Real-Time Adaptive Strategies for StarCraft: BroodWars

Lucas Silva

Instituto Superior Técnico

Abstract. In this document we introduce the idea of an adaptive strategy for the *StarCraft: BroodWars* test bed. The motivation behind such strategy is that only one agent has yet been capable of playing and defeating a professional level player and most of the existing agents have non-adaptive strategic components. This lack of adaptability renders agents unable to react effectively to certain unforeseen strategies that a human can quickly come up with just by observing and exploiting these limitations. The proposed architecture will behave in a way that it can adapt to any of these situations without the need of coding specific strategies. We will also discuss the results obtained by applying this adaptive strategy and how to further improve its efficiency.

Keywords: StarCraft; Adaptive; Real-Time Strategy; Macro-management; Genetic Algorithm.

1 Introduction

Starcraft is a real-time strategy based game (RTS) released by Blizzard Entertainment in 1998. The game consists of a fog-of-war state in which the only objective is to destroy the enemies' units and buildings. To achieve this, we are given a choice between three races that can, in total, build x units and y buildings while managing and gathering resources to build said units and buildings of our own. The game popularity and professionalization (*Starcraft* became a professional electronic sport in 1999) led to the creation of standard text book strategies widely used throughout the community.

In 2017 Blizzard Entertainment released and API for *Starcraft 2*, the latest game in the franchise, and earlier this year Google's DeepMind released *AlphaStar*, the first bot to ever defeat a professional human player in any game of the franchise [6]. *AlphaStar* uses a deep neural network that is trained directly from raw game data through the use of supervised and reinforcement learning as a base start. The bot was later trained against multiple instances of itself in a competition ladder faction (in order to improve even further) until it was able to beat Team Liquid's Grzegorz "MaNa" Komincz, one of the world's strongest professional *Starcraft 2* players, five matches to zero, in December 19, 2018 .

1.1 Problem Definition

It is easier for a bot than a human to micromanage units, as bots can easily control single units individually, even in large armies, making their efficiency a lot higher in battles. As such, macro-management is the aspect of the game in which human players best bots. Although there are several "text book" strategies, they are not very flexible when strictly applied, even when a bot changes between several of these strategies, they may not be well adjusted to the current enemy strategy, thus having low efficiency. Being able to adapt in reaction to the enemy strategy is the greatest advantage human players have against bots, which makes the use of algorithms that create adaptive strategies a step towards closing this gap.

With this in consideration, this thesis addresses the problem of how to create an adaptive model for strategy generation in *Starcraft* that is able to counter an opponent's strategy.

1.2 Hypothesis

Due to macro-management being the main advantage point human players have against bots, in this paper it's proposed an adaptive strategy module for *Starcraft's* bots, that uses as its core, a genetic algorithm. The main purpose of said algorithm is to be able to create its own strategies while maximizing unit advantage and minimizing the resources spent. For this to happen, the bot scouts the enemy base location and army in order to be able to adapt to the enemy current strategy.

Being easy to implement and adaptable to other games and/or purposes, a genetic algorithm was chosen to implement the proposed algorithm as it can quickly provide results, proving the validity of the hypothesis.

2 State of the Art

This section's main focus is to provide detailed information over the *Starcraft* competition, as well as the current researches developed for this environment. It also details the current uses of some algorithms and their strengths and weaknesses.

2.1 Genetic Algorithm

Genetic algorithms (GA)[7, p. 619] are adaptive heuristic searches that make use of a set o methodologies inspired by biological natural selection methods greatly used to solve optimization problems. Genetic algorithms make use of biological mechanisms to generate better offspring's, enclosing optimal solutions.

These mechanisms can be defined, in this context, as the following:

· **Selection** - The process of selecting the fittest individuals among the population;
· **Reproduction** - Generating an individual consisting of the mix between two previously selected ones, with random partitions of said individuals;
· **Mutation** - Randomly introducing alterations on the individual, based on a probability;
· **Recombination** - Basically the same as reproduction, but with fixed partitions of the individuals.

The *GA's* simulate a survival of the fittest natural selection over consecutive generation of individuals, where each of these individuals represent a possible solution and a point in the search space that will then suffer from the biological mechanisms introduced above, generating fitter and fitter offspring, tending to better solutions.

These algorithms usually begin by randomly generating a population of individuals that will be scored according to a fitness function. From this population the highest scored individuals are selected for reproduction and will then be submitted to mutation and recombination. This process is then repeated until the terminating conditions are met, usually a time limit, number of generations produced or a sufficient score is achieved. Although some conditions and weights, or even mechanisms utilized may vary from different implementations in genetic algorithms, they all follow this general description.

This kind of approach is widely used in the generation of autonomous agents in a wide range of different applications, one of which is in real-time strategy games, and has shown great results, especially when conducted in coarser granularities (using *macro actions*)[10].

Although having a wide range of applications, genetic algorithms usually suffer from a single type of problem. The most difficult aspect when applying a genetic algorithm to a problem is defining what kind fitness function to use, how long the algorithm will run for, the size of the population, the probability of mutations and selection and how to define the parameters, in summary, subjectivity, as these values are problem specific and cannot be reused from different problems and implementations.

2.2 Build Order Optimization

One of the main differences in turn-based to real-time strategy games is time itself. In real-time strategy the game state changes, even though a player has taken no action whatsoever, while in turn-based strategy the game states only changes because one of the players performed an action. This makes it so that resource gathering and allocation are important components when applying strategies in *Starcraft*, as a efficient resource management can speed up the players response time, possibly changing the tide of a match. In order to address this issue, Michael Buro and David Churchill proposed a build order optimization algorithm[4].

The proposed algorithm, given an objective strategy (which units and buildings the bot intends to build), will return an array of units and buildings that the

bot will follow to better allocate its resources and quickly achieve its objective. This algorithm uses a depth-first branch and bound search from a starting state S until it satisfies a goal G. In order to make this algorithm implementation possible, three key features were built in:

· **Action Legality Check** - This feature ensures that a child node of any given state can only be generated if the action required for this state to be achieved would be valid;
· ***Fast-Forwarding Simulation*** -This feature's purpose is to ignore null-actions, greatly reducing the search span, by simulating the state the game would be in if null actions are taken, assuming the resources, buildings and units constructed as if they were gathered/built instantaneously;
· **Macro Actions** - Grouping actions in macro action will reduce the search span while also guaranteeing that known effective set of actions are always taken as a singular action;

Result analysis on this approach show that the build orders produced are comparable to that of professional *Starcraft* players, being able to defeat non-trivial opponents.

2.3 Strategy Prediction

One of the main advantages human players have against bots is the ability to predict strategies. As simple as this advantage may seem, experienced players, are able to recognize and adapt, early in the game, to an opponents strategy, simply by scouting the enemies units and buildings. Having this in consideration, Henrik Sørensen and Johannes Garm Nielsen proposed a prediction algorithm for *Starcraft*[9].

This algorithm consists of a multi-layer perceptron based system that uses replay analyses as a training data pool, with a resilient propagation algorithm used to train the strategy prediction. Result analysis shows great improvements when comparing this algorithm to the statistically best guess.

Strategy Prediction algorithms, such as this one, work great when paired with automated strategy generation algorithms, as the one proposed in this document, making it so that a bot can respond to the opponent's goal strategy rather than the current one.

2.4 Toward Automatic Strategy Generation

In order to demonstrate the viability of bots adaptability for *Starcraft*, Pablo García-Sánchez et al[5] conducted a study by creating a genetic algorithm and using two different fitness functions. These two fitness functions consist of distinct ways to measure the reliability of its agent performance. One of the fitness functions followed a victory based approach, while the other followed a report based approach and both were later compared, after playing several matches against bots with hand-coded text-book strategies.

The victory based function consists of using the final score returned by *StarCraft* at the end of each match held, and the report based function is a more complex one, separating military and economic development. Although focusing only on the strategical part of the game, these bots managed to achieve good results against fully coded bots with hand-coded strategies.

Result analysis in this paper showed that, although only focusing on the strategical aspect of the game, the victory based approach was capable of beating complex *non-adaptive* agents with different sets of strategies.

3 Methodology

The central concept in this paper is the idea of having a bot use an algorithm that is able to generate, in run-time, reliable strategies in order to gain advantage over its opponents. The following subsections detail the different components of the algorithm proposed, and figure 1 shows how they intertwine and communicate.

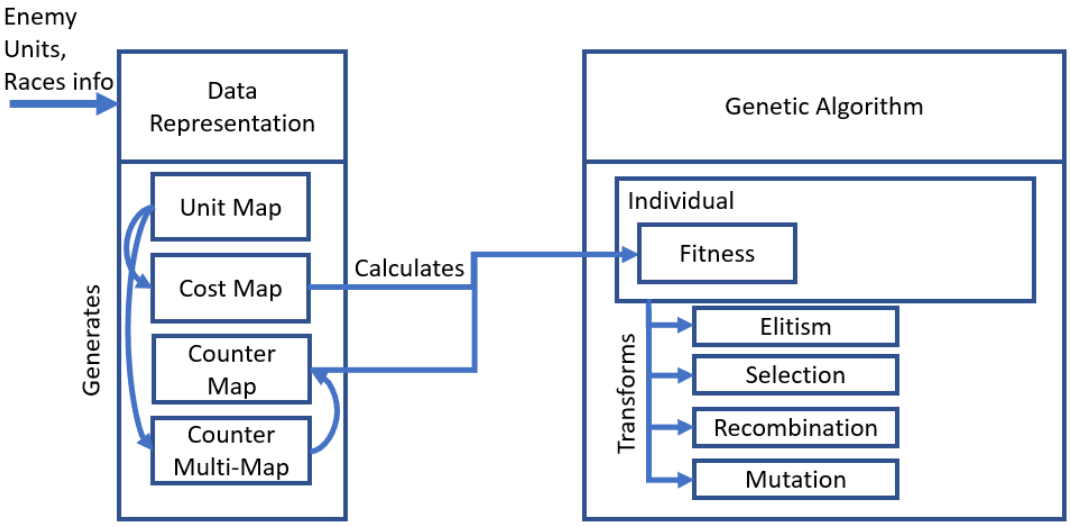

Fig. 1: Data treatment and algorithm structure.

3.1 Algorithm Overview

The algorithm takes into account the enemy race and units built in order to score the units it can produce. The better a unit is against the opponents, the higher its score. The genetic algorithm will then calculate which units should be built and how big should the army be, so that it can gain advantage over the enemy army, while minimizing the cost of production.

3.2 Data Treatment

As the first enemy unit is seen, the algorithm stores data about which race the opponent is using, initializing the unit and cost maps as well as the counter multi-map. The unit map is responsible for literally mapping the possible units the bot can produce into integers, facilitating the information treatment. The cost map stores the production cost of each unit the bot can produce. The counter multi-map, stores information about the relationship between the units the bot can produce and the units the enemy can produce, keeping track of which units are strong, or weak, versus which.

The algorithm will then calculate a counter value for each of the units the bot can produce, this value represents how well the unit type would fare versus the opponent current units. These values are then mapped to each unit, storing only the units with non-negative counter values are stored, along with their corresponding value, on the counter map. Figure 1 shows how these structures link to one another.

3.3 Fitness

As a way to minimize the production cost while maximizing units efficiency, the fitness function was created as a simple subtraction between the total counter value of each unit in an individual from the genetic algorithm's population, and its production cost. In order to achieve this, a division, instead of a subtraction, could be used. The reason behind a subtraction was used, instead of a division is because, the division version of the fitness (x/y) is a non-linear function. Non-linear functions are a lot more complex to compute than linear functions and since the genetic algorithm proposed is used in run-time, several times per match, and the fitness value for each individual is calculated in every generation in a single run of the algorithm, there is a significant impact on the time the algorithm would take to run. Figures **??** and **??** show the graph generated by the non-linear (x/y) and linear $(x - y)$ functions, respectively.

To calculate the counters value $(Counter_{(individual[i])})$ it is only needed to sum each unit counter value, stored in the counter map (1), from each unit in the current individual being evaluated in the fitness function.

Similar to the way the counter value is calculated, the cost value $(Cost_{(individual[i])})$ is also calculated as a sum. The difference between the two variables being that the cost value is calculated using the cost map (1).

$$Cost_{(individual[i])} = \sum_{n=0}^{individual.size()} CostMap_{(individual[i],gene[n])} \tag{1}$$

As the cost and counter values have different orders of magnitude, the cost value is normalized according to following function:

$$normalization = \frac{individual.size() \times enemyUnits.size()}{MaxCost_{(race)}} \tag{2}$$

Situationally, in a *Starcraft* match, fast unit production may be more valued than a few strong units that can be overwhelmed by numbers. As measure against, this algorithm assigns weights to the cost and counter values in the fitness function. The weight value is mutable, depending on the enemy army size and is calculated as shown in the following function.

After normalizing the counter and cost values, since they are independent from one another and both range from zero to $maxCounter$, it would be possible for the fitness of a given individual to be a negative value. Some of the genetic algorithms mechanisms, namely the proportional roulette wheel selection algorithm, struggles with negative fitness values. As a measure avoid this problem, a constant $maxCounter$ is added to the fitness value of every individual, and is calculated once for every run of the genetic algorithm.

$$maxCounter = individual.size() \times enemyUnits.size() \tag{3}$$

$$\begin{aligned} Fitness_{(individual[i])} = \\ & maxCounter + (Counter_{(individual[i])} \times weight) \\ & - \\ & (Cost_{(individual[i])} \times (1 - weight) \times normalization) \end{aligned} \tag{4}$$

3.4 Algorithm Configuration

To maximize the effectiveness of the genetic algorithm, several configurations were tested, using varying inputs, so that the best configuration for this specific problem could be found.

Initially the algorithm used two stopping conditions, a maximum number of generations, set at two hundred, and a verification that if the best solution was the close to the previous best solution, the algorithm would stop. Firstly, it was verified that in order to avoid local maximums the second stop condition had to be ignored, and so it was removed from the algorithm. As for the first stop condition, empirical results showed that there was little to no difference in running the algorithm for one hundred, or two hundred generations, and since this algorithm is used at runtime, the faster solution was chosen.

In a *Starcraft: Broodwars* match, two hundred is the limit to the number of units a player can have at a time and as such, each individual is represented by a vector of two hundred units. Although this is true, when defining which units to build to gain strategic advantage, defining two hundred units as a counter strategy is not viable. In order to go around this problem, the individuals in a population having varying size, depending on the size of the enemy army (units that swarm are considered in double), this way we can ensure that the bot will not overproduce units, nor overspend resources.

As a way to ensure the best solution was kept, the elitism mechanism was used, making sure the best result would continue on to the next generation of the algorithm. The genetic algorithm also uses the two-point crossover algorithm as

a method for the individuals recombination, as it offered a higher flexibility and better results when tested against other methods for the recombination purpose.

The selection and mutation algorithms were used in such a way that they would let the genetic algorithm avoid local maximums. To do so, the mutation mechanism would change a random chromosome in an individual genome with a given probability. This probability would be higher in early generations (maximizing diversity) and lower in later generations (converging faster in order to achieve the best possible result). The selection algorithm chosen was the roulette wheel selection. Yet again this was done based on empirical testing, comparing several configurations for this genetic algorithm. The only exception to this was the mutation algorithm. The non-uniform mutation algorithm was chosen simply because it was easier to work with given the individual data representation as integers.

Instead of testing each possible algorithm singularly, different possible algorithm configurations were tested instead, using three sets of inputs, ten times each, for any of the possible configurations. The obtained fitness and the time spent in each test was taken in consideration when choosing the best configuration for the algorithm. The proportional roulette wheel selection with the two points crossover algorithm was the configuration that achieved the best results regarding the time spent and fitness values, justifying the genetic algorithm's current configuration.

4 Results and Discussion

In this section, we analyze the results of applying the proposed algorithm to *UAlbertaBot*. The results here discussed were obtained through several matches against bots using different races and opening build orders. Test results stored information on the algorithm run time, how many times it ran during a single match, and how good was the strategy resulting from the algorithm's output.

4.1 Results

Throughout the matches it was verified that the algorithm ran every time a change was detected in the enemy units, corresponding to the objective regarding this aspect.

The most challenging aspect to test, was the algorithm's output. Military victory does not necessarily mean that the army had an advantage over the opponent. Depending on the bot's efficiency in micromanaging units, or how fast it can produce them, a battle outcome can be unforeseen. Because of this, unit by unit analysis was done to every output of the algorithm throughout every match done during the tests and it was verified that the algorithm prioritized a mix of units with low cost of production and efficiency versus the enemy units (good counters), depending on the situation, as intended.

Although the algorithm objectives were achieved, when playing against other bots, the match results were not as good as intended, and the reasons why this happened are discussed below.

4.2 Discussion

Match results against other bots showed that, although the algorithms worked as intended, there is a long way before the bot can play in an efficient manner. Matches against *UAlbertaBot's* base version, with the *Zerg* race, in a mirror match-up, had a one hundred per cent win rate, but if *UAlbertaBot* was to play any other race the bot would struggle. After analyzing each match, it was observed that, while playing the *Zerg*, *UAlbertaBot*, and by extension, our bot, had some worker management issues, translating in a delay in the bot's ability to respond in time to incoming threats. Contrary to when using the *Zerg*, the *Protoss* race, the bot's original choice of race to play with, won every match-up, except for the mirror match-up.

UAlbertaBot would also struggle when different kinds of resources were needed in order to produce units and buildings, regardless of the race being played. The bot would assign too many workers to a single resource and very few to the other. This lead to a halt in production of units that need the neglected resource to be produced.

Another aspect that hindered the obtained match results, was the fact that every time the algorithm changed the objective strategy *UAlbertaBot* would prioritize producing units with higher production cost, instead of units it could already produce. This meant that units lost in combat were not being replenished fast enough, rendering the bot unable to defend.

With all this behavioural divergences between *UAlbertaBot* and how the proposed algorithm intended it to react, a conclusion was reached in which *UAlbertaBot* is too incompatible with the proposed algorithm for it to translate into positive match results.

5 Conclusion

The tests scenarios could be subdivided into two different groups. In first group both agents suffered from the same handicaps (lack of a proper resource gathering and worker management). In this scenario, corresponding to the *Zerg vs Zerg* and *Protoss vs Zerg* match-ups, the agent was able to achieve victory in every single match.

In the last scenario, however, the agent was mainly defeated even though that the algorithm here proposed behaved as intended. In this scenario, it was verified that the agent tested had a handicap when it comes to resource gathering and worker management, while the opponent behaved normally, with no handicaps. This happened because in *Starcraft* every decision has a weight on the match outcome and the improvement of a single aspect cannot show significant differences if the other aspects of an agent are lacking. Although this may be truth, the single aspect that most contributed to the negative game outcomes in some scenarios, was that the original agent was created with some specific strategies in mind. This resulted in an incompatibility between the way the units are managed and prioritized, and how the algorithm proposed expected them to work. This

incompatibility resulted in faulty resource gathering. Instead of spreading workers according the resources needed the most, the workers were assigned in a fixed manner, and a faulty prioritization when deciding which units to build first after a change in its strategy, as the agent would focus on the most expensive units instead of the cheaper ones, limiting the agent's ability to quickly respond to the opponent.

As so, to fully seize the capabilities of the algorithm presented, we intended to upgrade the current agent and fix some of its existing issues.

5.1 Future Work

Although the algorithm achieved good results, in order for it to be the most effective it can be, a few improvements can be implemented. The algorithm implemented gives information about which units should be built to gain military advantage, but another algorithm can be implemented to determine when and how many defensive buildings should be built, or even how many copies of strategic buildings should be used.

Another aspect that can be improved is the need to upgrade technologies in order to make the best use of the units built so far.

Lastly a prediction algorithm can be implemented. It would feed information to the algorithm here presented so that its response would be faster and always one step ahead of the opponent. Based on the enemy units, their buildings and our own units, said algorithm could predict the enemy response. This would allow for the algorithm presented to react to the strategy the enemy is going for, instead of seeing it first and then reacting.

References

1. Bellemare, M.G., Ostrovski, G., Guez, A., Thomas, P.S.: Increasing the Action Gap : New Operators for Reinforcement Learning (2012)
2. Bishop, C.M.C.C.M.: Pattern recognition and machine learning. Pattern Recognition 4(4), 738 (2006)
3. Churchill, D.: UAlbertaBot (2011)
4. Churchill, D., Buro, M.: Build Order Optimization in StarCraft pp. 14–19 (2007)
5. Cig, I.: 2015 IEEE Conference on Computational Intelligence and Games (2015)
6. Google's DeepMind: AlphaStar, https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/
7. J. Russell, S., Norvig, P.: Artificial Intelligence: A modern approach (1995)
8. Nguyen, K.: Potential flows for controlling scout units in StarCraft. The Proceedings of the IEEE Conference on Computational Intelligence in Games 2012(Sscai 2012), 344 –350 (2013)
9. Nielsen, J.G.: Strategy Prediction in StarCraft : Brood War using Multilayer Perceptrons. Strategy (2011)
10. Samothrakis, S., Roberts, S.A., Perez, D., Lucas, S.M.: Rolling Horizon methods for Games with Continuous States and Actions (2014)