Contents

Glossary **ix**

1 Introduction **1**

1.1 Artificial Intelligence for Games 2

1.2 Real-Time Strategy Games 3

1.3 StarCraft 7

1.4 Learning by Observation 9

1.5 Thesis Objectives 10

1.6 Thesis Contributions 12

1.7 Thesis Outline 12

2 Background: A Review of RTS Game AI **15**

2.1 Introduction 16

2.2 Tactical Decision-Making 16

2.2.1 Reinforcement Learning 17

2.2.2 Game-Tree Search 19

2.2.3 Other Techniques 22

2.3 Strategic Decision-Making 23

2.3.1 Case-Based Planning 24

2.3.2 Hierarchical Planning 26

2.3.3 Behaviour Trees 27

2.3.4 Goal-Driven Autonomy 28

2.3.5 Automated Planning 29

2.3.6 Evolutionary Algorithms 31

2.3.7 Cognitive Architectures 32

2.3.8 Spatial Reasoning 33

2.4 Plan Recognition and Learning 36

2.4.1 Deductive 37

2.4.2 Abductive 37
2.4.3 Probabilistic 39
2.4.4 Case-Based 41
2.4.5 Learning by Observation 42
2.4.6 Learning from Demonstration 45
2.5 Open Research Areas 47
2.5.1 Game AI in Industry 47
2.5.2 Multi-Scale AI 51
2.5.3 Cooperation 52
2.5.4 Standardised Evaluation 53
2.6 Conclusion 54

3 Main Approaches and Related Work 57
3.1 Learning by Observation 57
3.1.1 Main Concepts 58
3.1.2 Motivations and Challenges 60
3.1.3 Related Work 61
3.2 Case-Based Reasoning 62
3.2.1 Main Concepts 63
3.2.2 Motivations and Challenges 63
3.2.3 Related Work 65
3.3 Behaviour Trees 67
3.3.1 Main Concepts 68
3.3.2 Motivations and Challenges 70
3.3.3 Related Work 71

4 Case-Based Reasoning for Learning by Observation in RTS Games 75
4.1 Introduction 76
4.2 Requirements 77
4.3 Method 78
4.4 Experimental Setup 83
4.5 Results 84
4.6 Discussion 85

5 An Improved Dataset for RTS Game AI Research 89
5.1 Introduction 90
5.2 Motivation 91

5.3 Requirements 93
5.4 Method 94
5.4.1 Overview 94
5.4.2 Extraction Process 95
5.4.3 Adaptive Granularity 98
5.5 Evaluation 99
5.6 Conclusions and Future Work 101

6 Data Mining RTS Game Data 105
6.1 Introduction 106
6.2 Association Rule Mining 108
6.3 Chosen Approaches 109
6.4 Alternative Rule Mining 111
6.5 Method 113
6.6 Results 114
6.7 Conclusion 115

7 Learning Behaviour Trees by Observation from RTS Game Data 117
7.1 Introduction 118
7.2 Relation to Planning Systems 119
7.3 Method 120
7.3.1 Creating the original BT 121
7.3.2 Reducing the BT 122
7.4 Experimental Setup 125
7.5 Results 127
7.6 Discussion 130
7.7 Conclusion and Future Work 132

8 Concluding Discussion 135
8.1 Discussion of Main Results 136
8.1.1 Literature Review 136
8.1.2 Investigation of Case-Based Reasoning for LBO 137
8.1.3 Improved StarCraft replay dataset 138
8.1.4 Investigation of data mining for LBO 140
8.1.5 Method for building Behaviour Trees for LBO 141
8.2 Future Directions 143

Appendix A Top-k Non-redundant Rules (TNR) output **147**

Appendix B Alternative rule mining output **149**

Bibliography **151**

Glossary

Acronyms

ABL	A Behavior Language
AGI	Artificial General Intelligence
AI	Artificial Intelligence
BT	Behaviour Tree
BWAPI	Brood War Application Programming Interface
CBP	Case-Based Planning
CBR	Case-Based Reasoning
FPS	First-Person Shooter
FSM	Finite State Machine
GDA	Goal-Driven Autonomy
GOAP	Goal-Oriented Action Planning
HTN	Hierarchical Task Network
ICCup	International Cyber Cup
jLOAF	Java Learning by ObservAtion Framework
LBO	Learning By Observation
LFD	Learning From Demonstration
MNR	Minimal Non-redundant Rules
PF	Potential Field
RL	Reinforcement Learning
RTS	Real-Time Strategy
SPMF	Sequential Pattern Mining Framework
TNR	Top-k Non-redundant Rules

Terms

agent software system capable of taking autonomous action based on its environment and internal state

bot game-playing agent

build order order in which buildings are constructed in a game, defines a strategy

choke point narrow opening between two map areas

frame simulation step in an RTS game

map two-dimensional terrain on which an RTS game plays out

micromanagement rapid low-level unit control

minimap miniature map, giving an abstracted overview of the whole map

race playable team in *StarCraft* with distinct units and structures; named Protoss, Terran, & Zerg

replay game log, can be used to re-simulate the events of a game

THE UNIVERSITY OF AUCKLAND
SCHOOL OF GRADUATE STUDIES

Co-Authorship Form

Graduate Centre
ClockTower – East Wing
22 Princes Street, Auckland
Phone: +64 9 373 7599 ext 81321
Fax: +64 9 373 7610
Email: postgraduate@auckland.ac.nz
www.postgrad.auckland.ac.nz

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.

Parts of chapter 1 and all of chapter 2 are based on:
Robertson, G. and Watson, I. (2014). A review of real-time strategy game AI. AI Magazine, 35(4):75–104

Nature of contribution by PhD candidate	All writing and research work
Extent of contribution by PhD candidate (%)	95

CO-AUTHORS

Name	Nature of Contribution
Ian Watson	Supervision, feedback

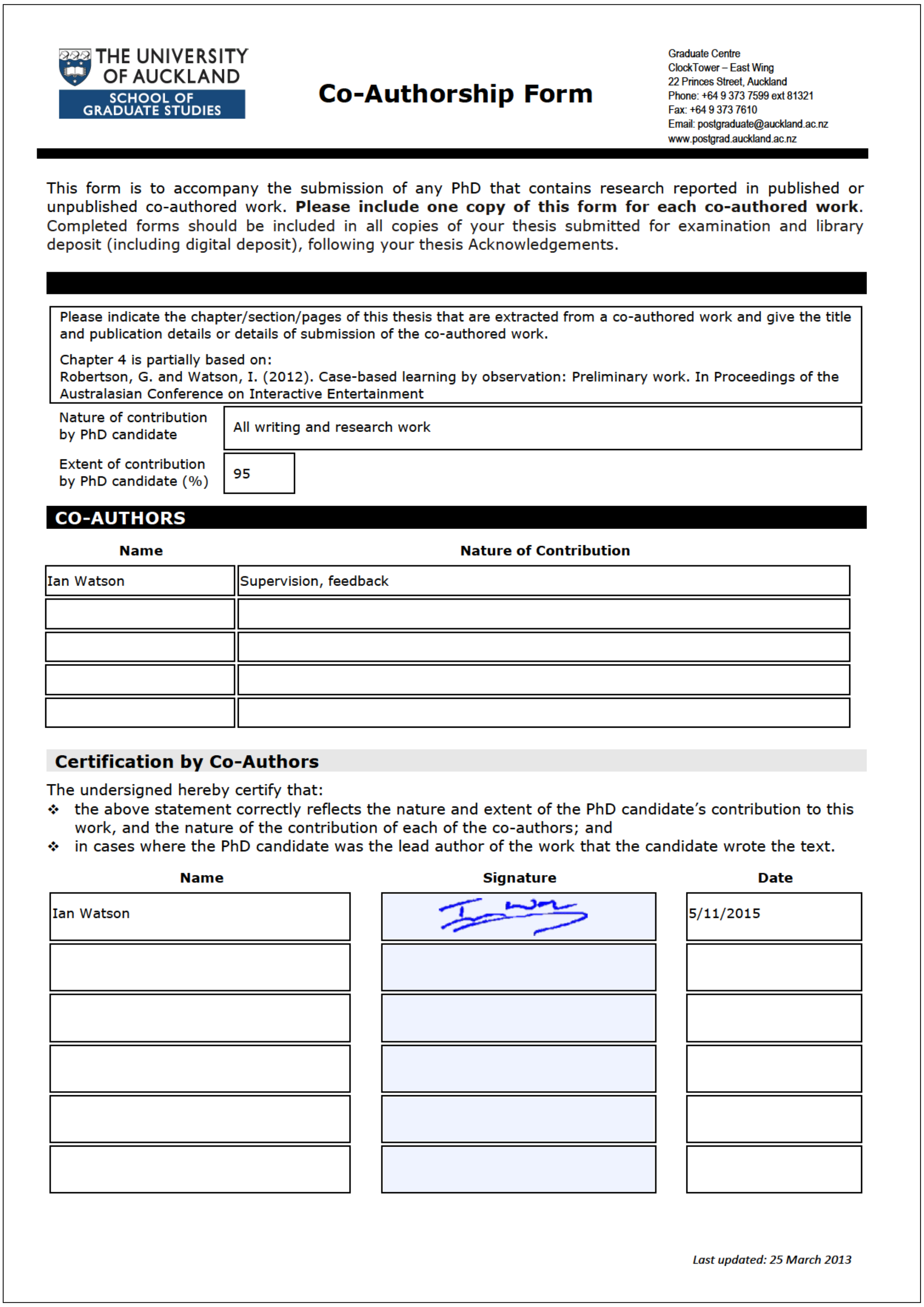
Certification by Co-Authors

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	[illegible]	5/11/2015

Last updated: 25 March 2013



THE UNIVERSITY OF AUCKLAND
SCHOOL OF GRADUATE STUDIES

Co-Authorship Form

Graduate Centre
ClockTower – East Wing
22 Princes Street, Auckland
Phone: +64 9 373 7599 ext 81321
Fax: +64 9 373 7610
Email: postgraduate@auckland.ac.nz
www.postgrad.auckland.ac.nz

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 partially based on:
Robertson, G. and Watson, I. (2012). Case-based learning by observation: Preliminary work. In Proceedings of the Australasian Conference on Interactive Entertainment

Nature of contribution by PhD candidate	All writing and research work
Extent of contribution by PhD candidate (%)	95

CO-AUTHORS

Name	Nature of Contribution
Ian Watson	Supervision, feedback

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	[illegible]	5/11/2015

Last updated: 25 March 2013

THE UNIVERSITY OF AUCKLAND
SCHOOL OF GRADUATE STUDIES

Co-Authorship Form

Graduate Centre
ClockTower – East Wing
22 Princes Street, Auckland
Phone: +64 9 373 7599 ext 81321
Fax: +64 9 373 7610
Email: postgraduate@auckland.ac.nz
www.postgrad.auckland.ac.nz

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 based on:
Robertson, G. and Watson, I. (2014). An improved dataset and extraction process for StarCraft AI. In Proceedings of the Florida Artificial Intelligence Research Society (FLAIRS) Conference

Nature of contribution by PhD candidate	All writing and research work
Extent of contribution by PhD candidate (%)	95

CO-AUTHORS

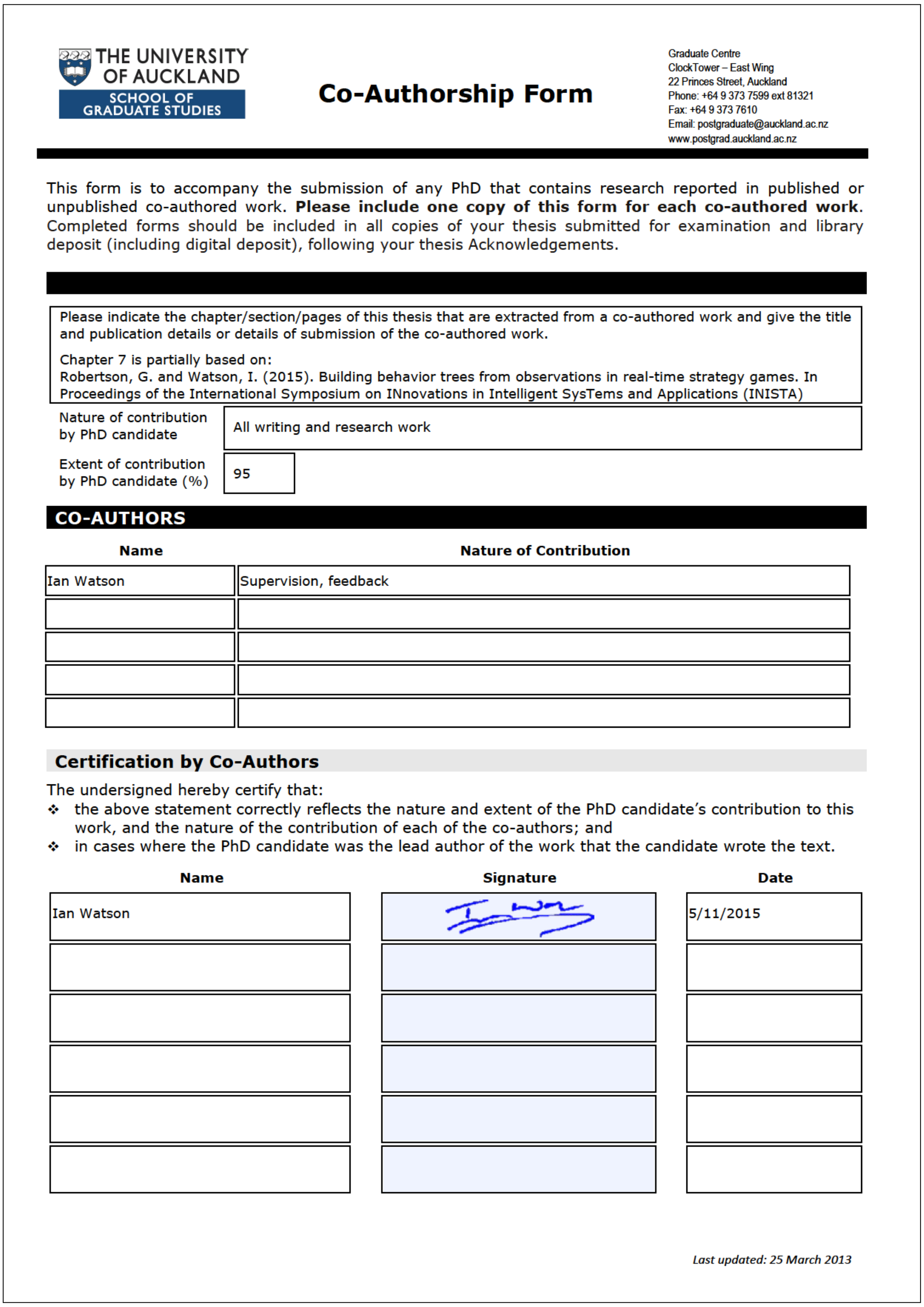
Name	Nature of Contribution
Ian Watson	Supervision, feedback

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	[illegible]	5/11/2015

Last updated: 25 March 2013



THE UNIVERSITY OF AUCKLAND
SCHOOL OF GRADUATE STUDIES

Co-Authorship Form

Graduate Centre
ClockTower – East Wing
22 Princes Street, Auckland
Phone: +64 9 373 7599 ext 81321
Fax: +64 9 373 7610
Email: postgraduate@auckland.ac.nz
www.postgrad.auckland.ac.nz

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 partially based on:
Robertson, G. and Watson, I. (2015). Building behavior trees from observations in real-time strategy games. In Proceedings of the International Symposium on INnovations in Intelligent SysTems and Applications (INISTA)

Nature of contribution by PhD candidate	All writing and research work
Extent of contribution by PhD candidate (%)	95

CO-AUTHORS

Name	Nature of Contribution
Ian Watson	Supervision, feedback

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	[illegible]	5/11/2015

Last updated: 25 March 2013

Introduction 1

Parts of this chapter are based on the following publication:
Robertson, G. and Watson, I. (2014b). A review of real-time strategy game AI. *AI Magazine*, 35(4):75–104

This thesis details an exploratory investigation into methods for automatically learning complex sequential behaviour from observations using minimal knowledge engineering effort. These methods are applied to the real-time strategy video game *StarCraft*, which has become a common testbed for artificial intelligence research in games. This chapter introduces the field of artificial intelligence for games and the domain of real-time strategy games and *StarCraft*. It describes the core challenges presented by this application area and the motivations for using *StarCraft* as a testbed for research. Next, the aims and contributions of the thesis are set out, followed by an outline of the rest of the thesis.

1.1 Artificial Intelligence for Games

Artificial Intelligence (AI) is in some sense a holy grail of Computer Science. To be able to create a machine capable of learning, reasoning, and acting of its own accord is a wondrous (and perhaps frightening) prospect. It is no surprise that people have been interested in AI since the very earliest days of computing (Turing, 1950). However, making software that people deem "intelligent" has proven to be an extremely difficult and elusive goal – especially the sort of widely-applicable intelligence known as strong AI or *Artificial General Intelligence* (AGI). Due to the difficulty of creating AGI, most AI work has focused on solving specific problems – known as weak or narrow AI. This specific focus has nonetheless led to many useful applications such as speech recognition, path finding, object identification in images, and limited personal assistant software like Siri[1] and Google Now[2].

One specific problem area that has seen major long-term interest is game-playing. The challenge in this area is to develop intelligent software agents (hereafter referred to simply as *agents*) that are able to play a particular game and win against other agents and human players. Games are an ideal domain for exploring the capabilities of AI within a constrained environment and a fixed set of rules, where problem-solving techniques can be developed and evaluated before being applied to more complex real-world problems (Schaeffer, 2001). Games naturally lend themselves to competition, which generates additional interest – especially agent versus human matches – and makes for a practical evaluation metric that guides the direction of research. AI has historically been applied to board games, such as *chess*, *Scrabble*, and *backgammon*, and competition in these games has sped the development of many AI techniques (Schaeffer, 2001). Board games present a number of challenges, often including:

large decision spaces due to the many possible actions that a player may take on each turn, compounded by the number of sequential turns the player might take. This quickly produces a very large number of possible game states.

adversarial reasoning due to the opponent(s) taking actions that affect the player, the player's decisions must take into account the expected decisions of their opponent(s). This can require complex opponent modelling.

stochasticity due to dice rolling, card drawing, or other unpredictable events. This makes it difficult to plan accurately for future actions and can also add to the difficulty of evaluating how well a player is playing.

[1]Apple - iOS - Siri: `https://www.apple.com/ios/siri/`

[2]Google Now: `https://www.google.com/landing/now/`

incomplete information due to particular parts of the game state being known only by particular players (or no players). This may prevent players from knowing exactly what other players are doing, and may force players to reason about uncertainty or attempt to infer information indirectly.

Through improvements in techniques and computer hardware, game-playing agents are now able to beat top human players at the aforementioned board games, so the focus of research has shifted to more complex board and card games like *Go*, *bridge*, and *poker* (Schaeffer, 2001) as well as video games and general game-playing. These games present more difficult versions of similar challenges – for example, *Go* has a much larger decision space than *chess*, and *bridge* and *poker* require complex adversarial reasoning with a large amount of hidden information and stochasticity (Schaeffer, 2001). In addition, many of these games present entirely new challenges, as will be detailed for real-time strategy games in the next section.

Over the past decade, there has been increasing interest in research based on video game AI, initiated by Laird and van Lent (2001) in their call for the use of video games as a testbed for AI research. They proposed video games as an area for iterative advancement in increasingly sophisticated scenarios, eventually leading to the development of human-level AI. Buro (2003) later called for increased research in real-time strategy games as a sandbox for exploring various complex challenges that are central to game AI and many other problems. They both agreed that video games are an attractive alternative to robotics for AI research because games increasingly provide a complex and realistic environment for simulation, with few of the messy properties (and cost) of real-world equipment (Buro, 2004; Laird and van Lent, 2001).

1.2 Real-Time Strategy Games

Real-Time Strategy (RTS) is a genre of video games that are essentially simplified military simulations. In an RTS game, a player indirectly controls many units and structures by issuing orders from an overhead perspective (figure 1.1) in "real-time" in order to gather resources, build an infrastructure and an army, and destroy the opposing player's forces. The real-time aspect comes from the fact that players do not take turns, but instead may perform as many actions as they are physically able to make, while the game simulation runs at a constant *frame rate* to approximate a continuous flow of time – the logical frame rate is approximately 24 frames per second at the most commonly-used game speed (fastest) in *StarCraft*[3]. Some notable RTS games include *Dune II* and the *Command & Conquer* series (Westwood Studios), *Total Annihilation* (Cavedog Entertainment), *Age of Empires* (Ensemble Studios), and the *Warcraft* and *StarCraft* series (Blizzard Entertainment).

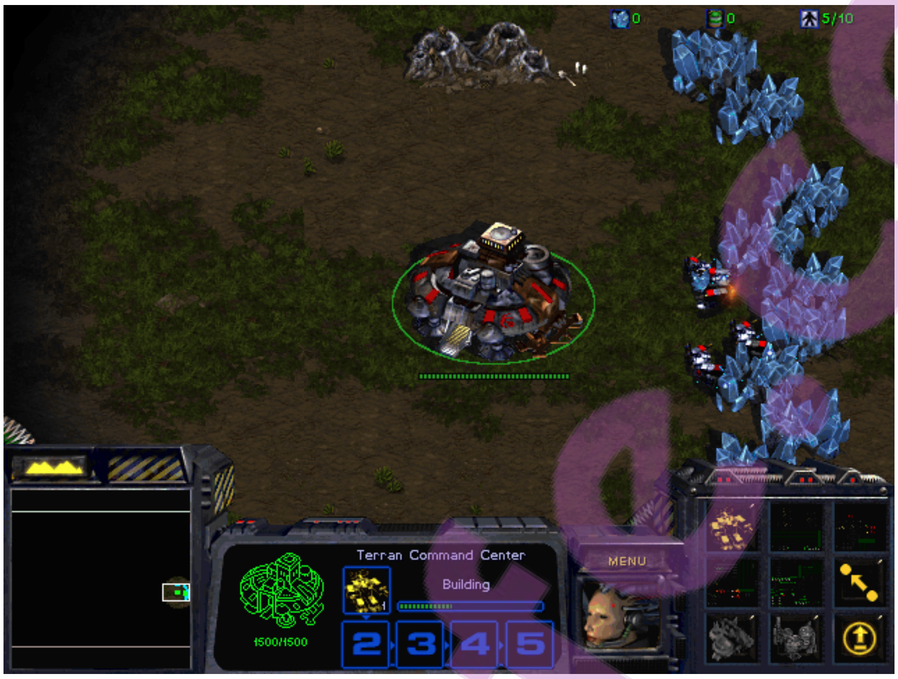


Figure 1.1: A typical match start in an RTS game. Worker units have been sent to gather resources (crystals, right) and return them to the central building. Resources (recorded top right) are being spent building an additional worker (bottom centre). "Fog of war" (left) blocks visibility away from player units.

Generally, each match in an RTS game involves two (or more) players starting with a few units and/or structures in different locations on a two-dimensonal terrain called a *map*. Resources on the map can be gathered (using a unit or structure) in order to produce additional units and structures and purchase upgrades, thus gaining access to more advanced in-game technology (units, structures, and upgrades). Additional resources and strategically important points are spread around the map, forcing players to spread out their units and structures in order to attack or defend these positions. Visibility is usually limited to a small area around player-owned units, limiting information and forcing players to conduct reconnaissance in order to respond effectively to their opponents. In most RTS games, a match ends when one player (or team) destroys all buildings belonging to the opponent player (or team),

[3]Note that this is separate from the graphical frame rate. See `https://code.google.com/p/bwapi/wiki/StarcraftGuide`

although often a player will forfeit earlier when they see they cannot win.

RTS games have a variety of military units, used by the players to wage war, as well as units and structures to aid in resource collection, unit production, and upgrades. During a match, players must balance the development of their economy, infrastructure, and upgrades with the production of military units, so they have enough units to successfully attack and defend in the present and enough resources and upgrades to succeed later. They must also decide which units and structures to produce and which technologies to advance throughout the game in order to have access to the right composition of units at the right times. This long-term high-level planning and decision-making, often called *macromanagement*, is referred to in this thesis as **strategic decision-making**. In addition to strategic decision-making, players must carefully control their units in order to maximise their effectiveness on the battlefield. Groups of units can be manoeuvred into advantageous positions on the map to surround or escape the enemy, and individual units can be controlled to attack a weak enemy unit or avoid an incoming attack. This short term control and decision-making with individual units, often called *micromanagement*, and medium-term planning with groups of units, often called *tactics*, is referred to collectively in this thesis as **tactical decision-making**.

Most RTS games include all of the game AI challenges listed above. Competitive RTS game play requires effective adversarial reasoning about possible opponent actions and strategies in order to predict attacks, strengths, and weaknesses throughout a match. This adversarial reasoning must take place under uncertainty due to the limited view each player has of the map, opponent units and structures, and players must additionally reason about the resources required to actively discover or hide information. Most RTS games also have some degree of stochasticity, such as randomised starting positions, variance in attack damage, or chances to miss on attacks.

In particular, the decision space in RTS games is massive: compared with *chess* or *Go*, most RTS games present at least an order of magnitude increase in the number of possible game states, actions to choose from, actions per game, and actions per minute (using standard rules) (Buro, 2004; Schaeffer, 2001; Synnaeve and Bessière, 2011b). This is because, in a typical RTS match, a player may be controlling hundreds of units and tens of structures, which are positioned on a map that has thousands of possible positions (or more, depending on how finely the space is divided up). Each of the units (and many of the structures) can be be issued individual orders, for example to move in a direction or attack a unit, potentially every frame (multiple times per second). Ontañón et al. (2013) estimated the state space size of *StarCraft* to be 10^{1685}

(compared to 10^{170} for *Go*), and the branching factor to be at least 10^{50} (compared to at most approximately 300 for *Go*). The decision space is so large that traditional heuristic-based search techniques, which have proven effective in a range of board games (Schaeffer, 2001), have so far been unable to solve all but the most restricted sub-problems of RTS AI.

In addition to the game AI challenges mentioned above, RTS games present a number of challenges that set them apart from the board and card games used in AI research. The main additional challenges are:

spatial & temporal reasoning about unit movement and structure positioning on the map is vital to effectively avoid or engage enemy forces, attack weak points in bases, build defensible bases, and predict or react to attacks (Buro and Furtak, 2004). This challenge is further complicated by uncertainty about an opponent's unit numbers and positioning under limited visibility.

real-time constraints due to the game simulation continuing at a constant frame rate, an agent has a very limited amount of time to consider actions before the game state changes (approximately 42ms in *StarCraft*). Some situations – tactical decision-making, in particular – require very rapid action choices in order to be effective (Buro and Furtak, 2004; Laird and van Lent, 2001).

multi-scale AI challenges because of the interacting long-term and short-term actions and effects in RTS games. For example, at the tactical decision-making level, precise timing and localised information is important, while at the strategic decision-making level, more time can be taken and precise information may be unimportant. Pursuing one goal may affect other parallel goals, such as when resources are spent on economic or military expansion. Multiple levels of abstraction and reasoning are likely required for these challenges (Buro and Furtak, 2004; Weber et al., 2010b).

state evaluation is difficult due to the complex interactions between units, armies, economy, technology, and positioning. For example, it is possible to win a battle with a smaller, lower value army if the units have particular abilities, are surrounding the enemy, are better upgraded, or are effective against the opposing unit types. This property makes search and optimisation approaches difficult (Erickson and Buro, 2014).

Due to their complexity and challenges, RTS games are probably the best current environment in which to pursue Laird & van Lent's vision of game AI as a stepping stone toward human-level AI. Within game AI, RTS games present many of the most difficult challenges, solutions for which could be highly applicable to AI operating in real-world environments. It is a particularly interesting area for AI research because, despite the apparent complexity of the task, humans are easily able to play the game, and even the best agents are outmatched by experienced humans (Buro and Churchill, 2012; Huang, 2011; Ontañón et al., 2013). This is likely due to the human abilities to abstract, reason, learn, plan, and recognise plans (Buro and Churchill, 2012).

1.3 StarCraft

The RTS game *StarCraft*[4] (figure 1.2) was chosen as the specific application for this research (unless specified otherwise, StarCraft includes its *Brood War* expansion set). StarCraft is an iconic RTS game, like *chess* is to board games, with a huge player base and numerous professional competitions. The game has three different but very well balanced teams (called *races* in StarCraft), allowing for varied strategies and tactics without any dominant strategy, and requires both strategic and tactical decision-making roughly equally (Synnaeve and Bessière, 2011b). These features give StarCraft an advantage over other RTS titles which are used for AI research, such as Wargus[5] and ORTS[6].

StarCraft was chosen due to its increasing popularity for use in RTS game AI research (figure 1.3), driven by the *Brood War Application Programming Interface* (BWAPI)[7] and the AIIDE[8] and CIG[9] StarCraft AI Competitions. BWAPI provides an interface to programmatically interact with StarCraft, allowing external code to query the game state and execute actions as if they were a player in a match (including being unable to cheat by doing anything a normal player couldn't do). The competitions pit StarCraft AI agents (called *bots*) against each other in full games of StarCraft to determine the best agents and improvements each year (Buro and Churchill, 2012) - similar to competitions in other areas of game AI that have helped to drive AI research and quickly advance the field (Schaeffer, 2001). Initially the StarCraft AI competitions also involved simplified challenges based on subtasks in the game, such as controlling a given army to defeat an opponent with an equal army, but more recent competitions have used only complete matches. For more detail on StarCraft

[4]Blizzard Entertainment: StarCraft: `http://blizzard.com/games/sc/`
[5]Wargus: `http://wargus.sourceforge.net`
[6]Open RTS: `http://skatgame.net/mburo/orts`
[7]Brood War API: `https://bwapi.github.io/`
[8]AIIDE StarCraft AI Competition: `http://www.starcraftaicompetition.com`
[9]CIG StarCraft AI Competition: `http://cilab.sejong.ac.kr/sc_competition/`

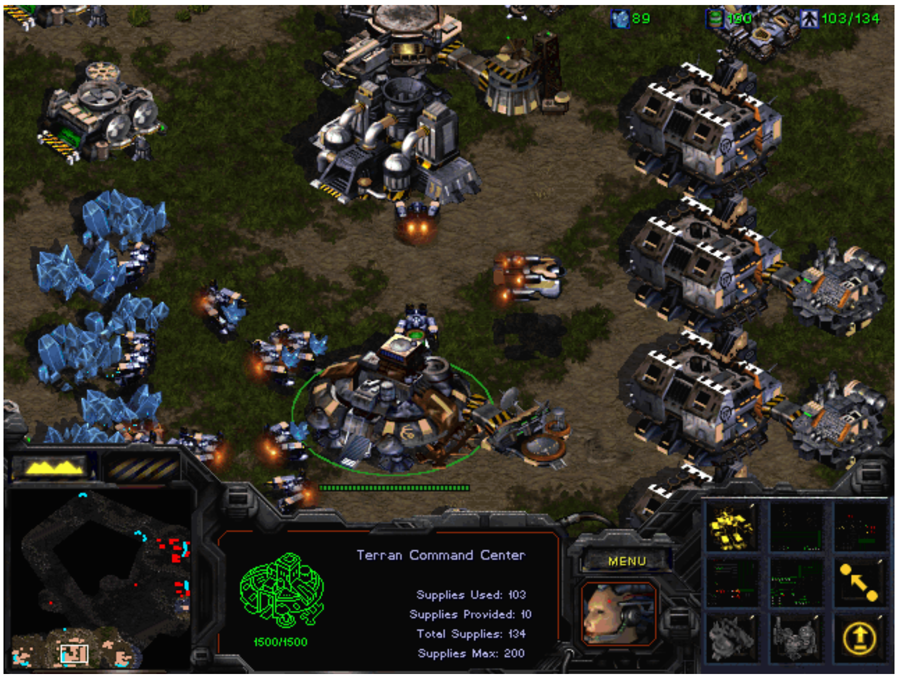


Figure 1.2: Part of a player's base in StarCraft. The white rectangle on the minimap (bottom left) is the area visible on screen. The *minimap* shows area that is unexplored (black), explored but not visible (dark), and visible (light). It also shows resources (light blue) the player's forces (tan) and last-seen enemy buildings (red).

competitions and agents, see Ontañón et al. (2013).

In order to develop AI for StarCraft, researchers have tried many different techniques (see chapter 2). A community has formed around the game as a research platform, enabling people to build on each other's work and avoid repeating the necessary groundwork before an AI system can be implemented. This work includes: a terrain analysis module (Perkins, 2010), well-documented source code for a complete, modular agent (Churchill and Buro, 2012), and preprocessed data sets assembled from thousands of professional games (Synnaeve and Bessière, 2012; Robertson and Watson, 2014a). StarCraft has a lasting popularity among professional and amateur players, including a large professional gaming scene in South Korea, with international competitions awarding millions of dollars in prizes every year (Churchill and Buro, 2011). This popularity means that there are a large number of high-quality game logs (called *replays*) available on the internet which can be used for data mining, and there are many players of all skill levels to test against (Buro and Churchill, 2012; Synnaeve and Bessière, 2011b; Weber et al., 2011a).

Despite increasing academic interest in video game AI over the past decade, and a rapidly changing game industry which often competes on new technology (Laird

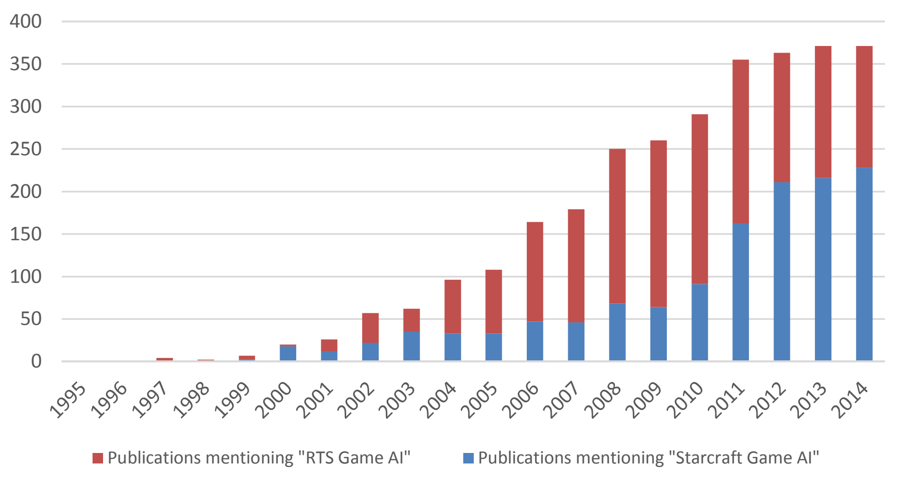


FIGURE 1.3: Number of publications mentioning "Real Time Strategy Game AI" and those mentioning "StarCraft Game AI", by year (Wender, 2015)

and van Lent, 2001), AI in video games has not improved as much as graphics, sound, animation or gameplay (Mehta et al., 2009; Tozour, 2002). Adoption of academic research in AI is slow, largely because the industry considers it to be too impractical or risky to be applied in commercial games and because the underlying goals of academic and industry game AI often differ (Baekkelund, 2006; Woodcock, 2002). In order to be applicable in industry, AI needs to be fun to play against – such as having behaviour that is reactive, varied, and difficult to repeatedly exploit – as well as being easy to apply, customisable by developers, or clearly providing some other advantage over the prevalent manually-scripted AI (see section 2.5.1). By making AI systems that are easier to apply, we may be able to encourage increased industry use of more recent AI techniques and simultaneously find many applications for AI research in games.

1.4 Learning by Observation

Learning By Observation (LBO) is an approach to machine learning in which the learning agent must learn to perform a task *solely* by observing examples of an expert completing the task. The agent is not informed of the reasoning behind the expert's actions, or any internal state of the expert – only observations of actions and the environment are available (Ontañón et al., 2011). This learning method is analogous to the way humans are thought to accelerate learning through observing an expert and emulating their actions (Mehta et al., 2009).

It is usually far easier to find or create examples of humans completing a task than to program a computer to complete the task. This is especially true for a complex domain like StarCraft with its readily available *replay* files. It can also be very time-consuming and difficult to specify a complex domain or goal such that a learning agent can reason about it, as required for unsupervised learning techniques. Therefore LBO is an ideal approach to take for learning with minimal engineering effort.

LBO does not specify any particular learning algorithm to be used, so there are examples of various methods used even within the area of RTS games – discussed in chapter 2. Various methods were explored for this thesis. A detailed description of LBO and related work to this thesis are covered in chapter 3.

1.5 Thesis Objectives

The main motivation of this research is to *make it easier to create AI systems for complex domains*, working towards improving the quality of AI used in real-world and game industry applications, especially – but not limited to – the domain of RTS games. In order to take steps toward this goal, we aim to find methods for automatically learning complex sequential behaviour using minimal knowledge engineering effort. Such capabilities would allow an agent to be easily applied to the many domains in which sequences of actions are required, without a difficult, time-consuming, and costly set-up. The research is necessarily exploratory in nature because existing approaches to machine learning have generally been applied to much simpler domains (Ontañón et al., 2013) or it requires much greater knowledge engineering effort, such as hand-crafted domain-specific decision information, domain models, or evaluation metrics (see chapter 2). This thesis details an investigation in pursuit of this overall motivation, broken up into the following objectives:

1. *Determine the feasibility of learning by observation in a complex domain using case-based reasoning.* As outlined in the section above, LBO is a potential way to vastly reduce the knowledge engineering effort required to apply AI to a new domain, if observations of an expert are available. Similarly, case-based reasoning can be used without a comprehensive domain model, provided examples of past problems and their corresponding solutions are available. For the domain of StarCraft, replays files may provide sufficient examples of a task being performed and may also work as problem and solution examples. Due to the popularity and professional competition in the game, many replays of expert players are freely available online. Thus we aim to explore the feasibility of using these replays in order to learn to play StarCraft. More information about these approaches and the reasons they were chosen can be found

in chapter 3 and the experiment and findings are described in detail in chapter 4.

2. *Produce a comprehensive dataset of professional level StarCraft play that can be used for further analysis and learning.* This is needed because, despite the wide availability of StarCraft replay files, most of the information about each match is difficult to access. StarCraft replay files store only the starting conditions and player actions in a match, but are able to reproduce the full game state when executed as a deterministic simulation within the game engine. So, in order to learn using the full game state information, not just the player actions, replays must be simulated so the data can be captured. To avoid repeatedly re-simulating replays (particularly if non-sequential access to game state information is needed) and to make this information useful for later research, it must be stored in a readily accessible format. Details of the method used to extract and store data from StarCraft replays, and the resulting dataset, are provided in chapter 5.

3. *Explore the use of data mining to automatically discover domain knowledge by observation in a complex domain.* Given a dataset of observations like that produced in objective *2*, it would be very useful to be able to apply data mining to learn more about the domain. This is especially true if the learned information is understandable to humans or if it can be used to aid subsequent machine learning – association rules could be used to infer missing information or preconditions of actions, and sequential rules could be used to predict future observations or reveal action effects. Together, a set of rules could form a partial domain model, to be used for automated planning or as short-cuts in another decision-making approach. However, it may be difficult to apply data mining to such a complex domain as StarCraft without extensive effort abstracting and selecting the data, so various promising approaches should be explored. This investigation and its results are discussed in chapter 6.

4. *Investigate offline learning by observation of a representation of sequential behaviour in a complex domain.* For this objective we have already looked at the feasibility of case-based reasoning for learning by observation (objective *1*), and have determined some issues for application areas with limited resources, such as games (due to the demand for computational resources for other aspects of the game) and robotics. In order for a method to be practically useful in such domains, the bulk of the computational effort must be done offline so that fewer resources are required at run-time. However, the behaviour learned must still be able to react to different observations. The structure used to represent sequential behaviour is a behaviour tree, which is described along with the motivations for using it in chapter 3. The exploration into representing sequential behaviour knowledge as a behaviour tree is

detailed in chapter 7.

1.6 Thesis Contributions

The four major contributions made by this thesis are as follows:

1. A **comprehensive review** of the major approaches previously investigated in the domain of real-time strategy game AI – specifically focusing on StarCraft – together with a summary of the open research areas in this domain.

2. The development and experimental evaluation of a **case-based learning by observation system using minimal knowledge engineering** for learning to play from expert examples in the domain of StarCraft. We found that case-based reasoning is able to make reasonable decisions in this complex domain, but has issues with solution length (action ordering vs reactivity), and scale (for large case bases and complex features).

3. The development and execution of a StarCraft **replay extraction process**, producing a **complete dataset of StarCraft replay information**, including detailed game state information.

4. The development and evaluation of a **new method for automatically producing a behaviour tree structure from examples of expert behaviour**, using minimal knowledge engineering, applied to the domain of StarCraft.

1.7 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 provides a comprehensive review of the algorithms and approaches that have been used in the popular domain of real-time strategy games, with a focus on StarCraft. Chapter 3 narrows down this wider body of work to the approaches chosen for attaining the thesis objectives, providing a complete description of the approaches used in this thesis and the reasoning for choosing these approaches.

Chapter 4 details the development of a case-based reasoning system able to use information from StarCraft replay files in order to play the game. This system is evaluated, resulting in a direction for subsequent chapters of the thesis: using more detailed observation information with offline analysis to learn about behaviours and the domain. Chapter 5 explains the methodology and design considerations behind the creation of a comprehensive StarCraft replay dataset. This comprehensive dataset is then used in chapter 6 to try to learn rules about the domain through data mining techniques. The dataset is also used in chapter 7 to explore the feasibility of building behaviour trees from observations of expert behaviour.

Finally, chapter 8 discusses the main results and conclusions of the thesis, and provides directions for future research.

Background: A Review of RTS Game AI 2

This chapter is based on the following publication:
Robertson, G. and Watson, I. (2014b). A review of real-time strategy game AI. *AI Magazine*, 35(4):75–104

This chapter provides a general background on *Real-Time Strategy* (RTS) game *Artificial Intelligence* (AI) concepts, approaches, and current directions. As a published work, it functions as an introduction to people new to the field of RTS game AI, helping them gain an overall understanding of work that has been done. Although the approaches used in this thesis are outlined in this chapter, they are expanded upon with further detail and related work in chapter 3.

2.1 Introduction

The field of game *Artificial Intelligence* (AI), and the *Real-Time Strategy* (RTS) video game *StarCraft* were introduced in chapter 1. StarCraft was identified as an ideal domain for exploring and improving AI approaches to a range of challenges, with rapidly growing interest from the academic community.

This chapter presents a comprehensive review of the literature on AI techniques used for StarCraft, also including research based on other RTS games in the case that significant literature based on StarCraft is not yet available in an area. It begins by outlining the different AI techniques used, grouped by the area in which they are primarily applied. These areas are tactical decision-making, strategic decision-making, plan recognition, and learning (table 2.1). This is followed by a comparison of the way game AI is used in academia and the game industry, which outlines the differences in goals and discusses the low adoption of academic research in the industry. Finally, some areas are identified in which there does not seem to be sufficient research on topics that are well-suited to study in the context of RTS game AI. This last section also proposes standardisation of the evaluation methods used in StarCraft AI research in order to make better comparison possible between papers.

Tactical Decision-Making	Strategic Decision-Making	Plan Recognition & Learning
Reinforcement Learning	Case-Based Planning	Case-Based Reasoning
- with neural networks	- selecting scripts	- learning by observation
- with case-based reasoning	- selecting plans & goals	- learning from demonstration
Evolutionary Algorithms	- with replay information	Decision Trees
- evolving scripts	- with decision trees	Hidden Markov Models
- evolving neural networks	- with fuzzy features	Bayesian Models
Game-Tree Search	- with behaviour trees	
- alpha-beta search	- with reinforcement learning	
- Monte Carlo tree search	Automated Planning	
- searching set strategies	Hierarchical Planning	
	Goal-Driven Autonomy	
	- with reinforcement learning	
	- with replay information	
	Cognitive Architectures	

Table 2.1: AI techniques used for StarCraft, grouped by the primary category in which they are used

2.2 Tactical Decision-Making

Tactical and micromanagement decisions – controlling individual units or groups of units over a short period of time – often make use of a different technique from the

AI making strategic decisions. These tactical decisions can follow a relatively simple metric, such as attempting to maximise the amount of enemy firepower which can be removed from the playing field in the shortest time (Davis, 1999). In the video game industry, it is common for simple techniques, such as finite state machines, to be used to make these decisions (Buckland, 2005). However, even in these small-scale decisions, many factors can be considered to attempt to make the best decisions possible, particularly when using units with varied abilities (figure 2.1), but the problem space is not nearly as large as that of the full game, making feasible exploratory approaches to learning domain knowledge (Weber and Mateas, 2009). There appears to be less research interest in this aspect of RTS game AI than in the area of large-scale, long-term strategic decision making and learning.

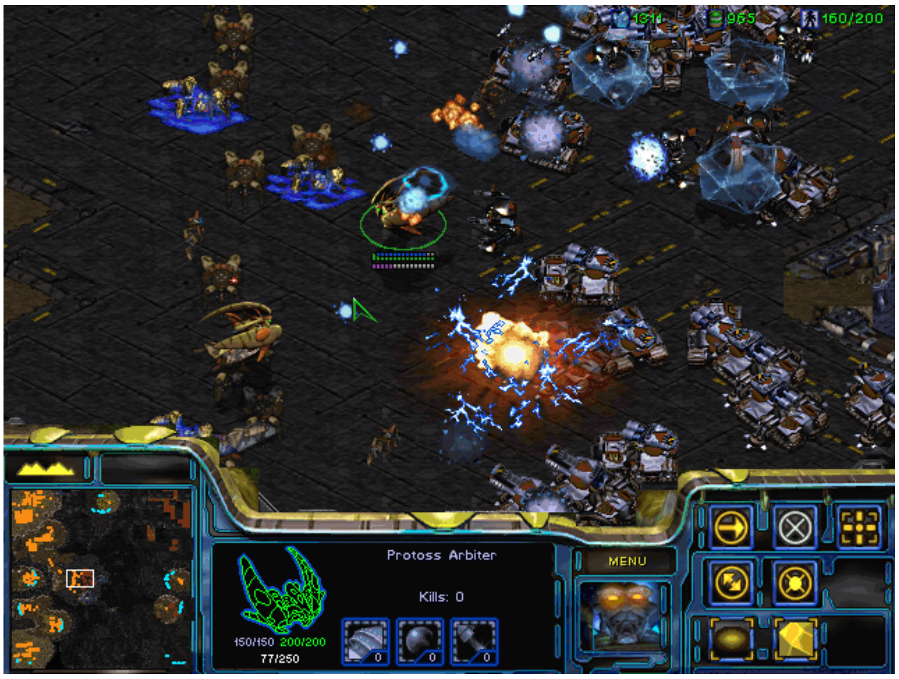


Figure 2.1: A battle in StarCraft – intense micromanagement is required to maximise the effectiveness of individual units, especially "spellcaster" units like the Protoss Arbiter

2.2.1 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning in which an agent must learn, by trial and error, optimal actions to take in particular situations order to maximise an overall reward value (Sutton and Barto, 1998). Through many iterations of weakly supervised learning, RL can discover new solutions which are better than previously known solutions. It is relatively simple to apply to a new domain, as it requires only a description of the situation and possible actions, and a reward metric (Manslow, 2004). However, in a domain as complex as an RTS game – even just for tactical decision-making – RL often requires clever state abstraction mechanisms in order to learn effectively. This technique is not commonly used for large-scale strategic decision-making, but is often applied to tactical decision-making in RTS games, likely due to the huge problem space and delayed reward inherent in strategic decisions, which make RL difficult.

RL has been applied to StarCraft by Shantia et al. (2011), where Sarsa, an algorithm for solving RL problems, is used to learn to control units in small skirmishes. They made use of artificial neural networks to learn the expected reward for attacking or fleeing with a particular unit in a given state (figure 2.2), and chose the action with the highest expected reward when in-game. The system learned to beat the inbuilt StarCraft AI scripting on average in only small three-unit skirmishes, with none of the variations learning to beat the inbuilt scripting on average in six-unit skirmishes (Shantia et al., 2011).

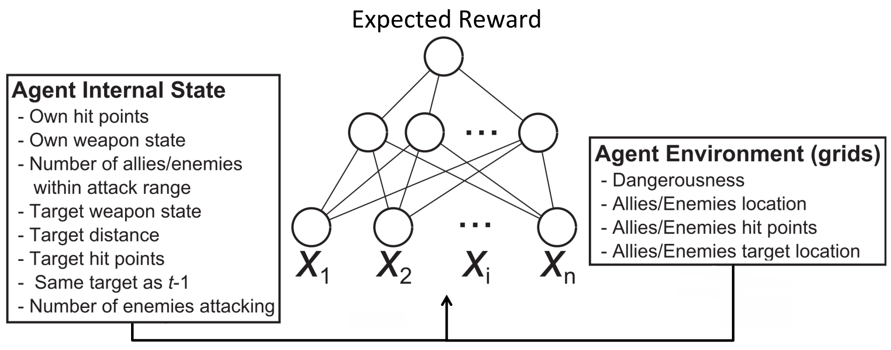

Expected Reward
Agent Internal State
- Own hit points
- Own weapon state
- Number of allies/enemies within attack range
- Target weapon state
- Target distance
- Target hit points
- Same target as t-1
- Number of enemies attacking
Agent Environment (grids)
- Dangerousness
- Allies/Enemies location
- Allies/Enemies hit points
- Allies/Enemies target location
X1 X2 Xi Xn

Figure 2.2: Game state information fed into a neural network to produce an expected reward value for a particular action. Adapted from Shantia et al. (2011)

RL techniques have also been applied to other RTS games. Sharma et al. (2007) and Molineaux et al. (2008) combine *Case-Based Reasoning* (CBR) and RL for learning tactical-level unit control in MadRTS[1] (for a description of CBR see section 2.4.4). Sharma et al. (2007) was able to increase the learning speed of the RL agent by beginning learning in a simple situation and then gradually increasing the complexity of the situation. The resulting performance of the agent was the same or better than an agent trained in the complex situation directly. Their system stores its knowledge in cases which pertain to situations it has encountered before, as in CBR. However, each case stores the expected utility for every possible action in that

[1]Mad Doc Software. Website no longer available.

situation as well as the contribution of that case to a reward value, allowing the system to learn desirable actions and situations. It remains to be seen how well it would work in a more complex domain. Molineaux et al. (2008) describe a system for RL with non-discrete actions. Their system retrieves similar cases from past experience and estimates the result of applying each case's actions to the current state. It then uses a separate case base to estimate the value of each estimated resulting state, and extrapolates around, or interpolates between, the actions to choose one which is estimated to provide the maximum value state. This technique results in a significant increase in performance when compared with one using discrete actions (Molineaux et al., 2008).

Human critique is added to RL by Judah et al. (2010) in order to learn tactical decision-making for controlling a small group of units in combat in Wargus. By interleaving sessions of autonomous state space exploration and human critique of the agent's actions, the system was able to learn a better policy in a fraction of the training iterations compared with using RL alone. However, slightly better overall results were achieved using human critique only to train the agent, possibly due to humans giving better feedback when they can see an immediate result (Judah et al., 2010).

Marthi et al. (2005) argues that it is preferable to decrease the apparent complexity of RTS games and potentially increase the effectiveness of RL or other techniques by decomposing the game into a hierarchy of interacting parts. Using this method, instead of coordinating a group of units by learning the correct combination of unit actions, each unit can be controlled individually with a higher-level group control affecting each individual's decision. Similar hierarchical decomposition appears in many RTS AI approaches because it reduces complexity from a combinatorial combination of possibilities – in this case, possible actions for each unit – down to a multiplicative combination.

2.2.2 Game-Tree Search

Search-based techniques have so far been unable to deal with the complexity of the long-term strategic aspects of RTS games, but they have been successfully applied to smaller-scale or abstracted versions of RTS combat. To apply these search methods, a simulator is usually required to allow the AI system to evaluate the results of actions very rapidly in order to explore the game tree.

Sailer et al. (2007) take a game theoretic approach by searching for the Nash equilibrium strategy among a set of known strategies in a simplified RTS. Their simplified RTS retains just the tactics aspect of RTS games by concentrating on unit group movements, so it does not require long-term planning for building infrastructure and also excludes micromanagement for controlling individual units. They use a simulation to compare the expected outcome from using each of the strategies against their opponent, for each of the strategies their opponent could be using (which is drawn from the same set), and select the Nash-optimal strategy. The simulation can avoid simulating every time-step, skipping instead to just the states in which something "interesting" happens, such as a player making a decision, or units coming into firing range of opponents. Through this combination of abstraction, state skipping, and needing to examine only the possible moves prescribed by a pair of known strategies at a time, it is usually possible to search all the way to an end-game state very rapidly, which in turn means a simple evaluation function can be used. The resulting Nash player was able to defeat each of the scripted strategies, as long as the set included a viable counter-strategy for each strategy, and it also produced better results than the max-min and min-max players (Sailer et al., 2007).

Search-based techniques are particularly difficult to use in StarCraft because of the closed-source nature of the game and inability to arbitrarily manipulate the game state. This means that the precise mechanics of the game rules are unclear, and the game cannot be easily set up to run from a particular state to be used as a simulator. Furthermore, the game must carry out expensive calculations such as unit vision and collisions, and cannot be forced to skip ahead to just the "interesting" states, making it too slow for the purpose of search (Churchill et al., 2012). In order to overcome these problems, Churchill et al. (2012) created a simulator called "SparCraft"[2] which models StarCraft and approximates the rules, but allows the state to be arbitrarily manipulated and unnecessary expensive calculations to be ignored (including skipping uninteresting states). Using this simulator and a modified version of alpha-beta search, which takes into consideration actions of differing duration, they could find effective moves for a given configuration of units. Search time was limited to approximate real-time conditions, so the moves found were not optimal. This search allowed them to win an average of 92% of randomised balanced scenarios against all of the standard scripted strategies they tested against within their simulator (Churchill et al., 2012).

Despite working very well in simulation, the results do not translate perfectly back

[2]SparCraft: `http://code.google.com/p/sparcraft/`

to the actual game of StarCraft, due to simplifications such as the lack of unit collisions and acceleration, affecting the outcome (Churchill and Buro, 2012; Churchill et al., 2012). The system was able to win only 84% of scenarios against the built-in StarCraft AI despite the simulation predicting 100%, faring the worst in scenarios which were set up to require hit-and-run behaviour (Churchill and Buro, 2012). The main limitation of this system is that due to the combinatorial explosion of possible actions and states as the number of units increases, the number of possible actions in StarCraft, and a time constraint of 5ms per logical game frame, the search will only allow up to eight units per side in a two player battle before it is too slow. On the other hand, better results may be achieved through opponent modeling, because the search can incorporate known opponent actions instead of searching through all possible opponent actions. When this was tested on the scripted strategies with a perfect model of each opponent (the scripts themselves), the search was able to achieve at least a 95% win rate against each of the scripts in simulation (Churchill et al., 2012).

2.2.2.1 Monte Carlo Planning

Monte Carlo planning has received significant attention recently in the field of computer Go, but seems to be almost absent from RTS AI, and (to the author's knowledge) completely untested in the domain of StarCraft. It involves sampling the decision space using randomly-generated plans in order to find out which plans tend to lead to more successful outcomes. It may be very suitable for RTS games because it can deal with uncertainty, randomness, large decision spaces, and opponent actions through its sampling mechanism. Monte Carlo planning has likely not yet been applied to StarCraft due to the unavailability of an effective simulator, as was the case with the search methods above, as well as the complexity of the domain. However, it has been applied to some very restricted versions of RTS games. Although both of the examples seen here are considering tactical- and unit-level decisions, given a suitable abstraction and simulation, MCTS may also be effective at strategic level decision-making in a domain as complex as StarCraft.

Chung et al. (2005) created a capture-the-flag game in which each player needed to control a group of units to navigate through obstacles to the opposite side of a map and retrieve the opponent's flag. They created a generalised Monte Carlo planning framework and then applied it to their game, producing positive results. Unfortunately, they lacked a strong scripted opponent to test against, and their system was also very reliant on heuristic evaluations of intermediate states in order to make planning decisions. Later, Balla and Fern (2009) applied the more recent technique of Upper Confidence Bounds applied to Trees (UCT) to a simplified Wargus scenario. A major benefit of their approach is that it does not require a heuristic evaluation function for intermediate states, and instead plays a game randomly out to a terminal state in order to evaluate a plan. The system was evaluated by playing against a range of scripts and a human player in a scenario involving multiple friendly and enemy groups of the basic footman unit placed around an empty map. In these experiments, the UCT system made decisions at the tactical level for moving groups of units while micromanagement was controlled by the inbuilt Wargus AI, and the UCT evaluated terminal states based on either unit hit points remaining or time taken. The system was able to win all of the scenarios, unlike any of the scripts, and to overall outperform all of the other scripts and the human player on the particular metric (either hit points or time) that it was using.

2.2.3 Other Techniques

Various other AI techniques have been applied to tactical decision-making in StarCraft. Synnaeve and Bessière (2011b) combines unit objectives, opportunities, and threats using a Bayesian model to decide which direction to move units in a battle. The model treats each of its sensory inputs as part of a probability equation which can be solved, given data (potentially learned through RL) about the distributions of the inputs with respect to the direction moved, to find the probability that a unit should move in each possible direction. The best direction can be selected, or the direction probabilities can be sampled over to avoid having two units choose to move into the same location. Their Bayesian model is paired with a hierarchical finite state machine to choose different sets of behaviour for when units are engaging or avoiding enemy forces, or scouting. The agent produced was very effective against the built-in StarCraft AI as well as its own ablated versions (Synnaeve and Bessière, 2011b).

CBR, although usually used for strategic reasoning in RTS AI (see section 2.4.4), has also been applied to tactical decision-making in Warcraft III[3], a game which has a greater focus on micromanagement than StarCraft (Szczepański and Aamodt, 2009). CBR generally selects the most similar case for reuse, but Szczepański and Aamodt (2009) added a conditional check to each case so that it could be selected only when its action was able to be executed. They also added reactionary cases which would be executed as soon as certain conditions were met. The resulting agent was able to

[3]Blizzard Entertainment: Warcraft III: `http://blizzard.com/games/war3/`

beat the built-in AI of Warcraft III in a micromanagement battle using only a small number of cases, and was able to assist human players by micromanaging battles to let the human focus on higher-level strategy.

Neuroevolution is a technique that uses an evolutionary algorithm to create or train an artificial neural network. Gabriel et al. (2012) use a neuroevolution approach called rtNEAT to evolve both the topology and connection weights of neural networks for individual unit control in StarCraft. In their approach, each unit has its own neural network that receives input from environmental sources (such as nearby units or obstacles) and hand-defined abstractions (such as the number, type, and "quality" of nearby units), and outputs whether to attack, retreat, or move left or right. During a game, the performance of the units is evaluated using a hand-crafted fitness function and poorly-performing unit agents are replaced by combinations of the best-performing agents. It is tested in very simple scenarios of 12 versus 12 units in a square arena, where all units on each side are either a hand-to-hand or ranged type unit. In these situations, it learns to beat the built-in StarCraft AI and some other agents. However, it remains unclear how well it would cope with more units or mixes of different unit types (Gabriel et al., 2012).

2.3 Strategic Decision-Making

In order to create a system which can make intelligent actions at a strategic level in an RTS game, many researchers have created planning systems. These systems are capable of determining sequences of actions to be taken in a particular situation in order to achieve specified goals. It is a challenging problem because of the incomplete information available – "fog of war" obscures areas of the battlefield that are out of sight of friendly units – as well as the huge state and action spaces and many simultaneous non-hierarchical goals. With planning systems, researchers hope to enable AI to play at a human-like level, while simultaneously reducing the development effort required when compared with the scripting commonly used in industry. The main techniques used for planning systems are *Case-Based Planning* (CBP), *Goal-Driven Autonomy* (GDA) and Hierarchical Planning.

A basic strategic decision-making system was produced in-house for the commercial RTS game Kohan II: Kings of War[4] (Dill, 2006). It assigned resources – construction, research, and upkeep capacities – to goals, attempting to maximise the total priority of the goals which could be satisfied. The priorities were set by a large

[4]TimeGate Studios: Kohan II Kings of War: `http://www.timegate.com/games/kohan-2-kings-of-war`

number of hand-tuned values, which could be swapped for a different set to give the AI different personalities (Dill, 2006). Each priority value was modified based on relevant factors of the current situation, a goal commitment value (to prevent flip-flopping once a goal has been selected) and a random value (to reduce predictability). It was found that this not only created a fun, challenging opponent, but also made the AI easier to update for changes in game design throughout the development process (Dill, 2006).

2.3.1 Case-Based Planning

CBP is a planning technique that finds similar past situations from which to draw potential solutions to the current situation. In the case of a CBP system, the solutions found are a set of potential plans or sub-plans which are likely to be effective in the current situation. CBP systems can exhibit poor reactivity at the strategic level and excessive reactivity at the action level, not reacting to high-level changes in situation until a low-level action fails, or discarding an entire plan because a single action failed (Palma et al., 2011b).

One of the first applications of CBP to RTS games was by Aha et al. (2005), who created a system which extended the "dynamic scripting" concept of Ponsen et al. (2005) to select tactics and strategy based on the current situation. Using this technique, their system was able to play against a non-static opponent instead of requiring additional training each time the opponent changed. They reduced the complexity of the state and action spaces by abstracting states into a state lattice of possible *build orders* combined with a small set of features, and abstracting actions into a set of tactics generated for each state. This allowed their system to improve its estimate of the performance of each tactic in each situation over multiple games, and eventually learn to consistently beat all of the tested opponent scripts (Aha et al., 2005).

Ontañón et al. (2007) use the ideas of behaviours, goals, and alive-conditions from *A Behavior Language* (ABL) (introduced by Mateas and Stern (2002)) combined with the ideas from earlier CBP systems to form a case-based system for playing Wargus. The cases are learned from human-annotated game logs, with each case detailing the goals a human was attempting to achieve with particular sequences of actions in a particular state. These cases can then be adapted and applied in-game to attempt to change the game state. By reasoning about a tree of goals and sub-goals to be completed, cases can be selected and linked together into plan to satisfy the overall goal of winning the game (figure 2.3). During the execution of a plan, it may be modified in order to adapt for unforeseen events or compensate for a failure to achieve a goal.

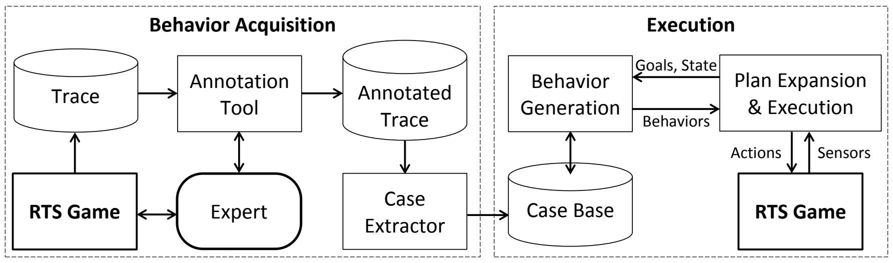


Figure 2.3: A case-based planning approach: using cases of actions extracted from annotated game logs to form plans which satisfy goals in Wargus. Adapted from Ontañón et al. (2007)

Mishra et al. (2008) extend the work of Ontañón et al. (2007) by adding a decision tree model to provide faster and more effective case retrieval. The decision tree is used to predict a high-level "situation", which determines the attributes and attribute weights to use for case selection. This helps by skipping unnecessary attribute calculations and comparisons, and emphasising important attributes. The decision tree and weightings are learned from game logs which have been human-annotated to show the high-level situation at each point throughout the games. This annotation increased the development effort required for the AI system but successfully provided better and faster case retrieval than the original system (Mishra et al., 2008).

More recent work using CBP tends to focus on the learning aspects of the system instead of the planning aspects. As such, it is discussed further in section 2.4.

A different approach is taken by Cadena and Garrido (2011), who combine the ideas of CBR with those of fuzzy sets, allowing the reasoner to abstract state information by grouping continuous feature values. This allows them to vastly simplify the state space, and it may be a closer representation of human thinking, but could potentially result in the loss of important information. For strategic decision-making, their system uses regular cases made up of exact unit and building counts, and selects a plan made up of five high-level actions, such as creating units or buildings. But for tactical reasoning (micromanagement is not explored), their system maintains independent fuzzy state descriptions and carries out independent CBR for each region of the map, thus avoiding reasoning about the map as a whole at the tactical level. Each region's state includes a linguistic fuzzy representation of its area (for example, small, medium, big), choke points, military presence, combat intensity, lost units, and amounts of each friendly and enemy unit type (for example, none, few, many). After building the case base from just one replay of a human playing against the inbuilt AI, the system was able to win around 60% of games (and tie in about 15%) against the AI on the same map. However, it is unclear how well the system would fare at the task of playing against different races (unique playable teams) and strategies, or playing on different maps.

2.3.2 Hierarchical Planning

By breaking up a problem hierarchically, planning systems are able to deal with parts of the situation separately at different levels of abstraction, reducing the complexity of the problem, but creating a potential new issue in coordination between the different levels (Marthi et al., 2005; Weber et al., 2010b). A hierarchical plan maps well to the hierarchy of goals and sub-goals typical in RTS games, from the highest level goals such as winning the game, to the lowest level goals which map directly to in-game actions. Some researchers formalise this hierarchy into the well-defined structure of a *Hierarchical Task Network* (HTN), which contains tasks, their ordering, and methods for achieving them. High-level, complex tasks in an HTN may be decomposed into a sequence of simpler tasks, which themselves can be decomposed until each task represents a concrete action (Muñoz-Avila and Aha, 2004).

HTNs have been used for strategic decision-making in RTS games, but not for StarCraft. Muñoz-Avila and Aha (2004) focus on the explanations that an HTN planner is able to provide to a human querying its behaviour, or the reasons underlying certain events, in the context of an RTS game. Laagland (2008) implements and tests an agent capable of playing an open source RTS called Spring[5] using a hand-crafted HTN. The HTN allows the agent to react dynamically to problems, such as rebuilding a building that is lost or gathering additional resources of a particular type when needed, unlike the built-in scripted AI. Using a balanced strategy, the HTN agent usually beats the built-in AI in Spring, largely due to better resource management. Efforts to learn HTNs, such as Nejati et al. (2006), have been pursued in much simpler domains, but but never directly used in the field of RTS AI. This area may hold promise in the future for reducing the work required to build HTNs.

An alternative means of hierarchical planning was used by Weber et al. (2010b). They use an active behaviour tree in ABL, which has parallel, sequential and conditional behaviours and goals in a tree structure (figure 2.4) very similar to a behaviour

[5]Spring RTS: http://springrts.com

tree (see section 2.3.3). However, in this model, the tree is expanded during execution by selecting behaviours (randomly, or based on conditions or priority) to satisfy goals, and different behaviours can communicate indirectly by reading or writing information on a "shared whiteboard". Hierarchical planning is often combined as part of other methods, such as how Ontañón et al. (2007) use a hierarchical CBP system to reason about goals and plans at different levels.

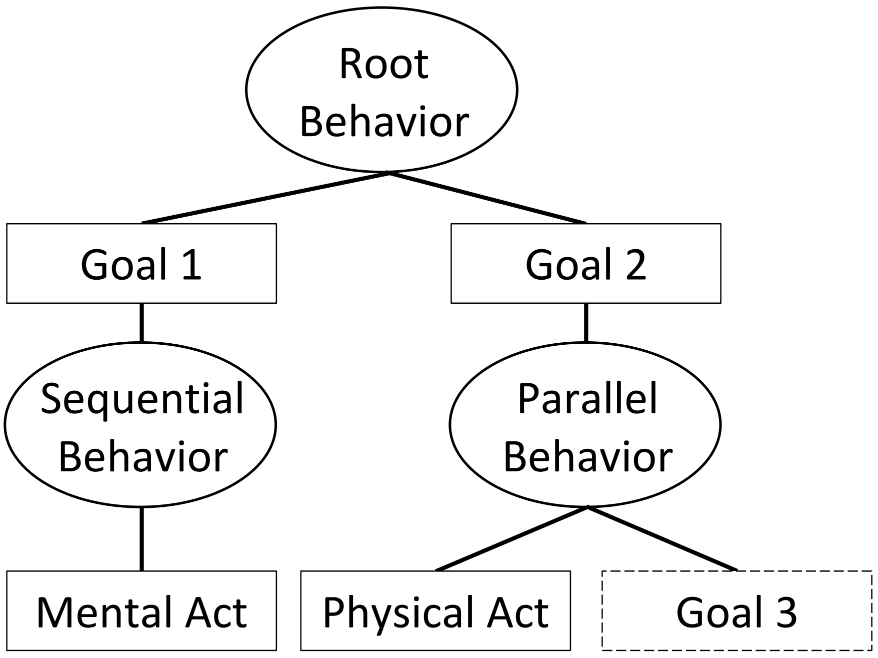


Figure 2.4: A simple active behaviour tree used for hierarchical planning, showing mental acts (calculation or processing), physical acts (in-game actions), and an unexpanded goal. Adapted from Weber et al. (2010b)

2.3.3 Behaviour Trees

Behaviour trees are hierarchies of decision and action nodes which are commonly used by programmers and designers in the game industry in order to define "behaviours" (effectively a partial plan) for agents (Palma et al., 2011b). They have become popular because, unlike scripts, they can be created and edited using visual tools, making them much more accessible and understandable to non-programmers (Palma et al., 2011b). Additionally, their hierarchical structure encourages reuse as a tree defining a specific behaviour can be attached to another tree in multiple positions, or can be customised incrementally by adding nodes (Palma et al., 2011b). Because behaviour trees are hierarchical, they can cover a wide range of behaviour, from very low-level actions to strategic-level decisions. Palma et al. (2011b) uses behaviour trees to enable direct control of a case-based planner's behaviour. With their system, machine learning can be used to create complex and robust behaviour through the planner, while allowing game designers to change specific parts of the behaviour by substituting a behaviour tree instead of an action or a whole plan. This means they can define custom behaviour for specific scenarios, fix incorrectly learned behaviour, or tweak the learned behaviour as needed.

2.3.4 Goal-Driven Autonomy

GDA is a model in which "an agent reasons about its goals, identifies when they need to be updated, and changes or adds to them as needed for subsequent planning and execution" (Molineaux et al., 2010). This addresses the high- and low-level reactivity problem experienced by CBP by actively reasoning about and reacting to why a goal is succeeding or failing.

Weber et al. (2010a) describe a GDA system for StarCraft using ABL, which is able to form plans with expectations about the outcome. If an unexpected situation or event occurs, the system can record it as a discrepancy, generate an explanation for why it occurred, and form new goals to revise the plan, allowing the system to react appropriately to unforeseen events (figure 2.5). It is also capable of simultaneously reasoning about multiple goals at differing granularity. It was initially unable to learn goals, expectations, or strategies, so this knowledge had to be input and updated manually, but later improvements allowed these to be learned from demonstration (discussed further in section 2.4.6) (Weber et al., 2012). This system was used in the *Artificial Intelligence and Interactive Digital Entertainment* (AIIDE) StarCraft AI competition entry EISBot and was also evaluated by playing against human players on a competitive StarCraft ladder called *International Cyber Cup* (ICCup)[6], where players are ranked based on their performance – it attained a ranking indicating it was better than 48% of the competitive players (Weber et al., 2010a,b).

Jaidee et al. (2011) integrate CBR and RL to make a learning version of GDA, allowing their system to improve its goals and domain knowledge over time. This means that less work is required from human experts to specify possible goals, states, and other domain knowledge because missing knowledge can be learned automatically. Similarly, if the underlying domain changes, the learning system is able to adapt to the changes automatically. However, when applied to a simple domain, the

[6]International Cyber Cup: `http://www.iccup.com`

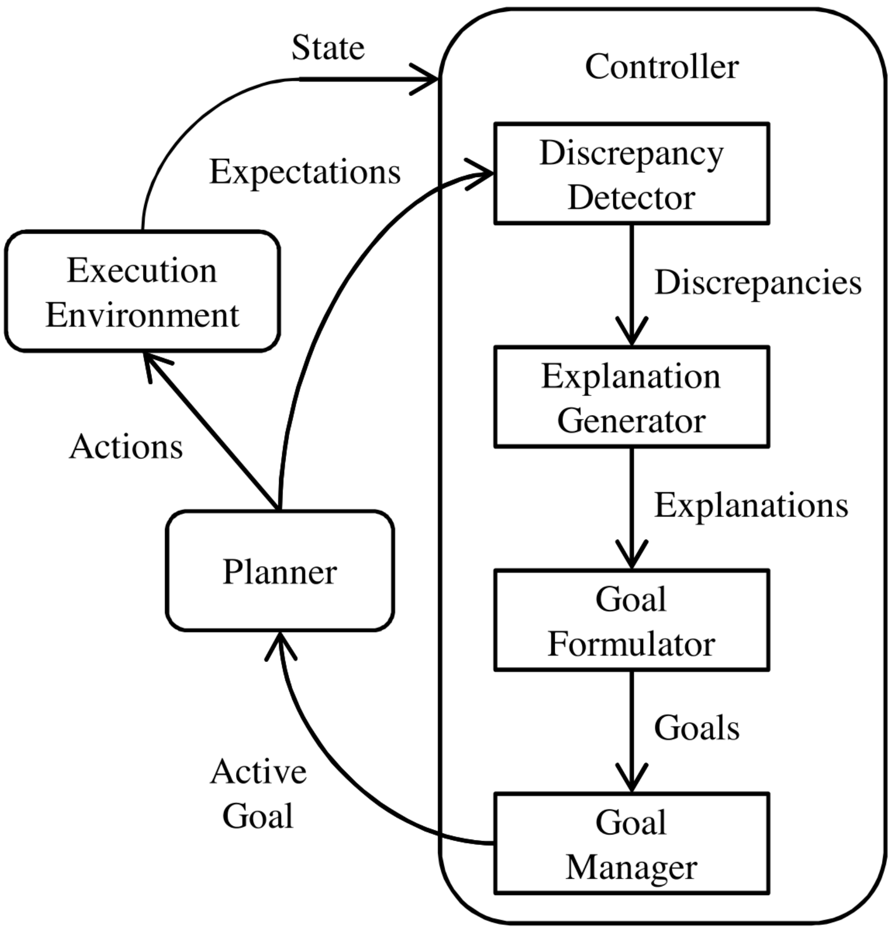


FIGURE 2.5: GDA conceptual model: a planner produces actions and expectations from goals, and unexpected outcomes result in additional goals being produced (Weber et al., 2012)

system was unable to beat the performance of a non-learning GDA agent (Jaidee et al., 2011).

2.3.5 Automated Planning

Automated planning and scheduling is a branch of classic AI research from which heuristic state space planning techniques have been adapted for planning in RTS game AI. In these problems, an agent is given a start and goal state, and a set of actions which have preconditions and effects. The agent must then find a sequence of actions to achieve the goal from the starting state. Existing RTS applications add complexity to the basic problem by dealing with durative and parallel actions, integer-valued state variables, and tight time constraints.

Automated planning ideas have already been applied successfully to commercial

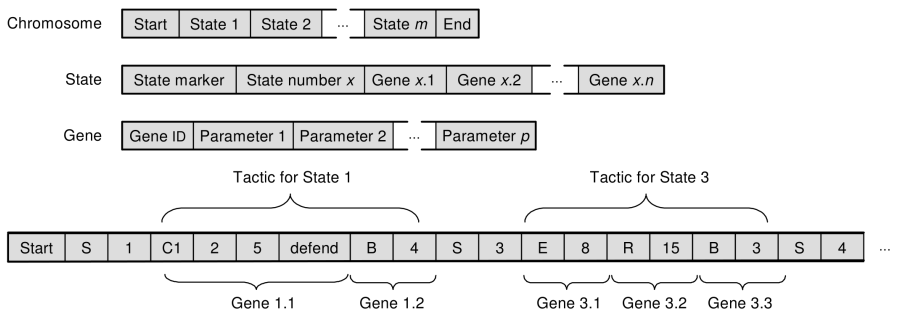


Figure 2.6: Design of a chromosome for evolving RTS game AI strategies (Ponsen et al., 2005)

First-Person Shooter (FPS) games within an architecture called *Goal-Oriented Action Planning* (GOAP). GOAP allows agents to automatically select the most appropriate actions for their current situation in order to satisfy a set of goals, ideally resulting in more varied, complex, and interesting behaviour, while keeping code more reusable and maintainable (Orkin, 2004). However, GOAP requires a large amount of domain engineering to implement, and is limited because it maps states to goals instead of to actions, so the planner cannot tell if achieving goals is going to plan, failing, or has failed (Orkin, 2004; Weber et al., 2010a). Furthermore, Champandard (2011) states that GOAP has now turned out to be a dead-end, as academia and industry have moved away from GOAP in favour of hierarchical planners to achieve better performance and code maintainability.

However, Chan et al. (2007) and Churchill and Buro (2011) use an automated planning-based approach similar to GOAP to plan *build orders* in RTS games. Unlike GOAP, they are able to focus on a single goal: finding a plan to build a desired set of units and buildings in a minimum duration (makespan). The RTS domain is simplified by abstracting resource collection to an income rate per worker, assuming building placement and unit movement takes a constant amount of time, and completely ignoring opponents. Ignoring opponents is fairly reasonable for the beginning of a game, as there is generally little opponent interaction, and doing so means the planner does not have to deal with uncertainty and external influences on the state. Both of these methods still require expert knowledge to provide a goal state for them to pursue.

The earlier work by Chan et al. (2007) uses a combination of means-ends analysis and heuristic scheduling in Wargus. Means-ends analysis produces a plan with a minimal number of actions required to achieve the goal, but this plan usually has a poor makespan because it doesn't consider concurrent actions or actions which produce greater resources. A heuristic scheduler then reorganises actions in the plan to start each action as soon as possible, adding concurrency and reducing the makespan. To consider producing additional resources, the same process is repeated with an extra goal for producing more of a resource (for each resource) at the beginning of the plan, and the plan with the shortest makespan is used. The resulting plans, though non-optimal, were found to be similar in length to plans executed by an expert player, and vastly better than plans generated by state-of-the-art general purpose planners (Chan et al., 2007).

Churchill and Buro (2011) improve upon the earlier work by using a branch-and-bound depth-first search to find optimal *build orders* within an abstracted simulation of StarCraft. In addition to the simplifications mentioned above, they avoid simulating individual time steps by allowing any action which will eventually complete without further player interaction, and jumping directly to the point at which each action completes for the next decision node. Even so, other smaller optimisations were needed to speed up the planning process enough to use in-game. The search used either the gathering time or the build time required to reach the goal (whichever was longer) as the lower bound, and a random path to the goal as the upper bound (Churchill and Buro, 2011). The system was evaluated against professional build orders seen in replays, using the set of units and buildings owned by the player at a particular time as the goal state. Due to the computational cost of planning later in the game, planning was restricted to 120 seconds ahead, with replanning every 30 seconds. This produced shorter or equal-length plans to the human players at the start of a game, and similar-length plans on average (with a larger variance) later in the game. It remains to be seen how well this method would perform for later stages of the game, as only the first 500 seconds were evaluated and searching took significantly longer in the latter half. However, this appears to be an effective way to produce near-optimal build orders for at least the early to middle game of StarCraft (Churchill and Buro, 2011).

2.3.6 Evolutionary Algorithms

Evolutionary algorithms search for an effective solution to a problem by evaluating different potential solutions and combining or randomising components of high-fitness potential solutions to find new, better solutions. This approach is used infrequently in the RTS Game AI field, but it has been effectively applied to the sub-problem of tactical decision-making in StarCraft (see section 2.2.3) and learning strategic knowledge in similar RTS titles.

Although evolutionary algorithms have not yet been applied to strategic decision-making in StarCraft, they have been applied to its sequel, StarCraft II[7]. The Evolution Chamber[8] software uses the technique to optimise partially-defined *build orders*. Given a target set of units, buildings, and upgrades to be produced by certain times in the match, the software searches for the fastest or least resource-intensive way of reaching these targets. Although there have not been any academic publications regarding this software, it gained attention by producing an unusual and highly effective plan in the early days of StarCraft II.

Ponsen et al. (2005) use evolutionary algorithms to generate strategies in a game of Wargus. To generate the strategies, the evolutionary algorithm combines and mutates sequences of tactical and strategic-level actions in the game to form scripts (figure 2.6) which defeat a set of human-made and previously-evolved scripts. The fitness of each potential script is evaluated by playing it against the predefined scripts and using the resulting in-game military score combined with a time factor which favours quick wins or slow losses. Tactics are extracted as sequences of actions from the best scripts, and are finally used in a "dynamic script" that chooses particular tactics to use in a given state, based on its experience of their effectiveness – a form of RL. The resulting dynamic scripts are able to consistently beat most of the static scripts they were tested against after learning for approximately fifteen games against that opponent, but were unable to consistently beat some scripts after more than one hundred games (Ponsen et al., 2005, 2006). A drawback of this method is that the effectiveness values learned for the dynamic scripts assumes that the opponent is static and would not adapt well to a dynamic opponent (Aha et al., 2005).

2.3.7 Cognitive Architectures

An alternative method for approaching strategic-level RTS game AI is to model a reasoning mechanism on how humans are thought to operate. This could potentially lead towards greater understanding of how humans reason and allow us to create more human-like AI. This approach has been applied to StarCraft as part of a project using the Soar cognitive architecture, which adapts the BWAPI interface to communicate with a Soar agent (Turner, 2012). It makes use of Soar's Spatial Visual System to deal with reconnaissance activities and pathfinding, and Soar's Working Memory to hold perceived and reasoned state information. However, it is currently limited to

[7]Blizzard Entertainment: StarCraft II: `http://blizzard.com/games/sc2/`
[8]Evolution Chamber: `http://code.google.com/p/evolutionchamber/`

playing a partial game of StarCraft, using only the basic Barracks and Marine units for combat, and using hard-coded locations for building placement (Turner, 2012).

A similar approach was taken by Wintermute et al. (2007) but it applied Soar to ORTS instead of StarCraft. They were able to interface the Soar cognitive architecture to ORTS by reducing the complexity of the problem using the concepts of grouping and attention for abstraction. These concepts are based on human perception, allowing the underlying Soar agent to receive information as a human would, post-perception – in terms of aggregated and filtered information. The agent could view entire armies of units as a single entity, but could change the focus of its attention, allowing it to perceive individual units in one location at a time, or groups of units over a wide area (figure 2.7). This allowed the agent to control a simple strategic-level RTS battle situation without being overwhelmed by the large number of units Wintermute et al. (2007). However, due to the limitations of Soar, the agent could pursue only one goal at a time, which would be very limiting in StarCraft and most complete RTS games.

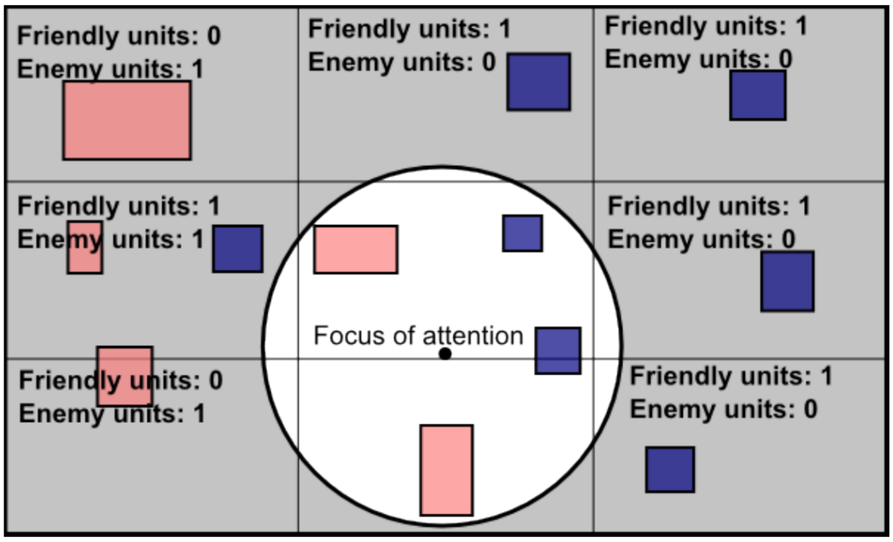


Figure 2.7: Attention limits the information the agent receives by hiding or abstracting objects further from the agent's area of focus (Wintermute et al., 2007)

2.3.8 Spatial Reasoning

RTS AI agents have to be able to reason about the positions and actions of often large numbers of hidden objects, many with different properties, moving over time, controlled by an opponent in a dynamic environment (Weber et al., 2011b; Wintermute et al., 2007). Despite the complexity of the problem, humans can reason about this information very quickly and accurately, often predicting and intercepting the location of an enemy attack or escape based on very little information, or using terrain features and the arrangement of their own units and buildings to their advantage. This makes RTS a highly suitable domain for spatial reasoning research in a controlled environment (Buro, 2004; Weber et al., 2011a; Wintermute et al., 2007).

Even the analysis of the terrain in RTS games, ignoring units and buildings, is a non-trivial task. In order to play effectively, players need to be able to know which regions of the terrain are connected to other regions, and where and how these regions connect. The connections between regions are as important as the regions themselves, because they offer defensive positions through which an army must move to get into or out of the region (choke points). Perkins (2010) describes the implementation and testing of the Brood War Terrain Analyzer, which has become a very common library for creating StarCraft agents capable of reasoning about their terrain. The library creates and prunes a Voronoi diagram using information about the walkable tiles of the map, identifies nodes as regions or choke points, then merges adjacent regions according to thresholds which were determined by trial and error to produce the desired results. The choke point nodes are converted into lines which separate the regions, resulting in a set of region polygons connected by choke points (figure 2.8). When compared against the choke points identified by humans, it had a 0–17% false negative rate, and a 4–55% false positive rate, and took up to 43 seconds to analyse the map, so there is still definite room for improvement (Perkins, 2010).

Once a player is capable of simple reasoning about the terrain, it is possible to begin reasoning about the movement of units over this terrain. A particularly useful spatial reasoning ability in RTS games is to be able to predict the location of enemy units while they are not visible to a player. Weber et al. (2011b) use a particle model for predicting enemy unit positions in StarCraft, based on the unit's trajectory and nearby choke points at the time it was seen. A single particle was used for each unit instead of a particle cloud because it is not possible to visually distinguish between two units of the same type, so it would be difficult to update the cloud if a unit was lost then re-sighted (Weber et al., 2011b). In order to account for the differences between the unit types in StarCraft, they divided the types into broad classes and learned a movement model for each class from professional *replays* on a variety of maps. The model allowed their agent to predict, with decreasing confidence over time, the subsequent locations of enemy units after sighting them, resulting in an increased win rate against other agents (Weber et al., 2011b).

The bulk of spatial reasoning research in StarCraft and other RTS games is based

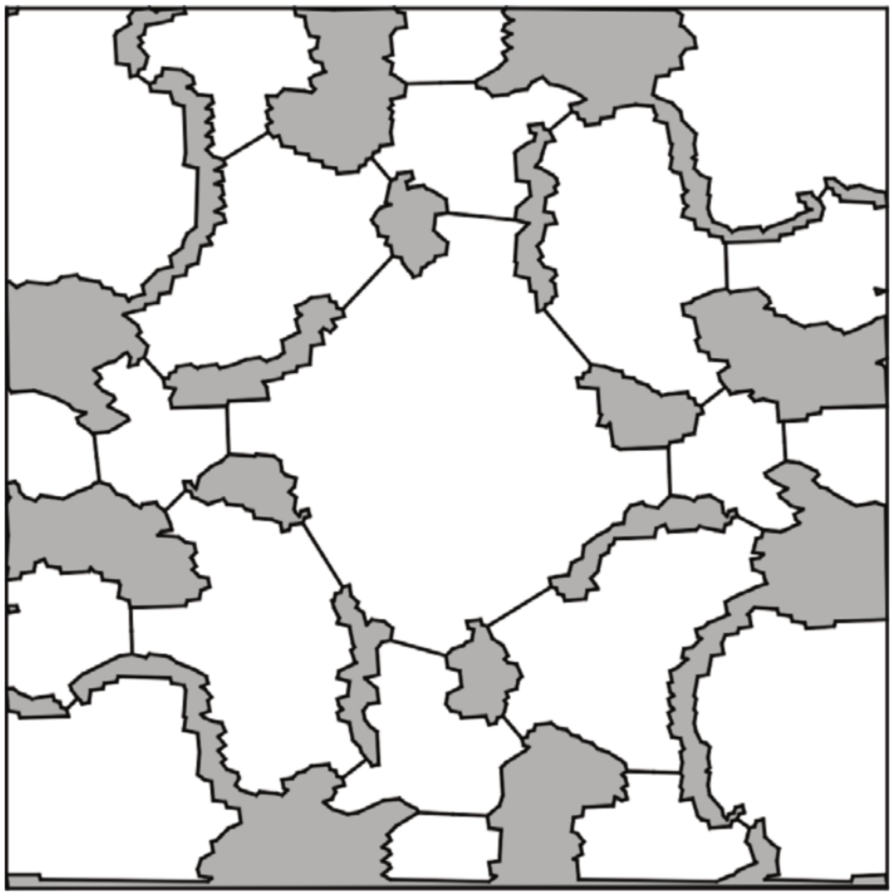


Figure 2.8: Terrain after analysis, showing impassable areas in grey and choke points as lines between white areas (Perkins, 2010)

on *Potential Fields* (PFs), and to a lesser extent, influence maps. Each of these techniques help to aggregate and abstract spatial information by summing the effect of individual points of information into a field over an area, allowing decisions to be made based on the computed field strength at particular positions. They were first applied to RTS games by Hagelbäck and Johansson (2008), before which they were used for robot navigation. Kabanza et al. (2010) uses an influence map to evaluate the potential threats and opportunities of an enemy force in an effort to predict the opponent's strategy, and Uriarte and Ontañón (2012) uses one to evaluate threats and obstacles in order to control the movement of units performing a hit-and-run behaviour known as kiting. (Baumgarten et al., 2009) uses a few different influence maps for synchronising attacks by groups of units, moving and grouping units, and choosing targets to attack. Weber and Ontañón (2010) uses PFs to aid a CBP system by taking the field strengths of many different fields at a particular position, so that the position is represented as a vector of field strengths, and can be easily compared to others stored in the case base. Synnaeve and Bessière (2011b) claims that their Bayesian model for unit movement subsumes PFs, as each unit is controlled by Bayesian sensory inputs that are capable of representing threats and opportunities in different directions relative to the unit. However, their system still needs to use damage maps in order to summarise this information for use by the sensory inputs (Synnaeve and Bessière, 2011b).

PFs were used extensively in the "Overmind" StarCraft agent, for both offensive and defensive unit behaviour (Huang, 2011). The agent used the fields to represent opportunities and threats represented by known enemy units, using information about unit statistics so that the system could estimate how beneficial and how costly it would be to attack each target. This allowed attacking units to treat the fields as attractive and repulsive forces for movement, resulting in them automatically congregating on high-value targets and avoiding defences. Additionally, the PFs were combined with temporal reasoning components, allowing the agent to consider the time cost of reaching a faraway target, and the possible movement of enemy units around the map, based on their speed and visibility. The resulting threat map was used for threat-aware pathfinding, which routed units around more threatening regions of the map by giving movement in threatened areas a higher path cost. The major difficulty they experienced in using PFs so much was in tuning the strengths of the fields, requiring them to train the agent in small battle scenarios in order to find appropriate values (Huang, 2011). To the author's knowledge, this is the most sophisticated spatial reasoning that has been applied to playing StarCraft.

2.4 Plan Recognition and Learning

A major area of research in the RTS game AI literature involves learning effective strategic-level gameplay. By using an AI system capable of learning strategies, researchers aim to make computer opponents more challenging, dynamic, and human-like, while making them easier to create (Hsieh and Sun, 2008). StarCraft is a very complex domain to learn from, so it may provide insights into learning to solve real-world problems. Some researchers have focused on the sub-problem of determining an opponent's strategy, which is particularly difficult in RTS games due to incomplete information about the opponent's actions, hidden by the "fog of war" (Kabanza et al., 2010). Most plan recognition makes use of an existing plan library to match against when attempting to recognise a strategy, but some methods allow for plan recognition without any predefined plans (Cheng and Thawonmas, 2004; Synnaeve and Bessière, 2011a). Often, data is extracted from the widely available *replays* files of expert human players, so a dataset was created in order to reduce repeated work (Synnaeve and Bessière, 2012). This section divides the plan recognition and learning methods into deductive, abductive, probabilistic, and case-based techniques. Within each technique, plan recognition can be either intended – plans are denoted for the learner and there is often interaction between the expert and the learner – or keyhole – plans are indirectly observed and there is no two-way interaction between the expert and the learner.

2.4.1 Deductive

Deductive plan recognition identifies a plan by comparing the situation with hypotheses of expected behaviour for various known plans. By observing particular behaviour a deduction can be made about the plan being undertaken, even if complete knowledge is not available. The system described by Kabanza et al. (2010) performs intended deductive plan recognition in StarCraft by matching observations of its opponent against all known strategies which could have produced the situation. It then simulates the possible plans to determine expected future actions of its opponent, judging the probability of plans based on new observations and discarding plans which do not match (figure 2.9). The method used requires significant human effort to describe all possible plans in a decision tree type structure (Kabanza et al., 2010).

The decision tree machine learning method used by Weber and Mateas (2009) is another example of intended deductive plan recognition. Using training data of building construction orders and timings which have been extracted from a large selection of StarCraft *replay* files, it creates a decision tree to predict which mid-game strategy is being demonstrated. The *replays* are automatically given their correct classification through a rule set based upon the build order. The learning process was also carried out with a nearest neighbour algorithm and a non-nested generalised exemplars algorithm. The resulting models were then able to predict the build order from incomplete information, with the nearest neighbour algorithm being most robust to incomplete information (Weber and Mateas, 2009).

2.4.2 Abductive

Abductive plan recognition identifies plans by making assumptions about the situation which are sufficient to explain the observations. The GDA system described by

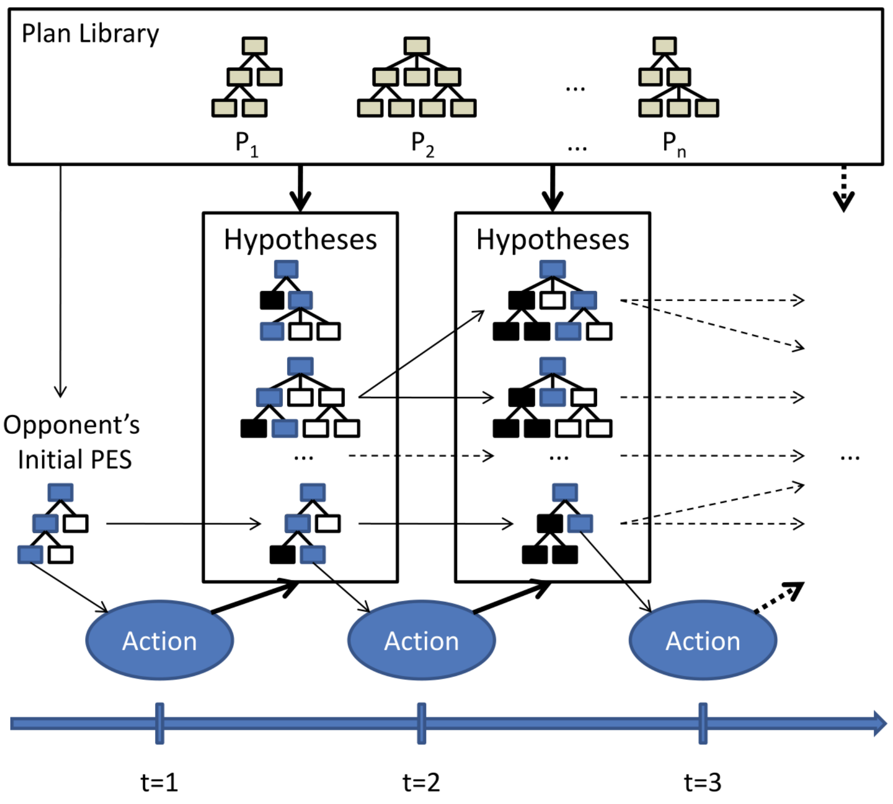


Figure 2.9: New observations update an opponent's possible Plan Execution Statuses to determine which plans are potentially being followed (Kabanza et al., 2010)

Weber et al. (2010a) is an example of intended abductive plan recognition in StarCraft, where expectations are formed about the result of actions, and unexpected events are accounted for as "discrepancies". The planner handles discrepancies by choosing from a set of predefined "explanations" which give possible reasons for discrepancies and create new goals to compensate for the change in assumed situation. This system required substantial domain engineering in order to define all of the possible goals, expectations, and explanations necessary for a domain as complex as StarCraft.

Later work added the ability for the GDA system to learn domain knowledge for StarCraft by analysing *replays* offline (Weber et al., 2012). In this modified system, a case library of sequential game states was built from the replays, with each case representing the player and opponent states as numerical feature vectors. Then case-based goal formulation was used to produce goals at run-time. The system forms predictions of the opponent's future state (referred to as explanations in this chapter) by finding a similar opponent state to the current opponent state in the case library, looking at the future of the similar state to find the difference in the feature vectors over a set period of time, and then applying this difference to the current opponent state to produce an expected opponent state. In a similar manner, it produces a goal state by finding the expected future player state, using the predicted opponent state instead of the current state in order to find appropriate reactions to the opponent. Expectations are also formed from the case library, using changes in the opponent state to make predictions about when new types of units will be produced. When an expectation is not met (within a certain tolerance for error), a discrepancy is created, triggering the system to formulate a new goal. The resulting system appeared to show better results in testing than the previous ones, but further testing is needed to determine how effectively it adapts to unexpected situations (Weber et al., 2012).

2.4.3 Probabilistic

Probabilistic plan recognition makes use of statistics and expected probabilities to determine the most likely future outcome of a given situation. Synnaeve and Bessière (2011a), Dereszynski et al. (2011), and (Hostetler et al., 2012) carry out keyhole probabilistic plan recognition in StarCraft by examining *build orders* from professional *replays*, without any prior knowledge of StarCraft build orders. This means they should require minimal work to adapt to changes in the game or to apply to a new situation, because they can learn directly from *replays* without any human input. The models learned can then be used to predict unobserved parts of the opponent's current state, or the future strategic direction of a player, given their current and past situations. Alternatively, they can be used to recognise an unusual strategy being used in a game. The two approaches differ in the probabilistic techniques that are used, the scope in which they are applied, and the resulting predictive capabilities of the systems.

Dereszynski et al. (2011) use hidden Markov models to model the player as progressing through a series of states, each of which has probabilities for producing each unit and building type, and probabilities for which state will be transitioned to next. The model is applied to one of the sides in just one of the six possible race match-ups, and to only the first seven minutes of gameplay, because strategies are less dependant on the opponent at the start of the game. State transitions happen every 30 seconds, so the timing of predicted future events can be easily found, but it is too coarse to capture the more frequent events, such as building new worker units. Without any prior information, it is able to learn a state transition graph which closely resembles the commonly-used opening build orders (figure 2.10), but a thorough analysis and evaluation of its predictive power is not provided (Dereszynski et al., 2011).

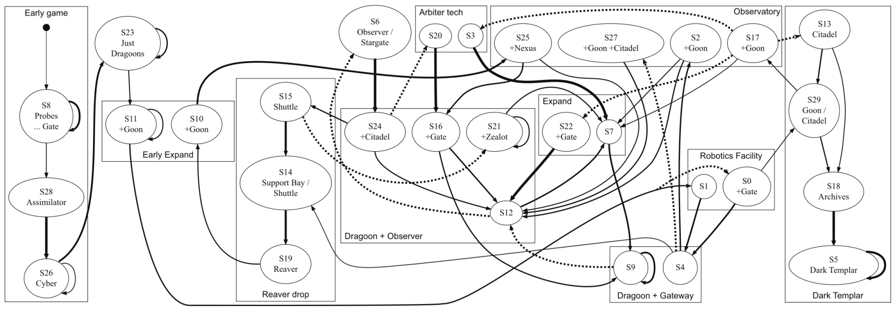


Figure 2.10: State transition graph learned in Dereszynski et al. (2011), showing transitions with probability at least 0.25 as solid edges, and higher-probability transitions with thicker edges. Dotted edges are low-probability transitions shown to make all nodes reachable. Labels in each state are likely units to be produced, while labels outside states are a human analysis of the strategy exhibited. (Dereszynski et al., 2011)

Hostetler et al. (2012) extends previous work by Dereszynski et al. (2011) using a dynamic Bayesian network model for identifying strategies in StarCraft. This model explicitly takes into account the reconnaissance effort made by the player – measured by the proportion of the opponent's main bases that has been seen – in order to determine whether a unit or building was not seen because it was not present, or because little effort was made to find it. This means that failing to find a unit can actually be very informative, provided enough effort was made. The model is also more precise than prior work, predicting exact counts and production of each unit and building type each 30 second time period, instead of just presence or absence. Production of units and buildings each time period is dependent on the current state, based on a hidden Markov model as in Dereszynski et al. (2011). Again, the model was trained and applied to one side in one race match-up, and results are shown for just the first seven minutes of gameplay. For predicting unit quantities, it outperforms a baseline predictor, which simply predicts the average for the given time period, but only after reconnaissance has begun. This highlights a limitation of the model: it cannot differentiate easily between sequential time periods with similar observations, and therefore has difficulty making accurate predictions for during and after such periods. This happens because the similar periods are modelled as a single state which has a high probability of transitioning to the same state in the next period. For predicting technology structures, the model seems to generally outperform the baseline, and in both prediction tasks it successfully incorporates negative information to infer the absence of units (Hostetler et al., 2012).

Synnaeve and Bessière (2011a) carries out a similar process using a Bayesian model instead of a hidden Markov model. When given a set of thousands of replays, the Bayesian model learns the probabilities of each observed set of buildings existing at one second intervals throughout the game. These timings for each building set are modelled as normal distributions, such that few or widely spread observations will produce a large standard deviation, indicating uncertainty (Synnaeve and Bessière, 2011a). Given a (partial) set of observations and a game time, the model can be queried for the probabilities of each possible building set being present at that time. Alternatively, given a sequence of times, the model can be queried for the most probable building sets over time, which can be used as a build order for the agent itself (Synnaeve and Bessière, 2011a). The model was evaluated and shown to be robust to missing information, producing a building set with a little over one building wrong, on average, when 80% of the observations were randomly removed. Without missing observations and allowing for one building wrong, it was able to predict almost four buildings into the future, on average (Synnaeve and Bessière, 2011a).

2.4.4 Case-Based

Plan recognition may also be carried out using *Case-Based Reasoning* (CBR) as a basis. CBR works by storing cases which represent specific knowledge of a problem and solution, and comparing new problems to past cases in order to adapt and reuse past solutions (Aamodt and Plaza, 1994). It is commonly used for learning strategic play in RTS games because it can capture complex, incomplete situational knowledge gained from specific experiences to attempt to generalise about a very large problem space, without the need to transform the data (Aamodt and Plaza, 1994; Floyd and Esfandiari, 2009; Sánchez-Pelegrín et al., 2005).

Hsieh and Sun (2008) use CBR to perform keyhole recognition of *build orders* in StarCraft by analysing *replays* of professional players, similar to Synnaeve and Bessière (2011a) above. Hsieh and Sun (2008) use the resulting case base to predict the performance of a build order by counting wins and losses seen in the professional *replays*, which allows the system to predict which build order is likely to be more successful in particular situations.

In RTS games, CBR is often not only used to recognise plans, but as part of a more general method for learning actions and the situations in which they should be applied. An area of growing interest for researchers involves learning to play RTS games from a demonstration of correct behaviour. These learning from demonstration techniques often use CBR and CBP, but they are discussed in their own section below.

Although much of the recent work using CBR for RTS games learns from demonstration, Baumgarten et al. (2009) use CBR directly without observing human play. Their system uses a set of metrics to measure performance, in order to learn to play the strategy game DEFCON[9] through an iterative process similar to RL. The system uses cases of past games played to simultaneously learn which strategic moves it should make as well as which moves its opponent is likely to make. It abstracts lower-level information about unit and structure positions by using influence maps for threats and opportunities in an area and by grouping units into fleets and metafleets. In order for it to make generalisations about the cases it has stored, it groups the cases similar to its current situation using a decision tree algorithm, splitting the cases into more or less successful games based on game score and hand-picked metrics. A path through the resulting decision tree is then used as a plan that is expected to result in a high-scoring game. Attribute values not specified by the selected plan are chosen at random, so the system tries different moves until an effective move is found. In this way, it can discover new plans from an initially empty case base.

2.4.5 Learning by Observation

For a domain as complex as RTS games, gathering and maintaining expert knowledge or learning it through trial and error can be a very difficult task, but games can provide simple access to (some of) this information through replays. Most RTS games automatically create replays, recording the events within a game and the actions taken by the players throughout the game. By analysing the replays, a system can learn from the humans' examples of correct behaviour, instead of requiring programmers to manually specify its behaviour.

Although the concept can be applied to other areas, learning by observation (as well as learning from demonstration, discussed in the next section) is particularly applicable for CBR systems. It can reduce or remove the need for a CBR system designer to extract knowledge from experts or think of potential cases and record them manually (Hsieh and Sun, 2008; Mehta et al., 2009). The *replays* can be transformed into cases for a CBR system by examining the actions players take in response to situations and events, or to complete certain predefined tasks.

In order to test the effectiveness of different techniques for Learning by Obser-

[9]Introversion Software: DEFCON: `http://www.introversion.co.uk/defcon`

vation, Floyd and Esfandiari (2009) compared CBR, decision trees, support vector machines, and naïve Bayes classifiers for a task based on RoboCup robot soccer[10]. In this task, classifiers were given the perceptions and actions of a set of RoboCup players, and were required to imitate their behaviour. There was particular difficulty in transforming the observations into a form usable by most of the the classifiers, as the robots had an incomplete view of the field, so there could be very few or many objects observed at a given time (Floyd and Esfandiari, 2009). All of the classifiers besides k-nearest neighbour – the classifier commonly used for CBR – required single-valued features or fixed-size feature vectors, so the missing values were filled with a placeholder item in those classifiers in order to mimic the assumptions of k-nearest neighbour. Classification accuracy was measured using the f-measure, and results showed that the CBR approach outperformed all of the other learning mechanisms (Floyd and Esfandiari, 2009). These challenges and results may explain why almost all research in learning by observation and learning from demonstration in the complex domain of RTS games uses CBR as a basis.

Bakkes et al. (2011) describes a case-based learning by observation system which is customised to playing Spring RTS games at a strategic level (figure 2.11), while the tactical decision-making is handled by a script. In addition to regular CBR, with cases extracted from *replays*, they record a fitness value with each state, so the system can intentionally select suboptimal strategies when it is winning in order to make the game more even (and hopefully more fun to play). This requires a good fitness metric for the value of a state, which is difficult to create for an RTS. In order to play effectively, the system uses hand-tuned feature weights on a chosen set of features, and chooses actions which are known to be effective against its expected opponent. The opponent strategy model is found by comparing observed features of the opponent to those of opponents in its case base, which are linked to the games where they were encountered. In order to make case retrieval efficient for accessing online, the case base is clustered and indexed with a fitness metric while offline. After playing a game, the system can add the replay to its case base in order to improve its knowledge of the game and opponent. A system capable of controlled adaptation to its opponent like this could constitute an interesting AI player in a commercial game (Bakkes et al., 2011).

Learning by observation also makes it possible to create a domain-independent system which can simply learn to associate sets of perceptions and actions, without knowing anything about their underlying meaning (Floyd and Esfandiari, 2010,

[10]RoboCup: `http://www.robocup.org`

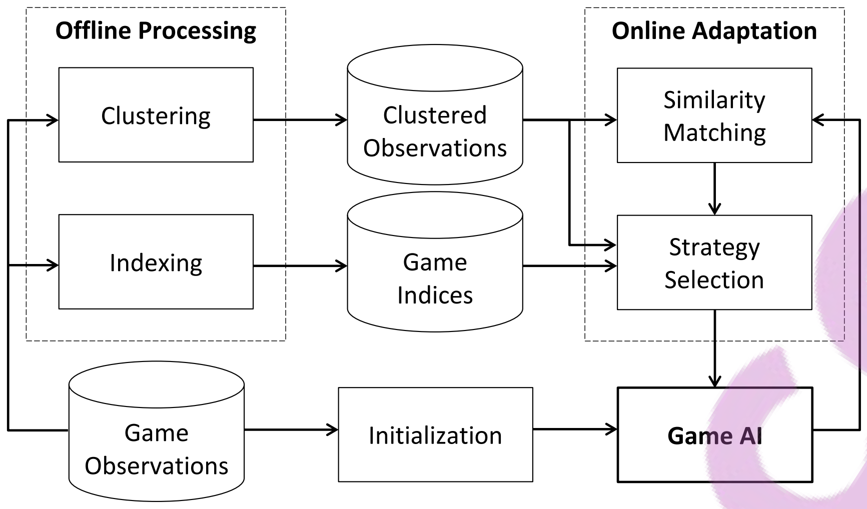


Figure 2.11: Learning by observation applied to an RTS: offline processing generalises observations, initialisation chooses an effective strategy, and online adaptation ensures cases are appropriate in the current situation. Adapted from Bakkes et al. (2011)

2011b). However, without domain knowledge to guide decisions, learning the correct actions to take in a given situation is very difficult. To compensate, the system must process and analyse observed cases, using techniques like automated feature weighting and case clustering in order to express the relevant knowledge.

Floyd and Esfandiari (2011b) claim their system is capable of handling complex domains with partial information and non-determinism, and show it to be somewhat effective at learning to play robot soccer and Tetris, but it has not yet been applied to a domain as complex as StarCraft. Their system has more recently been extended to be able to compare perceptions based on the entire sequence of perceptions – effectively a trace – so that it is not limited to purely reactive behaviour (Floyd and Esfandiari, 2011a). In the modified model, each perceived state contains a link to the previous state, so that when searching for similar states to the current state, the system can incrementally consider additional past states to narrow down a set of candidates. By also considering the similarity of actions contained in the candidate cases, the system can stop comparing past states when all of the candidate cases suggested a similar action, thereby minimising wasted processing time. In an evaluation where the correct action was dependent on previous actions, the updated system produced a better result than the original, but it is still unable to imitate an agent whose actions are based on a hidden internal state (Floyd and Esfandiari, 2011a).

2.4.6 Learning from Demonstration

Instead of learning purely from observing the traces of interaction of a player with a game, the traces may be annotated with extra information – often about the player's internal reasoning or intentions – making the demonstrations easier to learn from, and providing more control over the particular behaviours learned. Naturally, adding annotations by hand makes the demonstrations more time-consuming to author, but some techniques have been developed to automate this process. This method of learning from constructed examples is known as Learning from Demonstration. Learning from Demonstration is sometimes used synonymously with Learning by Observation in the literature. However, for clarity in this chapter, we use it to separate systems that use additional knowledge from those that use exclusively observable information. To the author's knowledge, neither approach has been used in a commercial RTS game.

Given some knowledge about the actions and tasks (things that we may want to complete) in a game, there are a variety of different methods which can be used to extract cases from a trace for use in Learning by Observation or Learning from Demonstration systems. Ontañón (2012) provides an overview of several different case acquisition techniques, from the most basic reactive and monolithic learning approaches, to more complex dependency graph learning and timespan analysis techniques. Reactive learning selects a single action in response to the current situation, while monolithic sequential learning selects an entire game plan; the first has issues with preconditions and the sequence of actions, whereas the second has issues managing failures in its long-term plan (Ontañón, 2012). Hierarchical sequential learning attempts to find a middle ground by learning which actions result in the completion of particular tasks, and which tasks' actions are subsets of other tasks' actions, making them subtasks. That way, ordering is retained, but when a plan fails it must only choose a new plan for its current task, instead of for the whole game (Ontañón, 2012).

Sequential learning strategies can alternatively use dependency graph learning, which uses known preconditions and postconditions, and observed ordering of actions, to find a partial ordering of actions instead of using the total-ordered sequence exactly as observed. However, these approaches to determining subtasks and dependencies produce more dependencies than really exist, because independent actions or tasks which coincidentally occur at a similar time will be considered dependent (Ontañón, 2012). The surplus dependencies can be reduced using timespan analysis, which removes dependencies where the duration of the action indicates that the second action started before the first one finished. In an experimental evaluation against static AI, it was found that the dependency graph and timespan analysis improved the results of each strategy they were applied to, with the best results being produced by both techniques applied to the monolithic learning strategy (Ontañón, 2012).

Mehta et al. (2009) describe a CBR and planning system which is able to learn to play the game Wargus from human-annotated *replays* of the game (figure 2.12). By annotating each *replay* with the goals which the player was trying to achieve at the time, the system can group sequences of actions into behaviours to achieve specific goals, and learn a hierarchy of goals and their possible orderings. The learned behaviours are stored in a "behaviour base" which can be used by the planner to achieve goals while playing the game. This results in a system which requires less expert programmer input to develop a game AI because it may be trained to carry out goals and behaviour (Mehta et al., 2009).

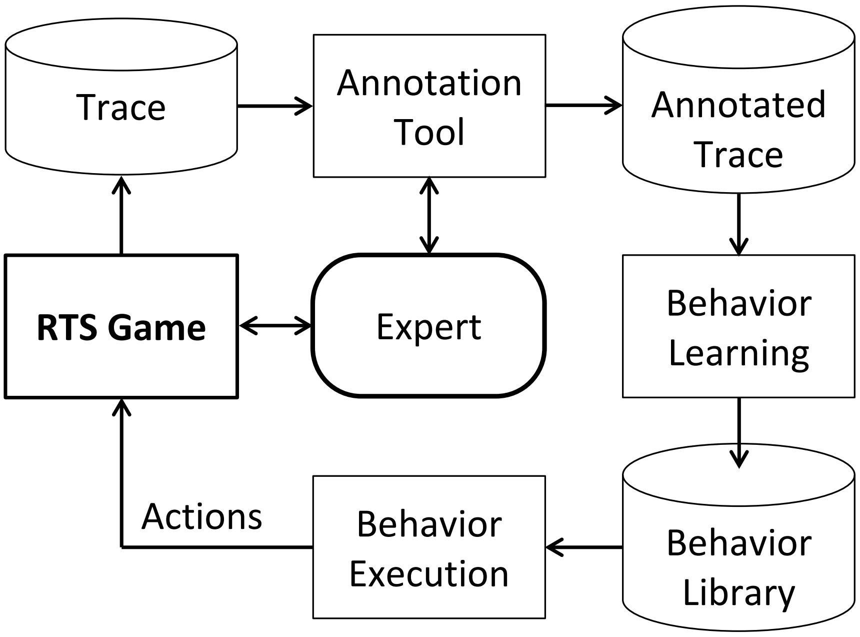


Figure 2.12: General architecture for a learning by demonstration system. Adapted from Mehta et al. (2009)

The system described by Weber and Ontañón (2010) analyses StarCraft *replays* to determine the goals being pursued by the player with each action. Using an expert-defined ontology of goals, the system learns which sequences of actions lead to goals being achieved, and in which situations these actions occurred. Thus, it can automatically annotate *replays* with the goals being undertaken at each point, and convert this knowledge into a case base which is usable in a case-based planning system. The case-based planning system produced was able to play games of StarCraft by retrieving and adapting relevant cases, but was unable to beat the inbuilt scripted StarCraft AI. Weber and Ontañón (2010) suggest that the system's capability could be improved using more domain knowledge for comparing state features and identifying goals, which would make it more specific to StarCraft but less generally applicable.

An alternative to analysing traces is to gather the cases in real-time as the game is being played and the correct behaviour is being demonstrated – known as online learning. This method has been used to train particular desired behaviours in robots learning robot soccer, so that humans could guide the learning process and apply more training if necessary (Grollman and Jenkins, 2007). The training of particular desired behaviours in this way meant that fewer training examples could be covered, so while the robot could learn individual behaviours quickly, it required being set into explicit states for each behaviour (Grollman and Jenkins, 2007).

2.5 Open Research Areas

As well as the areas covered above, most of which are actively being researched, there are some areas which are applicable to RTS AI but seem to have been given little attention. The first of these areas are found by examining the use of game AI in industry and how it differs from academic AI. The next area – multi-scale AI – has had a few contributions but have yet to be thoroughly examined, while the third – cooperation – is all but absent from the literature. Each of these three areas raises problems that are challenging for AI agents, and yet almost trivial for a human player. The final section notes the inconsistency in evaluation methods between various papers in the field, and calls for a standardised evaluation method to be put into practice.

2.5.1 Game AI in Industry

Despite the active research in the RTS AI field, there seems to be a large divide between the academic research, which uses new, complex AI techniques, and the game industry, which usually uses older and much simpler approaches. By examining the differences in academic and industry use of AI, we see new opportunities for research which benefit both groups.

Many papers reason that RTS AI research will be useful for new RTS game development by reducing the work involved in creating AI opponents, or by allowing game developers to create better AI opponents (Baekkelund, 2006; Dill, 2006;

Mehta et al., 2009; Ontañón, 2012; Ponsen et al., 2005; Tozour, 2002; Woodcock, 2002). For example, the RTS game DEFCON was given enhanced, learning AI through collaboration with the Imperial College of London (discussed in section 2.4.4) (Baumgarten et al., 2009). Similarly, Kohan II: Kings of War was produced with flexible AI through a dynamic goal selection mechanism based on complex priority calculations (discussed in section 2.3) (Dill, 2006). More recently, the 2014 RTS game Planetary Annihilation[11] is using flow fields for effective unit pathfinding with large numbers of units, and neural networks for controlling squads of units (Robbins, 2013).

In practice, however, there is very low rate of industry adoption of academic game AI research. It is typical for industry game producers to manually specify and encode the exact behaviour of their agents instead of using learning or reasoning techniques (Mehta et al., 2009; Tozour, 2002; Woodcock, 2002). Older techniques such as scripting, finite state machines, decision trees, and rule-based systems are still the most commonly used (Ontañón, 2012; Robbins, 2013; Tozour, 2002; Woodcock, 2002) – for example, the built-in AI of StarCraft uses a static script which chooses randomly among a small set of predetermined behaviours (Huang, 2011). These techniques result in game AI which often has predictable, inflexible behaviour, is subject to repeatable exploitation by humans, and doesn't learn or adapt to unforeseen situations or events (Dill, 2006; Huang, 2011; Ontañón, 2012; Woodcock, 2002).

There are two main reasons for this lack of adoption of academic AI techniques. Firstly, there is a notable difference in goals between academia and industry. Most academic work focuses on trying to create rational, optimal, agents that reason, learn, and react, while the industry aims to create challenging but defeatable opponents that are fun to play against, usually through entirely predefined behaviour (Baumgarten et al., 2009; Davis, 1999; Lidén, 2004; Ontañón, 2012; Tozour, 2002). The two aims are linked, as players find a game more fun when it is reasonably challenging (Dicken, 2011a; Hagelbäck and Johansson, 2009), but this difference in goals results in very different behaviour from the agents. An agent aiming to play an optimal strategy – especially if it is the same optimal strategy every game – is unlikely to make a desirable RTS opponent, because humans enjoy finding and taking advantage of opportunities and opponent mistakes (Schwab, 2013). An optimal agent is also trying to win at all costs, while the industry really wants game AI that is aiming to lose the game, but in a more human-like way (Davis, 1999; Schwab, 2013). Making AI that acts more human-like and intelligent – even just in specific circumstances

[11]Uber Entertainment: Planetary Annihilation: `http://www.uberent.com/pa`

through scripted behaviours – is important in the industry as it is expected to make a game more fun and interesting for the players (Lidén, 2004; Scott, 2002; Woodcock, 2002).

The second major reason for the lack of adoption is that there is little demand from the game industry for new AI techniques. Industry game developers do not view their current techniques as an obstacle to making game AI that is challenging and fun to play against, and note that it is difficult to evaluate the potential of new, untested techniques (Robbins, 2013; Schwab, 2013; Woodcock, 2002). Industry RTS games often allow AI opponents to cheat in order to make them more challenging, or emphasise playing against human opponents instead of AI (Davis, 1999; Laird and van Lent, 2001; Synnaeve and Bessière, 2011a). Additionally, game development projects are usually under severe time and resource constraints, so trying new AI techniques is both costly and risky (Buro, 2004; Robbins, 2013; Tozour, 2002). In contrast, the existing techniques are seen as predictable, reliable, and easy to test and debug (Dill, 2006; Baekkelund, 2006; Schwab, 2013; Tozour, 2002; Woodcock, 2002). Academic AI techniques are also seen as difficult to customise, tune, or tweak in order to perform important custom scripted tasks, which scripted AI is already naturally suited to doing (Robbins, 2013; Schwab, 2013).

Some new avenues of research come to light considering the use of game AI in industry. Most importantly, creating AI that is more human-like, which may also make it more fun to play against. This task could be approached by making an RTS AI that is capable of more difficult human interactions. Compared to AI, human players are good at working together with allies, using surprises, deception, distractions and coordinated attacks, planning effective strategies, and changing strategies to become less predictable (Scott, 2002). Players that are able to do at least some of these things appear to be intelligent and are more fun for human players to play against (Scott, 2002). In addition, being predictable and exploitable in the same fashion over multiple games means that human players do not get to find and exploit new mistakes, removing a source of enjoyment from the game. AI can even make mistakes and still appear intelligent as long as the mistake appears plausible in the context of the game – the sort of mistakes which a human would make (Lidén, 2004).

An alternative way to create AI that is more human-like is to replicate human play-styles and skills. Enabling an AI to replicate particular strategies – for example a heavily defensive "turtle" strategy or heavily offensive "rush" strategy – would give the AI more personality and allow players to practice against particular strategies (Schwab, 2013). This concept has been used in industry AI before (Dill, 2006) but may be difficult to integrate into more complex AI techniques. A system capable of learning from a human player – using a technique such as Learning from Demonstration (see section 2.4.6), likely using offline optimisation – could allow all or part of the AI to be trained instead of programmed (Floyd and Esfandiari, 2010; Mehta et al., 2009). Such a system could potentially copy human skills – like unit micromanagement or building placement – in order to keep up with changes in how humans play a game over time, which makes it an area of particular interest to the industry (Schwab, 2013).

Evaluating whether an RTS AI is human-like is potentially an issue. For FPS games, there is an AI competition, BotPrize[12], for creating the most human-like agents, where the agents are judged on whether they appear to be a human playing the game – a form of Turing Test (Dicken, 2011b). This test has finally been passed in 2012, with two agents judged more likely to be humans than agents for the first time. Appearing human-like in an RTS would be an even greater challenge than in an FPS, as there are more ways for the player to act and react to every situation, and many actions are much more visible than the very fast-paced transient actions of an FPS. However, being human-like is not currently a focus of any StarCraft AI research, to the author's knowledge, although it has been explored to a very small extent in the context of some other RTS games. It is also not a category in any of the current StarCraft AI competitions. The reason for this could be the increased difficulty of creating a human level agent for RTS games compared with FPS games, however, it may simply be due to an absence of goals in this area of game AI research. A Turing Test similar to BotPrize could be designed for StarCraft agents by making humans play in matches and then decide whether their opponent was a human or an agent. It could be implemented fairly easily on a competitive ladder like ICCup by simply allowing a human to join a match and asking them to judge the humanness of their opponent during the match. Alternatively, the *replay* facility in StarCraft could be used to record matches between agents and humans of different skill levels, and other humans could be given the *replays* to judge the humanness of each player. Due to the popularity of StarCraft, expert participants and judges should be relatively easy to find.

A secondary avenue of research is in creating RTS AI that is more accessible or useful outside of academia. This can partially be addressed by simply considering and reporting how often the AI can be relied upon to behave as expected, how performant the system is, and how easily the system can be tested and debugged. However,

[12]BotPrize: `http://botprize.org`

explicit research into these areas could yield improvements that would benefit both academia and industry. More work could also be done to investigate how to make complex RTS AI systems easier to tweak and customise, to produce specific behaviour while still retaining learning or reasoning capabilities. Industry feedback indicates it is not worthwhile to adapt individual academic AI techniques in order to apply them to individual games, but it may become worthwhile if techniques could be reused for multiple games in a reliable fashion. A generalised RTS AI middleware could allow greater industry adoption – games could be more easily linked to the middleware and then tested with multiple academic techniques – as well as a wider evaluation of academic techniques over multiple games. Research would be required in order to find effective abstractions for such a complex and varied genre of games, and to show the viability of this approach.

2.5.2 Multi-Scale AI

Due to the complexity of RTS games, current agents require multiple abstractions and reasoning mechanisms working in concert in order to play effectively (Churchill and Buro, 2012; Weber et al., 2010b, 2011a). In particular, most agents have separate ways of handling tactical and strategic level decision-making, as well as separately managing resources, construction, and reconnaissance. Each of these modules faces an aspect of an interrelated problem, where actions taken will have long-term strategic trade-offs affecting the whole game, so they cannot simply divide the problem into isolated or hierarchical problems. A straightforward hierarchy of command – like in a real-world military – is difficult in an RTS because the decisions of the top-level commander will depend on, and affect, multiple sub-problems, requiring an understanding of each one as well as how they interact. For example, throughout the game, resources could be spent on improving the resource generation, training units for an army, or constructing new base infrastructure, with each option controlled by a different module which cannot assess the others' situations. Notably, humans seem to be able to deal with these problems very well through a combination of on- and off-line, reactive, deliberative and predictive reasoning.

Weber et al. (2010b) defines the term "multi-scale AI problems" to refer to these challenges, characterised by concurrent and coordinated goal pursuit across multiple abstractions. They go on to describe several different approaches they are using to integrate parts of their agent. First is a working memory or "shared blackboard" concept for indirect communication between their modules, where each module publishes its current beliefs for the others to read. Next, they allow for goals and plans generated by their planning and reasoning modules to be inserted into their central reactive planning system, to be pursued in parallel with current goals and plans. Finally, they suggest a method for altered behaviour activation, so that modules can modify the preconditions for defined behaviours, allowing them to activate and deactivate behaviours based on the situation.

A simpler approach may be effective for at least some parts of an RTS agent. Synnaeve and Bessière (2011b) use a higher-level tactical command, such as scout, hold position, flock, or fight, as one of the inputs to their micromanagement controller. Similarly, Churchill and Buro (2012) use a hierarchical structure for unit control, with an overall game commander – the module which knows about the high-level game state and makes strategic decisions – giving commands to a macro commander and a combat commander, each of which give commands to their sub-commanders. Commanders further down the hierarchy are increasingly focused on a particular task, but have less information about the overall game state, so therefore must rely on their parents to make them act appropriately in the bigger picture. This is relatively effective because the control of units is more hierarchically arranged than other aspects of an RTS. Such a system allows the low-level controllers to incorporate information from their parent in the hierarchy, but they are unable to react and coordinate with other low-level controllers directly in order to perform cooperative actions (Synnaeve and Bessière, 2011b). Most papers on StarCraft AI skirt this issue by focusing on one aspect of the AI only, as can be seen in how this review paper is divided into tactical and strategic decision-making sections.

2.5.3 Cooperation

Cooperation is an essential ability in many situations, but RTS games present a particular complex environment in which the rules and overall goal are fixed, and there is a limited ability to communicate with your cooperative partner(s). It would also be very helpful in commercial games, as good cooperative players could be used for coaching or team games. In team games humans often team up to help each other with coordinated actions throughout the game, like attacking and defending, even without actively communicating. Conversely AI players in most RTS games (including StarCraft) will act seemingly independently of their teammates. A possible beginning direction for this research could be to examine some techniques developed for opponent modeling and reuse them for modeling an ally, thus giving insight into how the player should act to coordinate with the ally. Alternatively, approaches to teamwork and coordination used in other domains, such as RoboCup (Kitano et al., 1998) may be appropriate to be adapted or extended for use in the RTS domain.

Despite collaboration being highlighted as a challenging AI research problem in Buro (2003), to the author's knowledge just one research publication focusing on collaborative behaviour exists in the domain of StarCraft (and RTS games in general). Magnusson and Balsasubramaniyan (2012) modified an existing StarCraft agent to allow both communication of the bot's intentions and in-game human control of the bot's behaviour. It was tested in a small experiment in which a player is allied with the agent, with or without the communication and control elements, against two other agents. The players rated the communicating agents as more fun to play with than the non-communicating agents, and more experienced players preferred to be able to control the agent while novice players preferred a non-controllable agent. Much more research is required to investigate collaboration between humans and agents, as well as collaboration between agents only.

2.5.4 Standardised Evaluation

Despite games being a domain that is inherently suited to evaluating the effectiveness of the players and measuring performance, it is difficult to make fair comparisons between the results of most literature in the StarCraft AI field. Almost every paper has a different method for evaluating their results, and many of these experiments are of poor quality. Evaluation is further complicated by the diversity of applications, as many of the systems developed are not suited to playing entire games of StarCraft, but are suited to a specific sub-problem. Such a research community, made up of isolated studies which are not mutually comparable, was recognised as problematic by Aha and Molineaux (2004). Their Testbed for Integrating and Evaluating Learning Techniques (TIELT), which aimed to standardise the learning environment for evaluation, attempted to address the problem but unfortunately never became very widely used.

Partial systems – those that are unable to play a full game of StarCraft – are often evaluated using a custom metric, which makes comparison between such systems nearly impossible. A potential solution for this would be to select a common set of parts which could plug in to partial systems and allow them to function as a complete system for testing. This may be possible by compartmentalising parts of an open-source AI used in a StarCraft AI competition, such as UAlbertaBot (Churchill and

Buro, 2012), which is designed to be modular, or using an add-on library such as the BWAPI Standard Add-on Library (BWSAL)[13]. Alternatively, a set of common tests could be made for partial systems to be run against. Such tests could examine common sub-problems of an AI system, such as tactical decision-making, planning, and plan recognition, as separate suites of tests. Even without these tests in place, new systems should at least be evaluated against representative related systems in order to show that they represent a non-trivial improvement.

Results published about complete systems are similarly difficult to compare against one another due to their varied methods of evaluation. Some of the only comparable results come from systems demonstrated against the inbuilt StarCraft AI, despite the fact that the inbuilt AI is a simple scripted strategy which average human players can easily defeat (Weber et al., 2010a). Complete systems are more effectively tested in StarCraft AI competitions, but these are run infrequently, making quick evaluation difficult. An alternative method of evaluation is to automatically test the agents against other agents in a ladder tournament, such as in the StarCraft Brood War Ladder for BWAPI Bots[14]. In order to create a consistent benchmark of agent strength, a suite of tests could be formed from the top three agents from each of the AIIDE StarCraft competitions on a selected set of tournament maps. This would provide enough variety to give a general indication of agent strength, and it would allow for results to be compared between papers and over different years. An alternative to testing agents against other agents is testing them in matches against humans, such as how Weber et al. (2010a) tested their agent in the ICCup.

Finally, it may be useful to have a standard evaluation method for goals other than finding the AI best at winning the game. For example, the game industry would be more interested in determining the AI which is most fun to play against, or the most human-like. A possible evaluation for these alternate objectives was discussed in section 2.5.1.

2.6 Conclusion

This chapter has reviewed the literature on artificial intelligence for real-time strategy games focusing on StarCraft. It found significant research focus on tactical decision-making, strategic decision-making, plan recognition and strategy learning. Three main areas were identified where future research could have a large positive impact.

[13]BWAPI Standard Add-on Library: `http://code.google.com/p/bwsal`
[14]StarCraft Brood War Ladder for BWAPI Bots: `http://bots-stats.krasi0.com`

Firstly creating RTS AI that is more human-like would be an interesting challenge and may help to bridge the gap between academia and industry. The other two research areas discussed were noted to be lacking in research contributions, despite being highly appropriate for Real-Time Strategy game research: multi-scale AI, and cooperation. Finally, the chapter finished with a call for increased rigour and ideally standardisation of evaluation methods, so that different techniques can be compared on even ground. Overall the RTS AI field is small but very active, with the StarCraft agents showing continual improvement each year, as well as gradually becoming more based upon machine learning, learning from demonstration, and reasoning, instead of using scripted or fixed behaviours.

Main Approaches and Related Work 3

While the previous chapter gave a general overview of techniques used for *Artificial Intelligence* (AI) in *Real-Time Strategy* (RTS) games, this chapter provides more details about the specific main approaches used in this thesis and the reasons these approaches were chosen. The three areas covered by this chapter are *Learning By Observation* (LBO), *Case-Based Reasoning* (CBR), and *Behaviour Trees* (BTs).

3.1 Learning by Observation

As mentioned in chapter 1, *Learning By Observation* (LBO) is not a specific technique, but a general approach to an AI learning system: *using examples of an expert carrying out a task in order to learn to carry out the task.* With LBO, the learning agent is not provided with any background knowledge or model of the domain, any internal reasoning or state of the expert, nor with a metric to evaluate how well a task is being achieved – only the observable information from the environment and the expert's actions are known.

3.1.1 Main Concepts

References to learning a task from examples can be seen as far back in the literature as 1979 (Bauer, 1979) and it has long been applied to robotics (Argall et al., 2009; Lozano-Perez, 1983), although it is given a wide variety of names in the literature, such as *apprenticeship learning*, *imitation learning*, *behavioural cloning*, *programming by demonstration*, *learning from observation*, *learning from demonstration* and *learning by demonstration* (Argall et al., 2009; Ontañón et al., 2011). Many of these other names carry particular connotations about the learning process, application area, or information provided to the learner, but they are all considered in this thesis to be synonymous with, or relatively minor variations of LBO. The term *learning from demonstration* in particular has been widely used in robotics research (Argall et al., 2009) and recent game AI research (section 2.4.6), and usually indicates that expert demonstrations are carried out expressly for the agent to learn from, sometimes with additional non-observable information provided to the agent. Ontañón et al. (2011) notes that despite the long term use of LBO and related concepts, there is little consensus on terminology or a formalised problem definition, and provides a framework for these concepts that we adopt for this thesis[1].

Ontañón et al. (2011) characterises an LBO system by a *learning agent* **A** learning a *task* **T** by observing an *expert* (or actor) **C** performing the task in an *environment* **E** (figure 3.1). The expert takes a sequence of actions to complete the task, resulting in a *learning trace* **LT** consisting of (*timestamp*, *input variables*, *control variables*) triples (**t**, **x**, **y**) in a sequence: $LT = [(t_1, x_1, y_1), \ldots, (t_n, x_n, y_n)]$. *Input variables* (perceptions) **x** are the features of the environment observable by the agent, and *control variables* (actions) **y** are what the expert output in this environment and time. It is assumed that the agent can observe the same environmental features and take the same actions as the expert. This description is intentionally general enough to allow for discrete or continuous time domains (continuous time domains can be sampled at intervals by sensors), and for instantaneous or durative actions, parallel actions, or even continuous output signals (sampled at intervals).

The LBO problem is defined as (Ontañón et al., 2011):

- Given a collection of *learning traces*: $LT_1, \ldots, LT_n$
- Given an *environment* with a set of *input variables* and *control variables*: $E(x, y)$
- Optionally, given a target *task*: T

[1]Minor exceptions to our use of the terminology from Ontañón et al. (2011) are that they use the term Learning *from* Observation while the more common term Learning *by* Observation is used in this thesis, and similarly the more common term *expert* is used instead of *actor*.

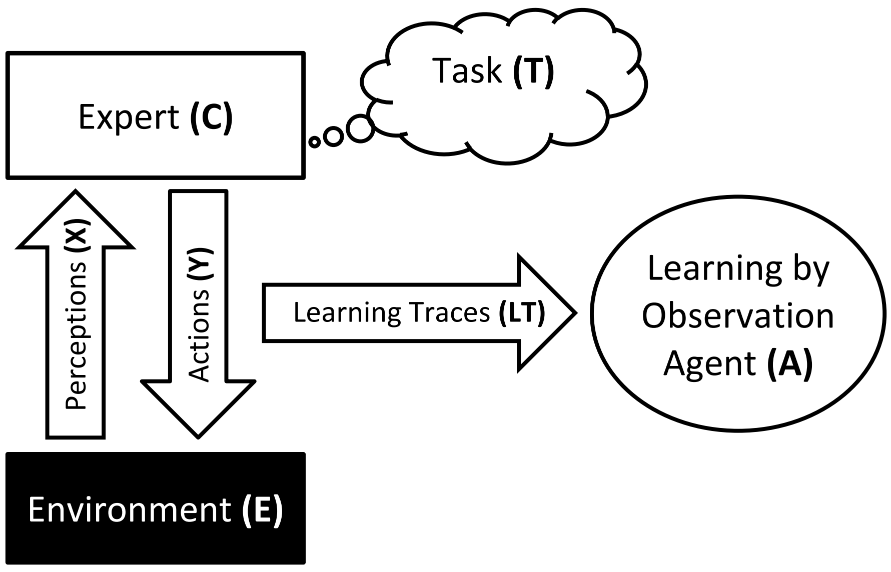


Figure 3.1: Overview of the learning process in Learning by Observation.

- Find a *behaviour* **B** that behaves in the same way as the experts (while achieving task T if specified)

This behaviour found by the LBO system can be any sort of policy or algorithm that determines how to set the control variables given different inputs over time.

We agree with Ontañón et al. (2011) in classifying LBO as distinct from unsupervised, supervised, and reinforcement learning problems. It is not an unsupervised learning problem because the control variables seen in the learning traces provide information on correct behaviour. However, it is also distinct from (or at least a more complex version of) supervised learning because the correct behaviour is not directly apparent: learning traces are a continuous sequence of input and control variables without cause and effect information. This means that the agent must perform additional analysis in order to learn the appropriate behaviour in complex situations, but also means that no extra development or annotation effort is required to train an agent in new behaviours (Floyd and Esfandiari, 2011b).

In contrast with *Reinforcement Learning* (RL), in LBO there is no performance metric in order to learn correct behaviour and it is not necessarily feasible to learn from trial and error (Ontañón et al., 2011). van Lent and Laird (2001) characterises LBO as partway between supervised and unsupervised learning, therefore requiring more research effort and less expert effort than supervised learning, but less research effort and more expert effort than unsupervised learning.

3.1.2 Motivations and Challenges

Learning by observation is challenging, but comes with significant benefits. By requiring only examples of experts carrying out a task, LBO seeks to shift the work required in creating AI from the trainer, designer, or developer to the agent itself (Mehta et al., 2009; Ontañón et al., 2008). This makes it much easier to apply an LBO system to a new domain or task, or to train one with knowledge from domain experts without programming or AI backgrounds, because the expert simply needs to demonstrate the desired behaviour. LBO does require more effort from the expert trainer than required for unsupervised learning, but larger and more complex problems can become tractable in return (van Lent and Laird, 2001).

If the agent's behaviour needs to change, for example with the evolving set of common strategies used in RTS games, LBO also makes it easy to retrain the agent by showing it examples of correct behaviour in the altered environment. This should result in reduced development time required to create or modify an agent, and a wider range of agent behaviour, as the correct behaviour can be observed more easily than it can be programmed. It should also result in reduced debugging effort, as the correct behaviour can be demonstrated and incorporated if the agent is acting incorrectly in a particular situation (Mehta et al., 2009; Ontañón et al., 2008).

LBO may also help to create more human-like agents – often a desirable feature in the game industry or with human-interacting robots – because the agents behaviour can be directly learned from humans. RTS games in particular are such a difficult area in which to create effective AI that RTS games generally rely on fixed, scripted strategies or finite state machines to govern agent behaviour, leading to predictable agents that cannot react effectively to player actions. Additionally it is difficult and time-consuming to try out new AI techniques in new games, so it is not worth the risk during the intense schedules of video game development (see section 2.5.1). An LBO system could be easy enough to apply that it could be trialled in a game without significant time investment, and could produce an agent with varied and reactive behaviour, making it an ideal choice for RTS game AI.

LBO does not require a reward signal like RL, which can be helpful in domains where it is difficult to evaluate the value of a state. It is often especially challenging to effectively evaluate a state part-way through a long task, such as a chess position during a match or the extremely complex state of an in-progress StarCraft match. RL is also difficult to use when it is difficult to explore the state space effectively, such as when the state space is extremely large, as with RTS games, or if it is expensive or slow to explore the state space, as with real-world robots. Because it is difficult to explore the state space, only a small proportion of states could be visited to associate a reward signal and make better decisions later. By contrast, even in huge decision spaces, the examples given in LBO can help to guide the agent to the useful areas of the space.

For these reasons, LBO is an ideally suitable approach for the overall thesis objective of making it easier to create AI systems for complex domains such as RTS games.

A limitation of LBO is that it is very difficult for the agent to ever exceed the performance of the expert examples it was given. However, if a way of measuring performance is available, and the agent is able to explore the use of different actions, LBO may be combined with reinforcement learning in order to improve beyond the performance of the original expert demonstration, while still taking advantage of the knowledge provided by the expert demonstrations, similar to Grollman and Jenkins (2007) and Judah et al. (2010).

LBO can also be difficult to evaluate, because there is not necessarily any performance metric or goal criteria, and even the expert learning traces do not necessarily represent the only correct behaviour. In situations where a *task* is specified, *performance evaluation* can be carried out, measuring how well the learning agent is able to perform the task. Even without a specified task, *output evaluation* can be used, comparing the actions chosen by the learning agent against actions chosen by experts (for the same task, if specified). The learning traces used for evaluation in this manner are usually excluded from use in training the learning agent. Alternatively, using *model evaluation*, the model learned by the learning agent may be directly inspected and compared to a known correct model (Ontañón et al., 2011). All three of these evaluation methods are employed in this thesis.

3.1.3 Related Work

Existing research work on learning from examples of expert players in RTS games mostly rely on extra non-observable information provided to the learning agent, with annotated traces showing expert reasoning, underlying sub-tasks being performed, or goals being achieved (see section 2.4.6). For example, the systems described by Mehta et al. (2009) and Ontañón et al. (2008) require expert behaviour to be annotated with the goals being undertaken by each action in the learning trace. This adds to the development effort required each time a new behaviour is demonstrated, but makes learning the correct behaviour easier for the agent and allows more direct control of the agent behaviour. It also increases the amount of domain knowledge required by the learning agent, making it more difficult to apply to new domains. Weber and Ontañón (2010) shows that it is possible to automate much of the annotation process, but this still requires all of the possible goals an agent could be undertaking to be defined, adding to the work required and resulting in a limited set of available goals. Work is needed to be able to create true LBO systems that do not rely on such extra information, and are therefore easier to apply to new domains.

LBO has also been used for many simpler domains with promising results (see section 2.4.5). For example, Synnaeve and Bessière (2011a) uses a Bayesian Model in order to learn to predict *build orders* from *replays* in StarCraft. The system builds a model which can be queried for the most likely set of current or future buildings, given the current game time and the current set of known buildings (Synnaeve and Bessière, 2011a) (opponent buildings are hidden from the player unless they have a unit nearby). Despite the uncertainty of unseen buildings, and without any labelling of replays or domain knowledge of build orders, the system is effective at predicting the current and future sets of buildings (Synnaeve and Bessière, 2011a). Although it learns just a subset of the whole task of playing the game, this is a good example of LBO applied successfully to StarCraft. Gemine et al. (2012) later used artificial neural networks to make similar *build order* predictions, learning by observing the scripted AI opponents in *StarCraft 2*[2]. LBO has also been applied to flying virtual military aircraft (van Lent and Laird, 2001), Tetris and robot soccer (Floyd et al., 2008; Floyd and Esfandiari, 2011b), and numerous other robotics tasks (Argall et al., 2009). Work is needed to see how well LBO methods can be extended to the full complex RTS domain.

3.2 Case-Based Reasoning

Case-Based Reasoning (CBR) is a method of reasoning by analogy to past experience – knowledge about past problems and related solutions is stored, and later retrieved and reused to make new decisions (Aamodt and Plaza, 1994; Kolodner, 1993; Watson and Marir, 1994). This process is thought to be similar to how much of human problem-solving works, by simply reusing and adapting past solutions instead of logically deducing a solution from first principles each time. Recalling a solution is likely much quicker and easier (for both humans and computers) than fully solving the

[2]StarCraft II Official Game Site: `http://us.battle.net/sc2/en/`

problem again. CBR in its modern form is first defined in detail in Kolodner (1993) (although earlier work applied partial or similar concepts), and the well known CBR cycle (figure 3.2) was defined in Aamodt and Plaza (1994).

3.2.1 Main Concepts

A core concept in CBR is that of knowledge stored as a *case base*, which is a collection of *cases*. Cases are simply $(problem, solution)$ pairs that each record an appropriate response to a situation that the system may deal with. Cases may be elicited from experts, traditionally through interviews or having experts solve problems and encoding their responses as a solution. This is typically a costly process. However, as a CBR system is used, any new problems that arise and are solved (often with expert guidance) can contribute new cases to improve the system.

The CBR cycle (figure 3.2) is a general overview of how a problem is solved in a CBR system:

1. *Retrieve* the most similar past cases to the current problem
2. *Reuse* the solution information stored in the cases to propose a new solution
3. *Revise* the new solution
4. *Retain* any knowledge gained from this new experience

When a new problem needs to be solved, the case base is searched for similar problems, and cases with with similar problems are *retrieved*. This requires some sort of similarity metric for comparing problems. Next, the solutions from the discovered cases are *reused*, sometimes after adaptation to better fit the new problem, and presented as the proposed solution. The proposed solution is then evaluated, either by testing it in practice or by human examination, and *revised* to generate the confirmed solution. Finally a new case containing the new problem and the confirmed solution may be *retained* in the case base for solving future problems.

3.2.2 Motivations and Challenges

CBR is an immediate candidate for use in this thesis because of the way it effectively maps to LBO problems. From the learning traces produced by experts in LBO – *(timestamp, input variables, control variables)* triples – the timestamp and input variables (perceptions) can become the *problem* part of a case and the control variables (actions) can become the *solution*. Each learning trace can therefore be converted into a large number of cases.

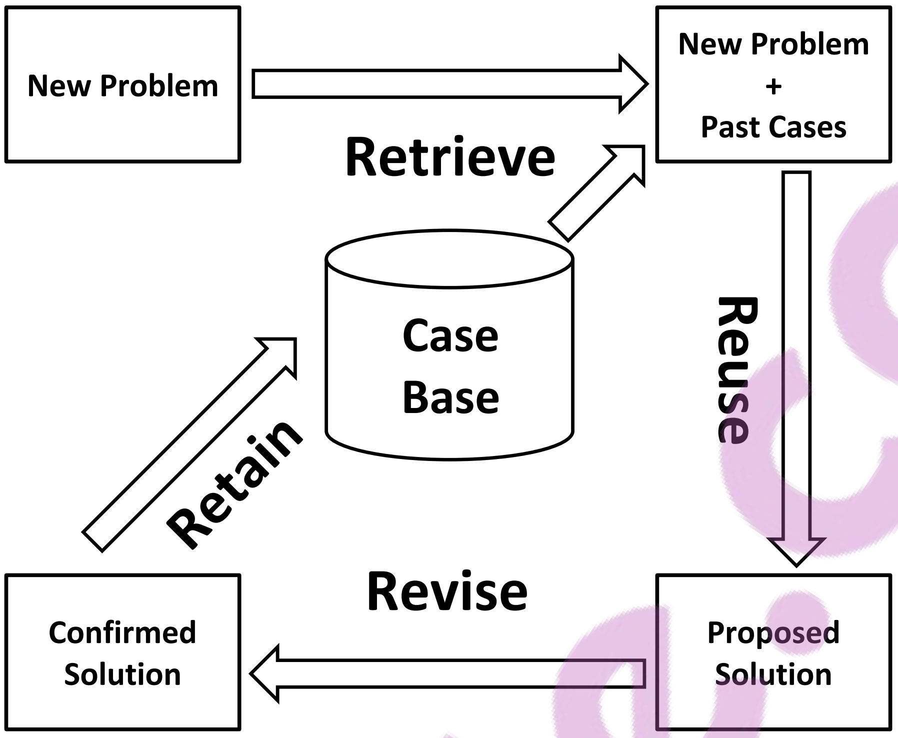


Figure 3.2: The *CBR Cycle*, a general architecture for a case-based reasoning system. Adapted from Aamodt and Plaza (1994)

Variations are possible in terms of how to split up learning traces into cases – whether to split every recorded timestamp into a separate case, or group together sequences of recorded timestamps to make larger cases with sequences of actions as their solution part. Ontañón (2012) characterises this splitting decision as a continuum between *reactive learning*, which has a case for each individual action, and *monolithic sequential learning*, which has a case for each entire learning trace. Reactive learning can fail to account for preconditions and long-term effects, while monolithic sequential learning cannot easily react to changes in situation or failures in execution. This means CBR systems often suffer from excessive reactivity to low (tactical) level state changes or poor responsiveness to high (strategic) level state changes (Palma et al., 2011b).

Alternatively, using additional knowledge, it is possible to split the learning traces based on information about tasks being undertaken (Ontañón, 2012), or using trace-based reasoning it is possible to reason using unstructured traces directly instead of dividing them into separate cases (Mille, 2006). Without additional information in LBO it seems that an ideal route for investigation involves using reactive learning, including past information in cases, to allow the reasoner to locate cases where actions are based on previous events that are no longer observable.

A study that compared various techniques for LBO in a robot soccer task found that CBR was the most effective technique tested (Floyd and Esfandiari, 2009). CBR was compared with decision trees, support vector machines, and naïve Bayes classifiers in learning to imitate the behaviour of RoboCup soccer players by observation, and found that CBR was better at dealing with the hidden internal state of the players, and better at utilising the available heterogeneous observational information (particularly with incomplete information). CBR is also able to handle complex state descriptions (beyond simply an array of boolean or numeric values), and so is easier to use with complex realistic environments.

Another reason CBR is an attractive candidate for LBO is because it requires little in the way of domain-specific knowledge (besides cases). The only place in which additional domain-specific knowledge seems to be essential is in the *retrieve* step, when the system must be able to locate similar cases – what is similar depends on the domain. The *revise* step in the CBR cycle can be skipped and still leave a functioning system, but it would learn only from the learning traces. Alternatively, a *proposed solution* may be able to be tried in the environment when the LBO agent is making decisions to allow revision of the solution and retaining of new knowledge.

3.2.3 Related Work

CBR is one of the most commonly-used AI techniques for learning to play RTS games, having been applied as early as Aha et al. (2005). It also often underlies other techniques such as *Case-Based Planning* (CBP) and *Goal-Driven Autonomy* (GDA). Because of the huge decision spaces in RTS games, CBR is rarely used directly to make game decisions at both strategic and micromanagement levels, and decisions are rarely based on the complete game state information. Instead, the states or actions are abstracted to reduce decision complexity (Aha et al., 2005; Cheng and Thawonmas, 2004; Weber et al., 2011a). Currently this abstraction is done by hand and customised for each domain, so in order to make CBR more easily applicable work could be done on automating state abstraction or allowing CBR to work with less-abstracted state representations.

CBR has been shown to be effective in a range of RTS problem domains, when using additional hand-defined scripts or other domain knowledge, and combined with reinforcement learning. For example, Aha et al. (2005) uses CBR to play *War-*

gus[3] by choosing from a set of scripts to execute based on the current situation. The decision complexity is vastly reduced by basing the state and action information on the current set of constructed building types, and choosing a new action only when this set changes. This means that CBR is used only for infrequent strategic decisions from a reduced set of possibilities, helping to avoid performance issues. Additionally, it requires that a set of scripts has been produced for the CBR system to choose from at each stage of the game.

Baumgarten et al. (2009) applies CBR in conjunction with decision trees and simulated annealing in order to control building placement, opponent prediction, and unit movements in the RTS game *DEFCON*[4]. Complexity is reduced by considering only cases with the same starting positions for the players, and abstracting information by grouping unit movements together into large-scale fleet movements (Baumgarten et al., 2009). Using this subset of cases in the case base, opponent prediction is carried out by finding the most frequent opponent actions – allowing movement orders to be defined relative to likely opponent fleet positions – and movement plans are generated by building a decision tree from the cases, which classifies decisions as leading to higher- or lower-scoring games (Baumgarten et al., 2009). Any parts of the plan which are not decided by the decision tree are chosen randomly, forming a new case which is stored with the outcome at the end of the match. This results in a process of iterative exploration and improvement of the agent (Baumgarten et al., 2009).

Jaidee et al. (2011) combines CBR with RL and GDA to create an agent that can reason about goals, generate plans to achieve them, and resolve discrepancies between the expected and actual outcome of plans. CBR allows this system to learn new domain knowledge about goals and expectations on the fly, reducing the otherwise onerous requirement for domain knowledge to be extracted from experts and encoded for the system (Jaidee et al., 2011). Reinforcement learning allows it to learn the expected reward associated with each goal, given the current goal and discrepancy, allowing it to find an optimum strategy (Jaidee et al., 2011). The system is applied to a relatively simple (compared with RTS games) tactical AI domination game, where it learns to near the performance of a hand-crafted agent (Jaidee et al., 2011).

There are also a number of successful applications of CBR to perform LBO in simplified RTS domains or other simpler tasks, such as robot soccer or Tetris (Floyd and Esfandiari, 2011b). Hsieh and Sun (2008) applied CBR to predict the *build order* of

[3]Wargus: `http://wargus.sourceforge.net/`

[4]Introversion Software | DEFCON: `http://www.introversion.co.uk/defcon/`

a player in a game of *StarCraft* by learning from *replays*. Similarly to prior work (Aha et al., 2005), the complexity was reduced by basing decisions on the currently constructed buildings, and only the immediate next building was predicted. This system also ranked the decisions based on whether they lead to the player winning or losing the game, resulting in a predicted "best" *build order* (Hsieh and Sun, 2008). Floyd and Esfandiari (2011b) describes the *Java Learning by ObservAtion Framework* (jLOAF), which allows learning by observation to be more easily applied to a wide variety of situations because it uses a CBR framework that operates on generic inputs (observations about the current state) and outputs (actions in response to the current state). In order to apply jLOAF to a new domain, all that should be required is an adaptor to convert from a state description to the generic input format, and from the generic output format to an action description (Floyd and Esfandiari, 2011b). This makes jLOAF very interesting for further investigation as a CBR LBO system. Further investigation is needed to determine how well this sort of generic CBR system would work for LBO in a more complex domain.

There are also a few examples of CBR used to play full RTS games using *Learning From Demonstration* (LFD) – adding extra non-observable information to the LBO learning trace. For example, Mehta et al. (2009) and Ontañón et al. (2008) use hand-annotated learning traces to learn to play Wargus (as outlined in section 3.1). Weber et al. (2012) also applies CBR to learn to play RTS games. They use GDA to create agents that form plans to achieve goals, and can dynamically respond to failures in the plan execution by choosing new goals and plans to compensate. Further work could be done on creating similar systems that rely on less additional non-observable information in order to make decisions.

3.3 Behaviour Trees

Behaviour Trees (BTs) are a structure for defining agent behaviour in a logical, hierarchical, and human-editable way. They emerged relatively recently from the video game industry itself as a way to reduce the complexity of AI scripting and increasingly convoluted *Finite State Machines* (FSMs). BTs were introduced at the Game Developers Conference[5] having being used for the popular FPS game *Halo 2*[6] (Isla, 2005), and were later detailed and revised by Champandard (Champandard, 2007, 2008, 2012). BTs have quickly become widespread in the game industry and are now

[5]Game Developers Conference: `http://www.gdconf.com/`

[6]Bungie.net | Halo 2: `http://halo.bungie.net/projects/Halo2/`

supported in major game engines such as Unity[7], Unreal Engine[8], and CryEngine[9].

A BT is a static, tree-like, hierarchical structure representing the behaviour of an agent, in which different sub-behaviours or subtasks are grouped into different subtrees. They are normally created and edited by hand by game developers and designers in order to define or tweak how an in-game agent will act. BTs are used as a replacement for scripting and FSMs because they can more easily define complex behaviours through a natural grouping and reuse of behaviours in sub-trees. BTs allow for reactive behaviour through conditional sections and a mechanism for handling failure, as well as flexible responsive behaviour through a simple hierarchical priority system, and purposeful plan-like behaviour through sequential execution (Champandard, 2008).

3.3.1 Main Concepts

BTs have a hierarchical structure in which top levels generally represent abstract tasks or goals, and subtrees represent subtasks and behaviour for achieving each task. Deeper subtrees represent increasingly-specific behaviours, and leaf nodes represent conditions and primitive actions that interact with the agent's environment (figure 3.3). Although conceptually represented as a tree, it is common for sub-behaviours to be reused at different places in the tree, so the resulting structure is really a directed acyclic graph (Isla, 2005).

The composition of nodes into a tree structure defines behaviour that is essentially a program, executable by a depth-first traversal of the directed graph structure starting at the root. The order in which the nodes are traversed and executed is also dependent on the non-leaf nodes of the tree, as the currently executing node can decide the order in which its children are executed. Nodes may also return a status code, such as success or failure, when they finish executing, allowing their parent node to react by, for example, executing a different child or returning a status code itself. This allows failures to propagate up the tree so that a different course of action can be taken, producing low-level reactive behaviour. Nodes whose children are executing may also update in response to new observations, causing the execution of a whole subtree to be cancelled or suspended so another subtree may be executed and producing higher-level responsive behaviour. Leaf nodes of the tree are usually

[7]Unity | Behavior Designer: `https://www.assetstore.unity3d.com/en/#!/content/15277`
[8]Unreal Engine | Behavior Trees:
`https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/`
[9]CryEngine | Modular Behavior Tree:
`http://docs.cryengine.com/display/SDKDOC4/Modular+Behavior+Tree`

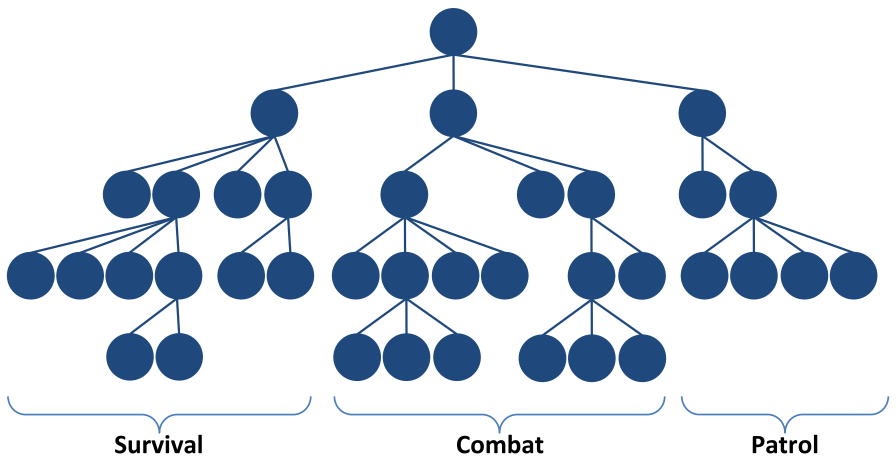


Figure 3.3: Conceptual layout of a BT for a guard agent. The highest-priority subtree at the root would be survival, so that when certain conditions are met the agent abandons combat and patrol actions to survive. Likewise, combat would take priority over patrolling when applicable. Combat subtrees could represent fighting or searching. Adapted from Champandard (2012)

action nodes, which cause the agent to take a basic action in its environment when executed (Champandard, 2008, 2012).

There are four main non-leaf node types that control the flow of execution in a BT: sequence, selector, parallel, and decorator nodes (Champandard, 2008). Each node type has a different effect on the execution of its children, and respond differently to failures reported by their children. *Sequence* nodes run their children in sequence, and usually return with a failure status if any of their children fail. *Selector* nodes run their children in a priority order, switching to the next child if one of their children fails, and usually return with a success status if any of their children succeed. *Parallel* nodes run all their children in parallel, and usually return with a success status if a certain number of their children succeed, or a failure status if a certain number of their children fail. Finally, *decorator* nodes add extra modifiers or logical conditions to other nodes, for example always returning a success status, or executing only when it has not run before. Extensions of the basic node types are possible, such as sequence nodes that loop or selector nodes that cancel the execution of a child if a higher-priority child becomes executable. The specific behaviour and even types of nodes can vary depending on the needs of the designer or developer (Champandard, 2008, 2012).

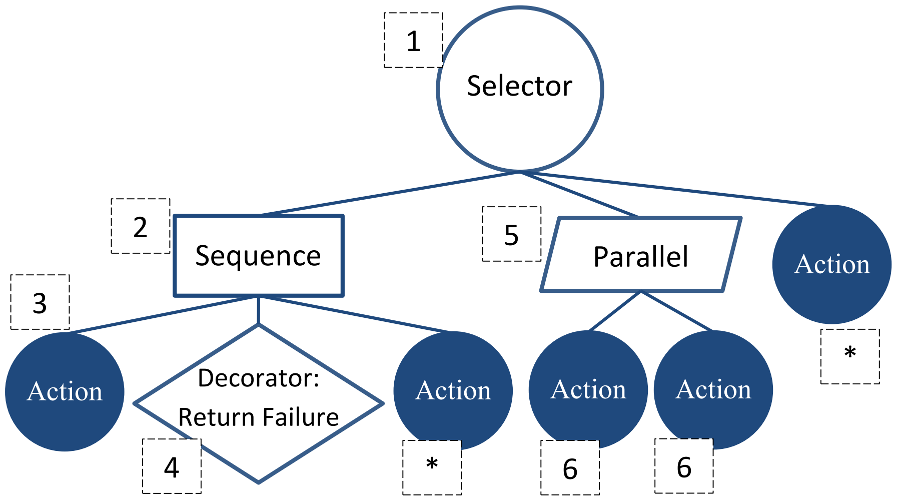


Figure 3.4: An example BT, showing the order in which each node would execute. Asterisks indicate nodes which are not executed. Execution begins at the root *selector* node. Next the *sequence* node begins execution – assuming the leftmost child is selected first – and executes its children until a failure is returned by the *decorator* node. The *sequence* node returns a failure and the *selector* node executes its next child. The *parallel* node executes both children simultaneously and successfully returns, allowing the *selector* to return successfully.

3.3.2 Motivations and Challenges

We would like to be able to generate BTs from examples through LBO, as this would make it much easier to create an agent for many complex domains which is the overall objective in this thesis. BTs may appear to be an unusual choice for combination with an LBO approach, as they certainly do not map as neatly and easily to the LBO inputs as a CBR system does. However, BTs have many properties that make them ideal for real-time and embedded systems, and for the game industry, so it is worthwhile investigating whether they can be made easier to apply using LBO.

A major drawback of CBR for the domain of RTS games is its heavy runtime resource requirements (as found in chapter 4). CBR must constantly refer to a case base to make decisions, requiring computational power to search for and compare cases, and memory to make cases quickly accessible. These resources are usually scarce when playing a real-time game or on an embedded system like a robot. BTs, by contrast, use few resources at runtime, because they need only store the tree structure and only compute an update to the currently-executing node (and possibly its parents).

BTs is also be able to avoid the reactivity versus planning trade off that case-based systems must face. As described in section 3.2.2, CBR systems must choose how to split up learning traces into cases, and this affects how well they can react in the short term or plan for the long term. Because of the way BTs can interrupt execution through failures at either high or low levels, it can be reactive at the low level and also responsive at the high level. Even with its low-level reactivity, when BT execution is not interrupted by external factors it is able to follow a sequence of actions and avoid getting actions out of order.

Because of its hierarchical structure, a BT may also be better able to handle the multiple scales of reasoning and abstraction present in RTS games. This could potentially make it one of the few approaches where a single reasoning mechanism can be used for the whole game, instead of splitting the AI into multiple different reasoning modules for different aspects of the game (Weber et al., 2010b).

There also exist practical benefits of BTs that are often overlooked in academic discussion: they are easily human-understandable and editable. The ability to understand the structure easily means that less-technical people can work with them, and it is easier to find problems and fix them. Because BTs can also easily be changed by hand, designers can adjust and tweak agent behaviour to precisely suit their needs. This is actually very important to game industry veterans (Robbins, 2013; Schwab, 2013) and is lacking in most academic approaches to AI.

Finally, it is worth noting that after this research began, there has been a large amount of work done (relative to the size of the RTS AI field) by research groups around the world investigating the use of CBR for LBO (or closely related techniques). Despite their potential and increasing use in the game industry, BTs have received less attention and, to the author's knowledge, none have tried creating BTs using LBO, making this an unexplored area of research.

3.3.3 Related Work

BTs have become increasingly common in the game industry but they are not yet widespread in academic literature, so most related work concerns learning other behaviour models or planning knowledge. There also appears to be relatively little work investigating learning such static behaviour models by observation, except in the area of automated planning, in which stronger assumptions make the learning task simpler. To the best of the author's knowledge, no prior work has investigated automatically building BTs purely from observing examples of expert behaviour.

Probably the most closely related work involves automatically learning "domain-specific planners" from example plans (Winner and Veloso, 2003). These domain-specific planners are static structures for solving specific planning tasks, and are made up of programming code components such as loops and conditionals, combined with classical planning operators. Programming code may be viewed as an abstract syntax tree, which makes these structures analogous to BTs – both are static executable tree structures representing behaviour. The system is provided with accurate action models in order to build the plans, and implicitly assumes fully observable, deterministic domains, so still requires significant engineering effort and would not be applicable to RTS domains. However, the learned domain-specific planners are able to compactly represent the experiences observed in example plans, find and generalise conditional parts of plans, and produce a structure that solves some automated planning tasks.

Much recent work in automated planning looks at learning planning knowledge by observing plan executions carried out by an expert. This work examines automatically learning action models (Ilghami et al., 2005; Wang, 1995; Yang et al., 2007; Zhuo et al., 2009), task models (Hogg et al., 2008; Mohan and Laird, 2014), or hierarchical structure (Mehta, 2011; Nejati et al., 2006). Some work also expands the applicability of planners by relaxing assumptions from classical planning, addressing learning with nondeterminism (Hogg et al., 2009; Pasula et al., 2004), partial observability (Schmill et al., 2000; Shahaf and Amir, 2006; Yang et al., 2005; Zhuo et al., 2009), or durative actions (Lanchas et al., 2007). However, all of these systems still require some strong assumptions about the domain that do not hold in general LBO problems, in RTS games, or in real-world scenarios. Domain knowledge is also usually required in the form of accurate action models that are not known in LBO problems.

Other work in learning from examples has examined building probabilistic behaviour models using Hidden Markov Models (Dereszynski et al., 2011) and Bayesian Models (Synnaeve and Bessière, 2011a). These approaches require very little domain knowledge and are capable of recognising or predicting plans. However, these systems are not designed to be used for creating plans – their predictions could be extrapolated into a plan but this would likely lead to increasing error, unpredictability, and cyclic behaviour. Perhaps more importantly, the models produced are not easily read and modified by humans, so they become much less useful when behaviour needs to be carefully controlled and modified, as in the game industry (as described in the preceding section).

Instead of focusing on learning from examples, some closely related work has instead used genetic algorithms to evolve BTs in an exploratory process (Lim et al., 2010; Kadlec, 2008). Because BTs can be viewed as programs, the challenge of generating BTs may be related to the challenges of automatic program learning by genetic programming. These approaches hold promise but may become prohibitively computationally expensive or struggle to learn without hand-defined abstractions in complex domains. They also require the addition of a fitness function for evaluating the evolved individuals, which, in an RTS domain, usually means simulating matches played at high speed against an AI opponent, and is not necessarily possible in LBO problems. However, Lim et al. (2010) demonstrates the effectiveness of learned BTs in an RTS game context: starting with random BTs for each in-game goal, it mutates and recombines BTs to learn to play the game approximately as well as a static scripted opponent (that is also used to determine BT fitness).

Exploratory algorithms have also been used in combination with expert examples in order to create behaviour models. Fernlund et al. (2006) uses genetic algorithms to learn and optimise behaviours within a hierarchical "context-based reasoning" structure. It learns by observation except that additional human effort was needed to define the contexts (in which different behaviour would arise) and to annotate the contexts during the learning traces. Observations of human behaviour are used as a fitness measure by simulating agents with the same input variables and comparing their behaviour to the observed human behaviour – those that are able to more closely replicate the human behaviour have a higher fitness. Starting with randomly generated behaviours for each individual, they were evolved to eventually be able to reproduce the expert behaviour and even generalise between different observed behaviour. Using a similar method of genetic programming, it may be possible to evolve BTs to behave like observed learning traces.

Hand-designed BTs have been applied to StarCraft already, and their use indicates that BTs are a feasible approach. Palma et al. (2011b) replaced some low level actions in a CBP system with BTs, allowing customisation and hand-optimisation of low-level actions, while simplifying the strategic decisions to be learned by the CBP system. Weber et al. (2010b) created a "multi scale" AI (able to handle strategic and tactical level decisions) for StarCraft written in *A Behavior Language* (ABL). A core part of the ABL agent is an "active behaviour tree" that is very similar to a BT. Weber et al. (2010b) emphasises the difficulty of separating complex domains into different independent decision-making modules for different tasks and abstractions, and notes the benefits of a unified approach that can handle multiple levels of abstraction and dynamically follow goals and subgoals. This indicates that BTs may likewise be an effective unified single approach to RTS games and other complex multi-scale domains.

There are many areas relating to BTs in which further work can be done, but

LBO approaches in particular seem to be unexplored. While genetic programming approaches seem effective in existing work, they require a method of fitness evaluation (or simulation) that is not necessarily available in LBO, so an alternative technique may need to be used.

Case-Based Reasoning for Learning by Observation in Real-Time Strategy Games 4

This chapter is based on the following publications:
Robertson, G. and Watson, I. (2012). Case-based learning by observation: Preliminary work. In *Proceedings of the Australasian Conference on Interactive Entertainment*

Robertson, G. (2012). Applying learning by observation and case-based reasoning to improve commercial RTS game AI. In *Proceedings of the AIIDE Conference*

This chapter details an experimental evaluation of the applicability of *Learning By Observation* (LBO) using *Case-Based Reasoning* (CBR) for *Real-Time Strategy* (RTS) games. It is motivated by the overall thesis objective of making it easier to create AI systems for complex domains, and specifically focuses on the applicability of AI research in the game industry, as AI in games has seen little improvement over the past decade, and newer "academic" AI techniques are not adopted by the industry. This chapter introduces additional requirements that we believe are likely necessary for AI to be practically applicable and favourably compared with existing industry approaches. It then explains the method used to create an AI framework that can be easily applied to new domains with minimal development effort. This framework is applied to StarCraft in order to test its effectiveness, and evaluated against the requirements set out earlier. The evaluation demonstrates the need for reducing run-time resource requirements with offline processing, and for further investigation into ordering constraints among sequences of actions observed in a trace.

4.1 Introduction

As described in chapters 1 and 2, RTS is a genre of games that presents some of the toughest challenges for AI agents, making it an interesting area for academic research and a difficult area for developing competent AI. The best academic agents are outmatched by experienced humans (Buro and Churchill, 2012; Huang, 2011; Ontañón et al., 2013), while non-cheating industry agents are unable to provide a challenge to players past an intermediate level of skill, as they tend to exhibit predictable, inflexible behaviour (Baumgarten et al., 2009). Furthermore, AI used in commercial RTS games has remained largely the same over the past decade, generally using scripted behaviour with manually predetermined rules and behaviours. For example, the relatively recent major RTS game *StarCraft II*[1], which was released over a decade after the original, still uses a hand-made script-based AI (Sigaty, 2008). An improvement in the AI used could have a significant effect on the genre of RTS games.

There are many likely reasons for this lack of adoption of newer AI techniques in industry. Additional time and risk is involved in attempting to use a new and more complex AI technique, particularly in a large and very time-pressured project like a commercial game (Baekkelund, 2006). The risk is amplified if the AI is difficult to understand, test or debug, which is especially the case for nondeterministic techniques (Florez-Puga et al., 2009; Tozour, 2002). Another reason may be the high run-time resource usage of some techniques, which would reduce resources available to other elements of the game, such as graphics (Baekkelund, 2006). Linked to this problem is the view that most game AI research is not practical enough to be used in commercial games, with most research using games only as a testbed (Champandard, 2011). Finally, there is a difference in underlying goals in academia and industry: academia tends to aim to create strong players which are most capable of winning the game, while AI in commercial games aims to provide a fun challenge to the player (Baumgarten et al., 2009; Davis, 1999; Tozour, 2002). This divide between between academia and industry is discussed in detail in section 2.5.1.

The difficulty of making AI for RTS games leads academia and industry in divergent directions: researchers tend to create complex AI agents that attempt to deal with the challenges but require large amounts of development effort, while industry agents are often allowed to "cheat" – they are given an advantage in resources or information – in order to provide a challenge to advanced players. Cheating AI can make the game less enjoyable if noticed by the players (Davis, 1999), and the simpler com-

[1]Blizzard Entertainment: StarCraft II: `http://blizzard.com/games/sc2/`

mercial game agents are unable to adapt to situations unforeseen by their developers, making them monotonous or easily exploitable by human players (Baumgarten et al., 2009; Tozour, 2002). The work in this chapter makes a first attempt at finding a middle ground between these divided approaches by making it easier to create better RTS game AI suitable for commercial games. It investigates the use of a LBO CBR agent that can be applied to new RTS games with minimal development effort. To be successful, this agent must compare favourably with standard commercial RTS AI techniques: it must be easier to apply, have reasonable resource requirements, and produce a better player. The research contributes to a relatively neglected aspect of the game AI field by examining practically applicable AI techniques instead of focusing on optimal solutions or rational decision-making. Additionally, it will contribute to the larger field of work on RTS game AI, and may be adaptable to other genres of game AI as well.

4.2 Requirements

To carry out the aim of making it easier to create better commercial RTS game AI, an AI framework was produced for this research. For the framework to address the problems set out above and be preferable to existing techniques used in commercial AI, we believe the following requirements must be met:

- The AI should easier to create and approximately as easy to modify as commonly used AI methods for commercial RTS games. This means it should be easy to test, debug and understand, and easy to apply to a range of RTS games, so that it could be quickly tried or used in a new game project. It must also be easily customisable, so it carries out particular behaviour in certain situations, to allow for specific customisation or story elements to be added (this is particularly valuable for industry game AI (Robbins, 2013; Schwab, 2013)).
- The AI produced by the framework must be at least as proficient at playing RTS games as existing AI in commercial games. It should also be more varied in behaviour, and less prone to exploitation by particular tactics which cause it to behave inappropriately. Ideally this would result in an AI which is more human-like and fun to play against, but measuring those factors is beyond the scope of this work.
- The AI must not be computationally resource intensive at run-time – although it is unlikely to be as simple to run as existing methods, it should require a minority of the resources of a modern computer system.

4.3 Method

In order to address the thesis objectives and meet the requirements set out above, *Learning By Observation* (LBO) was employed in conjunction *Case-Based Reasoning* (CBR) in a domain-independent framework called the *Java Learning by ObservAtion Framework* (jLOAF) (Floyd and Esfandiari, 2011b). This should allow the system to be simple to apply to to different LBO problems with little development effort, yet able to exhibit appropriate behaviour in a wide range of situations. Input and control information must simply be adapted to a generic format for the CBR system, and behaviour is learned and selected automatically by the framework.

CBR is effective at working with missing information and uncertainty (Floyd and Esfandiari, 2009), and given sufficient training data, it should be able to behave correctly in a wide range of situations. LBO will allow an agent to be trained by simply demonstrating correct play, thus making such an agent easy to create. This will build upon the work on jLOAF (Floyd and Esfandiari, 2011b), adding to the framework and testing its effectiveness at RTS gameplay.

When the system is learning, it simply observes the state of the game (input variables) and the actions the player makes in response to this state (control variables), and stores these $(state, action)$ pairs as cases in the case base, as shown in figure 4.1. These states and actions are not abstracted in any way, so the system must simply record a large amount of low-level information at this point. In order for the LBO system to remain domain independent, the states and actions are converted into generic inputs and outputs as they are fed in. In this project, it is expedient to use *replays* to learn from instead of actual players because the learning can take place at a much higher rate, and it is easily reproducible.

Before the system is able to play the game, the case base should be processed to extract the useful information from the stored inputs and controls. This processing would reduce the size of the case base and optimise it for quick retrieval of cases. It could even exploit the trace-based nature of the cases in order to avoid storing repeated information and make related cases easier to find. When playing the game, the system observes the state of the game, exactly as before, but it now uses CBR to select actions using knowledge stored in the case base, as shown in figure 4.2. It does this by converting the game state into a generic input, finding the most similar input in the case base using the k-nearest-neighbours algorithm, and converting the associated output back into an action. The resulting action is then sent to the game to be executed.

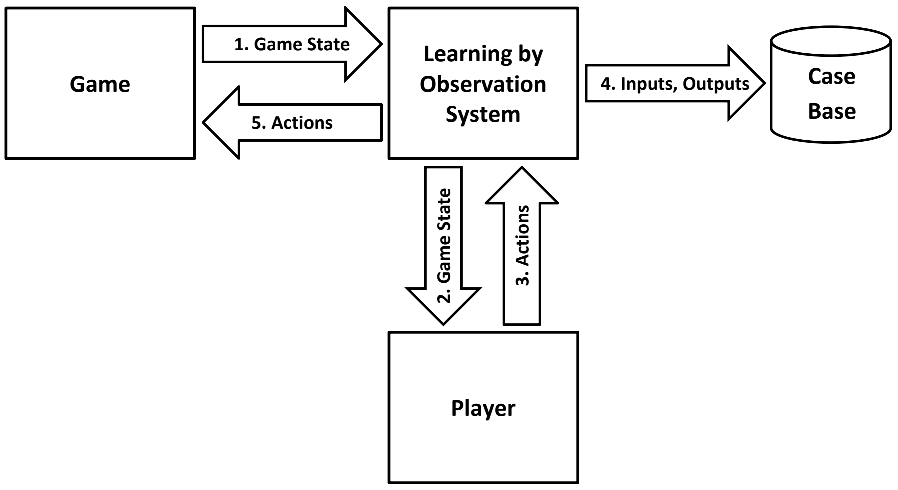


Figure 4.1: Information flow when learning from a player

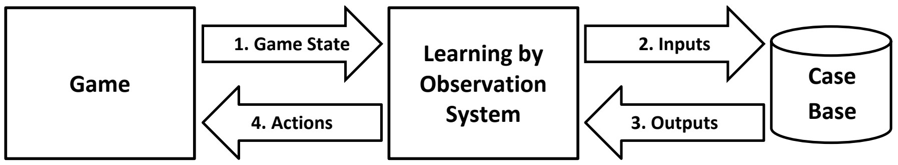


Figure 4.2: Information flow when playing the game

To test this approach, it was applied to *StarCraft*, using observations of expert players to train the agent. Examples of correct behaviour were acquired from human actions in *replays* of expert players and converted into cases for the CBR framework. Each case includes input variables, made up of the map information, buildings and units a player knows about, and an action (control variables), such as moving a group of units or constructing a building. These input and control variables are complex hierarchical structures including variable-sized collections of items, such as player units, that do not easily map to a traditional fixed-size feature array used in many machine learning problems. In any situation during a game, the system finds cases containing similar input variables in order to choose an appropriate action to take, and may also record the situation to add to its case base. This should allow a player more freedom to try new strategies and tactics which were not anticipated by the game's designers, as the AI system is capable of reacting to a very wide range of situations provided it has observed similar situations previously. The gameplay of a human and AI agent can therefore gain depth through more involved interaction, exploration, and trial and error, instead of simply exploiting known weaknesses.

The jLOAF framework was modified to work as the underlying CBR LBO system, and human behaviour examples were parsed from the processed dataset[2] from Synnaeve's "bwrepdump" tool[3] to use as training data (Synnaeve and Bessière, 2012). Initially just the *Protoss vs Protoss* games were used (one of six possible match-ups of the three StarCraft *races*). This produced an extremely large number of cases (approximately 90 million) which had to be reduced to a manageable size in order to be used in real-time. In order to produce a working prototype, a subset of the matches in the input dataset are being used although most of the available features are still included. This highlights the need for generalisation of the case base in order to combine the information of many similar cases into a few representative cases, and optimisation of the case base in order to make the relevant cases quickly accessible during game play.

Although jLOAF is able to use generic inputs and outputs as case information, it is not able to automatically decide how to compare those cases. This meant a similarity metric had to be manually defined (equation 4.1). In the interests of using minimal engineering effort, we used a very simple similarity metric that should be possible to automatically determine based on the data type of the features. For non-numeric features, exact equality was required, producing either 100% or 0% similarity. For numeric features, the absolute difference between the values was found and divided by the overall range observed for that feature in the data (*ObsRange*). That difference value was inverted to find the similarity. To find the overall similarity for multiple features grouped together in the input, we simply find the unweighted mean of the individual similarities.

$$similarity(i,j) := \begin{cases} 1 & \text{if equal} \\ 0 & \text{if non-equal and non-numeric} \\ 1 - \frac{|i-j|}{ObsRange} & \text{otherwise} \end{cases} \tag{4.1}$$

An extra similarity metric was developed for features with variable numbers of unordered items, because items first need to be matched to determine the overall similarity (algorithm 4.1). For example, when comparing a group of units belonging to a player to a group of units in a stored case, each of the player's units must be assigned to a similar unit in the case (if one exists) to be compared. In these situations, the similarity of every pair (using one item from the case and one item from the input) is calculated, and then pairs with the highest similarity are greedily removed until one or both groups is empty. The overall group similarity is calculated as the sum of the

[2] Synnaeve | Gosus dataset: `http://emotion.inrialpes.fr/people/synnaeve/TLGGICCUP_gosu_data.7z`

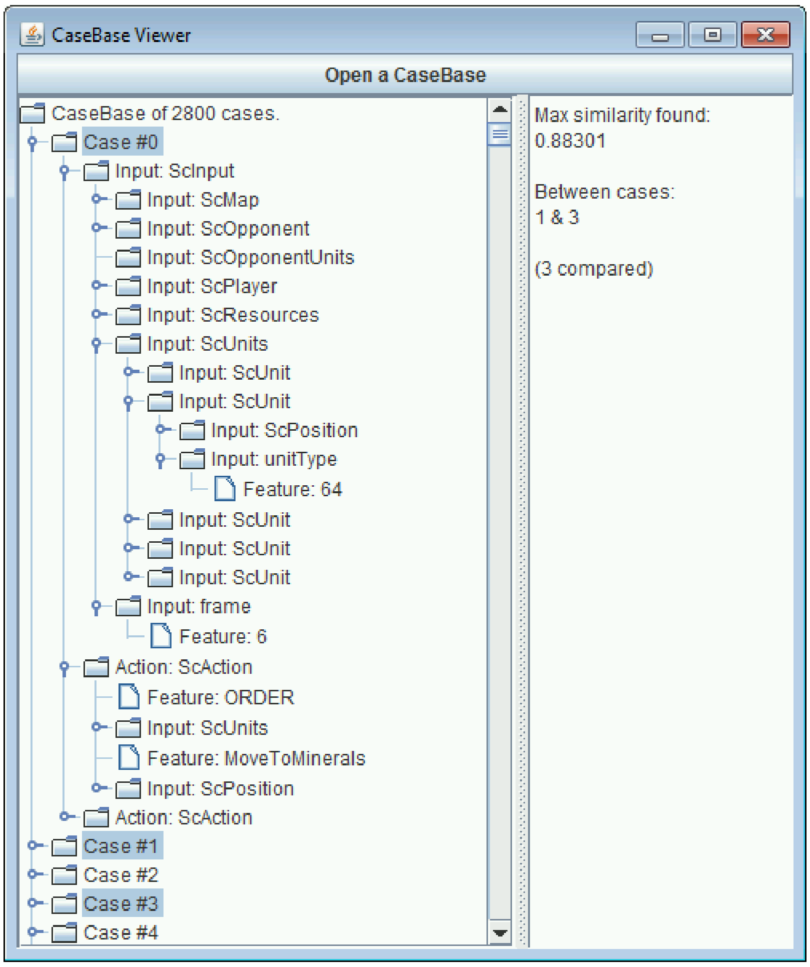
[3] Synnaeve | bwrepdump: `https://github.com/SnippyHolloW/bwrepdump`

pair similarities divided by the total number of items in the largest set. This takes $O(nm\log(nm))$ time for groups of size n and m being compared. Because this is a greedy algorithm, it is not guaranteed to find the optimal pairing of items that would maximise the similarity measure. An optimal solution for this assignment task is the Munkres algorithm (Bourgeois and Lassalle, 1971) but it takes $O(nm^2)$ time (where m is the smaller group), and because comparison speed is more important than optimally matching in this situation, the greedy algorithm was used instead.

```
function group_similarity(group1, group2):
    # Find all pairwise similarities $O(nm)$
    all_pairs = new List()
    for item1 in group1:
        for item2 in group2:
            similarity = item_similarity(item1, item2)  # $O(1)$
            all_pairs.add( [similarity, item1, item2] ) # $O(1)$
    all_pairs.sort() # Sort by similarity $O(nm\log(nm))$
    # Greedily collect highest-similarity pairings
    used_items = new Set()
    total_similarity = 0
    for pair in all_pairs: # $O(nm)$
        similarity, item1, item2 = pair # Split into parts
        # Check if items have already been paired: $O(1)$
        if item1 not in used_items and item2 not in used_items:
            total_similarity = similarity + total_similarity
            used_items.add(item1) # $O(1)$
            used_items.add(item2) # $O(1)$
    # Calculate overall similarity
    largest_group_size = max( group1.size(), group2.size() )
    if largest_group_size == 0: return 1
    else: return total_similarity / largest_group_size
```

Algorithm 4.1: Similarity metric for groups of unordered items

One pair of features that required particular attention were the coordinates that made up the position attribute. Nearly every in-game object has a position given as (x, y) coordinates, but given jLOAF's application-agnosticism these coordinates would be simply compared as two numeric features, ignoring things like map terrain, base locations, and nearby unit positions. These coordinates are essentially a reference to other information that is observable, but without special interpretation, the system can only figure out when positions are nearby to each other in value. Comparing co-ordinate values is meaningless if the cases are learned from games with different map terrain or base locations. We add an abstract feature to position inputs, denoting the region associated with the position. The regions are ranked and numbered using their distance from the player's starting region, so the starting region is always zero and the closest neighbour is one, and so on. In this experiment we are limited somewhat by the dataset so sophisticated spatial reasoning is not possible.



```
CaseBase of 2800 cases.
  Case #0
    Input: ScInput
      Input: ScMap
      Input: ScOpponent
      Input: ScOpponentUnits
      Input: ScPlayer
      Input: ScResources
      Input: ScUnits
        Input: ScUnit
        Input: ScUnit
          Input: ScPosition
          Input: unitType
            Feature: 64
        Input: ScUnit
        Input: ScUnit
        Input: ScUnit
      Input: frame
        Feature: 6
    Action: ScAction
      Feature: ORDER
      Input: ScUnits
      Feature: MoveToMinerals
      Input: ScPosition
    Action: ScAction
  Case #1
  Case #2
  Case #3
  Case #4
```

```
Max similarity found:
0.88301

Between cases:
1 & 3

(3 compared)
```

Figure 4.3: Case base viewer

Additionally, a case base viewer was written to help analyse the case base and choices made by the agent (figure 4.3). The agent playing could be recorded as a separate case base, creating new cases for each of its decisions, so decisions could later be analysed. Using the viewer, the user could compare cases encountered by the agent with cases in the case base to see which cases matched and why. A similarity metric could be applied to determine the similarity of selected cases and find the most-similar cases among a set of selected cases.

4.4 Experimental Setup

This experiment involved training the agent with replay data from 21 two-player *Protoss vs Protoss* matches (more information on this choice of matches in section 4.5). One learning trace was created from each player's actions throughout the entire game, producing 42 learning traces and a total of 2,800 cases. The actions were *not* filtered to leave just strategic or tactical actions – all player actions were used, although a few actions that would have no effect were filtered out, such as moving a unit to the region it is already in (due to the position abstraction described in the previous section). All matches were played on the "Heartbreak Ridge" map, a popular rectangular (128 by 96 build tiles) tournament map with two starting positions, one at either end. However, not all maps were the same version, so some had 4 or even 8 starting positions and slight rearrangements of the terrain.

Each case recorded a hierarchical structure of made up of all input information available, and the action (and parameters) chosen, at the point in time an action was taken by the player. This had to be parsed from the event-based listing in the dataset and converted to imitate sampling observations of a changing system over time. All of the information in the dataset, except the customised "choke-dependent region" and attack analysis information, is made part of the input or action structures, including all units and structures in the match. However, the positioning information was abstracted to use ordered regions instead of (x, y) coordinates, as described above. The case structure used was:

- Input
  - Game Frame
  - Map, with map name, number of start positions, and x & y size
  - Player, with player name and in-game race
  - Player Units[4], with a variable number of unit items
    - Unit, with a position and unit type
    - … (repeated for each unit)
  - Player Resources, with gas, minerals, supply, total gas, total minerals, and total supply

[4]Structures are included in the units because they are treated as simply stationary units in StarCraft.

  - Opponent, with opponent name and in-game race
  - Opponent Units, structured as Player Units above
- Action
  - Action Type: Create Unit, Upgrade Technology, or Issue Order
  - Unit, structured as above (only for Create Unit)
  - Upgrade Name (only for Upgrade Technology)
  - Units Ordered, as Player Units above (only for Issue Order)
  - Target Position (only for Issue Order)
    OR
  - Target Unit, structured as above (only for Issue Order)
- … (may be multiple associated actions due to case merging)

The learning traces were fed in to the agent, then it was tested against the built in AI in StarCraft (both playing as *Protoss*). The agent was manually observed while playing the game, and the win-rate was recorded.

4.5 Results

The final agent was tested against the built-in *StarCraft* AI and evaluated against the requirements set out in section 4.2. It was able to begin to play StarCraft and make actions that looked similar to early gameplay by humans, such as directing starting workers to begin mining resources, constructing initial buildings, and even sending a starting worker out of the base mimicking a human reconnaissance tactic. However, the agent often had difficulties with the ordering of actions, which in many cases prevented any further progress. For example, it could decide that the situation was most similar to one in which the human built a "gateway" structure, but not realise that a "pylon" structure is required first. As the game progressed, the game state would naturally become less and less similar to the early-game state in which humans would build a "pylon" – even if the agent is not making any more effective actions, the timestamps and resources available would continue increasing – so the agent would keep choosing invalid actions and be unable to recover. As such, it was never able to win against the built-in StarCraft AI opponent.

However, the experiment revealed some important challenges to address for applying LBO using CBR. Initially the agent was trained on all *Protoss vs Protoss* matches in the dataset, but it was found to be extremely slow at making decisions, and memory usage became an issue as the case base alone required approximately

16 GB of memory. This initial case base had 1,767,687 cases from 445 two-player matches, and the system had to search through the entire case base every time a decision was needed – no efficient retrieval algorithm had been implemented in jLOAF. To reduce the problem size, only matches played on the most popular map among the set "Heartbreak Ridge" were used, resulting in 67,183 cases from 21 two-player matches, requiring only 75 MB of memory. However, even this case base was too large for rapid decisions to be made, as each decision would still take seconds on a regular desktop computer, clearly too slow for what is supposed to be real-time decision making.

A simple optimisation process was implemented to allow for large numbers of cases to be merged into a more concise set of representative cases, without any human input to decide which cases are similar or unique. When cases were added to the case base, they were compared to existing cases to test for similarity, so that highly similar redundant cases (determined by a 95% or higher similarity in input state) could be immediately merged. Merging cases simply added the new action to the existing case because both actions were used in essentially the same situation. This technique is similar to the classic condensed nearest neighbour rule (Hart, 1968) but we cannot take advantage of exact matching among actions to more rigorously ensure coverage of the state space. This is because LBO actions are arbitrary control variables that do not necessarily fall into discrete categories (in the StarCraft domain, actions have parameters such as the units an order was issued to, which will rarely match exactly).

This optimisation resulted in a 96% reduction in case base size, leaving just 2,800 cases, and a corresponding huge speed improvement because far fewer cases had to be searched for matches at run time. The system still made heavy use of processing power on the machine to make all of the similarity comparisons required to search the case base each time a decision was made, but it could now complete each search in under 100ms. By limiting the agent to one decision per in-game second, it was able to keep up and play the game.

4.6 Discussion

This experimental implementation shows some promise for using CBR for true LBO in RTS games. Although the resulting agent was unable to beat the scripted built-in opponent, it did appear to make sensible actions until it became "stuck" due to poor action ordering. In terms of the requirements set out in section 4.2, it was likely easier to apply than typical commercial game AI, and relatively easy to test, debug, and understand. However, it is not currently easy to customise, and is clearly not as proficient at playing the game as the scripted opponent. It is relatively computationally expensive to run, but it could definitely be made more efficient. This implementation highlighted a number of areas that would need to be addressed and improved for a practical system.

Firstly, and most importantly, a method for automated preprocessing of the observations is necessary to reduce the vast amount of information available in observations and extract useful knowledge. This is especially important for LBO because it is intended to be used with minimal human effort, so manually filtered and extracted data should not be expected. Although jLOAF is designed to *allow* offline analysis of cases to improve retrieval speed and quality (Floyd and Esfandiari, 2011b), such preprocessing features were not actually implemented in the available versions of the software. Similar issues of scale have since been noted in other work that uses CBR for learning StarCraft gameplay, such as Eriksson and Tornes (2012).

The simple method we implemented to merge very similar cases already vastly reduced the number of cases, speeding retrieval and reducing resource usage. However, more could be done to automatically generalise cases by finding commonalities among similar cases to produce fewer, more representative cases. It may be effective to treat very high-similarity actions as equal in order to use existing case base reduction techniques such as Hart (1968). Automatic feature weighting could improve the accuracy of the chosen actions, and automatic feature removal could eliminate redundant features, making case comparison faster and reducing the overall case base size. More compactly stored knowledge also means that more observations can be fed in to the learner to give it a better coverage of the decision space.

The case base structure itself could also be optimised and indexed for more efficient case storage and retrieval. This could involve clustering cases or automating case removal through case base maintenance (Smyth, 1998), combined with efficient retrieval using cover trees (Beygelzimer et al., 2006) or algorithms such as *Fish and Shrink* (Schaaf, 1996). Although it is possible to improve case retrieval efficiency and accuracy, it is worth noting that CBR is fundamentally challenging for making decisions in real time with low computational resource overhead. This conflicts with our requirements, and it may therefore be worthwhile to evaluate alternative approaches.

A different way of improving the agent may be to integrate it with *Reinforcement Learning* (RL), in order to produce better short-term tactical behaviour, or with case-based planning for better long-term strategic behaviour. There are other examples in the literature of combining CBR with RL to improve the quality of solutions chosen

(Molineaux et al., 2008; Sharma et al., 2007), however, this requires a fitness metric which would have to be added or inferred. The agent could also be augmented with decision tree or rule learning algorithms. These approaches could generate classifiers to act as heuristics for making decisions in places where a particular action is always chosen, short-cutting the slower CBR decision process.

Even though this system takes a holistic approach and is capable of making high- and low-level actions (strategic and tactical), it may not be a good idea to use the same reasoning mechanism for both (at least initially). Low-level actions need to be fast and reactive, while high-level actions tend to be less time-dependent and more plan-based. Other work almost always splits up high- and low-level decision-making (as seen in chapter 2). However, splitting these actions is non-trivial with an LBO system because it has no way of knowing which inputs are which without additional human input.

This experiment also exemplified challenges to do with splitting a continuous trace into inputs and actions to make up a case. We used individual actions as case "solutions", but merging cases resulted in many cases having multiple possible actions in their solution. This is actually similar to a technique coined "similarity based chunking" described in later work (Ontañón and Floyd, 2013). Because of the lack of connections between most stored actions, however, the agent had serious issues with action ordering, and was clearly not robust enough to recover from these failures. Andreeva et al. (2014) later found similar issues in using CBR for LBO in FPS games: the CBR agent had no knowledge of past actions and therefore had issues exploring the environment. This is not a trivial issue to solve, but a possible direction is to retain the original ordering information with a reference from each case to the previous case in the learning trace (Floyd and Esfandiari, 2011a). This way, the CBR agent can compare a whole series of cases when determining similarity. Alternatively, we may need more in-depth analysis of cases to discover long-term effects and patterns (see chapter 6).

The case base viewer was a useful tool for understanding and debugging the agent's actions: finding issues within the case base data and analysing why different choices were made. This tool brings the implementation closer to addressing the requirement for being easy to test, debug, and understand. However, with CBR it is still more difficult to manually modify particular decisions, and the system does not currently have an easy way to retrain subsets of the case base by demonstrating a desired behaviour.

Separate from the functional requirements of the LBO system, there were a number of issues and challenges with the dataset from Synnaeve and Bessière (2012). Firstly, to create the conceptual model of an observer seeing the expert's inputs and controls, a complex parser was required to read the text-based dataset (from three separate files per replay) and combine and convert this information into StarCraft data objects. The data objects could then be converted into the generic input and control variables for jLOAF. Creating the parser was time-consuming and would ideally be avoidable for future researchers if a more easily accessible dataset was available.

Additionally, more comprehensive data would be useful. The dataset does not provide enough information to make good low-level decisions. For example, unit orders are not recorded, so it is impossible to tell from a state whether workers are idle or harvesting resources, and unit health is not recorded, so it is impossible to tell if a unit should run or fight. More detailed data may also allow for accurate patterns and rules to be discovered with offline data mining (see chapter 6). Even the spatial information – one of the most detailed parts of the dataset – is limiting. The dataset includes spatial abstractions created by breaking maps into regions and "choke-dependent regions" (regions around *choke points* in the map that are often important for defence or attack). Distances between region centres are given, and positions are given in coordinates as well as region IDs. However, the actual layout of the map and arrangement of the regions is unknown, so it is not possible to tell if regions are strategically important, or which parts of a region are closer to the enemy and therefore likely to be attacked. It also prevents the use of techniques like influence maps that could abstract and combine spatial features and map data.

All of these issues with the dataset raise the need for a much improved StarCraft dataset, that is more comprehensive and easier to work with. This resulted in the work of chapter 5.

An Improved Dataset for Real-Time Strategy Game AI Research 5

This chapter is based on the following publication:
Robertson, G. and Watson, I. (2014a). An improved dataset and extraction process for StarCraft AI. In *Proceedings of the Florida Artificial Intelligence Research Society* (FLAIRS) *Conference*

In order to experiment with machine learning and data mining techniques in the domain of *Real-Time Strategy* (RTS) games, a dataset is required that captures the complex detail of interactions between players and the game, and our previous work (described in chapter 4) highlighted the need for a more detailed and more easily accessible dataset.

This chapter describes a new process by which game data is extracted both directly from *replay* files, and indirectly through simulating the replays within the StarCraft game engine. Data is then stored in a compact, hierarchical, and easily accessible format. This process is applied to a collection of expert replays to produce a new standardised dataset. The dataset is detailed enough for almost the complete game state to be reconstructed, from either player's viewpoint, at any point in time (to the nearest second). This process has revealed issues in some of the source replay files, as well as discrepancies in prior datasets. Where practical, these errors have been removed in order to produce a higher-quality reusable dataset.

5.1 Introduction

As explained in chapter 1, games are an ideal domain for exploring the capabilities of AI within a constrained environment and a fixed set of rules, where problem-solving techniques can be developed and evaluated before being applied to more complex real-world problems (Schaeffer, 2001). Board game AI has historically received a lot of academic and public attention, but over the past decade there has been increasing interest in research based on video game AI.

The RTS genre is a particularly attractive area for AI research because it presents some of the toughest challenges for AI agents, making it a difficult area for developing competent AI, and yet human players can quickly become adept at dealing with the complexity in these games (Buro and Churchill, 2012). RTS games have huge state spaces and delayed rewards, so heuristic-based search techniques, which have proven effective in a range of board games (Schaeffer, 2001), have difficulty with anything but the most restricted subproblems of RTS AI. Many researchers in the field, including our own work in chapter 4, have sought to deal with this challenge by examining the actions taken by human players, using techniques based around keyhole plan recognition (Dereszynski et al., 2011; Hsieh and Sun, 2008; Synnaeve and Bessière, 2011a) or learning from demonstration (Ontañón et al., 2008; Palma et al., 2011b; Weber et al., 2012) (see section 2.4).

Most RTS games can save a *replay* file when a match ends, and expert players often upload their replays to websites for others to watch. Due to the popularity of StarCraft, there are plentiful replays available from expert players. In StarCraft, a replay file records only the starting conditions and player actions in a match, allowing the entire match to be played back as a deterministic simulation within the game engine. This makes for very compact replay files, but means that game state information is not directly available. In order to apply machine learning or data mining to StarCraft data, researchers usually need to run a simulation or use a tool to extract the relevant information. This creates a time-consuming hurdle for each new researcher, therefore a comprehensive and accessible dataset, suitable for a wide range of applications, is needed.

This chapter starts by outlining the existing work related to extracting and using data from StarCraft replay files, demonstrating the need for a better extraction method and dataset than is currently available. Next it gives the main goals for producing the dataset, followed by the design of the extraction process and data recording used to meet those goals. This is followed by a detailed description of the dataset and what is recorded. An evaluation of the resulting dataset is carried out, comparing it to the previous best data available, leading to a conclusion on whether the dataset meets the specified goals and is an improvement on prior work. Finally, areas of future work and improvements are identified.

5.2 Motivation

A number of papers have focused on extracting information from StarCraft replay files, even in the relatively short time since interest began to grow in using StarCraft as a research platform. Before then, the RTS games used for research purposes, such as Wargus[1] and ORTS[2], lacked the expert player base and wide availability of replays to make the approach worthwhile. Information is usually extracted for analysis, such as determining common strategies, and for creating or evaluating *bots*. In many cases, machine learning algorithms are applied to predict a player's strategic choices given the (often incomplete) information known at an earlier point in time. When applied to a bot, this approach can be used to predict opponent actions and to select actions for the bot itself.

To the author's knowledge, the first published work focusing on data extraction from player replays in StarCraft was Hsieh and Sun (2008). They used an existing tool to convert the player actions and their timings – stored in replay files found on a popular StarCraft site – into readable textual log files. Because the replay file does not store game states, basic state information was inferred based on the construction actions taken by the players. A case base and state lattice were created for each of the game's three *races*, allowing the prediction of strategies and analysis of the popularity and effectiveness of build orders (the orders in which buildings are constructed in a game).

Weber and Mateas (2009) followed a similar route, downloading a set of over 5400 replay files from popular StarCraft sites, and using an existing tool to extract player actions into textual log files. However, in this case each resultant log was labeled with a strategy based on expert-defined rules for the *build order*. This labeled data was used to train classifiers to predict the labeled strategy with missing or noisy information, as well as to train regression algorithms to predict the timing of certain actions.

Later, a similar process was undertaken in Churchill and Buro (2011), Dereszyn-

[1]Wargus: `wargus.sourceforge.net`
[2]Open RTS: `skatgame.net/mburo/orts`

ski et al. (2011), and Hostetler et al. (2012). Again, each went through the process of collecting replay files from websites, however, this time the BWAPI was used to connect to StarCraft while playing back the replays as a simulation, allowing for much more complete state information to be extracted. However, each still focused on strategic-level *build order* information, recording numbers of units and buildings in existence every 21 or 30 seconds. In Churchill and Buro (2011) the information was used for comparison with their own build order planner, while in Dereszynski et al. (2011) and Hostetler et al. (2012) it was used to train models for strategy analysis and prediction. Wender et al. (2013) also extracted replay files through BWAPI, however this time focusing on *micromanagement* and visualisation and transformation of replay data.

The most similar work to this work is Synnaeve and Bessière (2012), as it focused on producing a reusable dataset and extraction process, as well as carrying out extraction, analysis and machine learning processes like the other work outlined here. They collected over 8000 replays from popular StarCraft sites, and filtered out many problematic files to result in a set of 7649 replays. The replay files were simulated within StarCraft and information was recorded to three separate text files per replay. Although this work was a useful contribution to the field, some issues remain. Firstly, it is tuned to high-level (strategic) information, so it records only the position attributes of units from over a hundred possible attributes, and stores this only every hundred game frames – approximately every four seconds – providing insufficiently fine-grained data for examining mid-level (tactical) or low-level (micromanagement) activities. Secondly, due to a limitation of BWAPI, it cannot record the actual actions taken by players, but instead must watch for changes in in-game unit orders and try to filter out changes which were not the result of player actions, resulting in discrepancies between the true actions and those seen in output. Thirdly, the output format of three text files, two of which store multiple different types of data in different sections, makes parsing and using the data an arduous process, particularly if searching the data for particular pieces of information.

Our own previous work, described in chapter 4, highlighted the need for a more accessible dataset, as parsing the text files produced by Synnaeve and Bessière (2012) became a non-trivial task. Being able to search the dataset easily and efficiently may have enabled a more responsive case base search, or simply allowed for manual interrogation of the case base for testing and debugging purposes. Additionally, the region and unit information provided was useful, but was very limited for unit micromanagement tasks, in which sub-second reaction times may be required, and additional attributes – in particular, unit hit points (health) – may be vital information.

Recently, Cho et al. (2013) again followed a very similar process of replay downloading and extraction as in Weber and Mateas (2009), but this time used BWAPI to additionally extract the unit visibility events. This provided enough information to determine which opponent units and buildings each player knew about throughout the game, taking into account the fact that the game limits player visibility to an area surrounding their own units. Strategy and victory prediction was then carried out both with and without the limited information.

Extracting information from the immense quantities of expert knowledge encoded in the form of StarCraft replay files is clearly an area of high interest within the field of RTS game AI, yet, until recently, each researcher was forced to reinvent the wheel with a new extractor in order to glean the data they require from the encoded replay files. Synnaeve and Bessière (2012) sought to move the field away from this repetition and unnecessary work, but the dataset is not flexible or fine-grained enough to be able to be used for machine learning at all of the different levels of granularity seen within StarCraft. This work seeks to address these issues.

5.3 Requirements

In order to create an improved standard StarCraft dataset which builds on Synnaeve and Bessière (2012) and yet is appropriate for the full range of research in StarCraft AI, four major requirements were identified: completeness and accuracy of the information stored, and accessibility and extensibility of the dataset and extraction process itself.

For the information to be complete and accurate, the extractor will need to capture as much useful data about the game state as possible, from a wide range of replays, to provide a much more complete rendering of the available information than other datasets. With this level of detail, the user of the dataset should be able to reconstruct the complete game state at any point in the game, from either player's viewpoint. The dataset should become usage agnostic, instead of being aimed at just high- or low-level play, as the fine-grained detail can be used, abstracted, or ignored as required.

To be accessible and extensible, the dataset must obviously be far easier to read than the StarCraft replay files, and ideally should be easier to read than the text format used in prior work. It should enable quick access to information about states without requiring scanning of an entire match's information, so that a user can efficiently find states of interest. It should also be able to be altered or updated easily, and the extraction process re-run relatively quickly, so that a user can modify the extraction process and update the result instead of waiting for a (lengthy) full extraction run. Finally, the output should be as compact as possible so that the extracted data from many replays may be stored and examined or downloaded by new users.

5.4 Method

5.4.1 Overview

Over 7500 professional-level matches were analysed, using the same set of replay files used in Synnaeve and Bessière (2012) for consistency and comparability with prior work. Player actions in the matches were recorded by directly parsing replay files, allowing the true player actions to be extracted, including unit groupings used. This approach simplifies the action extraction (ignoring the complexity in the external code used to parse replay files) and makes it simple to identify observers (non-player participants) in a match early in the extraction process, because they have few actions. In a separate process, game states throughout the matches were recorded by simulating the matches within StarCraft and reading the state using BWAPI (figure 5.1). All unit attributes are recorded, making this a complete representation of the state.

A database-centred design was chosen to allow for structured data to be stored and accessed quickly with a well-known query language. The hierarchical and referential data inherent in RTS games – for example, each unit must belong to a player – can be effectively represented using tables with foreign keys. Databases also provide powerful indexing capability for fast lookup of information even in large datasets, so that game state information about a particular subset of features at a particular point in time can be retrieved easily and efficiently. To reduce the recording size, only changes in game state are recorded. Additionally, it is possible to skip frames in order to trade off accuracy for accessibility (in file size). Appropriate indices allow the most recent value of an attribute to be retrieved efficiently even when the actual time it changed is unknown, and they also facilitate updating of entries, so the extraction process can be re-run quickly. Using the indices and relational information, the extractor can check for unwanted entries and remove them during the extraction process. If the process is to be altered to store more data, it is simple to add additional rows, columns, or tables as desired.

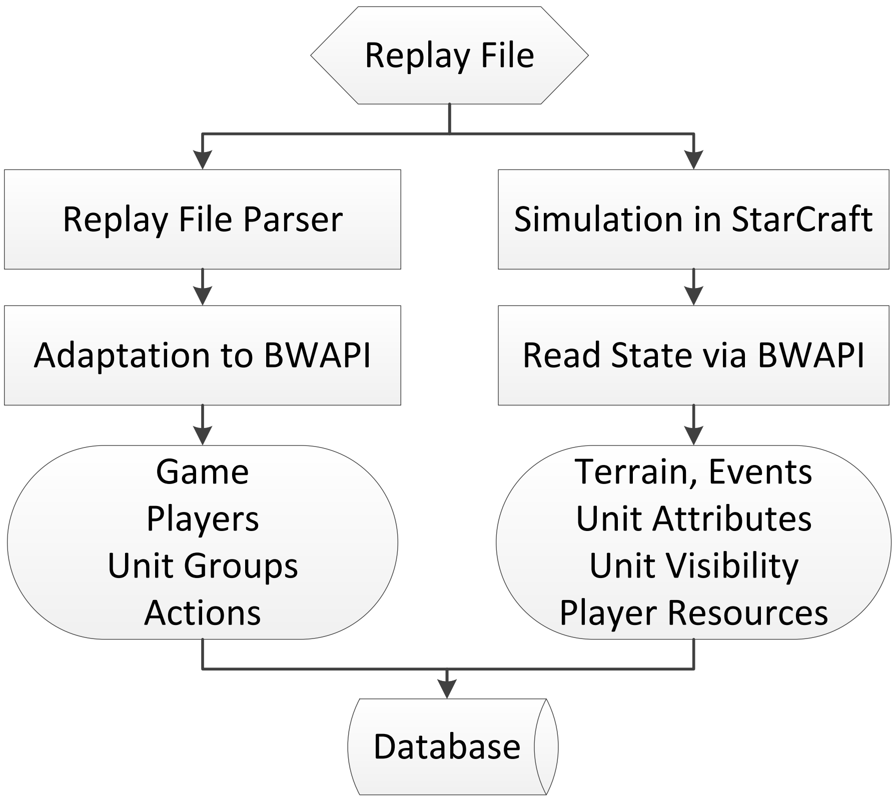


Figure 5.1: Overview of the extraction process.

5.4.2 Extraction Process

The data stored represents interactions over time between players and the game, recording static player and terrain information, as well as dynamic player actions, resources, events, unit attributes and visibility in a database (figure 5.2). A careful method was devised to process the replays consistently and without introducing errors.

First, the replay name and duration (in game frames), along with the names, actions, and in-game *races* of the players are parsed from the replay file. Before the information is stored in the database, the actions are processed as follows.

1. Control groups – used by players to store and retrieve a selection of units using number keys – are replaced with regular unit selection actions. A limitation here is that dead units cannot be filtered out of the unit groups at this point, as unit status is not stored in the replay, so some actions will be incorrectly recorded as if issued to groups in which some or all units are dead (not possible in the game).
2. Consecutive unit selection actions are removed except for the final selection,

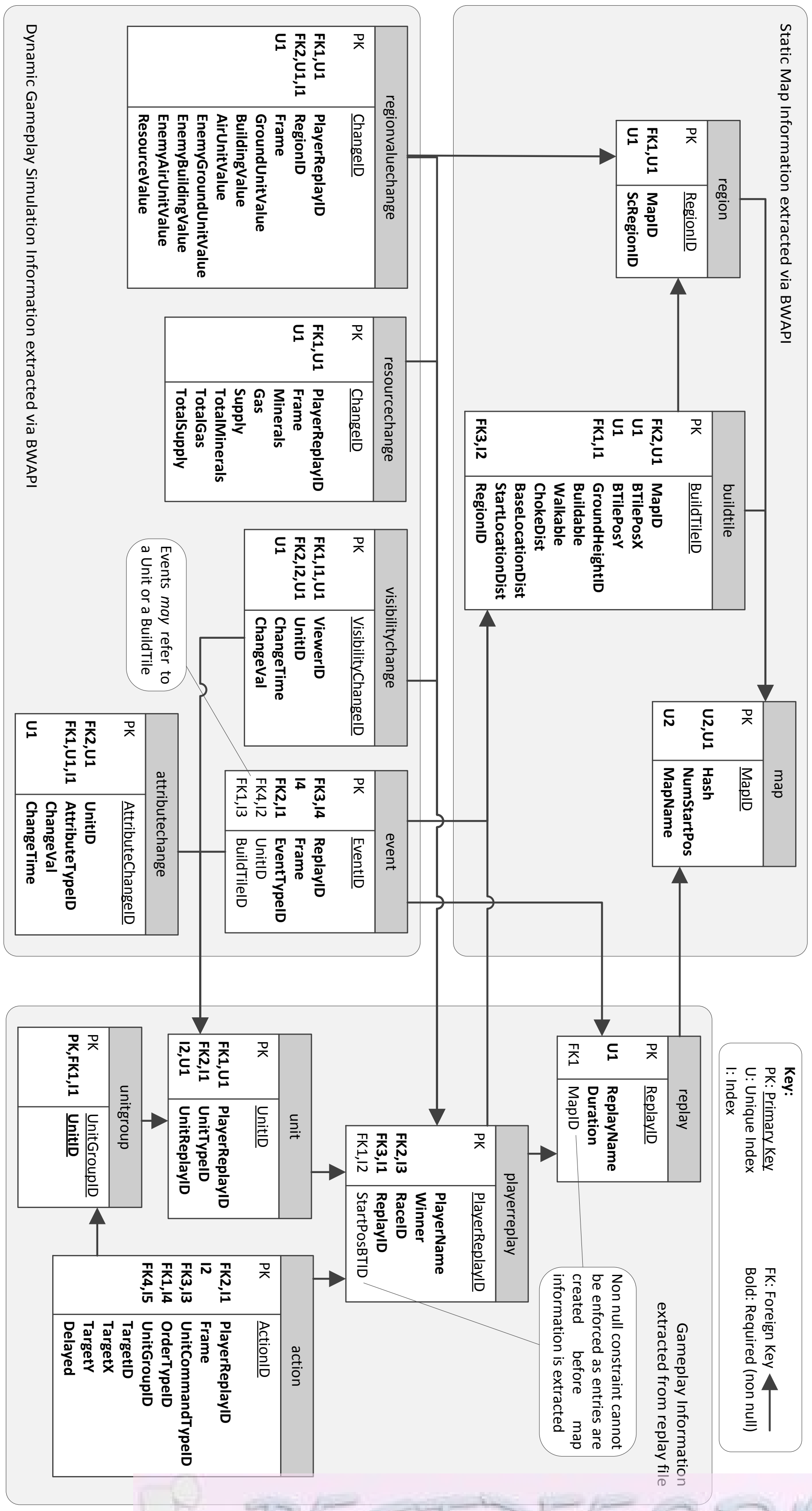


Figure 5.2: Database design, showing relations between tables. Tables containing only type identifier names have been omitted for clarity.

since unit selection actions in StarCraft have no effect on the state except when followed by a non-selection action.

3. Players with the fewest actions are removed until only two remain, as matches often have additional players who are actually observing the match, but have to join as participants due to a limitation in StarCraft. An additional check is made to ensure none of the excluded players performed many actions compared to the included players.
4. A winner is determined if the recording shows one player leaving the game before the other (not always).

At this point, the information can be stored in the database. Selection actions are used only to identify the units that were selected and the groups in which they were selected, so that the non-selection actions performed with these unit groups may be stored.

For the remaining information, the replay is loaded in StarCraft and accessed through BWAPI. First, static map information is recorded, including the name and number of player starting positions, as well as buildability, walkability, ground height, and region identifier of each map tile. This static information could equivalently be read from the replay file, but is more easily accessible through BWAPI. In order to ease spatial reasoning, instead of simply storing a list of *choke points*, base locations and start locations, a walking distance measure to the nearest choke point, base location, and start location is stored with each map tile.

Next, dynamic game information is recorded as the match is simulated. By default, changes are recorded every in-game second (24 frames) to limit the amount of space required while still providing four times the resolution of prior work – enough to capture in full detail everything except precise micromanagement reactions. If changes are recorded every frame approximately eight times more space is required – this tradeoff is discussed further in the next two sections. The extractor records changes to all unit attributes accessible through BWAPI, changes in unit visibility from each player's perspective, and changes in resources and supply (population limit) held by each player, enabling a complete view of the game state to be reconstructed from either player's perspective for any given second in the game. Additional information is recorded for convenience, as it is mostly derivable from the change information stored above. This includes in-game events such as units being created and destroyed, or changing type (redundant), players leaving, and nuclear launches being detected (non-redundant). It also includes a set of aggregate region values stored for each player, summing the value of ground units, air units, buildings, and resources of which they are aware, for themselves and the enemy, in that region.

Extra care is taken to interpret all string values using a Korean character encoding "CP949", as many player and map names contain Korean characters due to the game's immense popularity and professional scene in Korea. Without this step, names with Korean characters are interpreted as strings of unreadable or invalid characters, and the chosen character set was found to produce the fewest invalid characters – just two names contained invalid characters. All strings were stored using UTF-8 encoding to prevent further instances of this problem.

Notably, the unit visibility information recorded is vital to reconstructing a game state as a player would see it in-game, as a player's vision of the map is limited to areas near their own units. Prior work has almost always ignored the visibility of units, as it cannot be extracted from the replay files directly, making it impossible to tell which unit movements (or other attribute changes) each player is aware of. Ignoring visibility limitations makes strategy prediction challenges vastly easier, as most of the hidden information in the game derives from units and buildings which are hidden from a player. Only Cho et al. (2013) and Hostetler et al. (2012) address this issue, as they were specifically examining strategy inference with limited information. Synnaeve and Bessière (2012) records the first time a unit or building is seen, but doesn't record subsequent changes in visibility.

5.4.3 Adaptive Granularity

A challenge when recording information in a game as complex as StarCraft is the tradeoff between information granularity and storage space. Storing all of the game state information every frame – even just the changes – is costly in terms of space, yet fine-grained information can be important to playing the game. This is particularly true in the realm of *micromanagement*, in which professional players quickly and carefully control individual or small groups of units to maximise their effectiveness, usually in combat. In order to better handle this potential use of the dataset, experimentation was carried out to evaluate two potential new ways to adapt the granularity, which we refer to as attack-based adaptation and action-based adaptation.

Attack-based adaptation builds on the basic fixed interval recording, by recording the game state at fixed intervals but reducing the intervals during attacks. It uses the same base frame-rate as the default recording method, but records four times more frequently during combat (as determined by any unit attacking or being attacked).

This rate was chosen because it equates to a very high rate of 240 effective actions per minute, similar to that of the fastest players in the world, and therefore should capture all of the detail seen in player behaviour. A potential drawback of this method is that it cannot distinguish between attacks which require fast player control, such as a large battle, from those that do not, such as a turret automatically firing at nearby enemies. Likewise, it cannot detect other non-attack situations in which fast control is needed.

With action-based adaptation, frame recording happens each time a player makes an action instead of being time-based. This means that fewer frames per second are recorded when players don't need to make many decisions, such as at the start of the game, while more frames per second are recorded when players are rapidly controlling many units and buildings, such as during the intense later stages of the game. Another benefit of this approach is that it stores the exact state the game was in when a player made an action, which could help to detect reactions to changes in state. However, occasionally – particularly early in a match, when few actions are being made – it could actually hinder detection of changes because non-action states are not recorded. This drawback could potentially be mitigated by requiring a minimum recording frame rate in situations where few actions are made.

5.5 Evaluation

In addition to the expected advantages of greatly increased information accuracy and faster querying, the described method of extracting and storing replay data yields some unexpected findings when compared with prior methods. Firstly, it is possible to identify corrupted replays which occur due to a replay being recorded in an older version of StarCraft. In these replays, the rules of the game have changed between recording and playback, causing the simulation to increasingly deviate from the correct state. By comparing the units in the replay file with those seen in the game, 3751 of the 7660 replays were identified as containing invalid units, although 668 of those replays had fewer than 1% invalid units. All replays with more than 1% invalid units were removed from the final dataset.

Comparing the player actions recorded directly from replay files to those recorded in-game in previous work, the higher fidelity of the new recording method becomes evident. By referring to the actual unit groupings used by the player, far fewer orders are recorded, despite the orders showing greater detail and better representing actual player actions. Certain player actions that don't correspond to unit orders, such as setting an exit point for a factory, are now recorded. Additionally, unit order changes that don't correspond to player actions, such as automatically attacking a nearby enemy, are no longer recorded as if they were player actions. This comparison has also helped to identify likely errors in the previous action recording, as certain actions appear to be repeated multiple times in the recording.

The extraction method described in this paper and the adaptive granularity alternatives were evaluated on a test dataset consisting of the games in which both players chose the "Protoss" *race* – one of the six possible *race* match-ups. The unit attribute changes form the vast majority of the data, averaging 96% of the total size of the test dataset, so it is worthwhile to examine these attribute changes further. Looking at the proportion of attribute changes per unit type (figure 5.3), we see that originally, 61% of attribute change records are related to "probe" worker units, which is by far the highest proportion of any unit. Worker units move around automatically and are fairly numerous, so their attribute changes take up a substantial amount of space, yet they are rarely involved in combat or other micromanagement. Therefore, action-based adaptation was applied to individual workers, recording their attribute changes less frequently unless they had recently been given an action. This change reduced them to to 30% of attribute changes, and reduced the overall dataset size by a similar proportion.

Looking at attribute changes per attribute (figure 5.4), we see that 15% of attribute changes record an order timer, and a further 22% (total) record angle and velocity information. Based on domain knowledge, these attributes are unlikely to be important for most analysis and probably could be filtered out completely, while the position attributes are much more likely to be important. However, in the interests of keeping the dataset as complete as possible, these attributes have remained in the dataset.

Finally, we may compare the effects of the adaptive granularity methods (figure 5.5). There is clearly a tradeoff between accuracy and size, but it is difficult to determine whether this tradeoff is worthwhile for a general case. The fixed interval extraction is able to capture sufficient information to understand all but the most fast-paced decisions, and the adaptive granularity methods should cover even those situations. However, the number of frames extracted increases by over an order of magnitude when using either of the adaptive granularity methods, and the storage space required approximately doubles. Given the already large size of the dataset – multiple gigabytes for just the fixed interval extraction of the test dataset – the adaptive granularity methods will not be used for the final dataset. However, because

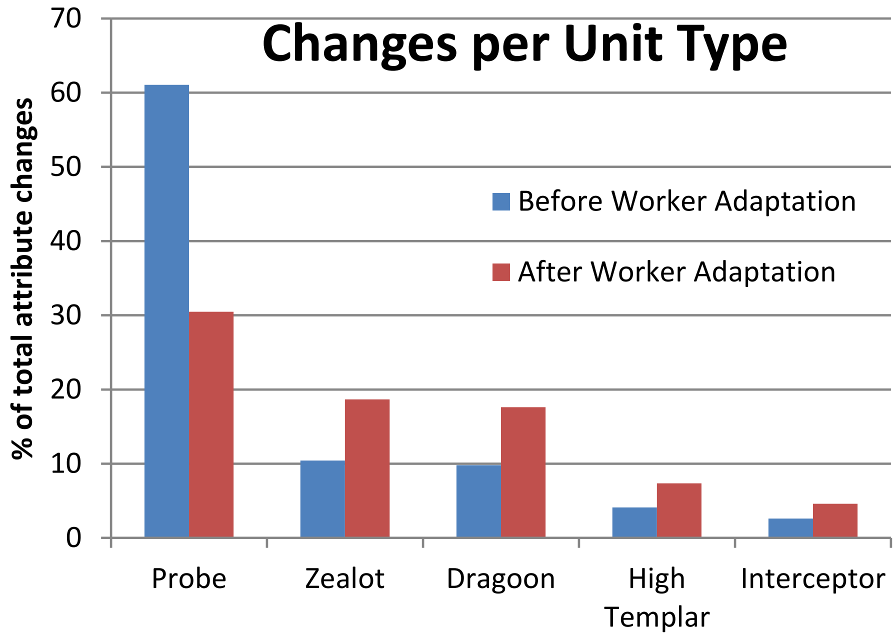


Figure 5.3: Frequency of attribute changes grouped by unit type, showing top 5. Using data from the test dataset recorded at fixed intervals of 24 frames.

the dataset can be relatively quickly modified by re-running the extractor, it can still easily be customised to particular needs.

5.6 Conclusions and Future Work

This chapter has presented a new method for extracting StarCraft replay data for machine learning and data mining. The method combines the strengths of two different information sources: direct parsing of replay file data and simulation of replay data within the StarCraft game engine. By directly parsing replay files, we are able to accurately record the actual actions the players made, instead of watching for the actions' effects, and we can much more easily identify corrupted replay files. By simulating the replays in the game, we can record the complete set of unit attributes, including visibility information, so that the game state at any point can be reconstituted. This produces complete and accurate data, especially compared with prior work, which recorded at most one quarter of the frame rate and just a few of the approximately one hundred unit attributes.

In addition, the chapter describes an effective structure for storing the data such that it is easily accessible and extensible. The source code for the extractor is available[3]

[3]Data extractor code available at: `https://github.com/phoglenix/ScExtractor`

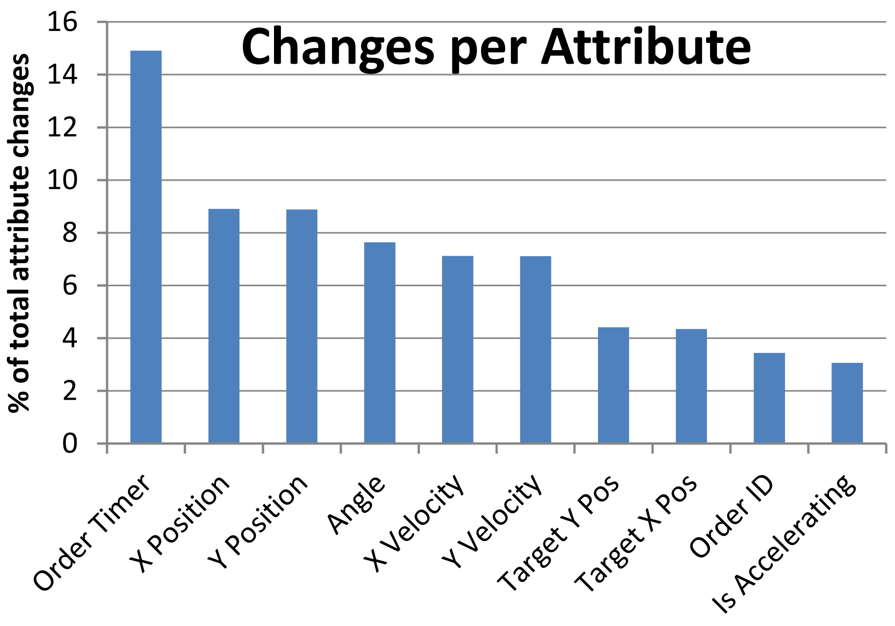


Figure 5.4: Frequency of attribute changes grouped by attribute, showing top 10. Using data from the test dataset recorded at fixed intervals of 24 frames.

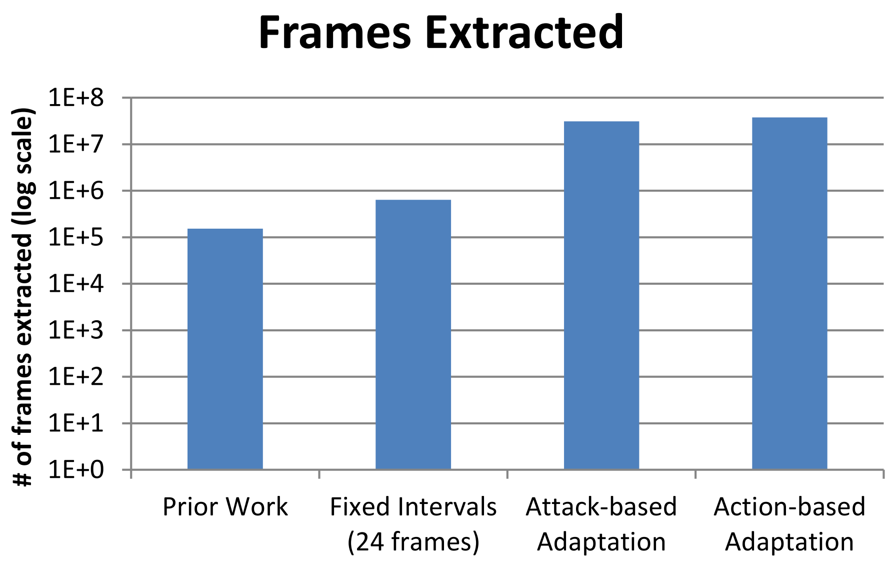


Figure 5.5: Number of frames recorded by each extraction method. Prior work refers to Synnaeve and Bessiere (2012). Using data from the test dataset.

so that further extensions and modifications can be made.

Three methods were tested which varied the choice of frames to extract: extracting frames at fixed intervals, extracting at fixed intervals but with a higher rate during attacks, and extracting frames whenever players made actions. For the full extraction process of a standardised dataset[4], the simplest, fixed interval extraction method was used, because it provides a comprehensive recording, which should be sufficient for anything except precise micromanagement analysis. If more fine-grained analysis is required, the standard dataset is easily modified by reducing the interval or using an adaptive granularity method.

Although not used in the final extraction process, the adaptive granularity extraction methods showed promise for data of widely varying levels of abstraction, and may prove useful in other fields. They could be better optimised by restricting the fine-grained information recording spatially and contextually, instead of just temporally. For example, when using attack-based adaptation, the extra information could be recorded only for units nearby to those involved in the attack, and when using action-based adaptation, the extra information could be recorded just for units that were included in the action. However, these sorts of optimisations require more domain knowledge to implement well, and are thus difficult to generalise.

This dataset allows future work (including our own, described in chapters 6 and 7) to easily work with StarCraft replay data. They are now able to recreate the entire game state at any point in the game (to one-second accuracy), from either player's perspective, for any replay in the dataset. They can also very quickly scan through subsets of the data in which they are interested, such as specific unit attributes or time periods, without having to re-run StarCraft and simulate through the replay, or even use BWAPI at all. It enables researchers to start working on machine learning in this complex domain much more quickly and easily than before.

Acknowledgements

Special thanks to Stefan Wender for the original database design built upon in this work.

[4]Dataset available at: `https://www.cs.auckland.ac.nz/research/gameai/projects.php`

Data Mining Real-Time Strategy Game Data 6

Our experiments in chapter 4 revealed some issues in using *Case-Based Reasoning* (CBR) for *Learning By Observation* (LBO), including high computational resource usage at run-time and difficulties with correctly ordering actions. In this chapter we experiment with data mining techniques, in particular rule mining and pattern mining, in order to address both of these issues.

We apply various data mining algorithms from the *Sequential Pattern Mining Framework* (SPMF) to *Real-Time Strategy* (RTS) replay data, using our StarCraft dataset from chapter 5, but experience a number of issues. Many of the algorithms are unable to work with the complexity of the data available: continuous-valued attributes and time-based entries are handled by few techniques, and none are able to work with hierarchical or referential structure, or with varying numbers of attributes. This required the data be transformed to a flat structure with a fixed number of attributes. Some of the algorithms tested were also unable to cope with the size or arrangement of data produced, failing to complete on the test data after using over 128 GB of memory and three days of computation time. For the algorithms that completed, a huge number of semi-redundant rules were produced.

We also developed a new method to discover rules that have 100% confidence, considering only rules that involve particular attributes of interest. This simplified the problem such that our algorithm could complete in under an hour on a conventional desktop computer with 8 GB of memory. The rules produced by the SPMF *top-K non-redundant association rules* algorithm, or the more concise set produced by our own algorithm, may be useful for our future work.

6.1 Introduction

RTS games are a complex domain in which sequential decisions are made based on past and current state, where game rules and player strategies govern the decisions made. For an LBO agent trying to play an RTS game, with no prior knowledge of the domain, having to decide amongst all actions (that it has observed) is a very difficult challenge, as we found in chapter 4. Fortunately, there is a plethora of data available from which we are able to learn, in the form of expert player replays in an accessible and comprehensive dataset (chapter 5). In order to aid in "online" decision-making while playing – at which point the time and computational resources available may be very limited – we can spend time and resources beforehand to process the data "offline".

One potential way in which data may be processed offline to gain useful insights is through data mining. Data mining is a major field that has recently come to even greater public attention because of the focus on "big data" and the business value of gleaning useful knowledge from the large amounts of information that computer systems collect. Data mining techniques may aid an LBO system by discovering associations, rules, or patterns, that can then be exploited for faster or more accurate online decision-making. This could essentially enable an LBO system to learn about the domain.

Discovered relations could be used to reason with a combination of rule-based approaches and case-based approaches, potentially producing better results than either approach separately (Prentzas and Hatzilygeroudis, 2007). Learned rules could be used as "short cuts" to avoid the need for slower and more computationally expensive reasoning mechanisms in cases where a particular action is always used. Rules could also reveal ordering constraints amongst actions, and be used to enforce these constraints at run-time. So an effective set of learned rules could help alleviate the run-time resource usage issues and the action ordering issues found in chapter 4. Alternatively, a set of rules could be used in a bottom-up approach to build (or guide the construction of) a different reasoning system. This places our work somewhat on the boundary between between classification and rule mining: we are not attempting to classify only a single attribute, but would like fewer, more accurate rules than conventional rule mining. It may be possible to find a middle ground between the two approaches, as discussed in Liu et al. (1998).

We can apply association rule mining algorithms by exporting data from our StarCraft dataset such that actions are linked with the game state at the point they were used. In that format, rules discovered associating state attributes to actions may indicate reactive actions. These could be used when making decisions in order to immediately choose the appropriate action if that state arises, instead of searching for the best action to take through a slower reasoning mechanism. Rules linking actions to states may indicate action preconditions (if they always hold) or common situations in which a certain action is used. Action preconditions could be helpful for planning, or for determining whether actions are likely to succeed. Knowing the situation in which an action is commonly used could help to build a decision-making system that acts similarly to given examples, as we are trying to do in LBO. Rules linking state attributes to other state attributes may indicate domain restrictions on attributes in certain situations, or simply state attributes that tend to appear together. They could potentially be used to predict the hidden state attributes, such as opponent information, in an environment that includes hidden information.

In addition to looking for association rules between attributes at a single time, data may be exported in sequences with time information added so that sequential rule mining algorithms can be used. Sequential rules linking states (or actions) to later states (or actions) could indicate action effects, action ordering constraints, or expected changes in state over time. This could be used to predict future states based on the current state, or to learn the preconditions and effects of actions. If enough domain knowledge is found in this manner, this may even enable a planning algorithm to be used in the domain without requiring any hand-defined knowledge.

Sequential *pattern* mining algorithms can similarly be used on data with time information, in order to find patterns of typical behaviour. These patterns could indicate common high-level strategies or low-level micromanagement sequences used by players. This approach was taken by Leece and Jhala (2014) with StarCraft data at the time this research was underway. They used generalised sequential pattern mining to find patterns under particular restrictions so they could find relevant low-level or high-level patterns separately. This also helped the algorithm to deal with the high complexity and quantity of StarCraft data. For example, by limiting the time between matched items they could find short term micromanagement patterns, or by enforcing a minimum time between matched items they could find long term strategic patterns.

In the remainder of this chapter, we describe association rule mining in more detail, and outline our considerations for the chosen algorithms from the SPMF framework (Fournier-Viger et al., 2014). Next, we describe the alternative rule mining approach developed to better deal with the challenges faced by the other rule mining algorithms, and the method used to evaluate the different rule-mining algorithms on StarCraft data. Finally, we present the results of the experiment and our conclusions.

6.2 Association Rule Mining

Association rule mining is one of the major categories of data mining problems, and has a wide range of research finding and improving solutions (Kotsiantis and Kanellopoulos, 2006). Introduced by Agrawal et al. (1993), it is widely used to discover "rules" in the form of if-then statements involving attribute values seen together in input data. These rules show correlations, patterns, or potentially causation between different items in the database. It is the main category of data mining techniques used in our experiments in this chapter because of the potential uses for learned rules in a decision-making system.

Before we describe the rule mining approaches taken, we must first outline some important terminology in association rule mining (Kotsiantis and Kanellopoulos, 2006; Hipp et al., 2000):

Item Distinct attribute (of which there is usually a fixed total set)

Itemset Set of items

Database Sequence of input itemsets, called transactions

Association Rule Relation between two itemsets $X \Rightarrow Y$ in which:

The antecedent (left) implies the consequent (right)

No item appears in both itemsets, ie. $X \cap Y = \emptyset$

Support (of a rule) Proportion of transactions containing the combined itemsets

ie. The proportion of transactions containing $X \cup Y$

Confidence (of a rule) Proportion of times the rule is correct

ie. The number of transactions containing $X \cup Y$ divided by the number of transactions containing X

The task of association rule mining is to find all association rules in a database having a user-defined minimum support and minimum confidence (Fournier-Viger and Tseng, 2012). The support and confidence restrictions help to narrow down the potentially enormous quantity of rules present in the database to a smaller set of rules that are more common and have more predictive power. For the purposes of learning rules to be used in a decision-making system, we are most interested in high-confidence rules because these will allow us to make predictions and learn how the domain works. We would also like to allow fairly low-support rules, because even rarely occurring situations could involve a rule that may be useful or important to operate in the domain. However, because rule mining algorithms use the support measure to limit the number of rules considered, we may have to increase the support setting if the algorithms run too long searching for low-support rules.

In addition to the fact that rule mining algorithms tend to produce a huge number of rules, many produce a large proportion of redundant rules (Fournier-Viger and Tseng, 2012; Kryszkiewicz, 1998). A rule is considered redundant compared to another if it provides no additional information, for example the rule $\{a\} \Rightarrow \{b\}$ is redundant if we find another rule $\{a\} \Rightarrow \{b, c\}$ with the same support and confidence. Conversely, the first rule is not considered redundant if it has a higher support or confidence. Some algorithms, such as Kryszkiewicz (1998) are capable of eliminating redundant rules, and instead can produce a set of rules with minimal antecedent (conditions) and maximal consequent (effects), at the cost of extra computation. Furthermore, a concise set of rules can be found from which all other association rules can be deduced, called the set of representative association rules (Kryszkiewicz, 1998).

For our system, we would like to avoid generating redundant rules because this would complicate any subsequent process that builds upon the rules, and would compound the difficulty of any potential human interpretation of the rules. Preferably, we would have a set of representative association rules that allow the prediction of as much as possible from the minimum information possible. However, even the representative association rules are not as minimal as we would like: the set of representative rules will still contain pairs of rules that differ by only one added consequent item and a lower support value. Because we are not concerned by the support of a rule, we would prefer to just retain the lower-support rule with the extra item in the consequent, and discard the higher-support rule. So we are interested in algorithms that are able to produce a concise set of non-redundant rules, but may want to post-process the rules to reduce them even further.

6.3 Chosen Approaches

In order to test out a variety of different data mining approaches, we use the *Sequential Pattern Mining Framework* (SPMF)[1]. SPMF is an open-source framework written in Java, with implementations of a large number of different data mining algorithms.

[1]SPMF Data Mining Library: `http://www.philippe-fournier-viger.com/spmf/index.php`

This allows us to simply select already-implemented algorithms based on what is appropriate for the problem we are trying to solve.

As discussed in the previous section, we would like to find a concise set of non-redundant, high-confidence rules, regardless of support, preferably with minimum antecedents and maximum consequents. Ideally these rules would even be human-readable and editable in order to allow for manual inspection and customisation of the automatically-learned rules, as per the requirements in section 4.2. However, it is likely that a rule mining system operating on StarCraft data will produce too many rules for a human to easily interpret.

The first algorithm that may be able to find the concise rules required is the *Minimal Non-redundant Rules* (MNR) algorithm (Kryszkiewicz, 1998). This algorithm seems fairly ideal for our requirements because it generates the minimum representative set of rules, which are inherently non-redundant. It doesn't have any particular features aimed at finding high-confidence or low-support rules, so we will experiment with possible parameter values.

The second algorithm that will be used is the *Top-k Non-redundant Rules* (TNR) algorithm (Fournier-Viger and Tseng, 2012). This algorithm is also aimed at finding non-redundant rules, but does not attempt to find the complete set of minimal representative rules. Instead it is an approximate algorithm that attempts to find the *k* non-redundant rules with the highest support, for a user-specified value of *k*. This algorithm may be less appropriate for our needs because it doesn't attempt to find minimum-antecedent maximum-consequence rules, and we are more interested in high-confidence rules than high-support rules. However, it does produce non-redundant rules and is easier to use to get a limited number of rules by simply setting *k*.

In addition to discovering association rules within a particular game state and action, we are also interested in finding relations between current and future states and actions in the game. For this purpose we include a sequential pattern mining algorithm, the *Fournier-Viger et al. algorithm* (Fournier-Viger et al., 2008). This algorithm was chosen because, in addition to producing a minimum representative set of rules, it is also capable of mining sequences that include integer-valued items. This potentially makes the algorithm much more easily applicable than the other algorithms chosen, in which integer values must be discretised to be represented as items.

6.4 Alternative Rule Mining

After some experimenting with the rule and pattern mining algorithms, we also tried creating our own method of rule mining. Because we are interested in high-confidence non-redundant rules, and are not concerned about support, we took the approach of finding 100% confidence rules only. Although it would have been useful to be able to find rules that did not always hold, doing so would greatly increase the difficulty of the problem. This method also reduces the problem difficulty by allowing a list of "important attributes" to be specified, and avoids generating any rules that do not involve an important attribute. This lets the user (or LBO system) focus the rule miner to produce only rules and patterns involving the attributes in which they are interested. In our case, these important attributes were the chosen player actions, but in a more general LBO system the important attributes could be set to all output (control) variables.

This method is capable of finding association rules relating items in the same transaction, and also finding sequential rules relating items in one transaction to those in later or earlier transactions. A customisable time offset allows the user to adjust the time allowed between the antecedent and consequent of sequential rules found. The algorithm is as follows:

1. Given a list of important attributes: I and a pair of offsets: $\mathit{offset}_{\text{min}}, \mathit{offset}_{\text{max}}$
2. Iterate through all transactions and find attribute values always present: P
3. Record rule: $\emptyset \Rightarrow P$
4. For each value v of each attribute in I:
    - 4.1. Iterate through transactions that contain v and find attribute values always present: R
    - 4.2. Record rule excluding always-present values: $\{v\} \Rightarrow R - P$
    - 4.3. Iterate through all transactions with timestamps between $\mathit{offset}_{\text{min}}$ and $\mathit{offset}_{\text{max}}$ from any transaction containing v, and find attribute values always present: S
    - 4.4. Record sequential rule excluding always-present values: $\{v\} \Rightarrow S - P$

We note that finding attribute values that are always present is possible to do efficiently using set intersection operations. Additionally, finding transactions with timestamps in an offset range from when v was present is possible to do relatively efficiently ($O(n \log n)$ time, where n is the number of transactions) by first finding all timestamps in which v was present and storing them in a binary search tree structure.

Then, to check if a timestamp t is within the offset range $[\mathit{offset}_{\min}, \mathit{offset}_{\max}]$, simply search the tree for its highest entry e such that $e \leq t - \mathit{offset}_{\min}$. If e exists and $e \geq t - \mathit{offset}_{\max}$ then the transaction is within the offset range. The algorithm can also be set to use each timestamp only once, by simply removing each timestamp from the tree as it is used.

Because all rules discovered have 100% confidence, and we are not concerned with support, we have significant flexibility in rearranging rules to make them more compact and readable. This can be done by finding rules with common consequent items and creating a new rule allowing either antecedent and the shared consequent, increasing the predictive power of the rules. This is shown in the following algorithm:

```
1. Merge similar rules. Repeat:
    1.1. Find two rules with maximum proportion of overlap in their consequents:
         $\arg\max_{C_1,C_2}(|C_1 \cap C_2| \div |C_1 \cup C_2|)$
    1.2. If maximum proportion overlap is not over set threshold, stop.
    1.3. Add a new rule using the disjunction of their antecedents and the intersection of their consequents: $A_1 \vee A_2 \Rightarrow C_1 \cap C_2$
    1.4. Remove the merged parts from the original two rules: $C_1 \leftarrow C_1 - C_2$ and $C_2 \leftarrow C_2 - C_1$
    1.5. Delete any rules with an empty consequent
```

For continuous attributes, instead of merely comparing the presence or absence of particular values, it is useful to compare the entire distribution of values in order to determine if there is a relation between important attributes and particular distributions. To compare distributions, we gather all values for an attribute when the important attribute is present, and all values when the important attribute is absent (or within the offsets of interest, if examining past or future attributes). Attribute values are first clustered using a simplified one-dimensional DBScan algorithm[2] with a very large *epsilon* value to discover extreme values used as indicators of missing or unknown data, or special values. The clusters related to the present important attribute are compared to the clusters related to the absent important attribute using Pearson's chi-squared test[2] to test whether there is the same distribution amongst the categories of normal and extreme-valued data. Finally, the Kolmogorov-Smirnov test[2] is applied to test whether the values within each cluster with the present important attribute appear to be drawn from the same distribution as those with the absent important attribute. If

[2]DBScan, Pearson's chi-squared test, and Kolmogorov-Smirnov test algorithm implementations from Apache Commons `https://commons.apache.org/`

either test rejects the hypothesis that the categories or distributions are the same this is recorded as a correlation between the important attribute value and the distribution of values of the continuous attribute.

6.5 Method

The chosen data mining algorithms and our alternative algorithm were applied to learn rules from strategic-level action data from our StarCraft dataset (chapter 5). Replays from the *Protoss vs Protoss race* match-up were used, with a sequence produced from each player's perspective in 392 matches for a total of 784 game traces. From these traces, a transaction was written for each strategic action: build, train, morph, research, and upgrade, along with the most recent game data recorded when the action was chosen (at most one second before). This produced a total of 314,423 transactions. This game data was made up of: the time stamp, action name, action location information, attributes of the unit to which the action was issued, player resources, counts of each unit type owned by the player, and counts of each unit type owned by the opponent (that the player had seen). Because a transaction is simply a collection of items without hierarchical structure, this data had to be flattened, taking the attributes from each game data object and appending them to a single list of item values. This resulted in a list of 531 items per transaction (largely made up of the 101 unit attributes, 207 player unit types, and 207 opponent unit types).

The data were first exported into the Weka (Hall et al., 2009) attribute-relation file format (ARFF), and Weka was used to remove "useless attributes" that had a single value in all transactions. These "useless attributes" are present in the data because it is based on the raw attribute information available from StarCraft and some attributes exported are rarely used, or used only in particular race match-ups. For example, a unit count was exported for every unit type in the game, but many of these units cannot be built because they belong to a different race. Removing the useless attributes reduced the number of items per transaction from 531 to 135, which should vastly reduce the computational resources needed for the data mining algorithms – algorithms that consider all frequent itemsets would have to consider $O(2^n)$ itemsets (where n is the length of the longest frequent itemset) (Zaki, 2004).

Because the MNR and TNR algorithms are unable to deal with continuous attributes, Weka was also used to discretise all integer- and real-valued attributes (excluding identifier references such as unit type IDs). Values were discretised into 10 bins, with an equal frequency of items per bin. Equal frequency bins were used instead of equal range bins because of the extreme indicator values on some attributes, which would have caused all of the non-indicator items for those attributes to be placed in a single bin.

Each data mining algorithm was run on a high performance cluster computer with access to 128 GB of memory, and allowed 36 hours of processor time to complete.

6.6 Results

The MNR algorithm begins with a process of generating candidate itemsets, relying only on the minimum support parameter to eliminate candidates. When the minimum support value was set to a desirable low amount for our problem, such as 50%, the candidate generation very quickly experienced an explosion in numbers. Because all of the candidates needed to be stored, the process very quickly filled all available memory and stopped, never even able to generate candidates past a length of four items. However, by increasing the minimum support to a high threshold, such as 90%, the candidate production could complete and the algorithm produced a set of high support and high confidence rules.

The *Fournier-Viger et al. algorithm* for sequential pattern mining had similar issues. It required a very large amount of both processing time and memory available, quickly exhausting the memory of a regular desktop computer when tested. Running on the high performance computer, it did not fill up the 128 GB of memory available but did not complete the algorithm run within the 36 hour time limit. On a previous test run, it had exhausted 64 GB of memory in just over 7 hours, so it seems likely that it was limited by an excessive amount of processing after the initial stage of candidate generation.

Of the SPMF data mining algorithms tested, only TNR was able to produce an output reliably, even if memory available was limited. This is likely due to its approximate approach allowing it to be more efficient at the cost of accuracy – it does not have to exhaustively search and check to ensure it finds the exact most highly supported rules, unlike MNR. Another main reason for the effectiveness of this algorithm lies in the way it automatically selects the top-k rules. It is able to automatically adjust a minimum support parameter so that the desired number of rules are produced. So when it was run with a *k* value of 1000 it automatically adjusted for a very high minimum support threshold: it produced only rules that had over 99.9% support (see appendix A). Unfortunately, the rules produced by this method appear to be largely spurious, despite having mostly high confidence values.

Our own alternative rule mining method was similarly effective at operating even when memory was limited. It was able to run in under one hour on a conventional desktop machine with 8 GB of memory and produce a set of rules (see appendix B). Some of the rules produced by this method show actual recognisable in-game rules and relations, such as build actions always selecting locations that are buildable and fully walkable, or that the action to build a "Protoss Reaver" is always used with a unit of type 155, which is the "Robotics Facility" the Reaver is constructed from. Conversely, the numeric values analysed by the system seem to have produced a large number of spurious relations, so this approach may be ineffective or the statistical methods used may need to be adjusted to more easily reject the hypothesis that the samples are from the same distribution. The method was also potentially hampered by the many "indicator values" used for missing or unknown information, or to indicate a special status by StarCraft. It may be necessary to filter out such outliers in a preprocessing stage. Our method for combining similar rules appears to have worked, allowing our system to produce just 300 rules in total, whereas the other algorithms would easily produce thousands of rules even with a high support value. However, the rules produced by our system are extremely long, so it is debatable whether they are any more human-interpretable or useful than the many short rules produced by the other algorithms.

6.7 Conclusion

Our results in these experiments indicate that common data mining techniques may simply not be very effective for discovering high-confidence rules, as needed for reasoning about a domain. This seems especially true if we would like to include rules with low support. Rule mining (and pattern mining) algorithms seem to rely strongly on the support threshold in order to limit the quantity of rules produced, and the corresponding time taken, so trying to find low-support high-confidence rules results in exponentially increasing memory and computation requirements.

The algorithms also had some difficulty in dealing with the more complex data available. Continuous-valued items had to be discretised, and even items with categorical values are treated as multiple separate boolean items that are either present or absent. This likely adds complexity to the problem because information is lost about the possible combinations of items – it is no longer inherently obvious that a single categorical attribute cannot take two different values after it is split into two boolean attributes – and the number of possible items multiplies. This data format was also incapable of representing hierarchical structure in the data, forcing hierarchies to be flattened, and has no straightforward way of including variable numbers of the same type of items in a group. The latter limitation makes it very challenging to represent some parts of the StarCraft domain, such as a player's units, so the data mining algorithms can find rules about them.

Despite using non-redundant rule mining algorithms, we observed the well-known problem of rule mining algorithms producing too many rules to be human-interpretable. Even these non-redundant rules were often unhelpful, representing only slight variations or rearrangements of other rules with a different support. Even our own approach produced a large number of unhelpful rules, although it did manage to combine the rules to fewer, rules with more predictive power – a greater number of items in the consequent without increasing the number of items in the antecedent. This may not be a fundamental problem for use in LBO if the rules learned are interpreted only by other algorithms, but if the rules are to be used directly in a final reasoning system then their quantity makes human understanding and modification difficult.

Our alternative rule mining algorithm seems to have produced slightly better results (see appendix B) in less time than the other data mining algorithms tested, but it is addressing a much simpler problem. It is able to be much better optimised by focusing only on "important attributes" and rules that hold true in every situation. Its rule merging process seems to be fairly effective at creating a much more concise and non-redundant set of rules, but again it is only possible because the rules are 100% confident and support is considered unimportant. The results show issues with the distribution comparison process, as it seems to be discovering a large number of spurious relationships involving item distributions.

Although our attempts at rule mining and pattern mining were met with limited success, we were able to produce rules with some of the algorithms that may be able to aid in future decision-making systems. These rules could potentially be added to our CBR system from chapter 4, or used to inform the BT building process in chapter 7.

Learning Behaviour Trees by Observation from Real-Time Strategy Game Data 7

This chapter is based on the following publication:
Robertson, G. and Watson, I. (2015). Building behavior trees from observations in real-time strategy games. In *Proceedings of the International Symposium on INnovations in Intelligent SysTems and Applications (INISTA)*, pages 361–367. IEEE

One of the major limitations found with using *Learning By Observation* (LBO) for *Real-Time Strategy* (RTS) games in chapter 4 was the high overhead of the decision-making process at run-time. Offline processing of the learning traces can help to alleviate the computation required at run-time, but attempts at offline processing using data mining in chapter 6 were met with limited success. This chapter takes an alternative approach to offline processing, with a novel use of motif-finding techniques from computational biology to find recurring action sequences within many learning traces. These recurring sequences are used to produce a behaviour tree without any additional domain knowledge, besides a simple similarity metric – no action models or reward functions are provided. This technique is applied to produce a behaviour tree for strategic-level actions in StarCraft, using the dataset from chapter 5. The behaviour tree was able to represent a large amount of information from the expert behaviour examples much more compactly, and can accurately predict player actions in the early stages of the game. The method could be improved by discovering reactive actions present in the expert behaviour and encoding these in the behaviour tree.

7.1 Introduction

Since their introduction by the video game industry in 2005 (Isla, 2005), *Behaviour Trees* (BTs) have become increasingly common in the industry for encoding agent behaviour (Champandard, 2008; Florez-Puga et al., 2009; Palma et al., 2011a). They have been used in major published games (Isla, 2005) and they are now supported by most major game engines.

BTs are hierarchical goal-oriented structures that appear somewhat similar to *Hierarchical Task Networks* (HTNs), but instead of being used to dynamically generate plans, BTs are static structures used to represent and execute plans (Champandard, 2007; Florez-Puga et al., 2009). This is actually a vital advantage for game designers because it allows them fine control over agent behaviour by editing the BT, while still allowing complex behaviour and behaviour reuse through the hierarchical structure (Champandard, 2007; Florez-Puga et al., 2009). Although they have fixed structure, BTs can produce reactive behaviour by the interaction of conditional checks and success and failure propagation within the hierarchy. Various types of nodes can be composed to produce parallel or sequential behaviour, or choose amongst different possible behaviours based on the situation (Champandard, 2007). For a complete description of BTs, refer to section 3.3.

One of the main reasons BTs are being used in this chapter is that they can expressively represent planned and reactive behaviour while requiring very little computation at run time. This directly addresses a major issue found in chapter 4, when using CBR as the decision-making process: the CBR system needed to search the case base every time a decision was made, a slow and computationally-intensive process. This reliance on online reasoning makes CBR poorly-suited to real-time decision-making when computational resources are limited, such as in games and robotics. By contrast, a BT has a relatively small number of conditions to check each decision cycle, only needing to determine if reactive actions have been triggered and sometimes selecting among a small set of choices at a decision point. In turn, however, a large amount of offline computation may be necessary in order to build the BT, so this work explores and tests how such a process could operate.

We aim to create an LBO system able to automatically learn to carry out a task from examples of expert behaviour. The learned knowledge will be represented and acted upon in the form of a BT, so the main challenge will be to build a BT from learning traces. The resulting BT is able to be hand-customised, so this approach could be used as an initial step, followed by human refinement, in the process of defining new behaviour for an agent. As with our earlier work, the LBO system will be applied to the complex domain of StarCraft. However, in order to make the domain slightly more manageable, we will deal with only the strategic-level actions: the *build, train, morph, research, and upgrade* actions. We also simplify the problem by showing only successfully executed actions in the learning trace, not all inputs from the human. This is because in the game most professional players "spam" actions – very rapidly repeating action inputs until they are executed in order to make actions execute as soon as possible. Including these repeated inputs that have no effect would make recognising patterns of actual effective input very difficult.

In this chapter we start by outlining some existing work on automatically learning planning knowledge in the form of automated planning and case-based planning systems. We then present our approach to the BT building system: using a motif-finding technique to find and collapse repeated patterns of actions. We describe an experiment run to test the system, present some results, and discuss its limitations. Finally we discuss potential future directions and conclude the chapter.

7.2 Relation to Planning Systems

An ongoing challenge in AI is to create problem-solving agents that are able to carry out some task by selecting a series of appropriate actions to get from a starting state to achieve a goal – the field of automated planning. Ideally these agents would be able to be applied to the many practical problems that require a sequence of actions in order to carry out a task, such as robotic automation, game playing, and autonomous vehicles. However, applying a classical planning agent to a new domain typically requires significant knowledge engineering effort (Ilghami et al., 2005). This effort is not in designing the plans themselves, but in modelling the environment, goals, and actions such that the planning system can reason about them. It would be preferable if domain knowledge could be learned automatically from examples, but current automated planning systems capable of learning domain knowledge are generally designed to operate under strong assumptions that do not hold in complex domains (see section 3.3.3). Even when domain knowledge is provided, planning is difficult in domains with incomplete information and nondeterminism, which are present in many real-world and video game applications.

Conversely, *Case-Based Planning* (CBP) systems capable of acquiring domain knowledge can work with few assumptions about the domain, but can have difficulty reacting quickly to failures or exogenous events (Aha et al., 2005; Ontañón, 2012). However, CBP systems face different challenges in the form of case base maintenance and performance, as with our own CBR system in chapter 4.

In many potential application areas, a planner capable of transitioning from any starting state to any goal state is not actually required, and instead it is sufficient or even desirable to have an agent capable of robustly carrying out a specific task or behaviour. For example, in game playing, there is usually a very similar starting state and goal for each match or activity within a game – in board games this is the starting board layout and object of the game, and in video games this could be the starting and win conditions of a match or the daily activities of a non-player character.

An LBO problem makes few assumptions about the domain, allowing for parallel and durative actions, incomplete information, and nondeterminism. This makes it extremely difficult to automatically learn the required world and action models for a planning system (besides perhaps CBP), so instead we aim to automatically learn to carry out a single complex task within the domain, creating a less-flexible but still widely applicable planning-like system. A BT is ideal for representing the behaviour of carrying out a single task within a domain. Using a BT, we are able to combine some of the benefits of learning systems in automated planning and CBP, relying less on perfect models than in automated planning, while being more flexible to changes than CBP.

7.3 Method

We found in chapter 6 that offline processing with data mining was not very effective at learning clear rules and patterns, so we take a different approach to offline learning in this chapter. Instead of combining individual rules to build up a tree in a bottom-up process, we instead will take a top-down approach, starting with a large structure made up of specific learning traces, and reduce it to a smaller tree of generalised common behaviour. Our general approach is to locate and merge areas of commonality within action sequences, as these likely represent common or repeated sub-behaviours. Although it otherwise fits within the restrictions of an LBO problem, this method does require a similarity metric to be defined between a pairs of states (input variables) and pairs of actions (control variables). However, this similarity metric could potentially be learned automatically, as discussed in chapter 4.

The overall method for creating the behaviour tree is an iterative process as follows (figure 7.1). First, a "maximally-specific" BT is created from the given example case sequences. The BT is then reduced in size by repeatedly finding and combining common patterns of actions. When no new satisfactory patterns are found, the process stops. By merging similar action patterns, we are forced to generalise the BT and can find where common patterns diverge so we can attempt to infer the reasons for different actions being chosen. Reducing the size of the BT will also help to make it more understandable if people wish to read and edit it.

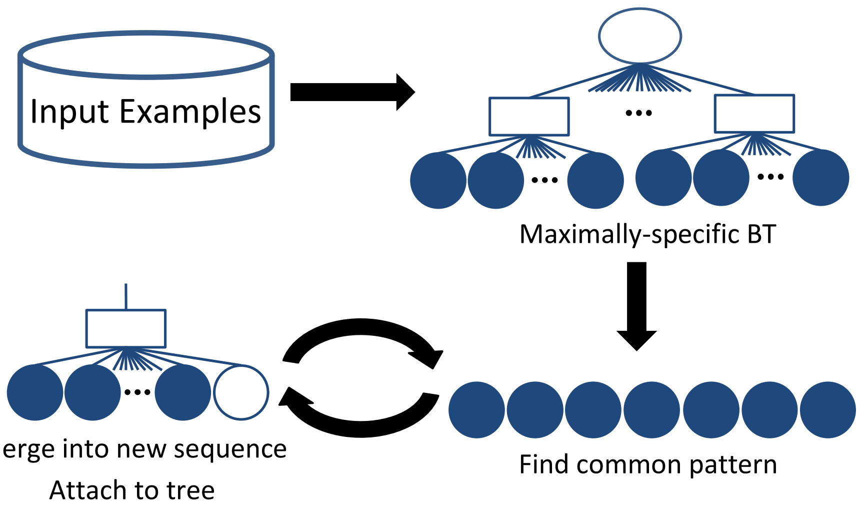


Figure 7.1: Overview of the general BT construction process. Input examples are converted into a *maximally-specific BT*. The BT is then iteratively reduced by finding common patterns, merging them into new sequences, and attaching them to the tree. When no more patterns are found, the process stops.

7.3.1 Creating the original BT

The process of creating a maximally-specific BT (figure 7.2) from a set of examples is actually very straightforward. All actions in a single learning trace can simply be made children of a single BT sequence node (timestamps can be included in the action information for correct action timing). All of these sequence nodes can then be joined by adding a selector node as their parent to make a complete BT. The selector node can be set to choose randomly among its children, or its children compared with the current state using the observations and similarity metric at runtime to select the most-similar option. The tree in its current state is equivalent to the concept of a "monolithic sequential plan" in CBP (Ontañón, 2012) that selects an entire plan and does not deviate. We call this tree maximally-specific because it exactly represents the input example sequences without any generalisation or other processing. This tree is clearly extremely over-fit to the example data, so it needs to be generalised and reduced, which is done by by finding common patterns.

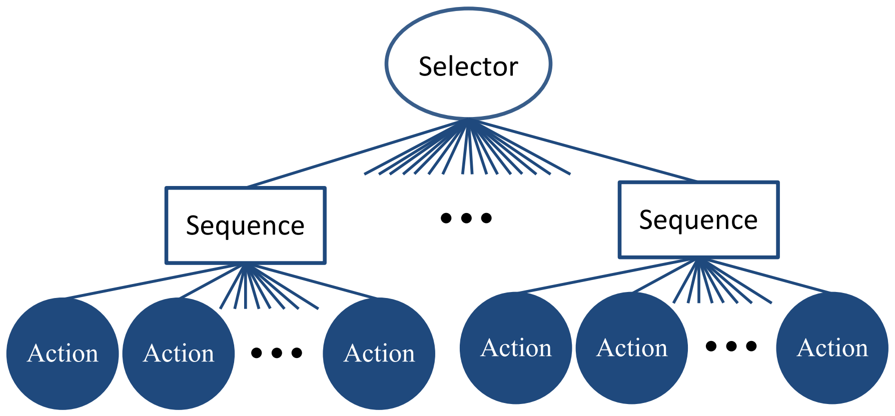


Figure 7.2: A *maximally-specific* BT consisting of a sequence node for each learning trace joined by a single selector node.

7.3.2 Reducing the BT

We iteratively reduce the tree by identifying and merging common subsequences within the sequence nodes, and rearranging the tree to share these common sections (figure 7.3).

The core of the BT reducing method relies on local sequence alignment techniques. These techniques are commonly used to compare two strings, especially DNA strings in computational biology, to find the indices at which one string aligns best with another. The best alignment is defined by a scoring system that rewards matching or similar characters at a position, penalises mismatching characters, and, importantly, allows but penalises extra or missing characters. For strings of length m and n, efficient implementations of this algorithm run in $O(mn)$ time. In order for this algorithm to be used in our situation, we can extract sequences of actions on different branches of the BT and compare them using the provided similarity metric as a scoring system.

While the local alignment algorithm is effective for aligning entire sequences against one another, in this case we are trying to find similar subsequences in cases where the sequence as a whole may not be similar. For this task we can make use of another technique: motif finding. Specifically, we use the Gapped Local Alignment of Motifs (GLAM2) software (Frith et al., 2008). This software uses a simulated annealing based approach to gradually select and refine a short pattern that matches well (scores highly when locally aligned) to many sequences at once. Similar multiple sequence alignment algorithms have been shown to be effective at generalising inexactly-matching items to find patterns data without requiring human intervention

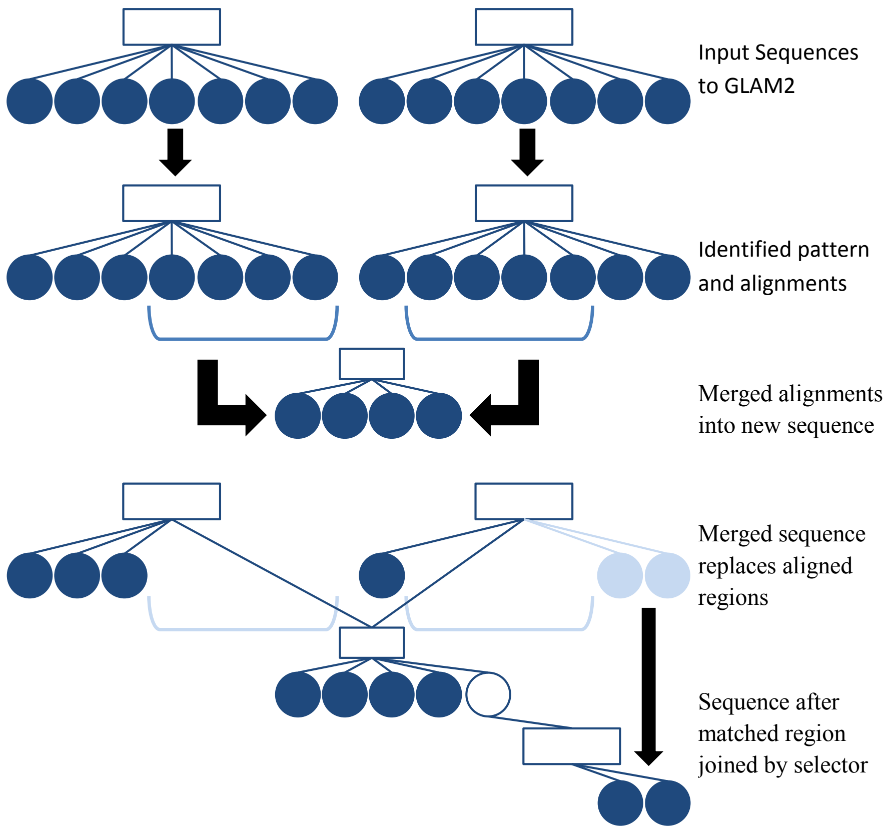


Figure 7.3: Reducing the BT. Sequences are passed in to GLAM2 for pattern discovery and alignment. Aligned regions are then merged to form a new sequence. The merged sequence then replaces the aligned regions of the original patterns. Finally, any sequences following the aligned region are joined by a selector node.

(Chang and Lui, 2001). The BT is converted into a set of sequences for GLAM2 by simply finding all sequence nodes in the tree. When a pattern has not been improved by GLAM2 for a set number of iterations, it is returned along with the alignments and scores for each sequence.

GLAM2 always returns a pattern, so the quality of the pattern and alignments must be checked. We check that all aligned sections have a score above a set threshold, and any alignments with a score below the threshold are discarded. This threshold can be set by informally testing and inspecting the alignments and scores, or can be more rigorously informed by shuffling the input sequences and checking the scores found, or concatenating sequences with shuffled versions of themselves and checking the alignments are more often in the unshuffled regions (Frith et al., 2008). If no aligned sequences have a score above the threshold, the BT reduction process stops.

Using the pattern and alignments found by GLAM2, we begin to construct a new sequence node. This sequence is a generalisation of all of the matching aligned sections of the sequences. For each position in the matched pattern, all nodes at that position in the alignment are merged. This merging produces a weighted combination of the attributes of the nodes. For example, if five nodes had an action with a "name" attribute set to "Train Protoss Probe" and two with "Train Protoss Zealot", the merged node would have a "name" attribute with "Train Protoss Probe"$\times 5$ and "Train Protoss Zealot"$\times 2$. Any attribute values that were seen in just one node of the merge are discarded, because they likely represent unique identifiers or unusual values.

Next, insertion and deletion positions in the sequence are checked for possible transpositions, where actions have occurred in different orders in different sequences. These are detected if an action is almost always inserted either before or after another sequence of actions, but not both (or equivalently, deleted from the pattern and inserted somewhere nearby). In these cases, a parallel node is added with the transposed action (or action sequence) as one child, and the sequence it would move around as another child. In cases where insertions and deletions are not detected as parallel or unordered, conditional decorators are added with records of the state observations. When executing these actions, the decorator will be able to check the stored and current state observations in order to decide whether to execute, based on the similarity metric.

Finally, the newly constructed sequence can be used to replace the aligned regions of the original sequences. For each matched sequence, the region before and after the aligned region are separated. For each section before the aligned region, the new sequence is added as the final node. Next, a selector node is added to the end of the newly constructed sequence. For each section after the aligned region, the section is added as a child of the selector node. This will allow the node to select the sequence with the most-similar state observations at execution time. At this point, each sequence node in the tree is passed back to GLAM2 for analysis. Because the previously-found pattern has been collapsed into one sequence node, it will be far less common, so a different pattern will be found.

It is worth noting that this method will never be able to produce a fully compact tree with conditional sections and reactive actions, because it is looking only for common patterns of actions, and not the reasons they may have been taken. So this method is intended as a first stage in a multi-stage process of converting learning traces into a compact and human-readable BT. Following stages could reduce the tree further by examining the conditions under which certain actions are taken, or locating reactive actions and differentiating them from normal changes in strategy.

This is discussed further in section 7.7.

7.4 Experimental Setup

In order to experiment with building behaviour trees for StarCraft, we used data from our existing dataset (chapter 5) consisting of 392 matches of expert human players in the "Protoss vs Protoss" race match-up. The matches were read from each player's perspective for a total of 784 learning traces. Each trace had an entry for every strategic action the player made during the game, paired with the most recent game state observations when the action was made (to the nearest second because game state is sampled every second in the dataset). This resulted in a total input of 218,053 actions. To compare actions, a very simple action similarity metric was used, producing a score of 1 if the action names were the same, and 0 otherwise, avoiding the full complexity of the action parameters (see below). Action names were based on the action type and interpreted target identifier, for example *Train Protoss Probe*.

Although there is nothing fundamentally preventing this algorithm being implemented with only a similarity metric between actions, for this iteration an additional mapping function was required to map actions to characters. This is because GLAM2 operates only on strings of characters, with exact similarity or difference between characters. So actions needed to be encoded as characters before being run in GLAM2, and the results decoded in order to be used as actions again. An extended character set of 183 characters was used so that the 183 most-common action types (differentiated by action names, as above) could be represented in GLAM2. This was sufficient to fully represent the approximately 60 action types observed in the data.

The learning trace information is similar to the case information used in chapter 4, but was slightly simplified to make the learning process easier. However, the BT building process relies only on the similarity metrics (and currently, the mapping function mentioned above) so the complexity of the trace information should not affect performance. Each item in the learning traces consisted of:

- Input
    - Game Frame
    - Map, with map name, number of start positions, and x & y size
    - Player Unit Counts[1] for each unit type

[1]Structures are included in the units because they are treated as simply stationary units in StarCraft.

- Player Resources, with gas, minerals, supply, total gas, total minerals, and total supply
- Opponent Unit Counts for each unit type, for units that have been seen

• Action

- Game Frame (0–1s after Input frame)
- Action Type: build, train, morph, research, or upgrade
- Target ID[2] representing unit type, research type, or upgrade type depending on Action Type.
- Units Ordered, as a set of Unit objects
- Target Position, as x & y coordinates

Once the learning traces were read in to create a maximally-specific tree, the iterative reduction process began. This repeated a process of outputting sequences from the tree to GLAM2, reading the GLAM2 result, and modifying the tree, until the stopping condition (no alignment score above 50). Parameters for GLAM2 were chosen by empirical testing to select values that resulted in high-scoring alignments. The following parameters were used (defaults were used for non-listed parameters):

Param	Value	Description	Default
r	3	number of alignment runs	10
n	50000	num iterations without improvement before stop	10000
z	2	minimum number of sequences in the alignment	2
a	2	minimum number of aligned columns	2
b	20	maximum number of aligned columns	50
w	15	initial number of aligned columns	20
D	500000	deletion pseudocount	0.1
E	2000000	no-deletion pseudocount	2.0
I	10000	insertion pseudocount	0.02
J	40000000	no-insertion pseudocount	1.0

Note that the deletion and insertion parameters have been significantly changed from the defaults. This is to allow alignments with more deletions and insertions than would be present in DNA strings (for which GLAM2 is tuned), and to allow more leniency in the position of these changes (GLAM2 prefers alignments in which the deletions or insertions are all at particular positions). The number of alignment runs was reduced simply because the variation between runs was low and selecting the best of three runs instead of ten saved time.

[2]StarCraft uses a single ID that is interpreted differently based on the Action Type. Here we directly use the StarCraft action representation so actions have a single Target ID. However, we add a helper function to interpret this ID as the appropriate type.

7.5 Results

The reduction process, using GLAM2, was able to find clear motifs in the dataset and the process was able drastically reduce the total number of nodes required to represent the tree, from 218,832 in the original tree (note this is slightly higher than the total number of actions because sequence and selector nodes are required), down to 71,294 in the final tree (figure 7.4). The opening actions in each match, in particular, were always discovered as a strong motif early on in the process, as these are very similar in most matches.

Figure 7.4: Number of nodes in the BT throughout a run reducing the StarCraft *Protoss vs Protoss* dataset.

Ontañón et al. (2011) suggests three main strategies for evaluating LBO systems: evaluating performance in the environment, evaluating the model compared to a known model, or evaluating the output compared to experts.

The trees produced by this method have not yet been tested in the environment – actually playing the game of StarCraft – because they would be unable to perform the low level unit commands that are required to complement the high-level strategic commands they represent. These responsibilities are separated out into different modules in almost all StarCraft bots due to the difficulty of multi-scale reasoning (Ontañón et al., 2015; Weber et al., 2010b) (see chapter 2). It may be possible to test the performance of these BTs in future using the low-level unit control modules from an existing StarCraft bot such as Ontañón et al. (2013) or Wender and Watson (2014).

The second evaluation method, evaluating the model directly, is not possible either because we do not have a known correct model StarCraft BT to compare against. However, it is possible to visualise and empirically examine the BT model produced

Figure 7.5: Cropped section from the top of a BT, as learned from the StarCraft *Protoss vs Protoss* dataset. Top selector node is the tree root, and the standard opening action sequence is visible directly below the root. Numbers indicate the quantity of original items that have been merged into a particular node or edge.

by the process (figure 7.5). The model produced is actually fairly understandable near the root, with a selector node choosing among different opening moves with the vast majority following a standard opening sequence of creating four *Probe* workers followed by a *Pylon* structure. Even here though, there are many extremely rare sequences that haven't been merged, such as producing two additional *Probe* workers before continuing the regular sequence (seen in the sequence to the left of the root in figure 7.5), or many other opening variations (cut off to the right of the section of tree shown). Further down the tree, the number of interconnections increases and the tree looks increasingly "messy" and unintelligible (to humans). This overall structure could be simplified by removing such rare branches in the tree, but that would result in the loss of a large amount of sequence information gained from the learning traces. Ideally these rare branches would be further examined by another process to find if they should be removed, merged into a sequence, or treated as reactive actions.

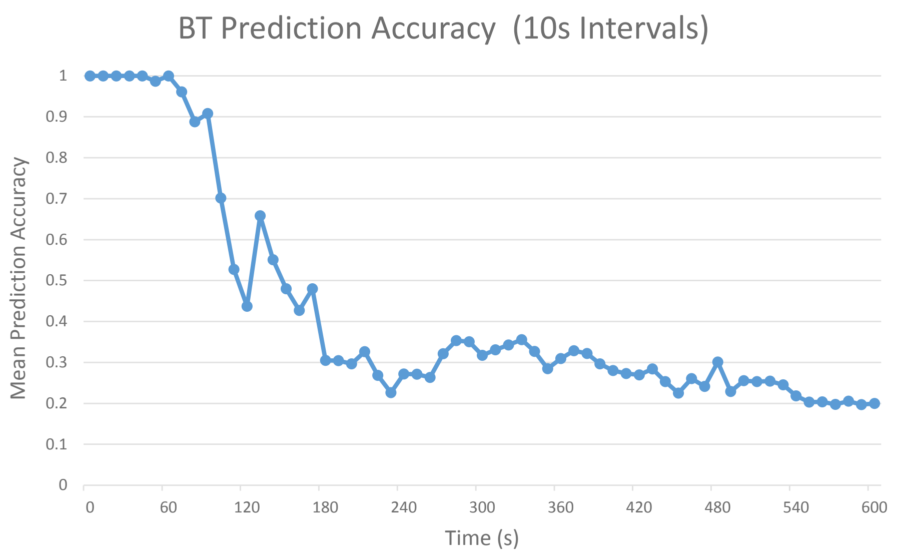


Figure 7.6: Accuracy of action prediction averaged over ten-second intervals from cross-validation using the StarCraft *Protoss vs Protoss* dataset .

The third evaluation strategy, comparing the tree output against expert output, was possible in this experiment, using expert actions from unseen replays to evaluate the predictions made. Ten-fold cross validation was used, so the set of input *Protoss vs Protoss* replays was randomly divided into ten subsets, and a tree was built using nine subsets and evaluated against the unused tenth subset. This was repeated using each subset as the unseen subset in turn. This is a standard evaluation technique for supervised learning and has been applied to LBO before (Ontañón and Floyd, 2013). To test a tree, it was simulated as running with the input from the validation trace until an action was produced, and the tree's action was compared (using the simple action similarity metric described in the previous section) to the action from the validation trace. This was repeated for all (input, action) pairs in the learning trace.

Although the cross-validation method does show a prediction accuracy for the BTs built, it is potentially misleading because the BT is trying to produce a behaviour, not predict a behaviour, so it is not aimed at maximising its prediction accuracy. There is also a potential problem with the alignment of actions: if the player in the validation trace chooses one additional (or less) action than the expected strategy, the subsequent actions will all be misaligned, so likely all evaluate as incorrect, at least until a decision point in the tree. Furthermore, there are many valid strategies in StarCraft, and just because one is chosen by the player does not mean it is the only strategy to use in that situation. The tree could output a completely valid sequence of actions for the game, yet these actions could be very different to the actions a particular player used in that game.

As could be expected, the prediction accuracy of the BT was high in the early stages of the game, and dropped off to become very poor as a match went on. In the first minute of the game, prediction accuracy was almost 100%, but by two minutes into a game the prediction accuracy was around 50% and from three minutes onward accuracy was slowly declining from 30% to 15%. This corresponds to the fact there are few variations in strategy in the early game, but more as the game progresses, and that interactions with an opponent (likely around the 1–2 minute mark) can greatly vary the actions a player takes.

7.6 Discussion

This chapter presents a promising start to automatically producing a reactive AI system from observations, but this approach clearly has some significant limitations. Despite the approach managing to collapse large amounts of repeated or similar sequences of actions, it is not sophisticated enough to separate out most parallel or reactive actions. Parallel actions can be seen in some motifs, in which certain actions are placed before or after other actions in the motif. Currently, GLAM2 cannot detect transpositions, so such variability appears in the alignment as one or more insertion and deletion pairs. This could potentially be mitigated by searching matched regions for inserted and deleted items appearing at a similar frequency across all matched regions, or by analysing the action sequences for ordering relations or a lack thereof.

Reactive actions are somewhat captured by the context-sensitive selector nodes that are inserted after each merged region, but BTs have the potential for much more sophisticated reactive behaviour. Certain motifs found may very well be reactions to conditions in a game, but they are currently treated as if they are a decision point in the sequence, instead of an interruption to the normal sequence. Analysis of the observed conditions leading up to each discovered motif could allow the addition of conditional nodes to trigger the reactive behaviour dynamically, which would make the BT much more robust to changes as well as becoming a better and more compact representation. Ideally, we could continuously process the tree to have fewer and fewer nodes while still representing the original information, similar to Pasula et al. (2004) or Winner and Veloso (2003), who both also use LBO.

The method by which the trees are reduced removes information, so it is possible that important information is lost in the process. This is particularly true of the steps in which discovered motifs are used to merge nodes and subsequently join the sequence back to the original locations of the matched regions. In the merging process, only unique attribute values are discarded, but more attention could be paid to generalising these values. Numerical values, in particular, may often be unique, but could be generalised to a range or distribution. There may also be correlations between attributes, which are lost if multiple values for those attributes are merged. This situation might actually be an indication that the node should be merged into multiple nodes instead of just one. For the joining of merged regions back to subsequent behaviour, this could potentially break or incorrectly connect longer sequences for which the merged region was just an interruption (reactive actions). This issue would be solved with better identification of reactive actions, as discussed above.

A limitation in the way GLAM2 works is that it always finds at most one pattern match per input sequence. This means that sequences may have to be split up before a pattern that repeats within one sequence will become common enough to be found as the most prominent motif. A related issue is that the algorithm becomes less effective at finding motifs as sequences get shorter and more numerous, which is what happens naturally as they are broken up by the tree-building process. This clearly isn't too major of a problem, because GLAM2 is still able to find motifs quite effectively for many iterations, but it does limit its usefulness. Finally GLAM2 complicates the process due to its use of character encodings for comparison, instead of a more flexible similarity metric. This is understandable, as it was designed to work with DNA and nucleotide sequences and is being extended to work in this scenario, but a new implementation of the same algorithm would allow arbitrary actions to be compared instead of only characters.

Looking back at the requirements set out in section 4.2, this approach comes much closer to achieving them than our prior work. The AI is easy to create, requiring only learning traces and a similarity metric, and the inherent action ordering likely means that a BT would be more proficient at playing the game than the CBR-based system. It is also far less computational-resource intensive than the CBR approach, with BTs taking under 10 MB of memory, and actions either being selected amongst a small number of options (at most the number of learning traces read) or simply executed in sequence. However, the trees would have to be further reduced in order to be considered easy for humans to understand, edit, and debug. The system also requires more work to become as proficient at playing the game as standard scripted AI, including adding the ability to produce low-level actions.

7.7 Conclusion and Future Work

In this chapter we focused on behaviour trees as a plan-like structure that can represent and execute knowledge about carrying out a particular task. We outlined the difficulty of learning automated planning knowledge under the constraints of an LBO problem, and suggested learning BTs as an alternative.

We then described our mechanism for building BTs, which involves searching for common patterns among sequences of actions, joining the sequences found, and reconnecting them with selector nodes to allow some reactivity in the plan[3]. The behaviour tree learning mechanism was shown to successfully reduce sequences of player actions from the real-time strategy game StarCraft by 67%, and showed a high prediction accuracy (decreasing over time) when tested against expert actions on unseen learning traces.

A useful extension to this work would be to integrate an unsupervised data mining approach for inferring action preconditions and effects, such as Leece and Jhala (2014) or use the results of our work in chapter 6. Even a partial understanding of the preconditions and effects of actions could help to guide the BT building process, without having to strongly rely on accurate action models like in HTN planning. As an addition to the problem, or possibly an alternative to the similarity metric, a fitness metric could be provided to the agent to allow a more search-based strategy of learning.

[3]Code available online at: `https://github.com/phoglenix/bt-builder`

To further reduce the amount of human effort required to apply the system, it would be helpful if the system did not require a similarity metric to be defined. Removing the requirement for a similarity metric would also make this a true LBO system. This should be possible by making the system define a similarity metric automatically based on the observed contents of the learning traces, similar to what was done in chapter 4.

The method presented in this chapter represents only the first iteration of the overall process of building a concise, human-understandable BT. There are a number of areas in which future work would be able to continue to reduce the tree size or make it better represent the observed behaviour. One such area is in adding conditional nodes to the tree. This could be done by examining the selector nodes in the tree after the motif finding has finished, looking for unique input features among the nodes being selected. This may help to identify why a certain alternative action was taken at this point, especially if multiple merged nodes all had the same input value. Similarly, common conditions shared among multiple parent nodes of a sequence may indicate a reactive behaviour that should be moved up the tree. These conditional nodes can also help to speed decision-making at run time, acting as localised rules to short-cut some of the selection nodes.

Once conditionals are added it may also be possible to add in success and failure codes within the tree. Because LBO does not provide any feedback on whether any actions succeeded or failed, the triggers for success and failure would have to be added as conditional nodes. This could allow further simplification of the tree and improve its generality by replacing selection nodes at each observed failure point with a condition node that stops the execution of its children as soon as a particular change is detected.

Existing work in genetic programming may be able to be applied to this related problem in order to help build or refine the tree structure. For example, genetic programming has a known challenge in dealing with "bloat" – large amounts of ineffective code in the learned model – and ways of eliminating it (Naoki et al., 2009). It may be possible to adapt the approach used to reduce ineffective sections of the BT.

Optimisation techniques could help to further reduce the tree and may be able to simultaneously introduce conditional nodes and identify reactive behaviour. This could be done by optimising for a smaller tree without changing the "meaning" of the tree – the actions that would be output for the given set of inputs. Various tree-altering operations that preserve the meaning could be used to allow gradual mutation of the tree to search for trees with fewer nodes. If these operations included inserting conditional nodes and pushing sequences up the tree then this process would automatically introduce conditions and reactions where appropriate. Additionally, by allowing some amount of change to the meaning of the tree (at some optimisation penalty) the optimisation process could be used to eliminate noise from the tree and reduce it even further.

Once the tree is better able to be reduced and optimised, it would be worthwhile to experiment with adding in low-level actions to the tree. These actions were excluded for the experiments in this chapter because they would likely make it more difficult to find patterns in the actions, and would slow the tree building process. BTs are one of the few decision-making systems that may be suitable for making both high- and low-level actions in RTS games, so it is worthwhile exploring whether such a holistic approach is possible, especially when using LBO. Adding in low-level actions to an already optimised tree may actually help to guide the process, as they could be placed nearby in the tree to the strategic actions they were nearby in the learning trace. This may even help in determining where the tactical actions are related to the strategic actions and where they are not, by comparing the tactical actions from multiple traces that have been placed at one position in the tree.

Ideally, the system should allow low-level actions as part of the learning traces, without any indication to the learner that they are distinct. This makes the problem very challenging, but an LBO system able to handle such diverse scale would certainly help to reduce the effort required to apply AI in new domains.

Concluding Discussion 8

This chapter concludes the thesis and discusses the overall findings and contributions, as well as the limitations encountered in each area. It also suggests possible future directions for research following this work.

The overall aim in thesis is to make it easier to make AI for complex domains, in order to allow better AI to be used in complex games and real-world applications. As part of this aim, we are concerned with the practical applicability of techniques, instead of focusing on their accuracy or optimality, so we attempt to address some of the practical barriers to adoption. One of the largest barriers to the use of new AI techniques is the engineering effort required to create an advanced AI system for a new domain, so a core requirement in all of this work is creating AI systems that require low engineering effort to apply.

We approach the aim using the domain of RTS games because they present many challenges that are present in real-world problems, and that AI research has yet to overcome. This also helps to focus our approach, because many AI techniques are unable to work under conditions of uncertainty, hidden information, and real-time constraints. This also directs our attention at learning why newer AI techniques are not being adopted by the game industry. We specifically use the RTS game StarCraft as a testbed for our experiments because of its programmatic interface (BWAPI) and wide availability of learning data.

We use the architecture of an LBO system to pursue the aim because expert training data in these real-world or video game domains is usually relatively easy to produce, compared with defining the goals, heuristics or fitness metrics, or world and action models. Also, exploration-based approaches usually struggle with such large decision spaces. LBO also few assumptions about the domain, so LBO systems are intended to be able to cope with these challenges that are present in RTS and real-world problems. A caveat of this research as a whole is that, due to having limited time available, it is only ever applied to StarCraft, so these results may not extend to other applications. However, StarCraft is a very complex domain and no features or abstractions used were specifically related to the domain, so it is likely that these methods and results apply at least to other RTS games, and probably to other simpler domains as well.

At the time the work began, StarCraft was just becoming a widely used domain to study AI, and very little research had been done on LBO or related approaches in such complex domains. This necessitated an exploratory approach in this thesis, in order to investigate multiple different avenues and determine methods that may be effective in the domain. Since this work began there has been additional work in LBO for complex domains, particularly by Floyd and Esfandiari, and also work in LFD for RTS domains, particularly by Ontañón. Both use case-based techniques underlying their approaches, so in the later parts of this research, alternative techniques (data mining and BT building) were explored in order to broaden knowledge in the field and avoid repeating work on the same problems.

Overall, this thesis contributes towards the aim of making it easier to create AI systems for complex domains. Although it does not develop a complete solution to the problem, it helps to advance the field by showing the benefits and limitations of three different approaches to LBO when applied to StarCraft, including an entirely new method for building BTs by observation. The work in this thesis also contributes useful resources for future researchers with its comprehensive literature review and comprehensive StarCraft replay dataset. The remainder of the chapter discusses these main results and how they relate to the thesis objectives, followed by some possible directions for future research.

8.1 Discussion of Main Results

8.1.1 Literature Review

The thesis began with an introduction to research in game AI, RTS games, and StarCraft, followed by a comprehensive review of the literature on this topic (chapters 1 and 2). In this chapter, the main approaches to RTS AI are categorised based on application: as tactical decision-making, strategic decision-making, or plan recognition and learning. It is noted that most AI techniques used in the field are used almost exclusively in one of these categories: exploration-based methods are used for tactical decision-making, planning methods, especially CBP, are used for strategic decision-making, and CBR and probabilistic models are used for plan recognition and learning.

The chapter also identifies open research areas, including in particular game industry feedback about the lack of adoption of academic AI research. This lack of adoption is found to be caused by a perception that common industry techniques work reliably, while more sophisticated techniques are risky to use, unpredictable in behaviour, and importantly, that behaviour is difficult to customise. Additionally, the game industry is found to have different goals in agent behaviour: instead of trying to win or optimally solve a problem, the industry is trying to create AI that can lose in a human-like way, using varied behaviour across matches and avoiding being repeatedly exploitable with one approach. The findings of this section on game AI in industry helped to shape the requirements set out in section 4.2.

This chapter represents published work (Robertson and Watson, 2014b) that can be used by newcomers to the field of RTS AI to get an understanding of the major research directions in the field. They can find existing work to build upon, problems or techniques that haven't been explored, and avoid redoing work that has already been done. Thus, the review itself contributes to the field, helping to reduce the time required for future researchers to get started in the field.

8.1.2 Investigation of Case-Based Reasoning for LBO

Chapter 4 explores the use of CBR for LBO in StarCraft, directly addressing objective 1: *determine the feasibility of learning by observation in a complex domain using case-based reasoning*. This work is done by extending the jLOAF framework to work in StarCraft, and adapting a dataset of human games to be used as LBO learning traces. CBR-based LBO was evaluated in the complex domain of StarCraft by actually playing the game, and despite being unable to actually beat the built in scripted AI it showed an ability to pick correct actions in the early stages of the game. However, it suffered from difficulty choosing the correct order of actions when ordering mattered, as well as high computational overhead at run-time, and a dataset with insufficient information to make informed decisions at all levels. This suggests challenges to investigate in future work.

One of the limitations of this work is that the agent's decision-making ability was only tested by empirical interpretation of its in-game actions, and more rigorous testing of the decisions made would have been helpful. However LBO systems can be difficult to test (Ontañón et al., 2011) and even with the limited testing there were clear issues that helped to guide the following research. One of the main limitations of the agent itself was that it was too slow and computationally expensive to be practical for real-time decisions. A significant amount of learning data preprocessing and decision-making efficiency optimisation would be required to remedy this issue, and even then this approach may simply be unsuitable for applications in which computational resources are limited, like in games and robotics. Another limitation of the work lies in the dataset used, which was too imprecise and limited to make informed decisions about many low level tactical actions. A more precise and comprehensive dataset is needed.

This chapter contributes to the field by creating and testing a pure LBO system based on CBR. It highlights the importance of preprocessing and case base maintenance in the system, in order to handle large amounts of input data, remove irrelevant data, and to reduce or limit case base size – all contributing to reduce the memory usage and processing time required at runtime. It also shows a need for efficient case search algorithms and data structures to lower computational cost of search in practical CBR systems. In addition, it highlights the difficulty that CBR can have with correctly choosing a sequence of actions, and the corresponding challenge of how best to break learning traces into cases. The work in this chapter indicates that LBO with CBR would be feasible for complex domains, if the mentioned improvements to preprocessing and efficiency are made, thus completing objective 1. However, for real-time decisions with low computational resources available, like while running a game or on an embedded robot, CBR does not appear to be an appropriate choice of underlying decision-making process because it must do so much work at run-time.

8.1.3 Improved StarCraft replay dataset

The CBR-based LBO system detailed in chapter 4 relies on learning from StarCraft replay data. However, significant issues were found with the dataset during that work. Most notably, the dataset was not detailed enough to effectively make various decisions, as it was missing information on important game aspects like unit attributes, terrain layout, and player vision. Another notable issue was that the dataset was difficult to programmatically access because of its plain text format, which required extensive effort in parsing, checking, and converting data back into learning trace data for LBO. Chapter 5 remedies the issues found in chapter 4 with a new dataset, and thereby completes objective 2: *produce a comprehensive dataset of professional level StarCraft play that can be used for further analysis and learning*.

Creating a dataset with the required gameplay information has been a very common hurdle for starting research in the field of RTS AI. So, the main contribution in this chapter is the dataset itself, which is more comprehensive, accurate, flexible, accessible, and tested than prior work. The dataset is more comprehensive because it has the complete game state recorded, thereby including unit attributes, visibility information, and map information that was unavailable in prior work. It is more accurate because all state changes are recorded at one-second accuracy, where prior work was at most every 100 frames (approximately 4 seconds). It is also more flexible because the extraction process can be modified and re-run very quickly to update or add to the stored information without having to re-record any existing information. It is more accessible to both programs and humans because it is in a structured relational database, able to be queried easily and efficiently to find specific pieces of information, generate statistics, or reconstruct game states at any point in time (to the nearest second). Finally, it is more rigorously tested than before, resulting in this work finding and removing replays that produced errors and incorrect states, such as missing units, when extracted – likely due to a match being recorded in older version of StarCraft, making it play out differently when re-simulated.

Another contribution is the new extraction method itself, and the program code used to produce the dataset, which is publicly available so other researchers can modify the dataset or extract from different replays as needed for their work. This dataset can save many future researchers a large amount of time in producing their own dataset, and it is comprehensive and flexible enough to be used in a wide range of learning approaches and tasks, for both high- and low-level decision-making.

A limitation of the dataset is that, because it is still based on the large machine-gathered replay set from Synnaeve and Bessière (2012) there are potentially still replays in the set that have errors, despite more rigorous testing of the data. Another limitation is that the dataset covers fewer matches than prior work because approximately 40% of the replays were removed after errors were detected in them. A separate limitation is that for most users, much of the data recorded are likely irrelevant, and the resulting dataset is approximately 10x larger than the prior work. However, it is still far easier to select a subset of this data (and delete irrelevant data if needed) than to extract more comprehensive data to add to a restricted dataset.

8.1.4 Investigation of data mining for LBO

In the experimental CBR LBO implementation in chapter 4, one of the major limitations was that the system was too slow and resource-intensive at run-time, when computational resources and time are limited. The system also had issues with correctly ordering actions. Chapter 6 tries to address both of these issues by learning from the LBO trace data offline, using data mining techniques. It takes a bottom-up approach, learning many individual rules or patterns that could be used to inform or build a reasoning system. This would mean that the reasoner would not have to do as much computational work at run-time, and could learn when a specific action ordering was necessary. This work is aimed at objective 3: *explore the use of data mining to automatically discover domain knowledge by observation in a complex domain*.

In order to test out a variety of different data mining approaches and algorithms the SPMF framework (Fournier-Viger et al., 2014) was used. Algorithms for frequent itemset mining, association rule mining, sequential rule mining, and sequential pattern mining were tested, and all had major issues with being used for LBO. They had difficulty dealing with continuous-valued items, missing data, time-series data, and complex input structure (such as hierarchical or referential data, or unordered variable-sized lists). The algorithms were surprisingly inefficient with the large numbers of attributes present, especially if some attributes were irrelevant, often using huge amounts of memory and processing time. They were also inefficient at finding high-confidence (accurate) rules (or patterns or associations) that would be most useful for this research, as they instead focus on finding high-support (frequent) rules. Finally, the rules, patterns, or associations produced were often redundant or very nearly redundant restatements of other rules produced.

In response to the poor performance of existing data mining algorithms on this LBO problem, an alternative data mining algorithm was devised. This algorithm had the simpler task of mining 100% confidence rules – rules that always hold true in the data – regardless of frequency. Its efficiency was further increased by allowing "important attributes" to be noted so that the search could be limited to rules that are of interest. To deal with the many continuous-valued attributes, the algorithm was able to compare entire distributions of continuous attributes to find correlations with other items. It was effective at finding some rules that were recognisable as game rules, but because only 100% confidence rules were found, it would dismiss any strategies or rules-of-thumb if they aren't followed every time.

The limitations of this work mostly stem from the relatively little experience we have in data mining, and the minor part this work plays in the overall thesis. For example, there may be more effective ways of dealing with some of the issues encountered that we did not know to implement, and were not implemented in SPMF, so we may have used naïve solutions instead. Similarly, the slow and extremely resource-heavy performance of the data mining algorithms was an issue in this work, but it could have been due to poor algorithm implementations, which were used mostly as-is from SPMF. However, it seems unlikely that this well-maintained framework of a large number of current algorithms would use extremely inefficient implementations. Also, this work is not an exhaustive study of the algorithms available in the varied and active field of data mining, so there could be other algorithms we did not know about that were more appropriate to the problem we are dealing with in this work. An additional limitation is that the alternative data mining algorithm developed may not be appropriate for other domains or tasks. It was developed specifically for the problems encountered in this situation and was not tested in other domains, and it does rely on some simplification of the LBO problem (although not as much as the other data mining algorithms).

These results and the new data mining method are considered a minor contribution in this thesis, because they do not show an effective general data mining approach for LBO, and instead suggest that existing data mining approaches may not be general and robust enough to deal with LBO challenges. However, without a comprehensive coverage of data mining approaches, which is beyond the scope of this work, we cannot make a strong conclusion that no data mining algorithm is appropriate. These results also serve to explain why an alternative approach was taken in the following chapter. Instead of taking the possible route of building a decision structure from rules learned in this chapter in a bottom-up manner, a top-down approach was taken that did not rely on learned rules.

8.1.5 Method for building Behaviour Trees for LBO

Given that data mining in chapter 6 was not as effective as hoped, chapter 7 takes a different approach to address the issue of high run-time overhead identified in chapter 4. Like the data mining, it uses offline processing, but instead of learning individual rules to combine in a bottom-up fashion, it experiments with reducing entire learning traces in a top-down fashion to create a BT. It therefore addresses objective 4: *investigate offline learning by observation of a representation of sequential behaviour in a complex domain*. BTs were chosen as the behaviour representation due to their excellent properties when compared to the requirements in section 4.2: they are human-readable, understandable, and editable, they have very low run-time overhead, and they can produce effective strategic and reactive behaviour. BTs also inherently use sequences of actions so should not suffer from the action ordering issues found in chapter 4. Furthermore they are already widely used in the game industry, making them more easily understood and adopted.

The main contribution in this chapter was the new method developed for building a BT from learning traces. It works by finding and merging common patterns of actions amongst many traces, using a motif-finding algorithm to find and align patterns. The algorithm was applied to the StarCraft dataset and was able to create a BT consisting of all player actions in one *race* match-up and reduce it by 67%. The resulting BT structure was inspected, and found to depict recognisable early-game action sequences, but become more complex and unrecognisable further away from the early-game actions. The BT was also tested using cross-validation against unused expert traces, with a similar result: prediction accuracy is high to begin with but drops as the match continues. This is likely because the possible action sequences a player may take would increasingly diverge as different strategies are used, especially if a strategy is interrupted by an opponent later in the game, forcing reactive actions. This is a promising start to automatically building a BT from examples, in order to bring the desirable properties of BTs into LBO systems. Further work in this area should eventually make it easy to make practical AI systems for these complex domains.

This approach was very experimental, so there are a number of limitations that need to be addressed before BTs can be practically applied. Firstly, more research is required to further improve the BT structure. Currently, even after reducing in size, the BT is still too large to be edited by hand, except at the early stages in the game where there are fewer branches in the tree. Further reduction likely requires a different approach in order to identify conditional and reactive actions and differentiate them from regular decision points where strategies diverge. In both situations multiple actions are possible, but reactive actions are triggered by external influences and could happen any time, so are better represented as a high priority conditional sequence further up in the tree. Another limitation is that the BT building process is currently only used for strategic-level decisions, so is untested as a holistic approach for learning to play all aspects of the game. This was necessary to limit the tree complexity and the time taken to build trees, but means further investigation is needed to assess BT building for all gameplay actions. Finally, the current motif-finding algorithm used has a limitation that requires actions be converted into a limited set of items. This requires a mapping of actions to items and means actions are treated as completely the same or completely different from one another. This should be able to work with a similarity metric instead, ideally one that is automatically generated based on the observed data. Future work should be capable of addressing all of these limitations to allow BTs to be automatically generated for any LBO problem.

8.2 Future Directions

A possible direction for continued research from this point is to go back to the CBR LBO framework from chapter 4 and make it capable of automated optimisation of its case base. Although CBR has unavoidable run-time overhead that may make it unsuitable for real-time games and embedded systems on robots, it is still an excellent approach for many tasks. By improving upon the idea in jLOAF of a very generalised LBO framework, it may be possible to create a very widely applicable and even practically useful system. This system would need to be capable of automated feature selection and feature weighting in order to identify and remove useless features, and to make better choices when matching cases. It would need to implement automated case base maintenance, to generalise similar cases and remove redundant cases for a more compact case base with a better representation of the whole state space. It would also require automated indexing or other algorithms for efficient case retrieval, and an automatically produced similarity metric so that a user-defined one was not required. Finally, it would need to effectively learn action ordering from traces so that it correctly handles situations in which a specific order of actions is required. In order to create such a system, it would be sensible to use jLOAF as a starting point, and add on features as part of an open source project to encourage contributors to fill in solutions where known solutions exist. For the parts in which there are no solutions that work under the constraints of LBO, additional research is required. If all of these improvements were made, this system should be easy to apply to problems, so it could be applied in various domains to be tested and improved. Such a system would potentially even finally achieve the aim of this thesis, making it easy to apply a fairly general AI to complex domains.

Another possible area to investigate in future is an effective data mining approach for LBO. As demonstrated in chapter 6, current data mining algorithms do not appear to be effective for use in LBO. In order to be effective, the system created would have to be able to cope with complex input data, including hierarchical structures, time series data, missing items, varying sized collections, and continuous-valued items. The system would also have to handle large input data, in terms of number of learning traces, length of each trace, and number of input variables at each timestamp in the sequence. It would also have to efficiently handle irrelevant input variables being included without human intervention in preparing the data. An effective data mining approach for LBO would be highly beneficial for other LBO systems, as offline analysis of learning traces could determine rules and patterns, potentially allowing an LBO system to learn game (or world) rules, strategies, or common rules-of-thumb that humans follow, without requiring any human extra effort to specify them. Potentially, a further direction from learning these rules is to make the LBO system capable of using rules and patterns to plan, predict future states, and react to expectations about the future. This relates to work on GDA (Molineaux et al., 2010; Weber et al., 2010a) and HTN learning (Mohan and Laird, 2014; Nejati et al., 2006) which have managed to do this sort of complex reactive planning, but only when given much more information than is available to an LBO system.

Looking at improving on the work in chapter 7, further work could be done to improve the top-down reduction approach to BT building. BTs seem to be a very promising reasoning structure for these real-time, complex domains where online reasoning overhead is at a premium, but the existing BT builder cannot reduce the tree structure enough to be practically usable. One possible avenue to further reducing the tree size is to use an optimisation algorithm, such as simulated annealing, to make changes optimising for a smaller tree without changing its "meaning" (the actions that the tree would produce if executed). This would require a set of BT modification operations to be devised that would alter the tree structure without changing the meaning, allowing the optimisation algorithm to find more compact tree representations. Random noise in the learning traces, caused by mistakes or unusual actions, could also be eliminated in the tree optimisation by allowing the optimiser to change the "meaning" with some associated cost. Further reducing the tree should force common patterns to emerge because they more efficiently represent the decision process common across experts.

An alternative approach to building BTs in an LBO system would be to investigate a bottom-up approach to BT building. This approach could be used instead of combining rules learned through data mining to make a BT, or used in conjunction with, and guided by, those rules. The likely method for achieving this would be to incrementally construct BTs using evolutionary algorithms, creating a population of BTs, mutating them, and selecting for the ones that best represent the observed learning traces. This is similar to the work of Lim et al. (2010) but for a more general case – instead of learning a BT for a specific goal, the BT would attempt to represent whatever behaviour was observed in the learning trace. It is also similar to Fernlund et al. (2006) but is again a less restricted problem due building BTs instead of reasoning steps within a single context. These evolutionary algorithms require performance measures to work, but it is possible to use the learning traces to evaluate performance, as is done by Fernlund et al. (2006). This approach is also related to genetic programming, which has long been trying to find effective ways to evolve executable tree structures to represent particular inputs and outputs.

Building AI for complex domains such as StarCraft is clearly a difficult problem that will require a significant amount of future work to advance. It is on the forefront of the challenges faced in AI systems and has the potential to advance the wider field of AI research.

Top-k Non-redundant Rules (TNR) output A

This appendix shows a sample of the output from the *Top-k Non-redundant Rules* (TNR) data mining algorithm when run on our StarCraft dataset. Rules discovered are represented with antecedent item(s) on the left and consequent item(s) on the right, separated by an arrow. Each line also lists the support and confidence values for the rule.

```
SelectedUnit_Under_Storm=0  ==> SelectedUnit_Energy=0 SelectedUnit_Spell_Cooldown=0  #SUP: 314265 #CONF: 0.9999490901454431

SelectedUnit_Under_Storm=0 SelectedUnit_Starting_Attack=0  ==> SelectedUnit_Energy=0  #SUP: 314268 #CONF: 0.9999809083793123

SelectedUnit_Under_Storm=0  ==> SelectedUnit_Spell_Cooldown=0  #SUP: 314271 #CONF: 0.9999681813409019

SelectedUnit_Energy=0  ==> SelectedUnit_Under_Storm=0  #SUP: 314275 #CONF: 0.9999936361871852

Num_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  #SUP: 314279 #CONF: 0.99954837622169

Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314421 #CONF: 1.0

NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314421 #CONF: 1.0

SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314279 #CONF: 1.0

SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314279 #CONF: 1.0

NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  #SUP: 314279 #CONF: 0.99954837622169

SelectedUnit_Energy=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314275 #CONF: 1.0

SelectedUnit_Under_Storm=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314279 #CONF: 0.9999936362681804

SelectedUnit_Energy=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314275 #CONF: 0.9999936361871852

SelectedUnit_Energy=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314275 #CONF: 1.0

NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Energy=0 Num_Protoss_Archon=0  #SUP: 314275 #CONF: 0.9995356544251179

Num_Protoss_Archon=0  ==> SelectedUnit_Energy=0 NumOpponent_Protoss_Archon=0  #SUP: 314275 #CONF: 0.9995356544251179

SelectedUnit_Under_Storm=0  ==> SelectedUnit_Energy=0  #SUP: 314275 #CONF: 0.9999809088045412

NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Energy=0 SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  #SUP: 314273 #CONF: 0.9995292935268318

SelectedUnit_Under_Storm=0  ==> SelectedUnit_Starting_Attack=0  #SUP: 314274 #CONF: 0.9999777269386314

SelectedUnit_Starting_Attack=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314274 #CONF: 0.999993636166936

SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314274 #CONF: 1.0

SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314274 #CONF: 1.0

NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  #SUP: 314274 #CONF: 0.9995324739759749

Num_Protoss_Archon=0  ==> SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  #SUP: 314274 #CONF: 0.9995324739759749

SelectedUnit_Energy=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  #SUP: 314273 #CONF: 0.9999936361466868

SelectedUnit_Energy=0 SelectedUnit_Under_Storm=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314273 #CONF: 0.9999936361466868

SelectedUnit_Energy=0 SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314273 #CONF: 1.0

SelectedUnit_Energy=0 SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314273 #CONF: 1.0

SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Energy=0 Num_Protoss_Archon=0  #SUP: 314273 #CONF: 0.9999809086830491

SelectedUnit_Energy=0 Num_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  #SUP: 314273 #CONF: 0.9999936361466868
```

```
SelectedUnit_Energy=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314273 #CONF: 0.9999872723743704
Num_Protoss_Archon=0  ==> SelectedUnit_Energy=0 SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  #SUP: 314273 #CONF: 0.9995292935268318
SelectedUnit_Under_Storm=0 SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314272 #CONF: 1.0
SelectedUnit_Under_Storm=0 SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314272 #CONF: 1.0
SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  #SUP: 314272 #CONF: 0.9999936361264374
SelectedUnit_Under_Storm=0 SelectedUnit_Starting_Attack=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314272 #CONF: 0.9999936361264374
SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  #SUP: 314272 #CONF: 0.9999936361264374
Num_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  #SUP: 314272 #CONF: 0.9995261130776888
SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  #SUP: 314272 #CONF: 0.9999777267968907
SelectedUnit_Starting_Attack=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314272 #CONF: 0.9999872723338722
NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  #SUP: 314272 #CONF: 0.9995261130776888
Num_Protoss_Archon=0  ==> SelectedUnit_Spell_Cooldown=0 NumOpponent_Protoss_Archon=0  #SUP: 314271 #CONF: 0.9995229326285459
SelectedUnit_Spell_Cooldown=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314271 #CONF: 1.0
SelectedUnit_Spell_Cooldown=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314271 #CONF: 1.0
NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Spell_Cooldown=0 Num_Protoss_Archon=0  #SUP: 314271 #CONF: 0.9995229326285459
SelectedUnit_Spell_Cooldown=0  ==> SelectedUnit_Under_Storm=0  #SUP: 314271 #CONF: 0.999993636106188
SelectedUnit_Spell_Cooldown=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314271 #CONF: 0.999993636106188
SelectedUnit_Spell_Cooldown=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9999936360656886
SelectedUnit_Loaded=0  ==> SelectedUnit_Transport_Unit_Replay_ID=-1  #SUP: 314270 #CONF: 1.0
SelectedUnit_Patrolling=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314270 #CONF: 0.9999936360859383
SelectedUnit_Patrolling=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314270 #CONF: 1.0
SelectedUnit_Patrolling=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314270 #CONF: 1.0
Num_Protoss_Archon=0  ==> SelectedUnit_Patrolling=0 NumOpponent_Protoss_Archon=0  #SUP: 314270 #CONF: 0.9995197521794028
NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Patrolling=0 Num_Protoss_Archon=0  #SUP: 314270 #CONF: 0.9995197521794028
SelectedUnit_Energy=0  ==> SelectedUnit_Starting_Attack=0  #SUP: 314270 #CONF: 0.9999777266551482
SelectedUnit_Patrolling=0  ==> SelectedUnit_Under_Storm=0  #SUP: 314270 #CONF: 0.9999936360859383
SelectedUnit_Under_Storm=0  ==> SelectedUnit_Patrolling=0  #SUP: 314270 #CONF: 0.9999649994749922
SelectedUnit_Starting_Attack=0  ==> SelectedUnit_Energy=0  #SUP: 314270 #CONF: 0.9999809085008082
SelectedUnit_Transport_Unit_Replay_ID=-1  ==> SelectedUnit_Loaded=0  #SUP: 314270 #CONF: 1.0
SelectedUnit_Spell_Cooldown=0 Num_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9999936360656886
SelectedUnit_Spell_Cooldown=0 SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314269 #CONF: 1.0
SelectedUnit_Spell_Cooldown=0 SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314269 #CONF: 1.0
SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Spell_Cooldown=0 Num_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9999681811384152
SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  ==> SelectedUnit_Spell_Cooldown=0 NumOpponent_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9999681811384152
SelectedUnit_Spell_Cooldown=0 SelectedUnit_Under_Storm=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9999936360656886
NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Spell_Cooldown=0 SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9995165717302598
SelectedUnit_Spell_Cooldown=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9999872722123758
Num_Protoss_Archon=0  ==> SelectedUnit_Spell_Cooldown=0 SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9995165717302598
SelectedUnit_Under_Storm=0  ==> SelectedUnit_Spell_Cooldown=0 Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314269 #CONF: 0.9999618176090823
SelectedUnit_Loaded=0 Num_Protoss_Archon=0  ==> SelectedUnit_Transport_Unit_Replay_ID=-1 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Patrolling=0 SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Energy=0 SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  ==> Num_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Energy=0 SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Patrolling=0 SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  ==> NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Loaded=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Transport_Unit_Replay_ID=-1 Num_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Patrolling=0 Num_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999649992522568
SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  ==> SelectedUnit_Energy=0 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999809083793123
SelectedUnit_Transport_Unit_Replay_ID=-1 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Loaded=0 Num_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Patrolling=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999936360454387
SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Energy=0 Num_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999809083793123
SelectedUnit_Energy=0 SelectedUnit_Starting_Attack=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999936360454387
SelectedUnit_Patrolling=0 SelectedUnit_Under_Storm=0  ==> Num_Protoss_Archon=0 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999936360454387
SelectedUnit_Energy=0 NumOpponent_Protoss_Archon=0  ==> SelectedUnit_Starting_Attack=0 Num_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999777265134039
SelectedUnit_Transport_Unit_Replay_ID=-1 Num_Protoss_Archon=0  ==> SelectedUnit_Loaded=0 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 1.0
SelectedUnit_Under_Storm=0 Num_Protoss_Archon=0  ==> SelectedUnit_Patrolling=0 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999649992522568
SelectedUnit_Patrolling=0 Num_Protoss_Archon=0  ==> SelectedUnit_Under_Storm=0 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999936360454387
SelectedUnit_Energy=0 SelectedUnit_Under_Storm=0  ==> SelectedUnit_Starting_Attack=0  #SUP: 314268 #CONF: 0.9999777265134039
SelectedUnit_Energy=0 Num_Protoss_Archon=0  ==> SelectedUnit_Starting_Attack=0 NumOpponent_Protoss_Archon=0  #SUP: 314268 #CONF: 0.9999777265134039
```

Alternative rule mining output B

This appendix shows a sample of the output from our alternative rule mining algorithm when run on our StarCraft dataset. Rules discovered are represented with antecedent item(s) on the left and consequent item(s) on the right, separated by an arrow. Numeric attributes are represented by distribution summaries, showing the count, minimum, average, maximum, and number of unique items in the distribution. Lines are very long so have been wrapped and cut short.

```
{} ==> ActionDelayed(false), Num_Terran_Marine(0), Num_Terran_Ghost(0), Num_Terran_Vulture(0), Num_Terran_Goliath(0), Num_Terran_Siege_Tank_Tank_Mode(0),
       Num_Terran_SCV(0), Num_Terran_Wraith(0), Num_Terran_Science_Vessel(0), Num_Hero_Gui_Montag(0), Num_Terran_Dropship(0), Num_Terran_Battlecruiser(0),
       Num_Terran_Vulture_Spider_Mine(0), Num_Spell_Disruption_Web(0), Num_Terran_Command_Center(0), Num_Terran_Supply_Depot(0), ...
ActionCommand(Build) ==> BTbuildable(true), BTwalkable(65535),
       SelectedUnit_Energy({count=268930,min=0,avg=1026289.4,max=2000000000,numUnique=3}->{count=45493,min=0,avg=87925.6,max=2000000000,numUnique=3}),
       Num_Protoss_Corsair({count=268930,min=0,avg=0.0,max=15,numUnique=14}->{count=45493,min=0,avg=0.0,max=15,numUnique=11}),
       Num_Protoss_Archon({count=268930,min=0,avg=0.0,max=0,numUnique=1}->{count=45493,min=0,avg=0.0,max=1,numUnique=2}),
       Num_Protoss_Scout({count=268930,min=0,avg=0.0,max=15,numUnique=15}->{count=45493,min=0,avg=0.0,max=15,numUnique=12}), ...
ActionCommand(Train) ==>
       SelectedUnit_Energy({count=64737,min=0,avg=4232510.0,max=2000000000,numUnique=4}->{count=249686,min=0,avg=24030.2,max=2000000000,numUnique=2}),
       Num_Protoss_Scout({count=64737,min=0,avg=0.0,max=15,numUnique=12}->{count=249686,min=0,avg=0.0,max=15,numUnique=15}),
       Num_Protoss_Arbiter({count=64737,min=0,avg=0.1,max=10,numUnique=10}->{count=249686,min=0,avg=0.1,max=18,numUnique=13}),
       Num_Protoss_Scarab({count=64737,min=0,avg=0.0,max=3,numUnique=4}->{count=249686,min=0,avg=0.0,max=3,numUnique=4}),
       NumOpponent_Protoss_Scarab({count=64737,min=0,avg=0.0,max=3,numUnique=4}->{count=249686,min=0,avg=0.0,max=3,numUnique=4}),
       NumOpponent_Protoss_Arbiter_Tribunal({count=64737,min=0,avg=0.1,max=2,numUnique=3}->{count=249686,min=0,avg=0.1,max=2,numUnique=3}), ...
ActionTarget(Protoss_Robotics_Support_Bay) ==> SelectedUnit_Shields(20), Num_Protoss_Reaver(0), NumOpponent_Protoss_Reaver(0),
       Frame({count=314030,min=4,avg=17782.4,max=179820,numUnique=33412}->{count=393,min=5412,avg=11090.1,max=60808,numUnique=380}),
       SelectedUnit_Pos_X({count=314030,min=11,avg=893555.1,max=2000000000,numUnique=4015}->{count=393,min=11,avg=1928.9,max=4058,numUnique=339}),
       SelectedUnit_Pos_Y({count=314030,min=11,avg=893520.0,max=2000000000,numUnique=4009}->{count=393,min=15,avg=1915.6,max=4029,numUnique=355}),
       NumOpponent_Protoss_Interceptor({count=314030,min=0,avg=0.3,max=126,numUnique=118}->{count=393,min=0,avg=0.2,max=64,numUnique=3}),
       NumOpponent_Protoss_Observatory({count=314030,min=0,avg=0.6,max=2,numUnique=3}->{count=393,min=0,avg=0.7,max=2,numUnique=3}), ...
ActionTarget(Protoss_Observer), ActionTarget(Protoss_Probe) ==> ActionCommand(Train)
ActionTarget(Protoss_Air_Armor), ActionTarget(Protoss_Air_Weapons) ==> SelectedUnit_Type_ID(164)
ActionTarget(Protoss_Reaver), ActionTarget(Protoss_Shuttle) ==> SelectedUnit_Type_ID(155)
ActionTarget(Khaydarin_Amulet), ActionTarget(Mind_Control) ==> SelectedUnit_Type_ID(165)
ActionTarget(Protoss_Shuttle) ==>
       Frame({count=311083,min=4,avg=17784.5,max=179820,numUnique=33381}->{count=3340,min=5298,avg=16799.7,max=67560,numUnique=2941}),
       SelectedUnit_Pos_X({count=311083,min=11,avg=902000.7,max=2000000000,numUnique=4015}->{count=3340,min=48,avg=2028.5,max=4048,numUnique=122}),
        SelectedUnit_Pos_Y({count=311083,min=11,avg=901966.2,max=2000000000,numUnique=4011}->{count=3340,min=32,avg=1945.9,max=4032,numUnique=120}),
        SelectedUnit_Velocity_X({count=311083,min=-1393,avg=900081.1,max=2000000000,numUnique=934}->{count=3340,min=-742,avg=2.2,max=1002,numUnique=26}),
        SelectedUnit_Order_ID({count=311083,min=1,avg=900105.9,max=2000000000,numUnique=33}->{count=3340,min=23,avg=23.3,max=44,numUnique=2}),
        SelectedUnit_Idle({count=311083,min=0,avg=900081.8,max=2000000000,numUnique=3}->{count=3340,min=0,avg=0.7,max=1,numUnique=2}),
        SelectedUnit_Training({count=311083,min=0,avg=900081.7,max=2000000000,numUnique=3}->{count=3340,min=0,avg=0.3,max=1,numUnique=2}),
        Num_Protoss_Corsair({count=311083,min=0,avg=0.0,max=15,numUnique=14}->{count=3340,min=0,avg=0.0,max=12,numUnique=5}),
        Num_Protoss_Dark_Templar({count=311083,min=0,avg=0.4,max=19,numUnique=19}->{count=3340,min=0,avg=0.4,max=9,numUnique=10}),
        NumOpponent_Protoss_Scout({count=311083,min=0,avg=0.0,max=15,numUnique=15}->{count=3340,min=0,avg=0.0,max=7,numUnique=2}),
        NumOpponent_Protoss_Carrier({count=311083,min=0,avg=0.1,max=21,numUnique=22}->{count=3340,min=0,avg=0.1,max=16,numUnique=13}),
        NumOpponent_Protoss_Interceptor({count=311083,min=0,avg=0.3,max=126,numUnique=118}->{count=3340,min=0,avg=0.4,max=90,numUnique=19}),
        NumOpponent_Protoss_Scarab({count=311083,min=0,avg=0.0,max=3,numUnique=4}->{count=3340,min=0,avg=0.0,max=2,numUnique=3}),
        NumOpponent_Protoss_Fleet_Beacon({count=311083,min=0,avg=0.0,max=2,numUnique=3}->{count=3340,min=0,avg=0.0,max=1,numUnique=2})
ActionTarget(Protoss_High_Templar) ==>
        Frame({count=291759,min=4,avg=16875.4,max=179820,numUnique=32340}->{count=22664,min=7316,avg=29342.5,max=94460,numUnique=14198}),
        SelectedUnit_Training_Queue_Size({count=291759,min=0,avg=959696.7,max=2000000000,numUnique=7}->{count=22664,min=0,avg=0.5,max=5,numUnique=6}),
        SelectedUnit_Being_Constructed({count=291759,min=0,avg=959696.2,max=2000000000,numUnique=3}->{count=22664,min=0,avg=0.0,max=1,numUnique=2}),
        SelectedUnit_Completed({count=291759,min=0,avg=959697.2,max=2000000000,numUnique=3}->{count=22664,min=0,avg=1.0,max=1,numUnique=2}),
        Num_Protoss_Corsair({count=291759,min=0,avg=0.0,max=15,numUnique=14}->{count=22664,min=0,avg=0.0,max=14,numUnique=4}),
        Num_Protoss_Scout({count=291759,min=0,avg=0.0,max=15,numUnique=15}->{count=22664,min=0,avg=0.0,max=15,numUnique=10}),
        Num_Protoss_Carrier({count=291759,min=0,avg=0.1,max=21,numUnique=22}->{count=22664,min=0,avg=0.1,max=18,numUnique=18}),
        Num_Protoss_Interceptor({count=291759,min=0,avg=0.3,max=126,numUnique=118}->{count=22664,min=0,avg=0.6,max=116,numUnique=51}),
        Num_Protoss_Scarab({count=291759,min=0,avg=0.0,max=3,numUnique=4}->{count=22664,min=0,avg=0.0,max=2,numUnique=3}),
        Num_Protoss_Fleet_Beacon({count=291759,min=0,avg=0.0,max=2,numUnique=3}->{count=22664,min=0,avg=0.0,max=2,numUnique=3}),
        NumOpponent_Protoss_Corsair({count=291759,min=0,avg=0.0,max=15,numUnique=14}->{count=22664,min=0,avg=0.0,max=14,numUnique=4}),
        NumOpponent_Protoss_Scout({count=291759,min=0,avg=0.0,max=15,numUnique=15}->{count=22664,min=0,avg=0.0,max=15,numUnique=10}),
        NumOpponent_Protoss_Carrier({count=291759,min=0,avg=0.1,max=21,numUnique=22}->{count=22664,min=0,avg=0.1,max=18,numUnique=18}),
ActionTarget(Gravitic_Boosters) ==> SelectedUnit_Shields(250), SelectedUnit_Type_ID(159), Num_Protoss_Scout(0), Num_Protoss_Observatory(1),
        NumOpponent_Protoss_Scout(0), NumOpponent_Protoss_Observatory(1),
        Frame({count=314359,min=4,avg=17770.7,max=179820,numUnique=33410}->{count=64,min=12690,avg=34213.4,max=61285,numUnique=64}),
        SelectedUnit_Pos_X({count=314359,min=11,avg=892622.0,max=2000000000,numUnique=4015}->{count=64,min=48,avg=1590.0,max=4048,numUnique=28}),
        SelectedUnit_Pos_Y({count=314359,min=11,avg=892586.9,max=2000000000,numUnique=4011}->{count=64,min=32,avg=1860.5,max=4032,numUnique=27}),
        SelectedUnit_Velocity_X({count=314359,min=-1393,avg=890701.2,max=2000000000,numUnique=934}->{count=64,min=-1052,avg=-2.5,max=441,numUnique=4}),
        SelectedUnit_Velocity_Y({count=314359,min=-1431,avg=890698.7,max=2000000000,numUnique=923}->{count=64,min=-236,avg=2.9,max=817,numUnique=4}),
        SelectedUnit_Order_ID({count=314359,min=1,avg=890725.9,max=2000000000,numUnique=33}->{count=64,min=23,avg=28.8,max=76,numUnique=2}),
        SelectedUnit_Target_Position_X({count=314359,min=11,avg=892622.0,max=2000000000,numUnique=4026}->{count=64,min=48,avg=1590.0,max=4048,numUnique=28}),
        SelectedUnit_Target_Position_Y({count=314359,min=11,avg=892586.7,max=2000000000,numUnique=4007}->{count=64,min=32,avg=1860.5,max=4032,numUnique=27}),
        SelectedUnit_Upgrading_Upgrade_ID({count=314359,min=5,avg=890762.2,max=2000000000,numUnique=16}->{count=64,min=39,avg=58.6,max=61,numUnique=2}),
        SelectedUnit_Remaining_Upgrade_Time({count=314359,min=0,avg=890714.0,max=2000000000,numUnique=113}->{count=64,min=0,avg=218.7,max=2000,numUnique=3}),
        SelectedUnit_Idle({count=314359,min=0,avg=890701.9,max=2000000000,numUnique=3}->{count=64,min=0,avg=0.9,max=1,numUnique=2}),
        SelectedUnit_Upgrading({count=314359,min=0,avg=890701.4,max=2000000000,numUnique=3}->{count=64,min=0,avg=0.1,max=1,numUnique=2}),
        Num_Protoss_Corsair({count=314359,min=0,avg=0.0,max=15,numUnique=14}->{count=64,min=0,avg=0.1,max=2,numUnique=2}),
        Num_Protoss_Shuttle({count=314359,min=0,avg=0.3,max=7,numUnique=8}->{count=64,min=0,avg=0.6,max=3,numUnique=4}),
ActionTarget(Khaydarin_Core) ==> ActionCommand(Upgrade), SelectedUnit_Hit_Points(500), SelectedUnit_Shields(500), SelectedUnit_Type_ID(170),
        Num_Protoss_Reaver(0), Num_Protoss_Scarab(0), Num_Protoss_Robotics_Facility(1), Num_Protoss_Observatory(1), Num_Protoss_Arbiter_Tribunal(1),
        NumOpponent_Protoss_Reaver(0), NumOpponent_Protoss_Scarab(0), NumOpponent_Protoss_Robotics_Facility(1), NumOpponent_Protoss_Observatory(1),
        NumOpponent_Protoss_Arbiter_Tribunal(1),
        SelectedUnit_Pos_X({count=314372,min=11,avg=892585.1,max=2000000000,numUnique=4015}->{count=51,min=80,avg=1853.2,max=3984,numUnique=20}),
        SelectedUnit_Pos_Y({count=314372,min=11,avg=892550.0,max=2000000000,numUnique=4011}->{count=51,min=32,avg=1894.9,max=3712,numUnique=17}),
        SelectedUnit_Velocity_X({count=314372,min=-1393,avg=890664.4,max=2000000000,numUnique=935}->{count=51,min=0,avg=1.9,max=32,numUnique=2}),
        SelectedUnit_Velocity_Y({count=314372,min=-1431,avg=890661.8,max=2000000000,numUnique=924}->{count=51,min=0,avg=29.2,max=500,numUnique=3}),
        SelectedUnit_Order_ID({count=314372,min=1,avg=890689.1,max=2000000000,numUnique=33}->{count=51,min=23,avg=32.3,max=76,numUnique=3}),
        SelectedUnit_Researching_Tech_ID({count=314372,min=0,avg=890670.6,max=2000000000,numUnique=9}->{count=51,min=3,avg=5.9,max=6,numUnique=2}),
        SelectedUnit_Upgrading_Upgrade_ID({count=314372,min=5,avg=890725.4,max=2000000000,numUnique=17}->{count=51,min=44,avg=58.7,max=61,numUnique=2}),
        SelectedUnit_Remaining_Research_Time({count=314372,min=0,avg=890665.7,max=2000000000,numUnique=33}->{count=51,min=0,avg=0.5,max=12,numUnique=2}),
        SelectedUnit_Remaining_Upgrade_Time({count=314372,min=0,avg=890677.1,max=2000000000,numUnique=113}->{count=51,min=0,avg=343.1,max=2500,numUnique=3}),
        SelectedUnit_Idle({count=314372,min=0,avg=890665.1,max=2000000000,numUnique=3}->{count=51,min=0,avg=0.8,max=1,numUnique=2}),
        SelectedUnit_Upgrading({count=314372,min=0,avg=890664.6,max=2000000000,numUnique=3}->{count=51,min=0,avg=0.1,max=1,numUnique=2}),
        Num_Protoss_Shuttle({count=314372,min=0,avg=0.3,max=7,numUnique=8}->{count=51,min=0,avg=0.5,max=1,numUnique=2}),
         ...
```

Bibliography

Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59.

Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216.

Aha, D., Molineaux, M., and Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. In Muñoz-Ávila, H. and Ricci, F., editors, *Case-Based Reasoning. Research and Development*, volume 3620 of *Lecture Notes in Computer Science*, pages 5–20. Springer Berlin / Heidelberg.

Aha, D. W. and Molineaux, M. (2004). Integrating learning in interactive gaming simulators. In *Proceedings of the AAAI Workshop on Challenges in Game AI*.

Andreeva, V., Beland, J., Gaudreau, S., Floyd, M. W., and Esfandiari, B. (2014). Creating non-player characters in a first-person shooter game using learning by observation. In *Presented at the International Conference on Case-Based Reasoning* (ICCBR) *Workshop on Case-Based Agents*.

Argall, B. D., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483.

Baekkelund, C. (2006). Academic AI research and relations with the games industry. In Rabin, S., editor, *AI Game Programming Wisdom*, volume 3, pages 77–88. Charles River Media, Boston, MA.

Bakkes, S., Spronck, P., and van den Herik, J. (2011). A CBR-inspired approach to rapid and reliable adaption of video game ai. In *Proceedings of the Workshop on Case-Based Reasoning for Computer Games at the ICCBR*, pages 17–26.

Balla, R. and Fern, A. (2009). UCT for tactical assault planning in real-time strategy games. In *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI), pages 40–45.

Bauer, M. A. (1979). Programming by examples. *Artificial Intelligence*, 12(1):1–21.

Baumgarten, R., Colton, S., and Morris, M. (2009). Combining AI methods for learning bots in a real-time strategy game. *International Journal of Computer Games Technology*, 2009:10.

Beygelzimer, A., Kakade, S., and Langford, J. (2006). Cover trees for nearest neighbor. In *Proceedings of the International Conference on Machine Learning* (ICML), pages 97–104. ACM.

Bourgeois, F. and Lassalle, J.-C. (1971). An extension of the munkres algorithm for the assignment problem to rectangular matrices. *Communications of the ACM*, 14(12):802–804.

Buckland, M. (2005). *Programming Game AI by Example*. Wordware Publishing, Inc.

Buro, M. (2003). Real-time strategy games: a new AI research challenge. In *Proceedings of the IJCAI*, pages 1534–1535. Citeseer.

Buro, M. (2004). Call for AI research in RTS games. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, pages 139–142.

Buro, M. and Churchill, D. (2012). Real-time strategy game competitions. *AI Magazine*, 33(3):106–108.

Buro, M. and Furtak, T. M. (2004). RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference*, pages 63–70. Citeseer.

Cadena, P. and Garrido, L. (2011). Fuzzy case-based reasoning for managing strategic and tactical reasoning in StarCraft. In Batyrshin, I. and Sidorov, G., editors, *Advances in Artificial Intelligence*, volume 7094 of *Lecture Notes in Computer Science*, pages 113–124. Springer Berlin / Heidelberg.

Champandard, A. (2008). Getting started with decision making and control systems. In *AI Game Programming Wisdom*, volume 4, pages 257–264. Charles River Media.

Champandard, A. J. (2007). Behavior trees for next-gen game AI. Video. Retrieved 15 November 2012.

Champandard, A. J. (2011). This year in game AI: Analysis, trends from 2010 and predictions for 2011. http://aigamedev.com/open/editorial/2010-retrospective/. Retrieved 26 September 2011.

Champandard, A. J. (2012). Understanding the second-generation of behavior trees. Video. Retrieved 6 November 2013.

Chan, H., Fern, A., Ray, S., Wilson, N., and Ventura, C. (2007). Online planning for resource production in real-time strategy games. In *Proceedings of the International Conference on Automated Planning and Scheduling* (ICAPS), pages 65–72.

Chang, C.-H. and Lui, S.-C. (2001). Iepad: information extraction based on pattern discovery. In *Proceedings of the 10th international conference on World Wide Web*, pages 681–688. ACM.

Cheng, D. and Thawonmas, R. (2004). Case-based plan recognition for real-time strategy games. In *Proceedings of the GAME-ON Conference*, pages 36–40, Reading, UK. University of Wolverhampton Press.

Cho, H.-C., Kim, K.-J., and Cho, S.-B. (2013). Replay-based strategy prediction and build order adaptation for StarCraft AI bots. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 329–335.

Chung, M., Buro, M., and Schaeffer, J. (2005). Monte carlo planning in RTS games. In Kendall, G. and Lucas, S., editors, *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 117–124.

Churchill, D. and Buro, M. (2011). Build order optimization in StarCraft. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment* (AIIDE) *Conference*, pages 14–19.

Churchill, D. and Buro, M. (2012). Incorporating search algorithms into RTS game agents. In *Proceedings of the AIIDE Workshop on AI in Adversarial Real-Time Games*, pages 2–7. AAAI Press.

Churchill, D., Saffidine, A., and Buro, M. (2012). Fast heuristic search for RTS game combat scenarios. In *Proceedings of the AIIDE Conference*, pages 112–117.

Davis, I. L. (1999). Strategies for strategy game AI. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Computer Games*, pages 24–27.

Dereszynski, E., Hostetler, J., Fern, A., Dietterich, T., Hoang, T., and Udarbe, M. (2011). Learning probabilistic behavior models in real-time strategy games. In *Proceedings of the AIIDE Conference*, pages 20–25. AAAI Press.

Dicken, L. (2011a). A difficult subject. `http://altdevblogaday.com/2011/05/12/a-difficult-subject/`. Retrieved 19 September 2011.

Dicken, L. (2011b). A turing test for bots. `http://altdevblogaday.com/2011/09/09/a-turing-test-for-bots/`. Retrieved 19 September 2011.

Dill, K. (2006). Prioritizing actions in a goal-based RTS AI. In Rabin, S., editor, *AI Game Programming Wisdom*, volume 3, pages 321–330. Charles River Media, Boston, MA.

Erickson, G. K. S. and Buro, M. (2014). Global state evaluation in starcraft. In *Proceedings of the AIIDE Conference*.

Eriksson, J. and Tornes, D. Ø. (2012). Learning to play Starcraft with case-based reasoning: Investigating issues in large-scale case-based planning. Master's thesis, Norwegian University of Science and Technology.

Fernlund, H. K., Gonzalez, A. J., Georgiopoulos, M., and DeMara, R. F. (2006). Learning tactical human behavior through observation of human performance. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 36(1):128–140.

Florez-Puga, G., Gomez-Martin, M., Gomez-Martin, P., Diaz-Agudo, B., and Gonzalez-Calero, P. (2009). Query-enabled behavior trees. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):298–308.

Floyd, M. and Esfandiari, B. (2009). Comparison of classifiers for use in a learning by demonstration system for a situated agent. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Floyd, M. and Esfandiari, B. (2010). Toward a domain independent case-based reasoning approach for imitation: Three case studies in gaming. In *Proceedings of the Workshop on Case-Based Reasoning for Computer Games at the ICCBR*, pages 55–64.

Floyd, M. and Esfandiari, B. (2011a). Learning state-based behaviour using temporally related cases. Presented at the UK Workshop on CBR.

Floyd, M. and Esfandiari, B. (2013). Supplemental case acquisition for learning by observation agents. To appear in Computational Intelligence.

Floyd, M., Esfandiari, B., and Lam, K. (2008). A case-based reasoning approach to imitating robocup players. In *Proceedings of the International Florida Artificial Intelligence Research Society* (FLAIRS) *Conference*, pages 251–256.

Floyd, M. W. and Esfandiari, B. (2011b). A case-based reasoning framework for developing agents using learning by observation. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, pages 531–538, Boca Raton, Florida, USA.

Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu., C., and Tseng, V. S. (2014). SPMF: a Java Open-Source Pattern Mining Library. *Journal of Machine Learning Research (JMLR)*, 15:3389–3393.

Fournier-Viger, P., Nkambou, R., and Nguifo, E. M. (2008). A knowledge discovery framework for learning task models from user interactions in intelligent tutoring systems. In Gelbukh, A. and Morales, E., editors, *MICAI 2008: Advances in Artificial Intelligence*, volume 5317 of *Lecture Notes in Computer Science*, pages 765–778. Springer Berlin Heidelberg.

Fournier-Viger, P. and Tseng, V. S. (2012). Mining top-k non-redundant association rules. In *Foundations of Intelligent Systems*, pages 31–40. Springer.

Frith, M. C., Saunders, N. F. W., Kobe, B., and Bailey, T. L. (2008). Discovering sequence motifs with arbitrary insertions and deletions. *PLoS Computational Biology*, 4(5):e1000071.

Gabriel, I., Negru, V., and Zaharie, D. (2012). Neuroevolution based multi-agent system for micromanagement in real-time strategy games. In *Proceedings of the Fifth Balkan Conference in Informatics*, pages 32–39. ACM.

Gemine, Q., Safadi, F., Fonteneau, R., and Ernst, D. (2012). Imitative learning for real-time strategy games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 424–429. IEEE.

Grollman, D. and Jenkins, O. (2007). Learning robot soccer skills from demonstration. In *Proceedings of the IEEE International Conference on Development and Learning*, pages 276–281.

Hagelbäck, J. and Johansson, S. (2009). Measuring player experience on runtime dynamic difficulty scaling in an RTS game. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 46–52. IEEE.

Hagelbäck, J. and Johansson, S. J. (2008). The rise of potential fields in real time strategy bots. In *Proceedings of the AIIDE Conference*, pages 42–47. AAAI Press.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD explorations newsletter*, 11(1):10–18.

Hart, P. (1968). The condensed nearest neighbor rule (corresp.). *IEEE Transactions on Information Theory*, 14(3):515–516.

Hipp, J., Güntzer, U., and Nakhaeizadeh, G. (2000). Algorithms for association rule mining—a general survey and comparison. *ACM SIGKDD Explorations Newsletter*, 2(1):58–64.

Hogg, C., Kuter, U., and Munoz-Avila, H. (2009). Learning hierarchical task networks for nondeterministic planning domains. In *Proceedings of the IJCAI*.

Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *Proceedings of the AAAI Conference on AI*, pages 950–956.

Hostetler, J., Dereszynski, E., Dietterich, T., and Fern, A. (2012). Inferring strategies from limited reconnaissance in real-time strategy games. In *Proceedings of the Annual Conference on Uncertainty in Artificial Intelligence*, pages 367–376.

Hsieh, J. and Sun, C. (2008). Building a player strategy model by analyzing replays of real-time strategy games. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 3106–3111, Hong Kong, China. IEEE.

Huang, H. (2011). Skynet meets the swarm: how the Berkeley Overmind won the 2010 StarCraft AI competition. `http://arstechnica.com/gaming/news/2011/01/skynet-meets-the-swarm-how-the-berkeley-overmind-won-the-2010-starcraft-ai-competition.ars`. Retrieved 8 September 2011.

Ilghami, O., Nau, D. S., Muñoz-Avila, H., and Aha, D. W. (2005). Learning preconditions for planning from plan traces and htn structure. *Computational Intelligence*, 21(4):388–413.

Isla, D. (2005). Proceedings of the game developers conference: Handling complexity in the Halo 2 AI. Web page.

Jaidee, U., Muñoz-Avila, H., and Aha, D. (2011). Integrated learning for goal-driven autonomy. In *Proceedings of the IJCAI*, pages 2450–2455.

Judah, K., Roy, S., Fern, A., and Dietterich, T. G. (2010). Reinforcement learning via practice and critique advice. In *Proceedings of the Association for the Advancement of Artificial Intelligence* (AAAI) *Conference on AI*.

Kabanza, F., Bellefeuille, P., Bisson, F., Benaskeur, A., and Irandoust, H. (2010). Opponent behaviour recognition for real-time strategy games. In *Proceedings of the AAAI Workshop on Plan, Activity, and Intent Recognition*.

Kadlec, R. (2008). Evolution of intelligent agent behavior in computer games. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague.

Kitano, H., Tambe, M., Stone, P., Veloso, M., Coradeschi, S., Osawa, E., Matsubara, H., Noda, I., and Asada, M. (1998). The robocup synthetic agent challenge 97. In Kitano, H., editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*, pages 62–73. Springer Berlin Heidelberg.

Kolodner, J. L. (1993). *Case-based reasoning*. Morgan Kaufmann, San Mateo, CA.

Kotsiantis, S. and Kanellopoulos, D. (2006). Association rules mining: A recent overview. *GESTS International Transactions on Computer Science and Engineering*, 32(1):71–82.

Kryszkiewicz, M. (1998). Representative association rules and minimum condition maximum consequence association rules. In *Principles of Data Mining and Knowledge Discovery*, pages 361–369. Springer.

Laagland, J. (2008). A HTN planner for a real-time strategy game. Available: `http://hmi.ewi.utwente.nl/verslagen/capita-selecta/CS-Laagland-Jasper.pdf`.

Laird, J. and van Lent, M. (2001). Human-level AI's killer application: Interactive computer games. *AI Magazine*, 22(2):15–26.

Lanchas, J., Jiménez, S., Fernández, F., and Borrajo, D. (2007). Learning action durations from executions. In *Proceedings of the ICAPS Conference*.

Leece, M. A. and Jhala, A. (2014). Sequential pattern mining in StarCraft: Brood War for short and long-term goals. In *In Proceedings of the AIIDE Conference*.

Lidén, L. (2004). Artificial stupidity: The art of intentional mistakes. In Rabin, S., editor, *AI Game Programming Wisdom*, volume 2, pages 41–48. Charles River Media, Hingham, MA.

Lim, C., Baumgarten, R., and Colton, S. (2010). Evolving behaviour trees for the commercial game DEFCON. In Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A., Goh, C.-K., Merelo, J., Neri, F., Preuß, M., Togelius, J., and Yannakakis, G., editors, *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, pages 100–110. Springer Berlin / Heidelberg.

Liu, B., Hsu, W., and Ma, Y. (1998). Integrating classification and association rule mining. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*.

Lozano-Perez, T. (1983). Robot programming. *Proceedings of the IEEE*, 71(7):821–841.

Magnusson, M. M. and Balsasubramaniyan, S. K. (2012). A communicating and controllable teammate bot for RTS games. Master's thesis, School of Computing, Blekinge Institute of Technology.

Manslow, J. (2004). Using reinforcement learning to solve AI control problems. In Rabin, S., editor, *AI Game Programming Wisdom*, volume 2, pages 591–601. Charles River Media, Hingham, MA.

Marthi, B., Russell, S., Latham, D., and Guestrin, C. (2005). Concurrent hierarchical reinforcement learning. In *Proceedings of the IJCAI*, pages 779–785.

Mateas, M. and Stern, A. (2002). A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4):39–47.

Mehta, M., Ontañón, S., Amundsen, T., and Ram, A. (2009). Authoring behaviors for games using learning from demonstration. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Mehta, N. (2011). *Hierarchical structure discovery and transfer in sequential decision problems*. PhD thesis, Oregon State University.

Mille, A. (2006). From case-based reasoning to traces-based reasoning. *Annual Reviews in Control*, 30(2):223–232.

Mishra, K., Ontañón, S., and Ram, A. (2008). Situation assessment for plan retrieval in real-time strategy games. In Althoff, K.-D., Bergmann, R., Minor, M., and Hanft, A., editors, *Advances in Case-Based Reasoning*, volume 5239 of *Lecture Notes in Computer Science*, pages 355–369. Springer Berlin / Heidelberg.

Mohan, S. and Laird, J. E. (2014). Learning goal-oriented hierarchical tasks from situated interactive instruction. In *Proceedings of the AAAI Conference*.

Molineaux, M., Aha, D., and Moore, P. (2008). Learning continuous action models in a real-time strategy environment. In *Proceedings of the International FLAIRS Conference*, pages 257–262.

Molineaux, M., Klenk, M., and Aha, D. (2010). Goal-driven autonomy in a navy strategy simulation. In *Proceedings of the AAAI Conference on AI*, Atlanta, GA. AAAI Press.

Muñoz-Avila, H. and Aha, D. (2004). On the role of explanation for hierarchical case-based planning in real-time strategy games. In *Proceedings of ECCBR Workshop on Explanations in CBR*. Citeseer.

Naoki, M., McKay, B., Xuan, N., Daryl, E., and Takeuchi, S. (2009). A new method for simplifying algebraic expressions in genetic programming called equivalent decision simplification. In *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, pages 171–178. Springer.

Nejati, N., Langley, P., and Konik, T. (2006). Learning hierarchical task networks by observation. In *Proceedings of the International Conference on Machine Learning*, pages 665–672.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. (2015). RTS AI problems and techniques. *Encyclopedia of Computer Graphics and Games*. In Press.

Ontañón, S. (2012). Case acquisition strategies for case-based reasoning in real-time strategy games. In *Proceedings of the International FLAIRS Conference*.

Ontañón, S. and Floyd, M. (2013). A comparison of case acquisition strategies for learning from observations of state-based experts. In *Proceedings of the International FLAIRS Conference*, pages 387–392.

Ontañón, S., Mishra, K., Sugandh, N., and Ram, A. (2007). Case-based planning and execution for real-time strategy games. In Weber, R. and Richter, M., editors, *Case-Based Reasoning. Research and Development*, volume 4626 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin / Heidelberg.

Ontañón, S., Mishra, K., Sugandh, N., and Ram, A. (2008). Learning from demonstration and case-based planning for real-time strategy games. In Prasad, B., editor, *Soft Computing Applications in Industry*, volume 226, pages 293–310. Springer Berlin / Heidelberg.

Ontañón, S., Montana, J., and Gonzalez, A. (2011). Towards a unified framework for learning from observation. In *Proceedings of the IJCAI Workshop on Agents Learning Interactively from Human Teachers*.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and 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*, 5(4):293–311.

Orkin, J. (2004). Applying goal-oriented action planning to games. In Rabin, S., editor, *AI Game Programming Wisdom*, volume 2, pages 217–227. Charles River Media, Hingham, MA.

Palma, R., González-Calero, P., Gómez-Martín, M., and Gómez-Martín, P. (2011a). Extending case-based planning with behavior trees. In *Proceedings of the International FLAIRS Conference*, pages 407–412.

Palma, R., Sánchez-Ruiz, A., Gómez-Martín, M., Gómez-Martín, P., and González-Calero, P. (2011b). Combining expert knowledge and learning from demonstration in real-time strategy games. In Ram, A. and Wiratunga, N., editors, *Case-Based Reasoning Research and Development*, volume 6880 of *Lecture Notes in Computer Science*, pages 181–195. Springer Berlin / Heidelberg.

Pasula, H., Zettlemoyer, L. S., and Kaelbling, L. P. (2004). Learning probabilistic relational planning rules. In *Proceedings of the ICAPS Conference*, pages 73–82.

Perkins, L. (2010). Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Proceedings of the AIIDE Conference*, pages 168–173. AAAI Press.

Ponsen, M., Muñoz-Avila, H., Spronck, P., and Aha, D. (2005). Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning. In *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, pages 1535–1540. AAAI Press.

Ponsen, M., Muñoz-Avila, H., Spronck, P., and Aha, D. (2006). Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27(3):75–84.

Prentzas, J. and Hatzilygeroudis, I. (2007). Categorizing approaches combining rule-based and case-based reasoning. *Expert Systems*, 24(2):97–122.

Robbins, M. (2013). Personal communication. Software Engineer at Uber Entertainment, formerly Gameplay Engineer at Gas Powered Games.

Robertson, G. (2012). Applying learning by observation and case-based reasoning to improve commercial RTS game AI. In *Proceedings of the AIIDE Conference*.

Robertson, G. and Watson, I. (2012). Case-based learning by observation: Preliminary work. In *Proceedings of the Australasian Conference on Interactive Entertainment*.

Robertson, G. and Watson, I. (2014a). An improved dataset and extraction process for StarCraft AI. In *Proceedings of the FLAIRS Conference*.

Robertson, G. and Watson, I. (2014b). A review of real-time strategy game AI. *AI Magazine*, 35(4):75–104.

Robertson, G. and Watson, I. (2015). Building behavior trees from observations in real-time strategy games. In *Proceedings of the International Symposium on INnovations in Intelligent SysTems and Applications (INISTA)*, pages 361–367. IEEE.

Sailer, F., Buro, M., and Lanctot, M. (2007). Adversarial planning through strategy simulation. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 80 –87.

Sánchez-Pelegrín, R., Gómez-Martín, M., and Díaz-Agudo, B. (2005). A CBR module for a strategy videogame. In *Proceedings of the Workshop on Computer Gaming and Simulation Environments at the ICCBR*, pages 217–226. Citeseer.

Schaaf, J. W. (1996). Fish and shrink. a next step towards efficient case retrieval in large scaled case bases. In *Advances in case-based reasoning*, pages 362–376. Springer.

Schaeffer, J. (2001). A gamut of games. *AI Magazine*, 22(3):29–46.

Schmill, M. D., Oates, T., and Cohen, P. R. (2000). Learning planning operators in real-world, partially observable environments. In *Proceedings of the Artificial Intelligence Planning and Scheduling Conference*, pages 246–253.

Schwab, B. (2013). Personal communication. Senior AI/Gameplay Engineer at Blizzard Entertainment.

Scott, B. (2002). The illusion of intelligence. In Rabin, S., editor, *AI Game Programming Wisdom*, volume 1, pages 16–20. Charles River Media, Hingham, MA.

Shahaf, D. and Amir, E. (2006). Learning partially observable action schemas. In *Proceedings of the AAAI Conference*, pages 913–919.

Shantia, A., Begue, E., and Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micro management in StarCraft. Presented at the International Joint Conference on Neural Networks.

Sharma, M., Holmes, M., Santamaria, J., Irani, A., Isbell, C., and Ram, A. (2007). Transfer learning in real-time strategy games using hybrid CBR/RL. In *Proceedings of the IJCAI*.

Sigaty, C. (2008). Blizzard answers your questions, from Blizzcon. `http://interviews.slashdot.org/story/08/10/15/1639237/blizzard-answers-your-questions-from-blizzcon`. Retrieved 13 June 2012.

Smyth, B. (1998). Case-base maintenance. In *Tasks and Methods in Applied Artificial Intelligence*, pages 507–516. Springer.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT Press, Cambridge Massachusetts.

Synnaeve, G. and Bessière, P. (2011a). A bayesian model for plan recognition in RTS games applied to StarCraft. In *Proceedings of the AIIDE Conference*, pages 79–84. AAAI Press.

Synnaeve, G. and Bessière, P. (2011b). A bayesian model for RTS units control applied to StarCraft. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 190–196.

Synnaeve, G. and Bessière, P. (2012). A dataset for StarCraft AI and an example of armies clustering. In *Proceedings of the AIIDE Workshop on AI in Adversarial Real-Time Games*.

Szczepański, T. and Aamodt, A. (2009). Case-based reasoning for improved micromanagement in real-time strategy games. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Tozour, P. (2002). The evolution of game AI. In Rabin, S., editor, *AI Game Programming Wisdom*, volume 1, pages 3–15. Charles River Media, Hingham, MA.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.

Turner, A. (2012). Soar-SC: A platform for AI research in StarCraft: Brood War. `https://github.com/bluechill/Soar-SC/tree/master/Soar-SC-Papers`. Retrieved 15 February 2013.

Uriarte, A. and Ontañón, S. (2012). Kiting in rts games using influence maps. In *Proceedings of the AIIDE Workshop on AI in Adversarial Real-Time Games*, pages 31–36.

van Lent, M. and Laird, J. (2001). Learning procedural knowledge through observation. In *Proceedings of the International Conference on Knowledge Capture*, pages 179–186. ACM.

Wang, X. (1995). Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the ICML*, pages 549–557.

Watson, I. and Marir, F. (1994). Case-based reasoning: A review. *The Knowledge Engineering Review*, 9(4):327–354.

Weber, B. and Mateas, M. (2009). A data mining approach to strategy prediction. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 140–147. IEEE.

Weber, B., Mateas, M., and Jhala, A. (2010a). Applying goal-driven autonomy to StarCraft. In *Proceedings of the AIIDE Conference*, pages 101–106. AAAI Press.

Weber, B., Mateas, M., and Jhala, A. (2011a). Building human-level AI for real-time strategy games. In *Proceedings of the AAAI Fall Symposium Series*, pages 329–336. AAAI.

Weber, B., Mateas, M., and Jhala, A. (2011b). A particle model for state estimation in real-time strategy games. In *Proceedings of the AIIDE Conference*, pages 103–108. AAAI Press.

Weber, B., Mateas, M., and Jhala, A. (2012). Learning from demonstration for goal-driven autonomy. In *Proceedings of the AAAI Conference on AI*, pages 1176–1182.

Weber, B., Mawhorter, P., Mateas, M., and Jhala, A. (2010b). Reactive planning idioms for multi-scale game AI. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 115–122. IEEE.

Weber, B. and Ontañón, S. (2010). Using automated replay annotation for case-based planning in games. Presented at the Workshop on Case-Based Reasoning for Computer Games at the ICCBR.

Wender, S. (2015). *A Multi-Layer Case-Based & Reinforcement Learning Approach to Adaptive Tactical Real-Time Strategy Game AI*. PhD thesis, University of Auckland.

Wender, S., Cordier, A., and Watson, I. (2013). Building a trace-based system for real-time strategy game traces. In *Proceedings of the ICCBR Workshop on Experience Reuse: Provenance, Process-Orientation and Traces*.

Wender, S. and Watson, I. (2014). Integrating case-based reasoning with reinforcement learning for real-time strategy game micromanagement. In *PRICAI 2014: Trends in Artificial Intelligence*, pages 64–76. Springer.

Winner, E. and Veloso, M. (2003). Distill: Learning domain-specific planners by example. In *Proceedings of the ICML*, pages 800–807.

Wintermute, S., Xu, J., and Laird, J. (2007). SORTS: A human-level approach to real-time strategy AI. In *Proceedings of the AIIDE Conference*, pages 55–60. AAAI Press.

Woodcock, S. (2002). Foreword. In Buckland, M., editor, *AI Techniques for Game Programming*. Premier Press.

Yang, Q., Wu, K., and Jiang, Y. (2005). Learning actions models from plan examples with incomplete knowledge. In *Proceedings of the ICAPS Conference*, pages 241–250.

Yang, Q., Wu, K., and Jiang, Y. (2007). Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2):107–143.

Zaki, M. J. (2004). Mining non-redundant association rules. *Data mining and knowledge discovery*, 9(3):223–248.

Zhuo, H. H., Hu, D. H., Hogg, C., Yang, Q., and Munoz-Avila, H. (2009). Learning htn method preconditions and action models from partial observations. In *Proceedings of the IJCAI*, pages 1804–1810.