NEUROEVOLUTION FOR MICROMANAGEMENT IN REAL-TIME STRATEGY GAMES

By

Shunjie Zhen

Supervised by Dr. Ian Watson

The University of Auckland

Auckland, New Zealand

A thesis submitted in fullfillment of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science

The University of Auckland, February 2014

ABSTRACT

Real-Time Strategy (RTS) games have become an attractive domain for Artificial Intelligence research in recent years, due to their dynamic, multi-agent and multi-objective environments. Micromanagement, a core component of many RTS games, involves the control of multiple agents to accomplish goals that require fast, real time assessment and reaction. This thesis explores the novel application and evaluation of a Neuroevolution technique for evolving micromanagement agents in the RTS game StarCraft: Brood War. Its overall aim is to contribute to the development of an AI capable of executing expert human strategy in a complex real-time domain.

The NeuroEvolution of Augmenting Topologies (NEAT) algorithm, both in its standard form and its real-time variant (rtNEAT) were successfully adapted to the micromanagement task. Several problem models and network designs were considered, resulting in two implemented agent models. Subsequent evaluations were performed on the two models and on the two NEAT algorithm variants, against traditional, non-adaptive AI. Overall results suggest that NEAT is successful at generating dynamic AI capable of defeating standard deterministic AI. Analysis of each algorithm and agent models identified further differences in task performance and learning rate. The behaviours of these models and algorithms as well as the effectiveness of NEAT were then thoroughly elaborated

ACKNOWLEDGEMENTS

Firstly, I would like to thank Dr. Ian Watson in taking on this project for supervision and for valuable advice throughout.

Secondly, thanks to Glen Robertson for his extensive background research and knowledgeble insights into SC: BW AI. Also thanks to Stefan Wender for helpful discussions and knowledge of micromanagement.

Thirdly, thanks to Kevin for injecting some noise to an otherwise quiet and lonely office.

Lastly, thanks to my family and friends for support and keeping me sane throughout the year.

CONTENTS

Chapter 1

INTRODUCTION

This thesis details research on developing and evaluating Artificial Intelligence (AI) agents for Micromanagement in Real-Time Strategy (RTS) games. More specifically, the NeuroEvolution of Augmenting Topologies (NEAT) method is used to evolve neural networks as controllers for dynamic learning agents. The RTS game StarCraft: Broodwar (SC: BW) is the chosen test-bed for this work, due to its popularity in present AI research, and a large body of available existing work and support tools. This chapter discusses the motivations behind this research, introduces its main objectives and contributions, and describes the organization of the thesis chapters.

1.1. MOTIVATIONS

More than a decade ago, Laird and van Lent (2001) predicted that interactive computer games (or video games) would emerge as an ideal platform for AI research. Laird argued that the increasingly complex and realistic environments of computer games were highly suitable as test-beds to develop human-level AI. Many of these games require intelligent decision making at an expert level, and provide a platform on which to emulate and evaluate this ability (Weber, 2012). Testing AI techniques on these simulated environments also circumvents many of the costs and limitations associated with testing in the real-world (Buro, 2004).

Since then, AI research using video games have become hugely popular, partly driven by the enormous growth in the video game industry and the advancement of video game hardware and software technologies. On the other hand, the contribution of AI research to commercial game development has become lacking in recent times (Yildirim & Stene, 2008). This is in part due to the divide between AI research and game development interests; the former pushing the boundary of AI problem solving, and the latter in delivering cost effective entertainment software. The result of which is a high dependency on scripted, deterministic and non-adaptive AI techniques in commercial games, which limits its realism, replayability and challenge (Olesen, Yannakakis, & Hallam, 2008). There is continued motivation to explore machine learning approaches for generating dynamic, human-like AI players, to extend the replay value of games (Yannakakis & Hallam, 2007).

Real Time Strategy games are a genre of video games that provide unique challenges to AI research (Buro & Furtak, 2004). A call for research in 2004, initiated by Michael Buro, noted the characteristics of RTS games as having real-time, stochastic environments with multiple objectives and enormous action and state spaces (Buro, 2004). It was argued that tackling these characteristics would require advancements in many AI domains, such as adversarial planning under uncertainty, learning, opponent modelling and spatial and temporal reasoning. Since then, RTS games have emerged as popular and successful test-beds for AI research (Buro & Churchill, 2012). Despite significant research effort, RTS games continue to be an open research problem, where the most sophisticated AI are unable to defeat expert level human players.

An example of the genre is StarCraft: Brood War, an RTS game developed by Blizzard Entertainment in 2002[1]. Due to its deep strategy base, highly balanced gameplay and community support, it has enjoyed enormous popularity over the years. Especially popular in South Korea, it attracts professional level tournament play, with prize money totalling millions of dollars every year (Churchill & Buro, 2011). Because of its popularity, it has been well studied by scholars and has become a popular test-bed for AI research. The release of the BroodWar API (BWAPI) [2] open source framework has enabled the development of complex AI agents for SC: BW, and facilitates the competitive evaluation between different AI approaches and human players.

StarCraft AI is a multi-scale problem, with research primarily divided between two hierarchical scopes (Weber, 2012). The first is the strategic level, or macromanagement, which is concerned with high level decision making such as resource planning and opponent modelling. The other is the tactical level, also known as micromanagement, concerning quick and reactive individual unit combat and short term squad objectives. Micromanagement has been successfully modelled as a reinforcement learning task, due to a small enough state and action space for exploratory learning, and a definable immediate reward function (Shantia, Begue, & Wiering, 2011; Wender & Watson, 2012). Evolutionary learning approaches have the same potential to tackle this problem, by searching over a space of solutions guided by a fitness function heuristic.

The NeuroEvolution of Augmenting Topologies (NEAT) framework, developed by Ken Stanley, is a Neuroevolution method used in evolving artificial neural networks (Stanley & Miikkulainen, 2002b). It has been proven to be more effective than other Neuroevolution techniques and reinforcement learning methods on standardized tasks with continuous and high dimensional state spaces (Stanley, 2004).

[1] StarCraft: Brood War. Blizzard Entertainment. http://us.blizzard.com/en-us/games/sc

[2] The Brood War Application Programming Interface. https://code.google.com/p/bwapi/

Furthermore, neural networks are highly effective at approximating complex non-linear functions, as which micromanagement in SC: BW can be aptly modelled. There has only been one instance of applying NEAT to the task of micromanagement in literature (Gabriel, Negru, & Zaharie, 2012), which specifically focused on the real-time variant of the algorithm. This leaves much room to study, improve and evaluate further, the use of NEAT for SC: BW micromanagement.

In summary, the primary focus of this thesis is the exploration of the NEAT framework as an approach to the micromanagement task in the RTS game StarCraft: Broodwar. The work is situated in the domain of RTS game AI, which is motivated by having unique and challenging characteristics for AI research. This is further motivated by the appropriateness of RTS games as test-beds for AI research, and in particular the growing body of work and available tools surrounding SC: BW. The NEAT algorithm is hypothesized to be appropriate for tackling the micromanagement task, due to previous work demonstrating NEATs advantage over reinforcement learning approaches, and the appropriateness of modelling micromanagement as a reinforcement learning problem. In the next section, objectives of the research are discussed in more detail, along with a summary of contributions.

1.2. Objectives and Contributions

The overall goal of the research is to contribute to the development of a complete AI system capable of learning and executing human expert level strategy in RTS games. Such a system is necessarily multi-scaled, and requires solving multiple open problems in AI research. Instead of tackling the problem as a whole, the focus of the thesis is on micromanagement, a crucial level of abstraction in the RTS domain which handles the fast combat component of the overall game. In particular, SC: BW is the chosen testing platform, and NEAT is the chosen approach to the problem. There are three main objectives to this project.

The first is the design and implementation of a NEAT based agent into the SC: BW game environment. This involves gaining an understanding of the NEAT framework and the SC: BW game, designing a model of the game appropriate for the NEAT algorithm, and using tools such as BWAPI and open source implementations of NEAT to implement and apply the agent. Accomplishing this objective lays the foundation for analysing the performance of the agent and allows further improvements and extensions.

Once the agent is implemented, the second objective is the evaluation of its performance. The default SC: BW AI serves as a baseline of performance to demonstrate the effectiveness of this approach against the existing technique in the commercial game. Such an evaluation must be designed to cover a range of game variations and allow for statistically significant results. The measure of interest in these evaluations is the win rate and evolutionary learning rate of the AI agent over time.

The third objective is to explore different agent designs and NEAT framework settings. For example, there are two main variants of the NEAT algorithms (classic and real-time NEAT), and their properties in relation to micromanagement is worthy of exploration. There are also many possible ways to model an agent in micromanagement, which affects the design, performance and learning rate of the neural networks. The goal is essentially to expand the understanding of the NEAT framework applied to micromanagement, and identify areas of improvements and possible extensions.

The contributions of this thesis are inherently tied to the above objectives. Initially, these objectives were accomplished with the focus of evaluations on comparing the performance between generational and real-time NEAT. This work and its findings were considered of enough novelty and interest to merit publication (Zhen & Watson, 2013, see Appendix A). This thesis discusses these findings in more detail and also attempts to explore extensions.

Besides the successful accomplishment of these objectives, the thesis also details significant background on game AI and AI research. This serves to motivate the areas of research surrounding this work, and to provide important context for this work to be situated. In the next section, the organization of the thesis is summarized.

1.3. Organization

Due to the specialized topics discussed in this thesis, a significant portion of it is dedicated to explaining the background and related topic areas. In Chapter 2, more detail is provided on the history of game AI research, along with an extended literature review on work within RTS and SC: BW game AI. A number of techniques in the field are covered, especially those most closely related to what is proposed in this thesis, in the area of reinforcement learning and Neuroevolution.

Chapter 3 and Chapter 4 are dedicated to discussing two key topics in this thesis: the NEAT algorithm framework, and StarCraft as an AI test-bed. First, an overview of the theory of the NEAT algorithm is discussed, along with a practical exploration of how it works. Next, the attributes of StarCraft as a test-bed are examined further, as well as the game environment and the micromanagement task, including many key terms and concepts used later on.

Chapter 5 covers the actual design and implementation of the AI agent for micromanagement in SC: BW. A number of modules and agent designs are explored, as well as NEAT specific parameters such as the neural network and fitness function designs. The implementation of the agent using an open source version of NEAT and the BWAPI framework is described in detail.

Chapter 6 discusses a number of experiments used to evaluate the viability of the agent modules and designs, as well as the NEAT algorithm variants. The measure is the win rate of agents in playing micromanagement games against the default SC: BW AI. Results are analysed and discussed along with unit behaviour observations.

Chapter 7 serves as an overall discussion on the results of the evaluations and the limitations of the approach. The results are discussed in relation to the original research objectives, and future work is outlined. This chapter also provides concluding remarks and highlights the contributions, successes and limitations of this thesis work.

Chapter 2

Background and Related Work

This chapter provides a selective overview of game AI research, beginning with classic game AI and ending with the Real-Time Strategy game genre. The purpose of this is to establish a solid grounding of the research motivations and to illustrate the relevance of this work on a breadth first view of the field. Next, various related work topics are reviewed including StarCraft AI research in general, micromanagement approaches and applications of the NEAT algorithm in games. This provides a depth first view of the closely related work to the thesis.

2.1. Game AI

Schaeffer (2001) described games as ideal domains for the exploration of artificial intelligence. Games contained constrained and well defined rules; a contrast to the dynamic nature and unbounded scope of real world problems. The types of games Schaeffer reviewed were primarily 'classic' games, such as board games and card games. Games that were played in physical forms throughout history, long predating computers and interactive computer games. With the advent of digital computers and the birth of AI research, these games began to challenge and inspire AI advances.

2.1.1. Chess AI

One of the early goals of AI research was to develop a program capable of defeating world champion chess players. The game of chess has been held to rigorous intellectual analysis for hundreds of years and is often considered the ultimate one-on-one intellectual sport. Newell, Shaw, and Simon (1958) described the act of creating a successful chess machine as to have "penetrated to the core of human intellectual endeavour". Indeed in the early days of AI, it was unclear whether chess programs would ever be capable of defeating top chess masters.

In 1950, Claude Shannon published a seminal paper on the foundations of a chess playing program (Shannon, 1950). Shannon modelled chess playing as a tree of possible future moves, and the value of the resulting chess board for each of these moves. The basic approach was the Minimax procedure (choosing the move which minimizes the possible loss in the future by searching over future moves) using an evaluation function of chess positions (Marsland & Björnsson, 1997). The deeper the search and the more accurate the evaluation function, the better the program would be at making the best move. In 1953, Turing published details about an algorithm with a similar approach to playing a full game of chess (A. Turing, 1988). Lacking a powerful enough computer to execute the algorithm, Turing played a game with a human player using pen and paper to simulate the program. The program was not successful even against a weak player, but nevertheless demonstrated the possibility of machine automated chess playing.

Almost a half century later, IBM's famous chess program, Deep Blue, controversially defeated Gary Kasparov, the then world chess champion (Schaeffer, 2001). This accomplishment was the result of numerous scientific and engineering advances since Shannon's and Turing's work. A better understanding of the chess problem domain allowed more accurate evaluation functions (determination of the quality of a chess board), and knowledge engineering strategies such as the storing of opening moves and pre-analysed end game positions (Levy, 1988). Major algorithmic developments allowed faster and more efficient search, such as alpha-beta pruning which prematurely terminates tree search paths based on bound estimates on already searched paths (Richards & Hart, 1961). Enormous hardware improvements allowed ever faster and deeper searches of the game tree, allowing more accurate evaluation moves and enabling chess programs to satisfy real match time constraints (Thompson, 1982).

Although an admirable achievement, many viewed the narrow focus on chess playing leading to Deep Blue as limiting to AI research (Schaeffer, 2001). Schaeffer argued that chess was like a ball and chain shackle, stifling on the creativity of AI research, and that the success of Deep Blue finally allowed the field to move on to more interesting problems. This point was exemplified in an article by Thompson (1982), where the success of chess playing programs was equated to simply faster chess search engines. Because faster search speed allowed a program to search deeper and make better decisions, the milestones in chess program development became milestones in high-performance computing. Laird and Van Lent (2001) have argued that this kind of specialization of problems and solutions ultimately fails to address the true goals of AI research, which is to develop human level AI.

2.1.2. Checkers AI and Machine Learning

Chess was not the only classical game of focus in AI research. Schaeffer (2001) discussed games where computer programs are now undefeated against human players (Checkers, Othello and Scrabble) and games in which programs are highly competitive against world champions (backgammon and chess). AI research on checkers began around the same time as chess, and both games share many commonalities. Both are board games with perfect information and deterministic states, but the space-action complexity of checkers ($\mathbf{5 \times 10^{20}}$) is only about the square root of that of chess ($\mathbf{10^{40} - 10^{50}}$) (Schaeffer et al., 2007).

One of the first examples of a checkers playing program was created by Arthur Samuel (Samuel, 1959). Samuel's approach was innovative because the emphasis was on the ability to learn to play well, rather than focusing on optimized playing from the start. The actual playing strategy was similar to chess programs; by using a Minimax procedure and an evaluation function to search moves ahead. However, various techniques were also introduced for the program to improve its playing with experience, which were the earliest examples of machine learning in AI history.

One technique termed 'rote learning' was where the program cached results of Minimax evaluations and board positions, such that future searches could be extended by using previously cached results (Samuel, 1959). This would later inspire transposition tables, a common construct in game AI such as chess playing programs, to reduce redundant searches (Slate, 1987). The program also performed "learning by generalization", where the parameters of the value function polynomial (or evaluation function) was constantly updated through playing games against versions of itself. This allowed the program to improve its estimation of board positions, and therefore learn to play better checkers. The constant on-move updating of the evaluation function would later inspire Temporal Difference learning and its famous application to Backgammon (Tesauro, 1994)

In 1963, Samuel's program won an exhibition match against Robert Nealey, a self-proclaimed checkers master. From this, the program was hyped to have 'solved' the checkers game for AI. In reality, it was not until 1994 with 'Chinook' that a program officially claimed a world championship title, and would continue undefeated until its retirement (Schaeffer, 2001). By then, AI for checkers had evolved to mirror that of chess, emphasizing fast game tree search and utilising a database of opening and end game positions. In 2007, the Chinook team announced results for the weak solution of checkers (the perfect move is known from the start of the game, ensuring either a win or a draw), and that the best result possible from any human player against Chinook is a draw (Schaeffer et al., 2007).

Samuel's machine learning approach was a noteworthy alternative to the later heuristic search based strategies which dominated chess and checkers. It was an important precursor to machine learning today, which is now able to solve important real world problems using flexible and autonomous learning programs, sometimes without expert defined knowledge. The work of David Fogel on a self-evolving checkers program also had this goal in mind (Chellapilla & Fogel, 2001). Fogel's program, named Blondie24, played checkers using a Minimax algorithm like many others. However, the evaluation function was approximated by a neural network and evolved through an evolutionary algorithm. The neural network took in as input, a vector representing the checkers board position and output a value used in the Minimax procedure as an estimate of the desirability of that board. The training process started with a population of randomly weighted neural networks, which played checkers with each other, and eliminating the worst performing and breeding the best performing. The criteria for good performance was simply determined from points accrued in winning games against other networks, with no feedback for actions within individual games, and no expert knowledge injected to guide evolution. Through this process, the program was able to evolve to an expert level player, which was confirmed through playing against hundreds of real human opponents online.

Although Fogel's program became highly skilled, it was not in the same class as the top checkers playing programs which employed the classic search based approaches. Along the same line of machine learning approaches is Tesauro's TD-Gammon program, which was comparatively more successful against its competition (Tesauro, 1994). Instead of an evolutionary approach, TD-Gammon used Temporal Difference reinforcement learning to train a neural network as an evaluator of board positions. By playing against itself millions of times, TD-Gammon reached an expert playing level, which was competitive with the world's best human players and better than all previous programs.

There are obvious parallels between the approaches of TD-Gammon and Fogel's Blondie24, and the difference in achievements may be due to the differences in the underlying game. Checkers is a perfect information game with deterministic states and actions space, which is why it was possible to compute its eventual solution. Backgammon on the other hand has a stochastic gameplay element, namely in in the rolling of a dice to determine movement of pieces. Although balanced in the long run, this level of randomness in the short run may add enough complexity to allow machine learning techniques like reinforcement learning and evolutionary algorithms to be more feasible.

Other classic games with more challenging attributes include such games as Poker and Go (Karakovskiy & Togelius, 2012). Poker is a card game that features incomplete information and non-determinism, which requires techniques such as expert imitation (Rubin & Watson, 2012), opponent modelling and statistical models (Norris & Watson, 2013) to play well. Go is a board game with a high branching factor in its game states, making it much more difficult for search based solutions when compared to chess or checkers. Both of these games are currently open areas of research, with yearly competitions dedicated to the latest breakthrough techniques in their respective game domains. The trend of AI research is seemingly and intuitively to tackle problem domains with more and more challenging attributes, which better approximate complex problems in the real world.

While there are numerous challenges to AI research posed by classic games such as poker and Go, many have argued for a focus on interactive computer games as a medium for AI research (Buro, 2004; Karakovskiy & Togelius, 2012; Laird & van Lent, 2001). Laird argued that classical games, such as chess and checkers, only emphasized specific human capabilities such as search and decision making, and have therefore motivated very specific and specialized AI solutions. This is not to say that the focus on classic games have not produced good results. Schaeffer concluded in his paper by acknowledging the success of classic game AI and its contributions to various areas of computing, particularly for search based applications and high performance computing (Schaeffer, 2001).

However, in order to bridge the gap between constrained game environments and real world problems, it necessarily requires more realistic, complex and challenging games. Interactive computer games pose challenges that are both unique to the domain and present them in higher dimensions than classic games (Laird & van Lent, 2001). For example, while the game of poker poses the challenge of incomplete information, opponent deception and non-determinism, certain genres of interactive computer games require capabilities such as visual pattern recognition, spatial navigation and reasoning, prediction of environmental dynamics, short-term memory, quick reactions, limited information and multi-dimensional states and actions, all at once occurring in real-time (Buro & Furtak, 2004). In the next section, the era of the Interactive Computer game is introduced, and its implications for AI research discussed.

2.1.3. Interactive Computer Games

Interactive computer games, or video games, are games that "use the computer to create virtual worlds and characters for people to dynamically interact with" (Laird & van Lent, 2001). They differ from classical board and card games in the complexity of its virtual environments and the scope of its required human capabilities. Video games, as a form of entertainment, have become hugely successful and pervasive in popular culture, with an industry posting $144.3 billion revenue in the US alone in 2009 (Siwek, 2010). The major trend for the video game industry is its ever increasing realism in game environments, with the aim of providing an increasingly immersive experience for players.

Realism in video games is affected by two main aspects. The first is the realism of graphical rendering, which has complimented great advances in the computer graphics field, to deliver ever more realistic rendering of virtual environments and characters. The second is the realism of the behaviour of virtual characters, which is achieved through AI techniques. While improvements in computer graphics are able to scale with yearly improvements in hardware speed (faster computing allows faster and more complex rendering), AI is less straight forward. Realistic AI behaviour cannot simply be achieved through faster hardware, but requires better techniques in approximating human capabilities.

More realistic AI is argued to improve gameplay experiences (Laird & van Lent, 2001). Gamers are driven towards online multiplayer games because the AI often lack the same level of challenge that makes for an interesting and enjoyable competitor. Popular games such as World of Warcraft[3] combine computer controlled characters and environments with large scale human player interactions. Others such as League of Legends[4] and Dota 2[5], are strictly human player-to-player focused, thus facilitating cooperative and competitive team gameplay with minimal AI involvement. Such games tend to enjoy long term popularity, as human players offer continuously varied and unpredictable challenge. Human level AI in games without multiplayer aspects would afford more dynamic and engaging interactions with human players, enhancing the game experience and prolonging its replayability (Laird & van Lent, 2001).

AI in video games is equally important to AI researchers for a number of reasons. Firstly, the increasing realism in computer games make them attractive test-beds for AI techniques (Laird & van Lent, 2001). Creating expert level AI to play video games requires tackling many more dimensions of human intellect than classical game AI, making it a greater and more realistic challenge (Weber, Mateas, & Jhala, 2011). Secondly, by abstracting away issues in real world environments, researchers can focus on testing the actual AI (Buro & Furtak, 2004). Computer game software and hardware is cheap when compared to building home-grown simulations or physical robots to test in the real world, and more and more often they come with high customization support. In the following sections, examples of commercial games are discussed, including those used in research as test beds.

2.1.4. Commercial Game AI

Many game AI innovations occurred during the 'golden age of arcade video games'[6] (late 1970s and early 1980s) in order to support the rise of games based around AI opponents. These were largely simple, static and logic based techniques used in creative ways. For example in the classic game Pac-Man, individual enemy 'ghosts' had different movement patterns to give the illusion of separate personalities (Mateas,

[3] World of Warcraft. Blizzard Entertainment Inc. us.battle.net/wow/en/
[4] League of Legends. Riot Games. http://beta.na.leagueoflegends.com/
[5] Dota 2. Valve Corporation. www.dota2.com/
[6] Timeline of Video Games. The History of Computing Project. www.thocp.net/software/games/games.htm

2003). Due to hardware limitations and the genre defining 2D designs of this era, much of the AI in these games was about following scripted movement patterns.

By the 1990's, advancements in hardware had prompted the rise of new game genres with novel interactions and AI opponents. Genres such as First Person Shooters (FPS) introduced fast action gameplay in a 3D environment, and Real-Time Strategy Games increased the number and complexity of computer characters (or units) within the game environment. This motivated the adoption of techniques, such as Finite State Machines, to more efficiently model and control complex agent behaviour, and better search algorithms such as A*, for agent path finding (Nareyek, 2004). These techniques allowed agents to exhibit more complex behaviour, but was still static in nature and did not promote the generation of novel or adaptive behaviour.

One of the first instances of machine learning AI in a commercial game was Creatures, a game published in 1996 (Zielke et al., 2009). The objective of the game was to nurture artificial creatures called Norns through an artificial life cycle. Norns had neural network controllers as brains, which were modified throughout the game by feedback from players. The architecture of the networks were also mutated and reorganized via genetic breeding game mechanics. The setting and design of the game made it extremely appropriate for the adaptation of machine learning techniques. Another example is Black and White, a game published in 2001 by Lion Head Studios (Yildirim & Stene, 2008). The game fell into the genre of 'god games' where the player is given great control over a simulated environment of artificial life. A key mechanic of the game is the control of a mythical creature which exists within the game environment. Implemented using reinforcement learning approaches, the creature exhibits learning behaviour towards feedback from player interactions.

Other examples of machine learning in games include applications in the driving game genre. For example, in the commercial driving game 'Colin McRae 2.0' published in 2001, a neural network approach is used to generate AI driving opponents of varying skill[7]. The training was done before publishing, so no new learning occurred as the game is played. The driving game 'Forza Motorsport' was shipped in 2005 with 'Drivatar', a technology created by Microsoft Research for the basis of all AI opponents in the game[8]. Drivatar continuously receives information about the players driving style and strategy, and incorporates this into a probabilistic model, used to control various AI opponents. Besides uses in creating AI opponents, machine learning techniques have also been applied to multiplayer game match making. Microsoft's large scale Xbox live online multiplayer service uses a Bayesian Inference technique to classify and track player skill levels, in order to provide skill balanced matches between players of varying skills (Herbrich, Minka, & Graepel, 2006).

[7] Interview with Jeff Hannan. AI Junkie. www.ai-junkie.com/misc/hannan/hannan.html

[8] Drivatar. Microsoft Research. http://research.microsoft.com/en-us/projects/drivatar/

Although there have been a few successful applications of machine learning techniques in commercial games, the majority of past and current games do not employ such techniques. A number of papers in literature have discussed the divide between commercial and academic AI, citing numerous problems of using academic techniques in commercial practise despite its great potential (Herik, 2005; Nareyek, 2004; Robertson & Watson, in press; Yildirim & Stene, 2008). Simple and static techniques such as scripting, rule based systems, finite state machines and decision trees are pervasive in commercial games because they are proven to work, easy to understand and implement and provide predictable results. In contrast, machine learning techniques in academia are often too complicated or inefficient to implement, especially in the context of time and resource constraints of commercial game development. Behaviour generated from machine learning techniques may also be difficult to test and debug (for example neural networks are complex to analyse), and games that are released with mechanisms for generating new behaviour cannot guarantee consistent nor desirable AI behaviour for all players.

Commercial games are popular test-beds for academic AI research, despite the low adoption of the results in the industry. This is partly because the goal of most researchers using commercial games as test-beds is not aiming to improve game AI, but to explore or improve techniques in approximating human intellectual capabilities. That is not to say that no research in the area directly addresses commercial game AI challenges (e.g. Bakkes et al., 2009; Olesen et al., 2008; Spronck, Ponsen et al., 2006). Rather that the complementary relationship between the video game industry and AI research is currently one sided in contribution, to the favour of AI research. In the next section, I discuss such examples of commercial games that have been heavily used in AI research.

2.1.5. Test-Beds for AI Research

Numerous commercial games have been used for AI research in literature, many of which have inspired competitive events where different techniques can be comparatively evaluated. Competitions based on simple, arcade style games include Ms. Pac-Man (Lucas, 2007), Cellz (Lucas, 2004) and X-pilot (Parker & Parker, 2007). Even these simplified games inspire the use of machine learning techniques such as evolutionary algorithms and neural networks. More complicated genres of games such as FPS, racing, RTS and platforming have also been used.

The Mario AI benchmark is a popular game benchmark for AI techniques, based on the commercial platforming game Super Mario Bros (Karakovskiy & Togelius, 2012). Platforming games are ones which the player navigates a character progressively through an environment with numerous platforms. In Super Mario Bros, the environment is two-dimensional, and the character can perform left and right walking, running and jumping actions. Other mechanisms add to the state and action complexity, such as collecting power-ups which changes the character's state, enemy characters which must be avoided or killed, and optional extra point objectives which represents a problem with multi-objective optimization.

In 2010, the competition was split into four separate areas and emphasized different types of AI techniques (Karakovskiy & Togelius, 2012). The first was the 'Gameplay Track', with the objective of an AI to clear as many levels as possible. The levels were predetermined and therefore encouraged techniques with offline optimization. The second was the 'Learning Track' which involved presenting unseen levels to agents during a training phase of 10,000 plays, and scoring on performance on the 10,001st play. Submissions involved learning and non-learning agents, with learning agents being clear winners even though non-learning agents performed well on the Gameplay Track. The other two tracks were 'Level Generation' (Shaker et al., 2011) and 'Turing Test' (Togelius, Yannakakis, Karakovskiy, & Shaker, 2012), which tested procedural content generation and human behaviour imitation. These tests used more subjective quality measures and demonstrated the unique human capabilities that interactive computer games can be suitable test-beds for.

Another popular test-bed game is the Unreal Tournament series of games (Hoang et al., 2005). It falls under the FPS genre of games, which is centred on projectile weapon based combat in a first person view. In a normal game, the player controls a single character in a three dimensional environment, that is able to navigate the level and interact with objects or other players, such as shooting them with a gun. The primary objective is to eliminate opposing enemies and/or to capture and hold strategic assets. Commercial AI for this genre of games have traditionally employed a finite state machine in an event driven reactionary paradigm.

The work of Hoang et al., (2005) first explored hierarchical (HTN) planning techniques to model and accomplish higher level 'grand strategies', while still retaining reactionary control for individual bots to deal with the highly dynamic and real time environment. The main idea is to decompose a game mode objective into goals of varying hierarchy, with the lowest being individual bot actions determined by a finite state machine or static rules. In Smith, Lee-Urban, & Muñoz-Avila (2007), the strategy of a bot team is not static, but learnt online using reinforcement learning. A variation of Q-Learning is applied to a simple state formulation which models the locations of strategic assets and the action of sending various bots to those locations. Again the individual bot actions are fixed and represented with static techniques, and so these techniques are more concerned with high level strategy, rather than individual bot actions. This abstraction seems to ignore the unique feature of the FPS genre, primarily the first person perspective shooting gameplay, and the proposed techniques may be more direct and suitable if applied to the RTS genre.

Perhaps more interesting is the use of Unreal Tournament in the 'BotPrize' series of competitions (Hingston, 2009). The competition is a version of the Turing test first posed by Alan Turing (A. M. Turing, 1950). The idea is that bots and humans play a series of games in UT, and each player is judged to be human or not. The bot with the highest 'human-ness' rating wins the competition for the year, and a rating higher than 50% indicated the test was passed. Starting in 2008, the test was not beaten until 2012, when two winners emerged. One winner was 'MirrorBot', a largely rule based approach (Polceanu, 2013). Using graph based navigation and rule based target selection as default behaviour, and a frame recording and playback technique for mirroring human player actions, MirrorBot received a humanness rating of 52.2%. The other winner employed a more sophisticated technique, including a replaying of human actions from a database of human played games, and a default controller evolved through Neuroevolution (Schrum, Karpov, & Miikkulainen, 2012). Neuroevolution selected for good performance under behavioural constraints (such as range and movement decreasing aim accuracy), resulting in good performance with believable human-like flawed behaviour.

The ability to generate AI opponents which perform human-like behaviour would be a valuable tool for commercial games. However, the most successful techniques thus far in Turing-like tests only aim to imitate human behaviour, and fool human judges by giving an appearance of human intelligence (mainly by emphasizing human flaws rather than strengths). The judging standard is also subjective to the perception of the human judges and have high margins of error (for example in 2012, the human judges on average did not pass the test themselves[9]). A better test-bed is one which emphasizes and requires the strengths of human intellectual capabilities, and which is judged in a competitive setting that can objectively determine a winner without subjective bias. In the next section the RTS game genre is discussed, which is an example of such a test-bed, as it requires numerous human intellectual capabilities, exhibits balanced gameplay, and has fair and competitive outcomes.

2.1.6. Real-time Strategy Games

Real-Time Strategy games are a genre of video games which can be viewed as simplified military simulations (Buro, 2003). Players control units and structures on a two-dimensional terrain in real-time, and aim to secure resources, build additional units and structures, and eliminate opponents. Buro noted in 2003 that RTS games offered a larger variety of problems for the AI community to tackle, than other game genres studied thus far. Such non-trivial problems include: adversarial planning under uncertainty, learning and opponent modelling and spatial and temporal reasoning.

Buro also saw the importance of competitive evaluation for driving innovation in the AI community, which is evident in research on chess and checkers. RTS games were a natural fit to extend this

[9] Botprize 2012. http://botprize.org/result.html

competitiveness, by allowing AI bots to play against each other in annual competitions. One barrier to RTS research at the time was the lack of tools and infrastructure to easily experiment AI techniques on RTS games. Commercial game companies were not inclined to open source or interface their game code to easily allow AI modules to be integrated. This and other technical issues motivated the ORTS project, the creation of an open source RTS game as a flexible framework for AI research (Buro & Furtak, 2004). A direct result was the first RTS game AI tournament in 2006, where researchers from all around the world competed within the ORTS framework (Buro, Bergsma, & Deutscher, 2006).

Instigated by the release of the BWAPI interface, the RTS research community shifted from using ORTS to SC: BW (Churchill, Saffidine, & Buro, 2012). This shift was partly due to the popularity of SC: BW as a commercial game, which draws more mainstream attention to RTS research. More importantly though, it was enabling AI techniques to compete against human players available from a large and active player base. Apart from validation of results against human experts, this also enabled research and techniques which mimicked or learnt from a large record of human replays (Cho, Kim, & Cho, 2013; Hsieh & Sun, 2008; Weber & Ontañón, 2010).

The first SC: BW tournament occurred as part of the 2010 Conference on Artificial Intelligence and Interactive Digital Entertainment[10], and continues annually today. The competition was split into four separate tournaments, with tasks in increasing complexity. The first two tournaments were dedicated to micromanagement battles and did not involve building units or resource gathering. The other two tournaments were focused on the full scope of the SC: BW game, and the highest complexity tournament played a full game. Since then, all tournaments (including subsequent tournaments in CIG[11] and the SSCAI[12]) only have the category of full SC: BW games. This poses a challenge for AI research which targets particular sub problems of the SC problem domain, such as micromanagement. In order to be evaluated as part of these tournaments, these techniques must be incorporated with other modules which form an all-encompassing AI capable of playing a full SC game. However, the success of the AI technique in review is then dependent on the performance of the other modules.

2.1.7. Summary of Game AI

In the above sections, I provided a comprehensive overview of many areas of game AI research. Starting from a historical perspective on the use of classical games in AI research, I discussed work with chess and

[10] AIIDE StarCraft competition hosted by Expressive Intelligence Studio. http://eis.ucsc.edu/StarCraftAICompetition
[11] Computational Intelligence and Games. http://ls11-www.cs.uni-dortmund.de/rts-competition/starcraft-cig2013
[12] Student StarCraft AI Tournament. http://www.sscaitournament.com/

checkers AI. These games spurred advances in search based techniques and also motivated the founding of machine learning strategies. More recently, the focus of AI research has shifted to interactive computer games, due to more realistic and challenging problem attributes. I discussed historical advances in commercial game AI and their usage as test-beds for AI research. There is currently a divide between commercial game AI techniques and AI research, with the former relying on deterministic approaches and the latter focused on developing dynamic and emergent systems.

I discussed numerous commercial games used as test beds for AI research, and some of the existing approaches. In particular, the RTS genre is examined in more detail, including the current popular use of SC: BW. In the next section, the focus will shift to more closely related work. First, examples of published AI techniques used to play SC is examined, briefly for macromanagement, and in more detail for micromanagement. Next, I discuss related work on reinforcement learning and NEAT, and how they are related in applications that are similar to the micromanagement task.

2.2. Related Work

There is a large body of existing work which uses SC: BW as a test-bed. With the best approaches still being unable to defeat human experts in annual tournaments, this work is expected to grow in the future. In this section, some of these approaches are addressed in both macromanagement and micromanagement areas. Part of this review is based on a comprehensive survey paper addressing AI research in RTS games (Robertson & Watson, in press). Related work in reinforcement learning and the NEAT framework is also discussed, particularly in instances where these techniques are applied to SC: BW, or similar game areas. Related work examined in this section involve discussions about SC: BW gameplay, which is explained in full detail in Chapter 4.

2.2.1. Macromanagement Techniques

Ben G Weber, (2012b) described macromanagement as working "towards long-term goals, such as building a strong economy and developing strategies to counter opponents." In Robertson & Watson (in press), many techniques applied to macromanagement in RTS games are discussed. Since macromanagement is not the focus of this thesis, only techniques that are specifically applied to SC: BW are highlighted here.

Weber, Mawhorter, Mateas, and Jhala (2010) advocated the use of reactive planning as a technique for authoring multi-scale game AI. It is an extension of classical planning, which formalizes a model of the problem domain, defining operators with pre and post conditions, and builds plans to solve the problem via heuristic search over the operators. The problem with classical planning for RTS games is that the dynamic and stochastic states require constant re-planning, which is compounded by multi-scale goals requiring multi-level plans and synchronization between them. Reactive planning solves these problems by supporting incremental decomposition and execution of tasks in real-time, allowing coordination between multiple goals and reacting to changes across scales. The technique is applied to a SC: BW bot called EISBot which was able to achieve a win rate over 60% against the default SC: BW AI (Weber, Mawhorter, et al., 2010).

The EISBot was further extended to improve its strategic decision making. One approach was applying Goal Driven Autonomy (GDA) for strategy selection and execution (Weber, Mateas, & Jhala, 2010). GDA is a conceptual model for enabling autonomous agents to respond to unanticipated failures in plan execution. Agents following this model are able to detect failures in their plans, reason about their goals and generate new plans in response. The EISBot was modified to include lists of expectations for every strategy, which must remain true during the execution of those strategies. If any of these are violated, the discrepancy is detected and the agent selects a new strategy to pursue with a new list of expectations.

The strategies it selects from are enabled via a form of Case Based Planning (CBP) (Weber, Mateas, & Jhala, 2010). Strategies are generated based on a library of human expert replays, which reduces the amount of knowledge engineering required to build the agent. The approach actually differed from an earlier attempt which required defining a goal ontology and annotating the expert replays to be used in a case based planner (Weber & Ontañón, 2010). Instead the case retrieval process is decoupled from the planning process, and cases are indexed, compared and retrieved via a trace algorithm, which is then used to generate new goal states for a planner to solve. The complete EISBot was evaluated against human players on the ICCUP tournament ladder[13], as well as against other StarCraft bots in the AIIDE 2010 competition; achieving an amateur player ranking of D against humans, and a win rate of 78% against other bots (Weber et al., 2011).

Techniques that combine and extend case-based approaches are also common. Cadena and Garrido (2011) combine Case-Based Reasoning with Fuzzy Set theory to play a full game of SC: BW. Fuzzy state descriptions of the game state enabled reasoning over an abstracted and simplified state space. This allowed the bot to achieve approximately 60% win rates against the default AI, using a case base of only one case and when player starting positions were randomized. Palma et al, (2011) advocated the

[13] International Cyber Cup. www.iccup.com

combination of CBP with Behaviour Trees (BT). BTs are a common technique in the game industry used to define agent behaviour as a hierarchy of actions and decisions. When combined with CBP, they serve as a simple way to inject expert knowledge into the CBP process. This was done in order to correct reactivity problems when extracting plans from expert traces, such as invalidated plans not being abandoned unless a low level action fails, or abandoning viable plans when a single low level actions fail (Palma et al., 2011). Although viable in theory, the technique was evaluated only in specific combat scenarios against the default AI, and was not applied to play a full game of SC: BW at the strategic level.

Approaches which focus on specific tasks in the strategic level have also been attempted. For example, Churchill and Buro (2011) applied heuristic search and a simulation of SC to optimize early game 'build orders' (optimized selection of unit and building creation order to balance resource gathering and spending for specific goals). When compared to professional build orders in replays, the technique was able to establish shorter or equivalent time plans to reach specific build orders. Cho et al., (2013) applied data mining and machine learning techniques to large corpuses of expert replays, in order to predict opponent strategies and adapt build orders. Evaluations were performed involving further replay analysis, and the results identified hybrid ensemble classifier approaches for accurate strategy prediction. This work is similar to previous work which analysed replays to build a case based model that is then able to predict player strategies (Hsieh & Sun, 2008). Another approach to strategy prediction is by building probabilistic models without prior knowledge (Dereszynski, Hostetler, & Fern, 2011; Synnaeve & Bessiere, 2011).

2.2.2. Micromanagement Techniques

In Churchill et al (2012), micromanagement is described as a core skill of successful human RTS experts, which often makes the decisive difference in both professional human games and AI competitions. It is a common misconception within the SC: BW game community that AI bots are able to achieve superior micromanagement, on the virtue of being able to perform actions much faster than humans in real time[14] (AI with upwards of thousands of actions per second versus hundreds of actions per minute by humans). In reality, micromanagement in RTS games is a computationally difficult task, where the game tree grows exponentially large with respect to unit numbers. Formal analysis on graph based attrition games, which are movement abstracted versions of RTS micromanagement, have been classified as PSPACE-hard and EXPTIME to compute winning strategies (Furtak & Buro, 2010).

The real-time constraints of RTS games compound the computational difficulty further, where decisions must be made during millisecond time frames. This essentially makes it impossible to compute

[14] Community Q/A with Dave Churchill. http://day9.tv/d/DaveChurchill/great-question/

optimal moves for all but the smallest unit numbers, and requires the use of approximate solutions (Churchill et al., 2012). In the video game industry, this is commonly accomplished via scripted behaviour based on some simple heuristic (e.g. target units based on distance, unit attributes or just simply at random) that is quick to compute but can be easily exploited. Properly designed scripted behaviour can achieve good results, as evidenced by the top three finishing bots of the 2012 StarCraft AI Competition which all use a variety of scripted behaviour for micromanagement (Churchill & Buro, 2013).

Another approach to approximating optimal solutions is via heuristic guided search. In general, the aim behind search based approaches is to reduce the amount of knowledge engineering in scripted approaches. It also enables emergent behaviour, where the AI is able to adapt to situations that authors never knew about and take advantage of scripted strategies, by searching ahead to see many possible future outcomes. Kovarsky and Buro (2005) introduced evaluation functions and compared different heuristic search algorithms to small-scale abstract combat games with real-time constraints. Experiments identified the strength of non-deterministic search methods against traditional Minimax algorithms, such as the Randomized Alpha-Beta Search, which addresses simultaneous move dependencies. Sailer, Buro, and Lanctot (2007) searches for the Nash equilibrium among a set of strategies in an abstracted RTS game. The Nash optimal strategy is found by simulating the sets of strategies against opponent strategies, thus guaranteeing wins against scripted strategies as long as a viable counter strategy exists in its search tree.

Further improvements to search approaches have been identified in Churchill et al., (2012) and Churchill and Buro (2013), which proposes new evaluation functions and search algorithms to deal with durative moves (moves with effects extending multiple time frames such as a weapon cool-down) and larger unit numbers under real-time constraints. Numerous comparative evaluations have also been conducted under simulations of the SC: BW game, validating the strength of the proposed methods against scripted behaviour and alternative search based approaches. In general, search based approaches have employed simulated evaluations for two reasons: full micromanagement in SC: BW is too complex for certain proposed approaches, and some technical limitations to the BWAPI framework reduce the full theoretical search speed of certain techniques. While simulations are becoming better approximates, and search approaches have shown effectiveness against scripted approaches, it is up to future work for these techniques to be implemented and tested against other bots in the full SC:BW game (Churchill & Buro, 2013).

Other approaches to micromanagement in SC: BW include Monte Carlo Planning and Bayesian Modelling approaches. Monte Carlo planning is based on the idea of a stochastic sampling of the solution space through numerous simulations. This was applied to SC:BW micromanagement in Wang, Nguyen, Thawonmas, & Rinaldo (2012), where a set of expert defined plans were evaluated through a Monte Carlo algorithm on a pre-defined simulation. Results showed that the trained AI was able to perform better than the default AI under specifically designed micromanagement scenarios where the enemy has a numerical advantage. However, its performance was worse than a human expert. Synnaeve & Bessière (2011b) modelled unit micromanagement as a combination of a simple finite state machine, and a Bayesian model. The model is centred on a probability equation that maps unit inputs to probability of movement directions, which can be solved given data about the distributions of this mapping (such as through reinforcement learning or mining expert replays). By combining simple attack or retreat FSM actions, with the probability distribution of movement directions (either choosing the highest probability direction, or sampled), the AI achieved great results against the default SC: BW AI and against a popular heuristic guided script.

Approaches inspired by the robotics field have also been applied to micromanagement. Artificial Potential Fields are a technique used to assist robotic manoeuvring, by modelling the topological environment with attractive and repelling fields (Hagelbäck & Johansson, 2009). This was applied to model StarCraft micromanagement, where unit actions are decided by attractive and repulsive forces defined by static and dynamic potential fields (Rathe & Svendsen, 2012). The parameters of a potential field, which defines its attractive and repulsive forces towards different kinds of fields, essentially define a unit's behaviour (e.g. a lower health enemy unit presents high attraction for all close by units to attack). To avoid a difficult and time consuming process of manually defining these parameters, work has been done to fine-tune them using an evolutionary algorithm (Sandberg & Togelius, 2011), and a multi-objective genetic algorithm (Rathe & Svendsen, 2012). Such approaches do not seem to have been evaluated against other AI techniques for comparative results. Potential Flows, an extension to Potential Fields which avoids local minimas, have been applied to the specific task of controlling scout units (Nguyen, 2013). In this task, Potential Flows guide a scouting unit to efficiently avoid obstacles and enemy units, while optimally discovering enemy structures. The measure of success is the number of enemy structures discovered and time of survival, and the work claimed effectiveness under these measures compared to an existing scouting technique and against average human players.

Most relevant to the approach taken in this thesis for SC: BW micromanagement are techniques applying reinforcement learning and Neuroevolution. Shantia et al., (2011) combined reinforcement learning with neural networks to train and control sets of units in SC: BW micromanagement combat. The work was novel in abstracting game environment data using 'vision grids', and to agent learning using the popular online Sarsa reinforcement learning algorithm, with neural-networks to efficiently approximate the state-action value function. The technique was trained and evaluated against the default StarCraft AI in same unit, 3 versus 3 and 6 versus 6 agent combat, with the results of the 3 versus 3 learning bootstrapping the 6 versus 6 evaluations. The results showed a significant winning advantage over the standard StarCraft AI, but required thousands of training rounds before converging to a competitive win rate. The evaluation scenarios are also limited in the type and number of units used, with only one type of range unit presented in relatively small scale combat.

In Wender & Watson (2012), a comparative evaluation of reinforcement learning techniques applied to StarCraft was presented. More specifically, four variants of popular reinforcement techniques were applied to a specific combat scenario, involving a range unit with high mobility against numerous short ranged enemy units. The idea behind the scenario was to specifically induce the learning of a micromanagement technique called 'kiting', where the range unit must constantly cycle between attacking and running away in order to win against greater enemy numbers. The results identified comparatively stronger and weaker techniques within the ones evaluated, and showed a high win rate against the default StarCraft AI, within a thousand training rounds. However, the results were evaluated against a very specific combat scenario, and the author acknowledges the solution developed is only the first part of a larger RL based StarCraft agent.

In Gabriel et al., (2012) the real time variant of NEAT was used to develop a micromanagement agent in StarCraft. Each unit is controlled by its own neural-network, which specifies whether the unit should attack or retreat, and in which direction it should move. The basic architecture is then evolved in real time by the rtNEAT algorithm over generations of 12 versus 12 agent combat. The evaluation involved four different combat scenarios with variations of range and melee unit matchups; first against the default StarCraft AI, and then against two other SC: BW AI systems. The results showed a significant win advantage against the default AI in all unit match up variations, and a weaker advantage against the other AI systems, within 300 training rounds. The authors also claimed the rtNEAT algorithm allowed significant real-time improvement over the course of a single training round, which suggested fast adaptive-ness in game strategy.

One limitation of the evaluation is the use of a custom SC: BW map which replenishes unit numbers throughout a single training round, with up to 100 unit reserves. This is an unrealistic depiction of real SC combat as units do not replenish immediately after a unit is killed off. This also skews the training results since a single generation is actually multiple 12 versus 12 rounds with 100 unit replacements. The claim of real-time improvement is weakened by the fact that the real-time fitness increase occurs over unit fighting time (0.02 fitness over 3000 game ticks), which is only a sub portion of a full SC: BW match and does not occur continuously. However, the work showcased the potential of applying NEAT based algorithms to SC: BW micro-management, which this thesis further explores.

2.2.3. Reinforcement Learning, Neuroevolution and NEAT

Reinforcement Learning (RL) is a machine learning method that models agents taking actions in an environment to maximize a defined reward (Sutton & Barto, 1998). It differs from standard supervised learning methods in that examples of correct and incorrect actions are not explicitly specified. Instead, the agent actively searches the solution space for an optimized policy by trying actions and receiving an immediate reward. Often, for more challenging cases, the agent must balance exploration and exploitation strategies and deal with problems of delayed reward. When a complete and accurate model of the problem exists, the optimal policy can be solved via Dynamic Programming methods. For problems with large state spaces and where an accurate model is not possible, statistical sampling methods such as Monte Carlo and Temporal Difference learning are effective (Sutton & Barto, 1998). These conventional reinforcement learning methods are based on approximating the optimal value functions, which give the expected optimal reward for each action in each state of the problem space.

In practice, conventional RL methods do not scale well for problems with large state-space and partial observability (Gomez & Miikkulainen, 2003). An alternative approach is Neuroevolution (NE), which combines the ability of neural networks in approximating non-linear functions, with the training strategy of evolutionary algorithms (Yao, 1999). NE searches through the space of neural networks according to the principles of natural selection. Instead of adapting a single agent, NE involves populations of candidate solutions, and a process of selecting and reproducing these solutions based on a quantitative fitness measure. It has been effectively applied in learning behaviour policies for reinforcement learning problems, and can find solutions faster than classic reinforcement learning methods on standard benchmarks (Gomez & Miikkulainen, 2003; Stanley, 2004).

Traditionally, NE worked on a pre-defined neural network topology, and the evolution searches over the space of connection weights between the network nodes. A class of NE algorithms, Topology and Weight Evolving Artificial Neural Networks (TWEANNs), attempts to also evolve the topology of the network, by adding or deleting network structure such as nodes and edge connections (Yao, 1999). By searching through the space of network topologies, it is possible to improve on training speed and the accuracy of solutions more so than searching weights alone. Furthermore, it reduces the uncertainty and effort of deciding on the network topology, for example in deciding the number of hidden nodes for any particular NE problem via trial-and-error processes (Stanley & Miikkulainen, 2002b).

However, TWEANNs techniques face numerous challenges (Stanley, 2004). Firstly, cross-over of network structure in evolution is non-trivial, and the result may be redundant or infeasible structures. Secondly, the adding of new network structure may impair the immediate fitness of the network, but should be retained to allow room for new solutions to converge. Thirdly, the setup of the initial population affects the evolution process: random topologies may yield infeasible or complicated starting topologies, resulting in highly complex and inefficient solutions. Stanley & Miikkulainen (2002a) developed the NEAT algorithm to tackle these issues. NEAT uses historical marking of genes and a compatibility operator to track topology evolution and gene crossover. The compatibility operator is used to define species in the population of networks, preventing incompatible genomes from crossing over, and together with fitness sharing, helps to protect new innovations from being removed prematurely. NEAT also follows the principle of complexification, where the starting typology is minimal, and the evolutionary search is expanded gradually in order to reach solutions with minimal structure and time taken. The workings of NEAT are further discussed in Chapter 3.

The classic NEAT algorithm was also expanded to a real-time variant (rtNEAT), where evolution occurs over a real time scenario (Stanley, 2005). The main difference in the new variant is that instead of evolution over an entire population at a given point in time, new neural networks are created and introduced to the population in gradual and continuous real-time. When applied to a game scenario, agents in the game show a gradual behaviour change, instead of an abrupt change to all agents that may break gameplay immersion. The rtNEAT algorithm was demonstrated in a game called NERO, where agents are evolved and adapted in real time to tackle changing objectives (Stanley, Bryant, & Miikkulainen, 2005). Players could train squads of units, each controlled by a neural network evolved through rtNEAT, and battle other player trained squads.

Regular NEAT has been successfully applied to RTS games (Jang, Yoon, & Cho, 2009), where neural networks are evolved to become controllers for AI characters. Multiple network controllers are combined in an ensemble process in order to choose the right action for an agent to take. In Olesen et al., (2008) both NEAT and rtNEAT were used to automatically balance the challenge of the AI in an RTS game called 'Globulation 2'. A challenge rating metric was generated by analysing aspects of the game that affected player performance. The fitness function, which guided the evolution of the behaviour of the AI, was based on the closeness of the AI challenge rating and the human player's rating. In this way, the AI was continuously evolved to play at the same challenge level as the human player, thus creating a better player experience where the AI was not too challenging and not too easy.

Stanley (2005) listed several challenges posed to traditional RL techniques when applied to real-time video games. For example, video games have large state and action spaces, which is a known challenge for RL learning. This is compounded in real-time games, where the value of every possible action for every agent and on every game tick, must be checked and updated for an accurate approximation of the value function. In contrast, NE and NEAT work well in high dimensional states, and evolved agents do not check the value of all actions, but only a single output per game tick. Another example is that traditional RL techniques do not support the diversification of solutions, while NEAT specifically supports diversification through speciation. This is important for agents in game environments to display heterogeneous behaviour to realistically simulate populations and make gameplay more interesting. Other examples on the list include agents having consistent individual behaviour (violated by RL when randomly sampling new actions in exploitation, while in NEAT individuals have consistent actions throughout its lifetime) and fast adaptation versus sophisticated behaviour trade-off (NEAT is flexible in generating simple solutions quickly that can grow more complicated later, while RL must change its representational model to allow the same).

2.2.4. Summary of Related Work

In the previous sections, work closely related to the thesis objectives were discussed. This began in section 2.2.1 where related AI research tackling the macromanagement layer of the SC: BW game domain was examined. The large action and state space complexity of macromanagement motivates the use of planning techniques to divide goals of varying hierarchy. The challenge here is synchronization between hierarchical goals and to be able to react to failed plans. New plans can be generated by building cases from expert human game plays, querying probabilistic models, or injecting expert knowledge. Approaches which look at specific parts of macromanagement include: build order optimization through heuristic search, opponent modelling and strategic prediction via data mining human player replays. Although macromanagement is not the focus of this thesis, many of the techniques here are related to micromanagement techniques.

In section 2.2.2 I discussed the importance of micromanagement in RTS games, and its computational complexity attributes. Existing approaches include numerous heuristic search based techniques, which often work on an abstracted version of SC: BW. Although micromanagement has a smaller state and action complexity space, it is still large enough to make heuristic search difficult in the non-abstracted game. Other approaches include Monte Carlo planning and Bayesian modelling, which are statistical techniques for approximating the right action, when given enough example data. Techniques inspired by the robotics field have also been applied to specific unit control in micromanagement, such as potential flows for controlling scouting units. The approaches to micromanagement that are most related to this thesis are RL and NE ones, of which only three examples could be found in the literature, and all were discussed. In general, the common goal of AI research in this area is to develop techniques that can approximate the optimal actions and generalize over many scenarios in a very large state and action space. Many of the benchmarks used to evaluate the approaches are scripted unit behaviours based on simple heuristics that are known to be effective.

I began a discussion of RL and NE in section 2.2.3, highlighting their similarities and differences. In particular, I discussed the NEAT method and its advantages over traditional RL. This was followed by examining examples of NEAT specifically applied to video games, and concluded with the advantages of the rtNEAT algorithm over traditional RL for real-time video games in general. In the next chapter, I discuss the primary framework used in this thesis, the NeuroEvolution of Augmenting Topologies.

Chapter 3

NeuroEvolution of Augmenting Topologies

In section 2.2.3 I discussed related work that motivated the use of NE algorithms for reinforcement learning problems, and the successful application of the NEAT framework for game scenarios. In this chapter I discuss the workings of the NeuroEvolution of Augmenting Topologies (NEAT) framework. First, I review some fundamentals in evolutionary computation and neural networks that are necessary components of Neuroevolution. Next I analyse the challenges with topology evolving Neuroevolution and begin to discuss different aspects of NEAT that address these problems. This chapter is adapted from the PhD thesis work of Kenneth O. Stanley (Stanley, 2004), and examines both the theoretical motivations and the practical algorithms of NEAT.

3.1. Genetic Algorithms and Artificial Neural Networks

NEAT is influenced by Genetic Algorithms (GA), a class of search algorithms inspired by biological evolution (Stanley, 2004). GA's search through a parameter space (binary, discrete, real or other encoding scheme) for parameters that optimize some performance goal. The search procedure is based on biological natural selection where a population of solutions, each representing a point in the search space, are continuously evaluated. A *fitness* heuristic is used to guide this evaluation, which defines solutions that are more suitable than others in reaching the performance goal. High fitness solutions are then selected for reproduction, a process that generates new solutions containing random changes, but which are still similar to its parent solutions. By replacing low fitness solutions with the offspring of high fitness solutions, the population begins to converge towards optimal parameters (Figure 1). GA's are particularly useful when applied to sparse domain knowledge problems, since it assumes no a priori of the problem domain or solution space, as long as there exists a fitness function that can differentiate low and high performing solutions.

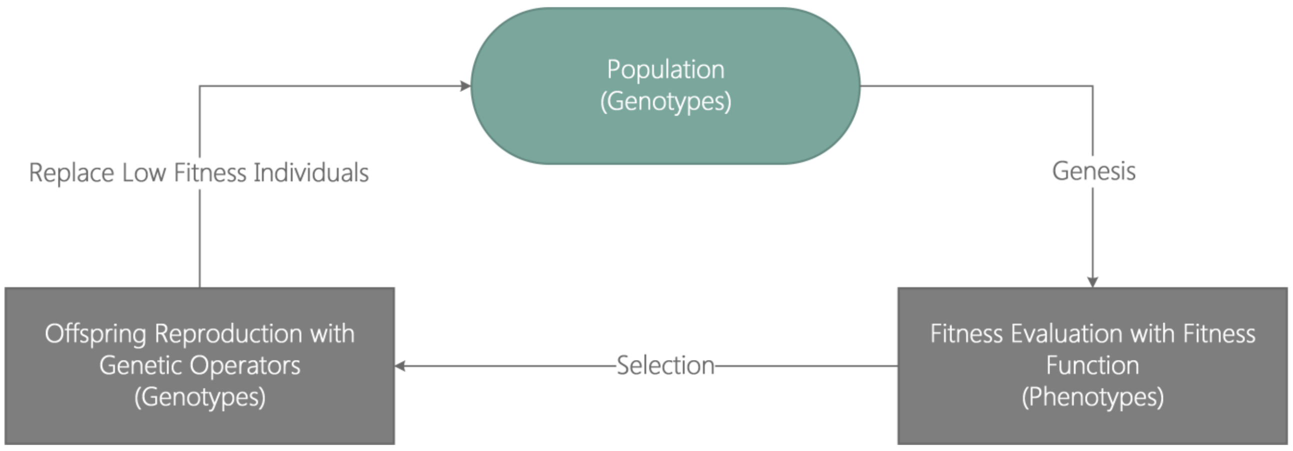

Population
(Genotypes)
Genesis
Fitness Evaluation with Fitness Function
(Phenotypes)
Selection
Offspring Reproduction with Genetic Operators
(Genotypes)
Replace Low Fitness Individuals

Figure 1: A Summary of the Genetic Algorithm Process.

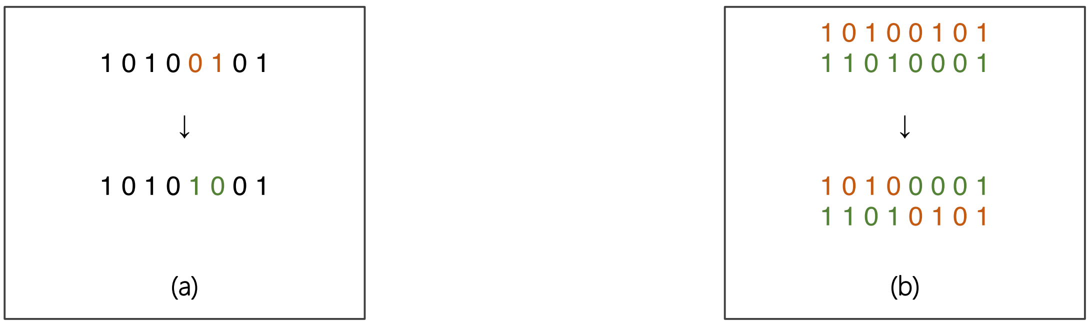


Figure 2: Examples of genetic operators on a binary string encoding. Bit string mutation (a) and string crossover to form two new offspring (b).

To facilitate reproduction, the parameters are usually encoded as a string called the *genotype*, which allows efficient genetic operations to be applied, such as mutations on parts of the string or exchanging parts between strings (crossover operation) (Figure 2). A genotype encoding typically goes through a genesis procedure into a *phenotype*, a format that can be evaluated for fitness. In NEAT, the GA model is applied to an encoding of ANN weight and topologies, and searches for the optimal network for a given task (Stanley, 2004). NEAT differs from standard GAs in that the dimensions of the parameter space can be modified during search (adding and subtracting nodes in the network), while GAs typically operate on a search space with a fixed number of dimensions (a fixed number of parameters in encoding). This gives the advantage of being able to generate smaller or larger solutions than fixed sized dimensional search, and is crucial for optimizing search speed (section 3.2).

ANNs are the phenotypes at the core of NEAT's evolutionary optimization. They are computational processing structures inspired by the biological nervous systems of animals, such as the brain. The basic structure consists of a network of nodes, with an input layer which models sensors that receive data from the external world, and an output layer which contains information resulting from processing the input data through the network. Between these two layers are hidden nodes that expand the variability of the processing potential of the network. Nodes are connected via adjustable weighted connections, and exchange data when they are activated. In theory, ANNs are able to approximate any continuous function (Hornik, Stinchcombe, & White, 1989), with some specific architectures proven to have the same power as a Universal Turing Machine (Siegelmann & Sontag, 1991).

Figure 3 is an example of a feed forward ANN, where the input layer sets off a forward passing activation over weighted connections ending at the output layer. In NEAT the structure of hidden nodes are evolved over time, which means they will not necessarily be in strict layers or be fully connected. It is also possible to evolve recurrent networks, which contain feedback connections that are useful for certain problems.

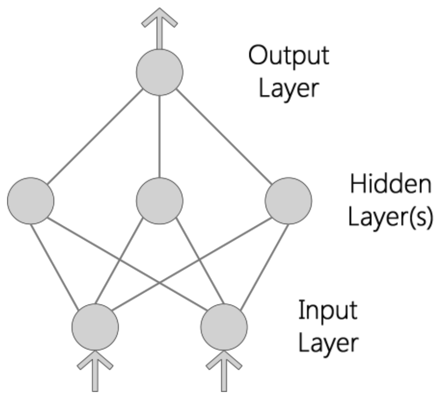


Figure 3: A simple feed forward Artificial Neural Network with one hidden layer.

A node in an ANN computes a weighted sum of its inputs and passes it through an activation function, which outputs a value between 0 and 1 onto other nodes. That is, for every node $\boldsymbol{i}$, its output $\boldsymbol{y_i}$ for a given input vector $\boldsymbol{x}$ of $\boldsymbol{n+1}$ size is:

$$y_i = a\left(\sum_{j=0}^{n} w_{ji}x_j\right) \tag{1}$$

where w_{ji} is the weight of the connection from node j to i and a is the activation function. Typically, the input j_0 is a bias with the value of $+1$, and weight w_0 denotes the bias weight, leaving n actual inputs to each node. The activation function is usually a non-linear sigmoid function such as the logistic function:

$$a(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

which outputs a smooth S shaped curve bounded between 0 and 1. It has advantages over linear and binary functions, such as having a smooth and continuous derivative for making learning via gradient descent methods easier, and can tackle problems that are not linearly separable (e.g. the Xor function).

NEAT does not train ANN with gradient descent methods like back-propagation, but instead uses Neuroevolution, a process similar to GAs described previously. This has the benefit of not requiring output targets (similar to reinforcement learning), and is better at avoiding local minimas (because evolution simultaneous evaluates multiple solution points at once) (Stanley, 2004). In the next section, I describe a class of Neuroevolution that NEAT belongs to, called Topology and Weight Evolving ANNs.

3.2. Topology and Weight Evolving Artificial Neural Networks

Topology and Weight Evolving Artificial Neural Networks (TWEANNs) are a class of Neuroevolution methods which evolve both the connection weights and the topology of the ANNs (Yao, 1999). This has a number of advantages over fixed-topology Neuroevolution. First, it reduces uncertainty and human effort in specifying network topologies manually, which is often done via trial and error processes. Secondly, topology evolution can increase the efficiency of solutions by keeping networks as small as possible. Thirdly, it has the potential to improve the training speed and accuracy of solutions; by finding the solution in a network with the smallest number of connections, or by finding complicated topologies to allow for a more accurate solution.

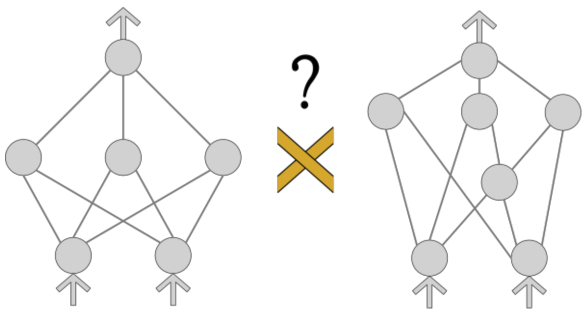


Figure 4: Deciding how to combine ANN topology during crossover to ensure viable offspring was a challenge for TWEANNs. This process is partial to the design of the genotype encoding scheme.

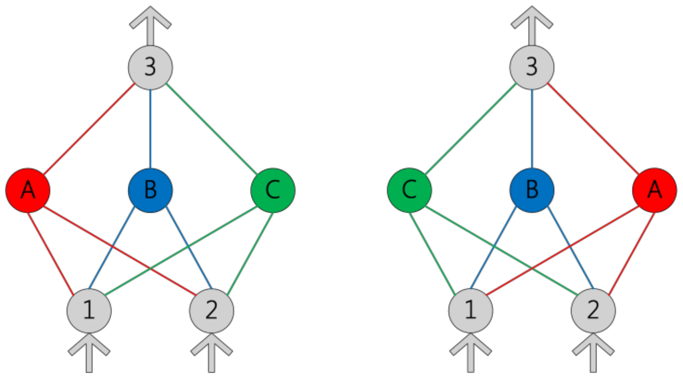


Figure 5: A simple example of the Competing Conventions problem. Two ANNs with the same functional structure but different genotype representation (permutation on the order of hidden nodes). Crossing $[\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}]$ with $[\boldsymbol{C}, \boldsymbol{B}, \boldsymbol{A}]$ can result in$[\boldsymbol{C}, \boldsymbol{B}, \boldsymbol{C}]$, a solution that has lost a third of the information from both parents. The higher the number of hidden nodes, the higher the number of possible competing conventions, since with $\boldsymbol{n}$ hidden nodes there are $\boldsymbol{n}!$ permutations.

However, TWEAANs face a number of unique challenges compared to fixed-topology Neuroevolution. One of these is the challenge of encoding topology along with weights in a genotype that allows for comparisons and crossovers. Stanley (2004) described a number of existing TWEANN encoding techniques prior to NEAT, labelled between direct (the topology is explicitly specified in genotype encoding) or indirect (the genotype only contains information about how to derive the topology, such as rules during the translation to a phenotype) methods. In either case, many of them required an explicit bounding size of topology growth, which puts a restriction on the space of possible solutions and brings back an element of human responsibility for deciding this bound. The existing encoding schemes also failed to support the evaluation of topology substructures during crossover, which results in many non-viable offspring topologies (Figure 4). This problem is compounded by the 'Competing Conventions' problem, where cross overs between solutions that are the same but expressed differently in a genotype, are very likely to produce less viable offspring (Figure 1). Attempted solutions to this problem either require computationally expensive topology analysis before crossover, constraints on topologies, or by removing crossovers entirely. NEAT addresses this problem with a direct encoding scheme based on *historical markers* (section 3.3)

A second challenge occurs when considering the result of topology crossover or mutation in TWEANNs. Often the addition of a structural component (such as a new node, or a new connection between nodes) introduces a nonlinearity to the existing network solution. It is unlikely that a new random addition comes already optimized and immediately expresses a useful function within an ANN. If fitness evaluation is allowed to run unchanged, these new additions will likely be removed from the population before they have enough time to optimize. The act of protecting solutions that are temporarily disadvantaged from new structures is known as the principle of Protection of Innovation (Stanley, 2004). NEAT accomplishes this by adapting two concepts from GAs known as *speciation* and *fitness sharing* (Section 3.4).

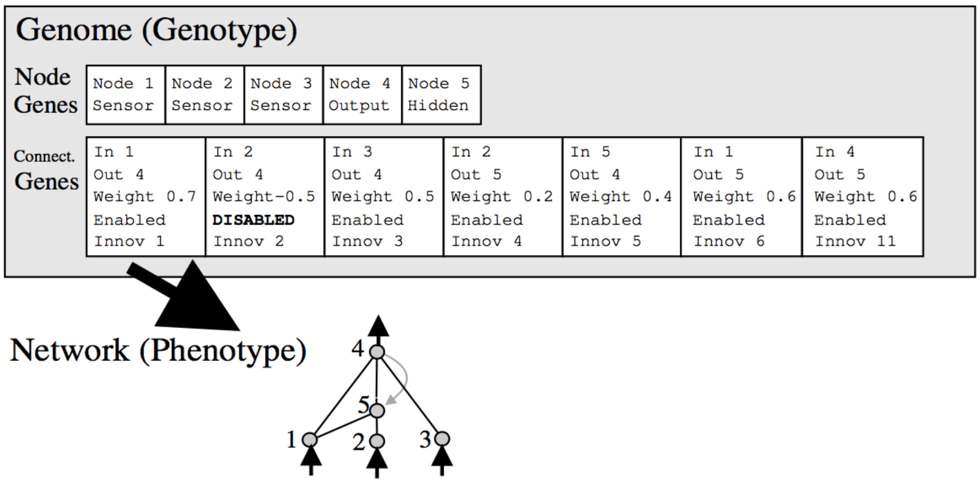
A third challenge arises when considering the need to evolve minimally complex topologies. TWEANN techniques prior to NEAT often started with an initial population of random topologies, so as to ensure diverse solutions. However, this often produces infeasible networks and variable starting topology sizes. The latter problem is especially dire, since it cannot guarantee that the search occurs over minimally complex topologies. Such a property was proven to be desirable by Stanley (2004), as it ensures that search time is reduced when searching and optimizing the lowest number of dimensions. Another problem with randomized topology sizes is that there are drastic differences between different dimensional spaces, where the increase of a single connection can radically improve or worsen the fitness landscape being searched. Through these observations, Stanley formulated the principle of Topology Innovation for TWEANNs, which states that a population should start with a minimal structure and grow incrementally, to improve the likelihood of finding a global optimum. NEAT directly follows this principle, by starting with a population of minimally structured ANNs and adds structures after optimizing the current dimension of weights.

The next sections begin to review the NEAT method in detail. Each component was designed to address the three principle problems discussed above. First, the genotype encoding was designed to allow efficient and suitable match-ups of topologies for crossover. Incidentally, such an encoding is also useful in defining similarity for speciation and fitness sharing. These are useful for the protection of new innovations. Next, the full NEAT process is summarized, and a real-time variant of NEAT is introduced.

3.3. Genetic Encoding and Operations in NEAT

NEAT employs a direct encoding, where the topology of a network is explicitly specified in its genotype. Each *genome* (an individual network's genotype) consists of *node genes* and *connection genes*. Node genes contain information about individual nodes of a network, such as whether they are input, output or hidden nodes. Connection genes specify the connections between nodes, and contains the weight of a connection, its input and output node identifiers, an activation flag, and a number known as the *innovation number* (Figure 6).

Connection weight mutations in NEAT occur with a fixed probability for each connection, and mutate by adding a floating point number from a distribution of positive and negative values. Structural mutations also occur via fixed probabilities, but differ between the types of mutations. There are two types of structural mutations: new connection and new node mutations. For a new connection mutation, a new connection gene is added that connects two previously unconnected nodes in the genome. In a new node mutation, an existing connection is split into two, and a new node is added between these two connections (Figure 7).



Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Output	Node 5 Hidden

In 1	In 2	In 3	In 2	In 5	In 1	In 4
Out 4	Out 4	Out 4	Out 5	Out 4	Out 5	Out 5
Weight 0.7	Weight-0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6	Weight 0.6
Enabled	**DISABLED**	Enabled	Enabled	Enabled	Enabled	Enabled
Innov 1	Innov 2	Innov 3	Innov 4	Innov 5	Innov 6	Innov 11

Figure 6: An example of an encoded genome directly mapping to a network phenotype in NEAT. Each node gene specifies a node and its type. Each connection gene specifies a weighted connection between two nodes, and can be disabled such that it is not expressed in the phenotype. Taken from Stanley, (2004).

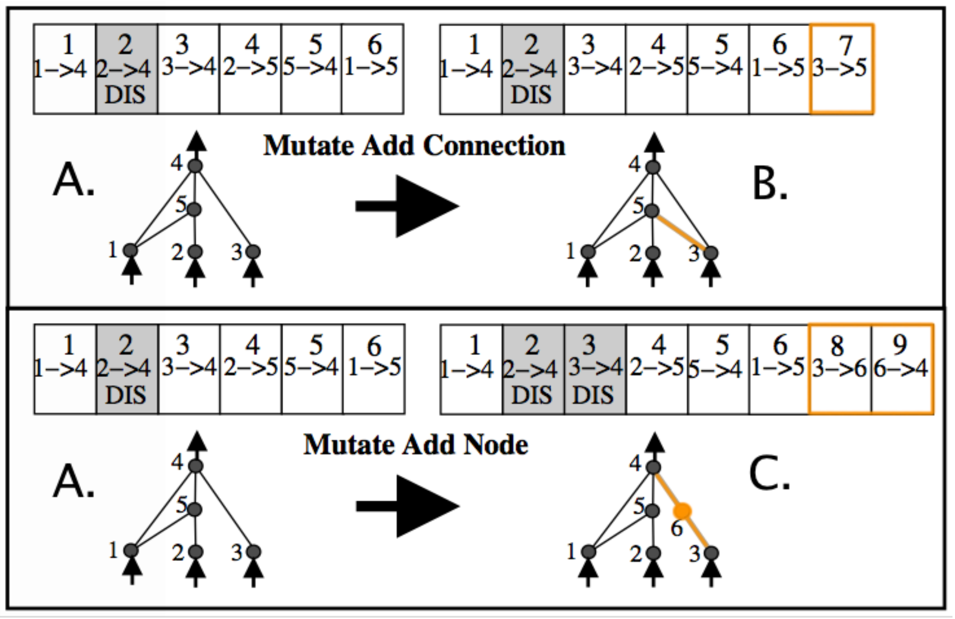


Figure 7: Example of NEAT structural mutations. The blocks represent the connection genes ordered by their innovation numbers. Adding a connection introduces a new connection gene. Adding a node disables an existing connection gene, and adds a new node and two connection genes. New connection genes are given unique chronological innovation numbers. B is the result of a first mutation on A and incurs a gene with innovation number 7, while C is a separate result from a second mutation on A, incurring genes 8 and 9. Adapted from Stanley, (2004).

Structural innovations are added to a network when mutations are introduced to a genotype. The genome length is unbounded so as to avoid arbitrary bounding decisions and to support incremental growth from minimally complex structures. Each new innovation in a population is assigned a unique innovation number, such that there is a total ordering on the chronological inception of all innovations (Figure 7). This number serves as a historical marker for genes and can be used to determine similarity between any two genomes in the population. Since the initial population in NEAT start with the same minimally complex structure, genes with the same historical origin represent the same structure. This is done with very little computation, as it only requires the increment of a global innovation counter. With an efficient measure of similarity between networks, NEAT avoids the Competing Conventions problem, and ensures that offspring from topology crossover preserves the overlapping historical innovations of both parents.

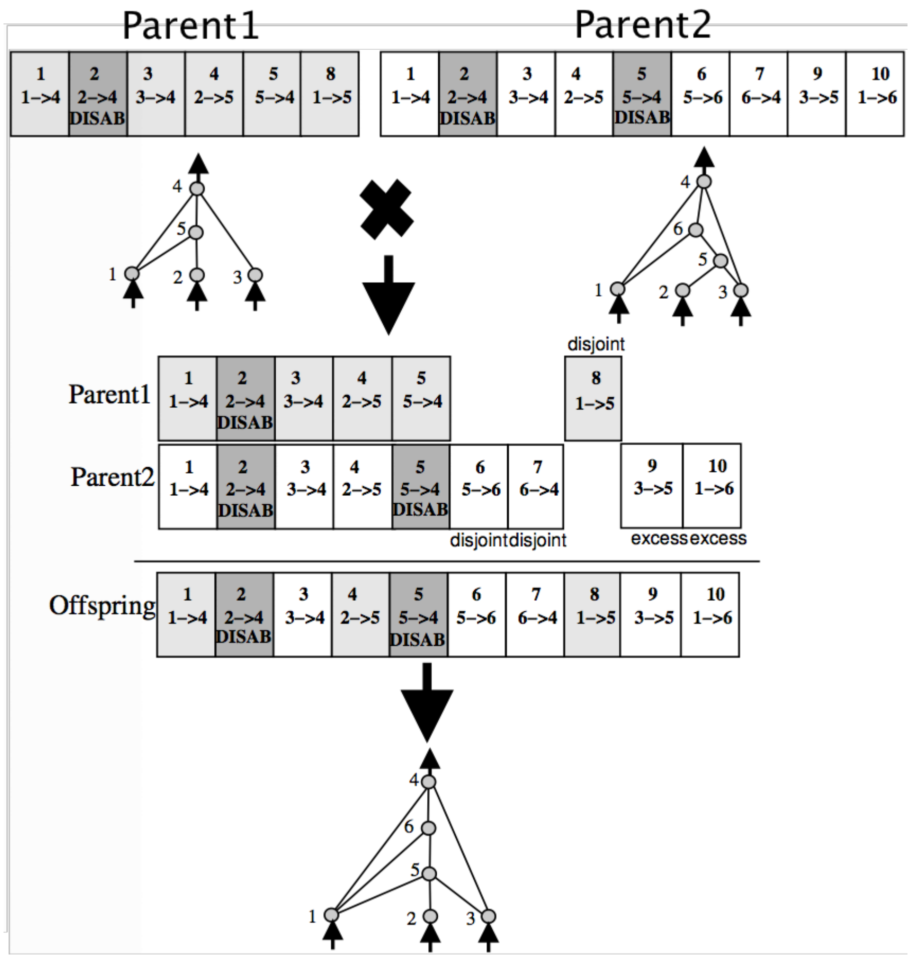


Figure 8: An example of crossover reproduction in NEAT using historical markers. Without analysing the topology of Parent1 or Parent2, the structural similarities and differences can be efficiently found by comparing the innovation number of genes. An offspring is produced by taking the overlapping parts of both parents (genes 1 to 5) and inheriting the disjoint and excess genes from the more fit parent, or inheriting random genes if fitness is equal (genes 6-10). The weights of the overlapping parts are either randomly chosen or averaged between the two parents. Taken from Stanley, (2004).

Figure 8 depicts an example of crossover reproduction in NEAT. By using the innovation numbers, structural similarity can be efficiently compared between any two networks. When comparing two parent genomes, genes which appear in one genome and not the other are known as *disjoint genes*. Genes which appear in one parent later in evolution than any genes in the other parent are known as *excess genes*. Overlapping genes between parents are always present in a resulting offspring, while disjoint and excess genes are inherited based on the fitness of each parent. Disabled genes can also be re-enabled in an offspring, which allows old connections to be expressed in a network once again.

This measure of similarity between two genomes is also useful for enabling speciation and fitness sharing. As described previously, these two concepts are important to ensure the principle of Protection of Innovation. In the next section I describe what these are and how they are implemented in NEAT.

3.4. Speciation and Fitness Sharing

Speciation is a technique in GAs used to optimize functions with multiple optimas. For these problems, the population as a whole is used to make decisions, where each solution should represent a viable optima that is different from other solutions (optimizing for different parameters). The idea behind speciation in this context is to force the convergence of solutions to be diverse throughout the population. This can also be useful for preventing premature convergence in single solution problems, where the entire population could get stuck converging towards a local optima. In TWEANNs, speciation is useful for allowing different topologies to develop simultaneously. In NEAT it is essential for the protection of new innovations and to prevent crossovers for incompatible structures (Stanley, 2004).

Fitness sharing is a method for maintaining species in GA population. It is inspired by biological populations, where species adapted to a certain niche are forced to share the payoff for that niche. NEAT uses a form of explicit fitness sharing, where similar individuals share their payoffs (i.e. networks with similar topologies share their evaluated fitness). This encourages solutions to diversify for fitness that does not have to be shared, while protecting new innovations by sharing fitness from parent species.

Central to these concepts is the idea of similarity between genomes. In NEAT this is measured using the historical markings of each genome in the population. More specifically, the distance between two genomes is modelled as a function of the differences in connection weights and genes. Let D be such a distance between any two genomes in a population. It is defined as a linear combination of the number of excess genes EG, disjoint genes DG and the average weight difference of matching genes $\overline{W}$:

$$D = \frac{c_1 EG}{N} + \frac{c_2 DG}{N} + c_3 \times \overline{W} \tag{3}$$

where c_1, c_2, and c_3 are adjustible weighting coefficients, and N is the number of genes in the larger genome which normalizes those parameters for genome size.

The initial population in NEAT begin with a single species. Each species in the population has a member of the species chosen as a representative. Newly reproduced genomes are compared to each species' representative using the distance metric defined above, and when the distance is less than a compatibility threshold (D_t), it is placed into the species being compared to. If the distance is higher than the threshold for all existing species, a new species is generated with the new genome. In NEAT, D_t is dynamically adjusted in order to match a target number of species (i.e. lowered to increase the number of species and raised to decrease it).

Fitness sharing within species takes place after raw fitness is evaluated for each genome. The adjusted fitness F'_i of an individual i in the population is divided amongst the population based on the distance metric D:

$$F'_i = \frac{F_i}{\sum_{j=1}^{n} sh(D(i,j))} \tag{4}$$

where F_i is the individual's raw fitness, n is the size of the population, $D(i,j)$ is the distance between i and another individual j in the population, and sh is the sharing function. The sharing function returns 0 when $D(i,j)$ is above the threshold D_t and 1 otherwise, such that $\sum_{j=1}^{n} sh(D(i,j))$ returns the number of individuals in the population in the same species as i.

Fitness sharing directly affects the reproduction process, where the number of offspring produced from each species in the population is in proportion to the adjusted fitness of its members. The number of offspring N_s produced by a species s during reproduction is defined by:

$$N_s = \frac{\bar{F}_s}{\bar{F}_{total}} |P| \tag{5}$$

where $\bar{F}_s$ is the average adjusted fitness of members in species s, $\bar{F}_{total}$ is the total of all species adjusted fitness averages, and $|P|$ is the size of the population. The choice of parents within each species is chosen randomly amongst its top performers, and the lowest performing individuals of each species is eliminated. The overall effect of speciation and fitness sharing on the population is the divergence of solutions between species, and the protection of innovation amongst species.

In the next section, I summarize the entire NEAT evolutionary process involving the core components discussed so far. The process is later modified to apply to a real time domain in the rtNEAT variant.

3.5. The Generational NEAT Process

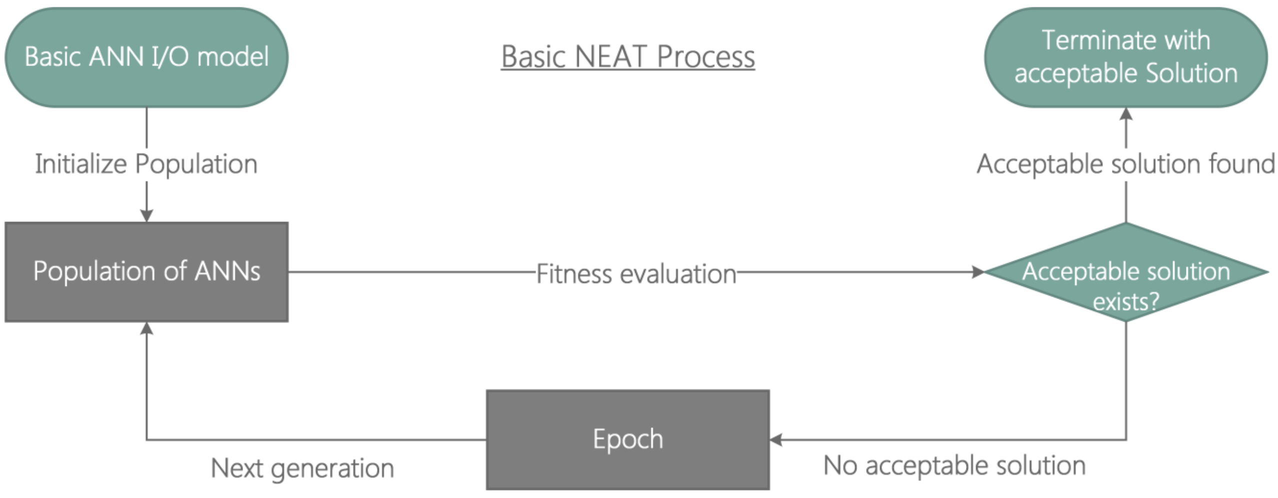


Figure 9: A model of the basic NEAT process for reinforcement learning problems. The initial population of ANNs share a common topology but with randomized weights. On each generation cycle, the population is evaluated via a fitness function. Until there is an acceptable solution, NEAT advances to a new generation by going through an Epoch process.

Figure 9 depicts the overall model of NEAT operating to solve a RL problem. An initial ANN topology design based on a model of the RL problem is used to seed an initial population with matching topology and randomized weights. For example, in a robotic control problem analogy (such as in the task of navigating a maze), the ANN design could incorporate each environmental sensor as input nodes, and output nodes as parameters specifying the robots actions to take on the environment. In each generation, the population is evaluated using a defined fitness function, which identifies how well performing each individual is at the given RL task. In the robot control analogy, evaluations may be a simulated test run of the robot using each individual ANN as a controller, and the fitness may be a function of errors and progress made through the maze. The NEAT process terminates when an individual is found to pass some defined performance criteria (such as successfully completing a maze). Otherwise, the generation is advanced through an *epoch* process, which modifies the population via reproduction and replacement.

The fitness function and acceptance criteria are problem domain specific, and can be adjusted to fit multiple objectives.

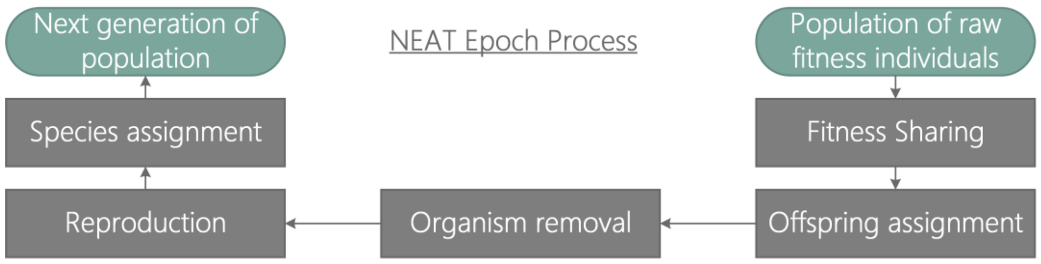


Figure 10: A model of the epoch process in more detail. The fitness sharing, offspring assignment and species assignment mechanisms are as those described previously in section 3.4. Reproduction mechanisms are those described in section 3.3.

Figure 10 depicts a model of the epoch process in more detail. After the population is evaluated via the fitness function, individual fitness is adjusted via fitness sharing. The number of offspring reproduced by each species is assigned based on each species average adjusted fitness. The worst performing individuals in each species are removed to make room for new offspring (with adjustments that prefer to remove more from species that have not improved in a while). Offspring are created via mutation and crossover operations, then assigned to species based on the distance metric. A new generation of the population is then ready for re-evaluation.

With a few changes to the basic NEAT model, it can be applied to real-time problem domains. In the next section, rtNEAT is introduced as a viable version of NEAT that is specifically tailored for real-time reinforcement learning and games.

3.6. Real-Time NEAT

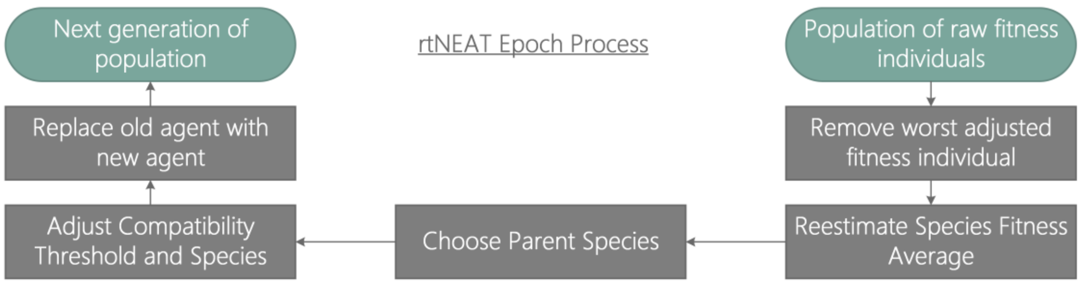

rtNEAT Epoch Process
Population of raw fitness individuals
Remove worst adjusted fitness individual
Reestimate Species Fitness Average
Choose Parent Species
Adjust Compatibility Threshold and Species
Replace old agent with new agent
Next generation of population

Figure 11: A model of the rtNEAT epoch process. It is a modified version of the generational NEAT epoch process, and occurs continuously in real-time at a specified interval. It attempts to simulate generational NEAT in real-time by removing and reproducing a single individual per run.

Real-time NEAT is a version of NEAT tailored for problem domains such as in video game environments, where multiple agents in the population play at the same time. Unlike generational NEAT, rtNEAT supports real-time fitness evaluation and incremental evolution that is important for believable population AI (Stanley, 2005). In order to do so, rtNEAT must encapsulate the fundamental NEAT components in a real-time incremental process.

Figure 11 summarizes the rtNEAT epoch process. Instead of modifying an entire population per iteration, rtNEAT removes and reproduces a single individual per loop. First, the worst performing adjusted fitness individual is found and removed from the population. Fitness adjustment is the same as in generational NEAT, through the use of speciation and fitness sharing to encourage diversification of solutions and to protect innovation. There is an added condition to this process, in the form of a specified organism age threshold, which protects individuals that may not have played long enough for an accurate fitness assessment. This contrasts with generational NEAT which does not have a concept of age since networks are generally evaluated for the same amount of time.

After removing an individual from some species, the species fitness averages must be recalculated. This ensures that the next step of choosing a parent species is accurate. Recall in section 3.4 that the number of offspring N_s reproduced by a species s is determined by $N_s = \frac{\bar{F}_s}{\bar{F}_{total}}|P|$, where $\bar{F}_s$ is the species fitness average of s, $\bar{F}_{total}$ is the total of all species fitness averages, and $|P|$ is the size of the population. This is approximated in rtNEAT for determining which species reproduces an offspring, by ensuring that over the long run, the number of offspring for each species is proportional to N_s. The probability P of a species s is chosen for reproduction is given as:

$$P(s) = \frac{\bar{F}_s}{\bar{F}_{total}} \tag{6}$$

where $\bar{F}_s$ and $\bar{F}_{total}$ are the same as in generational NEAT. The choice of parents within a chosen species are same as in generational NEAT (randomly chosen amongst top performers).

The next step is to readjust the species and compatibility thresholds. In generational NEAT, a number of new offspring are assigned to species based on the compatibility threshold and the distance metric on every generation. In rtNEAT the species reassignment must be explicit to ensure all organisms in the population are in the correct species, since only a single individual is replaced at a time. Once all the above is accomplished, the reproduced organism replaces the removed organism. This is problem domain specific, and may involve additional logic to rewire an agent's controller to use the reproduced neural network.

The last step remains of determining the time interval between replacement, or how often to run the rtNEAT epoch process within the real-time environment. If this occurs too quickly, then the population may not be accurately evaluated and new innovations may be needlessly thrown away. On the other hand, evolution must occur quick enough for agent learning to satisfy some real-time constraints, such as to continuously meet the challenge level of a human player (Olesen et al., 2008). A law of eligibility is formed by Stanley (2005) that states the number of game *ticks* $\boldsymbol{n}$ between replacement. This is based on the fraction of the population ineligible for replacement I, the minimum time for individuals to be alive m, and the population size $|P|$:

$$n = \frac{m}{|P| \times I} \tag{7}$$

where m and I are user defined values.

3.7. Summary of NEAT

In the above sections, the NEAT framework was discussed in detail. At the beginning, its foundations in Genetic Algorithms (GA) and Artificial Neural Networks (ANN) was explained. GAs are biologically inspired search algorithms which work well on sparse domain knowledge problems. ANNs are biologically inspired computational processing structures that can approximate any continuous function. NEAT is essentially a Neuroevolution (NE) algorithm, which are GAs that specifically search over an encoded space of ANNs.

Following on, Topology and Weight Evolving ANNs (TWEANNs) are introduced as the class of NE that allows topologies to be evolved as well as weights. TWEANNs have potential in performing better than static topology NE, as well as traditional reinforcement learning techniques on benchmark control tasks. NEAT was created by Stanley (2004) to address many of the challenges inherent in existing TWEANN techniques at the time. Firstly, NEAT avoids the competing conventions problem while also avoiding constraints on topology by having a genetic encoding that allows efficient topology comparison during topology crossovers. Secondly, NEAT uses speciation and fitness sharing to protect new innovations and to encourage diversification of solutions. Thirdly, NEAT is proven to speed up training time and increase solution accuracy, through minimally complex incremental topology evolution.

In the end, each of the NEAT components were summarized as the generational NEAT process. A modified variant called rtNEAT for real-time problem domains was also introduced. The next chapter begins to discuss the SC: BW environment in more detail, as the chosen test-bed for the thesis work. Particularly, the micromanagement task and its complexity attributes are discussed. In doing so, it can be seen that both generational NEAT and rtNEAT have the potential to be applied to SC: BW.

Chapter 4

STARCRAFT TEST-BED

StarCraft: Brood War is a Real-Time Strategy game released by Blizzard Entertainment in 1998. In short, it is a science fiction themed military simulator, where players take control of futuristic armies with the aim to destroy all other opponents (Figure 12). Due to its fast paced, deep and balanced gameplay, it fosters human competition up to professional levels. The nature of the gameplay requires many capabilities of human-level AI, thereby making it an attractive test-bed for evaluating and observing these capabilities. In this chapter, I provide an overview of the SC: BW gameplay sufficient for understanding what an AI agent ought to do in order to play well. In particular, the micromanagement layer is discussed in more detail as it is the focus of this thesis. An analysis of the complexity of the micromanagement task serves to illustrate its difficulty for an AI agent and the appropriateness of the NEAT method. Finally, the BWAPI plugin is introduced as the primary tool for interfacing AI agents in the SC: BW environment.

Figure 12: An in-game screenshot of a SC: BW battle between two air unit armies. Combat is a central part of the SC: BW gameplay, as is in any military simulation.

4.1. StarCraft Environment

The environment of a SC: BW game can be viewed as a large 2D geometric map seen from an isometric view (Figure 13). The player's view is restricted to a portion of the entire map, but can be adjusted by scrolling in any direction. A Head-Up Display (HUD) overlays the game view and displays information about the game world. For example, the Minimap is an abstracted overview of the entire map, with colour coded objects to identify enemy and allied units, buildings, resources and unexplored terrain. It can be used to quickly identify enemies and allies, as well as quickly navigating to specific points of the map. The map can be perceived as a grid represented by two dimensional Euclidean coordinates. Units and building locations are represented on the map as x and y pixel points. They can also be selected and controlled using mouse and keyboard buttons. For example, a group of units can be selected by dragging a selection box over their positions, or selected individually via mouse cursor pointing and clicking. All game actions such as unit movements occur in real time.

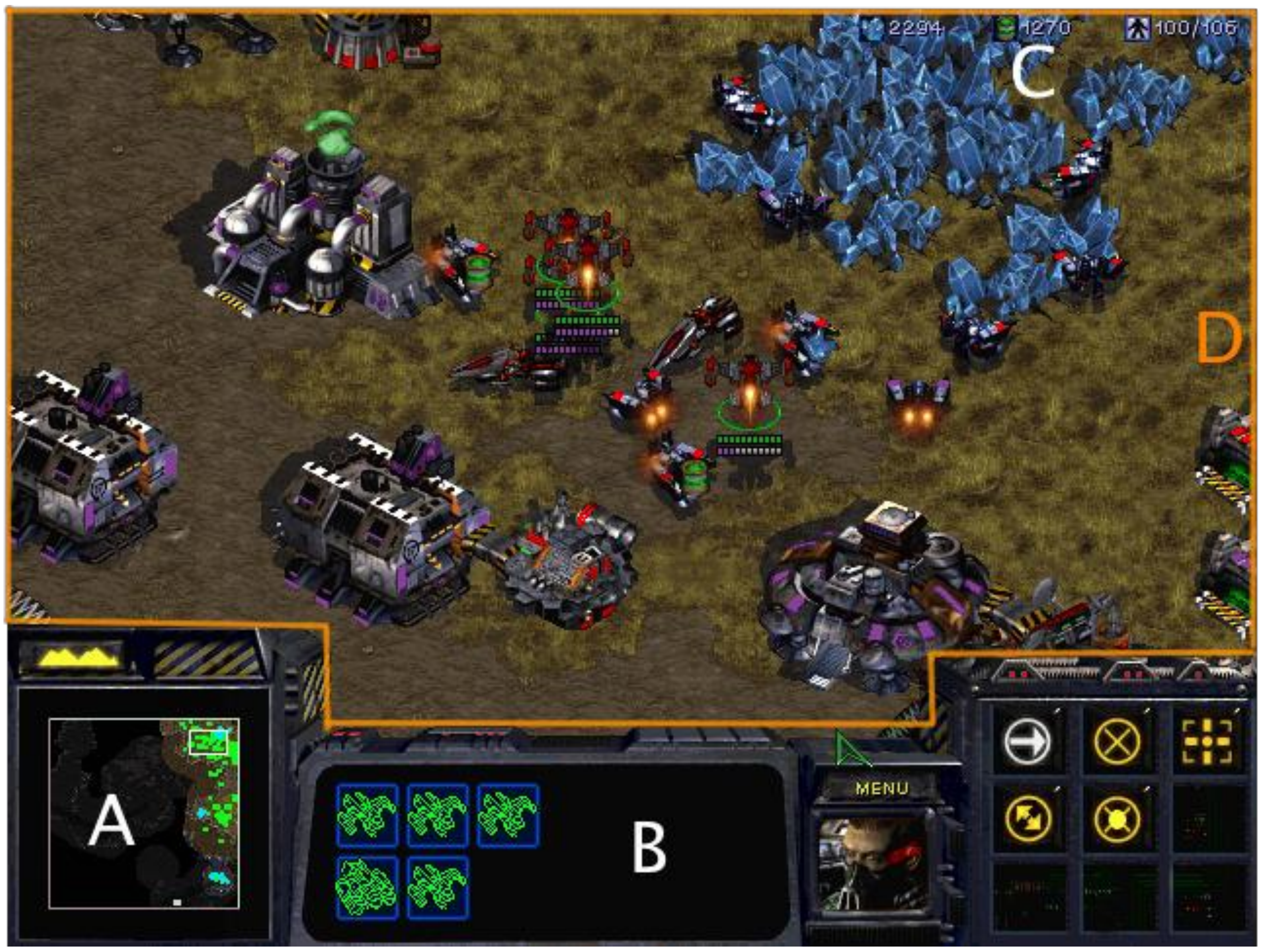


Figure 13: In-game screenshot of SC: BW. The game is viewed from an isometric view where 2D objects in the environment are projected such that they appear 3D. The HUD constitutes extra information such as the minimap (A), selected unit information (B) and resources (C). The rest of the screen is the view of the actual game environment (D).

Areas of the map that the player has not explored are covered in black. Units can scout unexplored areas simply by moving closer to them. Once an area is explored, the terrain can be seen. However, explored areas that do not have unit or building vision over them are covered by a Fog of War (FoW). Additionally, there are concepts of elevation in the terrain of SC: BW such that units on top of a higher elevation gain greater sight, while units on a lower elevation do not gain sight of the higher elevation (covered by FoW). FoW a common game mechanic in RTS games that hides the presence of enemy units and buildings on the map where there is no player unit vision. This constitutes the incomplete information attribute of the RTS game domain, where the knowledge of the environment and opponent states is limited to where a player has units and buildings. Therefore, the act of scouting the environment with units is an important aspect of SC: BW in order to predict and react to opponent build strategies and troop movements.

4.2. Gameplay

As previously mentioned, the objective of a SC: BW game is to eliminate opponent players. A player is eliminated when all their buildings are destroyed. Buildings and units have an attribute called Hit-Points (HP) that can be reduced from being attacked by enemy units (Figure 14). When HP drops to 0, the unit or building is destroyed. However, players do not simply begin with a full sized army ready for combat. At the beginning of a game, a player has the control of a single 'main' building and multiple worker units. The main building can produce more worker units, while the worker units themselves can be used to gather resources or to construct additional buildings that can eventually train combat units. Constructing new units and buildings expend resources, which the player has a small amount to begin with. Therefore the core gameplay of SC: BW involves gradually expanding from a small and weak base with no combat units, to a large base with an army of combat units and a sustainable economy.

There are a variety of buildings and units that facilitate different goals. Generally speaking, buildings can enhance either resource collection or military capabilities. For example, certain buildings can be constructed at new resource locations to exploit the resource, while others facilitate the training of combat units or the researching of combat enhancements (Figure 15). On the other hand, the majority of unit variations are for combat purposes, with the main exception being the worker units used for resource collection and constructing buildings, and other special purpose units such as those for transport or scouting. Combat units can be categorized in tiers, with lower tier units costing fewer resources but are less flexible and have lower combat strength than higher tiers. Combat unit tiers and enhancement research are unlocked via the construction of buildings in a hierarchical tree, such that certain lower tier buildings must first be constructed to unlock higher tiered buildings and units (Figure 16).

In general, collected resources can be used to increase resource gathering efficiency (via building additional resource gathering unit and buildings), or to increase military might (by training additional combat units or researching combat technologies). Therefore, part of the strategic challenge is to balance efficient resource gathering and combat unit production for both a sustainable economy and an effective army. During the early part of a game, players usually choose to follow a selection of optimized building orders that are able to achieve some set of units and buildings by some time constraint. These are akin to opening moves in chess, except spanning numerous actions over a timespan. For example, build orders that emphasize long term economic advantage may aim for as many workers and resource collecting buildings as possible while maintaining minimal combat numbers for defence. An aggressive 'rush' build order on the other hand, may aim to produce as many combat units as possible as quickly as possible to overwhelm an opponent in an early attack.

The sub-task of build order optimization is to facilitate the discovery of these build orders, by optimizing the order of economic and military spending that enables the goal set of units and buildings to be achieved within a certain timespan. The actual task of choosing a build order involves higher level strategic tasks such as scouting and opponent modelling, in order to choose the most favourable build order to counter an opponent's. For example, if an opponent adopts an extreme economic build with minimal defence, it may be very weak against a fast and aggressive build. On the other hand, if it is known that the opponent is following an aggressive build, the player can adapt a build order which achieves just enough defence to survive an early attack, while maintaining an economic edge that ensures its victory after the opponent exhausts its resources.

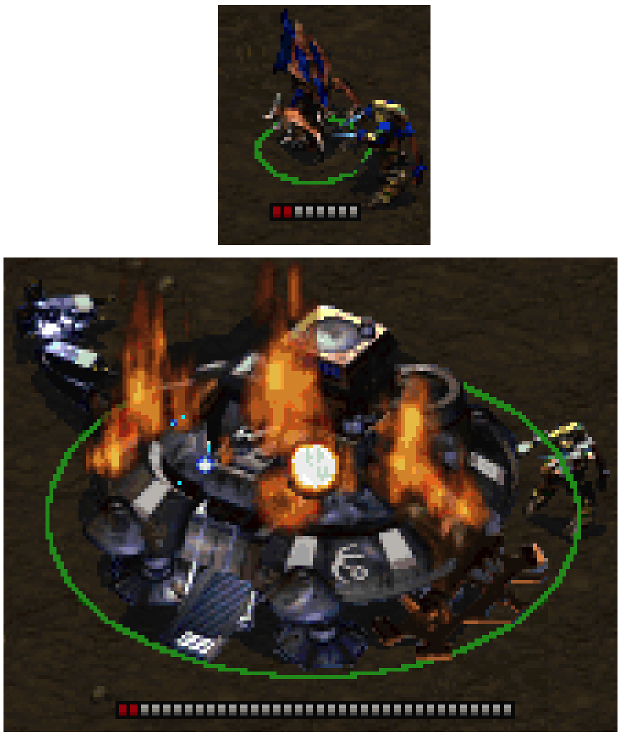


Figure 14: Unit and building Hit-Points (HP). Top: a damaged unit's HP is shown as a bar when selected. Bottom: likewise for a damaged building's HP. When a bar is full, the unit or building is undamaged. When the bar depletes completely, the unit or building is destroyed.

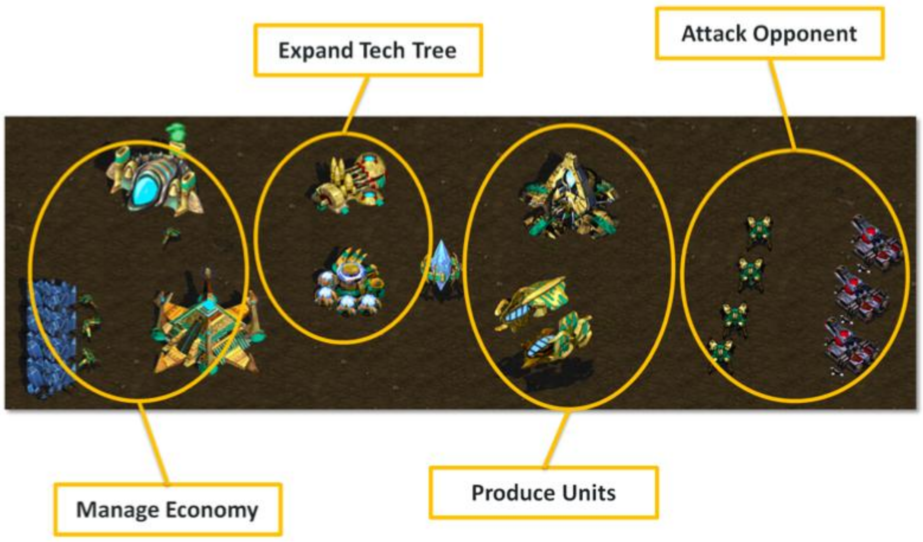


Figure 15: The general unit and building components of a SC: BW game. Resource gathering units and buildings are necessary to enable spending on other units and buildings. The construction of certain buildings enable spending on upgrades that enhance the combat capability of units, and unlocks higher tier units for training. Other buildings directly produce combat units that can be used to attack the enemy. Taken from (Weber, 2012).

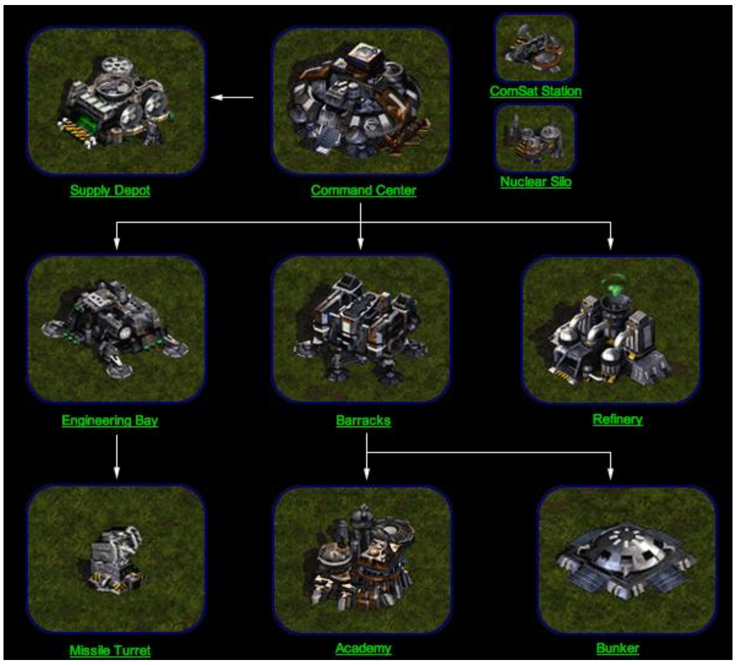


Figure 16: Part of the Terran race tech tree.

4.3. Races and Units

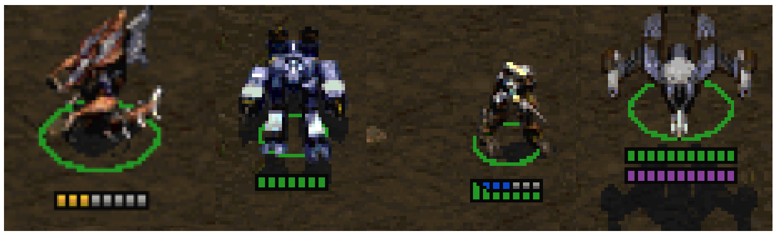


Figure 17: Examples of units in SC: BW. From left to right: Hydralisk, a Zerg ranged unit, SCV a Terran worker unit, Zealot a Protoss melee (short ranged) unit, and Wraith, a Terran flying unit. Green bars indicate a units HP, with the exception of the yellow bar on Hydralisk that indicates it has been damaged. The blue bar indicate shield points, which is a Protoss specific unit attribute that offers additional protection. The purple bar indicates energy points, which are used in certain units for special abilities.

The existence of three different races, each with its unique set of units, buildings and technology progression, adds to the variety and depth of gameplay. Each race's unique strengths are counteracted by their unique weaknesses, such that the gameplay is varied but generally balanced between any race. For example, the 'Zerg' race emphasizes cheap and fast produced units that are individually weak compared to other race units. Zerg units are purely biological and have the unique ability to regenerate HP over time. The 'Terran' race, modelled after futuristic humans, have slow producing units of moderate strength and cost. They also have units and buildings that facilitate strong defensive capabilities. Finally, the 'Protoss' race is a high tech alien race that emphasize strong and expensive units. Protoss units have a unique shield attribute that acts as extra regenerative protection against damage (Figure 17).

Besides race specific differences, many attributes differentiate units from each other. The basic attributes include the HP of the unit, its attack damage, attack cool-down and attack range. The attack damage of a unit is roughly how much HP it takes away from an enemy per attack, although other attributes such as weapon type and armour also affects this. The cool-down is the duration of pause between consecutive unit attacks, such that a unit that does half the damage but has half the cool-down as another unit can essentially inflict the same amount of damage, with all else being equal and given the same time span. The attack range of a unit determines a positional superiority, such that a long ranged unit can attack a shorter ranged unit without retaliation, until the shorter ranged unit moves closer.

Unit varieties increase even more when considering unit special abilities. For example, certain units can render themselves invisible while still attacking, which means enemy units cannot attack it unless special sensor units or buildings are nearby. Some units can expend an energy resource to execute abilities with wide ranging effects, such as high damage over all units in a specific area or attaching a protective shield over an allied unit. Units with flight capabilities can only be targeted by units with an air attack weapon. Due to such a large variety of unit attributes, there is no strict total ordering on unit strength. For example, a unit may have fewer HP and attack damage compared to another, but can defeat the other unit by landing more hits from a superior attack range. However there are still obvious unit quality differences, which are inherently balanced by different unit resource costs, training time and the technologies and buildings required to unlock the unit.

4.4. Multi-scale AI and Macromanagement

As previously mentioned, the gameplay of SC: BW involves tasks across multiple scales. At the macromanagement level are concerns of balancing economic development with military progress. For example, players aim for maximal vision of the map to predict and react to opponent strategies in the long term (such as build order adaptations) and short term (such as intercepting an advancing army). Control over a large portion of the map ensures maximal vision and access to resource locations, but would need a strong combat force to achieve. This would require immediate spending on military units, which may allow the opponent to gain an economic edge. If it can be determined that military spending is the best strategy for a scenario, it must still remain to decide what are the optimal types and numbers of units, and whether to spend on different unit upgrades. For example it is possible to spend on developing the technology for and training sensor units that can detect invisible enemy units. However, if the enemy does not develop such units then the spending would be wasted.

The challenges at the macromanagement level for an AI agent are numerous. Firstly, how does the agent generate effective strategies, such as: how to correctly predict opponent strategies, knowing which building and unit combinations best align together for specific strategies, the timing of attacks or when to expand to new resources, build orders, etc. Secondly, how does the agent detect and react to changes in the environment and opponent strategy, such as: acknowledging when the enemy is approaching for an attack, reacting to newly scouted information about enemy unit and building types, when and how to retreat or defend, when to change strategies, etc. Thirdly, how to synchronize these multi-scale objectives, from low level decisions such as an individual unit's actions, through to high level decisions such as planning the timing of expansions and attacks. Additional challenges arise when considering that the agent must make such decisions on a 2D environment with elevated terrain and obstacles. This requires terrain analysis techniques and for the agent to understand the significance of paths, obstacles and distances. In section 2.2.1, many of the existing approaches in tackling the macromanagement layer was discussed, and some have shown promising results compared to other AI techniques and against human players.

The focus of this thesis is on the lower scale task of micromanagement. In section 2.2.2, existing approaches to the problem and some analysis of its complexity was discussed. The actual problem and the general challenges are elaborated in the following section.

4.5. MICROMANAGEMENT

Micromanagement refers to the individual control of units in a combat scenario so as to override their default behaviours. This is done to maximize their individual combat effectiveness, since the default behaviour is often an inefficient control method. For example, when a unit is given an order to attack a location, it simply begins attacking the first enemy unit that comes into range. This leads to ineffective target selection and no coordination between multiple units. Instead, the player can explicitly direct unit attacks on prioritized targets, such as to eliminate the lowest HP units first so that the damage output of the opponent army is reduced as quickly as possible. On the other hand, it is possible to direct low HP units temporarily away from a fight until enemy units acquire a different target. In this way, the damage to a player's units is spread evenly across all units, increasing the average lifespan of each unit and maximizing the combined damage output over time (Figure 18).

Another example of micromanagement is the strategy of 'kiting', where a unit with a superior attack range can adopt a hit-and-run strategy against inferior attack range units. When properly micromanaged, the ranged unit can avoid all damage while successfully eliminating multiple enemy units. This works because a long ranged unit can attack a short ranged unit without retaliation until the short ranged unit moves into its own attack range. If the long ranged unit moves away after every attack, and if the short ranged unit does not have superior movement speed, then it will never catch up to attack the long ranged unit. The long ranged unit's damage output is not reduced, because it only moves after it has attacked during its weapon cool down period (Figure 19).

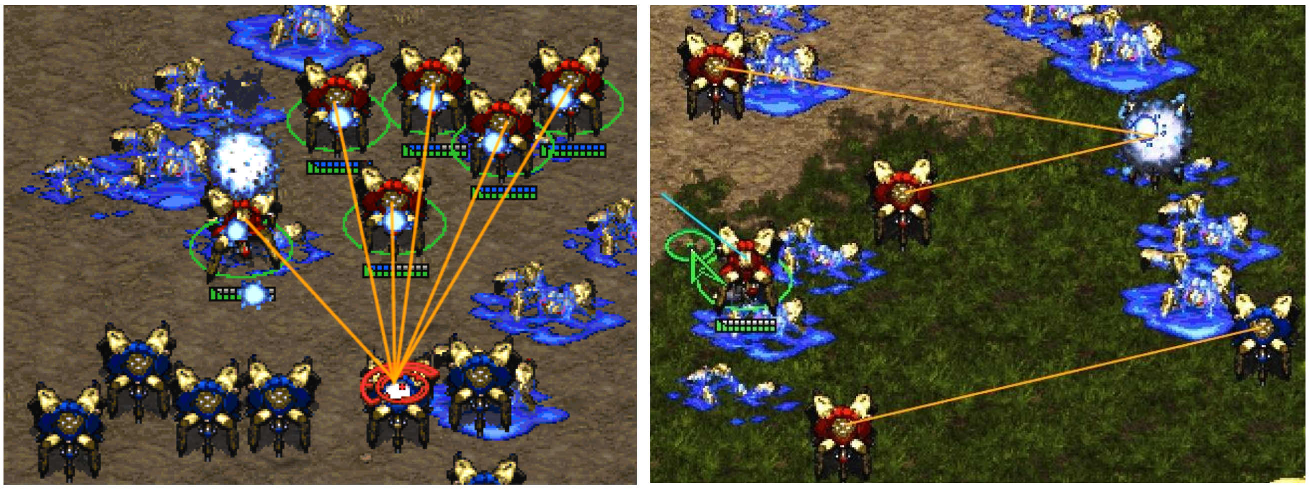


Figure 18: A combat scenario in SC: BW between two identically sized ranged unit armies. To gain an advantage, target prioritization and focus fire, such as aiming for the lowest health enemy unit to reduce enemy damage output is beneficial (on left, orange lines denote unit targeting). Also beneficial is to direct damaged units temporarily away from enemy units, so that the enemy reacquires targets. This will spread damage across units for maximum survivability and damage output (on right, blue line shows unit movement away from enemy units).

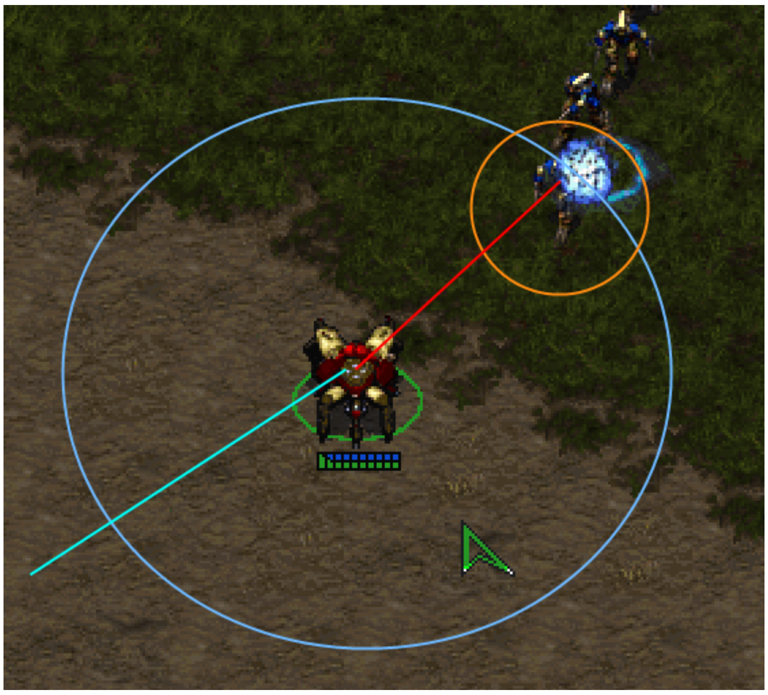


Figure 19: Example of kiting strategy. Blue circle denotes the attack range of the long ranged unit, while the orange circle denote the range of the short ranged enemy unit. The long ranged unit targets the short ranged unit for attack (red line) and when on weapon cool-down will proceed to move away (teal line), thereby always avoiding damage to itself while maintaining maximal damage output. If done correctly, a single long ranged unit can eliminate numerous short ranged units.

Simply adopting these two key principles of target selection and spreading incoming damage ensures an advantageous edge over non-adopters. However, to implement these principles for an AI agent controlling numerous individual units is a challenge. There are a number of choices of unit target selection, such as attacking the closest unit first or the weakest unit first. Simply directing all units to attack the weakest unit may waste some unit attacks (total damage of all attacks exceeding remaining enemy unit HP). Units with low attack range may take a long time to get into range to attack a target, and so it may be more beneficial for them to target the closest unit. Furthermore, it is more beneficial to include enemy unit attack damage into consideration, so as to target units with the highest damage output in proportion to its HP. The variety of unit types and attributes makes it difficult to define an optimal target selection or movement strategy for all units and in all scenarios.

Scripted behaviour is a popular way to implement these principles for AI agents. For example, different unit types can have their own scripts tailored for their unit's specific attributes (HP, damage, ranged etc.) However, scripted behaviour is a form of approximation that can be exploited, since it is limited by what the domain expert can conceive of, and is static once defined. Approximations are necessary to reduce the complexity of the state and action spaces in micromanagement, but there is still room for emergent behaviour. In the next section, the complexity of the micromanagement task is explored.

4.6. Complexity Attributes of Micromanagement

Micromanagement is a fast acting and reactive task. Many actions in quick succession must be performed to control each individual unit effectively, and in direct response to opponent actions. In professional gameplay, expert players are able to perform hundreds of actions per minute in their attempts to maximize combat unit control while also managing other areas in the strategic layer, such as queuing unit production, building construction, scouting and resource collection. For the average player, there is both a physical limitation of issuing commands quickly with a mouse and keyboard, and a mental limitation of making decisions over many different areas in quick succession. If too much focus is put on micromanaging units, then the overall economy and strategy of the player suffers. On the other hand, if micromanagement is not done properly, even a superior army can be defeated by a disadvantaged but well micromanaged enemy squad.

It is a common misconception that because computer AI can perform thousands of actions per second, it is necessarily better at micromanagement than human players. In reality, the AI must first deduce what is the correct move to make at each given point of a game, and not simply choose to perform thousands of random actions per second. It is more appropriate to consider how many actions the AI can evaluate within a game frame constraint in order to perform the approximately optimal move for that frame. If posed in this way, then it is easy to see why micromanagement is a challenging problem for AI.

Consider a trivial combat scenario involving two opposing armies of 12 units each. Suppose each unit has 8 possible directions to be moved, or each can attack any of the 12 units on the enemy's side. The total number of actions each unit can perform is therefore 20. For optimal control in each time frame, each unit should be assigned an action. In order to evaluate the optimal action for each unit, the combination of unit actions must be considered. This means there are $\mathbf{20^{12}}$ possible combination of actions to be evaluated for the optimal choice. In general, it is $\boldsymbol{A^U}$ combinations where $\boldsymbol{A}$ is the number of possible actions for each unit, and $\boldsymbol{U}$ is the number of units. The estimate becomes larger when considering the possible distances of movement actions, unit abilities and with increasing number of units. As mentioned in section 2.2.2, the complexity of the micromanagement task necessitates approximations, especially within real-time decision constraints. Although searched based techniques with appropriate heuristics are possible, they are currently confined to working within simulations of SC: BW with simplified and approximated states and actions.

Reinforcement Learning is a viable alternative to heuristic based search approximation. There are already examples of RL applied to micromanagement in the literature (section 2.2.2). NEAT based RL is a strong competitor against traditional RL techniques and is apt for the micromanagement task (section 2.2.3). Instead of evaluating unit moves and looking ahead as in heuristic search, NEAT searches for an ANN best able to specify the approximate optimal actions for a state. The challenge here is finding the appropriate state and action space representation of micromanagement, as well as a meaningful fitness function that allows successful incremental NEAT evolution. Part of this challenge is inherently tied to the technical workings of the SC: BW environment, such as in the way information about the environment is revealed to an agent and in how unit actions can be specified. In the following section, the BWAPI framework is discussed, which exists as the input and output layer between the SC: BW environment and custom AI.

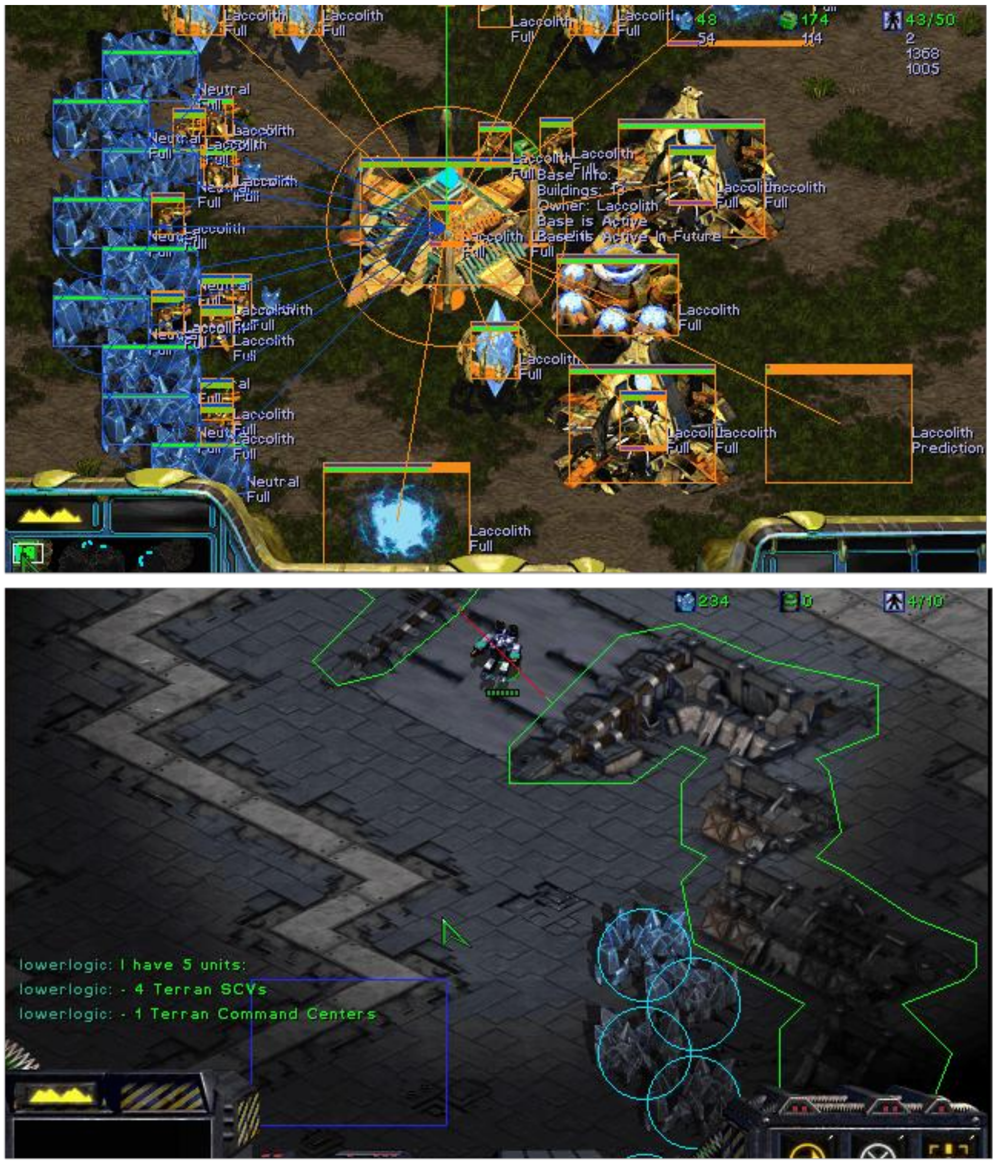
4.7. BWAPI

The Brood War Application Programming Interface is an open source framework for interacting with SC: BW. Almost all AI research involving SC: BW as the test-bed uses BWAPI for executing actions and for querying information in the game. For example every SC: BW AI tournament requires entries to be AI modules developed to run with BWAPI. Simply put, it is the most preferred framework for the development of AI agents in SC: BW.

Although many language extensions exist, the core BWAPI framework is developed in and applied with the C++ programming language. The framework functions as a plugin DLL that is injected into a running instance of SC: BW. Custom AI modules act as extensions to this plugin and are executed by the BWAPI DLL. With this, BWAPI gives full access to game state and actions that a human player is given. With the activation of some cheat flags it can provide even more information, such as the complete map information of all enemy units. All AI interactions with the SC: BW game are through the BWAPI's functions. By enforcing BWAPI as the standard for AI tournaments, it allows unbiased comparisons and ensures AIs do not employ cheating information if they are not allowed to.

In SC: BW, the game state is updated continuously every 56 milliseconds when set to the normal game speed. BWAPI exposes an event that is triggered on every update, which allows custom AI agent code to run in order to react to the updated game state. The state of the game can be queried, such as about each of the players units and buildings. Actions can also be issued for each of the units and buildings. Enemy unit and building information is restricted to those that can be seen via the player's unit sight. On top of these basic features is a library of UI drawing features that allow debug or analysis data to be rendered on screen. Other library extensions built on top of BWAPI offer additional features, such as terrain analysis to provide pathing and obstacle information for the AI agent (Figure 20).

The work in this thesis uses the BWAPI framework to develop AI agents that interact with SC: BW. In the following chapter, the implementation details of integrating the NEAT methodology into a BWAPI AI module is discussed in detail. The design and details of the micromanaging agent module is also discussed, which enables the learning of the correct unit behaviours for specific combat scenarios.



lowerlogic: I have 5 units:
lowerlogic: - 4 Terran SCVs
lowerlogic: - 1 Terran Command Centers

Figure 20: Top: example of debugging and analysis information rendered onto the UI in the SC: BW bot Skynet[15]. Bottom: the result of terrain analysis by a BWAPI extension called BroodWar Terrain Analyser[16].

[15] Skynetbot, a SC: BW Bot using BWAPI. https://code.google.com/p/skynetbot/

[16] BWTA, a terrain analyzer for BWAPI. https://code.google.com/p/bwta/

Chapter 5

DESIGN AND IMPLEMENTATION

In this chapter I discuss a number of design considerations in modelling SC: BW micromanagement for the application of NEAT. This involves the design of the basic starting network architecture as well as the fitness function used to guide evolution. A number of NEAT specific parameters are also discussed, such as the probability of different mutations and the values chosen to derive the number of game ticks between replacements in rtNEAT. Finally, the agent implementation details are discussed in terms of the architectural integration between BWAPI and NEAT.

5.1. NETWORK MAPPING

Recall in section 3.5 that the NEAT process involves a continuous evaluation of a population of ANNs. In order to apply NEAT for micromanagement, an appropriate model of the problem domain is needed. Fundamentally, there needs to be a mapping between the ANNs in a NEAT population and the combat units within a micromanagement scenario. This involves choosing the right abstraction of state information (ally and enemy unit attributes such as HP and weapon cool-down, distances and locations of units etc.) and actions (unit attack target selection, retreat movement direction etc.)

At first glance, the NEAT methodology maps naturally to multiple unit control in micromanagement. Each unit in a combat scenario can be modelled and mapped to a single neural network in the NEAT population. The fitness of the network is then determined by how well the single unit does within the micromanagement task. Such a one to one mapping is intuitive and should be easy to manage. Alternatively, multiple units can be mapped to a single network such that the network is evaluated based on the performance of the entire squad of units. If necessary, a single unit can be mapped to multiple neural networks, such as in Shantia et al 2011, where each ANN represented the reward function of a single action and where the action associated with the highest output network is chosen.

There are a number of issues with models that are not a one to one mapping, related to granularity of control and network complexity. When mapping multiple units to a single network, the basic network structure is necessarily complex. The network must be able to take in enough information about the problem state to give meaningful control outputs for multiple units. The number of outputs are also dependent on the number of units. Unless meaningful groupings and abstractions are used, the network structure will have too large a number of parameters for NEAT to search over (Figure 21). On the other hand, groupings and abstractions take away the granularity of state information and control of units (Figure 22).

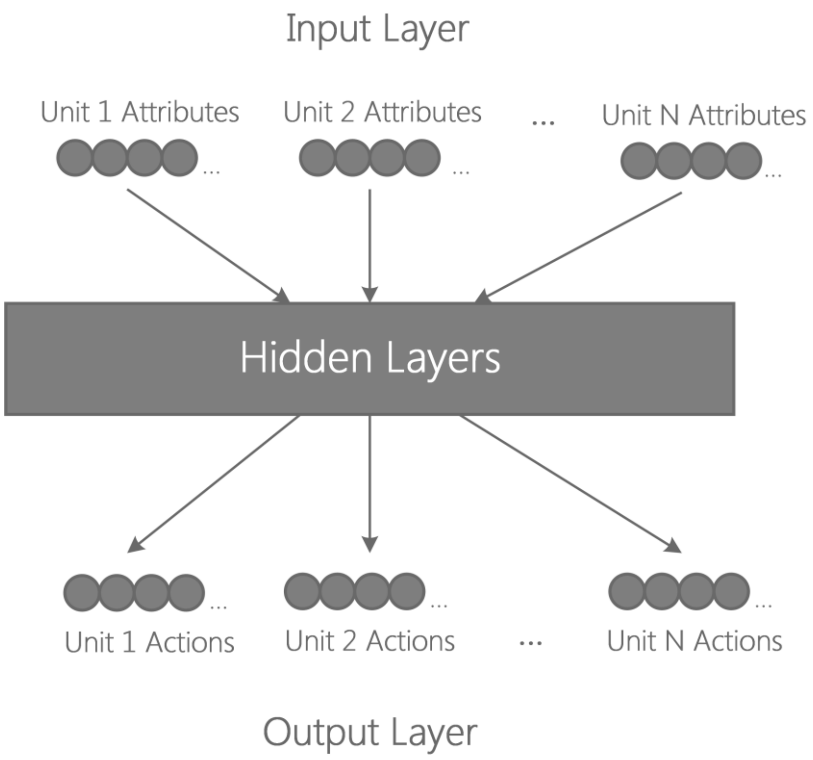


Figure 21: Example of a many unit to one network mapping model. A fine grain input layer must capture the attributes of each enemy and allied unit (such as current health and weapon cool-down), and the output layer must specify the action for each unit. Consider a 12 vs. 12 unit scenario with 4 unit attributes (HP, cool-down, damage and range) and 20 possible unit actions (move in 8 directions or attack one of 12 enemies). The number of input nodes is $\mathbf{24 \times 4 = 96}$ and output is $\mathbf{12 \times 20 = 240}$. This results in $\mathbf{240 \times 96 = 23,040}$ connections as a starting network topology and increases quadratically with number of units.

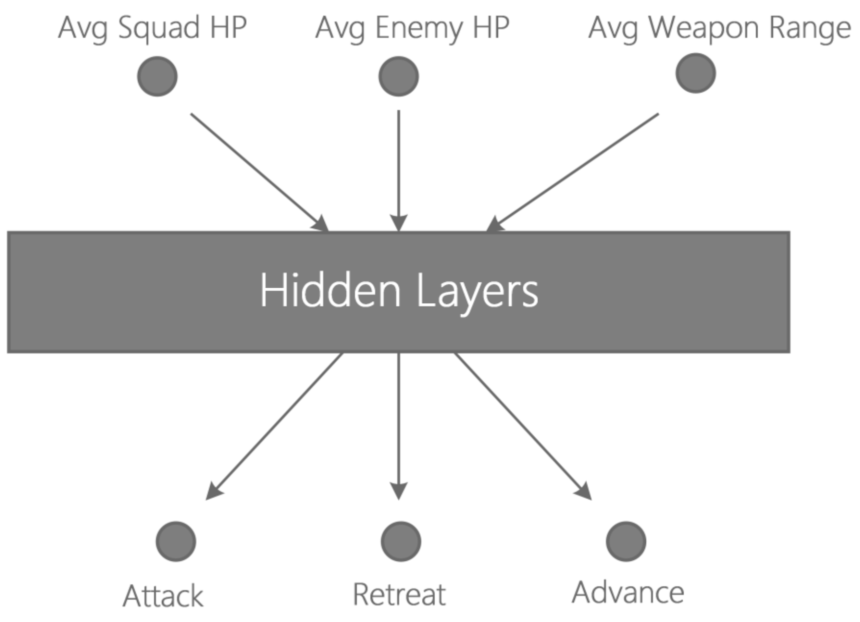


Figure 22: A highly abstracted, many unit to one network model. State inputs are aggregated and output actions are at squad level granularity (i.e. all units perform the same action). Such a model is very efficient for NEAT to start with (only 6 starting weights connections to optimize) but is very limiting in what can be learned and controlled.

Mapping multiple networks to a single unit is also problematic when considering the NEAT evolutionary process. The reason it works in Shantia et al 2011 is because a RL learning approach is used and each ANN approximates a state-action value function for a single unit action. All units use the same set of ANNs, and each network is updated via variants of the Sarsa RL algorithm. In contrast, NEAT cannot train multiple ANNs with different functions (action specific output), because you cannot meaningfully compare fitness or reproduce between different purposed networks. However, it may be possible to separately evolve different ANNs for different unit control purposes, such as one for enemy target selection, one for deciding whether to attack or retreat, and another for the direction of the unit movement. This would require separate runs of the NEAT algorithms and some consistent and default behavior to fill in for decision controllers that have not yet been evolved.

The model chosen in this thesis is a one to one mapping between network and unit. This is due to the complications as mentioned above with many to one mappings, and the technical intuitiveness of attaching a single network to a single unit. During a NEAT or rtNEAT evolution cycle, new networks can be easily reattached to an exisitng unit. Fitness evaluation can also be intuitively conveyed as the lifetime performance of a unit between matches (generational NEAT), or its immediate real time performance (rtNEAT). In the next section, a number of one to one mapping neural network models are discussed, which illustrates a network complexity versus information granularity trade off.

5.2. Neural Network Design

With a one unit to one network mapping in mind, it remains to decide a network's inputs and outputs. These constitute the state information necessary to learn to control a unit effectively, and the unit actions that can be performed for effective unit behaviour. In theory, a neural network can approximate a function mapping the entire state information of a SC: BW game as inputs, and all possible unit actions as the outputs. Learning the optimal mapping between these inputs and outputs would achieve the optimal unit behaviour for all SC: BW game states. However in practise, the state and action spaces are too large and complex to model completely as such, and learning such a complete mapping is intractable. Thus, abstractions are necessary.

5.2.1. Initial Model

A number of ways to abstract unit state and actions were considered. Figure 23 depicts the *initial model*, with inputs and outputs based on domain knowledge and previous work in the literature. Many of the inputs involve the unit attributes described in section 4.3 and are chosen to induce learning micromanagement behaviour described in section 4.5. For example, a unit with a superior weapon range to an enemy unit's should retreat some distance when its weapon is on cool down to minimize damage (kiting). Another example is if a unit's health is low, it may be beneficial to temporarily retreat so as to be out of the line of fire, and return to fight when more allies are in range.

In this model, the output of the neural network corresponds to two possible unit actions: fight or retreat. The action with the largest corresponding node output after network activation is the chosen action to be performed. If the fight action is taken, the unit follows a simple routine that attacks the enemy unit with the lowest HP within its weapon range. The retreat action makes the unit move a small distance in the direction of a computed vector. The vector is computed by weighing enemy and obstacle vectors such that the unit retreats away from enemy units and obstacles These actions are based on similar implementations in the literature (Shantia et al., 2011; Wender & Watson, 2012) as they are simple to implement, but complicated enough to produce sophisticated micro-management behaviour when performed in various sequences.

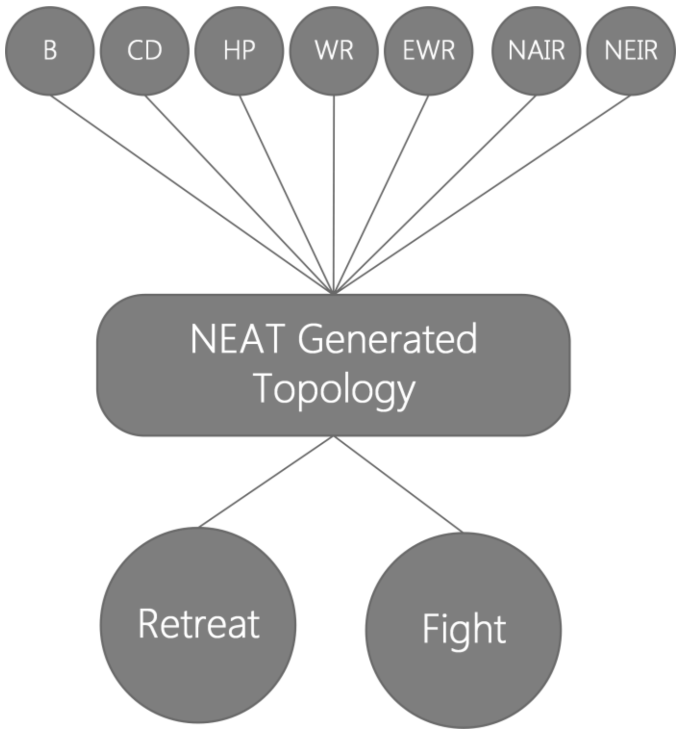


Figure 23: A fully connected and feed forward neural network architecture, which has its hidden nodes gradually added or changed via NEAT or rtNEAT evolutions. Nodes on top from left to right: Bias, weapon Cool Down, remaining Hit Points, Weapon Range, Enemy Weapon Range, Number of Allies in Range and Number of Enemies in Range. These denote a mixture of agent internal and external percepts as input to the network, while nodes Fight and Retreat denote the outputs as two possible unit actions. All inputs are normalized to $[\mathbf{0}, \mathbf{1}]$ for consistency.

The performance of this model was evaluated in a number experiments (section 6.1), with good results against the default SC: BW AI. However, there are a number of limitations to the model, namely in the granularity of its actions. The 'Fight' subroutine is comparable to a static script that targets lowest health enemy units, while the 'Retreat' subroutine is also a static script for navigating a unit away from enemies. Without finer granularity control, a unit cannot take advantage of other factors in enemy target selection (i.e. cases where low HP units are not the ideal targets such as high weapon damage targets, short ranged units or other unit attributes) and unit movement (e.g. the location of allied units or the nature of the map terrain may be important). Moreover, it would be interesting to see if an agent is able to learn these scripted behaviours when the model allows for it.

5.2.2. *Alternative Models*

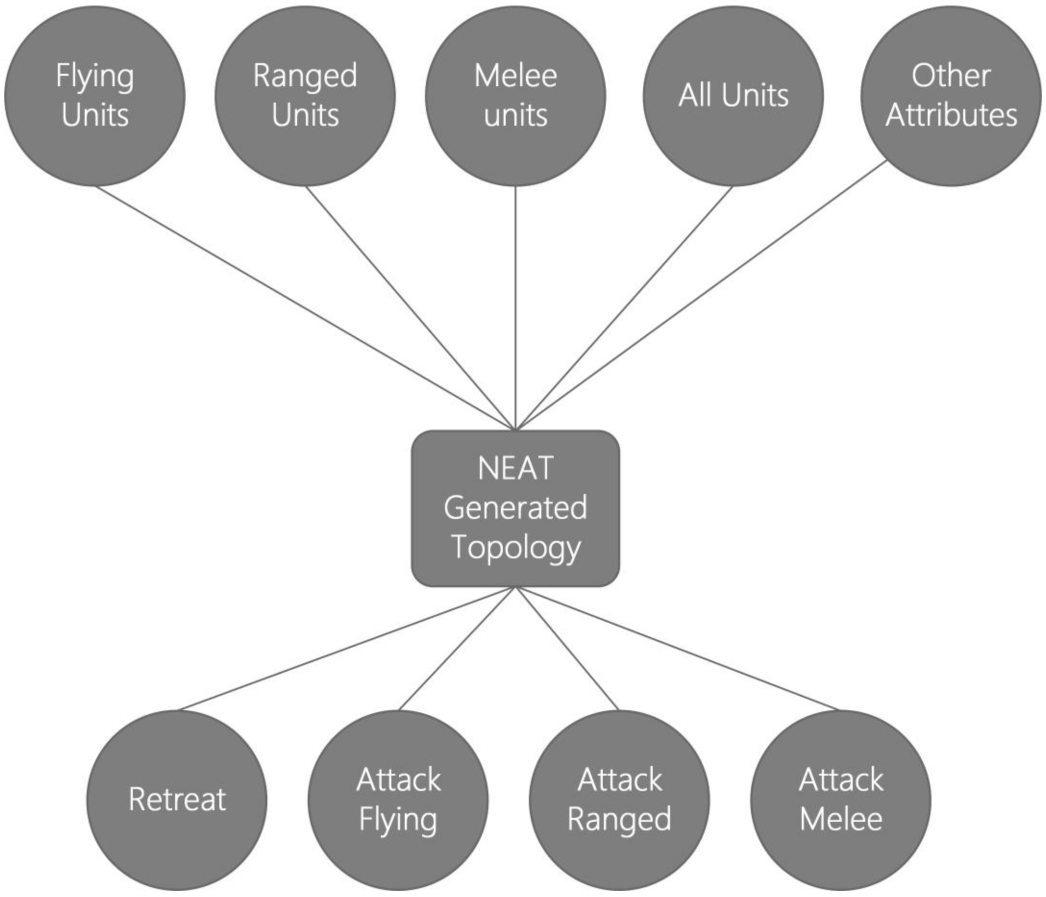


Figure 24: An alternative extension of the initial neural network model with unit type granularity. Here the 'Attack' output is replaced by three unit type specific attack targeting subroutines. Four additional inputs were also included, which specified the number each type of enemy unit that can be seen, normalized by dividing the total number of units.

An attempt was made to incorporate unit type granularity into the initial model (Figure 24). However, some initial testing scenarios involving multiple unit types showed the model was ineffective. This was due to units no longer prioritizing low HP units regardless of unit type, which would sometimes allow low HP enemy units to linger. Another alternative involved full granularity unit input and output (Figure 25). Since each unit is a possible output target and all unit attributes are included as input, both low HP targeting and unit type targeting behaviour can be learned and combined.

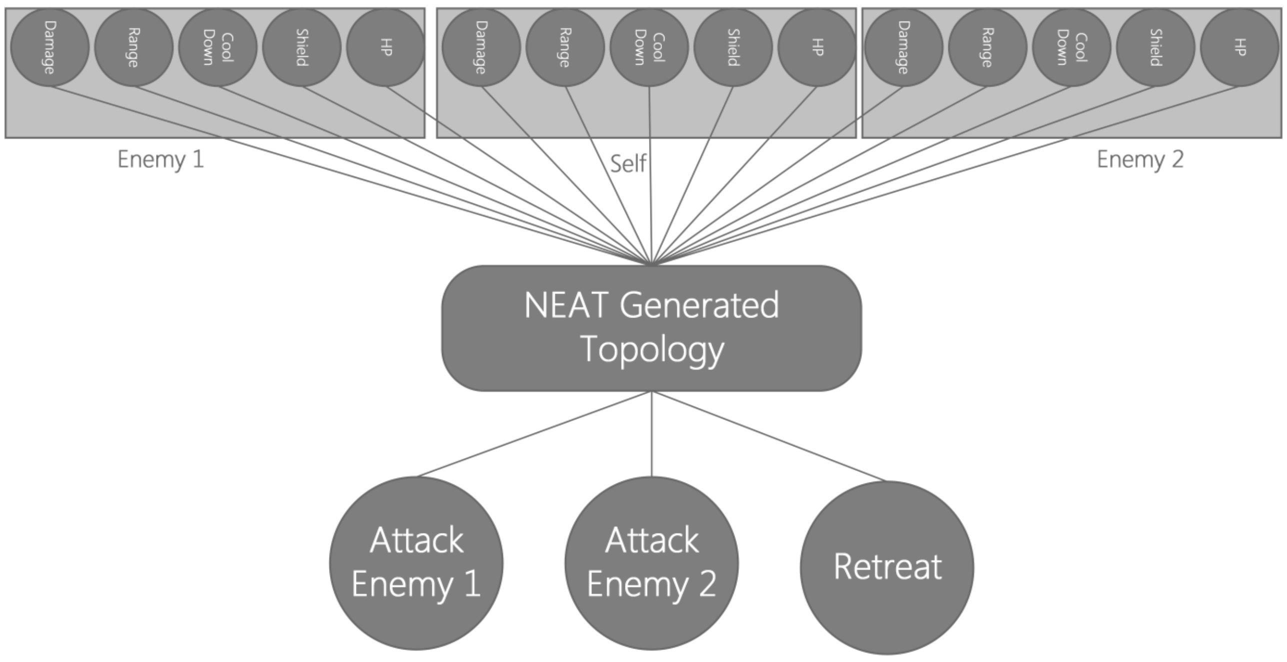


Figure 25: Full unit granularity attack model. Attributes of each enemy and the unit itself is included as input. Each enemy unit attack target corresponds to a single output. The figure depicts a simplistic scenario against two enemy units. Each additional enemy unit adds an additional output node, and a number of input nodes corresponding to unit attributes involved. Not depicted are allied unit attributes which may also be important.

There are two major problems with this model. Firstly, it does not incorporate game state information on location of enemy and allied units. Simple relative distances between units can be added as inputs, but this does not convey directional information. This type of information is important to expand the granularity of the Retreat action, and is also important in some decisions of target selection (e.g. melee units should not target a low health unit that is blocked by other enemy or allied units in the way).

Secondly, the number of units for which the network model can work on is static. NEAT cannot dynamically introduce or remove input and output nodes (i.e. with variable unit number scenarios). A possible solution to this is training multiple neural networks for separate unit number scenarios, but this requires multiple NEAT evolutionary runs for each possible unit number match up, and for an agent to dynamically switch out its neural network controller during a micromanagement engagement. Furthermore, such a model may not scale well for larger unit numbers, since the starting topology becomes more and more complex as the number of input and output nodes increase in respect to unit numbers.

5.2.3. Directional Granularity Model

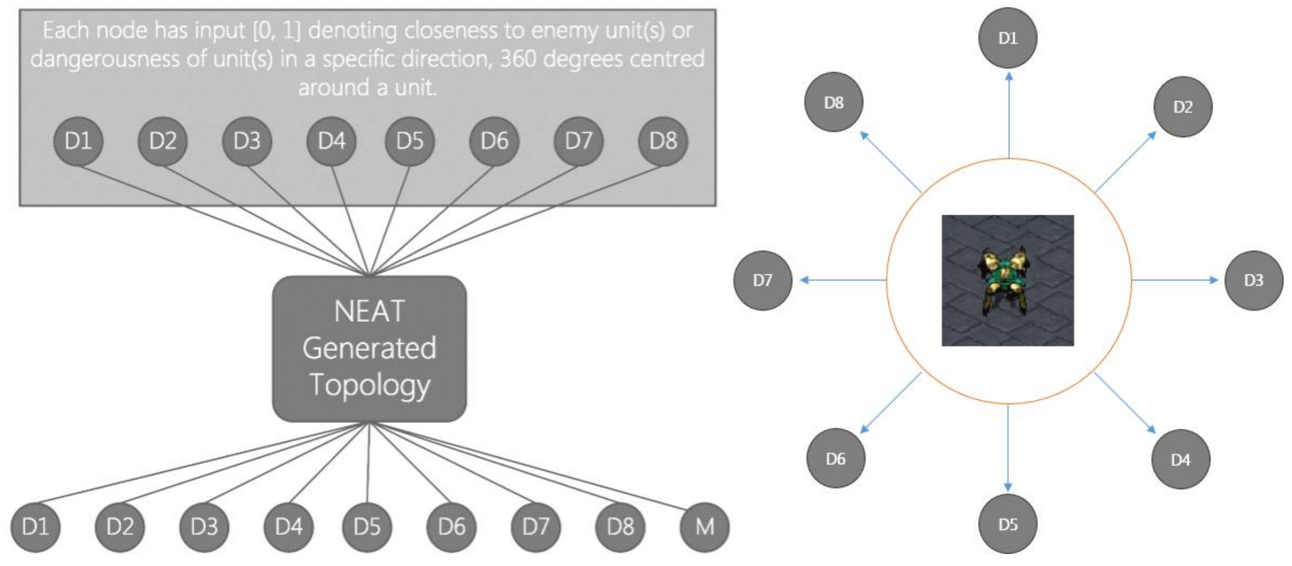

Each node has input [0, 1] denoting closeness to enemy unit(s) or dangerousness of unit(s) in a specific direction, 360 degrees centred around a unit.
D1
D2
D3
D4
D5
D6
D7
D8
NEAT Generated Topology
D1
D2
D3
D4
D5
D6
D7
D8
M
D1
D2
D3
D4
D5
D6
D7
D8

Figure 26: Directional sensors modelled as neural network inputs. The direction of enemy units can be specified as inputs to one of 8 nodes. Output nodes also map to 8 directions for movement, with an optional node to specify the magnitude of movement in a direction. This setup essentially replaces the Retreat routine in previous models, by moving the unit in the direction associated with the largest node output. Not depicted are all the other relevant percepts and outputs for attack target selection.

It is possible to model directional information as a set of sensors around a unit (Figure 26). Here the sensors map directly to input nodes in a neural network. Outputs also correspond to directions that the unit can move towards. The direction with the largest corresponding node output is the chosen direction of movement for the unit. Another node can specify the magnitude, or how far to move the unit in a specified direction. Such a model can replace the Retreat subroutine in the other models such that if the unit decides to retreat, it will do so at a direction and for a distance specified by the network outputs. A successful learning criterion mirroring the behaviour of the Retreat subroutine would be if the network mapped the input and output directional nodes to oppose each other, such that the unit will retreat in the opposite direction to enemy unit positions.

In section 6.2 I describe some tests to evaluate the effectiveness of this model. The directional node input and outputs are combined with the percepts and outputs of the initial model from section 5.2.1 (unit attributes and the fight subroutine). The directional inputs are defined as -1 if no enemy units are close by (within some distance threshold, such as twice the maximum weapon range of units) in that direction, -0.5 if an enemy unit is close by but neither unit is within range of each other, 0 if the enemy is within the current unit's weapon range and 1 if the current unit is within the enemy unit's weapon range.

5.3. Fitness Function

In both the NEAT and rtNEAT algorithms, the evolutionary process is guided by a simple fitness metric. In the context of SC: BW unit micromanagement, the fitness should reflect the performance of an individual neural network controlling a unit during a micromanagement scenario. This is similar to the reward signal function implemented in a reinforcement learning approach that rewards an agent given its performance (Wender & Watson, 2012). There are a number of ways to define a unit's success over time, for example its damage output over its lifetime, the number of enemies a unit eliminates, and it's remaining HP (or whether it survives). Such factors are considered in the fitness function in Gabriel et al., (2012) and in the reward function of Shantia et al., (2011). Ultimately, a successful unit should minimize damage dealt to it-self while maximizing damage dealt to enemy units, and a fitness function should reflect and encourage this.

With some consideration, the number of units eliminated and whether a unit survives or not can be argued as poor metrics for fitness. In SC: BW, the unit that deals the fatal blow (the final attack that reduces the enemy HP to 0) to an enemy unit is considered the eliminator of that enemy. Rewarding such a unit with higher fitness over others would disregard the contributions of allied units in reducing the enemy unit's HP. It is entirely possible for a unit to only contribute a single attack, but deals the fatal blow to claim the higher fitness reward for unit elimination, over other units that dealt more significant damage. The survivability of a unit also does not tell us whether the unit has contributed any significant damage towards enemies units. A unit's survival is only important if it maximizes damage dealt to enemies, but rewarding units survived may reward ones that simply retreat throughout an entire match.

Thus in the experiments described in this thesis for both NEAT and rtNEAT, the fitness function is conveyed strictly using the damage dealt and damage received for an individual unit. Specifically, the fitness $\boldsymbol{F_i}$ for a unit $\boldsymbol{i}$ is defined as:

$$F_i = \frac{TDD_i - HPL_i}{IHP_i} + 1 \tag{8}$$

The function takes in the Total Damage Dealt by a unit $\boldsymbol{i}$ ($\boldsymbol{TDD_i}$) and its Hit Point Loss ($\boldsymbol{HPL_i}$) accrued over a match and divides by its Initial Hit Points (IHP_i). In theory, the fitness is only upper bound by the total hit points of all enemy units (if a single unit eliminates all enemy units by itself, and receives no damage). However in practice, the average fitness of each unit falls under $[0, 2]$ where at the lowest, the unit has dealt no damage and does not survive, and at its highest value it has dealt twice as much damage than it has received.

5.4. NEAT Parameters

Parameter	Description	Value
Excess Coefficient	Weight of excess gene numbers in similarity metric. (c_1 of EG in section 3.4)	2.0
Disjoint Coefficient	Weight of disjoint gene numbers in similarity metric. (c_2 of DG in section 3.4)	2.0
Weight Difference Coefficient	Weight of weight differences in similarity metric (c_3 of $\overline{W}$ in section 3.4)	1.0
Compatibility Threshold	Threshold used in determining species based on similarity metric (D_t in section 3.4)	3.0
Mutation Power	The power of a weight mutation on a link	2.5
Survival Threshold	The amount of within each species allowed to breed which controls interspecies greediness.	0.2
rtNEAT specific		
P	Population size matching unit numbers used to determine replacement time (section 3.6).	12
I	The amount of population eligible for replacement at any time (section 3.6)	0.5
m	Minimum time alive for any unit (section 3.6)	300

Table 1: Summary of key parameters and their values in NEAT and rtNEAT.

Having decided on a model of the problem, network input and outputs, and a fitness function to guide NEAT selection, what remains is to select values for various parameters for running NEAT and rtNEAT. Table 1 summarizes some of the key parameters and their chosen values used in experiments of NEAT and rtNEAT in this thesis. The choice of these values are the result of a combination of empirical testing and from existing sources. For example the rtNEAT specific parameters are based on those used in (Stanley et al., 2005), and the base NEAT parameters are based on recommendations from example experiments on the NEAT users page and from source code[17].

[17] NEAT Users Page parameter recommendation and definitions. From http://www.cs.ucf.edu/~kstanley/neat.html.

5.5. Agent Implementation

In order to apply the theoretical designs discussed in the previous sections to SC: BW micromanagement, a NEAT agent must be integrated into the SC: BW game environment. The BWAPI open source framework discussed in section 4.7 is used to create and run AI modules in SC: BW. This is integrated with NEAT open source code and together constitutes an AI agent system for micromanagement. A number of open source implementations using different programming languages exist for both NEAT and BWAPI. Coincidentally the main development branches for BWAPI and the original NEAT package by Kenneth O Stanley are both in C++, which allows for a straight forward integration. With an emphasis for code efficiency and on using original sources, the default C++ packages are used in this thesis. The integrated development environment used was Visual Studios 2012, which has attractive integrated compilation and debugging facilities and is also a personal preference.

5.5.1. Agent Architecture

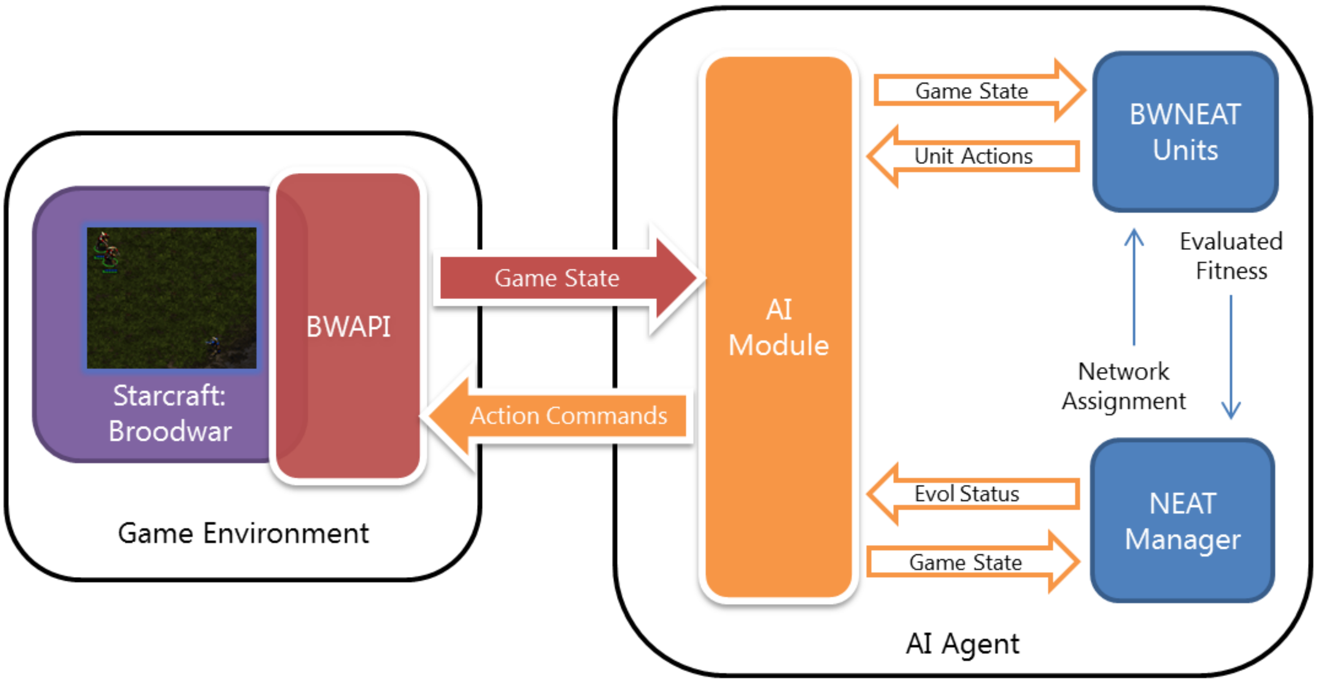


Figure 27: An overview of the agent architecture. BWAPI serves as the communication layer between the SC: BW and custom AI module codes. The AI module communicates with BWAPI to get information about game state and issue commands. BWNEAT objects represent an individual game unit, with an attached neural network which it activates to decide what actions to take. The NEAT Manager module manages NEAT and rtNEAT algorithm evolutions, based on fitness retrieved from BWNEAT units and controls the evolution and network replacement policies.

Figure 27 summarizes the architecture of the agent system. BWAPI acts as a communication layer for custom code modules and exposes functionality to retrieve information about the game state and to issue commands to game units. A unit in SC: BW is encapsulated as a BWNEAT class unit, which links a game unit with a neural network. The unit is responsible for using its internal percepts, as well as external game state percepts from the AI Module, as inputs to its neural network. It is also responsible for activating the network and interpreting the output as a decision for what the unit should do next. The NEAT Manager module is responsible for initially instantiating the BWNEAT Units, assigning each of them a neural network, and is an interface to the NEAT and rtNEAT algorithms. It receives evaluated fitness from each unit, performs the NEAT and rtNEAT epoch processes and reassigns neural network offspring to BWNEAT units.

5.5.2. Main Event Processes

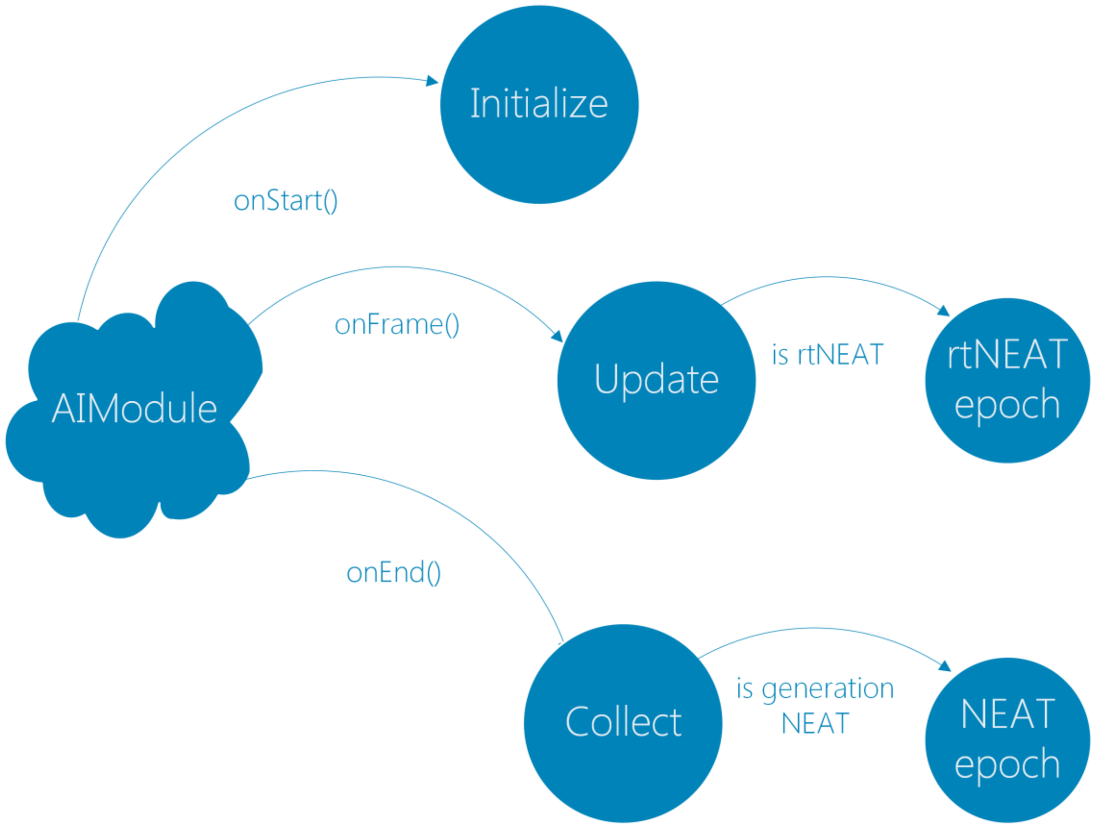


Figure 28: A summary of the main loops and processes of the AI agent. The AIModule has three main events that dictate the flow of the system. The onStart() event is triggered at the start of an SC: BW game and is when the BWNEAT units and the NEATManager is initialized. The onFrame() method is called continuously throughout the SC: BW game and is where each unit updates its percepts, and interprets its neural network controller for its next action. If rtNEAT is run, this is also where rtNEAT epoch occurs. In the onEnd() method called when the game ends, statistics are collected and the generational epoch process is run if it is a normal NEAT run.

Figure 28 summarizes the systems main event processes. The first event *onStart()* is called at the beginning of an SC: BW game, which begins the *Initialize* process. This process first initializes a NEATManager object by passing the collection of starting game units (represented in BWAPI in a Unit class). In turn the NEATManager initializes a population of neural networks (based on a basic input and output design or from a population saved in previous runs, and represented in NEAT as a population class) and a collection of BWNEATUnit objects. Each BWNEATUnit object consists of a BWAPI unit, a NEAT organism (a NEAT class encapsulating a neural network and its fitness) and a number of variables and functions to store and calculate a unit's percepts.

In SC: BW, the game state is updated continuously every 56 milliseconds when set to the normal game speed. BWAPI exposes an event that is triggered on every update called *onFrame()*, which allows custom AI agent code to run in order to react to the updated game state. In our system, it is during this event that the game state is fed to each BWNEAT unit and when each unit activates its associated neural network in order to decide on an action. This is summarized as the *update* process pseudo code as follows:

```
foreach (BWNEAT unit i in bwneatUnitList)     {

       foreach (Percept p in unit i percepts) {

              /** Update each percept p. Internal percepts such as unit health and
              cool down can be queried using BWAPI calls such as
              BWAPI::Unit::getHitPoints(). Environmental percepts such as enemy unit
              directions require a number of calls and additional calculations. **/
              Update(p);
              // Feed each p into NN inputs nodes
              Feed(p);
        }

       // Activate i's neural network, which has all percepts in its input nodes
       activateNN(i);

       /** Interpret the output of i's neural network. The nodes with the largest
       outputs denote the action to be taken. An action can be performed by using
       BWAPI calls such as BWAPI::Unit::attack() or BWAPI::Unit::move()
       **/
       interpretNN(i);

       //If rtNEAT is run and number of frames between replacements has passed
       //then perform epoch
       if (isRTNEAT && Game.frameCount % ReplacementInterval == 0) {

              NEATManager.Epoch();
       }
}
```

Figure 29: Pseudocode for the *Update* process in the AIModule.

Lastly, the *onEnd()* method called at the end of an SC: BW game begins the *collection* process. This does a saving to file of the population of neural networks, statistics collection (such as recording average fitness and win rate in experiments and some memory management (necessary in C++ to release unused dynamically allocated memory). If generational NEAT is run, here is also where the NEAT epoch process occurs. The NEAT and rtNEAT processes involved are those as described in section 3.5 and 3.6 and involve calls to the NEAT population library code.

5.5.3. Unit Percepts and Actions

As mentioned above, a unit's percepts and actions are implemented using a combination of BWAPI library calls and other calculations. In the initial model (section 5.2.1), the internal unit percepts are retrieved via BWAPI calls in the Unit class, such as **`BWAPI::Unit::getHitPoints()`** for querying a unit's current hit points and **`BWAPI::Unit::getShields()`** for a unit's current shield points. External percepts such as the number of ally and enemy units in range of the unit involves BWAPI calls to retrieve collections of observable units in the game, and conditionally counting units that are within the unit's range. For the *Fight* and *Retreat* actions, their behaviours are summarized as follows:

```
//for a current unit u:
void BWNEATUnit::Fight(){
     if(we have issued a command to unit u already in this frame){
             return;
     }
     else{
             find an enemy unit i that is closest to unit u;
             if (unit i exists){
                    if (the last command for unit u was to attack unit i){
                           return;
                    }
                    else{
                           u -> attack(i);
                    }
             }
     }
}
```

Figure 30: Pseudocode for the Fight unit action. A script that targets the closest enemy unit for attack. Checks must be made ensure a new attack command does not override an existing command frame, otherwise an attack may be unintentionally cancelled.

```
//for a current unit $u$:
void BWNEATUnit::Retreat(){
     if(we have issued a command to unit $u$ already in this frame){
               return;
     }
     else{
               initialize an empty flee vector $f$;
               foreach(enemy unit $i$ in enemy units){
                      create vector $iu$ from $i$'s position to $u$;
                      //weigh the influence of closer enemy units higher
                      divide $iu$ by length squared of $iu$;
                      add $iu$ to $f$;
               }
               find nearest terrain obstacle $w$;
               create vector $wu$ from $w$ to $u$ and normalize by distance;
               add $wu$ to $f$;
               normalize $f$;
               /** Create a position $p' = p + (f * 80)$, that is unit $u$'s current position
               $p$ moved in the direction of the flee vector $f$ by some static distance
               **/
               create $p'$;
               if (the last command for unit $u$ was to move to $p'$){
                      return;
               }
               else{
                      $u$ -> moveTo($p'$);
               }
     }
}
```

Figure 31: Pseudocode for the Retreat unit action. A script that calculates a flee direction that is a sum of weighted vectors of enemy unit positions to the current unit's position. The unit is moved in this direction by a static distance, if it does not interrupt a previous command.

Figure 30 and Figure 31 summarizes pseudocode for the Fight and Retreat actions for the initial model. In both cases, it must check that a previous command has been completed before pursuing a new one, otherwise commands may be constantly interrupted by each other, causing a jittering behaviour in units that does not allow them to get anything done.

The directional granularity model (section 5.2.3) adds further complexity to the implementation of unit percepts and actions. The main addition is the calculation of the direction of enemy units around a given unit, and mapping this to one of eight neural network input nodes. The output also mirrors this mapping into one of eight retreat movement directions. This is summarized as follows:

```
//for a current unit u:
void BWNEATUnit::InputDirectionalPercepts(){
      foreach(enemy unit i in enemy units){
            create vector iu from i’s position to u;
            //using std::atan2
            get the arc tangent angle of iu;
            /** Retrieve node associated with the angle. Nodes from 1 to 8 are
            mapped around a unit circle starting with node 1 aligned to the positive
            X axis, and going anti clockwise. Each node has an associated 45 degree
            arc which the angle of iu can fall under. **/
            retrieve node n associated with iu’s angle;
            /** Generate input ni for node n based on the distance of enemy unit i
            to unit u.**/
            generate input ni for node n;
            /** Because multiple enemy unit i’s can be in the same direction, the
            input that is highest (most dangerous) is chosen. **/
            if (n does not already have an input or input is less than ni){
                  change n’s input to ni;
            }
      }
}
```

Figure 32: Pseudocode of how directional inputs are calculated and mapped to an associated input node.

Figure 32 summarizes the pseudocode for calculating and mapping the direction of enemy units in respect to a given unit, for the directional granularity model. The value of inputs are based on the distance of the enemy unit to the current unit as described in section 5.2.3. The angle to node mappings using the arc tangent function is summarized as follows:

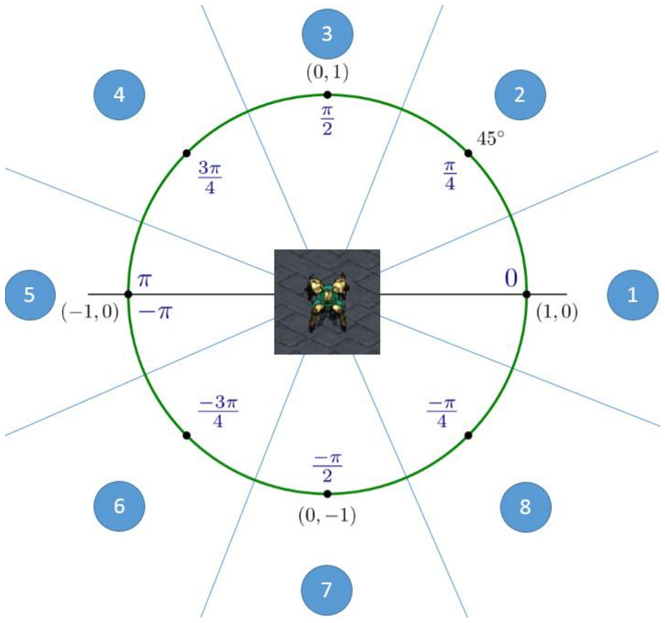


Figure 33: A summary of the 8 directional nodes mapping using the arctangent function. Blue circles indicate nodes.

This mapping is also used for the directional outputs, except that the node with the largest output is chosen. A unit retreats in the angle determined by this mapping, similar to in Figure 31, except that the flee vector is determined by the angle mapping result.

5.5.4. Fitness Evaluation

The fitness function discussed in section 5.5.4 is primarily based on the amount of damage a unit receives from and dealt to enemy units. Damage received is the difference between a unit's current and starting HP (and shield points for Protoss units). This is easily implemented via BWAPI calls such as **`BWAPI::Unit::getHitPoints()`** and by storing a units starting HP and shields at the initialization of the BWNEATUnit collection. However, there is no inbuilt library call or straightforward way to query a unit's damage dealt over time. This requires some estimation techniques based on observing game states.

The first attempt to estimate a unit's damage output was done by observing when a unit's weapon cool down resets. Whenever this happens, it is assumed that the unit has landed a successful attack. In tracking and counting the number of attacks at any time, the unit's damage dealt can be estimated by multiplying with its weapon damage. The problem with this estimation is that attacks may not have landed successfully, for example range projectiles can miss due to terrain elevation differences or just by random chance[18]. Another problem is that weapon and unit types affect the actual damage received by the enemy unit, for example a Dragoon unit only deals half damage to Zealot units (explosive damage type deals half damage to small sized units[19]). Furthermore, in the case of multiple units attacking a single low HP enemy unit, all units would count their attack as successful even though only one of the attacks actually reduced the enemy unit HP to 0.

The second attempt solves these issues by estimating based on observing *Bullet* objects in the game environment. On every game frame, a collection of observable Bullet objects (weapon projectiles) can be queried, which gives information about the source unit, the target unit, and the weapon type that fired the bullet. The bullet returns null for its target unit if the bullet is a miss, or the unit is no longer accessible (destroyed). It is then possible to assign the correct amount of damage dealt for any weapon types and on any unit types, while avoiding attack misses or attacks on destroyed units. The only exception are melee units such as Zealots, which do not generate Bullet objects when they attack, so their damage dealt is estimated with the previous weapon cool down method. However, melee attacks do not miss, so the method is relatively accurate for estimating melee damage.

With these techniques, a unit's fitness can be calculated at any point in time during the running of the system. This is important to support different epoch cycles for generational and real time NEAT runs. For rtNEAT runs, each unit's fitness is calculated on every epoch interval (section 3.6) during real time game frames. For generational NEAT runs, fitness evaluation does not occur until a match has ended (section 3.5).

5.6. Summary of Design and Implementation

This chapter began with discussions on designs and models for a micromanagement agent controlled via neural networks. Firstly, a one unit to one neural network mapping was discussed to be advantageous for evolution using the NEAT methodology. Secondly, a number of network designs were considered, differing in the number and types of game state information to be incorporated as neural network inputs, as well as unit actions as outputs. Lastly, the fitness function used to guide evolution was defined and discussed, as well as a number of NEAT specific parameter values.

[18]Liquidpedia: The StarCraft Encyclopedia. http://wiki.teamliquid.net/starcraft/Damage#Misses

[19] http://wiki.teamliquid.net/starcraft/Damage#Explosive_Damage

The implementation of the agent was also discussed in detail, including its architectural integration of the BWAPI framework and NEAT open source code. The inner workings of the agent system was further elaborated in an examination of its three main processes: an initialization of unit and neural network pairs, on frame updating of unit percepts and querying neural network controllers, and the collection of statistics and saving of network results. Further details of more specific parts of the system, such as how the unit percepts and actions are implemented, as well as how the fitness evaluation is calculated was also discussed.

In the next chapter, I discuss a number of experimentations conducted to validate the performance of the designs discussed previously. The implementation of the agents used in these experimentations are as those discussed in this chapter.

Chapter 6

EVALUATION

In this chapter, I discuss a number of experiments aimed at evaluating the effectiveness of the proposed AI system in Chapter 5. Experiments are grouped into two rounds, with the first examining the proposed initial agent neural network model (section 5.2.1). The result of these experiments lead to the extension of the model to include directional input (section 5.2.3), and a second round of experimentations for comparison. A primary goal behind these experiments is a comparison of the rtNEAT and generational NEAT algorithms. Specifically, in their evolutionary performance over time as well as their effectiveness at generating successful neural network controllers for micromanagement.

The primary measure in these experiments is the performance of the NEAT evolved AI agent against the default SC: BW AI. The default AI consists of a static unit behaviour script built-in to the SC: BW game. There are a number reasons for employing this measure. Firstly, this method of evaluation is employed in the majority of previous work on micromanagement AI. Secondly, the alternative measures (playing against real human players, or other AI agents) are problematic and difficult to set-up. I discuss the limitations of this benchmark and the alternatives later in section 7.2.

The parameters used for NEAT and rtNEAT are as those discussed in section 5.4. These were determined by examples in the literature and by informal tests using the experiment setups described below. The values for the parameters remain constant throughout each of the experiments.

6.1. INITIAL MODEL EXPERIMENTS

Experiments were first conducted using the initial model, to gauge the effectiveness of NEAT and rtNEAT evolved micromanagement agents in SC: BW. In these experiments, the NEAT and rtNEAT-trained agent AI played micromanagement battles against the SC: BW built-in AI. The initial focus was on the performance of each algorithm when left running over generations of evolution, in terms of agent win rate and network fitness. A variety of unit setup was used in order to simulate different micromanagement scenarios. The result of these experiments showed very high fluctuation and variation in the fitness and win rate of agents over generations. In order to adjust for these fluctuations, further experiments were conducted to show the number of generations taken for each algorithm to converge to a suitable solution, when evolution is halted upon finding a potential candidate.

6.1.1. Unit Set-Up

As mentioned in section 4.3, units vary based on a number of attributes, such as the race, weapon type and HP. In order to keep the experiment variables constant and to avoid an explosion of unit type permutations, the experiment setup is based on Gabriel et al., (2012) which compared 4 unit type variations (Zealots vs. Zealots, Dragoons vs. Dragoons, Zealots vs. Dragoons and Dragoons vs. Zealots). These variations require different types of agent behaviour to be evolved, in order for the agent to achieve a high win rate. For example as mentioned previously, a ranged unit has an inherent advantage against a melee unit if it adopts a hit-and-run strategy called kiting. While a Dragoon unit deals less damage to a Zealot unit and has a higher weapon cool down, it has a superior weapon range and should be advantageous if it employs kiting. On the other hand if the agent is controlling Zealots, it should be aggressive and quickly advance against the Dragoons to reduce their weapon range inferiority. Figure 34 summarizes the unit attributes for Zealots and Dragoons.

Unit Name	HP	Shield	Damage	Range	Cool Down
Dragoon	100	80	20*	128	30
Zealot	100	60	16	15	22

Figure 34: Main attributes for units used in initial model experiments. *Dragoon attack damage is halved against Zealots.

6.1.2. Scenario Set-Up

The scenario map used throughout experimentation is a simple and flat map with no elevated terrain (64 by 64 grid size), based on those used in the AIIDE 2010 StarCraft micromanagement tournament[20]. This is because the current models do not take elevation into consideration in its state inputs. Fog of War is

[20] AIIDE2010 micromanagement tournament: http://eis.ucsc.edu/StarCraftTournament1

enabled on the map, preserving the incomplete information attribute of the game which can hide enemy units outside of unit vision. The number of units is kept at constant 12 versus 12, which is the maximum selectable number of units for a human controlled squad. The enemy and ally squads are positioned in equal and symmetrical starting positions with a static distance away from each other, such that both sides are able to see some units of the enemy at the beginning of a match, but are not immediately in weapon range of each other (Figure 35). This setup is used to reduce unit starting positional advantages while allowing Fog of War, because if the default AI does not have vision of enemy units, it tends to break apart its squad into a disadvantageous layout while in search of the enemy.

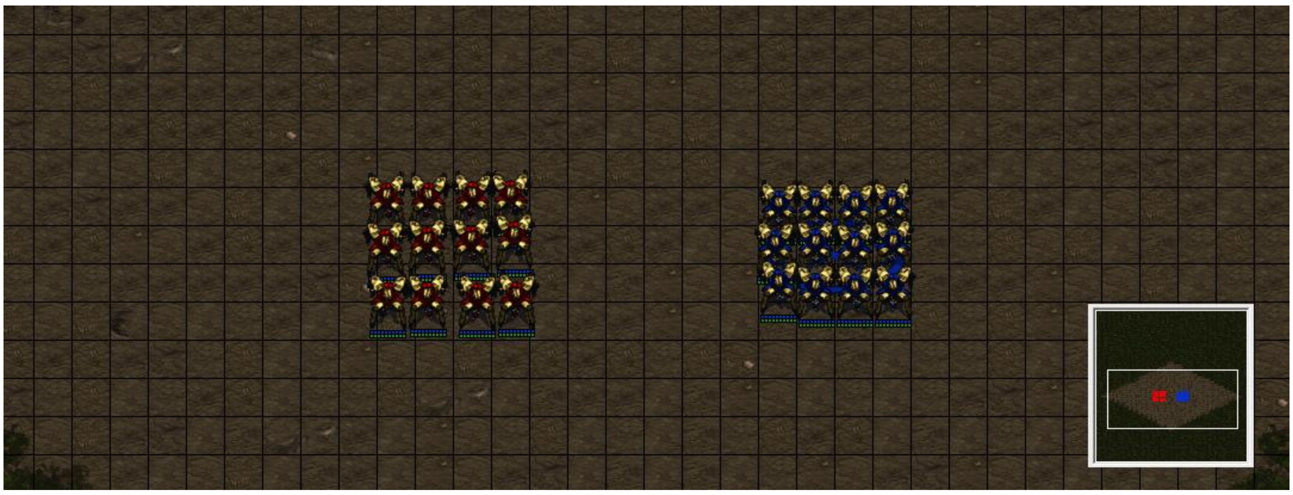


Figure 35: A screenshot of an example starting scenario map with unit layout (Dragoon vs Dragoon). Depicted bottom right corner is the minimap view of the entire map.

6.1.3. Technical Setup

All experiments are run with the following set-up:

- Intel i5-3550 Quad Core CPU clocked at 3.30 GHz with 8 GB RAM
- Windows 7 64-bit OS
- StarCraft: Brood War version 1.16.1
- Visual Studios 2012 for compiling and live debugging
- rtNEAT C++ version 1.0.2
- BWAPI C++ version 3.7.4
- Chaoslauncher version 0.5.5 for injecting BWAPI DLLs into SC:BW

During experimentation runs, BWAPI is configured for automatic and uninterrupted repeat runs of the same scenario map. For maximum running speed, the game speed is set to 0 which skips all delays between game frames, and the GUI and sound engines are also disabled. This ensures that as many experimentation runs can be completed as quickly as possible to allow for hundreds of generations of learning. After a match is finished, statistics such as match result and average unit fitness are recorded to a text file. The resulting population of neural networks are also serialized to a formatted file to be loaded back in at the beginning of the next match. When experimenting on both generational NEAT and rtNEAT, the matches are run in sequence, switching between matches of rtNEAT and NEAT one at a time.

6.1.4. Evolutionary Process Experiment

The first experiment compared the performance of the NEAT and rtNEAT algorithms on each of the 4 unit matchup variations. Each unit matchup is played over 300 matches of evolution, with the NEAT or rtNEAT agent controlling one side and the default SC: BW AI controlling the other. In the first match, the population is initialized with a basic neural network structure with no hidden nodes and random starting weights for all connections (each of the 12 units have the same neural network structure but with different random starting weights). For rtNEAT, evolution occurs in real-time over the 300 matches, while for generational NEAT, evolution occurs after each match. A single run of the experiment is the completion of 300 matches, upon which the population is reset to the starting basic structure with random weights. This 300-match run is repeated 25 times in order to adjust for the randomness in starting weights in each run. The average fitness of all units and the match outcome is recorded over each 300-match run, and averaged over the 25 runs.

The experiment ran successfully and terminated after a week of computation, with minor problems. The experiment had to be manually restarted (rebooting BWAPI and SC: BW) on occasion due to SC: BW crashing from memory issues when left running for extended matches, and also due to some memory leaks from the AI module itself. Transitioning between units match-ups were also done manually. Overall the results are summarized in the following figures:

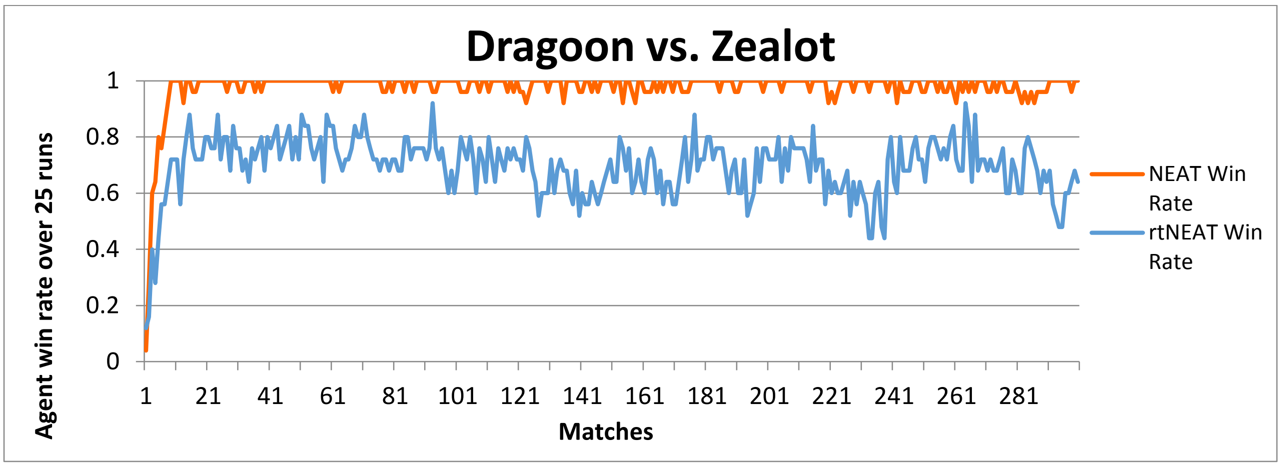


Figure 36: Plot of range unit Dragoon (NEAT agent) vs. Melee unit Zealots, mean win rate for 25 runs over 300 matches.

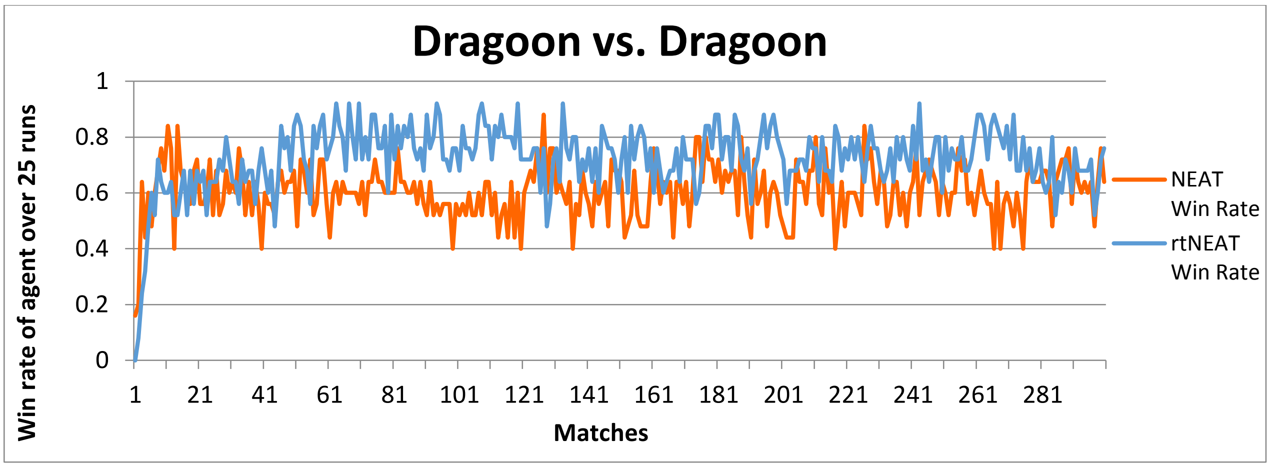


Figure 37: Plot of Dragoon vs. Dragoon mean win rate for 25 runs over 300 matches.

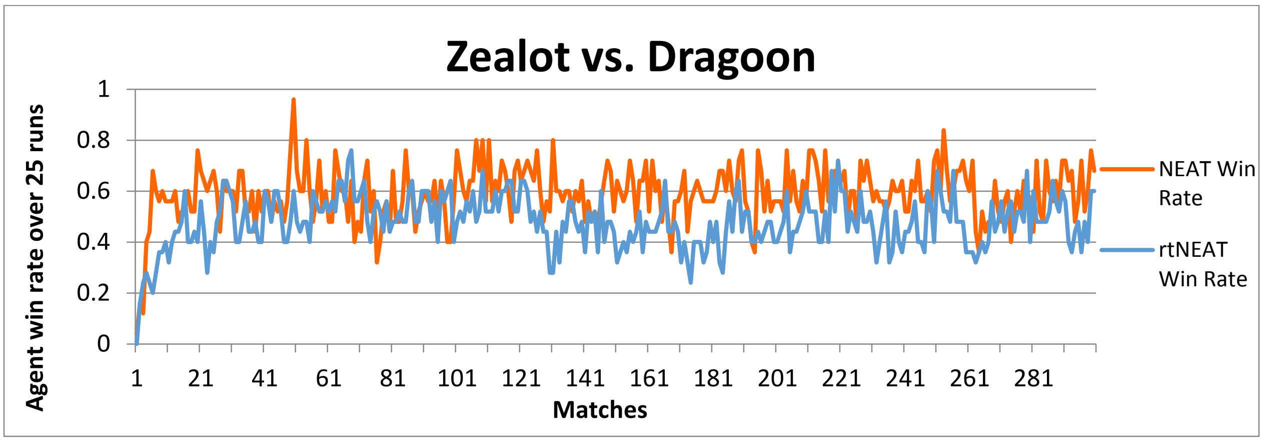


Figure 38: Plot of Zealot (NEAT agent) vs. Dragoon mean win rate for 25 runs over 300 matches.

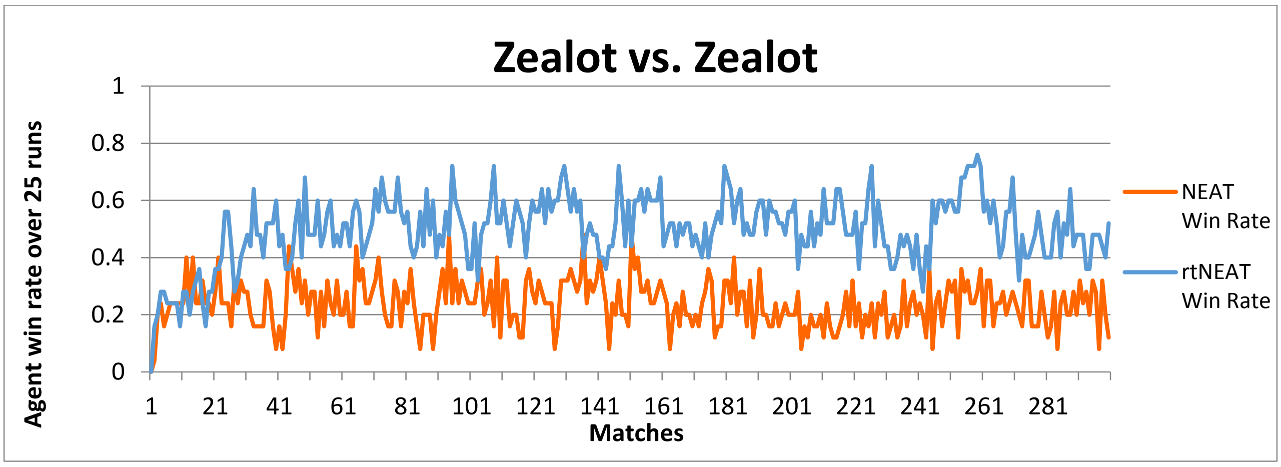
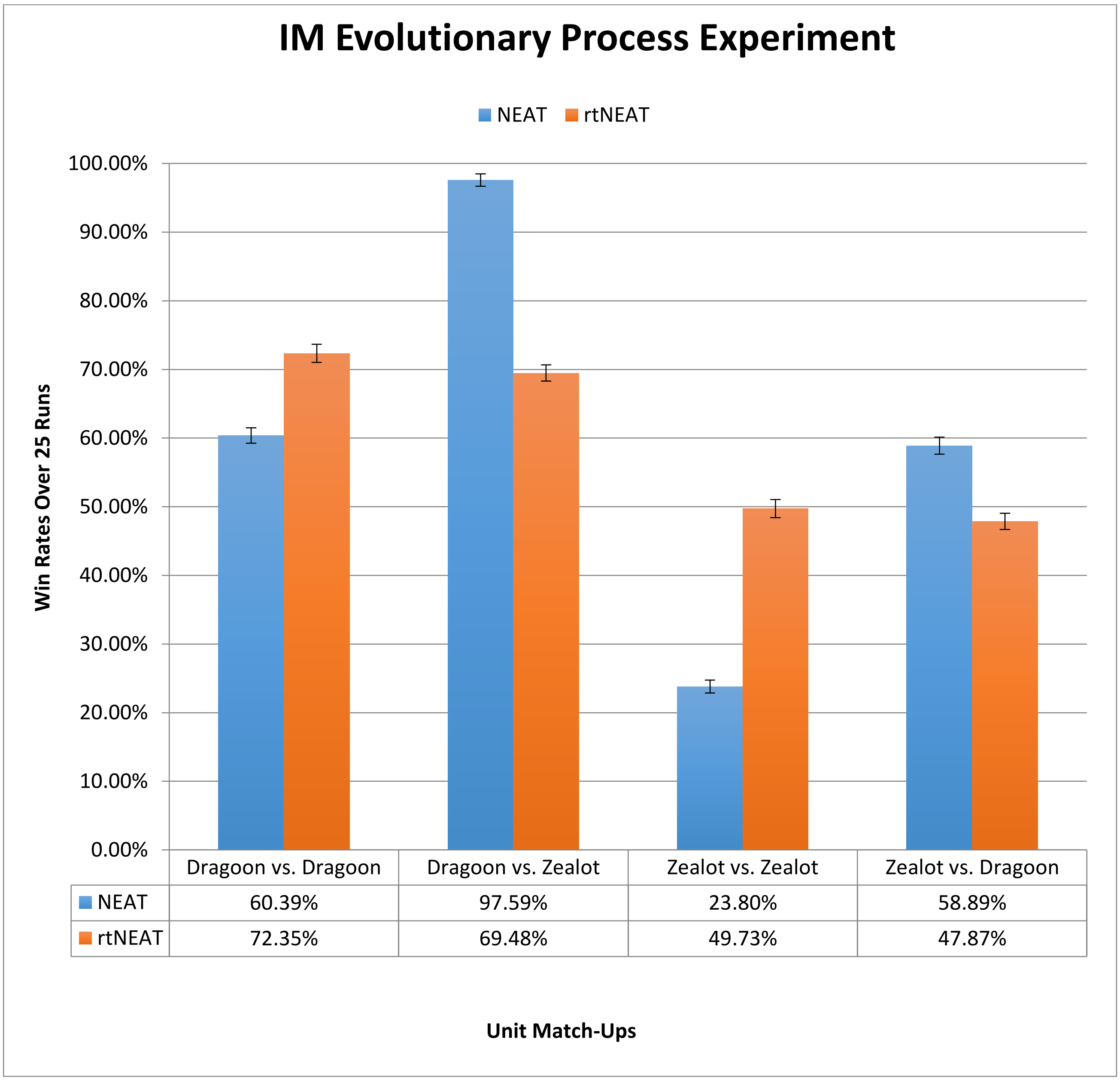


Figure 39: Plot of Zealot vs. Zealot mean win rate for 25 runs over 300 matches.



	Dragoon vs. Dragoon	Dragoon vs. Zealot	Zealot vs. Zealot	Zealot vs. Dragoon
🟦 NEAT	60.39%	97.59%	23.80%	58.89%
🟧 rtNEAT	72.35%	69.48%	49.73%	47.87%

Figure 40: Summary results for evolutionary process experiment using the initial model (IM). The column graph plot shows the mean win rate categorized by unit match up. The error bars indicate the 95% confidence interval for the mean.

The summary results (Figure 40) suggest that there is no single algorithm dominating all of the match variations. The mean win rate is higher for NEAT on Dragoon vs. Zealot (97.59% mean, 7.95% SD) and Zealot vs. Dragoon (58.89% mean, 10.79% SD), while rtNEAT is higher on Dragoon vs. Dragoon (60.39% mean, 9.86% SD) and Zealot vs. Zealot (49.73% mean, 11.63% SD).

If we consider a win rate higher than 50% to indicate better than baseline performance against the built-in SC: BW AI, then both NEAT and rtNEAT show effectiveness on Dragoon vs. Dragoon and Dragoon vs. Zealot battles. NEAT is also effective in Zealot vs. Dragoon (58.89% mean ± 1.23% at the 95% confidence level). Neither algorithm were able to have higher than 50% win rate on Zealot vs. Zealot, which indicates it is the most difficult matchup for the agent. The reason that a win rate above 50% indicates the agent's effectiveness is that it implies it has an edge over the default SC: BW AI. These are statistically significant results with minimal differences at the 95% confidence level (for all results greater than 50%, the 95% CL does not drop below 50% at the lower confidence bound). Note that even a win rate at 50% is far greater than random, since randomized unit control would rarely achieve a win against the default AI.

For both algorithms, the main advantage against the default AI occurs when controlling Dragoons (ranged unit). From examining the agent behaviour, the effectiveness of controlling Dragoon units can be attributed to having learnt the hit-and-run micromanagement strategy (kiting), both against melee units and also against other Dragoon units. It is interesting to note the generally poor performance of both algorithms when controlling Zealot (melee) units. This is discussed further in section 6.1.6.

Figure 36 to Figure 39 show the plots of average win rate over generations of matches for all match variations. The average fitness plot is not shown, but is highly correlated to the average win rate. From these plots, we see a trend of initial poor performance and a quick convergence to some local optima. This is typical of evolutionary algorithms where the initial starting solutions are randomized and are not expected to perform well, but quickly converge towards local optima.

On all variations the first convergence to a local optimal occurs between the 10th to 20th matches. From then on, there is a trend of significant fluctuation of win rate, which is further illustrated by the standard deviation of each match up (with values around 10%). With the exception of the Zealot vs. Dragoon matches, the standard deviation is higher with the rtNEAT algorithm, suggesting a greater fluctuation of evolutionary success over generations than the NEAT algorithm. This may be due to the nature of the rtNEAT algorithm, in introducing evolutionary changes to the population in real time, which is faster than generational NEAT.

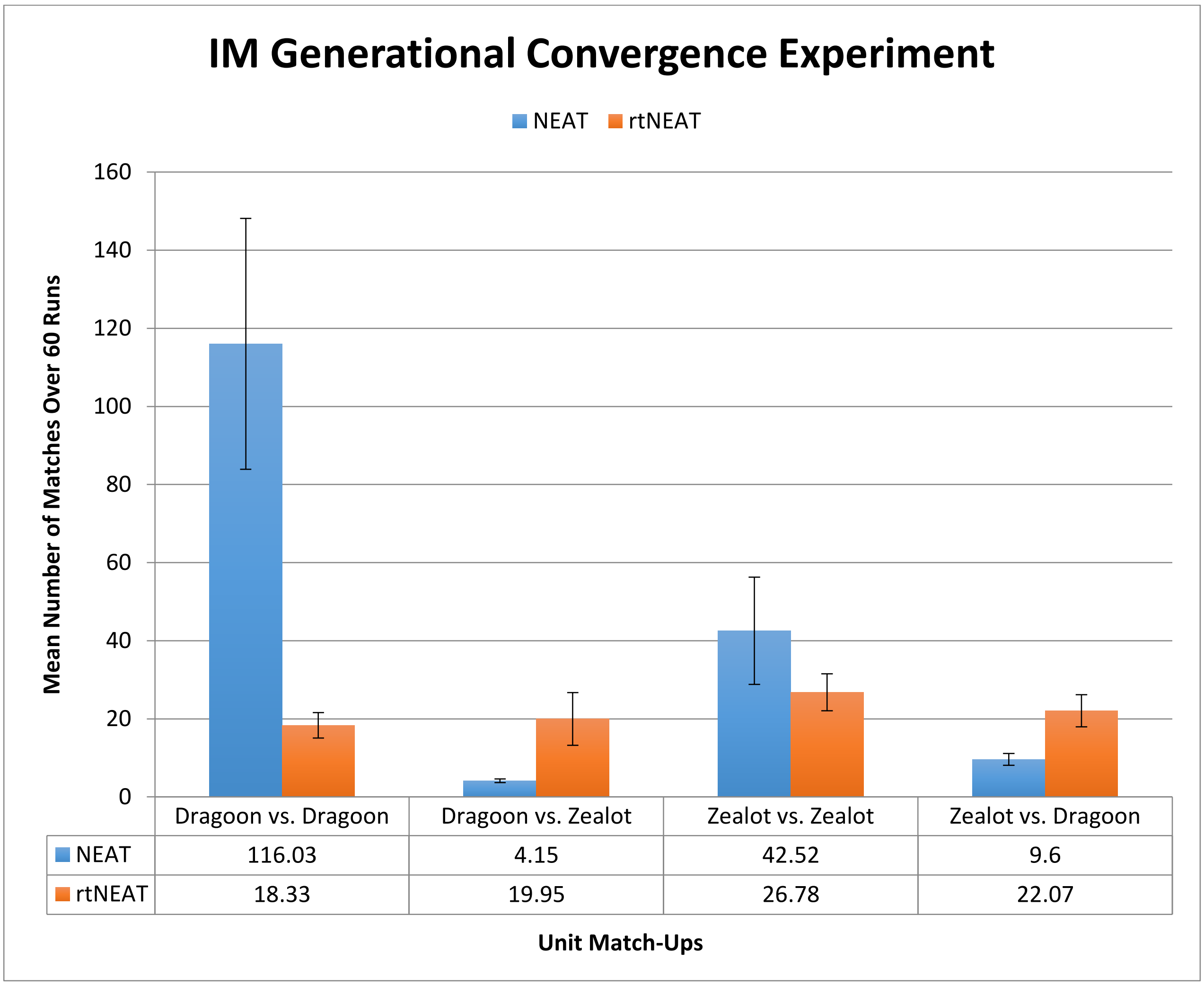
From this experiment it can be seen that both algorithms are able to produce high performing solutions (above and beyond 50% win rate), but are also quick to introduce mutational changes that weaken the solutions. This is partly due to the nature of the experiment where evolution is allowed to continue even after achieving a winning solution. If winning behaviour is explicitly preserved, such that no changes occur in candidate solutions, then their effectiveness can be better gauged. In the next experiment, I analyse the number of generations it takes for each algorithm to converge to a successful solution.

6.1.5. Generational Convergence Experiment

By defining a winning criterion, the evolution can be halted in both the NEAT and rtNEAT algorithms when a candidate solution is first found. A winning solution is defined to be an agent that achieves 10 consecutive wins as an indication of success (the probability an agent with at most 50% win rate can win 10 consecutive games is < 0.1%). This is a strict criterion as an agent may still be high performing even though it loses 1 game out of 10, for example due to the stochastic nature of the game states. However it is used to simplify the running of the experiment and to show it is possible to robustly generate agents of this level of performance.

For this experiment, all variables are kept the same as in the previous experiment (agent model, unit set-up, scenario set-up, technical set-up) except for the running procedure. Instead of a single run of 300 matches, a run in this experiment is defined as winning 10 consecutive matches, or failing to do so after 1000 matches. On encountering a win, evolution is halted for the agent so that its neural network population is unchanged for the next match. If the agent encounters a loss, evolution continues as normal. After 10 consecutive wins, or if not successful after 1000 matches, the experiment is reset to an initial population with randomized weights for the next run. For each algorithm and each matchup, the experiment is stopped and analysed after 60 runs. The experiment successfully ran and terminated after 4 days of computation with minor manual restarts (similar issues to previous experiment).

Figure 41 summarizes the results of this experiment. In all experimental runs, an acceptable solution (10 consecutive wins) could be found before 1000 matches, with most solutions converging in less than 100 matches. Once again there was high variability in the results, this time in the number of matches taken to arrive at an acceptable solution. This is evident in the high standard deviation in some match ups (e.g. 124.39 SD and 116.03 Mean for NEAT Dragoon vs. Dragoon, 26.03 SD and 19.95 Mean for rtNEAT Dragoon vs. Zealot). The mean number of matches taken between NEAT and rtNEAT is comparable to the average win rate performance of the previous experiment: NEAT takes fewer number of generations for Dragoon vs. Zealot and Zealot vs. Dragoon match ups, while rtNEAT takes fewer in Dragoon vs. Dragoon and Zealot vs. Zealot match ups.



	Dragoon vs. Dragoon	Dragoon vs. Zealot	Zealot vs. Zealot	Zealot vs. Dragoon
🟦 NEAT	116.03	4.15	42.52	9.6
🟧 rtNEAT	18.33	19.95	26.78	22.07

Figure 41: Summary results for the generational convergence experiment on the initial model (IM). The mean number of matches taken to produce an acceptable solution for each algorithm is plotted on the column graph categorized by unit match up. The error bars indicate the 95% confidence interval for the mean.

There is a much higher range in the number of matches taken between different matchups for NEAT (4.15 mean for Dragoon vs. Zealot and 116.03 mean for Dragoon vs. Dragoon) than for rtNEAT (18.33 mean for Dragoon vs. Dragoon and 26.78 mean for Zealot vs. Zealot). This suggests the performance of rtNEAT is more stable and robust under different unit variations. Overall, the experiment showed that both algorithms were capable of generating effective solutions for different unit variations in the task of micromanagement against the default SC: BW AI. In order to do so, it is necessary to establish an acceptance criterion for which to halt evolution and to preserve winning behaviour. In the next section, some of the observations and implications on the above experiments are discussed.

6.1.6. Unit Behaviour and Network Complexity Observations

In the evolutionary process experiment, the fluctuation of fitness and success rate of solutions can be due to a number of reasons. Firstly, it suggests that any structural innovations introduced were making significant differences in the performance of the neural networks. This is probably due to the simplicity of the network design, where only 2 outputs exist, such that any structural change may affect the action selected. A simple neural network allows faster convergence by reducing the search space of initial nodes and weights. But it also means it is faster to diverge from the local optima. On top of this, the stochastic nature of the game environment can result in the same solution having varied success over different runs.

This also explains the general poor performance of both algorithms on Zealot match ups in the evolutionary process experiment. Melee units do best in direct attack as they lack the weapon range to perform hit-and-run manoeuvres. Any innovation introduced to make melee units retreat will immediately reduce the success rate of the solution. In the second experiment, the algorithms have no problem generating a solution for melee match ups, when no new innovations were introduced after a solution begins to do well. However, it is still slower and thus more difficult to converge on good unit control behaviour for melee units, than with range units.

Another factor is an interesting behaviour exhibited by the units over generations of evolution: some units are evolved to retreat when enemies are first found, but come back to fight after allied units are engaged in combat. These units tend to generate more fitness than those directly attacking from the beginning, as they do not receive as much damage over time. However, as the population begins to favour this behaviour, there is a breaking point in which no units will stay to fight, leading to a match loss and a return to evolution favouring units that do not retreat. This cycle is highly correlated with the fluctuation of win rate over matches.

Interestingly, the first experiment showed that rtNEAT produced higher variation in success of solutions than NEAT over time. However in the second experiment, the average number of matches for an acceptable solution was less varied across different match ups than NEAT. This suggests real-time evolution can be quicker in introducing changes that reduce fitness, but also allow a more robust convergence to a solution regardless of unit variation (variability in state and solution space). This is intuitive, since rtNEAT should be faster in reacting to changes in the environment in real time, than regular NEAT evolution between matches.

From these experiments it was hypothesized that modifying the network complexity would induce a training time versus solution robustness trade-off. For example, the inputs can incorporate a deeper ontology of unit quality and type variations (armour, weapon and ability types etc.) and more precise directional and distance data. Instead of fight or retreat actions, the decisions can be to move at specific angles for specific distances, and to explicitly decide which units to attack. The advantage of more complicated neural network designs should be agents with more complicated behaviours that are able to perform well under a higher variety of conditions. The disadvantage is in a greater number of dimensions to search and optimize for, resulting in slower training time. In the next section, I discuss experiments for an initial attempt to explore increasing network complexity

6.2. Directional Granularity Experiments

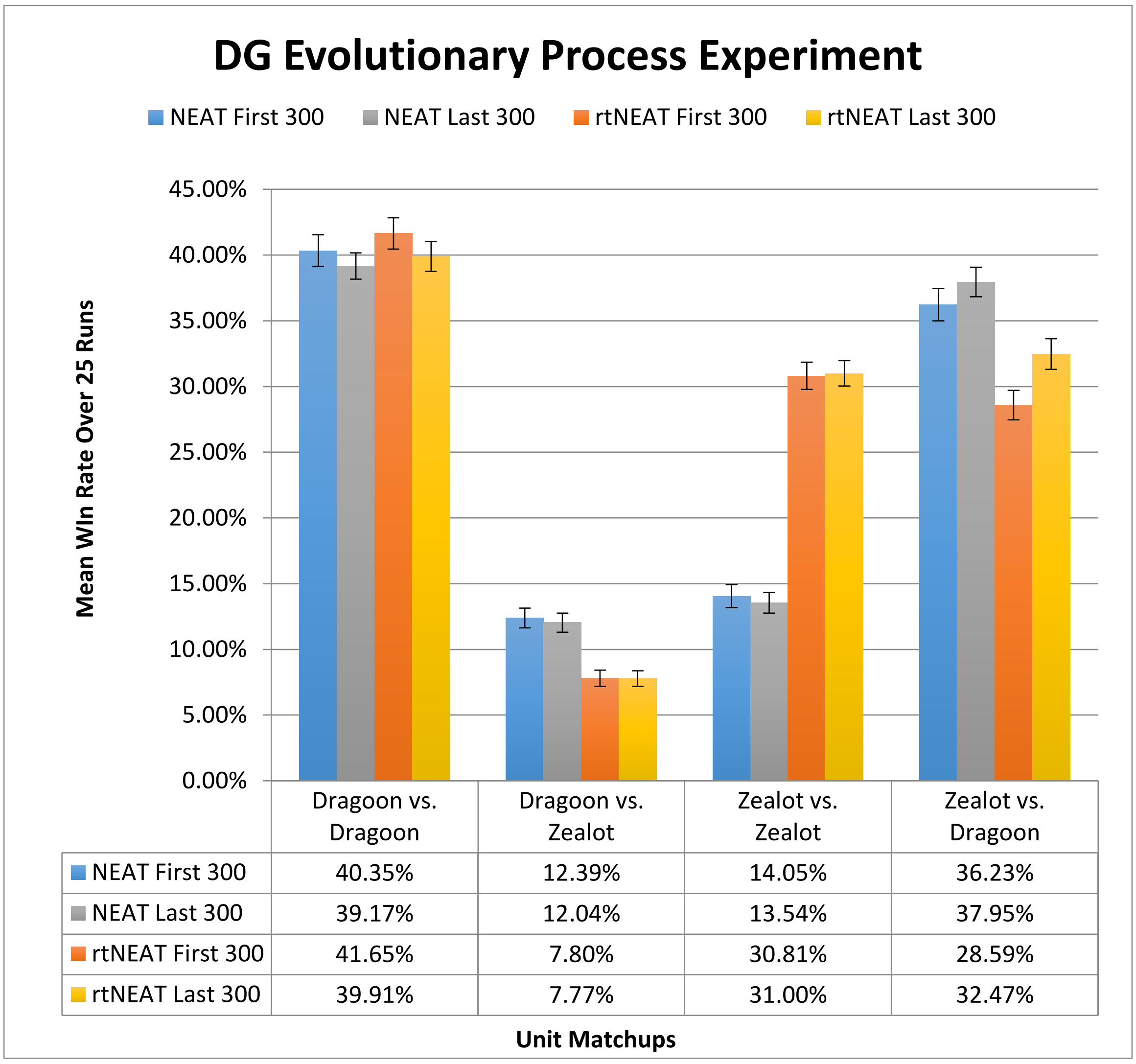
In order to explore increased network complexity, the AI agent was adapted to use the directional granularity neural network model discussed in section 5.2.3. This was done by modifying the starting network topology loaded at the beginning of a match, and implementing directional unit percepts and actions discussed in section 5.5.3. The main difference in this model is the replacement of the retreat subroutine and network output with a number of directional information input and outputs. In effect, it was an attempt to allow the agent to learn the directions to retreat on its own, as well as exploring increased network complexity.

Experiments conducted on the directional granularity model were similar to those done on the initial model. First, a version of the evolutionary process experiment was run, with an extended number of matches per run. The results showed similar fluctuations to the initial model experiments, even through extended matches. Secondly, the generational convergence experiment was also conducted, with the same setup as previous. The results were analysed and discussed in comparison to the first model.

In order for results to be comparable to the initial model experiments, controlled variables must be held constant. Therefore, the unit, scenario and technical set-ups remain the same as those described in the previous experiments. The NEAT parameter values also remain unchanged throughout.

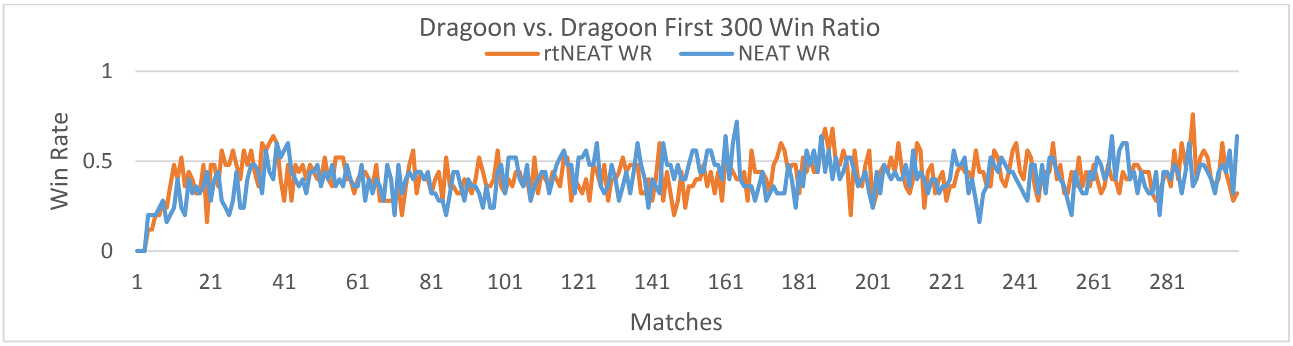
6.2.1. Evolutionary Process Experiment 2

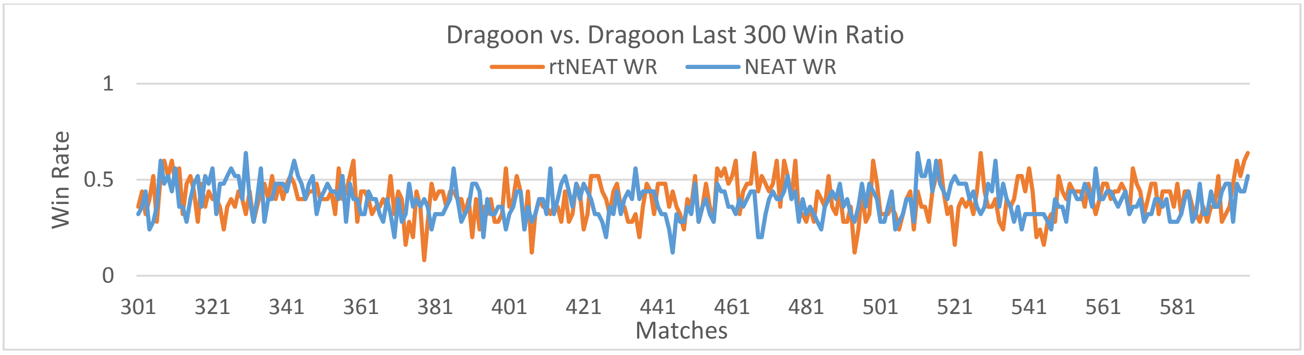
Experiments on the directional granularity model began with the same evolutionary process experiment as in the previous model, except with an extended number of matches. Each unit match-up run consists of 600 matches instead of 300 as in the previous model. This was done because I hypothesized that the extra complexity of the model would require more time for convergence. By extending the number of matches, the results can still be directly comparable between the first 300 matches, and also provide some insight into the models performance after the extended matches. Each unit match up was run 25 times and the results recorded as in the previous experiments. The experiment running time exceeded double the time taken in the previous model (around 16 days) and were successfully completed with minor manual restarts. The results are summarised below:

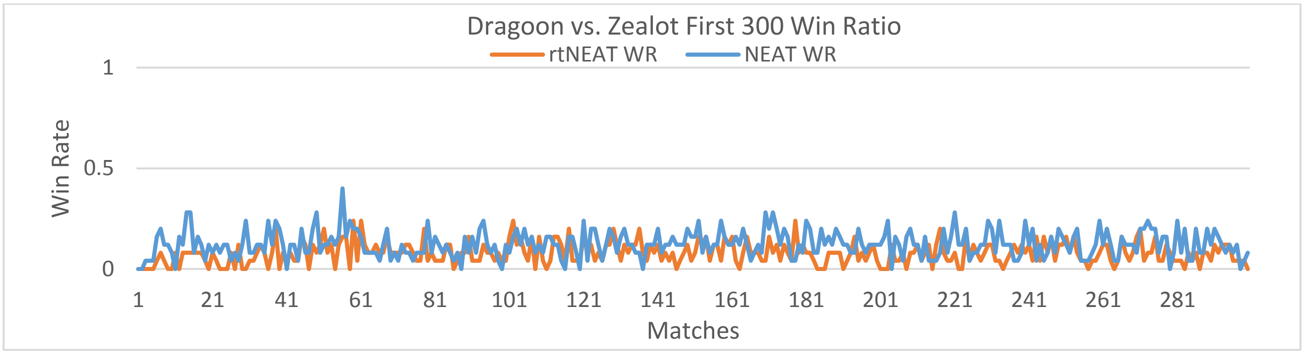


	Dragoon vs. Dragoon	Dragoon vs. Zealot	Zealot vs. Zealot	Zealot vs. Dragoon
🟦 NEAT First 300	40.35%	12.39%	14.05%	36.23%
⬜ NEAT Last 300	39.17%	12.04%	13.54%	37.95%
🟧 rtNEAT First 300	41.65%	7.80%	30.81%	28.59%
🟨 rtNEAT Last 300	39.91%	7.77%	31.00%	32.47%

Figure 42: Summary results for the evolutionary process experiment using the directional granularity (DG) model. A column graph categorized by unit matchup shows the mean win rates of each algorithm in the first and last 300 matches. The error bars show the confidence interval for the mean.







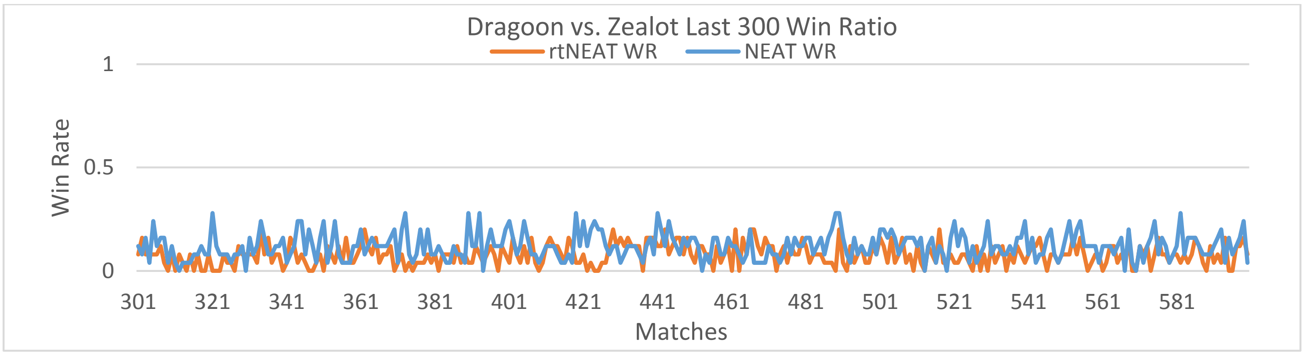


Figure 43: Plots of the best (Dragoon vs. Dragoon) and worst (Dragon vs. Zealot) performing unit match ups. Plots are categorized by unit match-up and by first and last 300 matches. Each plot indicates the mean win rate of both the NEAT and rtNEAT algorithms over 300 matches.

The first thing to note is that the results are generally poor (all below 50% win rate) for all algorithms and unit match-ups (Figure 42). There is no significant difference between the performance of the first and last 300 matches (except for a small but significant difference for rtNEAT on Dragoon vs. Zealot), which suggests the model generally does not converge better with extra training. I confirmed this point by running single runs of 10,000 matches for each algorithm and each match-up, and which indicated the same poor performance even for long extended evolution periods.

The best performing unit match-up was Dragoon vs. Dragoon, at around 40% win rate for both algorithms and on both the first and last 300 matches (Figure 42). The worst performing was the Dragoon vs. Zealot match-up, where the algorithms performed at 7% - 12% win rate. NEAT performed significantly worse than rtNEAT on Zealot vs. Zealot (around 14% for NEAT and 30% for rtNEAT), while performing better on Zealot vs. Dragoon (NEAT at 36% - 38% and rtNEAT at 28% - 32%). NEAT also performed better on Dragoon vs. Zealot, while there is no clear winner on the Dragoon vs. Dragoon match-ups.

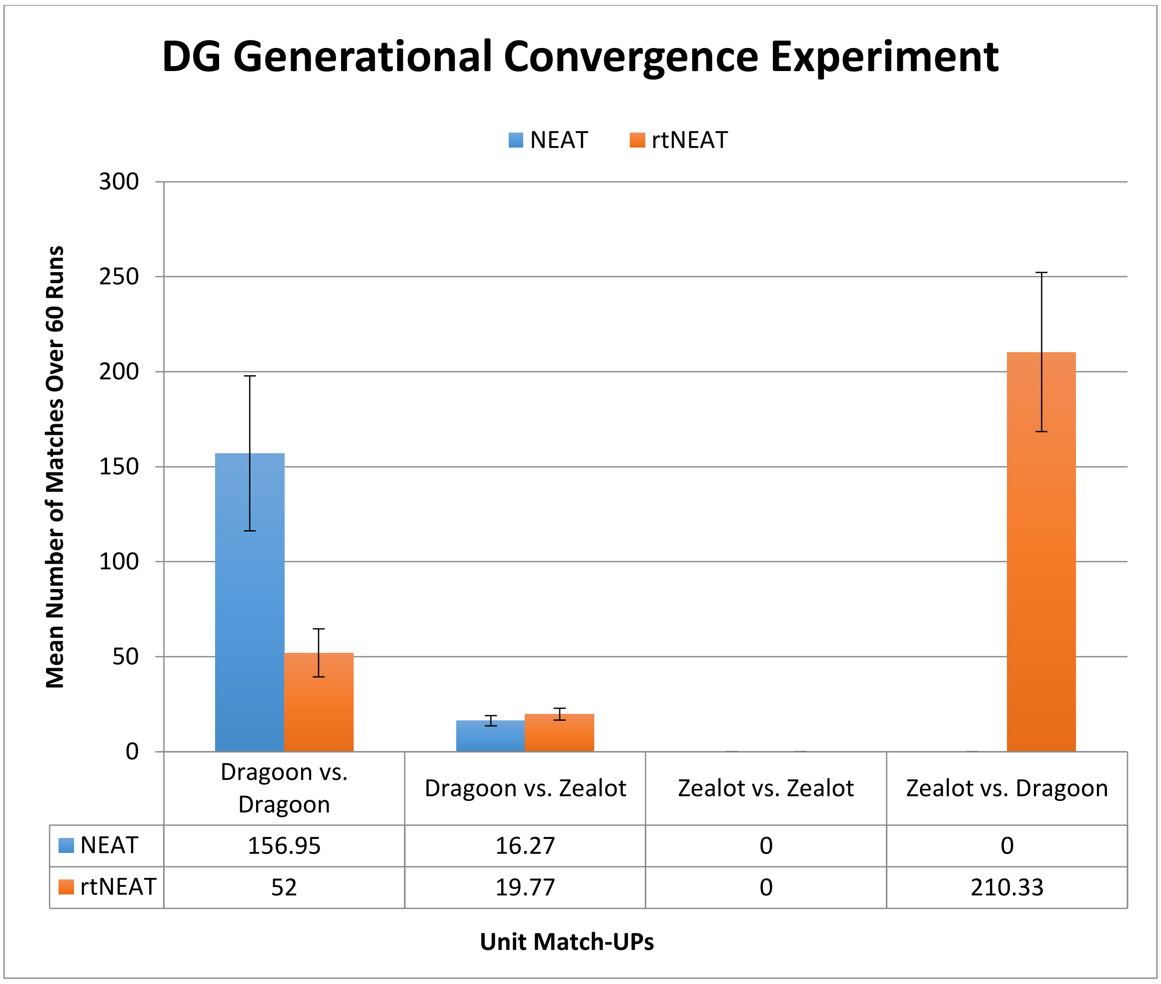
Figure 43 depicts the win-rates of the best and worst performing unit match-ups during the first and last 300 matches. The first 300 matches of the Dragoon vs. Dragoon match-up (best performing) shows the same quick convergence in the first 10 to 20 matches (starting from the default randomized weight topology) as in the initial model experiments. Also similar to previous experiments is a high fluctuation of win-rate for both NEAT and rtNEAT. There are spikes of up to 72% win-rate for rtNEAT and 74% for NEAT in some matches, but also dropping to below 20%. This fluctuation continues onto the last 300 generations where there seem to be even worse average performance, despite not having the initial poor performance of starting with a default topology (since the last 300 matches continue on from the first 300).

On the Dragoon vs. Zealot unit match-up, it can be seen that the AI could not converge during the first 300 matches, and constantly dropped to 0% win rate throughout matches (Figure 43). There is no significant difference on the last 300 matches either, which suggests that the directional granularity model performs very poorly controlling Dragoons vs. Zealots, even with extended training. Not depicted are the win rate plots for Zealot vs. Zealot and Zealot vs. Dragoon, but their performance is very similar, with high fluctuation over matches and insignificant difference between the first and last 300 matches.

Since there is the observed problem of high fluctuations of win-rate as in the initial model, it is a good candidate for a repeat of the generational convergence experiment. In the next section, I describe the results of this repeated experiment.

6.2.2. Generational Convergence Experiment 2

The generational convergence experiment from the initial model is repeated for the directional granularity model without introducing any changes. Once again the winning criteria is defined as 10 consecutive wins, and a run terminates on reaching the criteria or failing to do so after 1000 generations. Each unit match-up is again run 60 times, with the resulting number of matches taken to reach a winning solution (or a failure) recorded at the end of reach run. The unit, scenario and technical setup also remains unchanged. The experiment successfully ran and terminated after 9 days of computation. The results are summarized below:



	Dragoon vs. Dragoon	Dragoon vs. Zealot	Zealot vs. Zealot	Zealot vs. Dragoon
🟦 NEAT	156.95	16.27	0	0
🟧 rtNEAT	52	19.77	0	210.33

Figure 44: Summary results of the generational convergence experiment on the directional granularity model (DG). The mean number of generations taken to produce a winning solution is plot on a column graph. The error bars indicate the 95% confidence interval for the mean. A 0 value indicates the algorithm was unable to produce a winning solution for at least 1 of its 60 runs (hence not able to calculate a comparative mean).

Match Variation	Success Rate	Max	Min
NEAT Zealot vs. Zealot	0.30%	933	17
rtNEAT Zealot vs. Zealot	0.77%	923	11
NEAT Zealot vs. Dragoon	0.83%	860	9

Figure 45: Results for the unit match up and algorithms that could not generate acceptable solutions for at least 1 of its 60 runs. The success rate (out of the 60 runs), maximum (Max) and minimum (Min) number of generations taken for an acceptable solution over its successful runs is shown.

Unlike in the initial model experiments, the directional granularity model could not consistently generate winning solutions for all unit match-ups. This resulted in significantly longer experiment running time, as more runs had to complete a full 1000 matches. Figure 44 depicts the mean number of matches taken to generate a winning solution for the match-ups and algorithms that were able to do so for all 60 runs. Figure 45 depicts results for those match-ups and algorithms that failed one or more of its 60 runs, in the form of a success rate and maximum and minimum number of matches needed within its successful runs.

Both algorithms were able to consistently generate winning solutions for Dragoon vs. Dragoon and Dragoon vs. Zealot match-ups. NEAT takes significantly more matches than rtNEAT on Dragoon vs. Dragoon match-up, while there is an insignificant difference in the Dragoon vs. Zealot match-up. The lowest number of matches taken for both algorithms was on the Dragoon vs. Zealot match-up, which is surprising as it is the worst performing match-up during the evolutionary process experiment for both algorithms. This suggests that halting evolution is especially important for this match-up for the directional granularity model.

Neither algorithms were able to consistently generate winning solutions for the Zealot vs. Zealot match-up. NEAT is especially worse on this, having only 30% success rate versus rtNEAT at 77% (Figure 45). This is consistent with the evolutionary process experiment, where NEAT has a significantly worse win rate than rtNEAT on this match-up. For the Zealot vs. Dragoon matchup, only rtNEAT was able to consistently generate a winning solution (although taking the highest mean number of matches amongst all unit match-ups to do so). This is in contrast to the evolutionary process experiment where NEAT had a significant win rate advantage over rtNEAT, and suggests rtNEAT benefits from halting evolution more so than NEAT on evolving successful neural network controllers on the directional granularity model. In fact, on all match-ups where rtNEAT performed significantly worse in the evolutionary algorithms, it has taken less matches for rtNEAT than NEAT on the generational convergence experiment (or there is an insignificant difference) to generate a successful solution.

In general, there is a large variation in the number of matches taken to generate successful solutions for both algorithms (confidence intervals at almost 100), except on the Dragoon vs. Zealot match-ups. For the match-ups that the algorithms could not be consistent on, there is a large range of matches taken (maximum over 900, minimum below 10). This and the poor results of the evolutionary process experiment in general suggests unstable evolution and poor performance for the directional granularity model. In the next section, I discuss unit behaviour observations that elaborate the performance of this model, and directly compare the results of this model and the initial model.

6.2.3. Unit Behaviour and Comparing Models

Much of the poor performance of the directional granularity model is elaborated when examining the unit behaviour during evolution. For example it was observed in the initial model that when controlling Zealot units, the best result occurs when the network is trained to pursue aggressive attacking behaviour (section 6.1.6). In the initial model, the worst performing match-ups for the evolutionary process experiment were in controlling Zealot units (Figure 46). This was due to how quickly evolution introduced defensive unit behaviour which reduced the viability of Zealot units (by making units retreat instead of attacking).

The directional granularity model is also affected by this, and is made even worse because the defensive behaviour is error prone. In the initial model, the retreat action is a static script that directs a unit away from enemy units, ensuring that it does not take further damage. This is sometimes advantageous if the unit is low in HP, but disadvantageous most of the time when damage output is a priority. In the directional granularity model, the retreat action moves a unit in a direction based on the network outputs. If there is an incorrect mapping, the unit may move toward enemy units, which will quickly reduce its HP. In short, the model performs even worse than the initial model when evolving defensive behaviour for Zealots, and yet tends to do so as often.

This problem was solved in the initial model when evolution is halted in the generational convergence experiment. Both algorithms were able to successfully generate winning solutions for Zealot unit match-ups under 50 matches (Figure 47). This is not the case for the directional granularity model, where only rtNEAT could consistently generate a winning solution, and only for Zealot vs. Dragoons (also taking over 200 matches on average to do so). This suggests the model is simply not viable for consistently evolving good Zealot behaviour control. Increasing the complexity of the retreat action is detrimental to unit control that requires a focus on the aggressive attacking behaviour.

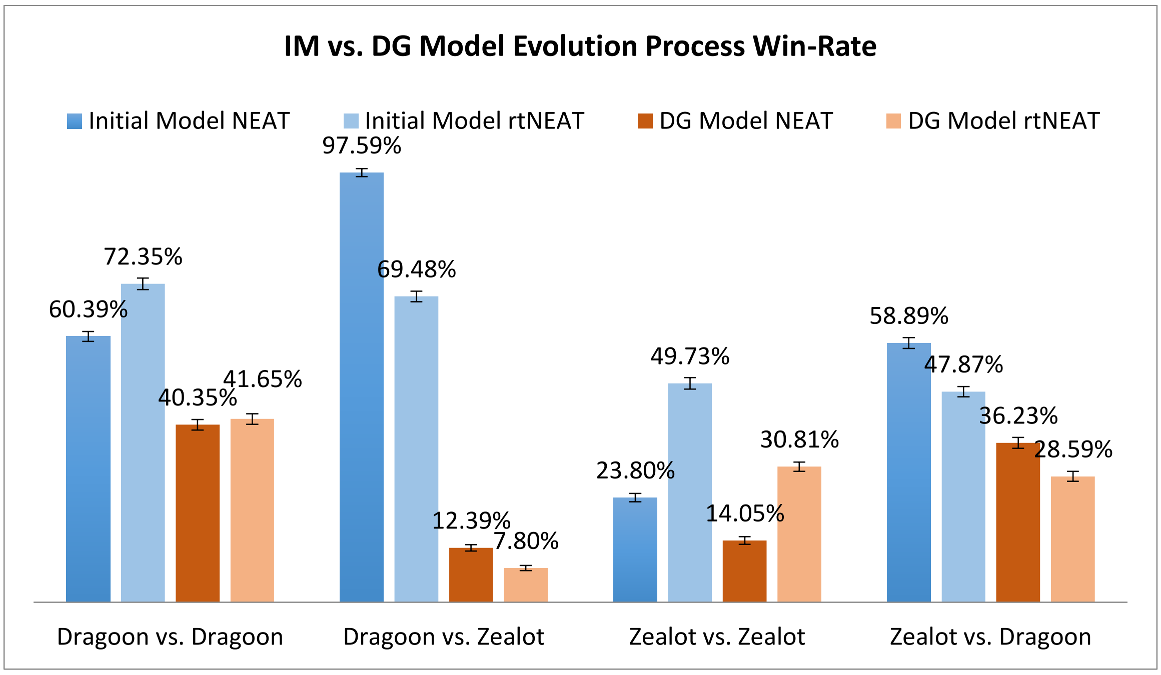


Figure 46: The results of the evolutionary process experiment for the initial model (IM) and the first 300 matches of the direction granularity (DG) model plotted together for comparison.

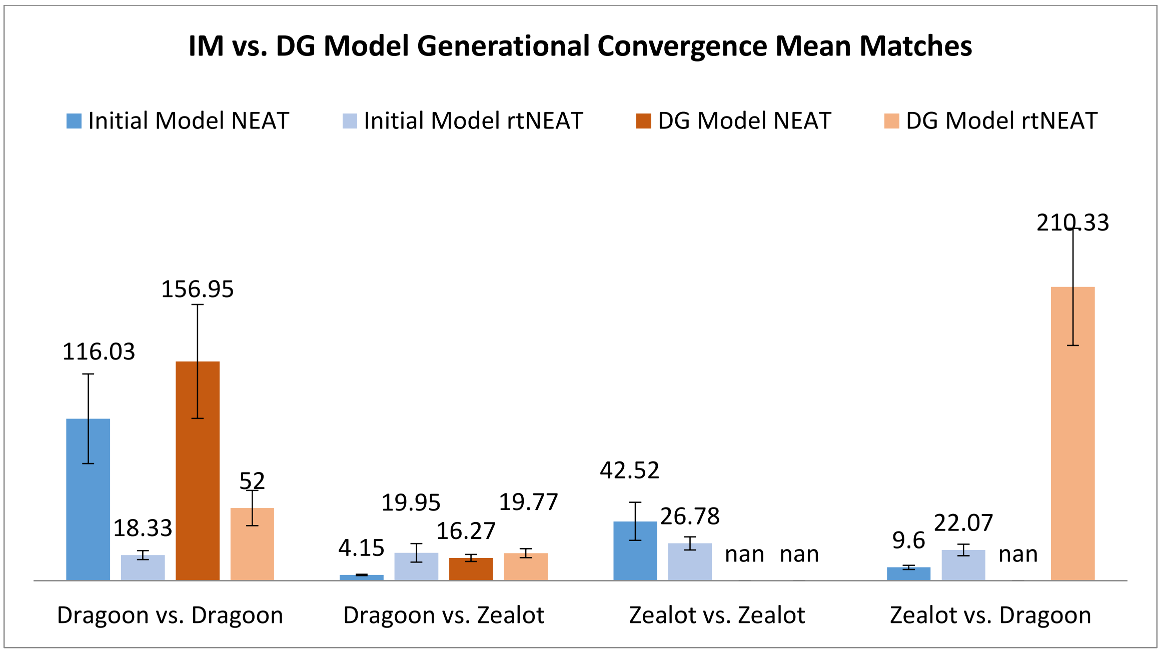


Figure 47: The results of the generational convergence experiment for the initial model (IM) and the direction granularity (DG) model plotted together for comparison.

Although the win-rates of the DG model is poor when controlling dragoons in the evolutionary process experiment, the results of the generational convergence experiment are comparable to those of the initial model. For Dragoon vs. Dragoon, rtNEAT is able to find successful solutions faster than NEAT much like for the initial model (Figure 47). For Dragoon vs. Zealots, the difference between DG and the initial model is small, especially between rtNEAT runs. In general, the DG model seems to perform best when controlling Dragoons, and when evolution is halted.

When examining the unit behaviour of DG model dragoons, it can be seen that the units are able to achieve a level of hit-and-run competency. Although the directional mappings for the retreat action may not always be ideal (opposite of enemy unit directional vectors), the units are still able to retreat in the right situations (weapon cool-down and when on low HP). However, in order for this behaviour to stabilize, evolution must be halted when the right innovations emerge. This is evident in the poor performance of dragoons in the evolutionary process experiment and the good performance in the generational convergence experiment.

In summary, the overall performance of the initial model appears to be superior to the DG model in comparative experiments. The simplicity of the initial model ensures a faster convergence when evolution is halted, as well as allowing higher win-rate when evolution is allowed to run its course. The initial model is consistent in generating winning solutions, while the DG model fails occasionally over 1000 matches. The insignificant difference in the result of the first and last 300 generation runs also suggest the DG model is not able to take advantage of longer evolution. Although the DG model has some success in generating solutions for controlling Dragoons, it performs poorly with Zealots units.

The next chapter continues to discuss the overall results of the experiments and its implications on the original research questions. It also addresses a number of limitations to the current approaches and identifies areas of future work.

Chapter 7

DISCUSSION AND CONCLUSION

This chapter begins with a discussion of the overall results of the previous chapter experiments. Following on, the core limitations of the methodology and evaluation are addressed. Some alternative solutions are considered, as well as areas of future work. Finally, some concluding remarks are made on the overall limitations and success of this thesis work.

7.1. OVERALL RESULTS

As part of the established objectives in section 1.2, the NEAT based micromanagement agent implemented in this thesis was evaluated in several ways. The results of the first round of experiments on the initial model (section 5.2.1) suggested a general viability for NEAT based techniques for adaptive micromanagement agents (consistent with the initial conclusions of Gabriel et al., 2012 on rtNEAT). It was first shown that when NEAT and rtNEAT are left running unchecked, the resulting solutions fluctuate between strong and weak performance against the default SC: BW AI (evolutionary process experiment). It was then shown that both NEAT and rtNEAT were capable of generating agents that can consistently defeat the default SC: BW AI, when the algorithms are halted upon a potential candidate solution (generation convergence experiment).

These experiments also suggest that rtNEAT was in general, more consistent and robust than NEAT in generating solutions with the initial model. This is evident in the generational convergence experiment where rtNEAT's performance is less variable between unit match-ups. It confirms rtNEAT's original design purpose as the algorithm that is quicker and more consistent in reacting to real time changes. However, generational NEAT is still capable of outperforming rtNEAT in some situations, as evident by higher win rates and faster convergence in some unit match-ups.

The second round of experiments with the directional granularity model was not as successful as the first. In the evolutionary process experiment, despite extending the number of generations, both algorithms performed poorly against the default SC: BW AI. The results of the generational convergence experiment were also generally worse than in the initial model, and both algorithms were inconsistent in generating solutions for some unit match-ups. Despite this, the performance of rtNEAT over NEAT was comparable to that of the initial model, such as having better or worse performances on the same unit match-ups.

The main difference between the two models is the directional input and output, which is also the main reason for the poor performance of the directional granularity model. Solutions were unable to learn the correct mapping between directional inputs (enemy and allied units) and outputs (direction to move). Even extending the number of generations of learning could not resolve this problem. This is inconsistent with the findings of Gabriel et al., 2012 which employed similar directional input and outputs. It is possible that the inconsistency results from implementation differences, and there is potential for future work in this direction (for example in developing an alternative encoding method for directional information, such as the vision grids used by Shantia et al., 2011).

The extension of the initial model to include more granularity in its inputs and outputs is not an arbitrary act of increasing complexity. The current action outputs of the initial model are static scripts and do not represent all possible actions (for example targeting an enemy for attack based on its damage output potential, rather than remaining hit points). Ideally, both target selection and unit movement should have the finest granularity input and output (capturing each enemy and allied unit attributes and locations), allowing the AI to learn the ideal action for all possible states. However, as demonstrated by the poor performance of the directional granularity model, this is not always practical or possible to learn and requires the right abstraction and encoding of the state and action space.

The overall objective stated in section 1.2 was to contribute to the development of a full SC: BW game playing AI capable of executing human level strategies. The results of the experiments in this thesis showed that NEAT is a viable learning technique for the micromanagement part of the SC: BW game domain. It is capable of generating AI that is able to defeat a static script strategy. Such emergent AI behaviour is an important precursor to human level strategies in the overall game. However, the extent of this viability is up for discussion and further evaluation. The input and output granularity is an example of the limitations of the methodology of this work. In the next section, several other limitations are discussed which requires addressing in order to expand the observed viability of NEAT based micromanagement.

7.2. Limitations and Future Work

As artificial neural networks are the core of NEAT algorithms, there are the associated limitations with using neural networks. The first is the difficulty of modelling and representing the problem as neural network inputs and outputs. As seen from the success of the initial model and the failure of the directional granularity model, it is not always intuitive or straight forward to model as complex a task as SC: BW micromanagement. This limitation of the methodology was addressed in the design section of Chapter 5. While a number of neural network mappings and models were considered, the ideal design is yet unknown. To address this, a potential area of future work is to employ automatic feature selection (Whiteson et al., 2005), so that NEAT learns to optimize for the ideal inputs when given a large variety.

Another limitation of neural networks is the difficulty of analysing resulting networks from training. A network evolved via NEAT may have hundreds of nodes and thousands of connections and weights. Retracing network activation or analysing network structure for meaning is a slow and arduous process. One of the important requirements for commercial game AI is transparency and simplicity, such that the behaviour of the AI can be explained and therefore tweaked if necessary. Complexity and unpredictability is not unique to Neuroevolution based game AI and is a common barrier for adapting machine learning techniques in commercial games in general. However, it is made worse for Neuroevolution because of the difficulty of analysing network structure.

Limitations also exist in the evaluation methodology, namely in the performance metric of playing against the default SC: BW AI. As previously mentioned, such a metric is commonly employed in existing micromanagement work. Due to the lack of implementation open sourcing, and differences in unit type and map scenarios used in evaluations between researchers, it is difficult to comparatively evaluate micromanagement agents. Ideally, training and evaluations should be against human players, and there are some avenues to do so for full game playing AI. However, the number of games and the variety of player skills needed for significance of results makes this difficult. More so for micromanagement agents because of the lack of existing tools to do so, and the fact that most human players online gather to play full SC: BW games, not custom micromanagement scenarios.

There exist many competitive avenues for full SC: BW game playing AI to be comparatively evaluated, and in theory it is possible to combine micromanagement agents as modules to these agents to be evaluated. However, the performance of the micromanagement component will then be dependent on the macromanagement components. In general there is a need to corroborate micromanagement techniques with research on macromanagement level learning, in order to further the goal of AI that can play complete matches of SC: BW at an expert human level. There is difficulty in merging standalone micromanagement agents together with agents making macromanagement level decisions. In future work, there is a need to establish standardized evaluation methodologies and to modularize micromanagement agents for full SC: BW match incorporation.

Following the idea of extending evaluations, of particular interest is comparative evaluations against existing reinforcement learning micromanagement agents (Shantia et al., 2011; Wender & Watson, 2012). As discussed in section 2.2.3, existing literature suggest that Neuroevolution and NEAT techniques are superior to traditional RL techniques for large and partially observable state-space problems. Using a standardized map and unit type set-up, SC: BW micromanagement can be aptly appropriate to further test this hypothesis. The challenge however, will be in standardizing the problem model employed across the different learning techniques, such as the unit inputs and actions.

Other potential areas of improvement include: exploring alternative network models, increasing the complexity of unit types and NEAT parameter tuning. As previously mentioned, a different encoding scheme for directional information may enable a network to include location information as input and output. Unit target selection itself could be a separate learning task with its own neural network input and output. Changing the types of units involved in training as well as their numbers will induce differences in the types of unit behaviour being learnt. For realistic micromanagement, the model must continue to work with a variable number of units and unit types. Finally, tuning the parameters of the NEAT algorithms will affect the evolution and learning processes, such as the speed of introducing new mutations or rate of retaining old structures. The use of recurrent networks and adaptive weights such as Hebbian update rules is also possible within the NEAT framework (Stanley, 2004), which poses potential future work.

The next section offers a summary of work within this thesis, as well as final concluding remarks on the contributions, successes and limitations.

7.3. Conclusion

This thesis presented the design, implementation and evaluation of a NEAT based learning agent for the task of micromanagement in the RTS game SC: BW. This concluding section summarizes the contributions made and readdresses the original research objectives stated in section 1.2. It also revisits and summarizes some of the key points of the previous chapters.

The work in this thesis was motivated and situated in the area of game AI research. In Chapter 1, I briefly introduced the motivations and benefits to AI research, as well as the challenges behind working in this area. RTS games and SC: BW is introduced as a unique test-bed for AI techniques, and the NEAT framework is introduced as a potential solution. This introduction is expanded in an extensive background and related work review on the history of game AI research, current work on SC: BW AI and NEAT in Chapter 2. Chapter 3 and Chapter 4 served as thorough reviews of the NEAT methodology and the SC:BW game environment. This was necessary to convey the theoretical motivations and practical workings of the NEAT algorithms, as well as explain the technical details of the SC: BW test-bed environment.

The rest of the chapters were dedicated to answering the research objectives established in section 1.2. The first research objective was the design and implementation of a NEAT based micromanagement agent in SC: BW. This was successfully accomplished and summarized in Chapter 5 of the thesis, opening way to experimentations on different NEAT algorithms and neural network models.

The accomplishment of the second objective, the basic evaluation of the NEAT based agent's performance against the default SC: BW AI was presented as part of the initial model experiments of Chapter 6. Evaluations here confirmed the viability of NEAT and rtNEAT algorithms in evolving agents for various SC: BW micromanagement scenarios. When the algorithms are allowed to run non-stop, win rate of agents against the default SC: BW AI fluctuates highly throughout generations. However when evolution is halted upon reaching an acceptable level of performance, both algorithms are able to consistently generate winning agents under 100 generations.

Following the third objective of extending the basic network model and exploring different NEAT algorithms, the rest of Chapter 6 detailed experiments on the directional granularity model. Throughout each experiment in the chapter, the performance of the NEAT and rtNEAT algorithms were also compared. Results of these experiments allowed comparative analysis between each algorithm, over different match-ups, models and evolution processes (generational convergence vs. evolutionary process experiments). This allowed insights into the strengths and weaknesses of each algorithm under various conditions, as well as identifying the stronger network model.

The stated overall objective of the thesis was to contribute to the development of a full SC: BW game playing AI capable of executing human level strategy. Early in this chapter, I discussed in part the viability of the NEAT based approach for the micromanagement aspect of the SC: BW game, as well as its limitations. Overall, NEAT's ability to generate AI capable of defeating the SC: BW static AI is an important precursor to a full game playing AI system with learning and emergent behaviour. Further work in comparatively evaluating the NEAT based approach against other techniques and human players will better gauge its effectiveness. More work is needed to adapt these techniques for full games of SC: BW, but results here have shown promising performance in a learning micromanagement AI capable of defeating scripted AI under short training time.

As each research objective has been addressed, the application of NEAT for SC: BW micromanagement can be considered successful. This thesis serves as a thorough first step in designing, implementing and evaluating NEAT based agents for SC: BW, and overall as an exercise in novel applications of machine learning in a complex, real-time problem domain.

References

Bakkes, S. C. J., Spronck, P. H. M., & van den Herik, H. J. (2009). Opponent modelling for case-based adaptive game AI. *Entertainment Computing*, *1*(1), 27–37.

Buro, M. (2003). Real-Time Strategy Games : A New AI Research Challenge. *IJCAI*, 1534–1535.

Buro, M. (2004). Call for AI research in RTS games. In *Proceedings of the AAAI Workshop on AI in Games* (pp. 139–141). AAAI Press.

Buro, M., Bergsma, J., & Deutscher, D. (2006). AI system designs for the first RTS-game AI competition. *Proceedings of the GameOn Conference*, 13–17.

Buro, M., & Churchill, D. (2012). Real-Time Strategy Game Competitions. *AI Magazine*, *33*(3), 106–108.

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

Cadena, P., & Garrido, L. (2011). Fuzzy case-based reasoning for managing strategic and tactical reasoning in starcraft. *Advances in Artificial Intelligence*, *7094*, 113–124.

Chellapilla, K., & Fogel, D. B. (2001). Evolving an expert checkers playing program without using human expertise. *Evolutionary Computation, IEEE Transactions on*, *5*(4), 422–428.

Cho, H., Kim, K., & Cho, S. (2013). Replay-based strategy prediction and build order adaptation for StarCraft AI bots. In *The Proceedings of the IEEE Conference on Computational Intelligence in Games* (pp. 265 – 266).

Churchill, D., & Buro, M. (2011). Build Order Optimization in StarCraft. *AIIDE*, 14–19.

Churchill, D., & Buro, M. (2013). Portfolio greedy search and simulation for large-scale combat in starcraft. *The Proceedings of the IEEE Conference on Computational Intelligence in Games*, 217 – 224.

Churchill, D., Saffidine, A., & Buro, M. (2012). Fast Heuristic Search for RTS Game Combat Scenarios. *AIIDE*, 112–117.

Dereszynski, E., Hostetler, J., & Fern, A. (2011). Learning Probabilistic Behavior Models in Real-Time Strategy Games. *AIIDE*, 20–25.

Furtak, T., & Buro, M. (2010). On the Complexity of Two-Player Attrition Games Played on Graphs. In *AIIDE* (pp. 113–119).

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

Gomez, F., & Miikkulainen, R. (2003). *Robust non-linear control through Neuroevolution*. The University of Texas at Austin.

Hagelbäck, J., & Johansson, S. (2009). A Multi-Agent Potential Field-based Bot for a Full RTS Game Scenario. *AIIDE*, 28–33.

Herbrich, R., Minka, T., & Graepel, T. (2006). Trueskill^{TM}: A Bayesian skill rating system. *Advances in Neural Information Processing Systems 19*, 569 – 576.

Herik, H. van den. (2005). Opponent Modelling and Commercial Games. In *Proceedings of IEEE 2005 Symposium on Computational Intelligence and Games* (pp. 15 – 25).

Hingston, P. (2009). A Turing Test for Computer Game Bots. *IEEE Transactions on Computational Intelligence and AI in Games*, *1*(3), 169–186.

Hoang, H., Lee-Urban, S., & Muñoz-Avila, H. (2005). Hierarchical Plan Representations for Encoding Strategic Game AI. *AIIDE*, 63–68.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, *2*(5), 359 – 366.

Hsieh, J. L., & Sun, C. T. (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* (pp. 3106–3111).

Jang, S.-H., Yoon, J.-W., & Cho, S.-B. (2009). Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary algorithm. In *Proceedings of the 5th international conference on Computational Intelligence and Games* (pp. 75–79).

Karakovskiy, S., & Togelius, J. (2012). The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 55–67.

Kovarsky, A., & Buro, M. (2005). Heuristic search applied to abstract combat games. *Proceedings of the 18th Canadian Society conference on Advances in Artificial Intelligence*, 66–78.

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

Levy, D. (1988). Computer chess compendium, 286–292.

Lucas, S. (2004). Cellz: a simple dynamic game for testing evolutionary algorithms. *Evolutionary Computation, 2004.*, 1007–1014.

Lucas, S. (2007). Ms pac-man competition. *ACM SIGEVOlution*, *2*(4), 37–38.

Marsland, T. A., & Björnsson, Y. (1997). From MiniMax to Manhattan. *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, 31–36.

Mateas, M. (2003). Expressive AI: Games and Artificial Intelligence. *Proceedings of International DiGRA Conference*.

Nareyek, A. (2004). AI in Computer Games. *Queue - Game Development*, *1*(10), 58–65.

Newell, A., Shaw, J., & Simon, H. (1958). Chess-playing programs and the problem of complexity. *Journal of Research and Development*, *2*(4), 320 – 335.

Nguyen, K. (2013). Potential flows for controlling scout units in StarCraft. *The Proceedings of the IEEE Conference on Computational Intelligence in Games*, 344 –350.

Norris, K., & Watson, I. (2013). A Statistical Exploitation Module for Texas Hold 'em And It's Benefits When Used With an Approximate Nash Equilibrium Strategy. In *The Proceedings of the IEEE Conference on Computational Intelligence in Games* (pp. 423 – 430).

Olesen, J. K., Yannakakis, G. N., & Hallam, J. (2008). Real-time challenge balance in an RTS game using rtNEAT. In *2008 IEEE Symposium On Computational Intelligence and Games* (pp. 87–94).

Palma, R., Sánchez-Ruiz, A. A., Gómez-Martín, M. A., Gómez-Martín, P. P., & González-Calero, P. A. (2011). Combining expert knowledge and learning from demonstration in real-time strategy games. *Proceedings of the 19th international conference on Case-Based Reasoning Research and Development*, *6880*, 181–195.

Parker, G., & Parker, M. (2007). Evolving parameters for xpilot combat agents. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 238–243.

Polceanu, M. (2013). MirrorBot: Using human-inspired mirroring behavior to pass a turing test. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 201 – 208.

Rathe, E., & Svendsen, J. (2012). *Micromanagement in StarCraft using Potential Fields tuned with a Multi-Objective Genetic Algorithm*. Norwegian University of Science and Technology.

Richards, D., & Hart, T. (1961). *The alpha-beta heuristic*. Massachusetts Institute of Technology.

Robertson, G., & Watson, I. (in press). A Review of Real-Time Strategy Game AI. *AI Magazine*.

Rubin, J., & Watson, I. (2012). Case-based strategies in computer poker. *AI Communications*, *25*(1), 19–48.

Sailer, F., Buro, M., & Lanctot, M. (2007). Adversarial Planning Through Strategy Simulation. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games* (pp. 80–87).

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, *3*(3), 210–229.

Sandberg, T., & Togelius, J. (2011). *Evolutionary Multi-Agent Potential Field based AI approach for SSC scenarios in RTS games*. IT University of Copenhagen.

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

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., … Sutphen, S. (2007). Checkers Is Solved. *Science* , *317* (5844), 1518–1522.

Schrum, J., Karpov, I. V, & Miikkulainen, R. (2012). Humanlike Combat Behavior via Multiobjective Neuroevolution. In P. F. Hingston (Ed.), *Believable Bots* (pp. 119–150). Springer Berlin Heidelberg.

Shaker, N., Togelius, J., Yannakakis, G. N., Weber, B., Shimizu, T., Hashiyama, T., … Baumgarten, R. (2011). The 2010 Mario AI Championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(4), 332–347.

Shannon, C. (1950). XXII. Programming a computer for playing chess. *Philosophical magazine*, *41*(314).

Shantia, A., Begue, E., & Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micro management in starcraft. In *The 2011 International Joint Conference on Neural Networks* (pp. 1794 – 1801).

Siegelmann, H., & Sontag, E. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, *4*(6), 77–80.

Siwek, S. E. (2010). *Video Games in the 21st Century*. Entertainment Software Association.

Slate, D. J. (1987). A chess program that uses its transposition table to learn from experience. *International Computer Chess Association Journal*, *10*(2), 59–71.

Smith, M., Lee-Urban, S., & Muñoz-Avila, H. (2007). RETALIATE: Learning winning policies in first-person shooter games. *Proceedings of the 19th national conference on Innovative applications of artificial intelligence*, *2*, 1801–1806.

Spronck, P., & Ponsen, M. (2006). Adaptive game AI with dynamic scripting. *Machine Learning*, *63*(3), 217–248.

Stanley, K. O. (2004). *Efficient evolution of neural networks through complexification*. The University of Texas at Austin.

Stanley, K. O., Bryant, B. D., & Miikkulainen, R. (2005a). Evolving neural network agents in the NERO video game. *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games*, 182–189.

Stanley, K. O., Bryant, B. D., & Miikkulainen, R. (2005b). Real-time Neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, *9*(6), 653–668.

Stanley, K. O., & Miikkulainen, R. (2002a). Efficient evolution of neural network topologies. *Proceedings of the Genetic and Evolutionary Computation Conference*, 1757–1762.

Stanley, K. O., & Miikkulainen, R. (2002b). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*(2), 99–127.

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

Synnaeve, G., & Bessiere, P. (2011). A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft. In *AIIDE* (pp. 79–84). AAAI Press.

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

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, *6*(2), 215–219.

Thompson, K. (1982). Computer chess strength. *Advances in computer chess*, *3*, 55–56.

Togelius, J., Yannakakis, G., Karakovskiy, S., & Shaker, N. (2012). Assessing believability. In P. Hingston (Ed.), *Believable Bots* (pp. 215–228). New York: Springer-Verlag.

Turing, A. (1988). Chess. In David Levy (Ed.), *Computer chess compendium* (pp. 14–17). New York, NY, USA: Springer-Verlag New York, Inc.

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

Wang, Z., Nguyen, K. Q., Thawonmas, R., & Rinaldo, F. (2012). Monte-Carlo planning for unit control in StarCraft. In *The 1st IEEE Global Conference on Consumer Electronics 2012* (pp. 263–264). IEEE.

Weber, B. (2012). *Integrating Learning in a Multi-Scale Agent*. University of California, Santa Cruz.

Weber, B., Mateas, M., & Jhala, A. (2010a). Applying Goal-Driven Autonomy to StarCraft. In *AIIDE* (pp. 101–106). AAAI Press.

Weber, B., Mateas, M., & Jhala, A. (2010b). Case-based goal formulation. *Proceedings of the AAAI Workshop on Goal-Driven Autonomy*.

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

Weber, B., & Mawhorter, P. (2010). Reactive planning idioms for multi-scale game AI. *IEEE Conference on Computational Intelligence and Games*, 115–122.

Weber, B., & Ontañón, S. (2010). Using Automated Replay Annotation for Case-Based Planning in Games. In *ICCBR Workshop on CBR for Computer Games* (pp. 15–24).

Wender, S., & Watson, I. (2012). Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft: Broodwar. In *IEEE Conference on Computational Intelligence and Games* (pp. 402–408). Ieee.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., & Kohl, N. (2005). Automatic feature selection in Neuroevolution. In *Proceedings of the 2005 conference on Genetic and evolutionary computation* (pp. 1225–1232). New York, New York, USA: ACM Press.

Yannakakis, G., & Hallam, J. (2007). Capturing player enjoyment in computer games. In N. Baba, L. Jain, & H. Handa (Eds.), *Advanced Intelligent Paradigms in Computer Games* (Vol. 71, pp. 175–201). Springer Berlin.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447.

Yildirim, S., & Stene, S. (2008). A Survey on the Need and Use of AI in Game Agents. In *Proceedings of the 2008 Spring simulation multiconference* (pp. 124–131).

Zhen, J., & Watson, I. (2013). Neuroevolution for Micromanagement in the Real-Time Strategy Game Starcraft: Brood War. In *AI 2013: Advances in Artificial Intelligence* (pp. 259–270).

Zielke, M. a, Evans, M. J., Dufour, F., Christopher, T. V, Donahue, J. K., Johnson, P., … Flores, R. (2009). Serious games for immersive cultural training: creating a living world. *IEEE computer graphics and applications*, *29*(2), 49–60.

Appendix A:

Nominated Best Student Paper at the 2013 Australasian Joint Conference on Artificial Intelligence (AI 2013)

Neuroevolution for Micromanagement in the Real-Time Strategy Game Starcraft: Brood War

Jacky Shunjie Zhen and Ian Watson

Department of Computer Science, University of Auckland
szhe024@aucklanduni.ac.nz,ian@cs.auckland.ac.nz

Abstract. *Real-Time Strategy (RTS) games have become an attractive domain for AI research in recent years, due to their dynamic, multi-agent and multi-objective environments. Micromanagement, a core component of many RTS games, involves the control of multiple agents to accomplish goals that require fast, real time assessment and reaction. In this paper, we present the application and evaluation of a Neuroevolution technique in evolving micromanagement agents for the RTS game Starcraft: Brood War (SC:BW). The NeuroEvolution of Augmented Topologies (NEAT) algorithm, both in its standard form and its real-time variant (rtNEAT) is comparatively evaluated in micromanagement tasks. Preliminary results suggest the general viability of these techniques in comparison to traditional, non-adaptive AI. Further analysis of each algorithm identified differences in task performance and learning rate.*

Keywords: Real-Time Strategy Games, Neuroevolution, Evolutionary Computation

1 Introduction

It was predicted more than a decade ago, that interactive computer games would emerge as an ideal platform for Artificial Intelligence research [1]. Due to their increasingly complex and realistic simulations, video games have become fine approximations of real world environments. AI techniques can be developed and evaluated in a cost effective and contained manner, before being applied to more complicated real world problems [1,2]. The popularity of video games as an entertainment medium has resulted in a consistently growing, multi-billion dollar software industry [3]. This in turn is a driver of video game technology and research, of which AI is a vital component [4].

The popularity and ease of access to videogame hardware and software has increased the accessibility of computing power and simulation environments for AI research. On the other hand, the contribution of AI research to commercial game development has been lacking in recent years [5]. This has resulted in high dependency on deterministic and non-adaptive AI techniques in commercial games that limit their realism, replayability and challenge [6].

Real Time Strategy (RTS) games are a genre of video games that provide unique challenges to AI research [7]. Characteristic of the genre is a real-time, stochastic environment, with multiple objectives and enormous action and state space. These features require AI agents with multiple levels of abstraction and reasoning, fast reaction and expert game knowledge. An example of the genre is Starcraft: BroodWar (SC:BW)[1], an RTS game that is a popular game environment for AI research. Using a third party plugin called the Brood War API (BWAPI)[2], it is possible to create complex AI agents to play matches of SC:BW.

In this paper, we aim to evaluate the effectiveness of Neuroevolution (NE) techniques in developing learning agents for playing SC:BW. The goal is to contribute to the development of a complete AI system capable of learning and executing human expert level strategy in SC:BW. In particular we focus on 'micromanagement', a crucial level of abstraction in the RTS domain, handling the fast combat component of the overall game. NE applies evolutionary algorithms to train artificial neural networks that are known to be effective approximators of complex, non-linear functions. Meanwhile, RTS games have a large state and action space that is suitable for neural networks. Furthermore, research on the NEAT algorithm [8] has shown the effectiveness of NE in reinforcement learning tasks, of which SC:BW has been successfully modelled [9,10].

We first implemented a micromanagement agent for SC:BW that uses NEAT and a real time variant rtNEAT for learning behavior. Next, the viability of the agent was evaluated against the existing SC:BW AI in multiple experiments. Finally, we analyzed the difference in performance between standard NEAT and the real-time variant, both in the rate of learning and in match performance. The rest of the paper is structured as follows: we provide a survey of related work around SC:BW AI and the NEAT algorithm. Next, we describe an overview of the implementation of our AI agent and the usage of NEAT, followed by the evaluation of the agent and a discussion of results. Finally, we give concluding remarks and highlight areas of future research.

2 Related Work

In many RTS games, the game strategy can be roughly divided into two levels of abstraction. Macromanagement is the level that is concerned with high level strategic decision making such as resource planning and opponent modeling. Techniques dealing with macromanagement must choose and adapt sequences of strategic actions to meet goals of varied hierarchy. Example of techniques applied to this domain include Case Based Reasoning[11] and Goal Driven Autonomy[12]. Micromanagement is the level concerned with direct combat tactics and unit control. Traditionally, micromanagement is accomplished via static AI techniques such as scripts based on simple metrics [13]. More complicated techniques in the literature include Reinforcement Learning (RL) and Evolutionary Algorithm approaches.

RL combined with neural networks has been applied to SC:BW micromanagement [10]. Agent learning was accomplished using the online Sarsa RL al-

[1] Starcraft: Brood War: http://us.blizzard.com/en-us/games/sc/
[2] Brood War API:http://code.google.com/p/bwapi/

gorithm, with neural-networks to approximate the state-action value function. Results showed a significant winning advantage against standard Starcraft AI, but required thousands of training rounds and are limited in the type and number of units represented. In [9], a comparative evaluation of RL techniques applied to SC:BW was presented. Four variants of RL algorithms were applied to a specific micromanagement task, involving a long ranged unit with high mobility against numerous melee (close ranged) enemies. Evaluations identified strengths and weaknesses of the different algorithms, and showed a high win rate against the default SC:BW AI. However, the results are derived from a very limited scenario, and the author acknowledges it is only the first part of a larger RL based SC:BW agent.

Work that is most related to ours is from [14], in which rtNEAT was applied to SC:BW micromanagement. Units were controlled by separate neural networks, specifying actions to take in real time. The network typology is evolved over generations of 12 vs 12 unit combat. Evaluations showed a significant win rate within 300 training generations and also claimed the rtNEAT algorithm allowed fast, real-time strategy adaptation. A limitation of the study is the use of a custom SC:BW map that replenishes unit numbers with up to 100 unit reserves. This is an unrealistic depiction of real SC:BW combat where units are not replenished immediately to replace dead units. Furthermore, real-time fitness improvement occurs only over unit combat time that is minimal in a full SC:BW match. However, the work showcased the potential of applying NEAT based algorithms to SC:BW micromanagement that our work analyzes further.

2.1 Neuroevolution and NEAT

Neuroevolution (NE) has shown effectiveness compared to standard RL, in problems with continuous and high-dimensional state spaces [8]. Traditional NE worked on pre-defined topologies, and searched over the space of connection weights. Topology and Weight Evolving Artificial Neural Networks (TWEANNs) attempts to also evolve the topology of the network, and has the potential to improve training speed and accuracy of solutions [15]. Furthermore, it reduces the uncertainty and effort of deciding on network topology by researchers [16].

However, TWEANN techniques face numerous challenges, such as complications with network structure in crossover and problems with genetic encoding. Work by [16] developed the NEAT algorithm to address these challenges. The classic NEAT algorithm was expanded to a real-time variant[17], in which evolution occured over a real time environment. The rtNEAT algorithm was demonstrated in a game called NERO, where agents are evolved and adapted in real time to tackle changing objectives. Regular NEAT has been successfully applied to RTS by [18], where neural networks are evolved to become AI players in an ensemble process. In [6] both NEAT and rtNEAT were used to automatically balance the challenge of the AI player in an RTS game. A novel challenge metric coupled with a fitness function guided the evolution of neural networks. The AI was continuously evolved to converge to the same challenge level as the human player, thus creating a more balanced gaming experience.

3 Implementation

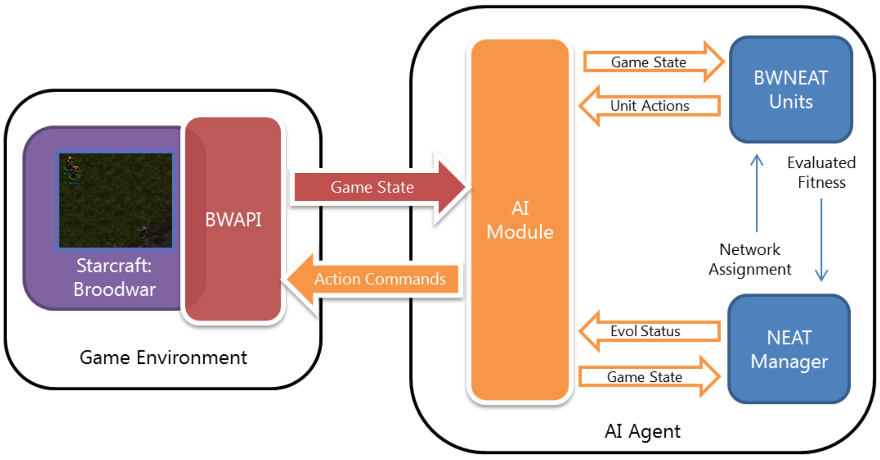


Fig. 1. An overview of the agent architecture.

We use the BWAPI open source framework for creating and executing AI modules. It exposes functionality to retrieve information about the game state and to issue commands to game units. Units are encapsulated as BWNEAT units, with an accompanying neural network for decision making. The SC:BW game state is updated once per frame, i.e. every 56 milliseconds on normal game speed. BWAPI triggers an event on every game state update, allowing AI code to react. During this event, each BWNEAT unit feeds internal and external percepts from the AI Module as inputs to its neural network, and interprets the network output as the next action to be performed. The NEAT Manager module is responsible for instantiating the BWNEAT Units and is an interface to the NEAT and rtNEAT algorithms. It receives evaluated fitness from each unit, performs NEAT evolution and reassigns neural networks to BWNEAT units. Fig. 1 summarizes the agent architecture.

3.1 Neural Network Architecture

The basic neural network architecture is fully connected and feed forward, with randomized starting weights (Fig. 2). It begins with 0 hidden nodes and gradually allows nodes and connections to be added via the NEAT and rtNEAT algorithms. The inputs were chosen as important percepts to induce learning behavior that would allow a unit to inflict as much damage to the enemy units, while taking as little damage as possible. For example, when the unit's weapon is on cooldown (a pause between consecutive attacks), it should do its best to avoid damage. Another aspect is the range of unit weapons. If for example, the unit has a

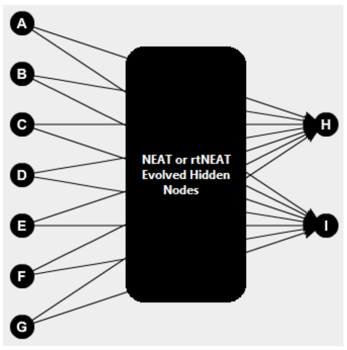


Fig. 2. Initial network architecture. Nodes A to G (*Bias, WeaponCooldown, RemainingHealth, WeaponRange, EnemyWeaponRange, NumAlliesInRange and NumEnemiesInRange*) denote a mixture of agent internal and external percepts as input to the network, while nodes H and I (*Fight and Retreat*) denote the outputs as two possible unit actions.

longer weapon range than the enemy units, it is possible to perform a hit-and-run strategy. These percepts are based on domain knowledge of SC:BW and is common in many RTS games.

The output of the neural network corresponds to two unit actions: fight or retreat. The action with the largest corresponding output is chosen by the unit. If the fight action is taken, the unit executes a simple routine that targets the enemy unit with the lowest hit points (health) within its weapon range. The retreat action makes the unit move a small distance away from enemies and obstacles, via a weighted vector. These actions are based on similar implementations in [9] and [10], as they are simple to implement, but complicated enough to produce sophisticated behavior when performed in varying sequences.

3.2 Fitness Function

The NEAT and rtNEAT algorithms are guided by a fitness metric. In the context of SC:BW unit micromanagement, the fitness should reflect the performance of an individual unit. For both NEAT and RTNEAT, we define the fitness F_i for a unit $_i$ as:

$$F_i = \frac{TDD_i - HPL_i}{IHP_i} + 1 \tag{1}$$

The function takes in the total damage dealt by the unit (TDD), its hit point loss (HPL) accrued over the match and its initial hit point (IHP). In theory, the fitness is only upperbound by the total hit points of enemy units. However in practise, the average fitness of each unit falls under $[0, 2]$ where at its lowest the unit has produced no damage and dies, and at its highest value it has dealt twice as much damage than it has taken.

3.3 NEAT Evolution

Training via the classic NEAT algorithm occurs over generations of SC:BW matches. After a match, regardless of win or loss, the population of neural networks go through an evolutionary process. First the fitness of each network is evaluated based on the units performance during the match. Next, some of the worst performing networks are replaced by the offspring of some of the best performing networks. This simple process is guided by three principles: tracking evolution via historical markers, protecting innovation via speciation and minimizing search via 'complexification' [8].

NEAT uses historical markings to efficiently evaluate similarity between network topologies. Networks are then speciated using a similarity metric formed via the number of disjoint genes D (genes that exist in one network and not the other), excess genes E (genes that appear in one network later in evolution than any genes on the other network) and the mean weight difference of matching genes W[8]:

$$S = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W \tag{2}$$

$c1$, $c2$ and $c3$ are adjustible weighting coefficients, and E and D are normalized by dividing N, the number of genes in the larger network. Networks are grouped into species via this similarity metric, and a compatibility threshold that can be modified to specify species bounds. A network shares its evaluated fitness with other members of its species, in order to encourage diversification of solutions and prevent single species dominance. NEAT adjusts each network's fitness based on its similarity metric against all other organisms in the population. The number of offsprings spawned by a species after each generation is based on the proportion of its average species fitness to the total of all average species fitness [8].

3.4 rtNEAT Evolution

The real-time variant of the NEAT algorithm is designed specifically to operate in a continuous, real-time domain. In particular when adapted to video games, the performance of AI agents is able to improve gradually as the game is played, without abrupt changes over a whole generation of evolution. In the context of SC:BW, rtNEAT applies evaluation and replacement on game units every n ticks of game time. The number of game ticks between replacement is an important factor that affects evolution. If new organisms are replaced too quickly, then they cannot be evaluated accurately and new innovations may be needlessly thrown away. A law of eligibility is formed by [17], stating the number of ticks between replacements, with respect to the fraction of the population that is too young to be replaced I, the minimum time alive m and the population size P:

$$n = \frac{m}{|P|I} \tag{3}$$

In our experiments we empirically define m as 300 game frames, to offset the delay between the start of a match and the first enemy encounter. We follow [17] in defining 50% of the population as eligible for replacement, and $p = 12$ the number of units which is constant. This gives us $n = 50$, the number of games frames between replacement in rtNEAT experiments. In the next section we describe in more detail, evaluations that incorporate these algorithms and principles to analyse the effectiveness of NEAT and rtNEAT for SC:BW micromanagement.

4 Experimentation and Results

We devised experiments to gauge the effectiveness of NEAT and rtNEAT evolved micromanagement agents against the standard SC:BW AI. A variety of unit setups were used in order to simulate different micromanagement scenarios. From these experiments, we saw very high fluctuation and variation in the fitness and win rate of agents over generations. In order to adjust for these fluctuations, we ran experiments to find the number of generations taken for each algorithm to converge to a suitable solution, when evolution is halted upon finding a potential candidate.

4.1 Experiment Setup

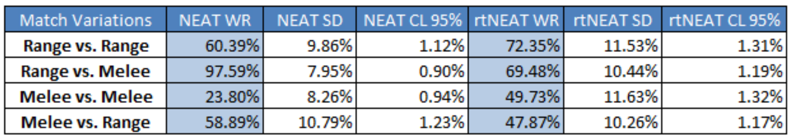
SC:BW units vary on attributes such as race, weapon and armour type. In order to keep the experimental variables constant and to avoid an explosion of unit type permutations, we based our experimental setup on [14] that compared 4 unit type variations (melee vs. melee, ranged vs. ranged, melee vs. ranged, ranged vs. melee). The number of units is kept at a constant 12 vs. 12, which is the maximum selectable number of units for a human controlled squad. The scenario used throughout experimentation is a flat map, based on those used in the AIIDE 2010 Starcraft micromanagement tournament[3].

4.2 Evolutionary Process Experiment

We first compared the performance of NEAT and rtNEAT algorithms on each of the 4 unit matchup variations, over 300 generations of evolution. Each matchup is repeated 25 times to reduce randomness in network starting weights. The average unit fitness and the match outcome is recorded over 300 generations, and averaged over the 25 runs.

The results suggest that there is no single algorithm dominating all match variations (Fig. 3). Mean win rate is higher for NEAT on range vs. melee (mean 97.59%, SD 7.95%) and melee vs. range (58.89% mean, 10.79% SD), while rtNEAT is higher on range vs. range (60.39% mean, 9.86% SD) and melee vs melee (49.73% mean, 11.63% SD). If we consider a win rate higher than 50% to

[3] AIIDE 2010 micromanagement tournament: eis.ucsc.edu/starcrafttournament1



Match Variations	NEAT WR	NEAT SD	NEAT CL 95%	rtNEAT WR	rtNEAT SD	rtNEAT CL 95%
Range vs. Range	60.39%	9.86%	1.12%	72.35%	11.53%	1.31%
Range vs. Melee	97.59%	7.95%	0.90%	69.48%	10.44%	1.19%
Melee vs. Melee	23.80%	8.26%	0.94%	49.73%	11.63%	1.32%
Melee vs. Range	58.89%	10.79%	1.23%	47.87%	10.26%	1.17%

Fig. 3. Summary statistics for our first experiment. Mean win rate (WR) over 300 generations, standard deviation (SD) and the 95% confidence level (CL) is shown.

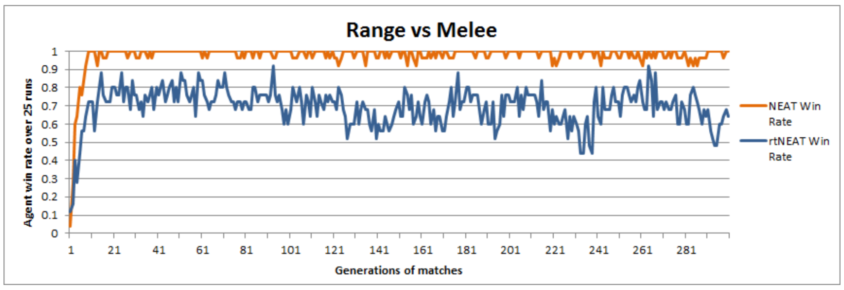


Fig. 4. Plot of the best performing match up (range vs. melee) average win rate.

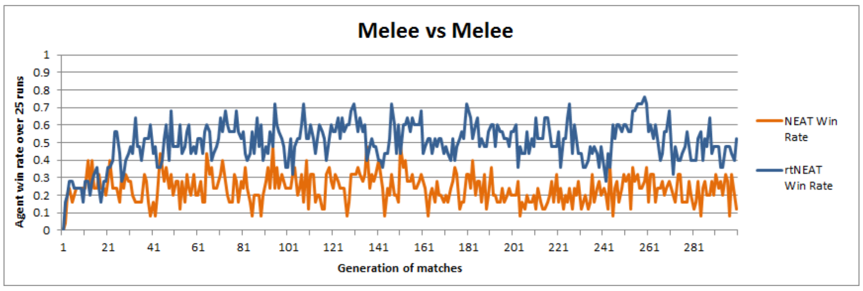


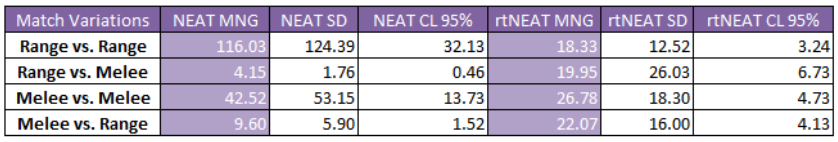
Fig. 5. Plot of the worst performing match up (melee v.s melee) average win rate.

indicate better than baseline performance against the built-in SC:BW AI, then both NEAT and rtNEAT show effectiveness on range vs. range and range vs. melee battles. NEAT is also effective in melee vs. range (58.89% mean ± 1.23% at 95% confidence level). From examining the agent behavior, the effectiveness of controlling ranged units can be attributed to having learnt the hit-and-run micromanagement strategies. It is interesting to note the generally poor performance of both algorithms when controlling melee units. We discuss in more detail the evolved behavior of the agents contributing to the performance in Section 5.

Fig. 4 and Fig. 5 show plots of some of the best and worst performing match up variations. The average fitness plot is not shown, but is highly correlated to the average win rate. From these plots, we see a trend of initial poor performance and a quick convergence to some local optima. This is typical of evolutionary algorithms where the initial starting solutions are randomized and are not expected to perform well. On all variations the first convergence to a local optimal occurs between the 10th to 20th generation. There is significant fluctuation of win rate throughout generations that is further illustrated by high variance shown by their standard deviations. In general, the standard deviation is higher on rtNEAT runs, suggesting greater variation in evolutionary success over generations than standard NEAT. This may be due to the nature of the rtNEAT algorithm, in introducing real time change. Both algorithms are capable of producing high performing solutions at various generations, but are also quick to introduce mutations weakening the solutions. This is partly due to the nature of the experiment where we allow evolution to continue even after achieving a winning solution.

4.3 Generational Convergence Experiment

By defining a success criteria, we can halt the evolution of both the NEAT and rtNEAT algorithms once an acceptable solution is achieved. We defined a successful solution to be an agent that achieves 10 consecutive wins (the probability an agent with 50% win rate can win 10 consecutive games is $< 0.1\%$). This is a strict criteria as an agent may still be high performing even though it loses 1 game out of 10 (e.g. due to the stochastic nature of the game state). However, we use this to simplify the running of the experiment and to show that it is possible to robustly generate agents of this level of performance. We keep all other variables the same as in our previous experiment, except that the evolution terminates when a solution reaches 10 wins, or after 1000 generations. When a solution achieves a win, evolution is halted until the agent either achieves 10 wins, or a loss is encountered, where upon evolution continues. After a successful solution is achieved, or if no solution is found after 1000 generations, the experiment is reset to an initial population with randomized weights. For each algorithm and each matchup, we stopped the experiment at 60 runs and analyzed the results.



Match Variations	NEAT MNG	NEAT SD	NEAT CL 95%	rtNEAT MNG	rtNEAT SD	rtNEAT CL 95%
Range vs. Range	116.03	124.39	32.13	18.33	12.52	3.24
Range vs. Melee	4.15	1.76	0.46	19.95	26.03	6.73
Melee vs. Melee	42.52	53.15	13.73	26.78	18.30	4.73
Melee vs. Range	9.60	5.90	1.52	22.07	16.00	4.13

Fig. 6. Summary statistics for generational convergence experiment. Mean number of generations (MNG) taken to produce an acceptable solution is shown. Standard deviation (SD) and the 95% confidence level (CL) for the mean was also calculated.

In all experiments, an acceptable solution was found before 1000 generations, with most converging under 100 generations (Fig. 6). There was high variability in the number of generations required to arrive at an acceptable solution, evident by the high standard deviation in some match ups (e.g. 124.39 SD and 116.03

Mean generations for NEAT range vs range). The mean number of generations taken between NEAT and rtNEAT is comparable to the average win rate performance of the previous experiment: NEAT converges faster for range vs. melee and melee vs. range match ups, while rtNEAT is faster for range vs. range and melee vs. melee. The range in generations taken between different matchups is higher for NEAT (4.15 mean for range.vs melee and 116.03 mean for range vs. range) than for rtNEAT (18.33 mean for range vs. range and 26.78 mean for melee vs. melee). This suggests the performance of rtNEAT is more stable under different unit variations.

Overall, the experiment showed that both algorithms were capable of generating effective solutions for micromanagement against the default SC:BW AI. However, it was necessary to establish an acceptance criteria for which to halt evolution and to preserve winning behavior. In the next section, we discuss further implications of the experimental results.

5 Discussion

In the first experiment, the fluctuation of fitness and success rate of solutions can be due to a number of reasons. Firstly, it suggests that any structural innovations introduced were making significant differences in the performance of the neural networks. This is probably due to the simplicity of the network design, where only 2 outputs exist, such that any structural change may affect the action selected. A simple neural network allows faster convergence by reducing the search space of initial nodes and weights. But it also means it is faster to diverge from the local optima. On top of this, the stochastic nature of the game environment can result in the same solution having varied success over different runs.

This also explains the general poor performance of both algorithms on melee match ups in the first experiment. Melee units do best in direct attack as they lack the weapon range to perform hit-and-run maneuvers. Any innovation introduced to make melee units run will immediately reduce the success rate of the solution. In the second experiment, the algorithms have no problem generating a solution for melee match ups, when no new innovations were introduced after a solution begins to do well. Another factor is an interesting behavior exhibited by the units over generations of evolution: some units are evolved to retreat when enemies are first found, but come back to fight after allied units are engaged in combat. These units tend to generate more fitness than those directly attacking from the beginning, as they do not receive as much damage over time. However, as the population begins to favour this behavior, there is a breaking point in which no units will stay to fight, leading to a match loss and a return to evolution favouring units that do not retreat. This cycle is highly correlated with the fluctuation of the win rate over generations.

Interestingly, the first experiment showed that rtNEAT produced higher variation in the success of solutions than NEAT over time. However in the second experiment, the average number of generations for an acceptable solution was less varied across different match ups than NEAT. This suggests real-time evolution can be quicker in introducing changes that reduce fitness, but also allow a more robust convergence to a solution regardless of unit variation (variability in state and solution space). This is intuitive, since rtNEAT should be faster in reacting to changes in the environment in real time, than regular NEAT evolution between generations.

It is possible to complicate the initial neural network architecture, by incorporating more percepts as input nodes and providing finer grain output decisions. For example, the inputs can incorporate a deeper ontology of unit quality and type variations (armour, weapon and ability types etc) and more precise directional and distance data. Instead of fight or retreat actions, the decisions can be to move at specific angles for specific distances, and to explicitly decide which units to attack. Enemy target selection is itself complicated enough to be a separate learning task, perhaps requiring the optimization of a separate neural network that takes into consideration enormous unit type variations and the location of units. More complicated neural network designs allow for agents with more complicated behaviors that are able to perform well under a higher variety of conditions. The disadvantage is a greater number of dimensions to search and optimize for, resulting in slower training time.

There are limitations to the evaluation methodology to be addressed. For example, while the experiments show that the technique is able to learn to defeat the standard SC AI, the results do not extend to human opponents. However, testing against the standard SC AI is a baseline measure used in much of the related work, particularly for micromanagement. It is difficult to evaluate against human players, due to the number of games required to be played, and the lack of an objective human skill measure for the micromanagement task (current measures exist only for full SC games). It is possible to evaluate against other micromanagement AI, but there is a lack of a standardized evaluation methodology to do so.

6 Conclusions

Our evaluations confirmed the viability of NEAT and rtNEAT algorithms in evolving agents for various SC:BW micromanagement scenarios. When the algorithms are allowed to run non-stop, the win rate of agents against the default SC:BW AI fluctuates highly over generations. However, when evolution is halted upon reaching an acceptable level of performance, both algorithms are able to consistently generate winning agents, with most under 100 generations. Each algorithm differs in the variability of performance over different unit matchups. Factors contributing to the difference in performance include the complexity of the network starting topology and the variation in unit types. There is room to explore network complexity further, and a need to establish standardized evaluation methods for micromanagement agent evaluations. More work is needed to adapt these techniques for commercial RTS game deployment, but results here have shown promising performance in a learning AI capable of defeating scripted AI under short training time.

References

1. Laird, J., VanLent, M.: Human-level AI's killer application: Interactive computer games. AI magazine **22**(2) (2001) 15–26
2. Buro, M.: Call for AI research in RTS games. In: Proceedings of the AAAI-04 Workshop on Challenges in Game AI. (2004) 2–4
3. Siwek, S.E.: Video Games in the 21st Century. Technical report, Entertainment Software Association (2010)
4. Yildirim, S., Stene, S.B.: A survey on the need and use of ai in game agents. In: Proceedings of the 2008 Spring simulation multiconference. (2008) 124–131
5. Mehta, M., Ontañón, S., Amundsen, T., Ram, A.: Authoring behaviors for games using learning from demonstration. Workshop on Case-Based Reasoning for Computer Games (ICCBR) (2009)
6. Olesen, J.K., Yannakakis, G.N., Hallam, J.: Real-time challenge balance in an RTS game using rtNEAT. In: 2008 IEEE Symposium On Computational Intelligence and Games. (2008) 87–94
7. Buro, M., Furtak, T.M.: RTS games and real-time AI research. In: Proceedings of the Behavior Representation in Modeling and Simulation Conference. (2004) 63–70
8. Stanley, K.O., Miikkulainen, R.: Efficient Evolution of Neural Network Topologies. In: Proceedings of the 2002 Congress on Evolutionary Computation (CEC02). IEEE. (2002)
9. Wender, S., Watson, I.: Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar. In: Computational Intelligence and Games (CIG). (2012) 402–408
10. Shantia, A., Begue, E., Wiering, M.: Connectionist reinforcement learning for intelligent unit micro management in starcraft. In: The 2011 International Joint Conference on Neural Networks (IJCNN). (2011) 1794–1801
11. Cadena, P., Garrido, L.: Fuzzy Case-Based Reasoning for Managing Strategic and Tactical Reasoning in StarCraft. In: Advances in Artificial Intelligence. Volume 7094. (2011) 113–124
12. Weber, B., Mateas, M., Jhala, A.: Applying goal-driven autonomy to StarCraft. Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2010) (2010)
13. Davis, I.L.: Strategies for strategy game AI. In: Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Computer Games. (1999) 24–27
14. Gabriel, I., Negru, V., Zaharie, D.: Neuroevolution based multi-agent system for micromanagement in real-time strategy games. In: Proceedings of the Fifth Balkan Conference in Informatics - BCI '12. (2012) 32
15. Yao, X.: Evolving artificial neural networks. In: Proceedings of the IEEE. Volume 87. (1999) 1423–1447
16. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2) (2002) 99–127
17. Stanley, K.O.: Evolving neural network agents in the NERO video game. In: Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games. (2005) 182–189
18. Jang, S.H., Yoon, J.W., Cho, S.B.: Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary algorithm. In: Proceedings of the 5th international conference on Computational Intelligence and Games. (2009) 75–79

ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Network
BT	Behaviour Trees
BWAPI	Brood War Application Programming Interface
CBP	Case-Based Planning
CBR	Case-Based Reasoning
FPS	First-Person Shooter
FSM	Finite-State Machine
GA	Genetic Algorithm
GDA	Goal-Driven Autonomy
HP	Hit-Points
HTN	Hierarchical Task Network
NE	Neuroevolution
NERO	Neuro Evolving Robotic Operatives
NN	Neural Network
ORTS	Open Real-Time Strategy
RETALIATE	Reinforced Tactic Learning in Agent-Team Environments
RL	Reinforcement Learning
RTS	Real-Time Strategy
SC: BW	StarCraft: Brood War
TWEANN	Topology and Weight Evolving Artificial Neural Network
UT	Unreal Tournament