Neural Networks for Small-Scale Combat Optimization in Real-Time Strategy Games

Jacob Menashe and Vineet Keshari

December 11, 2012

Abstract

A major challenge for players and AI modules alike in the real-time strategy community is the effective control of individual units in combat scenarios. Starcraft, a real-time strategy game released by Blizzard Entertainment, provides built-in heuristics for assisting players in combat management, but these heuristics are limited in their ability to make full use of a unit's potential. We present a series of techniques for training neural networks in small-scale combat situations, and lay the groundwork for further improvements to this system. We present a neural network design for controlling combat units, which, when coupled with the NEAT evolutionary learning algorithm, is able to consistently overcome Starcraft's built-in AI. In most cases our technique achieves optimal champion performance in less than 100 generations, and scales to large battles in a variety of unit configurations. Our system achieves a X% win rate against the AI, and a X% win rate against humans in small-scale battles.

1 Introduction

Micro-management is a major component of real-time strategy (RTS) games such as Starcraft. Players must effectively move individual units around for gathering resources, constructing buildings, or for tactical combat. In the case of tactical combat, micro-management requires quick action on the part of the player and thus can be difficult to master even with a small number of units. While modern games tend to minimize the degree to which players must control units on a small scale through the implementation of batch commands, this is usually achieved with simple heuristics a significant amount of potential is sacrificed for such enhancements. For a simple example of this, consider the following scenario: a player might take a heterogeneous group of close-range and long-range units and issue a batch attack command, at which point her units will either attack at random or attack one unit at a time. A more effective strategy might be to allow ranged units to disengage and re-engage based on the enemy's response, to allow these units to keep their distance, while close-range units focus on the greatest threat to allied ranged units. In this situation, the simplicity of the batch AI has diminished the effectiveness of the player's army. Such circumstances are prevalent in modern RTS games.

The domain of micro-management in combat environments is therefore well-suited to the application of intelligent, autonomous agents. While the best AI implementations still fall below the average competitive human player in terms of high-level planning and strategy [1], a trained unit controller agent should have little difficulty outperforming a human for any reasonably-sized army. Indeed, while an agent may potentially make hundreds of decisions and unit actions per second, even the most skilled human Starcraft players can only achieve about 5 actions per second [2].

In this paper we present a novel application of NEAT to small-scale combat problem, restricted to the domain of Starcraft. We devise a variety of network topologies and configurations for outperforming the game's built-in AI, as well as human players, and describe how our technique may be extended for outperforming winning bots from the Starcraft AI Competition X. In Section 2 we discuss some background to AI development in real-time strategy games and specifically in Starcraft, and examine another body of work that focuses on small-scale combat. In Section 3 we review the software libraries used for our analysis, and continue with a discussion of our approach to learning controller agents in Section 4. We evaluate our results agains the built-in Starcraft AI, against other bots from the Starcraft AI Competition, and other humans in Section 5. In Sections 6 and 7 we conclude and discuss additional updates and improvements to our method, toward the ultimate goal of providing a general-use combat manager.

2 Background

Classic AI techniques (or GOFAI) have been used on a variety of board games, such as Connect Four, Checkers, and Chess [3]. Due to their simplistic and discrete state spaces, board games can often be broken down by traditional AI techniques such as alpha-beta pruning, or in some cases even brute force search. Chess pushes the limits of these techniques with a search tree complexity of at least 10^{120} [4], but can still benefit from the prediction value of limited search. What these games have in common is that computers maintain a significant advantage over humans stemming from their predictive abilities. As computing technology has improved, however, games have begun to take on more realistic simulations of the world and therefore rely heavily on continuous and highly complex states and actions. This increase in state-action complexity has in turn rendered traditional AI approaches ineffective at performing competitively with human opponents.

Recent work has seen reasonable success at taking on the challenges of modern games and simulations, particularly in the domain of First-Person Shooters (FPS). FPS games are a genre of computer game in which the player views the world through an avatar from a first-person point of view. The point of an FPS is to navigate through the game world and kill opponents with weapons and tools found throughout the environment. Neuroevolution has been successfully applied toward the goal of producing human-like actions [5], resulting in creating bots that effectively pass the Turing test. In FPS games, creating bots that are simply better than human players can be achieved through simple physics calculations, so here the emphasis is on enabling human-like movements and downgrading combat prowess where it can be seen as unrealistic. In contrast, our work focuses on outperforming human- and AI-controlled systems alike.

There has been considerable progress toward this goal in Starcraft, primarily due to the Starcraft AI Competition. The competition pits Starcraft bots against one another to determine a winner, however this task lies parallel to that of defeating a human controller. EISBot, one of the more successful bot implementatoins, has been tested against skilled human players and achieved a win rate of approximately 32% [1]. Other top-performing bots from the competition have been unable to win against expert players [6]. In our work, we therefore train and evaluate our networks based on the built-in AI, with the long-term goal of competing against both bots in the competition and human players.

Typical approaches to Starcraft bots include case-based reasoning X and hierarchical skill management X. The former technique aims to learn skills, build order, and general strategy from replays of past games, while the latter technique organizes expert modules into a hierarchical decision model. These two approaches need not be mutually exclusive. Wender and Watson

refine the problem of creating expert modules by focusing exclusively on small-scale combat tactics in [7], and it is this work that relates most closely to our own. While Wender and Watson restrict their analysis to the particular combat configuration of a single, fast, ranged unit versus multiple slower short-ranged units, we seek to optimize small-scale combat in a more general sense by allowing for heterogeneous unit groups with an unspecified number of combatants. In this way, our approach moves toward a more flexible solution for the problem of micro-management in small-scale combat.

Another notable difference between our work and that of [7] is our use of NEAT (see Section 4.1) for learning unit controllers. We select NEAT for a number of reasons. First, as we will describe in Section 4.3, our state space is continuous and high-dimensional. Neural networks are naturally well-designed for parsing continuous state spaces and determining their value. Second, due to the computational cost of communicating with and executing commands on the Starcraft runtime, we are restricted in the number of iterations we may perform to learn this state space. NEAT has been shown to significantly outperform traditional reinforcement learning methods under these circumstances [8]. Finally, while we can accurately determine the value of a match, we cannot predict what intermediate behaviors might be optimal for our overall goal of achieving combat prowess. We therefore rely on NEAT to discover these innovations in the form of connections and connection weights in an efficient manner.

3 Software

Our research was enabled by the BroodWar API (BWAPI) [9], BWAPI Mono-Bridge [10], and Starcraft: Broodwar itself. We use SharpNeat [11], a C# NEAT implementation for neuroevolution. While BWAPI is natively coded in C++, we selected a C# implementation due to the following considerations:

- Our focus for this work is in testing new configurations for NEAT and new combinations with our own heuristics, so development speed is of particular importance. C# provides many high-level language features that allow for faster development than in C++.
- BWAPI only runs on Visual Studio in Windows, so there is no cross-platform or open-source advantage to using C++.
- SharpNEAT is an exceptionally well-coded and well-documented NEAT implementation. Along with working quite simply out of the box, the code is modularized and easily configurable, and has built in analysis and visualization tools.

Because C# communicates with BWAPI through a client-server interface, processing is slower than would be in the case of C++. We felt that the increased productivity more than made up for this disadvantage. In addition, the mono-bridge implementation uses an out-of-date version of BWAPI, although this didn't significantly affect our work. For future development it will be necessary to update this interface.

4 Technical Approach

We now cover our approach to the small-scale combat problem, beginning with a short introduction to NeuroEvolution of Augmenting Topologies (NEAT). We continue with a discussion of our implementation and network designs before proceeding to our software architecture.

Parameter	**Interpretation**
Input Nodes	Game State Representation
Output Nodes	Available Actions and Decisions
Evolution Parameters	Intensity and Character of Behavior Adjustments

Table 4.1: Parameters specified for the NEAT algorithm.

Connection Weight Mutation Probability	0.8
Add Connection Probability	0.1
Delete Connection Probability	0.001
Add Node Probability	0.1
Timesteps Per Activation	10
Connection Cycles	Enabled (i.e. Recurrent)

Table 4.2: Selected NEAT evolution parameters. Probabilities are normalized such that they sum to 1.0 upon evaluation.

4.1 The NEAT Algorithm

NEAT provides us with a fairly simple way of creating network designs without prior knowledge of the optimal target configuration. While traditional neural network algorithms such as backpropagation rely on a predefined network architecture, NEAT requires only that we specify a relatively small number of parameters. These parameters are shown in Table 4.1. Hidden nodes are added as needed based on random mutation and resulting fitness values.

Properly designing the input and output nodes is critical to the performance of a neural network. We therefore maintain the evolutionary parameters defined in Table 4.2 for all tests, and instead focus our attention on the network I/O architecture.

4.2 Training Setup

Our network designs are closely tied to our training configurations, and thus we will cover the general outline for training to give context to our designs in following sections. Starcraft's built-in Campaign Editor application allows for users to create custom maps complete with environment specifications and state-based triggers. We used the campaign editor to construct scenarios in which teams of units are repeatedly spawned at two starting points, one for our bot ("allies") and one for the built-in AI ("enemies"). Allies and enemies repeatedly fight one another until all of one side has been defeated, signaling the completion of that round. For each genome generated by NEAT, we assign fitness based on its average performance over 5 rounds using Equation 4.1:

$$f = \frac{(A - E) + A_{\max}}{A_{\max}} \tag{4.1}$$

where A and E are the ally and enemy counts at the end of the round, respectively, and $A_{\max} = E_{\max}$ is the starting count of allies or enemies in the training scenario. Using this formulation, the fitness value is always in the range $[0, 2]$, with 1 representing a tie. In practice, ties only occur when the learned network moves the allies out of the combat area, so to avoid this behavior we assign a fitness score of 0 in all scenarios where $A \neq 0$ and $E \neq 0$.

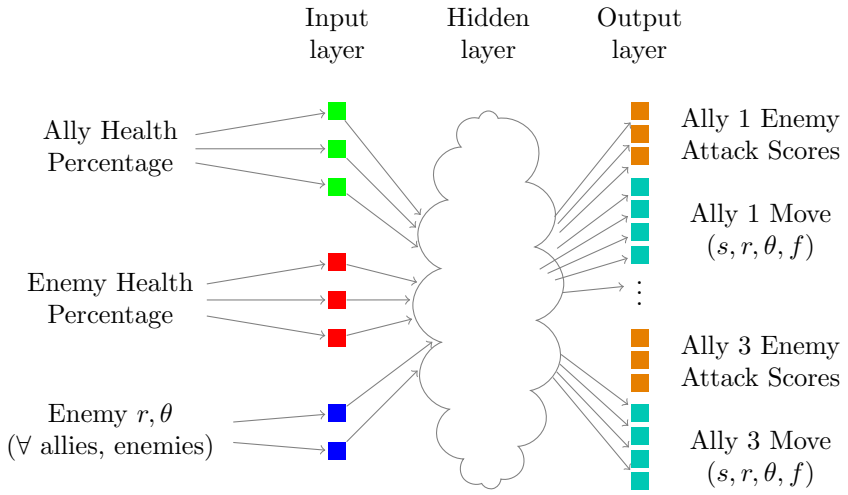

Figure 4.1: The multi-unit network design for 3 allies and 3 enemies. r and θ represent relative distance and angle. s represents a score in the range $[0, 1]$, and f is a flip.

4.3 Network Design

We attempted multiple network designs, each of which is evaluated in Section 5. In this section we make multiple references to training time in terms of hours. Training was performed on a desktop PC with Starcraft and NEAT running on a single core at 3.07 GHz.

4.3.1 Multi-Unit Control

We now begin with the multi-unit control network design in Figure 4.1.

This design represents our initial attempt at a network design. Inputs to the network include health percentages for each ally and each enemy in the training session, as well as the distances and angles of each enemy relative to each ally. Outputs include one score for each ally/enemy pair, as well as move score, distance, angle, and flip values for each ally. Flips are added as a way to differentiate between different extremes of the output range of the movement angle node. Originally we attempted a direct mapping of $\theta = 2 \cdot \pi \cdot o_\theta$, where $o_\theta \in [0, 1]$ is the value of the movement angle output node. This has the effect that the extremes 0 and 1 map to the same movement. Thus, we instead use the following formulation:

$$\theta = \begin{cases} o_\theta \cdot \pi & : f > 0.5 \\ o_\theta \cdot -\pi & : f \leq 0.5 \end{cases}$$

This way, units can move omnidirectionally with either extreme of the output range corresponding to different directions.

Using this network design, our units were able to achieve reasonable fitness levels in almost all trials with small combat scenarios within a few hours of training. Particularly, the design performed well with 3-5 units on either side. In the case of a 20-vs-20 battle, however, the design was not able to surpass an average population fitness of 1. This meant our units were losing battles more often than not, even after days of training. Reflecting on the design, this makes sense: with 20 allies and enemies there are $2 \cdot 20 + 20 \cdot 20 \cdot 2 = 880$ input nodes and $20 \cdot (20 + 4) = 480$ output nodes. This results in over 400,000 possible connections, without

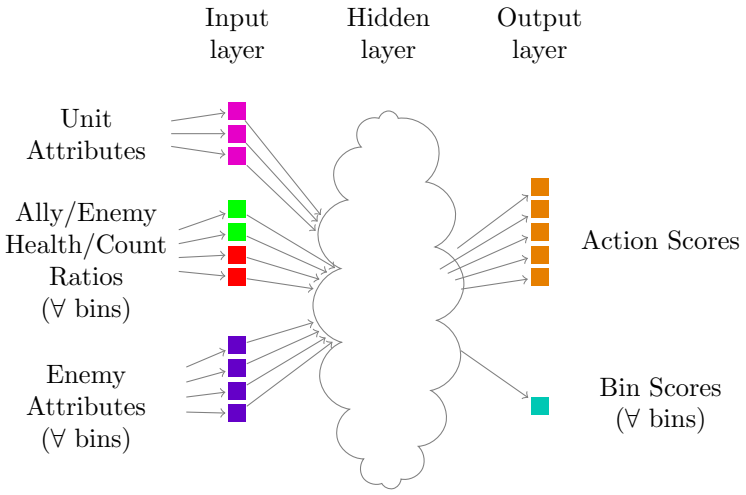

Figure 4.2: The individual controller network design.

taking into account hidden nodes that are added through neuroevolution. Given the constraints of interoperation with Starcraft, this presents a clear problem for quickly evolving networks to solve the small-scale combat problem. Moreover, the evolved networks are statically tied to the number of combatants; for any particular battle, we would need to have a precise controller pre-trained for that particular scenario. In practice, we would like our controllers to be more flexible, and thus we proceed to other solutions.

4.3.2 Individual Control

Our next attempt takes its inspiration from NERO [12], where we use a single neural network design to control each combat unit individually. This allowed us to create more complex designs with the knowledge that we would not be restricted by the number of allied units in the training scenario. As we see in Figure 4.2, unit attributes are encoded into the inputs to allow for simultaneous learning on heterogeneous unit types. We furthermore use a polar map based on the idea of log-polar bins to represent enemy positions, health, and attributes. In this way, we allow for arbitrary configurations of heterogeneous enemy combatants. To reduce the number of output nodes, we use a small set of nodes for action selection, where the action with the highest score is selected. We then use another cluster of nodes for bin selection so that the highest-scoring action is applied to the target bin. Actions are defined in Table 4.3. Finally, to ensure that all values are normalized to the range $[0, 1]$, each I/O value represents a percentage. Ally/Enemy counts are relative to the entire battle, so a bin with 7 enemies out of a total of 10 on the battlefield would have any enemy count value of 0.7, and a bin with all of the flying enemy units would have a `HasFlyer` value of 1.

This network design performed well in a variety of scenarios, from small (3v3) battles to larger 20v20 battles, however the networks that were generated lost some of the interesting behaviors we had seen in the multi-unit controller. A side goal of our experiments was to create networks that discovered interesting behavior, and so we decided to try another formulation that combines the scalability of the individual unit controller with the enhanced awareness of the multi-unit controller. We examine this controller in Section 4.3.3.

Action Type	Description
`AttackAir`	Attack air units
`AttackShort`	Attack short-ranged ground units
`AttackLong`	Attack long-ranged ground units
`AttackAirBonus`	Attack units that have a special bonus attack vs. air units
`Move`	Move to the center of the target bin

Table 4.3: Discrete action types selected through competing output nodes.

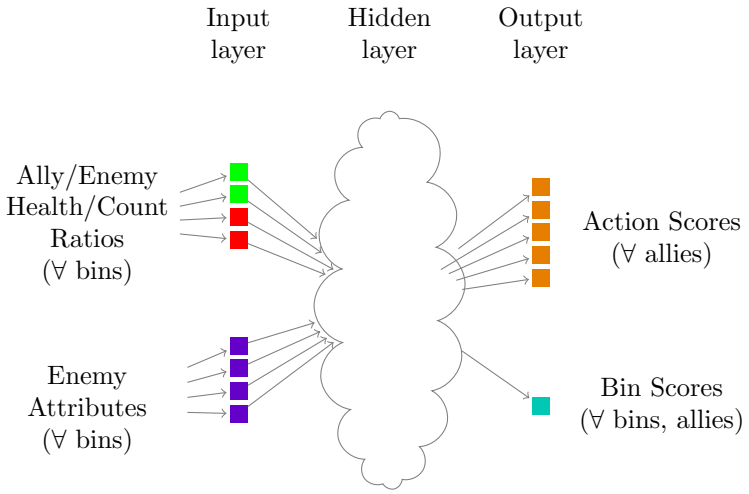

Figure 4.3: The squad controller network design.

4.3.3 Squad Control

In Figure 4.3 we see the basic design for the squad controller. The idea behind this design is to take a small number of predefined unit types and teach them to work together, so that they can be organized into a larger battalion with a squad commander. The network design is essentially a mix of the ideas from the multi-unit and individual-unit controllers: ally and enemy statistics are captured per bin (with respect to the squad's centroid), and action/bin scores are calculated for each ally. With 4 allies in one squad and 15 bins, this results in 105 input nodes and 80 output nodes.

Individual squads need to function together as a team, and thus we introduce the squad commander as the final network design we examined. The squad commander is much simpler than the other network designs, since it relies heavily on the fact that its squads are pre-trained. Figure 4.4 shows the details. The commander takes in ally and enemy health, and decides whether to move each squad to a different location or to delegate control to that squad. The commander uses heuristics with prior knowledge to divide enemy groups between squads. For example, a squad that is well-equipped to take out flying units will preferentially be assigned flying enemy combatants as targets. Squads can therefore focus specifically on their targets assigned by the commander and ignore the rest of the enemies on the battlefield. This hierarchical organization reduces complexity at each level, and reduces overall design complexity by orders of magnitude.

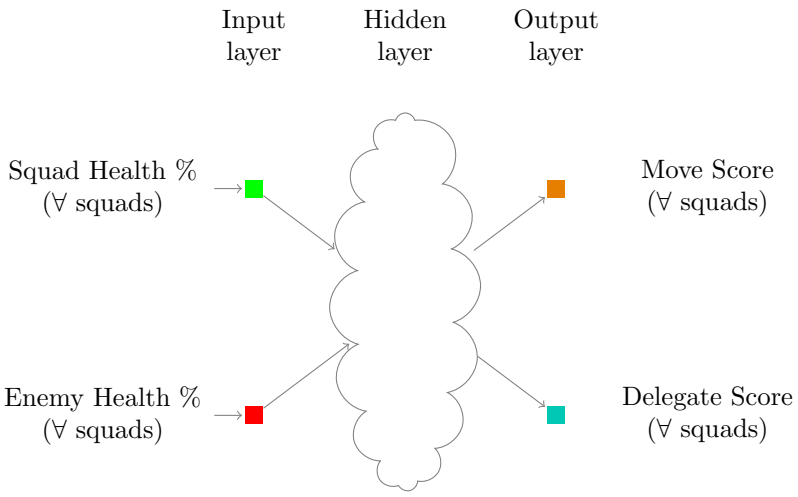

Figure 4.4: The squad commander network design.

Unit	**Description**
Marine	Long-range ground unit
Firebat	Short-range ground unit Can hit multiple enemies simultaneously
Ghost	Long-range ground unit More powerful than a marine
Goliath	Long-range ground unit Has attack bonus vs. air units More powerful than marines, firebats, and ghosts
Wraith	Long-range air unit

Table 5.1: Descriptions of units used in the training experiments.

5 Experiments

We now present a series of experiments to show the effectiveness of each network design in practice. In the following experiments we will refer to a handful of Starcraft units. These units are described briefly in Table 5.1. We proceed with the experiments analogously to Section 4.3, and conclude with small trials against an existing bot and a human competitor.

5.1 Multi-Unit Control

We performed our initial experiment to gauge the ability of a simple network design to outperform Starcraft's built-in AI. Using the controller from Section 4.3.1 we construct a training scenario in which 2 marines and 1 firebat spawn for each team on the two sides of the combat area. The built-in (i.e. enemy) AI attacks the spawn location of the allied AI, ensuring that if the allies do nothing then a battle will still take place. Figure 5.1 shows the fitness graph obtained for this experiment.

Of particular interest in this experiment is the development of interesting behavior that was found. The first of these behaviors was a tendency for units to run away when only 1 ally

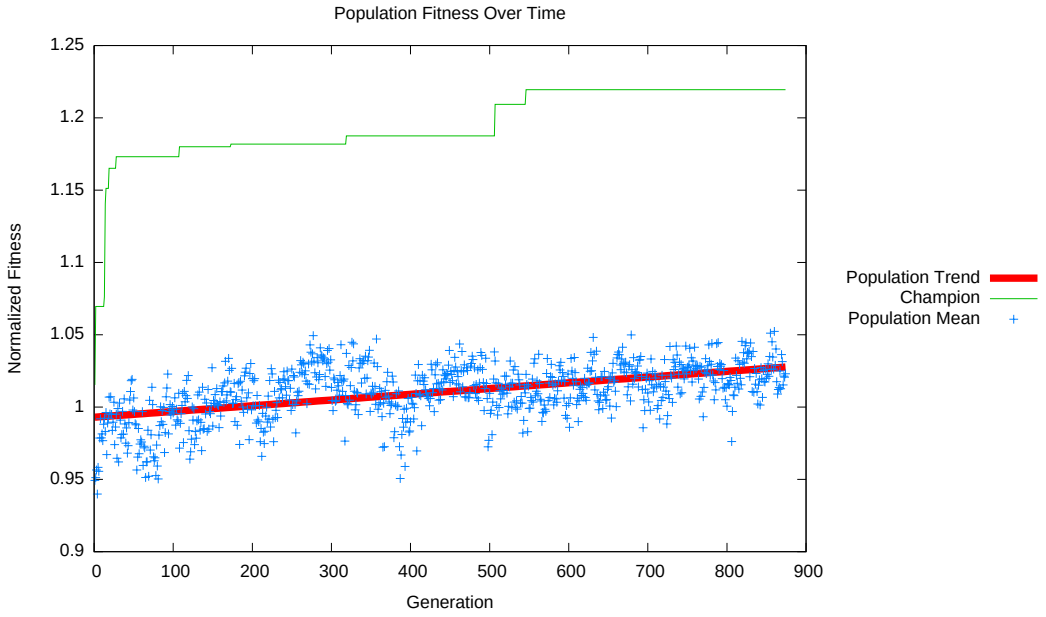

Figure 5.1: A fitness graph of the multi-unit controller in a 3v3 scenario, with 2 marines and 1 firebat on each team. Fitness progresses slowly over the threshold of 1.0, indicating that the average genome defeats the enemy units more often than not.

is left in battle. Our original fitness formulation didn't assign any fitness value when battles ended due to a timeout. Due to the stochastic nature of these battles, a team with occasional wins could avoid losses entirely by running away when such a loss was imminent. The second interesting behavior regards the dynamics of firebats and marines. Firebats do a large amount of damage, so killing them as early as possible at as great a range as possible is preferable. In these training sessions we saw a behavior surface in which the allied firebat would go straight for the enemy firebat at the beginning of the battle, and then switch targets to a marine when the enemy firebat had gotten low on health. This dealt out a maximal amount of damage to the enemy firebat initially, while in some cases preserving the allied firebat in the process. It also ensured that allied marines could keep a safe distance from the enemy firebat. A video of this behavior is given in Section **??**.

6 Conclusions

7 Future Work

8 Videos

Video	**Description**
`http://www.youtube.com/watch?v=OeMwC0bexlo`	3v3 battle with 2 marines and 1 firebat on each team

References

[1] B.G. Weber, M. Mateas, and A. Jhala. Building human-level ai for real-time strategy games. In *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, pages 329–336, 2011.

[2] J. McCoy and M. Mateas. An integrated agent for playing real-time strategy games. In *Proceedings of AAAI*, pages 1313–1318, 2008.

[3] Louis Victor Allis. Searching for Solutions in Games and Artificial Intelligence , 1994.

[4] C. E. Shannon. Computer chess compendium. In David Levy, editor, *Computer chess compendium*, chapter Programming a computer for playing chess, pages 2–13. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[5] Jacob Schrum, Igor V. Karpov, and Risto Miikkulainen. UT^2: Human-like Behavior via Neuroevolution of Combat Behavior and Replay of Human Traces. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2011)*, pages 329–336, Seoul, South Korea, September 2011. IEEE.

[6] Starcraft AI Competition — Expressive Intelligence Studio. `http://eis.ucsc.edu/StarCraftAICompetition\#Results`.

[7] Setfan Wender and Ian Watson. Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game Starcraft:Broodwar. In *Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game Starcraft:Broodwar*, IEEE IG 2012. IEEE Computer Society, 2012.

[8] Kenneth Owen Stanley. *Efficient evolution of neural networks through complexification.* PhD thesis, The University of Texas at Austin, 2004. AAI3143474.

[9] bwapi - An API for Interacting with Starcraft: Broodwar. `http://code.google.com/p/bwapi/`.

[10] bwapi-mono-bridge. `http://code.google.com/p/bwapi-mono-bridge/`.

[11] SharpNEAT. `https://sites.google.com/site/sharpneat/`.

[12] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, pages 653–668, 2005.