possible due to the way the *Attack* action is defined, is that any attributes that identify specific agent units can be ignored. Since all agent units are assigned the same target, the only thing that matters is the combined amount of damage that agent units can deliver, as well as the average distance from agent units to a given target and thus the average time it will take to complete the attack.

Even attributes that seem relevant to *Attack* actions, such as weapons cooldown and the exact position of agent units relative to their opponents, can be ignored. Both of these attributes are relevant for deciding which unit should be assigned to an *Attack* action, i.e. they are relevant for the *Tactical Unit Selection* decision on *Level One* of the architecture (see Chapter 9). Once a unit has been ordered to attack, only the amount of overall damage relative to a target units health and the distance from this target unit to all attacking units is important.

Category	Attribute	Type
Index	Units Opponent	Integer
Target Unit	Type	Enum
	Health	Integer
	Average Distance to Attackers	Integer
Agent	Combined Attacking Unit Damage	Integer

Table 8.8: Attack State Case Description

Table 8.8 lists the attributes that make up an attack state description, ordered by categories. The initial index that identifies cases in the case-base is the number of opponent units in this game state. Since solutions are strict per-target assignments with no contingency for missing or additional target units, there is no re-use between cases for different numbers of potential targets.

Each opponent unit is defined by its average distance relative to all agent units firing range and thus the time it takes for these units to attack (see Figure 8.14), their unit type and their health in terms of an absolute value. The average distance is measured in pixels.

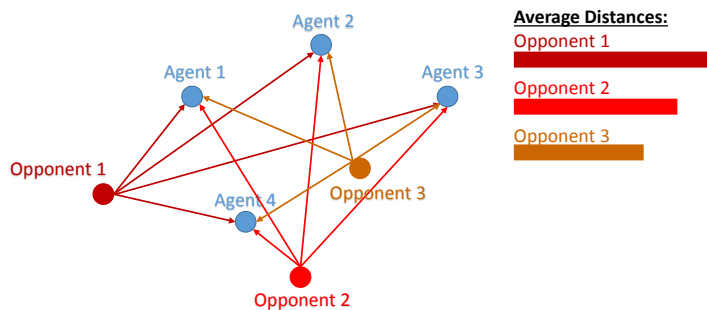


Figure 8.14: Average Distance from Opponent Units to Attackers

Since all attacking agent units have the same target, they can be abstracted into a single value per case - their combined attacking damage. This value is important for the agent when it wants to maximize the damage done. Ideally, the agent wants to choose a target that is close and whose hitpoints are exactly or just below the damage that can be done by the attacking agent units, thus gaining the elimination bonus while not over-committing firepower.

Attack Actions

The potential case solutions for attack cases are, as mentioned above, the attack targets. This means that there is one solution for each attack target. The unit mapping described in Section 6.3 is used to ensure that similar target units from the problem encountered in the current game are matched to targets in the case retrieved from the *Attack* case-base.

Attack State Transition Probabilities Given the high level of abstraction, especially through the simplification of all agent units into a single damage value and their average distance to the targets, transition probabilities are stochastic. This means that after executing the solution a_i in the same initial state s_t at two different times can lead to two different states s_{t+1} .

Attack Reward Similar to the reward signal in previously defined modules, the reward signal for the *Attack* action has several different components. The RL reward signal is built to entice the agent to learn how to fulfil the three main aims which have been identified in terms of the *Attack* action: Maximise the damage done, eliminate the target unit and complete the *Attack* action as quickly as possible.

The component of the reward signal that reflects the time taken to complete the attack, t , contributes a negative value relative to the duration of the action in in-game frames. The other two components of the reward signal are positive. There is a positive reward component dam for the damage to the target. This number is limited by the hitpoints of the target, e.g. if the target has only 1 hitpoint, $dam_{max} = 1$ for choosing that particular target. The other positive component dam_{elim} is either 0 (if the chosen target survives) or reflects the damage the opponent could potentially have dealt. This guides the agent to learn to first eliminate opponent units that deliver large amounts of damage. Both the negative and the positive components are normalised. The resulting composite reward signal is

$$r_{att} = dam + dam_{elim} - t_{att}.$$

As a result, the agent is compelled to choose targets that can be easily reached, have a high damage output, and whose health is just below the maximum damage all the combined agent's attacking units can effect.

8.2.2 Attack Evaluation and Training

The evaluation of the *Attack* task is designed to address all possible scenarios which can arise through varying the variables that make up the definition of the model in the previous section. The knowledge obtained during the evaluation also serves as training data for the case-base

for re-use by the *Tactical Unit Selection* component on *Level One* of the architecture. During this re-use, no new knowledge is obtained, neither in the shape of new cases nor through updating existing cases and their solutions through RL. This means that the scenarios used in this section are designed to address as much of the overall case-space as possible.

Similar to the composite nature of the reward signal, there are two distinct aspects that are evaluated in terms of *Attack* action performance: The damage that is done and the target units that are eliminated (i.e. the positive components of the reward signal) as well as the combined reward signal. The combined reward signal also includes a negative component that reflects the speed with which the action is performed. The combined reward signal thus shows how well the agent manages to learn the trade-off between selecting distant targets that result in higher positive scores versus selecting closer targets that result in lower scores. Unlike the *Formation* action evaluation, there is no scenario that only evaluates the speed of performing the action. This is because the speed with which the *Attack* action can be completed is directly tied to the *Average Distance* attribute that is part of the case description, i.e. this would result in the agents simply picking the case with the lowest average distance.

An important consideration is the general design of the evaluation and training scenarios. Given the prerequisite to create cases that cover as much as possible of the case-space for use in future, higher-level computations, it is important to have methods to vary the relevant case description attributes. Similar to the procedure used in the evaluation scenarios for the *Formation* action (see Section 8.1.4) the scenarios created for the training and evaluation in this section use a number of randomised map triggers to vary certain case attributes. Since the approach to train a single action in each game proved very successful for obtaining non-noisy training data, it was decided to design the evaluation for *Attack* actions in a similar fashion.

This learning strategy requires a large number of training episodes/games since only one possible solution is explored per game, compared to about 40 in the evaluation scenarios in Chapter 7 and even more in the evaluation scenarios in Chapters 4 and 5. However, contrary to the *Formation* problem, the model designed for the *Attack* action is straightforward and thus not expected to require the tens of thousands of episodes that it takes to learn optimal formation solutions.

However, since only one solution is picked at the beginning of a game and subsequently executed and evaluated before the scenario is restarted, in a fixed game scenario this would mean that the same cases are explored again and again. In order to acquire relevant knowledge for different game situations, a large number of different scenarios would have to be run. To avoid this and to cover a large amount of case space in a single scenario, randomised scenarios similar to those used in Section 8.1.4 are used. For the *Formation* module evaluation scenarios, **opponent** units were spawned in randomised locations. For the *Attack* module

on the other hand, **agent** units are spawned in randomised locations in order to vary the *AverageDistance* attribute that target units have. Figure 8.15 shows how the agent units can spawn at any of a number of predefined locations on the map, thus randomising the distance between units.

Defining the scenario in this way enables the agent to explore a large number of possible settings of the *AverageDistance* attribute. In order to explore different numbers of target units as well as different amounts of agent attack damage and target unit health levels, more than one evaluation scenario is still required. By having target units with different health levels as part of the same scenario (which is also shown in Figure 8.15), the number of required scenarios is reduced. Having target units with different health levels in the same scenario also means that the agent can learn the best possible policy for addressing the trade-off between target distance and target health.

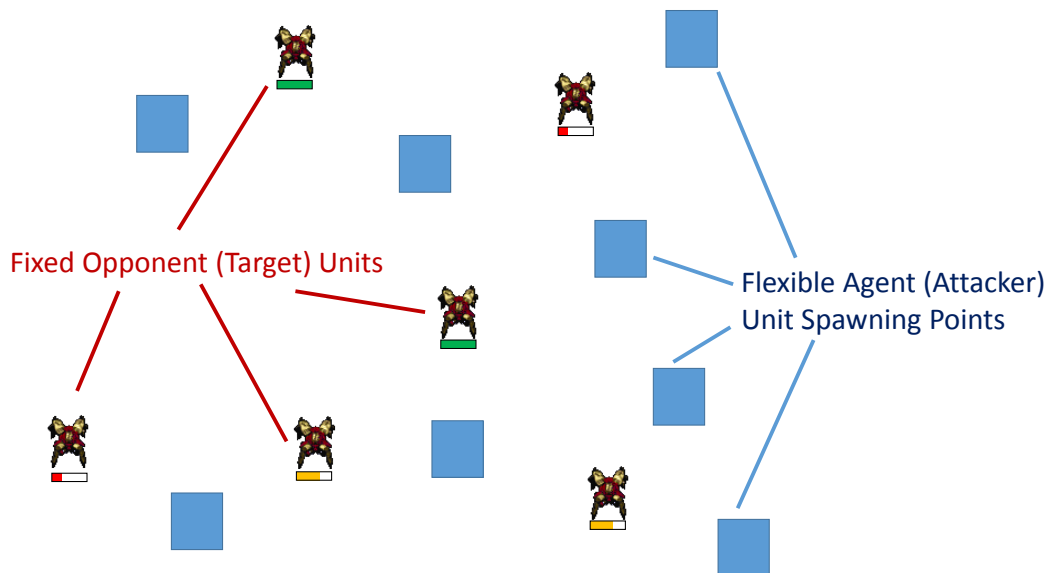


Figure 8.15: Randomised Attack Scenario

In order to evaluate the effectiveness of the model, i.e. to see if it works at all for the chosen learning task, the following scenario was selected as a representative. In *Attack Scenario A* the CBR/RL agent controls only ranged units against a number of fixed opposing ranged units. The agent units are created randomly (see Figure 8.15) among a number of locations, where each unit can potentially appear at each location, e.g. all units could also potentially appear at the same location. The aim is for the agent to assign optimal numbers of attackers in order to eliminate all opponents in the fastest possible way.

Given the model described in Section 8.2.1, the number of agent units is not relevant for the *Attack* action, only the sum of their attack damage. However, in order to get nuanced

values for the *AverageDistance* attribute that opposing target units have, it is important to use a number of agent units and not just a single one. *Scenario A* has eight randomly spawned attacking units, while other scenarios used for additional training and described in detail in the next section have between two and eight agent units.

Two more conditions were applied for the evaluation scenarios in order to avoid unnecessary noise that makes learning more difficult. The opponent target units do not move but stay fixed in place. This way, there are no unpredictable collisions and the *AverageDistance* attribute stays the same for each scenario. Additionally, opponent units do not attack agent units. Since there is only a single *Attack* action per episode and the damage done by the opponent is not included in the reward signal, this does not influence the outcome. While opponent damage counts in the overall *Tactical Unit Selection* model, the *Navigation* model and the *Formation* model, it is irrelevant for the *Attack* action that only serves to learn the best *Focus Fire* policy.

All test scenarios were run using a similar RL configuration as for the *Formation* evaluation described in Section 8.1.4. This means a learning rate $\alpha = 0.1$ and a discount rate $\gamma = 0.8$ for the one-step Q-learning algorithm. The agent also uses a declining ϵ -greedy exploration policy that starts with $\epsilon = 0.8$. The agent initially selects actions 80% random and 20% greedy, i.e. choosing the best known action. The exploration rate declines linearly towards 0 over the course of the experiments. After the exploration rate has reached 0, there are another number of games where the agent uses only greedy selection. The performance in this final phase therefore showcases the best policy the agent learned.

The general experimental setup was also similar to previous evaluations. Each experiment was run five times and the results were averaged of these runs. The length of each experiment was decided based on the observed coverage of the state-action space. If there were a large number of unexplored actions or actions that were only infrequently explored, more episodes had to be run. The state-action- or case-space for the *Attack* action is less extensive than that for the *Formation* actions. As a result the evaluation for *Scenario A* was run over 3,000 episodes, compared to up to 30,000 episodes for the *Formation* action. Subsequent additional training scenarios introduced in the next section required between 1,500 and 3,000 episodes.

The similarity threshold ψ_{att} that decides when new cases in the *Attack* case-base are created was set to 85% after some initial experimental runs. Table 8.9 sums up the parameters of the evaluation.

Parameter	Values
Scenario	A(8vs6)
Number of Games	3000
Algorithm	One-Step Q-learning
Attack Case-Base Similarity Threshold ψ_{att}	85%
RL Learning Rate α	0.1
RL Discount Factor γ	0.8
RL Exploration Rate ϵ	0.8 - 0

Table 8.9: Attack Evaluation Parameters

8.2.3 Initial Attack Results and Discussion

Figure 8.16 shows the results for *Scenario A* using the settings described in the previous section.

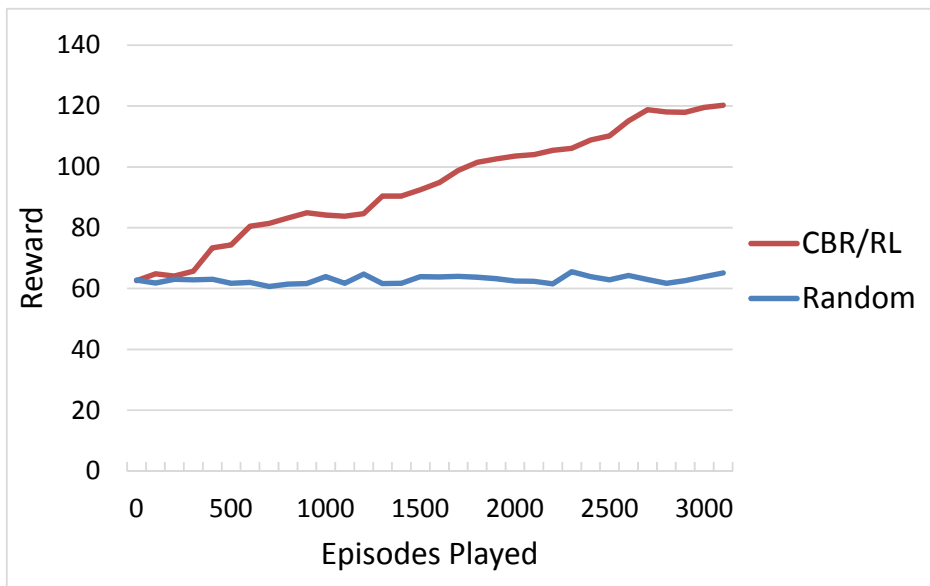


Figure 8.16: Results for Attack Scenario A

The CBR/RL agent manages to double its reward over time when compared to the random action selection, from a value of just above 60 to 120 for the composite reward signal. Since only a single *Attack* action is executed in each episode, it did not make sense to evaluate the

number of eliminated opponents over time as they would always be either 0 or 1. Longer experimental runs that re-use the knowledge gained in this section as part of the overall agent will show how well the agent learned to prioritise the elimination of opponent units.

The diagram shows that the increase in reward is evenly distributed over the number of training episodes. Looking at the case-base statistics also reveals that the number of non-explored case solutions is below 10%, indicating that the number of training episodes is well chosen for the similarity threshold $\psi = 85\%$. This also means that the agent has either managed to find the optimal policy π^* or a near-optimal policy.

8.2.4 Additional Attack Training Scenarios

The results of running the evaluation for *Scenario A* prove that the model described in Section 8.2.1 contains the knowledge necessary for the agent to learn the *Attack* task. Similar to the *Navigation* and *Formation* modules, the final goal in this section is to acquire the knowledge necessary to have a ‘good’ *Attack* action in any potential situation which can arise when working the highest *Tactical Unit Selection* actions on *Level One*. This is necessary since, as pointed out in Section 6.2, there is no more learning on *Level Two* during the evaluation of *Level One*. Any knowledge in terms of *Formation* and *Attack* and *Navigation* actions has to be acquired in previous training scenarios.

Therefore, a number of training scenarios were designed to cover as much of the case-space as possible, given the model that is used. The only variable of the model defined in Section 8.2.1 that can be covered sufficiently within a single scenario is the *AverageDistance* of targets to agent units. This leaves the following attributes to be covered in separate scenarios:

- Number of Target Units
- Target Unit Type
- Target Unit Health
- Summed Up Damage Agent

Given the goal of covering as many case-descriptions as possible and these guidelines, training scenarios with the permutations of parameters as listed in Table 8.10 were run.

Target Units	2, 3, 4, 5, 6, 7, 8, 9, 10
Target Unit Types	<i>Ranged, Melee, Ranged&Melee</i>
Target Unit Health	40, 80, 120, 160, 200
Agent Damage	25, 50, 75, 100, 150, 250

Table 8.10: Attack Training Scenario Parameters

Chapter 8. Architecture Level Two: Squad-Level Coordination

Creating one scenario for each possible permutation of the parameters leads to $9 * 3 * 5 * 6 = 810$ scenarios. The number of training episodes that each scenario requires varies. This number is tied directly to the number of *Target Units* and is set between 50 for only two target units up to 5,000 for ten target units. Due to the vast number, each scenario is not run five times as done previously, but only three times in order to limit the overall time the evaluation takes. The algorithmic parameters are identical to those used in the initial *Attack* evaluation given in Table 8.9. Similarly to the evaluation in the previous section, the two additional conditions that were put in place to avoid noise in the learning process also stay in place: the opponent target units do not move but stay fixed in place; and the opponent units do not attack agent units since opponent damage is irrelevant for the *Attack* action.

Using these parameters with a similarity threshold $\psi = 85\%$, the resulting fully trained case-base for the *Attack* action appears as shown in Table 8.11.

Number of Target Units	Cases	Case Solutions
2	27	54
3	101	303
4	386	1544
5	745	5215
6	1369	10952
7	2117	17129
8	3966	29073
9	5491	43155
10	7810	59111
Total	22012	166536

Table 8.11: Attack Case-Base after Training

8.3 Unit Retreat

The *Retreat* action is a single-unit action. This is in contrast to the *Attack* and *Formation* actions which coordinate among several agent units. Similar to those components however, *Retreat* uses the pathfinding component described in Chapter 7. *Retreat* is designed with similar aims as the retreat action of the RL agent in Chapter 4 where the unit attempts to avoid potential sources of damage. One of the downsides of the retreat action for that agent was the large amount of expert knowledge that it took into account: The retreat direction was computed as a direct product of all opponent units within a given radius. The *Retreat* action used in the hierarchical architecture is designed to be a more controlled and atomic action by using the already existing agent IMs as its basis.

In order to retreat, the agent selects the part of the IM with the lowest enemy damage potential and moves the unit towards it. *Retreat* is designed to be more sensitive than the movement actions that usually use the unit IM fields and that additionally also take agent damage potential into account. Furthermore, *Retreat* is designed to work on a larger scale than the basic unit movement that the 7×7 IM described in Section 7 was used for. Therefore, the process of deciding on a destination to retreat to was split into two steps to work on a larger subsection of the overall IM. The larger subsection measures 15×15 plots around the location of the unit in question. The size of the IM field that is taken into consideration was chosen after several experimental runs during which the effects of the retreat action were examined. The 15×15 size was determined as being close to the maximum distance the unit can potentially retreat in relation to the average time other actions take for different scenarios.

The first step for a retreating unit is the selection of the 5×5 quadrant on the outer edge with the overall lowest damage on average potential, i.e. one of the eight quadrants that surround the inner 5×5 quadrant. Subsequently, the IM field with the lowest damage potential within this pre-selected quadrant is chosen as the target destination for the retreating unit. Figure 8.17 illustrates the procedure.

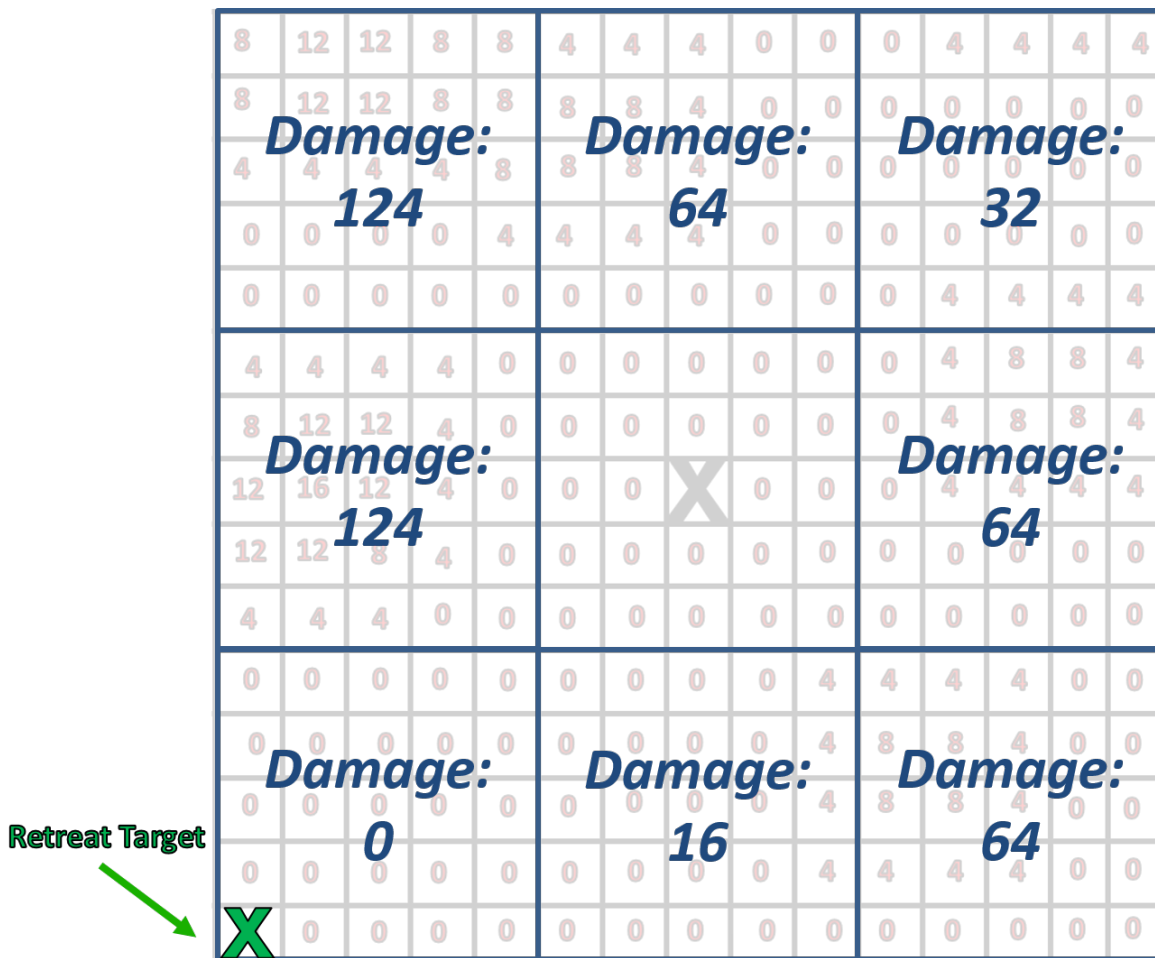


Figure 8.17: Retreat Destination Computation Based on IM Values

In this example, first the bottom-left 5x5 quadrant is selected since it has the lowest overall damage of 0. Subsequently, the IM field with the lowest damage within this quadrant is selected. If there are multiple fields with the same value which could be selected, the field that is furthest away from the current unit position is selected.

An alternative to using a two-step process would have been to simply select the single IM plot with the lowest threat potential on the 15x15 map. However, this would have ignored the influence of any neighbouring plots. Having a more high-level view is crucial for a medium term perspective on finding the safest area in the immediate surroundings of a unit. Using this two-step process is a trade-off between using as much information as possible and abstracting this information in an efficient fashion to optimize performance.

IM fields that go beyond the map borders either in horizontal or vertical directions are attributed with prohibitively large damage values and therefore never selected as retreat

destinations. A unit's *Retreat* action finishes once it reaches its target destination. The overall *Level Two Retreat* action finishes once all retreating units have reached their destinations.

Since the *Retreat* action is hard-coded and not learned, its effectiveness cannot be determined on its own. Instead, the evaluation of the *Tactical Unit Selection* component on *Level One* which makes use of the *Retreat* action shows if the implementation described here is an effective one.

8.4 Summary

This chapter introduced the individual modules that are part of *Level Two* of the hierarchical architecture. The two learning components, *Formation* and *Attack*, were described in detail. Each of these modules was shaped into a CBR/RL module that acquires the knowledge necessary to perform its respective task through interacting with the game environment. Both *Formation* and *Attack* tasks were found to require highly specialised evaluation and training scenarios due to the specific nature of the sub-problems they address. Both modules managed learn successful policies that solved their respective tasks while re-using knowledge from the *Navigation* component.

The next chapter introduces the highest level of the hierarchical architecture, *Tactical Unit Selection*. This module integrates all lower-level modules to solve general micromanagement scenarios. The quality of the knowledge acquired in modules that were described this chapter will thus directly influence the performance of the *Tactical Unit Selection* module. Furthermore, the next chapter should lead to interesting observations regarding the effects of re-using knowledge over three levels of hierarchically interconnected modules.

Architecture Level One: Tactical Decision Making

Any architecture that addresses mid-tier RTS game tasks as defined in Section 3.1, has one or more components which address tactical problems. However, depending on the researchers, the specifications can be very different in terms of exactly which problem is to be addressed by the module, i.e. which ‘tactical’ tasks fall into its area of responsibility. Often, tactical tasks are mixed with reactive control tasks or are very specific to the game environment.

Synnaeve et al. (2012) distinguished among a set of four different attack patterns, *ground, air, invisible* and *drop*. All of these concern sometimes large sets of units, however, the tactical module leaves any decision beyond the choice of one of these four strategies to lower-level modules. Cadena & Garrido (2011) used a task decomposition similar to the one used in the model developed in Section 6.1 and defined an intermediate level that manages groups of units. However, they defined tactical tasks mostly in terms of positioning groups of units on the overall map. Bowen et al. (2013) used a *Military Manager* to handle any action related to combat except for high-level attack and retreat commands. All tactical and micromanagement tasks are addressed by this component. Each of the bot architectures examined in Ontañón et al. (2013) has a different interpretation of what belongs to a tactical component. The architecture presented in this thesis makes a clear distinction between tactical and reactive tasks and can thus lead to a clearer general architecture for RTS game AI agents.

This chapter describes the highest level in the hierarchical architecture, the *Tactical Unit Selection* module. The *Tactical Unit Selection* reasoner can be described as an ‘inter-squad coordinator’. Its task is to distribute all units among the different action categories on *Level Two*. The *Tactical Unit Selection* module does not issue any actions to units directly; it only works on a meta-level and coordinates among squads of units.

This *Tactical Unit Selection* component is structured in a way similar to that of components in previous chapters. The module also uses the same algorithmic approach based on a hybrid CBR/RL integration. The module makes use of the knowledge acquired during the evaluation

and training of *Level Two* and *Level Three* modules and stored in the relevant case-bases. These lower-level case-bases are immutable after being trained to an extent that covers all relevant aspects of the case-space as described in Sections 8.1.4, 8.2.2 and 7.5. This means that the *Tactical Unit Selection* module described in this section re-uses the previously-acquired knowledge in the form of case-solutions for its higher-level goals without further adapting these solutions.

Given the decomposition of the problem as described in Section 6.1, the task of the *Tactical Unit Selection* component is to find an ideal distribution of units among the three different *Level Two* modules. In order to achieve this, the CBR/RL agent evaluates all possible unit distributions among the different tasks and evaluates the resulting performances. A major simplification was introduced in order to keep the number of possible solutions manageable: all units assigned to *Attack* or *Formation* actions will perform the same action. This means that any unit assigned to an attack will attack the same target. Any unit assigned to a formation, will be part of the same formation. If the environment were not restricted, it would be possible to freely create any number of *Level Two* actions with any number of units assigned to it. However, that would increase the number of possible solutions exponentially and make learning infeasible with the current model without further abstraction or simplification. Therefore, adapting the agent to allow multiple identical *Level Two* actions was left for future work (see Chapter 10). Furthermore, the number of units controlled by the CBR/RL agent in the evaluation scenarios has been limited to ten, a number that would usually not require the formation of many sub-squads. Ten units is at the lower end of the spectrum of what is a normal combat situation in StarCraft. Using scenarios with ten units can thus be regarded as addressing a subset of the overall problem, where the agent controls dozens of units in a complete game of StarCraft. However, ten target units is the maximum that the *Attack* case-base has been trained with. The *Formation* module is actually only trained with six units, thus necessitating a work-around in scenarios where more than six units can be assigned (see Section 9.3). An extension to allow for multiple separate groups of units which attack or create a formation would only require minor changes to the implementation and is further discussed in Chapter 10.

This section is structured as follows. First, the model that is created to represent the problem is described. This includes the MDP components *State*, *Action*, *Reward* and *Transition Probabilities* as well as further implementation details specific to this particular problem. Subsequently, the approach is tested in a number of scenarios. Some of them are considerably more complex than previously used scenarios in order to test the agent's capabilities to re-use the previously-acquired knowledge from lower levels of the architecture. The results of this evaluation are analysed and discussed in detail.

9.1 Tactical Decision Making Model

The model used for the *Tactical Unit Selection* module is very similar to those used for the *Formation* and the *Attack* components described in Sections 8.1 and 8.2.1. The model is designed to describe the problem in terms of an MDP. The underlying problem for the *Tactical Unit Selection* component is how to ideally distribute the available agent units among the three different *Level Two* actions. This problem is an integration of three modules and the model also combines elements of these modules. As a result, both state descriptions and the reward signal contain aggregated elements of the respective parts of the *Formation* and *Attack* modules. However, both the problem which is addressed by the *Tactical Unit Selection* component and the resulting model are considerably more complex than either squad-level component. Similarity computation also requires a much larger computational effort to account for the higher dimensionality of state/case descriptions when compared to previous problems. In order to speed up similarity computation and enable real-time case retrieval with large case-bases, the Hausdorff distance (see Section 3.4.2) between sets of agent or enemy units is used as an estimate of the overall case similarity. Section 9.1.2 explains the similarity computation in detail.

In the context of the StarCraft RTS game, the *Tactical Unit Selection* module attempts to win the overall combat scenario. This translates into dealing as much damage as possible with the units the CBR/RL agent has at its disposal while keeping those units alive for as long as possible. The case solutions for *Tactical Unit Selection* cases are distributions of units among the three possible actions *Formation*, *Attack* and *Retreat*.

Defining case solutions in this way means that the main parameter which can be adjusted is the number of units for each of the three categories. Given five agent units, the possible distributions are shown in Table 9.1.

Units <i>Attack</i> u_a	Units <i>Formation</i> u_f	Units <i>Retreat</i> u_r
5	0	0
4	1	0
4	0	1
3	2	0
3	1	1
...
0	0	5

Table 9.1: Possible Tactical Solutions for $n = 5$ Units

Using this method to define all possible solutions, the overall number of solutions for n units distributed among the 3 categories is $\binom{3+n-1}{n}$. For example for $n = 5$ the number of solutions is $\binom{3+5-1}{5} = 21$.

Additionally, specific solution slots could be numbered for each unit, similarly to how the *Formation* module works. That would mean that not only the category to which a unit is assigned would matter, but also the specific slot in that particular category. However, this would increase the number of possible solutions significantly from $\binom{3+n-1}{n}$ to 3^n . The example with $n = 5$ would then result in 243 solutions per case. For this reason, it was decided to assign only unit numbers to the three different action categories. The problem of consistently assigning similar units to the same slots for subsequent executions of the same solutions is handled by fixed logic, described in Section 9.1.3.

9.1.1 States

Tactical Unit Selection states (or cases) are basically a combination of *Attack* and *Formation* states. However, some of the attributes that the *Level Two* state models use are part of both *Attack* and *Formation*, while others contain the same information but in less detail. *Attack* requires detailed information on enemy units while *Formation* requires detailed information on agent units. In each of those modules, the other group is only recorded in abstract form: opponent damage in eight surrounding quadrants for *Formation* and the summed up agent unit damage for *Attack*. *Tactical Unit Selection* actions require detailed information on both groups of units, in addition to other attributes like the weapons cooldown of units, in order to decide when a unit is ready to attack again. The resulting composition of the case description of a *Tactical Unit Selection* state can be seen in Table 9.2.

Category	Attribute	Type
Index	Units Agent	Integer
	Units Opponent	Integer
Unit Agent	Type	Enum
	Health	Integer
	Damage	Integer
	Quadrant	Integer
	Cooldown	Boolean
Unit Opponent	Type	Enum
	Health	Integer
	Damage	Integer
	Quadrant	Integer
	AverageDistance	Integer

Table 9.2: Tactical State Case Description

The state composition consists mostly of attributes which exist in this form for *Attack* and/or *Formation* as well. Opponent units contain two attributes that indicate their position: *Quadrant* and *AverageDistance*. Both of these are abstractions not just of their own positions, but also of their position relative to agent units. As such, these attributes are easier to work with and contain more information than simply having x/y coordinates. The two attributes contain different information (direction versus distance) which is why both are required.

Agent units also have the *Quadrant* attribute to indicate their position relative to each other. One agent unit attribute that was not part of previous *Level Two* actions is *Cooldown*. This Boolean value indicates if a unit's weapon is currently in cooldown or if it can be used. This attribute is important for the RL learner when choosing how many units to assign to the *Attack* action. *Health*, *Type* and *Damage* are all attributes which have also been used in *Level Two* modules and which contain important information for the *Tactical Unit Selection* action. *Type* is more abstracted at this level and only distinguishes among *Melee*, *Ranged* and *Air* (even though the tested scenarios do not contain air units) instead of specific unit types.

Given the case description in Table 9.2, the dimensionality of a *Tactical Unit Selection* case description is considerably higher than for previous problems. This is mostly because the overall dimensionality is more closely linked to the number of agent units n_a and opponent units n_o . Additionally, each unit has more attributes recorded than in previous models. For example, in a scenario with $n_a = 4$ agent units and $n_o = 5$ opponent units, case descriptions have $2 + 4 * 5 + 5 * 5 = 47$ attributes. This is due to the increased complexity of the *Tactical Unit Selection* problem as well as the higher precision required in these decisions to enable consistent conditions for decisions on *Level Two* and *Level Three*.

As a measure to handle the high dimensionality of case-descriptions as well as the large number of cases, the *Hausdorff Distance* is used to pre-select by computing the similarity between the sets of agent and opponent units in a given case and the cases stored in the case-base. The approach in Section 5 uses a different high-level comparison by abstracting the case descriptions which consist to a large degree of spatial data in the form of IMs into histograms. While histogram-based similarity comparison worked well for the few high-dimensional cases in Section 5, the case-base in this section contains a large number of cases that are mostly based on unit characteristics. This problem translates badly into histograms (see Section 3.4.2). Preliminary testing using the Hausdorff distance to find similar cases led to fast and accurate selections.

9.1.2 Tactical Case Similarity

Using the Hausdorff distance requires for all values that are part of a unit description to be normalised. This is necessary, since units are treated as n -dimensional points in a Euclidean

space (see Section 3.4.2) where n is the number of attributes a unit is described by. The normalization that is used here maps all unit attributes to a value between 0 and 1. Table 9.3 lists the attributes as well as their normalisation functions.

Attribute	Type	Range	Similarity Function
Type	Enum	<i>Melee, Range, Air</i>	Custom
Health	Integer	0 - 550	Divide by Max
Damage	Integer	0 - 75	Divide by Max
Quadrant	Enum	0, 1, 2, 3	Custom
Cooldown	Boolean	0 or 1	0 or 1
AverageDistance	Integer	0 - MapSize*32	Divide by Max

Table 9.3: Tactical Unit Attribute Similarity Computation

After normalising each unit description, the Hausdorff distance is computed. This happens twice, once between the sets of units for agent units and once for the sets of opponent units. The two distances are then used in the overall similarity computation for the comparison of the current game state with the case from the case-base that is being examined. Given n attributes and a normalisation for each attribute to a value between 0 and 1, the maximum distance in the Euclidean space is \sqrt{n} . The similarity for each group of units for agent and opponent is normalised once more to a value between 0 and 1. Finally, both agent and opponent unit similarities are weighed and combined into the overall case similarity. The resulting overall case similarity c_{sim} is:

$$sim_{all} = \alpha * sim_{agent} + \beta * sim_{opponent}. \tag{9.1}$$

The weights α and β for the two similarities are chosen according to considerations as to which unit group is more relevant. This was determined through several preliminary test runs. As an additional safeguard to avoid the selection of unsuitable cases, a unit-to-unit similarity computation is done after the most similar case has been identified by using the Hausdorff similarity computation. This verification requires little extra logic since it simply reuses the functions used to do the unit-to-unit mapping described in Section 6.3. If this computation results in a similarity score that is $> 5\%$ less than the threshold ψ , a new case is created in the case-base.

Additionally, an index, consisting of the number of agent and opponent units, is used to do a case pre-selection. These two numbers have to be matched exactly, since the lower-level *Attack* and *Formation* actions require exactly matching numbers of opponent and agent units respectively as they currently have no mechanism to cope with additional or missing units. The requirement to match unit numbers exactly exists also in these lower-level modules.

9.1.3 Actions

Tactical Unit Selection case solutions are distributions of the available agent units among the three available *Level Two* actions, i.e. triples (n_a, n_f, n_r) that indicate how many units are assigned to each action type. As illustrated in Table 9.1, for n agent units this results in $\binom{3+n-1}{n}$ possible solutions. To sufficiently explore each possible solution, it is important to learn using a sufficiently large number of episodes that allows repeated exploration of every solution.

The way the solutions are defined also means that the number of agent units for which an effective solution can be learned is automatically limited, if the number of episodes is to remain reasonable. For 12 agent units, which in StarCraft combat scenario terms would count as a medium-sized squad, the number of possible solutions is $\binom{3+12-1}{12} = 91$. Since a larger number of agent units also results in more complex case descriptions, this automatically leads to a larger number of cases to be explored. For these reasons, the maximum number of agent and opponent units used in the evaluation scenarios is **ten**. By allowing a maximum of ten agent units in a game state, a single case can have at most $\binom{12}{10} = 66$ possible solutions.

Agent units are only assigned by choosing a particular number of units and not by selecting specific units. This is in contrast to the solutions for the *Formation* module (see Section 8.1.3) and avoids the combinatorial explosion an exact unit-to-action assignment would create for the number of possible solutions. Therefore, once a number of units has been assigned to a certain *Level Two* action, an additional mechanism is required to select which of the actual in-game units are selected. This is a crucial step as the effectiveness of the RL component is directly tied to the reproducibility of this selection. Randomness in the selection step would make learning an optimal solution basically impossible since both *Attack* and *Formation* actions are based on precise unit-to-unit assignments. (See Section 6.3.)

Because of this requirement for reproducibility, the selection process is, on the one hand, simple and is only based on few attributes to avoid unnecessary noise. On the other hand, it is specific enough to pick similar units under similar circumstances for a given category. For the different actions, the selection criteria are:

1. Select n_r *Retreat* units.
 - a) Sort all agent units by their health, weapons cooldown, damage (in that order) and select those with the lowest health.
 - b) If there are several units with identical health, select first those with a weapons cooldown that is not 0.
 - c) If there is still a tie, select those with the least damage.
 - d) If there is still a tie, select randomly among those tied units.

2. Select n_a *Attack* units.
 - a) Sort all agent units by their weapons cooldown, damage and health (in that order). Select those units with cooldown 0.
 - b) If there are more or less than n_a units with cooldown 0, select those with the most damage.
 - c) If there are several units with identical cooldown and damage, select those with the highest health.
 - d) If there is still a tie, select randomly among those tied units.
3. Select n_f *Formation* units.
 - a) Select all remaining units.

An alternative to this hard-coded selection process would be to have another case-base that assists the agent in the unit selection process, similar to the *Formation* solution case-base. This was left for future work. (See Section 10.)

9.1.4 Reward Signal

Similar to the *Tactical Unit Selection* states, the reward signal has strong similarities to the reward signals of both the *Attack* and the *Formation* modules, but is slightly closer to the reward of the *Attack* module. Similar to both these reward signals, there is a negative component t_{tac} for the time it takes for a *Tactical Unit Selection* action to complete. Furthermore, there is a negative component dam_{opp} for the damage that agent units received while performing the last action. There are two positive components, dam_{ag} for the damage done by agent units and dam_{elim} for the summed-up damage potential of all opponent units eliminated during the last action. Additionally, a third negative component dam_{loss} is added: this represents the damage potential lost when an agent unit is eliminated. This component is necessary in order to have an incentive for the agent to keep its units alive. Otherwise, losing an agent unit with 20% health would count the same as reducing an agent unit from 100% to 80%.

The resulting composite reward signal r_{tac} is

$$r_{tac} = dam_{ag} + dam_{elim} - dam_{opp} - dam_{loss} - t_{tac}. \quad (9.2)$$

This reward signal is designed to teach the agent that actions which finish quickly are to be preferred. However, the negative component for the time that is used is small compared to possible losses when agent units get damaged or even eliminated. Overall, the agent should

attempt to choose solutions which eliminate opposing units while sustaining no (or only very little damage) to its own units.

9.1.5 State Transitions

As with previous modules, state transitions are stochastic in spite of the mostly deterministic nature of StarCraft. The problem addressed by the *Tactical Unit Selection* module is more complex than those addressed in previous modules. As a result, a higher level of abstraction of both case description and solutions is required. This means that non-determinism shows even stronger than for modules on lower levels. The evaluation results in Section 9.4 show that even once the agent has been trained exhaustively and found an optimal policy π^* , there are occasions when it plays non-optimal games due to unexpected state-transitions.

9.2 Overall Hierarchical CBR/RL Algorithm

The *Tactical Unit Selection* module comprises the highest layer in the hierarchical CBR/RL architecture. Therefore, it is now possible to visualize the entire algorithmic implementation. Figure 9.1 shows a graphical representation of the steps and components involved in assigning actions to the available units. The algorithm chooses, from top to bottom, a *Tactical Unit Selection* unit distribution and, based on this distribution, an attack target, a formation unit-to-slot assignment as well as retreat destinations. Using the unit destinations computed through the *Level Two* components, the *Navigation* component on *Level Three* then manages the unit movement. There can be several *Navigation* actions until a unit reaches the destination assigned to it by one of the *Level Two* modules. There is always at most one action for each *Level Two* module, or zero, if no unit is assigned to a specific action category.

The overall *Tactical Unit Selection* action is finished once all modules on *Level Two* indicate they are finished with their tasks. Figure 9.2 shows the update-process for case solutions on the respective levels. As mentioned in Section 6.2, it was decided to not update lower-level case-bases while running higher-level modules. This means that while the diagram displays the retrieval of the reward information from the game environment and a subsequent update of the active case solution, this update process only happens during the evaluation of the relevant module and not during the evaluation and training of higher-level modules. This is indicated by greying out the affected steps on *Level Two* and *Level Three*.

Contrary to Figure 9.1, which works in a top-down fashion, the agent first controls lower-level incomplete actions and updates relevant solutions if necessary. Subsequently, these lower levels signal to higher levels which can then be updated themselves. As mentioned above, there can be several action-cycles (and thus also update-cycles) on *Level Three* for each *Level Two* and *Level One* action.

Hierarchical Agent Overview - Action Selection Phase

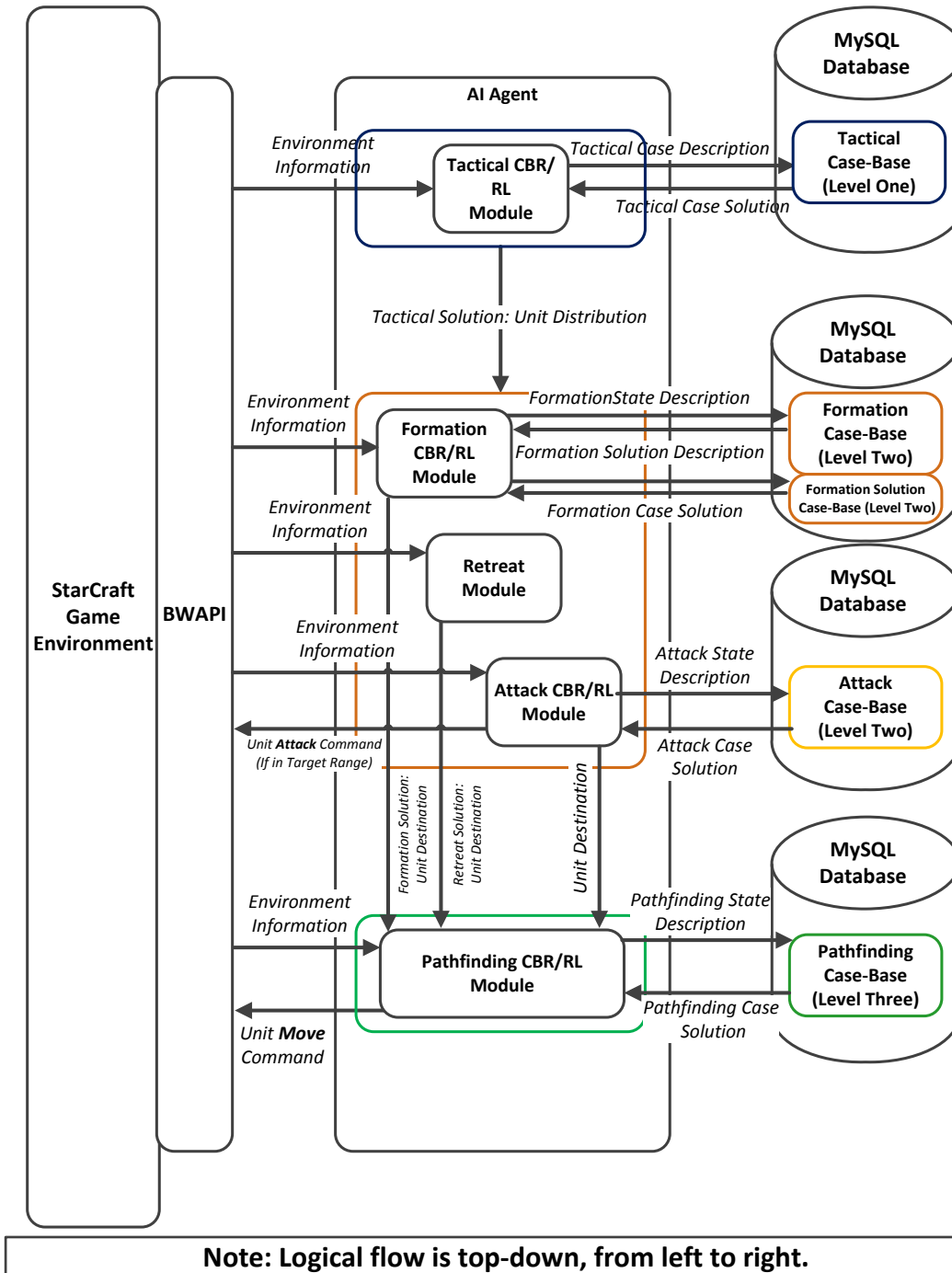


Figure 9.1: Action Selection using Hierarchical CBR/RL for Unit Micromanagement

Hierarchical Agent Overview - Action Termination and Reward Update

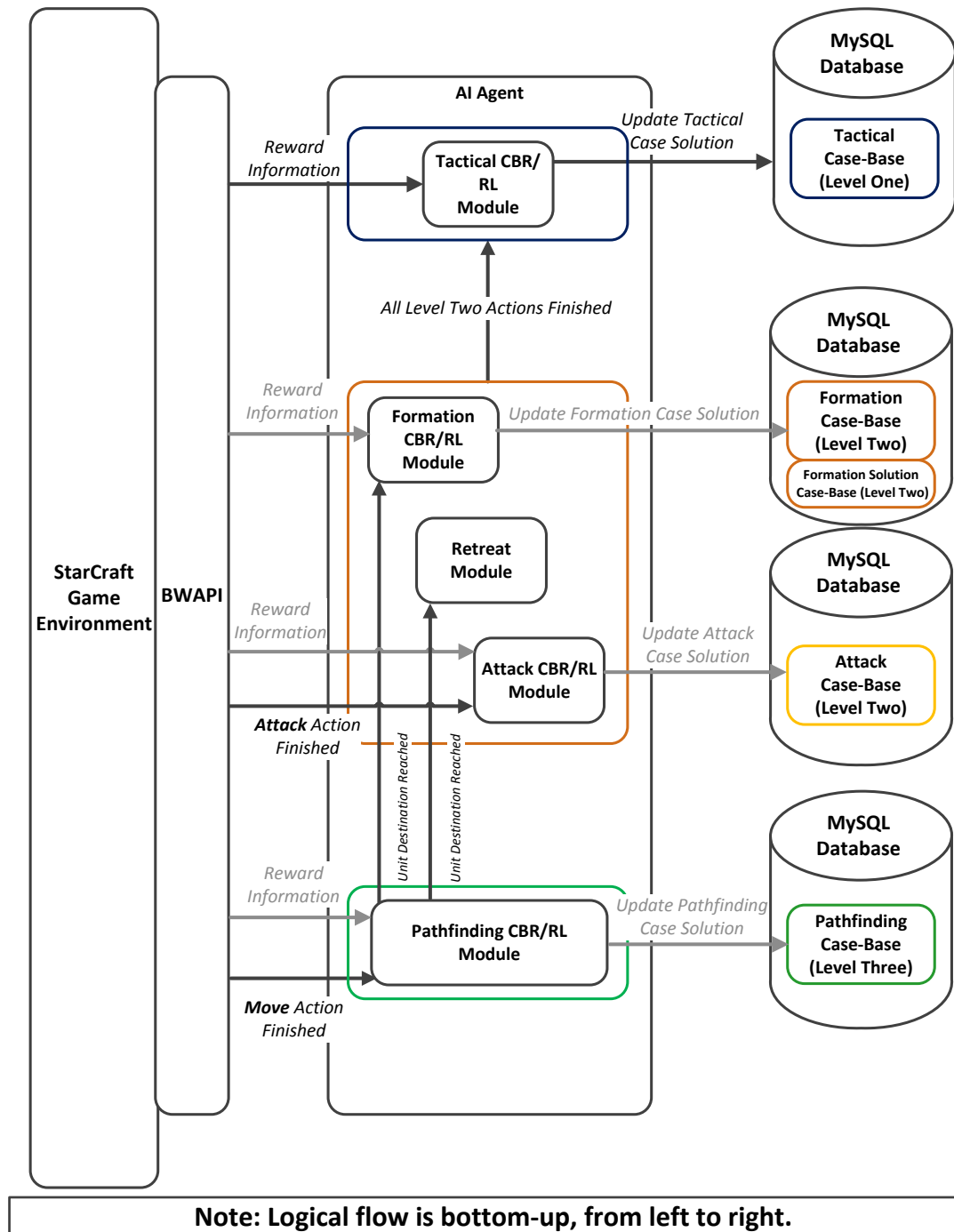


Figure 9.2: Reward Computation using Hierarchical CBR/RL for Unit Micromangement

9.3 Tactical Decision Making Evaluation

The evaluation for the *Tactical Unit Selection* action is similar to those of the *Attack* and *Formation* modules. However, because of the higher complexity of the problem, there are more features which have to be tested. As a result, more scenarios have to be tested. The evaluation of the *Tactical Unit Selection* module has to examine the performance of all features for each of the lower-level modules and combinations of those features. Another side-effect of the increased complexity is the number of episodes an evaluation takes. Depending on the choice of the similarity threshold ψ as well as the number of agent units, up to 100,000 training episodes are required. However, this is only the case for very complex scenarios with high similarity thresholds and a large number of agent units. Since the requirement for such a large number of training episodes can easily become prohibitive if a lot of different scenarios have to be tested or even more complex scenarios are required, the evaluation also includes analyses of the case-base behaviour during the relevant evaluation scenarios. This includes the measuring of case- and solution numbers as well as the number of unexplored solutions and average solution values. Using these analyses helps to decide the minimum similarity threshold ψ that is necessary to achieve optimal or near-optimal solution policies.

The scenarios used to evaluate the *Attack* and *Formation* modules only evaluated a single action in each episode and were designed to test very specific aspects of the respective actions. In contrast, the *Tactical Unit Selection* module plays ‘normal’ micromanagement scenarios in which the agent attempts to win the game by eliminating all opponents, a central aspect of general RTS and StarCraft gameplay. As a result, evaluation scenarios for this module are standard micromanagement combat situations without randomized triggers or unit spawning. Evaluating different values for fixed case-description attributes such as unit damage and type as well as unit numbers is done by using different types and numbers of units in different scenarios. Since games are played until one side wins, lower unit numbers are automatically covered in games with more units as units in those games are eliminated one-by-one. Variable attributes such as health, cooldown and unit positions are also covered as the game state changes during play.

Given these considerations, a number of representative micromanagement combat scenarios were created for the evaluation. Unit types are limited to standard non-flying units. Flying units are omitted since they do not have collision control, i.e. they can simply fly through each other. Since the *Level Two* and especially *Level Three* pathfinding case-bases were trained with units with collision-control, it is important that the scenarios for the *Tactical Unit Selection* module also use scenarios with only non-flying units. Another important factor regarding the *Formation* module is its limitation in terms of maximum controlled units. As the training and evaluation in Section 8.1.4 indicate, the *Formation* module is trained to

handle a maximum of six units due to exponential growth in the number of both cases and solutions when training the formation case-base for larger number of units. While this does not affect *Scenarios A-D*, *Scenario E* requires the agent to control ten units, which could potentially mean ten units are assigned to the *Formation* module. In order to avoid this undefined state, the algorithm implementation limits the agent to a maximum of six units assigned to the *Formation* module. The other units have to be assigned to *Attack* or *Retreat* actions.

The algorithmic parameters for CBR and RL are similar to those used successfully for evaluation and training of the *Attack* and *Formation* modules. The only change was based on a re-evaluation of the necessary similarity threshold ψ . ψ is expected to change due to a change of model and complexity of the problem. The problems addressed by the *Tactical Unit Selection* component are increasingly complex and the state representation is made up of a larger number of attributes per unit than any of the previous problems. Additionally, there are now more units involved in scenarios at this level, leading to an overall vastly bigger state-and-action space. As a result, the number of episodes that is required to learn a good policy has grown. If the similarity threshold is set to very high values of 90%+ to ensure that the optimal policy π^* is found, this further significantly increases the required learning episodes. Therefore, the first scenario that was used to determine the ideal similarity threshold, *Scenario A*, is a simple scenario where three ranged agent units compete against five melee opponents.

Ideally, determining the optimal threshold for a single scenario should mean that this threshold also works for other scenarios. This should be the case since the threshold is tied to the particular CBR/RL model used for representing the game environment for the *Tactical Unit Selection* module. This connection between environment model and similarity threshold was also observed in previous evaluations. However, to ensure this assumption is correct, a larger *Scenario B* where six melee agent units compete against a similar number of opponents was also tested with a range of different similarity thresholds. The results for *Scenarios A* and *B* were then compared to see if they were similar. Each set of experiments was repeated five times and results were averaged.

Chapter 9. Architecture Level One: Tactical Decision Making

Table 9.4 lists the resulting algorithmic parameters. Most are identical to those used in *Attack* and *Formation* evaluations, only the similarity threshold is set to the value determined above.

Parameter	Values
Scenario	A(3vs5), B(6vs6), C(5vs5), D(4vs9) , E(10vs10)
Number of Games	100 - 160,000
Algorithm	One-Step Q-learning
Case-Base Similarity Threshold ψ A, B	30% – 95%
Case-Base Similarity Threshold ψ C, D, E	85%
RL Learning Rate α	0.1
RL Discount Factor γ	0.8
RL Exploration Rate ϵ	0.8 - 0

Table 9.4: Tactical Decision Making Evaluation Parameters

Table 9.5 shows the army compositions for the five scenarios.















Scenario	Learning Episodes	Agent Ranged	Agent Melee	Opponent Ranged	Opponent Melee
A	100 - 100,000				
B	1,500 - 160,000				
C	15,000				
D	10,000				
E	50,000				

Table 9.5: Tactical Decision Making Evaluation Scenarios

The first two scenarios *A* and *B* are used to determine the ideal similarity threshold. Subsequently, three additional scenarios are run. *Scenario C* is used to evaluate agent performance when controlling a mix of unit types and pits 3 ranged and 2 melee units against 5 melee opponents. *Scenario D* evaluates how well an agent handles being severely out-

numbered and puts 4 slow agent units against 9 opponents. Finally *Scenario E* evaluates the possible boundaries of the current model in terms of unit numbers by setting 10 agent units against 10 opponents. Both groups are a mixture of ranged and melee type units.

In order to determine the optimal ψ , *Scenarios A* and *B* were run with a range of values for ψ between 30% – 95%. For both scenarios, $\psi = 30\%$ is the lowest discernible setting in terms of the number of distinct cases and solutions. This means that, at this setting, there is exactly a single case (and associated solutions) for each count of units. In *Scenario B* this means one case for six agent units against six opponents, one case for six agent units against five opponents etc., with the last case being for one agent unit against one opponent.

Given a particular threshold, the performance for the best possible policy π^* as well as the required number of explored case-solution pairs to achieve that performance are determined. Furthermore, the number of training episodes to achieve this optimal policy is decided. Determining these values is done in an iterative process by slowly extending the number of episodes that are played and testing for when a sufficient state-action space coverage for the given ψ is achieved. The analysis of the case-base statistics in the evaluation of the hybrid CBR/RL component in Section 5.3 and the development of the CBR/RL modules in Chapters 7 and 8 indicated that an exploration rate of 80% of the state-action space leads to optimal or near-optimal policies. Therefore, 80% was chosen as the target for state-action space coverage. For a given setting of ψ , the number of training episodes is increased in set increments until at least 80% of all state-solution pairs have been explored. At that point, the number of episodes it took to achieve 80% exploration as well as the number of cases and solutions at this point are recorded. Furthermore, a number of additional evaluation metrics are recorded, both to clarify the observed behaviour and to look for non-predicted side effects. The specific additional metrics are the average number of squad-level actions, as a percentage of the average overall actions during a particular game. The average duration in frames of evaluation episodes was also recorded.

Starting positions are always a random spread opposite each other and the map-size is 2048x2048 pixels, the smallest possible StarCraft map size. Similar to previous evaluations, every scenario was run five times and the results averaged.

9.4 Results

The first two scenarios were, as stated above, run with a number of different similarity thresholds ψ . Table 9.6 shows the results for *Scenario A*. Table 9.7 shows the results for *Scenario B*. The reward is normalised to a value between 0% and 100%. 0% is achieved in a game in which agent units are eliminated without doing any damage. 100% is a perfect game in which all opponents are eliminated without the agent units sustaining any damage. This is similar to what was done in the evaluation of Section 5 and allows to compare results of scenarios with different values for maximum and minimum rewards. The small negative reward component for the time that it takes for an action to finish is not part of the displayed 0% – 100% interval. However, due to its negligible size, this should not influence the overall display since it is only big enough to serve as a tie-breaker to favour quicker actions over slower ones. The number of episodes, which varies considerably between different scenarios and similarity thresholds, is also normalised to a 0% – 100% interval to make results comparable.

Threshold ψ	# Episodes	# Cases	# Solutions	# Actions	Max. % Reward
95%	100,000	2,376.4	18,853.0	47.32	92.48%
90%	60,000	1,265.2	9,976.4	45.27	87.33%
85%	20,000	366.8	2,570.2	41.15	82.93%
80%	8,000	192.0	1,299.6	41.06	81.82%
70%	1,500	52.6	293.4	35.43	70.16%
60%	800	39.2	224.6	30.96	60.73%
50%	500	29.2	156.2	23.7	52.79%
40%	300	17.6	95.8	11.49	33.35%
30%	100	13	69	10.36	25.47%

Table 9.6: Tactical Decision Making Evaluation Scenario A

Threshold ψ	# Episodes	# Cases	# Solutions	# Actions	Max. % Reward
95%	160,000	1570.4	31,201.6	7.10	78.15%
90%	75,000	699.8	12,339.0	6.92	75.13%
85%	30,000	324	5,324.20	6.96	73.01%
80%	15,000	259.8	3755.8	6.99	69.97%
70%	7,500	159.4	2092.6	7.45	63.09%
60%	5,000	95	1,309.2	8.72	57.99%
50%	3,000	63.2	801.4	9	55.19%
40%	2,000	47.2	631.4	9.5	45.53%
30%	1,500	38	517	10.18	43.44%

Table 9.7: Tactical Decision Making Evaluation Scenario B

Figures 9.3 and 9.4 display the associated development of the average reward per episode that the agent achieves. As results in both the tables and the diagrams show, similarity thresholds between 80% and 95% lead to results that are roughly within a 10% interval in terms of overall performance. However, the number of cases and, more importantly, the number of overall solutions increases significantly among the different thresholds. Therefore, it was decided to use a threshold of $\psi = 80\%$ for the subsequent evaluation scenarios. The next section, which discusses results in detail, also further explains this decision.

The case-base development for *Scenario A* shows a relatively linear relationship between the number of episodes required to achieve 80% coverage of the state-action space and the number of cases and solutions for a particular case. Case- and solution-numbers increase in a non-linear fashion with increasing ψ . Initially, for lower ψ , the increase in case-numbers is low and the number of solutions less than doubles for each 10% increase in ψ . For high ψ , the number of solutions often even doubles for 5% threshold increases. This behaviour is mirrored for *Scenario B*.

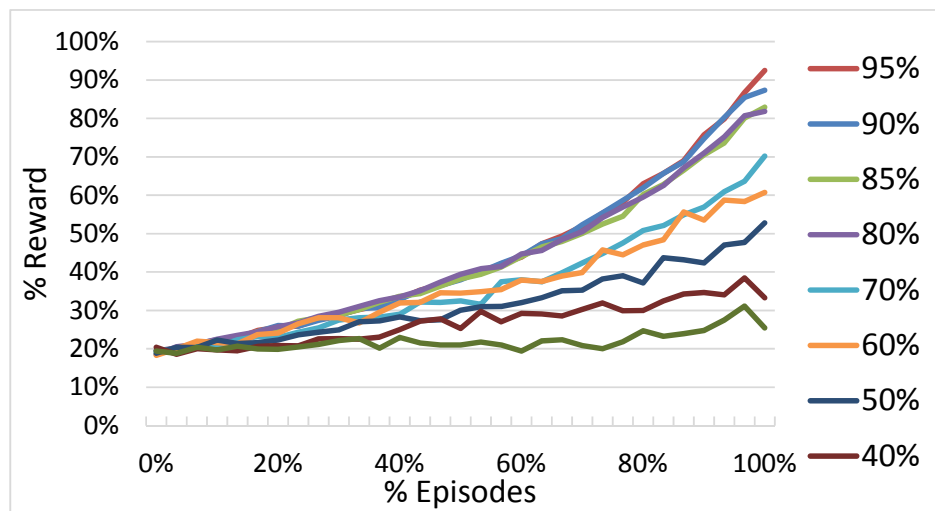


Figure 9.3: Performance Results for Scenario A for different ψ

After discovering a suitable similarity threshold of $\psi = 80\%$, the other three scenarios *C*, *D* and *E* are run using that threshold and similar algorithmic parameters as for *Scenarios A* and *B*. Given the results from the case-base analysis, the number of training episodes was set based on the number of agent units. The number of training episodes is set to 15,000 for *Scenario C*, 10,000 for *Scenario D* and 50,000 for *Scenario E*. These numbers might seem large when compared to other experimental evaluations in this thesis. On the one hand, this

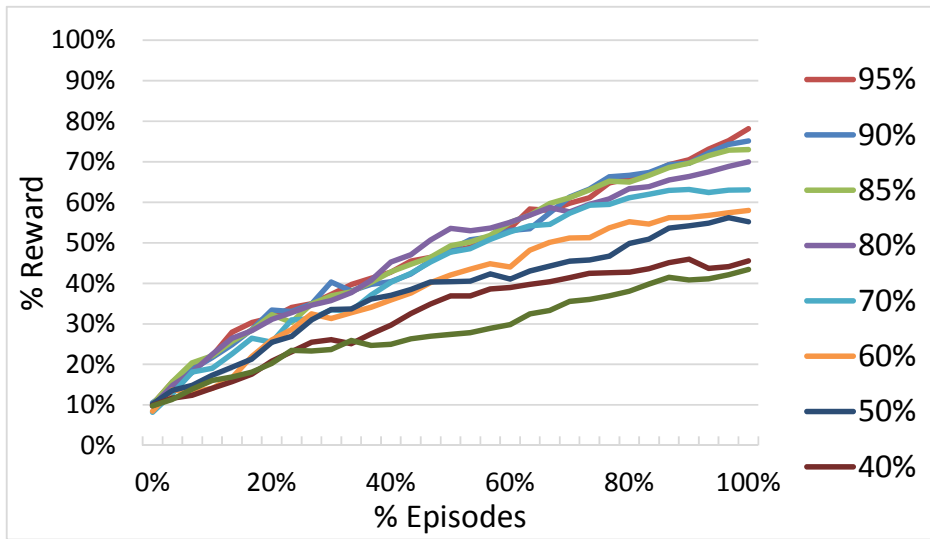


Figure 9.4: Performance Results for Scenario B for different ψ

high number is due to the target of exploring 80% of the case-solution space. This comparably high exploration rate was chosen to guarantee an optimal or near-optimal policy. While a less explored state-action space might often also result in a good policy, this is not guaranteed, as shown in the evaluation for lower thresholds ψ in Section 7.3. The other reason is the more complex model used in the *Tactical Unit Selection* module when compared to other CBR/RL modules. Especially for high unit numbers, this model leads to a significant increase in required evaluation episodes. Choosing the number of learning episodes for a scenario in this particular way also means that, unlike previous module evaluations, there is no additional evaluation of the speed of learning besides the one that is part of the evaluation of *Scenario A* and *B*.

The focus in the analysis of the results for Scenarios *C*, *D* and *E* is, on the one hand, performance. I.e. does the agent learn good policies for the particular setting the scenario in question addresses? On the other hand, the effectiveness of the hierarchical *Tactical Unit Selection* model is to be evaluated in terms of how the component re-uses lower-level *Attack*, *Retreat* and *Formation* actions, given a certain scenario. Analysing this decomposition also allows one to evaluate whether the agent manages to learn desired behaviour in terms of high-level strategies such as kiting, focus fire and in general prioritizing and learning valid ‘human-like’ strategies.

Table 9.8 shows a summary of the number and type of tasks per scenario for the best found policies, i.e. recorded at the end of the evaluation¹. Additionally, the *Average Duration* in frames is displayed. Figure 9.5 shows the development of the average reward obtained for all five scenarios. All scenarios were trained with a similarity threshold $\psi = 80\%$ and the number of training episodes as specified above for the respective scenarios.

Scenario	# Tactical Actions	# Attack Actions	# Formation Actions	# Retreat Actions	Average Duration in Frames
Scenario A	41.06	17.66	0.17	24.58	952.93
Scenario B	6.96	6.91	1.37	3.25	509.65
Scenario C	81.57	36.55	11.21	58.13	2747.05
Scenario D	18.01	10.41	1.34	8.47	568.01
Scenario E	137.58	30.68	9.27	126.55	3867.22

Table 9.8: Tactical Actions and Duration for all Scenarios

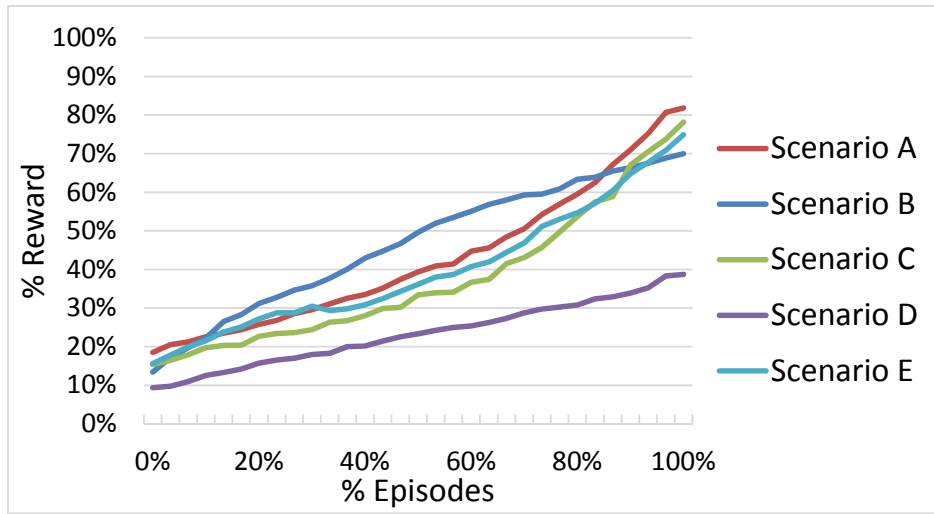


Figure 9.5: Performance Results for all Scenarios

The results in Figure 9.5 show that the hierarchical RL/CBR agent achieves a notable increase in average reward obtained for all five scenarios over the duration of their respective training runs. In terms of reward development, there is a difference between *Scenarios B* and *D* which use melee units only, and the other three scenarios. *Scenarios B* and *D* show an

¹ There can be one of each *Attack*, *Formation* and *Retreat* actions for a single tactical action, which is why the number of *Overall Actions* is usually less than the sum of the three individual *Level Two* action types.

almost linear reward development over the time their respective experiments run. *Scenarios A, C and E*, which all use both melee and ranged units, show reward development curves that are more similar to those encountered in previous evaluations. (See Sections 4.3 and 5.3.)

9.5 Discussion

In the first part of the evaluation, both *Scenarios A and B* were run with numerous different settings of ψ in order to determine the best setting for future experiments. The results in Tables 9.6 and 9.7 and Figures 9.3 and 9.4 led to the selection of $\psi = 80\%$ for subsequent experiments. The decision was mainly based on looking at the trade-off between the performance the agent achieves for a certain ψ and the number of solutions that has to be evaluated for that particular threshold. This would mean that the lowest thresholds produce the best ‘performance-per-solution’ ratios. Therefore, an additional subjective target of selecting a ψ that guarantees an ‘acceptable’ performance was added. Looking at both scenarios, this led to the exclusion of any $\psi < 80\%$, since these thresholds resulted in performance far inferior to that achieved when evaluating the identical *Scenario A* with the approach presented in Section 5. Finally, when considering the increase in solutions for each increase in ψ , 80% was selected as the threshold with the best trade-off while still allowing an optimal performance.

Scenarios A and B have about ten *Tactical Unit Selection* actions (i.e. *Attack*, *Formation* or *Retreat*) in an average episode for the lowest, worst-performing setting of $\psi = 30\%$ where there is only a single case for each agent-opponent unit number combination. For higher thresholds, which allow for a more optimized performance, the number of actions diverges significantly. For *Scenario A*, the number of *Tactical Unit Selection* actions exceeds 40 for $\psi \geq 80\%$. The reason for this is the learned hit-and-run strategy that performs best for the units in this particular scenario and which requires extensive use of *Retreat* actions. Lower similarity thresholds mean there is not enough distinction between inherently different cases, which in turn does not allow the agent to learn and effectively execute this hit-and-run strategy. The melee-unit-focused *Scenario B* teaches the agent a fundamentally different strategy, indicated by the average number of *Tactical Unit Selection* actions. For $\psi \geq 70\%$, the average number of actions per game is below nine. This is due to the main strategy in this scenario, which is based on focusing attacks (covered by the *Attack* action) combined with minimal regrouping or retreating through *Formation* or *Retreat* actions. There is no use for extensive *Retreat* patterns since opponent- and agent unit types are identical, which means hit-and-run style attacks are useless. The comparison between distributions of *Tactical Unit Selection* actions for all scenarios is shown in Table 9.8 and is analysed below.

The fact that agent and opponent use identical melee units in *Scenario B* also explains the difference in overall maximum rewards achieved. While the hit-and-run strategy allows

the agent to achieve perfect or near-perfect rewards of more than 90% for *Scenario A*, the average reward in *Scenario B* reaches a maximum value of just below 80%. This is because attacking melee units with other melee units will always lead to suffering a certain amount of damage. However, this is the only possible way of combat in *Scenario B*. The low number of actions required for optimal performance in *Scenario B* also means that it is easier to achieve good results in terms of average reward by using random untried solutions.

The number of possible solutions for a particular tactical case is always directly dependent on the number of agent units in the game state that the case abstracts. Each additional case automatically adds all possible solutions to that particular game state. This explains why at the highest $\psi = 95\%$ there are 2,376 cases with 18,853 solutions for *Scenario A*, while there are only 1,570 case but with 31,201 solutions for *Scenario B*. *Scenario B* is based on six initial agent units and thus has more high-unit cases than *Scenario A* with only three initial units. This also means that having **fewer** cases for a scenario with **more** units, results from the case-space being less explored. This corroborates the observation that the optimal strategy in *Scenario B* is based on mostly straight-out attacking the opponent and little exploring of the spatial surroundings through *Retreat* or *Formation* actions, as indicated by the analysis of the *Tactical Unit Selection* actions.

As mentioned above, the number of episodes required to sufficiently explore a certain scenario is mostly tied to the number of case-solution pairs that have to be explored. However, the number of *Tactical Unit Selection* actions per episode also plays a role, although less prominently than could be expected. Since a single episode in *Scenario A* has on average more than four to five times as many *Tactical Unit Selection* actions for $\psi \geq 60\%$ when compared to *Scenario B*, less episodes should be required to explore *Scenario A*. As Tables 9.6 and 9.7 show, *Scenario A* does require fewer episodes, although this is more evident for lower ψ . For $30\% \leq \psi \leq 70\%$, there are more cases for *Scenario B* due to the larger number of agent units. For $\psi \geq 80\%$, the number of episodes required to explore each scenario is more comparable and, if taking into account the number of case-solution pairs to explore, is more or less identical to or even in favour of *Scenario A*. The fact that not still more episodes are required is due to the use of an ϵ -greedy exploration policy which leads to the same case-solution pairs being explored more frequently as the games progress. While the average number of *Tactical Unit Selection* actions increases throughout an experiment, the biggest increase is only towards the end, where the almost entirely greedy policy at that point selects the same solutions again and again with little to no exploration. Looking at an entire experiment, the difference in distinct state-action (case-solution) pairs between *Scenario A* and *Scenario B* is far less noticeable than Tables 9.6 and 9.7 suggest.

A comparison between the performance results for *Scenario A* in this section and the results of the same scenario in Section 5 (see Figure 5.6, results for *Scenario B*) shows that

results for CBR/RL only can be replicated by the hierarchical CBR/RL agent when using the highest similarity threshold $\psi = 95\%$. However, due to the significantly more complex model used for the hierarchical agent this requires experiments with 100,000 episodes of learning compared to only 1,000 episodes for *Scenario B* in Chapter 5. The selected threshold of $\psi = 80\%$ leads to slightly worse performance in terms of average reward but still manages to find near-optimal policies and is much faster in terms of learning speed. While it is still not as fast as in Chapter 5, the fact that it can address scenarios with much higher complexity makes the lack of speed acceptable.

In all scenarios, the AI agent manages to obtain a significant improvement in the average reward. For all army compositions in the different scenarios, the agent finds optimal or near-optimal policies. *Scenario A* is the only scenario where the army composition theoretically allows a ‘perfect game’, i.e. eliminating all enemy units without sustaining damage. The agent manages to obtain more than 80% average reward in this scenario. In *Scenarios C* and *E*, which both contain melee units that are harder to manage and are basically guaranteed to sustain damage when they attack, the agent manages to obtain above 75% of the maximum possible reward. Even in *Scenario D*, which only uses melee units, the agent reaches nearly 70% of the possible reward, pointing to effective use of focus-fire and manoeuvring.

As expected, *Scenario D* shows the worst performance in terms of average reward. However, despite being outmatched by more than twice as many opponents, the agent units still manage to improve their performance significantly. After learning for 10,000 episodes, the average reward improves from the initial 10% to nearly 40%. This is a notable improvement since, in order to offset the negative reward for losing all its own units, the agent must eliminate more than half of the opponents, simply by using a combination of *Attack* and *Retreat* actions.

The fact that this performance is possible at all shows that the actions which were defined for *Level Two* of the hierarchical architecture are used successfully by the *Tactical Unit Selection* module for micromanaging groups of units. Furthermore, this evaluation of the *Tactical Unit Selection* component also serves as an evaluation of the successful transfer of *Attack* and *Formation* action knowledge from training scenarios to full combat scenarios. The quality of the results indicates that the training of the case-bases for the *Attack* and *Formation* modules that were run after their respective evaluations was successful. For all encountered *Attack* or *Formation* situations, cases which contained suitable solutions were created during the training phases described in Sections 8.1.4 and 8.2.4.

When comparing the reward development of the different scenarios as depicted in Figure 9.5, there is a difference between *Scenarios B* and *D* which use only melee units and the other three scenarios. This directly reflects the ideal behaviours in those scenarios and how these behaviours are reflected in action-selection policies in the RL model. Optimal behaviour in

a given scenario depends both on the layout of the scenario and on the agent and opponent army compositions.

The general rules are as follows.

- If the agent has agile, ranged units, then hit-and-run strategies, the so-called ‘kiting’, are the best possible strategies. Emphasis in terms of reward is on survival over damage.
- If the opponent either has equally or more agile units (ranged or melee), direct attack and focusing fire are the best possible strategies. The emphasis here is on dealing as much damage as possible, survival through *Retreat* only plays a secondary role.

These strategies are reflected in the composition of the *Level Two* actions shown in Table 9.8. For both *Scenarios B* and *D* the optimal learned policies include few overall *Tactical* actions with the focus being on *Attack* actions. While there are some *Retreat* actions and a single *Formation* action per experiment on average, these two squad-level action types are used far less than in the other scenarios. The average duration of the games also reflects that the focus is not (and cannot be, due to the unit types involved) on hit-and-run strategies. The few non-attack actions are likely when agent units’ weapons are in cooldown.

Scenarios A, C and *E*, which use some ranged agile units or even only ranged agile units in the case of *Scenario A*, show a different composition of tactical actions. Each of these scenarios is significantly longer in terms of duration, with more units leading both to longer games and to more actions taken. Experiments run using *Scenario E* are on average nearly eight times as long as experiments using *Scenario B*. Furthermore, these scenarios have an exponentially growing number of *Retreat* actions the longer they run, indicating the learning of hit-and-run strategies. *Scenario E* has fewer *Attack* actions than the shorter *Scenario C*, possibly because of the larger number of units involved in *Scenario E* which, ideally, can eliminate opponent units quicker through focus fire. Another contributing factor is the limitation of *Formation* actions to a maximum of six units. Units that cannot be assigned to a formation or where it does not make sense to attack due to weapons cooldown or other factors are thus simple retreated.

Given the design of the *Formation* actions as presented in Section 8.1, scenarios in which the CBR/RL agent uses only ranged or only melee units (*Scenarios A, B* and *D*) should gain the least from using this action. In these scenarios, *Formation* can only be used to protect damaged units and not to encircle ranged units with melee units. This assumption is confirmed by the numbers in Table 9.8 which indicate that *Formation* actions are seldom used in optimal policies in *Scenarios B* and *D* and rarely if at all in *Scenario A*. The complete lack of use in *Scenario A* can be explained by the low number of units that the agent controls there. With only three units, the formation layout does not even have a protectable core even with all units making up the formation. Furthermore, *Scenario A* only uses ranged units that

excel at hit-and-run. This means using the *Formation* action to simply regroup units gains less than from scenarios which put more emphasis on focus fire.

Overall, the results presented in the previous section and discussed in this section show that the hierarchical CBR/RL agent successfully learns the micromanagement tasks it was built to solve. The agent learns near-optimal policies in all evaluated scenarios which cover a broad range of in-game situations. The model that was created to abstract the game world to manageable yet sufficiently detailed levels succeeded in creating an appropriate, workable representation. The agent successfully re-uses the *Level Two* modules created for the squad-level tasks and the knowledge stored while training these modules. The *Level Two* components in turn use the *Navigation* component and knowledge on *Level Three* of the architecture. The composition of *Level Two* actions that are used shows that the learned behaviour resembles tactical strategies that, according to anecdotal expert knowledge, a human would choose in the particular situations the respective scenarios simulated.

9.6 Knowledge Transfer between Scenarios

One final aspect to be evaluated is how well acquired knowledge transfers from one scenario to another. This evaluation of knowledge transfer is similar in concept to the training phases of the case-bases for *Attack* and *Formation* actions described in Section 8. These training phases attempted to cover any potentially arising situations for the respective tasks by training a limited number of representative scenarios. The successful evaluation of the *Tactical Unit Selection* module that re-used the knowledge proved that the case-bases for the *Level Two* modules contained enough situational knowledge to solve all *Attack*, *Formation* and *Navigation* problems which arose with a high level of accuracy. Since there is no higher-level evaluation that could re-use (and thus evaluate the quality of) the knowledge in the *Tactical Unit Selection* case-base in different scenarios, it is instead done in specifically designed experiments.

When running experiments in a particular scenario with a certain number of particular units, any scenario that uses a subset of these units is automatically covered in the many different episodes which are played to evaluate the larger scenario. This is true if there is a sufficient coverage of the state-action space. Since the state-action space target coverage in the previous evaluation is 80%, there is a risk that the state-action pairs that are relevant for a particular subset of the given scenario are not fully explored. However, re-using the case-base that was trained on the larger scenario to solve the smaller scenario should generally lead to results that are comparable to training the subset scenario from scratch.

Taking the concept of knowledge transfer one step further, if a case-base that was previously trained on a related scenario is re-used, this should lead to a higher speed of convergence to

an optimal policy in a new scenario. Both of these concepts are tested in this section. The case-base trained for *Scenario E* serves as the knowledge base for both tests. This scenario is the most comprehensive in terms of unit numbers and types. Table 9.9 lists the number of cases in the *Scenario E* case-base, ordered by the agent’s own and opposing unit numbers. These two variables are chosen since they form the indices for the case-bases and there is no re-use among cases with differing unit numbers.

		<i>Agent Units</i>									
		1	2	3	4	5	6	7	8	9	10
Opponent	1	9	16	11	7	6	5	5	4	0	0
	2	8	13	17	11	10	6	5	5	3	3
	3	9	15	18	13	14	10	9	4	3	2
	4	9	18	14	17	12	12	10	10	5	3
	5	8	16	17	15	14	13	9	11	7	6
	6	9	16	17	18	15	15	12	11	8	8
	7	9	17	19	19	17	14	13	12	12	11
	8	8	15	18	13	11	16	15	13	13	13
	9	7	16	17	14	16	14	17	11	13	14
	10	8	12	15	14	13	14	13	12	11	12

Table 9.9: Cases per Agent and Opponent Unit Numbers for *Scenario E* for $\psi = 80\%$

Since *Scenario C* is basically a subset of *Scenario E*, the first experiment is to use the *Scenario E* case-base without further modification to play *Scenario C* and compare the performance to the original *Scenario C* results. Table 9.10 lists the unit indices for the original *Scenario C* case-base. As a comparison between Table 9.10 and the relevant partition in Table 9.9 (marked through black outline) shows, the raw number of cases concerning situations with 5vs5 and fewer units is roughly similar. The case-base for *Scenario C* has roughly 10% more cases than the one for partition of *Scenario E*. While this says nothing about the actual case descriptions, the roughly equal case-solution coverage of about 75% for both scenarios indicates that a large number of similar cases is contained in both case-bases.

		<i>Agent Units</i>				
		1	2	3	4	5
Opponent	1	9	13	14	16	14
	2	7	13	19	18	18
	3	7	15	16	18	20
	4	8	16	17	17	20
	5	8	10	13	14	14

Table 9.10: Cases per Unit Numbers for *Scenario C*

The results between original *Scenario C* and re-using knowledge gained in *Scenario E* are also very similar in terms of performance. Re-using the case-base trained in *Scenario E* to

Chapter 9. Architecture Level One: Tactical Decision Making

play *Scenario C* leads to an average reward of 72.85% when using a purely greedy policy with a win ratio of 83.71%. The original results were 78.18% average reward (see Figure 9.5) and a win ratio of 89.21%. These results confirm an assumption that was made during earlier evaluations: scenarios automatically learn knowledge necessary to solve any sub-scenarios.

The second concept to be evaluated is how well re-using knowledge from one scenario works in a similar scenario and how this re-use affects the learning behaviour. A new *Scenario F* is created that is similar to *Scenario C* in that it uses a small number of both ranged and melee units. However, the unit types and numbers are slightly different: there are now two ranged units and three melee units for agent and opponent. The different unit types leads to more resilient, more powerful but slower units. The average reward performance in that scenario over 1,500, 6,000 and 10,000 episodes of learning is compared both with and without the knowledge from the *Scenario E* case-base. Each scenario is once again run five times using the previously-used algorithmic parameters and an ϵ -greedy exploration policy. The results of all five runs are averaged. For scenarios that re-use the knowledge contained in the *Scenario E* case-base, the case-base is reset to its original post-*Scenario E*-state after each run. During the run, additional cases can be added and existing case-solution fitness values can be updated.

Table 9.11 lists the configurations of the three scenarios *C*, *E* and *F* relevant to this section.












Scenario	Learning Episodes	Agent Ranged	Agent Melee	Opponent Ranged	Opponent Melee
C	15,000				
E	50,000				
F	1,000 - 10,000				

Table 9.11: Knowledge Transfer Evaluation Scenarios

The results of running *Scenario F* with experiments of three different lengths, with and without existing knowledge are shown in Figure 9.6.

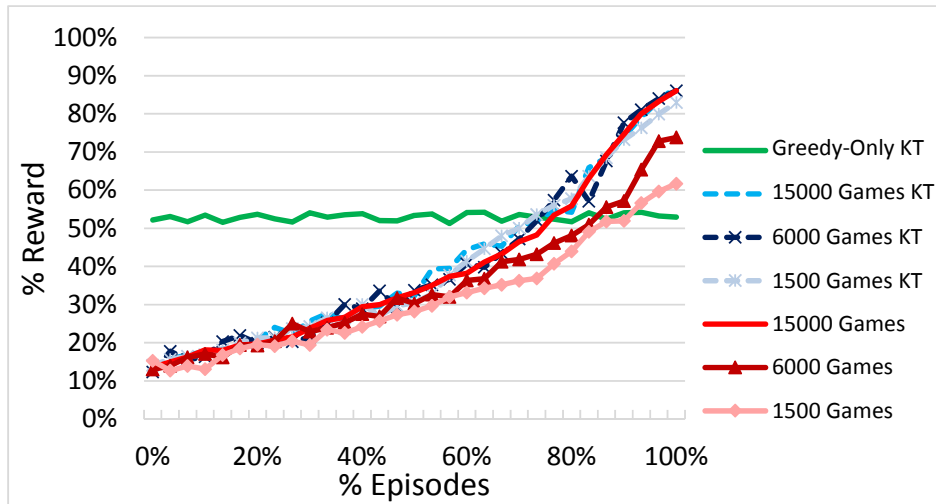


Figure 9.6: Performance Results for *Scenario F* without and with existing knowledge (KT = Knowledge Transfer, i.e. with prior knowledge)

The diagram shows that all experiments that re-use the *Scenario E* case-base, regardless of the length of the experiment, achieve roughly the same performance of about 85% of the maximum possible score. This is similar to the results achieved by the experiments 15,000 episodes long with no prior knowledge. Therefore, it can be assumed that this is the optimal result that can be achieved given this particular scenario, model and set of algorithmic parameters. Experiments without prior knowledge that run 1,500 and 6,000 games achieve about 60% and 75% of the maximum score respectively. As the diagram shows, using the *Scenario E* case-base for *Scenario F* without further training leads to scores of just above 50%. In comparison, a purely random action selection policy - which was not pictured in order to not overload the diagram - leads to about 15% of the maximum reward.

The results make it clear that using a pre-trained case-base from a closely related scenario can lead to a significant increase in speed of convergence to an optimal policy. A completely empty case-base requires about 15,000 games to achieve optimal results. When using existing knowledge, even experiments that have only 1,500 episodes to find optimality achieve the best possible outcome. However, achieving optimal results by simply re-using existing knowledge without any adjustment is not possible as the completely greedy baseline shows. There have to be some learning episodes to adjust to the new units. Yet, since the scenario is only slightly

different in terms of unit types, learning the required missing cases can be achieved in much less time than learning the entire scenario from scratch.

The results presented in this section show that transferring knowledge between different but related scenarios works well. The knowledge learned for one scenario is not specific to that particular scenario but generalises well to related situations. This implies that the architecture, model and learning procedure work well for creating such generalisable knowledge that, in future uses and extensions of the hierarchical architecture, can be employed in other scenarios in the same domain and, potentially, even in related problem areas.

This chapter presented and evaluated the highest level of the architecture. The results both in terms of overall performance and the agent's ability to transfer its learned knowledge were discussed. The next chapter discusses the overall implications of the approach created in this thesis. All three layers in the context of the overall architecture are analysed. The precursor RL and the CBR/RL modules and their implications for the eventual architecture are examined. Furthermore, high-level implications in the context of developing adaptive human-like AI agents are examined.

Chapter 10

Discussion and Future Work

This chapter discusses relevant findings and observations on a larger scale; in the context of the overall aim of the thesis, the creation of a learning agent that manages to acquire all knowledge necessary to address reactive and tactical layers in RTS game AI. This is done by drawing on the findings listed in the discussions relating to individual results of the different parts and modules in the respective Sections 4.3.2, 5.4, 7.4, 8.1.6, 8.2.3 and 9.5. Those findings are analysed as parts of the hierarchical CBR/RL framework described in this thesis. Furthermore the extent of their contributions to the overall research questions in Chapter 1 is debated. This discussion leads to conclusions on both the achievements and the limitations of the approaches that were developed. All of this is done in the context of existing work in the area of RTS game AI research, emphasising the contributions of this thesis. Furthermore, possible future work that can extend this thesis and overcome the discovered limitations is considered.

Initial RL and Hybrid CBR/RL Findings Chapter 4 vetted several prominent temporal-difference RL algorithms for an application for micromanagement in strategy games. As part of this evaluation, a simple AI agent is created and implemented in StarCraft. It is shown this agent can beat its opponents by learning a hit-and-run strategy, the optimal type of behaviour for the selected evaluation scenarios. All tested RL algorithms are shown to be suitable for learning the task chosen for the evaluation. While each of the four tested RL algorithms achieves very good results, one-step Q-learning performs the strongest. The algorithms using eligibility traces show a faster speed of convergence yet a slightly worse performance. Given these findings, the CBR/RL module in Chapter 6 used Watkins's $Q(\lambda)$ because of its performance while subsequent CBR/RL modules used one-step Q-learning both due to its performance and the ease of implementation.

The second step in this thesis is the integration of CBR with the initially-devised RL component. This integration is necessary due to the considerable limitations in terms of problem space that simple table-based RL has. Additionally, the model used by the RL agent is adapted to the new algorithmic approach. More complex state representations that

include spatial information in the shape of influence maps are used and the agent has more varied actions at its disposal. This change also means that the model moves away from hard-coded expert knowledge and towards more atomic actions which are combined in varied ways to recreate high-level actions. While the *Move* actions are atomic, the *Attack* action still builds considerably on expert knowledge by selecting the opponent with the lowest health within a given radius.

The hybrid CBR/RL module created in Chapter 5 also introduced the notion of hierarchically interconnected case-bases in order to abstract the state space. However, having an architecture with two layers of hierarchically interconnected case-bases, where each case on the higher level is linked to non-overlapping sets of cases on the lower layer, led to an agent that was unable to achieve a sufficient state-action-space coverage to learn good policies. As a result, the number of cases on the higher level was set to one per number of agents, thus essentially removing the relevance of the higher level. While using a two-level hierarchy of case-base effectively failed for this module, the expertise gained during the evaluation was important for the subsequent creation of the hierarchical architecture in Chapter 6. Two other insights from Chapter 5 that heavily influenced future decisions were the optimisation of the Q-learning parameters and the analysis of the case-base statistics. Previous Q-learning parameters had been chosen based on prior RL applications in Wender & Watson (2008) as well as considerations about the desired learning process and commonly used values (R. S. Sutton & Barto, 1998). The experimental optimisation led to slightly different settings for the key RL parameters α and γ . Additionally, the analysis of the case-bases led to more insights on ideal similarity thresholds ψ and required state-action space coverage to guarantee a good policy π . Having a sufficient coverage is crucial for any RL-based approach, since the theoretical guarantee for finding the optimal policy π^* depends on it. However, in real-world problems that use an ϵ -greedy exploration policy, it is basically impossible to achieve 100% coverage for many ($\gg 1,000$) state-action pairs. Therefore, it is important to know which level of coverage will lead to ‘good’ policies. In terms of game score performance, the results of the hybrid CBR/RL module show that the agent can learn the desired behaviour of managing multiple units in combat scenarios. This is in contrast to just a single unit that was managed by the initial RL approach. The scenarios were kept comparably simple since the higher level case-base couldn’t be used effectively, with more complex scenarios being delegated to the hierarchical approach developed in subsequent chapters.

Layered Architecture and Extensibility After the preparative steps and evaluations in the two previous chapters, Chapter 6 describes how these prior discoveries are used to design the hierarchical architecture that addresses the reactive and tactical tasks involved in RTS games. The architecture is similar to other hierarchical multi-scale approaches devised for RTS game agents such as those described in other research (Ontañón et al., 2013; Weber, 2012; Synnaeve & Bessiere, 2012) as the decomposition of the overall RTS problem into layers of abstraction is the most common approach. However, it also draws inspiration from the layered learning paradigm (Stone, 1998), previously described in Section 3.5, in that it translates an agent’s commands through multiple layers of reasoning into concrete game actions. Learning on each level is influenced by the components on the next lower level.

There are two major factors that set it apart from other existing hierarchical architectures which are used in RTS games (Ontañón et al., 2013) or related fields such as robot soccer (Behnke & Stückler, 2008). The first major factor is the homogeneity of the approach in that it applies the same CBR/RL ML technique to all modules and layers. This is an advantage in the implementation, where state representations can be shared among different modules that only differ in details. Furthermore, higher-level reward signals can be composed as composites of lower-level ones. Using multiple, interrelated CBR modules also means that the data stored in a database can be re-used across multiple case-bases, leading to more efficient storage and retrieval, a factor that was also utilised in the implementation of this thesis (see Appendix A for the database diagrams of the case-bases in the hierarchical modules). The second factor is the modularity that makes it easy to adapt to changing requirements. This is combined with a strict hierarchy where higher layers control more units or more aspects of a single unit. In contrast, other RTS agents are mostly clusters of components which are loosely organized in a hierarchical manner (Synnaeve & Bessiere, 2012; Hagelbäck, 2012).

Additional tasks that could arise in an RTS at the reactive or tactical level can readily be assigned to a new module which can be integrated into the architecture. For instance, *Special Unit Abilities* have already been identified in Section 6.1 as being part of RTS games but so far have not been addressed in the architecture due to their high requirements in terms of customisation. However, it is easy to see how any special ability could simply be added as another module either on *Level Two* or *Level Three*, depending on whether or not it requires coordination between multiple units. The architecture can also easily be adapted to fit a related problem area such as robot soccer. Even in its current form, simply by redefining the existing modules for formations and attacks opens up most basic activities which are part of that domain. Pathfinding in robotic soccer is related to RTS game navigation, making an adaptation of *Level Three* in the architecture simple. A specialised player such as the keeper could be controlled by an additional module with the *Tactical Unit Selection* layer coordinating the different robot actors.

Hierarchical Architecture Modules - Compromises and Contributions One major restricting condition which was introduced to curb complexity in the implementation of the hierarchical architecture is limiting *Attack* and *Formation* to a single action for all units assigned to the appropriate category on *Level One*. This was done to avoid a combinatorial explosion of possible solutions. This condition has serious effects on the application of the architecture to full micromanagement in RTS games such as StarCraft. It is inconceivable that a player only chooses one target at a time for its units or only creates one formation at a time if the player controls more than a dozen units. The evaluation of the hierarchical architecture showed that for the tested scenarios, the implementation achieved good to very good results on all occasions. However, it could already be observed that the performance suffered slightly for bigger scenarios when compared to the excellent results in scenarios with fewer units. While this is probably also related to a lesser state-action-space coverage, the fact that the agent cannot choose multiple targets or formations at the same time almost certainly plays a role. Furthermore, even the biggest tested scenario with ten units for player and opponent would only count as a medium-sized skirmish in a full game of StarCraft. There are two ways to extend the hierarchical architecture to overcome this limitation. The first possibility would be to introduce another level above the currently highest *Level One*. The additional level would then simply perform a pre-allocation of all available units among several *Tactical Unit Selection* modules. Alternatively, the existing *Tactical Unit Selection* module could be extended to allow for several simultaneous *Attack* and *Formation* actions. This would lead to the aforementioned exponential growth in possible solutions with growing numbers of units. However, that could possibly be overcome by using a solution case-base modelled along the lines of the one used for *Formation* solutions.

The use of a solution-case base in the *Formation* module proved to be successful in reducing the number of possible solutions while maintaining good performance. This solution case-base was inspired by the approach that Molineaux et al. (2008) used to address a continuous state- and action-space in RTS game scenarios. The authors' agents learn the best strategies to move groups of units on a map and thus gain map control. The secondary *Value* case-base described in Molineaux et al. (2008) models the value of a state based on its reward signal. This serves to discretize the continuous value space and is used to create a model which in turn helps to select the optimal next action. The *Formation* solution case-base, on the other hand, generalises over a large number of discrete solutions. The prior RL-based solution selection is used as case-description and a customised pre-computed similarity metric based on formation layouts determines when new cases are created. Since the case-base only uses static information and there is no dynamic update of solution fitness, it would theoretically be possible to compute all cases necessary for the optimal coverage, given a chosen similarity threshold. However, since the pre-computation of formation similarities has already been

proved to be prohibitively computationally expensive even for medium unit numbers, a more effective way of doing this would have to be devised first. Ideally, there would be a dynamic adaptation phase (as in the standard CBR-cycle (Aamodt & Plaza, 1994)) which makes the static pre-computation redundant and allows easy adaptation to larger scenarios.

In general, the usefulness of the *Formation* module suffers when it is assigned a low number of units, much more so than any other module. This effect is further amplified in the final evaluation in Chapter 9 as the *Formation* module is only trained for formations that contain up to six units because of the complex case descriptions. Looking only at scenarios with few units is a significant disadvantage since unit formations evolve their full potential in larger groups of units where more intricate positioning is possible. In fact, anecdotal observation of the agent’s performance in the evaluation scenarios showed that even in scenarios where the optimal strategies included a number of *Formation* actions (see Table 9.8), these were frequently ‘abused’ by the AI agent as a ‘regroup’-like action. This was especially noticeable in scenarios using only melee units. While it is interesting and desirable for the agent to devise novel, unpredicted strategies using the actions at its disposal, this also points to the problem that *Formation* actions only make sense either for groups of specialised, distinctively different units or for larger groups which can easily protect their core. This observation is confirmed in the results for *Scenario A* in Table 9.8. In the optimal strategies for this scenario, *Formation* actions are basically non-existent. This is because agent units in this scenario are all the same ranged, agile type which profit significantly from individual ‘kiting’ actions but not at all from spatial positioning without resilient units on the outside forming a shield. Scenarios with a mix of units, and especially larger scenarios, use more formations. Having more large-scale scenarios would thus increase the usefulness of the *Formation* module, provided it can acquire the necessary knowledge to handle more than six formation units at a time. In addition to this factor, formations do not offer directly tangible benefits in combat situations, a characteristic that led to the complex and targeted composition of the *Formation* evaluation presented in Section 8.1.4. These factors taken together indicate why the task of creating effective unit formations is one of the less explored problems in RTS game AI-related research. Notable exceptions such as Van Der Heijden et al. (2008) and Lin & Ting (2011) often focus more on the dynamics of the formations themselves rather than on concrete, discernible contributions to the resulting overall performance.

While the *Formation* module requires larger scenarios to show its full potential, the current *Attack* module profits the most from smaller scenario sizes. The limitation to just a single *Level Two Attack* action at a time means that only one target can be attacked at any one time. When including spatial dispersion and other unit characteristics, it usually does not make sense to assign more than a few units to the *Attack* action. As such, the module would profit most from an extension that allows several *Attack* actions at once. Given the relative

simplicity of the *Attack* action when compared to the *Formation* module, such an extension would be easier as well.

Having the *Attack* action as a separate module which is responsible only for finding the best target sets it apart from the common approach of integrating the attack with other actions such as movement or retreat (Szczepański & Aamodt, 2009; Wender, 2009; Uriarte & Ontañón, 2012). The approach by Gunnerud (2009) is similar to the *Attack* module in that the aim is to select ideal combat targets. However, it differs in that Gunnerud (2009) executes an entire battle scenario and subsequently evaluates how effective the choice of targets was. This learning strategy is also dictated through the authors' choice of testbed, since the interface to the utilised RTS game, *Warcraft 3*, does not allow easy direct online learning.

The combinatorial problem arising from freely assigning attacking units to targets could be solved either by an additional case-base in the *Attack* module, a hard-coded decision-module based on expert knowledge, or by simply having the option to subdivide the attacking units into more than one *Attack* action on *Level One*. The last option would then also require a redesign of the *Tactical Unit Selection* module in terms of case description and reward computation, similar to what would be necessary to allow more than one formation at once.

Agent Performance One of the recurring issues that is addressed in the implementation of the different modules of the hierarchical architecture is the mapping between units in the current game and units that are part of case descriptions in the various case-bases. The procedure used for the mapping is based on the *Kuhn-Munkres algorithm* and is described in Section 6.3. For the *Attack* and *Formation* modules, comparing detailed unit similarities is already a major part of the case similarity computation. Therefore, it is not surprising that the successful learning in general and the quality of the performance results in particular show that stored units are consistently mapped correctly to match in-game units. Another contributing factor for the good performance of these modules is the rather small size of the evaluation scenarios.

The most conclusive sign of successful mapping is the performance of the *Tactical Unit Selection* module. This is despite the performance of the *Tactical Unit Selection* module being slightly worse than the previous modules in Chapter 5.1. This can be attributed to the larger scenario sizes for the hierarchical architecture and also to added variability through the re-use of knowledge stored in lower-level CBR/RL modules. Because of the larger number of controlled units in the scenarios and the resulting larger number of cases in the case-bases, a detailed comparison among all case descriptions is not possible in real time. Instead, the *Hausdorff distance* is used to find the most similar case in a more abstract comparison before an exact unit matching is performed. The quality of the results shows that this procedure is

successful. Other approaches that use CBR for unit control either work on a single-unit base, such as in the approach described in Chapter 5, or on a higher level that does not require the displayed degree of reproducible precision (Sharma et al., 2007; Gunnerud, 2009; Szczepański & Aamodt, 2009). As such, the approach presented in this thesis stands out for the level of detail and reproducibility it entails, characteristics which are in turn required for successful reinforcement learning.

As the results of the hierarchical approach and the subsequent discussion of these results show, the agent manages to achieve its goal of learning how to perform tactical and reactive tasks in RTS game micromanagement. For the rather simple *Scenario A*, a performance of more than 90% of the possible score is achieved for the highest tested similarity threshold of 95%, pointing to an overall performance which is comparable with that achieved for the CBR/RL agent created in Chapter 5 despite that agent not coordinating units. Even the most complex scenario tested, *Scenario E*, still results in a performance of more than 70% of the maximum obtainable reward. Given the three-level structure of the hierarchical architecture and the re-use of knowledge obtained on lower levels in higher-level problems - and as such any inaccuracies learned on these levels - the performance is remarkable.

An important aspect which could be part of future work is the comparison of the approach presented in this thesis to other bot architectures such as those examined in Section 3.2. While this comparison will require additional logic to also address the strategic layer (a requirement that prevented a comparison in this thesis) such a test could provide valuable insights into the power of adaptive online ML in relation to other ML, static and search-based approaches. As the comparison in Section 3.2 shows, there are only a few approaches which use ML on a large scale as part of agent AI that addresses the entire game. A comparison should therefore also give valuable insight into the capabilities of a holistic approach when compared to more diverse approaches.

Knowledge Transfer and Hierarchical Learning Process Transferring knowledge that is gained in a source task to improve performance in a related but unknown task is a common research topic. Re-use of prior experience is also closely related to the CBR methodology. The RTS game domain and especially the generation of adaptive RTS game AI agents are rewarding areas of application for transfer learning. While the general gameplay in this domain remains the same, unit and environment compositions differ among scenarios and games.

An early and notable effort in this area was the approach by M. J. Ponsen et al. (2005), who generated RTS game tactics in Wargus automatically. They extracted successful strategy ‘chromosomes’ which were developed using a GA and which translate into game AI scripts. Each successful tactic which has been extracted from an evolved chromosome is assigned to

a knowledge-base specific to its game state in terms of technological advancement. In an evaluation against several different scripted opponents, an adaptive agent then re-uses this knowledge based on the best fitting. Sharma et al. (2007) used a hierarchical approach to test transfer learning in RTS games. They worked with layers similar to those identified in Section 3.1.1 for strategy, tactics and reactive reasoning to create an agent for the MadRTS game. CARL (CAse-Based Reinforcement Learner) consists of three modules on each of these layers: a planner, a controller and a learner. The planner selects from the actions available in the state that the layer is currently in. The controller acts as an interface which communicates perceptions and actions to lower layers and the learner modifies the data and the features used by the planner. The top layer is hand-coded to select the overall strategy; the bottom layer takes orders from above and translates them into MadRTS-specific orders. The middle layer contains the actual hybrid CBR/RL module. The hybrid layer uses CBR as an instance-based function approximator to retrieve matching cases from the case-base while a TD RL algorithm is used to revise the cases according to how the selected case performs. The performed experiments show that this architecture works well for the given task of controlling combat units in Wargus. Furthermore, transferring the gained knowledge to a slightly more complex task considerably speeds up the learning process for that task.

Similar to these approaches, the knowledge stored in the modules of the hierarchical architecture is re-used in different scenarios. However, the problem is aggravated through re-use over multiple layers which multiplies any lack of state-action-space coverage. When re-using knowledge stored in lower-level case-bases, there is a 100% greedy policy without limiting similarity threshold. This means there will always be a case that is retrieved, even on the off-chance that there may be only cases with 0% similarity. For each of the lower-level modules for *Navigation*, *Attack* and *Navigation* there is a training phase, during which the agent learns in scenarios that are supposed to emulate any potential situations. However, given the complexity of the problem and the limit to only a number of the most representative scenarios due to time constraints, it is unlikely that a perfectly matching case exists for all encountered situations. This is especially true since the training, just like the evaluation, was done using $\psi = 80\%$. This leads, as shown in the evaluation of transferring knowledge between scenarios in Section 9.6, to missing cases when evaluating the knowledge in a different, slightly altered scenario.

Given these limitations, the comparatively simple scenarios that are based on known unit types and easy map layouts are likely a contributing factor to the good performance of the hierarchical agent. The scenarios tested in the evaluation of the *Tactical Unit Selection* component are all well-covered by the training scenarios for *Attack* actions and adequately well for *Formation* actions. The *Formation* action, which is more complex than the *Attack* action, would be at greater risk of not working correctly. However, since *Formation* actions

are used less frequently, as stated above, this would not be as noticeable as a malfunctioning or inadequately trained *Attack* module. In future work, it will be interesting to see a comparison in more experiments that use entirely different scenarios in terms of both army compositions as well as of map layouts. Another concept that would be interesting to evaluate in future work would be transfer learning to a related domain. Knowledge gained by the hierarchical agent described in this thesis could be re-used in a related domain such as another RTS game, e.g. *Wargus*.

One limiting factor of the architecture is the learning process. While the entire approach was initially created with a concurrent online learning approach in mind, the results from the two-level hierarchical CBR/RL module created in Chapter 5 suggested that, no matter the model, this would make learning extremely slow and initially inconsistent. Therefore, an iterative learning and evaluation process was used in training the different layers, similar to the learning process in the standard layered learning paradigm (Stone, 1998). While this approach proved effective, it slowed the overall learning-speed considerably and required a step-by-step training process. When looking at related hierarchical approaches in the literature, it appears that there is no easy solution to this problem. Andersen et al. (2009) used hierarchical RL to control an agent in a simple custom-made RTS game. The authors built their model specifically for fast adaptation to new situations and heavily abstracted the game environment, resulting in a representation which has only a fraction of the complexity of that used in the CBR/RL modules in this thesis. Hanna et al. (2010) also employed their hierarchical RL approach in a much simpler environment and without the added burden of managing large case-bases. M. Ponsen et al. (2006) decomposed a unit navigation task hierarchically and addressed it through RL to let a single unit learn how best to avoid opponents and reach its target. That agent managed to achieve a better performance than flat RL by reducing the state-space considerably. However, the complexity for the tasks was relatively small. Results thus only prove how a reduction in possible feature values through a hierarchical decomposition of the problem leads to better and faster learning results. Smyth & Cunningham (1992) described the implementation of the HCBR system *Déjà Vu* which uses a blackboard system, i.e. a high-level case which makes use of a number of underlying case-bases to compose a final solution to software design problems. In Smyth et al. (2001) this system was employed to create plant control software. However, in both instances the system was based on a multi-pass CBR cycle acquiring knowledge as in the standard CBR-cycle (Aamodt & Plaza, 1994). This process would lead to problems similar to those encountered in this thesis if it was used in a hierarchical approach using hybrid CBR/RL modules. This was also the case in Sushmita & Chaudhury (2007) where a HCBR approach was combined with fuzzy case description attributes for further abstraction in order to accurately analyse stocks in financial markets.

Looking at other approaches which use the layered learning (LL) paradigm in the simulated robotic soccer domain is slightly more promising. Stone (1998) described the original LL paradigm. The domain of robot soccer that the paradigm was developed for is very closely related to RTS game AI micromanagement which is reflected in the analysis in Section 3.5. However, a central principle of the LL paradigm is an individual learning process for each layer. Additionally, the learning for the three particular layers of the robot soccer problem domain implemented in Stone (1998) happens offline, unlike the learning proposed in this thesis. Whiteson & Stone (2003) offset these problems in their proposed *concurrent* layered learning paradigm (*CLL*), which learns on several layers concurrently instead of iteratively. This is similar to what was originally envisioned for the hierarchical CBR/RL agent in this thesis. However, the prerequisite for the application of concurrent learning is an independence among tasks which are learned at the same time, which is not the case in the tasks addressed by the layers in this thesis. In general, the authors only used CLL in very closely-defined conditions for a single task that is spread over two layers. Both the premise and the eventual findings in their research were that the application of concurrent learning only benefits the overall learning process in very particular circumstances.

MacAlpine et al. (2015) create a hybrid approach between original and concurrent layered learning and term it *overlapping* layered learning (*OLL*). In this paradigm, given a number of components organised in hierarchically interconnected layers, it is possible to partially open up underlying or superior layers and components while learning a certain layer above or below. The model, which is based on a large number of shared attributes (which are thus relevant on several layers), is only remotely comparable to the one used for the agent in this thesis. Further, the scope of the high-level actions is rather limited when compared to RTS game micromanagement. However, this approach shows some interesting possibilities in terms of training multiple modules concurrently and re-training partial modules. Given the fact that cases in most modules of the hierarchical CBR/RL architecture are based largely on unit descriptions, there could be significant potential in future work for optimising the learning process using a similar partial concurrent learning, if the different components can be adapted in an appropriate fashion. However, the requirement for online learning might be prohibitive; MacAlpine et al. (2015) train all modules offline.

Creating independent modules which can then be trained concurrently would be one way to accelerate the learning process. Other possible ways of improving performance would be through speeding up the individual CBR/RL components by employing better algorithmic techniques. Possible examples for this would be improved case-retrieval through techniques such as hashing (Andoni & Indyk, 2004) or *kd*-Trees (Wess et al., 1994). Reducing the overall number of learning episodes is also an easy way to reduce learning times, therefore attempting different RL algorithms or exploration techniques (Brafman & Tennenholtz, 2003;

R. Sutton et al., 2009) would be an option, as would be a re-evaluation of eligibility traces which indicated the potential for faster convergence although at the cost of slightly worse performance (see Section 4).

Creating Human-Like Behaviour and Summary An interesting aspect to evaluate is the number of the different *Level Two* actions (shown in Table 9.8) which are selected by the *Tactical Unit Selection* module. There are some tangible effects that the numbers indicate and that are evaluated in detail in Section 9.5. However, the distribution of *Level Two* actions also contains information on the high-level behaviour that the agent learns. This high-level behaviour in turn can be generalised to how well the agent manages to learn desired behaviour which human experts would consider ideal.

Predictably, the distribution shows that any scenario with quick, agile units (such as *A*, *C* and *E*) uses a lot of *Attack* and *Retreat* actions, indicating the hit-and-run strategy these units perform. For the melee-only scenarios (*B* and *D*) there is a clear focus on attacking the opponent and only marginal retreat actions combined with few *Formation* actions. *Scenario D*, in which the agent is heavily outnumbered, uses more formations since this postpones the inevitable defeat slightly longer, whereas *Scenario B* is mostly won by using effective focus-fire.

The more prominent use of *Formation* actions in *Scenarios C* and *E* points to effective grouping strategies, where ranged units are covered through melee units, just as intended. The complete lack of *Formation* actions in *Scenario A* is, as mentioned previously, due to the fact that an individual unit control is the most effective strategy in that scenario since each unit can escape from its opponents.

Overall, the agent manages to learn a number of different high-level strategies. The agent thus combines atomic lower-level actions in efficient ways that it examined through trial-and-error and created behaviours which humans could perceive as ‘intelligent’. While there is only a small number of different high-level strategies which can be identified by looking at the use of *Level Two* actions, the general trend points to the desired ability to create new behaviour by combining previously learned lower-level actions.

To summarise the findings from the approach in this thesis; using the currently existing hierarchically-connected modules on the three different levels of abstraction the agent managed to learn how to effectively control its combat units. It also managed to find strategies which a human player would consider appropriate in the given scenarios. Combined with the performance results presented in Chapter 9, this is a powerful indicator that the creation of an adaptive AI agent was successful and that the agent manages to learn how to solve the reactive and tactical tasks that are part of RTS games.

This thesis investigated the creation of an adaptive machine learning agent that addresses both reactive and tactical tasks in RTS games. The initial literature review led to a number of observed potential areas for improvement and pointed out existing shortcomings in its summary of the current state-of-the-art of relevant research areas. There is considerable interest in using video games in general and RTS video games in particular as testbeds for AI research. The interesting AI problems that these games exhibit, combined with the complex, polished simulation environments that commercial games are, makes them rewarding testbeds.

While a hierarchical decomposition of RTS game problems is commonly used to reduce the significant complexity inherent in the domain, there is no widely accepted architecture that breaks this high-level decomposition into standardised problem building blocks. Every researcher approaches the problem with a different architecture implementation (Ontañón et al., 2013). In addition to the lack of a standardised architecture, the universal approach is to not use a holistic ML approach for all sub-tasks involved in the game. Game-playing bots were instead found to be patchworks of different approaches for different components. Some of these components do use adaptive ML approaches. However, even in AI-research-oriented bots, a large number of the sub-problems were often addressed through static, expert-knowledge-based techniques such as FSMs, decision trees and scripted behaviour. This characteristic is even more prevalent for the lower-level logic, i.e. the micromanagement part of the game, that was addressed as part of this thesis. While interest in this area has been increasing in recent years, approaches that address micromanagement through ML often look at it as an isolated problem. Given this current state of research, designing an adaptive ML approach that addresses the entire lower-level logic in an integrated way was determined to be a rewarding task.

The main research objective was thus to create an agent using adaptive ML that is able to acquire the necessary knowledge to perform those tasks. As the findings in Chapter 9 showed, this objective was achieved. While there remain a number of limitations in the presented implementation of the agent, the hierarchical micromanagement agent has been shown to

perform well in all tested scenarios with different unit numbers and different unit types. Furthermore, as pointed out in Chapter 10, extending the current implemented architecture by adding additional components can rectify these shortcomings without major structural changes. The second part of the main research question, regarding the specific requirements for the creation of such an agent, was answered in the individual components that are created for micromanagement sub-tasks.

The iterative process triggered through the main research questions led to the smaller, more technical questions. These smaller questions then allowed identification of potential solutions and led to the actual contributions.

1. *Given its similarity to the human learning process and its ability to find optimal policies in unknown environments, can reinforcement learning achieve the desired results of learning how to perform in the domain of RTS micromanagement?*

This question resulted in the first step in the iterative process towards the adaptive ML agent and was answered in Chapter 4 through an evaluation of the suitability of a number of TD RL algorithms. A fitting model was developed and all tested algorithms were found to perform well for the selected task. The one-step Q-learning algorithm was selected as the most appropriate algorithm as it balanced both performance and speed of learning well.

2. *If RL is not able to solve the problem on its own, does the creation of a hybrid approach that uses another ML technique to improve the agent's capabilities lead to a solution?*

Chapter 5 described the creation of a hybrid CBR/RL approach that uses the previously selected Q-learning. This TD algorithm was integrated with a CBR system that uses cases for memory management and generalisations over the vast state- and action-space inherent to RTS game micromanagement. The creation of the hybrid algorithm led to an improved speed of convergence when compared to simple RL. More importantly, hybrid CBR/RL enabled the extension of the model and architecture beyond a RL-only table-based approach. This part of the thesis also optimised important algorithmic RL parameters which were re-used in subsequent applications of RL. An analysis of the characteristics of the CBR component was used to gather insights on the development of the case-bases given the model and problem domain that the methodology was applied to. This included the experimental discovery of a state-action space coverage that is suitable to guarantee near-optimal performance. Furthermore, a similarity threshold for the generation of new cases was experimentally determined that leads to a balanced performance between both the obtained reward and computational effort required to explore all existing case solutions.

3. *When using a combination of RL and CBR, how can the problem as well as the relevant elements of the simulation environment be represented so that a ML agent can adequately address the micromanagement tasks using this hybrid technique?*

The agent architecture presented in Chapter 6.1 combined the existing concepts of hierarchical CBR and layered learning and adapted both to the task of online multi-layer learning for low-level RTS game AI. The newly-developed design was based on an investigation of the sub-problems involved in RTS game micromanagement. Finally, the chosen decomposition separated the overall problems into individual tasks which were identified as relevant to RTS game micromanagement. These tasks were *Navigation*, *Attack*, *Formation*, *Retreat* and, on the highest level, *Tactical Unit Selection*.

4. *In a hierarchical modular architecture that addresses reactive and tactical tasks which are part of RTS games, what are the challenges involved in addressing relevant sub-tasks through hybrid CBR/RL?*

Given the task hierarchy designed in Chapter 6.1, each identified task was shaped into a module able to solve that particular task. *Retreat* was modelled as a static component while *Navigation* (Chapter 7), *Attack* and *Formation* (Chapter 8) and *Tactical Unit Selection* (Chapter 9) were identified as learning tasks and represented through individual CBR/RL modules. Each of these adaptive modules used a CBR/RL approach to learn through interacting with the game environment. Extensive modelling was used to create appropriate MDPs for each sub-task. The process also involved the solution of numerous individual sub-problems associated with the respective domains. These sub-problems included the creation of a solution case-base in the *Formation* module and the recurring issue of determining appropriate similarity thresholds and, as a result, appropriate case-space coverage for a particular problem. The experimental evaluation of the individual modules showed that all the relevant sub-tasks were solved successfully.

5. *What interfaces are required in a hierarchical hybrid CBR/RL architecture and what are the potential effects, positive and negative, of using a homogeneous ML approach for significant parts of reasoning spanning multiple layers of abstraction in an RTS game?*

All previously-created individual modules were integrated into the overall architecture in Chapter 9. The evaluation of the resulting *Tactical Unit Selection* component showed that learning on this highest layer in the architecture worked well, enabled through good, consistent performance on the lower layers. Adding up learning- and training times for all modules meant that the agent required a significant learning phase which was projected to become even longer if more modules or even layers were added. Minor propagation of errors and missing knowledge from lower levels was observed, yet performance in the tested scenarios was only minimally affected.

6. *How successful is transfer learning across multiple levels of hierarchically interconnected modules?*

As the hierarchically-interconnected modules reused knowledge from lower layers, the evaluation of modules on the higher levels demonstrated successful transfer of acquired knowledge from the original learning scenarios to new problems. Additionally, Section 9.6 specifically evaluated the effects of transferring *Tactical Unit Selection* knowledge from one scenario to another. This evaluation of transfer learning showed significant performance improvements through transferring the knowledge gained in one scenario to another problem scenario. Furthermore, this capability to speed up learning between different scenarios can form the basis of increasing the speed of knowledge acquisition in future research.

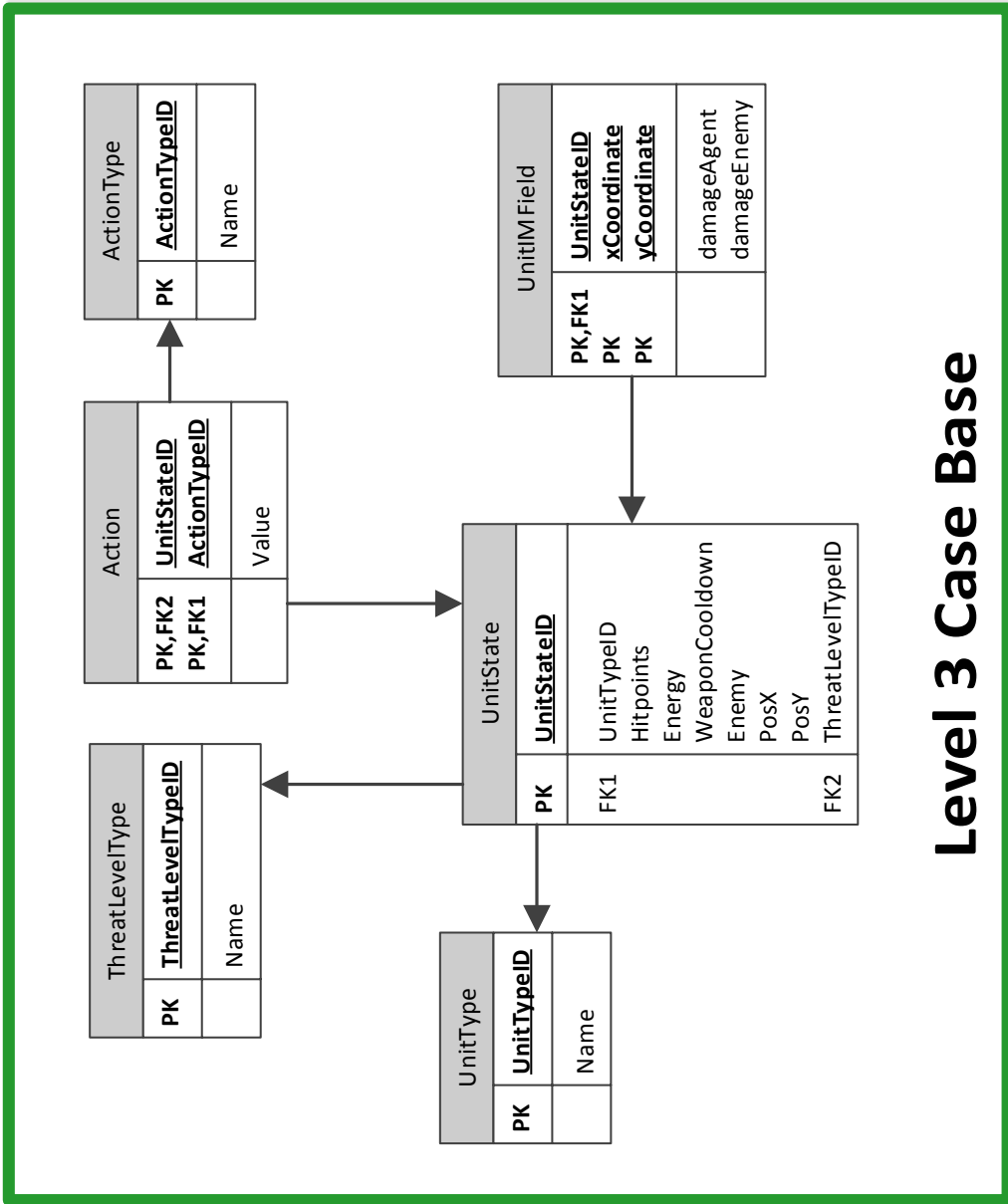
In summary, the key contribution of this thesis has been to provide an integrated hierarchical CBR/RL agent which learns how to solve both reactive and tactical RTS game tasks. The creation of the individual hybrid CBR/RL modules for tasks in RTS game micromanagement is based on thorough analyses of TD RL algorithms, CBR behaviour and the relevant problem domain tasks. The resulting agent architecture acquires the required knowledge through online learning in the game environment and is able to re-use the knowledge to successfully solve tactical RTS game scenarios.

Appendix A

Database Diagrams

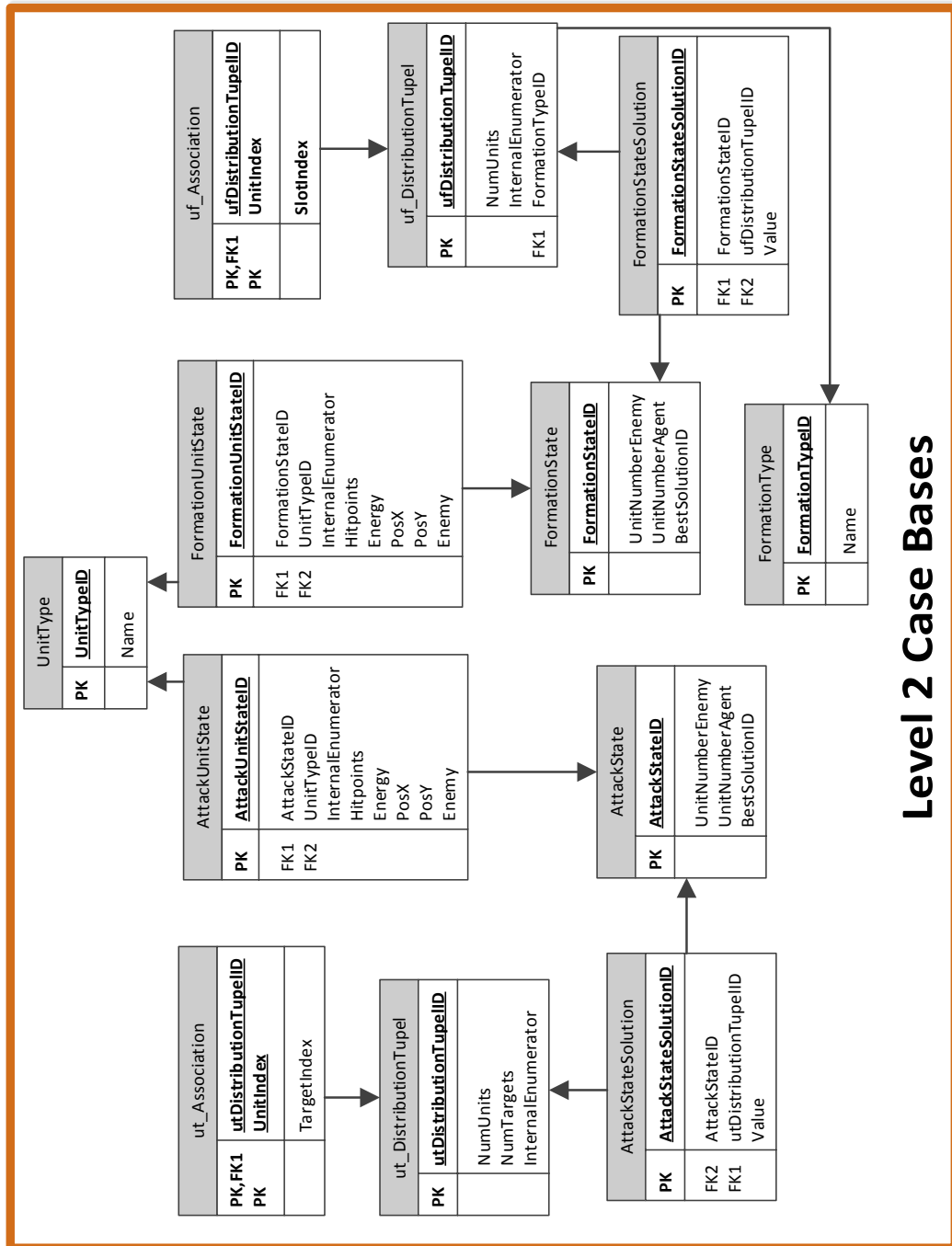
Figures A.1 to A.3 display the entity-relationship database diagrams for the table structures that were created to represent the case-bases in the different parts of the thesis.

Note: Figure A.2, which displays the database structure for *Level Two* of the architecture, does not display the additional tables of the solution case-base.



Level 3 Case Base

Figure A.1: DB Diagram of MySQL Table Structure Hierarchical Architecture Level 3



Level 2 Case Bases

Figure A.2: DB Diagram of MySQL Table Structure Hierarchical Architecture Level 2

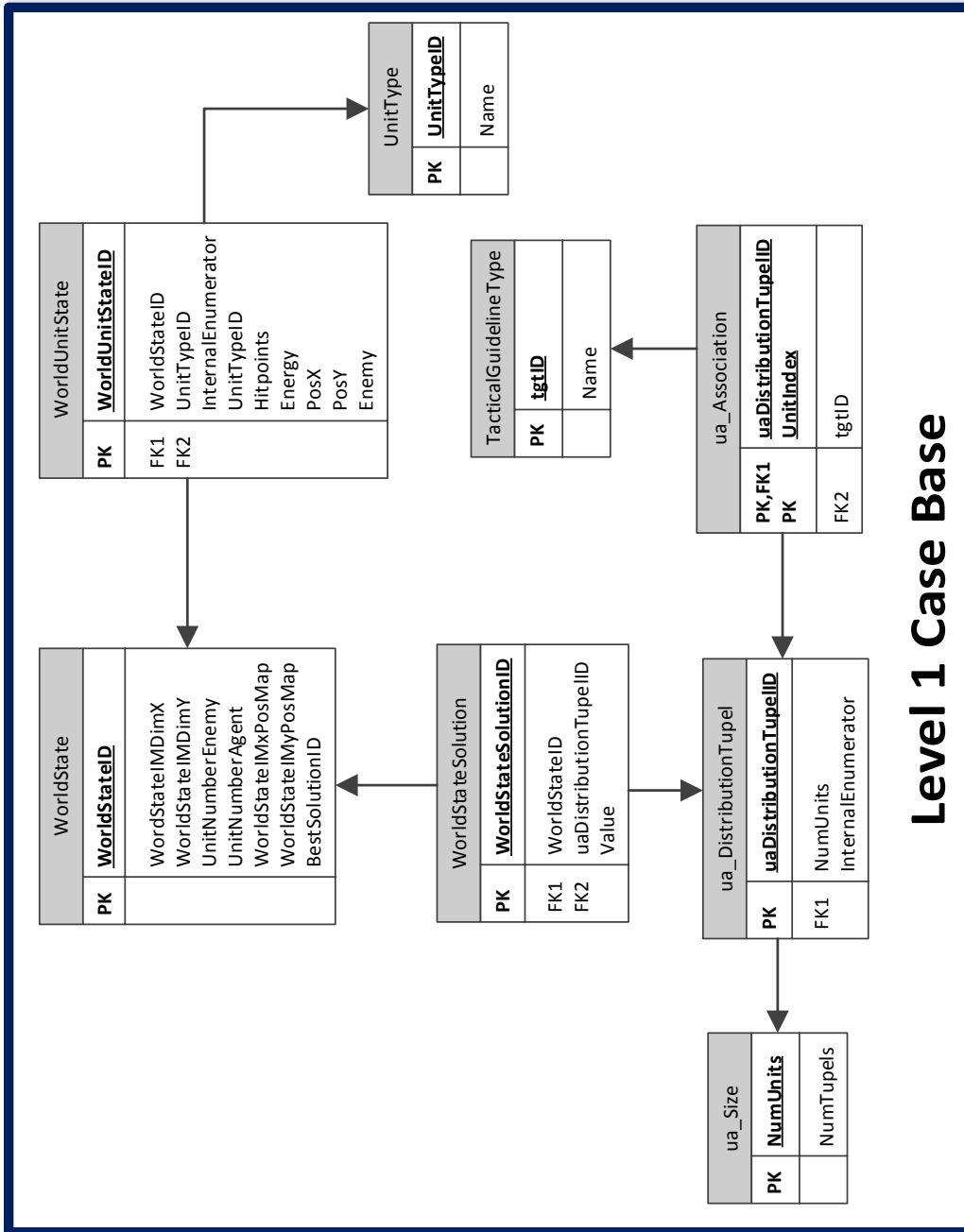


Figure A.3: DB Diagram of MySQL Table Structure Hierarchical Architecture Level 1

Appendix B

Munkres Assignment Algorithm

Figure B.1 shows the steps involved in the Munkres assignment algorithm, modified for rectangular matrices (Pilgrim, 2015).

0. Create an $n \times m$ matrix called the **cost matrix** in which each element represents the cost of assigning one of n workers to one of m jobs. Rotate the matrix so that there are at least as many columns as rows and let $k = \min(n, m)$.
 1. For each row of the matrix, find the smallest element and subtract it from every element in its row. Go to Step 2.
 2. Find a zero (**Z**) in the resulting matrix. If there is no starred zero in its row or column, star **Z**. Repeat for each element in the matrix. Go to Step 3.
 3. Cover each column containing a starred zero. If K columns are covered, the starred zeros describe a complete set of unique assignments. In this case, Go to **DONE**, otherwise, Go to Step 4.
 4. Find a non-covered zero and prime it. If there is no starred zero in the row containing this primed zero, Go to Step 5. Otherwise, cover this row and uncover the column containing the starred zero. Continue in this manner until there are no uncovered zeros left. Save the smallest uncovered value and Go to Step 6.
 5. Construct a series of alternating primed and starred zeros as follows. Let Z_0 represent the uncovered primed zero found in Step 4. Let Z_1 denote the starred zero in the column of Z_0 (if any). Let Z_2 denote the primed zero in the row of Z_1 (there will always be one). Continue until the series terminates at a primed zero that has no starred zero in its column. Unstar each starred zero of the series, star each primed zero of the series, erase all primes and uncover every line in the matrix. Return to Step 3.
 6. Add the value found in Step 4 to every element of each covered row, and subtract it from every element of each uncovered column. Return to Step 4 without altering any stars, primes, or covered lines.
- DONE.** Assignment pairs are indicated by the positions of the starred zeros in the cost matrix. If $C(i, j)$ is a starred zero, then the element associated with row i is assigned to the element associated with column j .

Figure B.1: Munkres Assignment Algorithm Steps, Adapted from (Pilgrim, 2015)

Appendix C

Algorithm Parameter Optimization

This appendix shows an excerpt of both the raw data file as well as the resulting model of the parameter optimisation done as part of the evaluation in Chapter 5. The results were obtained using the WEKA machine learning software (Hall et al., 2009) using linear regression.

```
@RELATION algorithm_parameters

@ATTRIBUTE Scenario
@ATTRIBUTE Algorithm
@ATTRIBUTE RunLength
@ATTRIBUTE Similarity
@ATTRIBUTE LearningRate
@ATTRIBUTE EligibilityDecayRate
@ATTRIBUTE DiscountRate
@ATTRIBUTE ResultAverage

@DATA
1,0,50,0.6,0.2,0,0.9,179.55
1,0,50,0.6,0.2,0,0.75,171.16
1,0,50,0.6,0.2,0,0.6,144.47
1,0,50,0.6,0.3,0,0.9,143.27
1,0,50,0.6,0.3,0,0.75,138.83
1,0,50,0.6,0.3,0,0.6,103.71
1,0,50,0.6,0.4,0,0.9,140.34
1,0,50,0.6,0.4,0,0.75,105.92
1,0,50,0.6,0.4,0,0.6,116.23
1,0,50,0.8,0.2,0,0.9,132.22
1,0,50,0.8,0.2,0,0.75,131.42
1,0,50,0.8,0.2,0,0.6,121.9
1,0,50,0.8,0.3,0,0.9,151.94
1,0,50,0.8,0.3,0,0.75,138.91
1,0,50,0.8,0.3,0,0.6,104.11
1,0,50,0.8,0.4,0,0.9,114.43
1,0,50,0.8,0.4,0,0.75,99.47
1,0,50,0.8,0.4,0,0.6,123.84
1,0,500,0.6,0.2,0,0.9,232.58
1,0,500,0.6,0.2,0,0.75,238.06
1,0,500,0.6,0.2,0,0.6,249.81
1,0,500,0.6,0.3,0,0.9,240.73
1,0,500,0.6,0.3,0,0.75,269.18
1,0,500,0.6,0.3,0,0.6,240.67
1,0,500,0.6,0.4,0,0.9,212.61
1,0,500,0.6,0.4,0,0.75,213.47
1,0,500,0.6,0.4,0,0.6,232.27
1,0,500,0.8,0.2,0,0.9,256.45
1,0,500,0.8,0.2,0,0.75,225.49
1,0,500,0.8,0.2,0,0.6,242.61
1,0,500,0.8,0.3,0,0.9,235.68
1,0,500,0.8,0.3,0,0.75,246.91
1,0,500,0.8,0.3,0,0.6,245.88
1,0,500,0.8,0.4,0,0.9,224
1,0,500,0.8,0.4,0,0.75,234.45714
[...]
```

Scenario	Algorithm	RunLength	ResultAverage	Note
1	0	50	-69.5778 * Similarity + -150.4083 * LearningRate + 81.9389 * DiscountRate + 163.5794	
1	0	500	-82.7108 * LearningRate + 262.0242	
1	1	50	-53.575 * Similarity + -201.3833 * LearningRate + -80.6315 * EligibilityDecayRate + 300.8636	
1	1	500	-116.3253 * Similarity + -321.8422 * LearningRate + -88.0553 * EligibilityDecayRate + 468.3321	
2	0	50	-130.9083 * LearningRate + 102.2444 * DiscountRate + 69.4225	
2	0	500	-428.0218 * LearningRate + 364.6956	
2	1	50	-307.9833 * LearningRate + -142.9593 * EligibilityDecayRate + -136.5889 * DiscountRate + 437.4389	
2	1	500	-320.3028 * EligibilityDecayRate + 528.555	
1	0 x		0.2344 * RunLength + -120.4023 * LearningRate + 32.9396 * DiscountRate + 130.9013	Note: runLength required to allow classification
1	1 x		0.1851 * RunLength + -77.7714 * Similarity + -250.8447 * LearningRate + -85.3792 * EligibilityDecayRate + -49.7187 * DiscountRate + 364.2367	
2	0 x		0.2902 * RunLength + -270.8604 * LearningRate + 87.1709 * DiscountRate + 108.2018	
2	1 x		0.2829 * RunLength + -294.2341 * LearningRate + -164.6701 * EligibilityDecayRate + -137.0481 * DiscountRate + 435.7942	
1 x	x		0.2007 * RunLength + -65.1383 * Similarity + -210.1575 * LearningRate + 237.3271	(with eDecay taken into account) 65.58 * Algorithm + 0.2013 * RunLength + -63.6683 * Similarity + -207.9525 * LearningRate + -86.3025 * EligibilityDecayRate + 235.0352
2 x	x		151.5349 * Algorithm + 0.2792 * RunLength + -69.1442 * Similarity + -280.8044 * LearningRate + -168.4926 * EligibilityDecayRate + 227.7718	

References

- Aamodt, A., & Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1), 39–59.
- Aha, D., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. *Case-Based Reasoning Research and Development*, 5–20.
- Aha, D. W., & Molineaux, M. (2004). *Integrating Learning in Interactive Gaming Simulators* (Tech. Rep.). Intelligent Decision Aids Group; Navy Center for Applied Research in Artificial Intelligence.
- Andersen, K. T., Zeng, Y., Christensen, D. D., & Tran, D. (2009). Experiments with online reinforcement learning in real-time strategy games. *Applied Artificial Intelligence*, 23(9), 855–871.
- Andoni, A., & Indyk, P. (2004). *E2lsh: Exact euclidean locality-sensitive hashing*. Retrieved 28/05/2015, from <http://web.mit.edu/andoni/www/LSH/>
- Andrade, G., Ramalho, G., Santana, H., & Corruble, V. (2005). Automatic Computer Game Balancing: A Reinforcement Learning Approach. In *Aamas '05: Proceedings of the fourth international joint conference on autonomous agents and multiagent systems* (pp. 1111–1112). New York, NY, USA: ACM.
- Aron, J. (2011). How innovative is apple's new voice assistant, siri? *New Scientist*, 212(2836), 24.
- Auslander, B., Lee-Urban, S., Hogg, C., & Muñoz-Avila, H. (2008). Recognizing the Enemy: Combining Reinforcement Learning with Strategy Selection using Case-Based Reasoning. In *Proceedings of the 9th european conference on advances in case-based reasoning (eccbr-08)*. Springer.
- Balch, T., & Arkin, R. C. (1998). Behavior-based formation control for multirobot teams. *Robotics and Automation, IEEE Transactions on*, 14(6), 926–939.

References

- Balla, R., & Fern, A. (2009). Uct for tactical assault planning in real-time strategy games. In *21st international joint conference on artificial intelligence*.
- Baumgarten, R., Colton, S., & Morris, M. (2008). Combining ai methods for learning bots in a real-time strategy game. *International Journal of Computer Games Technology*, 2009.
- Behnke, S., & Stückler, J. (2008). Hierarchical reactive control for humanoid soccer robots. *International Journal of Humanoid Robotics*, 5(03), 375–396.
- Bellman, R. (1957a). *Dynamic Programming*. Princeton, NJ: Princeton University Press.
- Bellman, R. (1957b). A Markov Decision Process. *Journal of Mathematical Mechanics*, 6, 679-684.
- Bellman, R. E. (1961). *Adaptive control processes: a guided tour* (Vol. 4). Princeton university press Princeton.
- Bethesda Game Studios. (2011). *The elder scrolls v: Skyrim*.
- Blizzard Entertainment. (1994). *Warcraft: Orcs & humans*. Retrieved 28/05/2015, from <http://us.blizzard.com/en-us/games/legacy/>
- Blizzard Entertainment. (1998). *Starcraft*. Retrieved 28/05/2015, from <http://us.blizzard.com/en-us/games/sc/>
- Blue Byte. (1993). *The settlers*. Retrieved 28/05/2015, from http://en.wikipedia.org/wiki/The_Settlers
- Bourg, D. M., & Seemann, G. (2004). *Ai for game developers*. " O'Reilly Media, Inc."
- Bowen, N., Todd, J., & Sukthankar, G. (2013). Adjutant bot: An evaluation of unit micro-management tactics. In *Computational intelligence and games (cig), 2013 ieee conference on* (pp. 409–416).
- Bradski, G. (2000). The opencv library. *Doctor Dobbs Journal*, 25(11), 120–126.
- Brafman, R. I., & Tennenholtz, M. (2003). R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 3, 213–231.
- Bridge, D. (2005). The virtue of reward: Performance, reinforcement and discovery in case-based reasoning. *Case-Based Reasoning Research and Development*, 1–1.

- Bulitko, V., Bjornsson, Y., & Lawrence, R. (2010). Case-based subgoaling in real-time heuristic search for video game pathfinding. *Journal of Artificial Intelligence Research*, 39, 269–300.
- Buro, M. (2003a). Orts: A hack-free rts game environment. *Computers and Games*, 280–291.
- Buro, M. (2003b). Real-time strategy games: a new ai research challenge. In *Proceedings of the 18th international joint conference on artificial intelligence* (pp. 1534–1535).
- Buro, M., & Furtak, T. (2004). Rts games and real-time ai research. In *Proceedings of the behavior representation in modeling and simulation conference (brims)* (pp. 63–70).
- Buro, M., & Furtak, T. (2005). On the development of a free rts game engine. In *Proceedings of the international conference on intelligent games and simulation (gameon)* (pp. 23–27).
- BWAPI. (2009). *Bwapi - an api for interacting with starcraft: Brood war*. Retrieved 28/05/2015, from <http://us.blizzard.com/en-us/games/sc/>
- Cadena, P., & Garrido, L. (2011). Fuzzy case-based reasoning for managing strategic and tactical reasoning in starcraft. In *Advances in artificial intelligence* (pp. 113–124). Springer.
- Campbell, M., Hoane, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1), 57–83.
- Chung, M., Buro, M., & Schaeffer, J. (2005). Monte carlo planning in rts games. In *Proceedings of the ieee symposium on computational intelligence and games*.
- Churchill, D., & Buro, M. (2011). Build order optimization in starcraft. In *Proceedings of the seventh artificial intelligence and interactive digital entertainment international conference (aiide 2011)* (pp. 14–19).
- Churchill, D., & Buro, M. (2012). Incorporating search algorithms into rts game agents. In *Workshop proceedings of the eight artificial intelligence and interactive digital entertainment international conference (aiide 2012)*.
- Churchill, D., & Buro, M. (2013). Portfolio greedy search and simulation for large-scale combat in starcraft. In *Computational intelligence in games (cig), 2013 ieee conference on* (pp. 1–8).
- Churchill, D., Saffidine, A., & Buro, M. (2012). Fast heuristic search for rts game combat scenarios. In *Proceedings of the eight artificial intelligence and interactive digital entertainment international conference (aiide 2012)*.

References

- Cothran, J., & Champandard, A. (2009). *Winning the 2k bot prize with a long-term memory database using sqlite*. Online Article. Retrieved 28/05/2015, from <http://aigamedev.com/open/articles/sqlite-bot> (Retrieved on 26.04.2015)
- Cox, M. (2007). Perpetual self-aware cognitive agents. *AI Magazine*, 28(1), 32.
- Cox, M., Muñoz-Avila, H., & Bergmann, R. (2005). Case-based planning. *Knowledge Engineering Review*, 20(3), 283–288.
- Craighead, J., Burke, J., & Murphy, R. (2007). Using the unity game engine to develop sarge: a case study. *Computer*, 4552, 366–372.
- Cunningham, P. (2009). A taxonomy of similarity mechanisms for case-based reasoning. *Knowledge and Data Engineering, IEEE Transactions on*, 21(11), 1532–1543.
- Dainotti, A., Pescape, A., & Ventre, G. (2005). A packet-level traffic model of starcraft. In *Hot topics in peer-to-peer systems, 2005. hot-p2p 2005. second international workshop on* (pp. 33–42).
- Davis, I. L. (1999). Strategies for strategy game ai. In *Proceedings of the aai spring symposium on artificial intelligence and computer games* (pp. 24–27).
- Davoust, A., Floyd, M., & Esfandiari, B. (2008). Use of fuzzy histograms to model the spatial distribution of objects in case-based reasoning. In S. Bergler (Ed.), *Advances in artificial intelligence* (Vol. 5032, p. 72-83). Springer Berlin Heidelberg.
- Dereszynski, E., Hostetler, J., Fern, A., Dietterich, T., Hoang, T., & Udarbe, M. (2011). Learning probabilistic behavior models in real-time strategy games. In *Proceedings of the seventh artificial intelligence and interactive digital entertainment international conference (aiide 2011)*.
- Ensemble Studios. (1997). *Age of empires*. Retrieved 28/05/2015, from <http://www.microsoft.com/games/empires/>
- ESA. (2011). *2011 sales, demographic and usage data - essential facts about the computer and video game industry* (Tech. Rep.). Electronic Software Association.
- Expressive Intelligence Studio. (2010). *Aiide 2010 starcraft ai competition*. Retrieved 28/05/2015, from <http://eis.ucsc.edu/StarCraftAICompetition>
- Farley, B., & Clark, W. (Sep 1954). Simulation of Self-Organizing Systems by Digital Computer. *Information Theory, IEEE Transactions on*, 4(4), 76-84.

-
- Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., ... others (2010). Building watson: An overview of the deepqa project. *AI magazine*, 31(3), 59–79.
- Firaxis Games. (2007). *Civilization iv software development kit*.
- Fürnkranz, J. (2001). Machine Learning in Games: A Survey. In *Machines that learn to play games* (p. 11-59). Nova Biomedical.
- Fürnkranz, J. (2007). Recent advances in machine learning and game playing. *ÖGAI Journal*, 26(2), 19–28.
- Galway, L., Charles, D., & Black, M. (2008). Machine learning in digital games: a survey. *Artificial Intelligence Review*, 29(2), 123–161.
- Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). Modification of uct with patterns in monte-carlo go.
- Graepel, T., Herbrich, R., & Gold, J. (2004). Learning to Fight. In *Proceedings of the international conference on computer games: Artificial intelligence, design and education*.
- Gudgeon, C. (1907). Maori wars. *Journal of the Polynesian Society*, 16(1), 13–42.
- Gundevia, U. (2006). *Integrating War Game Simulations with AI Testbeds: Integrating Call To Power 2 with TIELT* (Master's thesis). Lehigh University.
- Gunnerud, M. (2009). *A cbr/rl system for learning micromanagement in real-time strategy games* (Master's thesis). Norwegian University of Science and Technology.
- Gustafson, S. M., & Hsu, W. H. (2001). *Layered learning in genetic programming for a cooperative robot soccer problem* (Master's thesis).
- Hagelbäck, J. (2012). *Multi-agent potential field based architectures for real-time strategy game bots* (PhD thesis). Blekinge Institute of Technology.
- Hagelback, J. (2012). Potential-field based navigation in starcraft. In *Computational intelligence and games (cig), 2012 ieee conference on* (pp. 388–393).
- Hagelbäck, J., & Johansson, S. (2008). Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th international joint conference on autonomous agents and multiagent systems-volume 2* (pp. 631–638).
- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1), 10–18.

References

- Hammond, K. (1989). *Case-based planning: viewing planning as a memory task*. Academic Press Professional, Inc.
- Hanna, C. J., Hickey, R. J., Charles, D. K., & Black, M. M. (2010). Modular reinforcement learning architectures for artificially intelligent agents in complex game environments. In *Computational intelligence and games (cig), 2010 ieee symposium on* (pp. 380–387).
- Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2), 100–107.
- Hsieh, J., & Sun, C. (2008). Building a player strategy model by analyzing replays of real-time strategy games. In *Neural networks, 2008. ijcnn 2008.(ieee world congress on computational intelligence). ieee international joint conference on* (pp. 3106–3111).
- Hsu, W. H., & Gustafson, S. M. (2002). Genetic programming and multi-agent layered learning by reinforcements. In *Gecco* (pp. 764–771).
- Huang, H. (2011). Skynet meets the swarm: how the berkeley overmind won the 2010 starcraft ai competition. *Ars Technica*, 18.
- Humphrys, M. (1996). Action selection methods using reinforcement learning. *From Animals to Animats*, 4, 135–144.
- Huttenlocher, D. P., Klanderman, G. A., & Rucklidge, W. J. (1993). Comparing images using the hausdorff distance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(9), 850–863.
- id Software. (2011). *id software game engine source code*. Retrieved 28/05/2015, from <https://github.com/id-Software>
- Jaidee, U., & Muñoz-Avila, H. (2012). Classq-l: A q-learning algorithm for adversarial real-time strategy games. In *Proceedings of the eight artificial intelligence and interactive digital entertainment international conference (aiide 2012)*.
- Jaidee, U., & Muñoz-Avila, H. (2013). Modeling unit classes as agents in real-time strategy games. In *Proceedings of the ninth artificial intelligence and interactive digital entertainment international conference (aiide 2013)*.
- Jaidee, U., Muñoz-Avila, H., & Aha, D. (2011). Integrated learning for goal-driven autonomy. In *Proceedings of the twenty-second international conference on artificial intelligence (ijcai-11)*.

-
- Jain, A. K. (1989). *Fundamentals of digital image processing* (Vol. 3). Prentice-Hall.
- Järvinen, A. (2002). Halo and the anatomy of the fps. *Game Studies*, 2(1), 641–661.
- Johnson, S. (2008). *Playing to lose: Ai and civilization*. Conference Presentation. Retrieved 28/05/2015, from <https://www.youtube.com/watch?v=IJcuQQ1eWWI>
- Kaminsky, R., Enev, M., & Andersen, E. (2008). *Identifying game players with mouse biometrics* (Tech. Rep.). University of Washington.
- Karpov, I., D’Silva, T., Varrichio, C., Stanley, K., & Miikkulainen, R. (May 2006). Integration and Evaluation of Exploration-Based Learning in Games. *Computational Intelligence and Games, 2006 IEEE Symposium on*, 1, 39-44.
- Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1), 90–98.
- Kim, J., Yoon, K., Yoon, T., & Lee, J. (2010). Cooperative learning by replay files in real-time strategy game. *Cooperative Design, Visualization, and Engineering*, 6240/2010, 47–51.
- Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., & Osawa, E. (1997). Robocup: The robot world cup initiative. In *Proceedings of the first international conference on autonomous agents* (pp. 340–347).
- Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. In *Machine learning: Ecml 2006* (pp. 282–293). Springer.
- Kruusmaa, M. (2003). Global navigation in dynamic environments using case-based reasoning. *Autonomous Robots*, 14(1), 71–91.
- Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2), 83–97.
- Laird, J., & van Lent, M. (2001). Human-level ai’s killer application: Interactive computer games. *AI Magazine*, Summer 2001, 1171-1178.
- Lara-Cabrera, R., Cotta, C., & Fernández-Leiva, A. J. (2013). A review of computational intelligence in rts games. In *Foundations of computational intelligence (foci), 2013 ieee symposium on* (pp. 114–121).
- Laue, T., Spiess, K., & Röfer, T. (2006). Simrobot—a general physical robot simulator and its application in robocup. In *Robocup 2005: Robot soccer world cup ix* (pp. 173–183). Springer.
-

References

- Lewis, J., Trinh, P., & Kirsh, D. (2011). A corpus analysis of strategy video game play in starcraft: Brood war. In *The annual meeting of the cognitive science society (cogsci 2011)*.
- Liao, T. W., Zhang, Z., & Mount, C. R. (1998). Similarity measures for retrieval in case-based reasoning systems. *Applied Artificial Intelligence*, 12(4), 267–288.
- Lin, C.-S., & Ting, C.-K. (2011). Emergent tactical formation using genetic algorithm in real-time strategy games. In *Technologies and applications of artificial intelligence (taai), 2011 international conference on* (pp. 325–330).
- MacAlpine, P., Depinet, M., & Stone, P. (2015). Ut austin villa 2014: Robocup 3d simulation league champion via overlapping layered learning. In *Proc. of the twenty-ninth aaii conf. on artificial intelligence (aaii)*.
- Maei, H. R., Szepesvári, C., Bhatnagar, S., & Sutton, R. S. (2010). Toward off-policy learning control with function approximation. In *Proceedings of the 27th international conference on machine learning (icml-10)* (pp. 719–726).
- Marthi, B., Russell, S. J., Latham, D., & Guestrin, C. (2005). Concurrent hierarchical reinforcement learning. In *Proceedings of the nineteenth international conference on artificial intelligence (ijcai-05)* (pp. 779–785).
- Mateas, M., Bates, J., & Carbonell, J. (2002). *Interactive drama, art and artificial intelligence* (PhD thesis). School of Computer Science, Carnegie Mellon University.
- Max Design. (1998). *Anno 1602*. Retrieved 28/05/2015, from http://en.wikipedia.org/wiki/The_Settlers
- McCoy, J., & Mateas, M. (2008). An integrated agent for playing real-time strategy games. In *Proceedings of the aaii conference on artificial intelligence (aaii)* (pp. 1313–1318).
- McPartland, M., & Gallagher, M. (2008, October). Learning to be a Bot: Reinforcement Learning in Shooter Games. In *Proceedings of the fourth artificial intelligence and interactive digital entertainment conference (aiide)*. Stanford, California: AAAI Press.
- Micić, A., Arnarsson, D., & Jónsson, V. (2011). *Developing game ai for the real-time strategy game starcraft* (Tech. Rep.). Reykjavik University.
- Mille, A. (2006). From case-based reasoning to traces-based reasoning. *Annual Reviews in Control*, 30(2), 223–232.
- Millington, I., & Funge, J. (2009). Artificial intelligence for games. In (p. 144 - 196). CRC Press.

-
- Mishra, K., Ontañón, S., & Ram, A. (2008). Situation assessment for plan retrieval in real-time strategy games. *Advances in Case-Based Reasoning*, 355–369.
- Molineaux, M., Aha, D., & Moore, P. (2008). Learning continuous action models in a real-time strategy environment. In *Proceedings of the twenty-first annual conference of the florida artificial intelligence research society* (pp. 257–262).
- Molineaux, M., Klenk, M., & Aha, D. (2010). Goal-driven autonomy in a navy strategy simulation. In *Proceedings of the twenty-fourth aaii conference on artificial intelligence* (pp. 1548–1554).
- Muñoz-Avila, H., Aha, D., Jaidee, U., Klenk, M., & Molineaux, M. (2010). Applying goal driven autonomy to a team shooter game. In *Proceedings of the florida artificial intelligence research society conference* (pp. 465–470).
- Muñoz-Avila, H., Bauckhage, C., Bida, M., Congdon, C. B., & Kendall, G. (2013). Learning and game ai. In S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, & J. Togelius (Eds.), *Artificial and computational intelligence in games* (Vol. 6, pp. 33–43). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- Naveed, M., Crampton, A., Kitchin, D., & McCluskey, T. (2011). Real-time path planning using a simulation-based markov decision process. In *Research and development in intelligent systems xxviii: Incorporating applications and innovations in intelligent systems xix: Proceedings of ai-2011, the thirty-first sgai international conference on innov* (p. 35). New York, NY: Springer.
- Ontañón, S., Bonnette, K., Mahindrakar, P., Gómez-Martín, M., Long, K., Radhakrishnan, J., ... Ram, A. (2009). Learning from human demonstrations for real-time case-based planning. *Proceedings of the IJCAI-09 Workshop on Learning Structural Knowledge From Observations*.
- Ontañón, S., Mishra, K., Sugandh, N., & Ram, A. (2007). Case-based planning and execution for real-time strategy games. *Case-Based Reasoning Research and Development*, 164–178.
- Ontañón, S., Mishra, K., Sugandh, N., & Ram, A. (2010). On-line case-based planning. *Computational Intelligence*, 26(1), 84–119.
- Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., & Preuss, M. (2013). A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in Games*.
- Pearson, K. (1894). Contributions to the mathematical theory of evolution. *Philosophical Transactions of the Royal Society of London. A*, 71–110.
-

References

- Peikidis, P. (2010). *Demonstrating the use of planning in a video game* (Master's thesis). University of Sheffield.
- Peng, J., & Williams, R. J. (1994). Incremental Multi-Step Q-Learning. In *Machine learning* (pp. 226–232). Morgan Kaufmann.
- Pérez, A. U. (2011). *Multi-reactive planning for real-time strategy games* (Master's thesis). MS Thesis. Universitat Autònoma de Barcelona.
- Pfeifer, R. (1996). *Building fungus eaters: Design principles of autonomous agents*. From Animals to Animats, Cambridge, MA: MIT Press.
- Pilgrim, B. (2015). *Munkres assignment algorithm - modified for rectangular matrices*. Retrieved 28/05/2015, from <http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html>
- PLANIART Lab. (2012). *The spar ai agent*. Retrieved 28/05/2015, from <http://www.planiart.usherbrooke.ca/projects/spar/>
- Ponsen, M. (2004). *Improving adaptive game ai with evolutionary learning* (Master's thesis). Delft University of Technology.
- Ponsen, M., Spronck, P., & Tuyls, K. (2006). Hierarchical reinforcement learning with deictic representation in a computer game. In *Proceedings of the bnaic 2006*. Namur, Belgium.
- Ponsen, M. J., Muñoz-Avila, H., Spronck, P., & Aha, D. W. (2005). Automatically acquiring domain knowledge for adaptive game ai using evolutionary learning. In *Proceedings of the national conference on artificial intelligence* (Vol. 20, p. 1535).
- Rabin, S. (2002). Ai game programming wisdom. In (p. 272-281). Charles River Media, Inc.
- Ram, A., & Santamaria, J. C. (1997). Continuous case-based reasoning. *Artificial Intelligence*, 90(1), 25–77.
- Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21(4), 25–34.
- Richoux, F. (2014). *Aiur- artificial intelligence using randomness*. Retrieved 28/05/2015, from <http://code.google.com/p/aiurproject/>
- Robertson, G., & Watson, I. (2014). A review of real-time strategy game ai. *AI Magazine*, 35(4), 75–104.

-
- Rummery, G. A., & Niranjana, M. (1994). *On-line q-learning using connectionist systems* (Tech. Rep. No. CUED/F-INFENG/TR 166). Cambridge University Engineering Department. Retrieved from citeseer.ist.psu.edu/rummery94line.html
- Russell, S., & Norvig, P. (1995). *Artificial intelligence - a modern approach* (Vol. 25; Pearson, Ed.). Prentice Hall.
- Safadi, F., & Ernst, D. (2010). *Organization in ai design for real-time strategy games* (Master's thesis). Universit de Lige.
- Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3), 211-229.
- Sánchez-Ruiz, A., Lee-Urban, S., Muñoz-Avila, H., Díaz-Agudoy, B., & González-Calero, P. (2007). Game AI for a Turn-Based Strategy Game with Plan Adaptation and Ontology-based Retrieval. In *Proceedings of the icaps 2007 workshop on planning in games*.
- Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., ... Sutphen, S. (2007). Checkers is solved. *science*, 317(5844), 1518-1522.
- Shannon, C. E. (1950). *Programming a computer for playing chess*. Springer.
- Shantia, A., Begue, E., & Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micro management in starcraft. In *Proceedings of the international joint conference on neural networks (ijcnn), 2011*.
- Sharma, M., Holmes, M., Santamara, J. C., Irani, A., Jr., C. L. I., & Ram, A. (2007). Transfer learning in real-time strategy games using hybrid cbr/rl. In M. M. Veloso (Ed.), *Proceedings of the twentieth international conference on artificial intelligence (ijcai-07)* (p. 1041-1046). Retrieved from <http://dblp.uni-trier.de/db/conf/ijcai/ijcai2007.html#SharmaHSIIR07>
- Smith, M., Lee-Urban, S., & Muñoz-Avila, H. (2007). RETALIATE: Learning Winning Policies in First-Person Shooter Games. In *Proceedings of the seventeenth innovative applications of artificial intelligence conference (iaai-07)* (p. 1801-1806). AAAI Press.
- Smith, Q. (2012). *Skynet bot*. Retrieved 28/05/2015, from <http://code.google.com/p/skynetbot/>
- Smyth, B., & Cunningham, P. (1992). Déjà vu: A hierarchical case-based reasoning system for software design. In *Ecai* (Vol. 92, pp. 587-589).
-

References

- Smyth, B., Keane, M. T., & Cunningham, P. (2001). Hierarchical case-based reasoning integrating case-based and decompositional problem-solving techniques for plant-control software design. *Knowledge and Data Engineering, IEEE Transactions on*, 13(5), 793–812.
- Souto, J. H. (2007). *A Turn-Based Strategy Game Testbed for Artificial Intelligence* (Master's thesis). Lehigh University.
- Spronck, P., Sprinkhuizen-Kuyper, I., & Postma, E. (2003). Online adaptation of game opponent ai in theory and practice. In *Proceedings of the 4th international conference on intelligent games and simulation (game-on 2004)*.
- Stanley, K., Bryant, B., & Miikkulainen, R. (Dec. 2005). Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6), 653-668.
- Stanley, R. P. (1986). *What is enumerative combinatorics?* Springer.
- Stone, P. (1998). *Layered learning in multiagent systems: A winning approach to robotic soccer*. MIT Press.
- Stone, P., Sutton, R. S., & Kuhlmann, G. (2005). Reinforcement Learning for RoboCup Soccer Keepaway. *Adaptive Behavior*, 13(3), 165-188.
- Stone, P., & Veloso, M. (2000). Layered learning. In *Machine learning: Ecml 2000* (pp. 369–381). Springer.
- Stoykov, S. (2008). *Using a competitive approach to improve military simulation artificial intelligence design* (PhD thesis). Naval Postgraduate School.
- Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(2), 144–148.
- Sushmita, S., & Chaudhury, S. (2007). Hierarchical fuzzy case based reasoning with multi-criteria decision making for financial applications. In *Pattern recognition and machine intelligence* (pp. 226–234). Springer.
- Sutton, R., Szepesvári, C., & Maei, H. (2009). A convergent o(n) algorithm for off-policy temporal-difference learning with linear function approximation.
- Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1), 9–44.
- Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

- Synnaeve, G., & Bessiere, P. (2011a). A bayesian model for plan recognition in rts games applied to starcraft. In *Proceedings of the seventh artificial intelligence and interactive digital entertainment international conference (aiide 2011)*.
- Synnaeve, G., & Bessiere, P. (2011b). A bayesian model for rts units control applied to starcraft. In *Computational intelligence and games (cig), 2011 ieee symposium on*.
- Synnaeve, G., & Bessiere, P. (2012). Special tactics: a bayesian approach to tactical decision-making. In *Ieee symposium on computational intelligence and games 2012*.
- Synnaeve, G., Bessiere, P., et al. (2012). A bayesian tactician. In *Proceedings of the computer games workshop at the european conference of artificial intelligence 2012*.
- Szczepański, T. (2010). *Game ai: micromanagement in starcraft*. (Master's thesis). Norwegian University of Science and Technology.
- Szczepański, T., & Aamodt, A. (2009). Case-based reasoning for improved micromanagement in real-time strategy games. In *Proceedings of the iccb 2009 workshop on cbr for computer games*.
- Szita, I. (2012). Reinforcement learning in games. *Reinforcement Learning*, 539–577.
- teamliquid. (2010). *Mechanics of starcraft 2*. Retrieved 28/05/2015, from http://www.teamliquid.net/forum/viewmessage.php?topic_id=132171
- teamliquid. (2011). *Starcraft version history*. Retrieved 28/05/2015, from <http://wiki.teamliquid.net/starcraft/Patches>
- Tesauro, G. (1992). Temporal difference learning of backgammon strategy. In *Proceedings of the 9th international conference on machine learning 8* (p. 451-457).
- The Wargus Team. (2004). *Wargus*. Retrieved 28/05/2015, from <https://launchpad.net/wargus>
- Thorndike, E. (1911). *Animal Intelligence*. Hafner, Darien.
- Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelback, J., & Yannakakis, G. (2010). Multiobjective exploration of the starcraft map space. In *Computational intelligence and games (cig), 2010 ieee symposium on* (pp. 265–272).
- Uriarte, A., & Ontañón, S. (2012). Kiting in rts games using influence maps. In *Workshop proceedings of the eighth artificial intelligence and interactive digital entertainment conference*.

References

- Uriarte, A., & Ontañón, S. (2014). Game-tree search over high-level game states in rts games. In *Proceedings of the tenth artificial intelligence and interactive digital entertainment international conference (aiide 2014)*.
- Van Den Berg, J., Guy, S., Lin, M., & Manocha, D. (2011). Reciprocal n-body collision avoidance. *Robotics Research*, 3–19.
- Van Den Berg, J., Patil, S., Sewall, J., Manocha, D., & Lin, M. (2008). Interactive navigation of multiple agents in crowded environments. In *Proceedings of the 2008 symposium on interactive 3d graphics and games* (pp. 139–147). New York, NY: ACM.
- Van Der Heijden, M., Bakkes, S., & Spronck, P. (2008). Dynamic formations in real-time strategy games. In *Computational intelligence and games, 2008. cig'08. ieee symposium on* (pp. 47–54).
- Watkins, C. (1989). *Learning from Delayed Rewards* (PhD thesis). University of Cambridge, England.
- Weber, B. (2012). *Integrating learning in a multi-scale agent* (PhD thesis). University of California, Santa Cruz.
- Weber, B., & Mateas, M. (2009a). Case-based reasoning for build order in real-time strategy games. In *The proc. of the 24rd aaai conference on artificial intelligence* (pp. 1313–1318).
- Weber, B., & Mateas, M. (2009b). A data mining approach to strategy prediction. In *Computational intelligence and games, 2009. cig 2009. ieee symposium on* (pp. 140–147).
- Weber, B., Mateas, M., & Jhala, A. (2010a). Applying goal-driven autonomy to starcraft. In *Proceedings of the sixth conference on artificial intelligence and interactive digital entertainment*.
- Weber, B., Mateas, M., & Jhala, A. (2010b). Case-based goal formulation. In *Proceedings of the aaai workshop on goal-driven autonomy*.
- Weber, B., Mateas, M., & Jhala, A. (2011). Building human-level ai for real-time strategy games. In *2011 aaai fall symposium series*.
- Weber, B., Mawhorter, P., Mateas, M., & Jhala, A. (2010). Reactive planning idioms for multi-scale game ai. In *Computational intelligence and games (cig), 2010 ieee symposium on* (pp. 115–122).
- Weber, B., & Ontañón, S. (2010). Using automated replay annotation for case-based planning in games. In *Iccbr workshop on cbr for computer games (iccbr-games)*.

-
- Wen, H. (2004, 7). *Stratagus: Open source strategy games*. O'Reilly Media. Retrieved 28/05/2015, from <http://linuxdevcenter.com/pub/a/linux/2004/07/15/stratagus.html>
- Wender, S. (2009). *Integrating reinforcement learning into strategy games* (Master's thesis). The University of Auckland.
- Wender, S., & Watson, I. (2008). Using reinforcement learning for city site selection in the turn-based strategy game civilization iv. In *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games (CIG'08)* (p. 372-377).
- Wender, S., & Watson, I. (2012). Applying reinforcement learning to small scale combat in the real-time strategy game starcraft:broodwar. In *Computational intelligence and games (cig), 2012 IEEE Symposium on*.
- Wender, S., & Watson, I. (2014a). Combining case-based reasoning and reinforcement learning for unit navigation in real-time strategy game ai. In L. Lamontagne & E. Plaza (Eds.), *Case-based reasoning research and development* (Vol. 8765, p. 511-525). Springer International Publishing.
- Wender, S., & Watson, I. (2014b). Integrating case-based reasoning with reinforcement learning for real-time strategy game micromanagement. In *Prcai 2014: Trends in artificial intelligence* (pp. 64-76). Springer.
- Wess, S., Althoff, K., & Derwand, G. (1994). Using k-d trees to improve the retrieval step in case-based reasoning. *Topics in Case-Based Reasoning*, 167-181.
- Westwood Studios. (1995). *Command & conquer: Tiberian dawn*. Retrieved 28/05/2015, from <http://www.commandandconquer.com/en/games/bygameid/cnc>
- Whiteson, S., & Stone, P. (2003). Concurrent layered learning. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems* (pp. 193-200).
- Whiteson, S., & Stone, P. (2006). Evolutionary Function Approximation for Reinforcement Learning. *J. Mach. Learn. Res.*, 7, 877-917.
- Witten, I. H. (1977). An Adaptive Optimal Controller for Discrete-Time Markov Environments. *Information and Control*, 34, 286-295.
- Zhen, J. S., & Watson, I. (2013). Neuroevolution for micromanagement in the real-time strategy game starcraft: Brood war. In *Ai 2013: Advances in artificial intelligence* (pp. 259-270). Springer.