TÉCNICO LISBOA

Real-Time Adaptive Strategies for Starcraft: BroodWars

Lucas Terto Noblea da Silva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João Miguel De Sousa de Assis Dias

Examination Committee

Chairperson: Prof. Rui Filipe Fernandes Prada
Supervisor: Prof. João Miguel De Sousa de Assis Dias
Member of the Committee: Prof. Pedro Alexandre Simões dos Santos

June 2019

Acknowledgments

I would like to thank the Instituto Superior Técnico, and professor João Dias for supporting this topic and going forward with the theme of an agent for *StarCraft: BroodWars*, but also for all the help and advices given throughout the semester that were crucial to the work here presented. I would also like to thank my colleague André Lopes for his advices and discussions that also helped shape this document and improve the solution presented.

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 almost no 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 this 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.

Resumo

Neste documento é introduzida a ideia de estratégias adaptativas para o *Starcraft: BroodWars*. A motivação por trás de tais estratégias adaptativas é a de que apenas um agente foi capaz de derrotar jogadores profissionais. Para além disso, a grande maioria dos agentes existentes usa estratégias fixas, codificadas *a priori*. A falta de adaptabilidade aos oponentes faz com que os agentes sejam incapazes de reagir de forma eficaz a estratégias imprevistas, algo que um humano, rapidamente, consegue fazer pela observacão do comportamento do seu oponente, facilmente explorando as suas limitações. Este documento propõe, desta forma, uma arquitectura que pemitirá aos agentes adaptarem-se a variadas estratégias utilizadas pelo oponente, sem qualquer necessidade de codificação específica para qualquer tipo de estratégias. Iremos também discutir os resultados obtidos e possíveis melhorias à arquitectura proposta.

Palavras Chave

Adaptativa; Estratégias a Tempo Real; Macromanagement; Algorítmo Genético;

Contents

List of Figures

1 Introduction

Contents

1.1 Motivation

Strategy Game is a term used to label video or board games in which the player's decision making skills have a long therm impact on the game's outcome, as opposed to platform games, for example, in which the decision making skills have an immediate effect. Strategy games can be sub-categorized as turn-based or real-time. There are very well known examples of turn-based strategy games such as *Chess* and *Go*. Although these games complexities cannot be denied, their complexities are much lower than real-time strategy games such as *Starcraft* [4].

Starcraft is a real-time strategy based game (RTS) released by Blizzard Entertainment in 1998. The game consists of two or more players, facing each other 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 units and buildings of our own. Throughout a match, players do not have information over their opponents actions (fog-of-war), having visibility only over their own units line of sight.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.

Starcraft is considered one of the most well balanced *RTS* games ever, which led to the creation of an API in 2009 and, as a result, in 2010 the first annual *Starcraft* AI competition was held [4]. The competition's goal is to have an agent play and defeat a professional human player. In order to do so, all participating agents face each other a fixed number of times and the winner plays against a professional human player in a *best out of three* fashion. Although the competition has been around for almost ten years, only one agent has been able to defeat a professional player in *Starcraft: Broodwars*.

In 2017 Blizzard Entertainment released an API for *Starcraft 2*, the latest game in the franchise, and earlier this year Google's DeepMind released *AlphaStar* the first agent to ever defeat a professional human player in any game of the franchise [5].

1.2 Problem Definition

The problem of creating a *StarCraft* agent that is able to play the game, can be further subdivided in several sub-problems:

· **Scouting** - Scouting consists of using units to reveal remote areas of the map in order to gain information over the opposing player, such as their base(s) location, units and buildings built. With that information one can deduce possible build orders and overall strategy.

· **Resource Gathering** - Resource gathering, as the name implies, is the use of worker units to gather resources. As simple as this may be, there is more than one kind of resource in *Starcraft* and creating enough workers to gather the needed resources optimally is very important to the overall speed and quality of an agent.

· **Building Placement** - Simple building placement can be an important asset on *Starcraft* as it can help better defend your forces. This can be done either by the simple use of defensive buildings or by placing the buildings in a way that they are able to prevent enemy forces from going forward [1].

· **Micromanagement** - Micromanagement is a term used to describe how units are handled by the player, which includes how the units are moved, placed, use abilities, when they attack, or even where they retreat to. When done correctly micromanagement alone can turn tides in a battle or even change the game in the player's favour.

· **Macro-management** - Macro-management refers to everything that has to do with a player's economy. How to spend the resources, although seemingly simple, is no easy task as the player has to decide what buildings and units to build, as well as which technologies to upgrade. This Basically means that all strategical aspects depend on how well the resources will be spent.

It is easier for an agent than a human to micromanage units, as agents can easily control single units individually, even in large armies, making their efficiency a lot higher in battles. Agents can also spread the units easily (avoiding area attacks), focus fire (picking off important enemy troops) and kite (attacking with ranged units while maintaining distance) [3]. Macro-managing is a completely different task and although there are several "text book" strategies, they are not very flexible when strictly applied, even when an agent 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 agents, With this in consideration, this thesis addresses the problem of how to create a model for automatic strategy generation in *Starcraft* that is able to adapt to an opponent's strategy.

1.3 Hypothesis

As macro-management, specifically strategy adaptation, is the main advantage point human players have against agents, can a genetic algorithm be used to generate adaptive strategies, improving an agent base behaviour, for *StarCraft: Broodwars*?

In this thesis it's proposed an adaptive strategy module for *StarCraft's* agents, that uses as its core, a genetic algorithm.

The main purpose of said algorithm is to be able to create its own strategies, in such a way that the agent will always have unit advantage against it's opponent, while spending the minimum required resources to do so. For this to happen, as soon as the agent is able to scout the enemy base location and the current units and buildings built, the adaptive strategy algorithm will run, changing the agents initial strategy to an advantageous one. To maintain this advantage, every time changes are observed in the enemy strategy so will the agent change its own by using the proposed algorithm.

Being easy to implement, not needing big amounts of data and being adaptable to other games and/or purposes, a genetic algorithm was chosen to implement the proposed algorithm.

1.4 Structure

This thesis will be divided in three chapters. In chapter 2 (State of the Art) the paper's background will be discussed, including: the current state of the *Starcraft* competitions, the work done in the area and algorithms used. From this point, chapter 3 shows how the solution will be implemented and how will it work. The chapter 4 shows the outcome of applying the proposed algorithm and lastly chapter 5 discusses means to improve it.

2 State of the Art

Contents

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 *Starcraft* Competition

In 2010 the first *Starcraft* competition was held. This competition had as purpose to showcase the AI developments to date in RTS games and to create an environment in which the community could compete in order to further develop the current state of their AI. The first agents developed would focus around a single strategy and through greater micromanagement they would defeat their opponents. Although their micromanagement was efficient, the overall strategy was not, making the agent vulnerable until a certain point in the game was reached [3]. Nowadays the number of participating agents increased, as well as the number of developers (most agents are created by a team of programmers), and so the quality of the agents increased.

Agents no longer follow straight forward strategies and they try to adapt to their opponents, using well-known text-book strategies and its known counters. Although being more versatile, these adaptations are inefficient, for example, given strategy *A* and its counter strategy *B*. If an opponent changes its current strategy to strategy *A* and my agent has strategy *B* coded, then it can react promptly, but, if another opponent would change its strategy to a variation of strategy *A* (*A'*), then my agents' response, with strategy *B*, may not be efficient and may even struggle to act on this strategy *A'*. This is true to the great majority of agents currently competing [4], with a few exceptions.

One of these exceptions is the *Steamhammer* agent [6]. This agent focus its strategy around two different units, a central and a support one. *Steamhammer* agent will pick, in the beginning of the match, a main unit with a support that works well with said main unit, and changes this main unit every time it seems fit to do so. Although able to adapt, this agent does so in a not so efficient manner, as it does not take into consideration unit production cost, or how well the support for the main unit will be against the enemy strategy. Besides, a two unit type army may not always be the best course of action (strategy wise).

2.2 Related Work

The following subsections will introduce several relevant algorithms used or studied for application in *RTS* games.

2.2.1 Generic Algorithms

2.2.1.A Genetic Algorithm

Genetic algorithms (*GA*) [7] are adaptive heuristic searches that make use of a set of 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. Each individual can be viewed as a set of genes, each gene representing a parameter or variable that together form a possible solution or individual. 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. One of this mechanisms, that can be used alongside selection algorithms is the elitism. This mechanism consists of always passing the best *X* individuals to the next generation, as a way to always keep the best solution achieved so far in the algorithm's iteration.

Some of the main points in which genetic algorithms may differ, relate to the choices of selection, reproduction and mutation algorithms.

One example selection algorithms is the proportional roulette wheel selection algorithm. This method consists of assigning a probability of selection to each individual, directly dependent on how high it's fitness is, the higher the fitness, the higher the probability of being selected. T he selection probability of each fitness can be calculated according to the following formula:

$$P_{selection(i)} = \frac{fitness(i)}{totalFitness} \tag{2.1}$$

As far as reproduction goes, the commonly used method is the two-point crossover algorithm. This algorithm consists of randomly selecting two points in a individual and exchanging the information contained to create the offspring. Figure 2.1 shows an example of how this mechanism works.

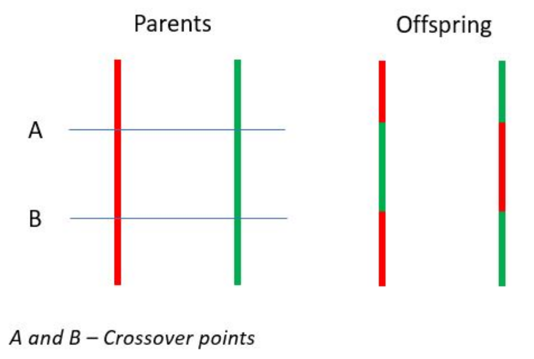


Figure 2.1: Two-point crossover.

The mutation mechanism is way for the algorithm to avoid getting stuck in solutions that often appear to be the best possible but are not. This mechanism usually has a probability of being triggered and said probability mat vary during the algorithms iterations. There are several ways to implement this mechanism, on of which is the non-uniform mutation. In this mutation algorithm the probability of mutation lowers with the increasing number of generations in the genetic algorithm and when the mutation happens it changes one of the individual's genes to a random value.

Genetic algorithms are 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*) [8].

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.

2.2.1.B Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (*MCTS*) [9] is a relatively recent tree based search algorithm, widely employed in game playing agents. This algorithm has shown great results for strategy board games, such as *Go*, which lead to its application in several artificial intelligence problems and games, such as *StarCraft* [10].

This algorithm consists mainly in four steps where the outcome (in this case, the game's outcome) will be simulated, through a series of *play-outs*, that will basically play the game until an outcome is reached (winning or losing), using random moves. These steps are consisted of:

· **Selection** - From the root node, successive child nodes are selected, down to a leaf node *L*;

· **Expansion** - Unless the leaf node *L* ends the game (win/loss for either player), generate node *L's* child nodes and of them, choose a child *C*;

· **Simulation** - From *C*, do a random *play-outs*;

· **Back propagation** - Update the information from node *C*, to the root of the search tree, using the outcome of the *play-outs*.

The move chosen through the algorithm will be the correspondent to the child node with highest visits, so that the strongest course of action is taken.

Studies show that *MCTS* presents better results than other algorithms while working with finer granularities, contrary to genetic algorithms, excelling when oriented to more concrete actions instead of more abstract ones, as controlling units, opposed to creating general strategies [8].

2.2.2 Applied Algorithms

2.2.2.A 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 [11].

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

2.2.2.B Strategy Prediction

One of the main advantages human players have against agents 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* [12].

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 an agent can respond to the opponent's goal strategy rather than the current one.

2.2.2.C Building Placement

A particular aspect in *Starcraft* is the need to create walls of buildings to defend your base location, because there are no specific construction with that purpose. This is often seen in competitive human play, as the employment of such tactics hinders the effectiveness of the opponent's early attacks and makes it possible for players to rely on mid-to-late game strategies.

This issue is addressed by Caio Freitas de Oliveira and Charles Andryê Galvão Madeira [1], focusing a particular race of the three possible race choices in *Starcraft*, the *Zerg*, which is the race that most benefits from this strategy.

In their approach, a mix of potential flows and a modified best-first search algorithm is used to fulfill several objectives that go from finding *choke-points*, to the creation of said walls. A terrain analysis algorithm locates the player's base and different resource locations, generating polygons of walk-able areas to better determine *choke-points*. The potential flow algorithm would then run, assigning different charges to the gaps in the buildings and *choke-points*, using then the modified search algorithm to create walls with building positioning. In Figure 2.2 we can see the gaps in buildings and the weights assigned that will be used throughout the algorithm.

A result analysis showed that this algorithm could create tighter walls than some proposed in *Star-Craft's* forums, but there is still work to be done. This algorithm has still to be tested using different maps, as it was run only in the "*Fighting Spirit*" map, one of the most played in *StarCraft's* library.

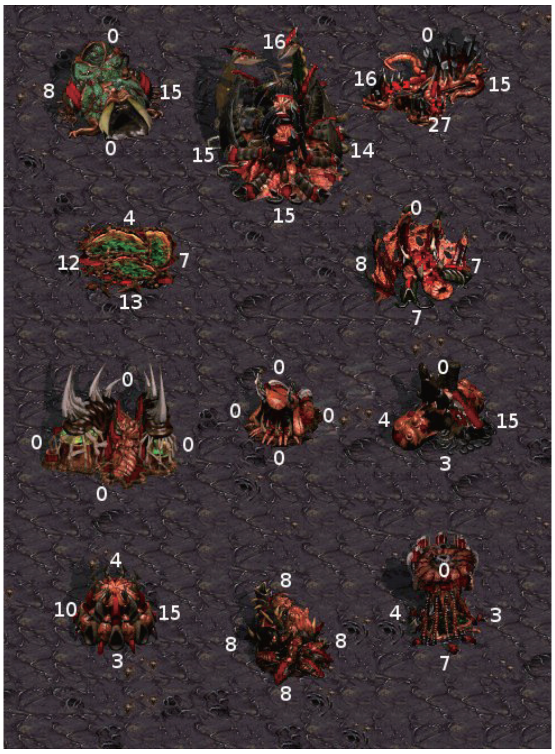


Figure 2.2: Gaps inside *Zerg* buildings. They were computed for each side of the building (left, right, top and bottom). As seen in [1]

2.2.2.D SOMA Swarm Algorithm

The SOMA (Self-Organizing Migrating Algorithm) swarm algorithm [13], as the name implies, is derived from the SOMA algorithm and focuses on the benefits of evolutionary algorithms.

This algorithm works by dividing different units as static and dynamic subspecies, for example, enemy base and military force, respectively. At the beginning a Leader is chosen, starting with the enemy base. Said Leader will be the target of the agent's forces (allies), meaning that the Leader will dynamically change depending whether the allied forces are under attack, in combat, or simply trying to destroy the enemy. As for the combat itself, other algorithms are used to decide if the allied forces should or not engage the enemy, making the SOMA algorithm responsible for moving the troops around the map, picking off the enemy forces, but not responsible for the combat itself working more as a targeter.

Upon testing the algorithm, text-book rush strategies were the main focus therefore, as it is the best fit for rush strategies, the *Zerg* race was used. During its experimental scenario, the SOMA agent won every match up held, against all the *StarCraft's* races, in a best out of ten games match. Although promising, one of this algorithm's limitations is that it works best in two-players maps, because the time needed to identify the enemy base location is smaller when compared to maps designed for a higher number of players.

2.2.2.E Replay Based Imitation

In an attempt to achieve results closer to that of a human player, work has been done regarding the analysis of *StarCraft's* professional matches, through replays. This work has been conducted by In-Seok Oh and Kyung-Joong Kim [2], achieving interesting results.

Their research focuses on simple attack or retreat group-level decisions, based on a frame matching mechanism to find similar situations to those of human players and then mimic their decisions. Figure 2.3 shows some aspects of this algorithm, that will be explained next.

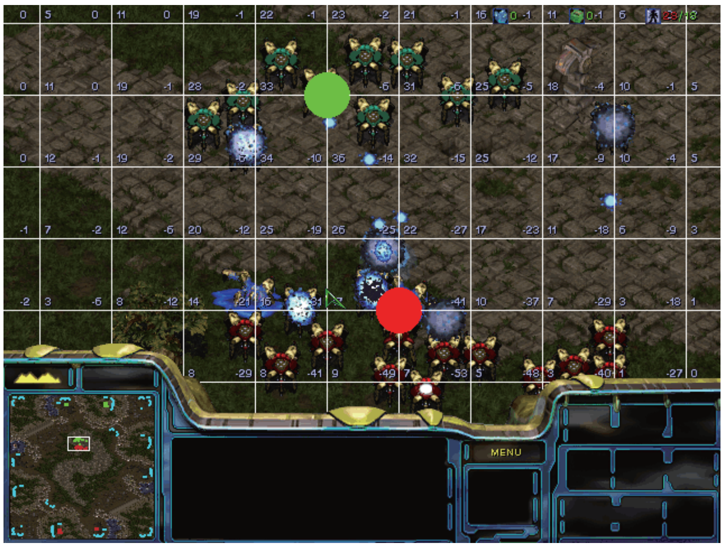


Figure 2.3: Influence map representation. For each cell, left-bottom number shows the agent's force; right-bottom number is for the enemy. The green and red circles point the location with highest influence. As seen in [2].

This algorithm firstly downloads replays (for one specific map only), using hashing techniques to speed up internal processing, and then compare spatial features between the replay and the current match. In combat engagement decisions the algorithm takes into account several features, such as, positioning, health, configuration, terrain and several others, creating influence maps to analyze the units spatial influence. While creating the influence maps, low-level unit-by-unit similarity check can be ignored to speed up the process.

Without any kind of filtering bad decisions in the human plays on the replays, this algorithm reached an overall of more than 80 percent correct decision, proving that greater results can be achieved by improving this work through the use of filtering algorithms and adding a map recognition feature so that this can be used throughout the array of available *StarCraft's* maps.

A great disadvantage in this algorithm is that it cannot exploit one of the main advantages an agent can have over a human player, that is single unit management. While a human cannot possibly move large groups of units singularly, for a computer this kind of task is simple, making it so that reacting in battle can be far more effective, turning seemingly disadvantageous situations in advantageous ones.

2.2.2.F Toward Automatic Strategy Generation

In order to demonstrate the viability of agents adaptability for *Starcraft*, Pablo García-Sánchez et al [14] 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 agents 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 agents managed to achieve good results against fully coded agents 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.

2.2.3 Agents

2.2.3.A *Overmind*

Overmind was the winner of the first ever held *Starcraft* autonomous agent competition, by *AIIDE* in 2010.

This agent focused its aim in managing *mutalisks*, a *Zerg* race, strong, agile flying unit, capable of attacking while on the move, with the only drawback being its weakness to anti-aerial, area attacks, as in human play they are usually controlled together, due to the impossibility of *micro-managing* them individually, resulting that only small numbers of this unit are used in human competitive plays. Although *mutalisks* have this known weakness, a computer would have no difficulty scattering and controlling huge numbers of this unit, making *mutalisks* armies a force to be reckon, if controlled by a computer.

Betting in this strategy through the use of potential fields to *micro-manage* the *mutalisk* army, training the agent against human players to improve *Overmind's* behavior, and a efficient scouting algorithm that searches for safe paths to follow, led *Overmind* to victory in this first competition.

Despite achieving first place in 2010, *Overmind* did not enter next years competition due to the fact that this agent's programmers realized a fatal flow in their strategy, their agent was weak against rushed strategies in the early game, being easily defeated by agents or players that invested in this kind of play.

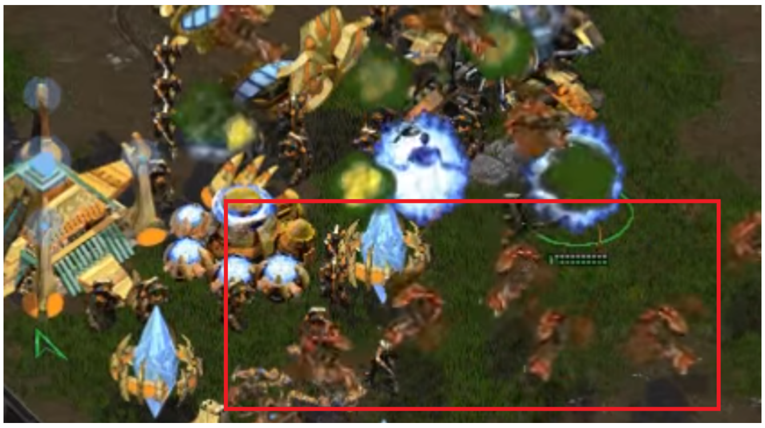


Figure 2.4: *Overmind's mutalisks* using the potential fields algorithm. As seen in [3].

2.2.3.B *UAlbertaBot*

UAlbertaBot [15] uses a "strategy selector" when facing an opponent, it's opening strategy will be chosen based upon previous matches against this specific opponent or randomly chosen in case it's the first time this match up is held. Throughout the rest of the match this agent uses a build ordering algorithm that aspires to find optimal building plans, to minimize the time needed to achieve a specific game state. This algorithm works as a constraint satisfaction problem solver, using macro-actions and a heuristic function to prune the tree, speeding up the search process.

During a match *UAlbertaBot* tends to start by rushing certain strategies, changing between other well-defined strategies every time certain conditions are met, in an attempt to adapt in disadvantageous situations.

As far as combat decision go *UAlbertaBot* runs a simulation of the outcome before deciding to run or engage.

UAlbertaBot has participated in the *StarCraft's* competitions since it's first editions and has had good results in it's participations, and serving as a good starting point to new competitors since the developers share their well documented structure and code.

2.2.3.C AlphaStar

AlphaStar [5] is an *Starcraft II* agent created by Google's DeepMind. Upon it's release, in December 19 2018, it was capable of defeating a top leader professional player "MaNa", five matches to zero, becoming the first agent to ever defeat a professional human player. This agent uses deep neural networks as its core, training directly from raw game data using reinforcement and supervised learning methods. Initially, *AlphaStar* is trained using replays of human vs human matches, but as soon as the basic micro and macro strategies were grasped, it was then tested in a complex environment involving several copies of itself with different learning objectives in order to evolve differently.

The second phase of *AlphaStar's* training sessions consisted of a league, with new competitors dynamically added by branching existing ones in such a way that the agents could explore the huge strategic space of *Starcraft* gameplay, while ensuring competitors would perform well against the strongest strategies, not forgetting how to deal with simpler, earlier ones. The final *AlphaStar* version consists of a Nash distribution of the league it was trained in, during which each competitor played what is equivalent of 200 years of *Starcraft* each.

Even though able to defeat human players, *AlphaStar* executes a lower number of actions per minute than the average professional human player. This means that *AlphaStar*, although able to defeat human players, it does so with worse micromanagement than the average professional player. This means that some strategies requiring high micromanagement skills (higher than a professional player can achieve, but a computer can, with ease) cannot be used by this agent. One example of such strategies is the one used by *Overmind*, the winner of the first *Starcraft* AI competition ever held. *Overmind's* strategy was a simple *Mutalisk* rush, while using the *Zerg* race. This agent would amass a large army of only *Mutalisk* units, something not seen in professional human plays because this unit is weak against area attacks and humans struggle when moving large number of units one by one (something necessary to avoid area attacks), a task easy for a computer.

3 Methodology

Contents

A strategy in *Starcraft* can be defined as the choice of which units and buildings to produce. This choice can be decisive in a match, as the units have different advantages and disadvantages over each other and a production cost associated. Since the buildings to be built can be viewed as dependent on the units choice and can be easily calculated given the units intended to be produced, this thesis will focus on the unit production aspect of the strategical component of the game.

As an example of the units impact on the game, if a player produces a unit A, that has an advantage over the majority of units in the opponent's army, that opponent's army will very unlikely be able to fight back without suffering huge losses, meaning that the opponent is forced to quickly produce units that, at the very least, do not have a disadvantage against said unit A. As such, choosing the right units to build can have a huge impact on the opponents economy.

The central concept in this paper is the idea of having an agent use an algorithm that is able to generate, in run-time, reliable strategies in order to gain advantage over its opponent, using only information about the enemy buildings, army composition and units relationship (which have advantage over which). The following subsections detail the different components of the algorithm proposed, and figure 3.1 shows how they intertwine and communicate.

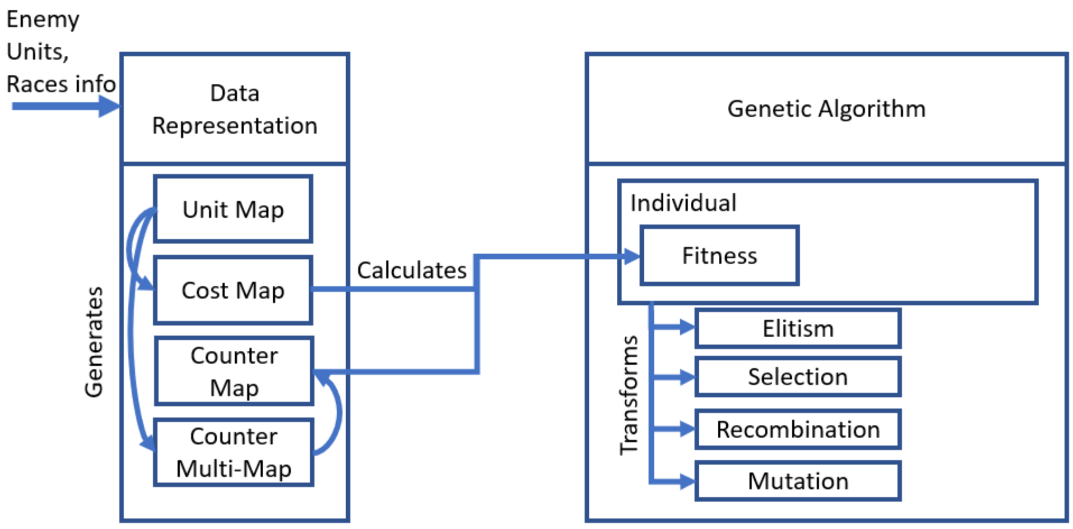


Figure 3.1: Data treatment and algorithm structure.

3.1 Algorithm Overview

The algorithm here proposed uses as input the enemy race and units in the enemy army built so far. With said information the algorithm will then calculate which units it can produce and score these units according to how well they would fare versus the enemy army, also taking into account their production cost. 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 Representation

This section main focus is to explain how the data is represented and why this representation was chosen among several ones tested in the algorithm.

3.2.1 Representation Alternatives

3.2.1.A Individual

Individuals in this algorithm represent the units to be built. The build order is later decided by other algorithms as it is out of this algorithm's scope.

In the earlier version of the algorithm the individuals where represented as a hash-map, the keys were data representations of the units that could be built by the agent and the values represented the number of each unit to build. This was a direct representation and for this reason it was the first one considered.

The map version of the individuals, upon testing, showed a fatal flaw. During the recombination of the individuals in the genetic algorithm, maps tended to be empty. This happened because a lower number of units produced translated in lower resources spent and, as such, higher fitness values, which lead this approach to be abandoned.

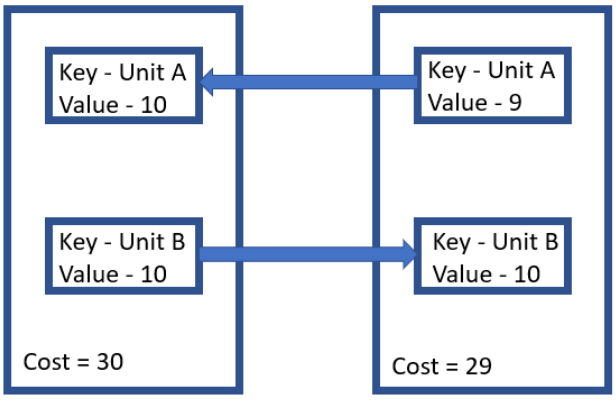


Figure 3.2: Recombination example using maps.

Figure 3.2 shows one step in the genetic algorithm and how the number of units to produce can easily tend to zero. In this example, assuming the total counter value for unit A is zero, the total cost will get lower after the recombination, making the fitness value higher, even though the only difference is that a lower number of units is gonna be built. Although we intend, in this example, to minimize the number of units A produced, we do not intend to do so at the cost of producing one less unit overall. The real goal would be to produce one less unit A and one more unit B. If this unit exchange does not happen the algorithm could end up producing half the units needed to gain advantage over the opponent, as even an army composed only of units that have an advantage over the enemy army can be overwhelmed if low on numbers. Another problem with this representation is that through recombination the highest value of a given unit cannot be increased, as the hash-map stores repetitions and not the unit themselves.

Since the map representation was unusable, a string of bits representation was tested next.

The string of bits representation of the individual was one in which the whole individual was represented as a string of bits, having each character in the string represented as either a zero or one.

With this configuration some of the possible mutation algorithms became faster. For example, the flip-bit mutation algorithm. This algorithm changes a random value in the string for the opposite one (a zero to one or a one to zero). In doing so the changed bit will generate a completely different integer, corresponding to a different unit.

Although this version of the individual representation made it easier to calculate some of the algorithms functions, when calculating the fitness, there was a constant need to transform sets of bits into integers, in order to map the representing value into a unit, using the unit map (the unit map seen in figure 3.1 is explained later in this chapter).

Another problem with this representation was the fact that the number of bits needed to represent the possible units to build was not an exact match, as the bits could represent invalid integers for the counter map. For example, if a given race can only build thirteen different units, four bits would be needed to fully represent the thirteen units but, four bits can be used to represent sixteen different numbers. This means that when a bit is flipped through the mutation algorithm invalid numbers could appear on the individual composition. This led to the need to create a function to transform these invalid integers into valid ones, but even this solution led to a problem. This meant that some units would have a higher chance to be mutated into than others, making it so that this approach had to be abandoned.

The final individual representation is that of a vector of integers.

With this representation each position of the vector has an integer that, using with the unit map, represents a unit possible to be built.

Another issue with the individuals, common to all representations, was its size, as it represented the number of units to build.

Initially, the size was fixed as the maximum number of units a player can build in *Starcraft*. In order to return viable strategies without the maximum number of units, a null unit was represented in the unit map. This unit has zero production cost and is neither good or bad against any other unit in the game, and was ignored in the fitness calculations. Although the null unit diminished the problem of outputting a high number of units to be produced, the resulting strategies still intended to build too many units, even after assigning weights to the fitness function, in an attempt to increase relevance to the production cost when calculating the fitness value (this would result in a higher number of null units in the algorithm's output, but still not enough).

As a way to go around this problem, the individuals size was then set to the enemy units size. In this way the agent wouldn't overproduce units. With this measure seemingly resolving the problem, another aspect had to be taken in consideration to finalize this issue. Some units are only reliable in large numbers and are usually cheap to produce, as such, the algorithm will double the given number of said units after choosing the best individual.

3.2.2 Final Representation

3.2.2.A Population

The population is comprised of two hundred individuals of varying size through the several runs of the algorithm.

This number was chosen in order to ensure diversity among the individuals without compromising too much of the algorithm's speed.

A low diversity can hinder the algorithm's results as it is easier for it to be stuck on local maximums upon runs.

The same way a low diversity can hinder the algorithm, too big population sizes can result in the same. This algorithm is used in run-time and it needs to ensure it is not too much time consuming, as an agent is a complex agent and many algorithms may be running at a given time and/or may need to run afterwards.

3.2.2.B Individual

The individual is set as an array of integers, and each of these integers (the genes) is mapped to a unit that could be produced by the current race.

The individual size is variable depending the enemy army size. This means that in a run of the genetic algorithm the individual size is fixed and the same as the enemy army size, but for different runs of the algorithm, the individual size will change, as the enemy army size will.

3.2.2.C Counters

The counter values, stored in the counter map (referenced in figure 3.1), represent how good is a unit type against the units in the enemy army. For each unit in the enemy army a given unit is good against, increases the counter value of said unit by one. The same way, for each unit on the enemy army a given unit is weak against, decreases its counter value by one. For example, if unit A is good against unit B and weak against unit C, and unit B is good against unit C, for an enemy army composition of three units B, the counter values for unit A, B and C would be three, zero and minus three, respectively, as shown in figure 3.3. This values are calculated, using the counter multi-map, as referenced in figure 3.1, every time the algorithm runs, as the enemy army composition changes between runs.

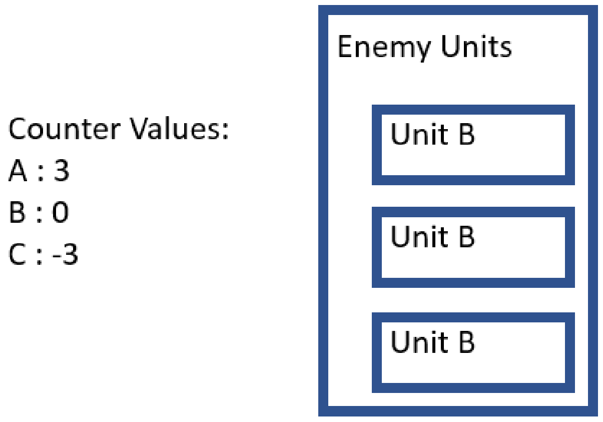


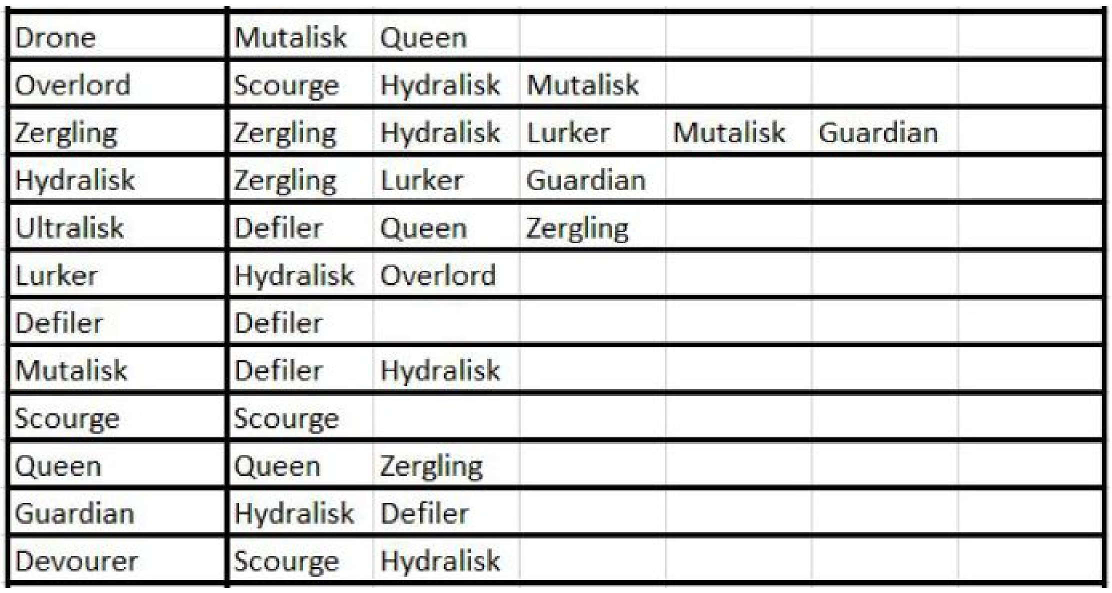
Figure 3.3: Counter values example, the higher the counter value, the better the unit against the enemy army composition.

The counter multi-map stores information about which unit is strong, or weak, versus which. Because this information is not available in-game, prior knowledge of the game and online forums held by a professional *Starcraft* team was used, in order to create the nine tables (one for each of the three races combination witch each other) needed to generate the multi-map.

As stated above *TeamLiquid*, a professional *Starcraft* team, maintains a forum regarding several aspects of the game, including unit counters [16]. The information in this forum was transformed into tables that were then fed to the algorithm, generating the counters table. Figure 3.4 show how the information was transformed and represented in the algorithm.

In order to calculate the counter values, the counter multi-map is treated as two different tables, one with information regarding the advantages held between the units the agent can produce and the units the enemy can produce, and another with the information regarding which units the opponent can produce are good against the agent's.

This algorithm was built in such a way that changing the information stored in the multi-map is enough for the algorithm to be able to play different games.



Drone	Mutalisk	Queen				
Overlord	Scourge	Hydralisk	Mutalisk			
Zergling	Zergling	Hydralisk	Lurker	Mutalisk	Guardian	
Hydralisk	Zergling	Lurker	Guardian			
Ultralisk	Defiler	Queen	Zergling			
Lurker	Hydralisk	Overlord				
Defiler	Defiler					
Mutalisk	Defiler	Hydralisk				
Scourge	Scourge					
Queen	Queen	Zergling				
Guardian	Hydralisk	Defiler				
Devourer	Scourge	Hydralisk				

Figure 3.4: Example of a *Zerg vs Zerg* counter multi-map. The units the enemy can produce on the left, my units that are good against them on the right.

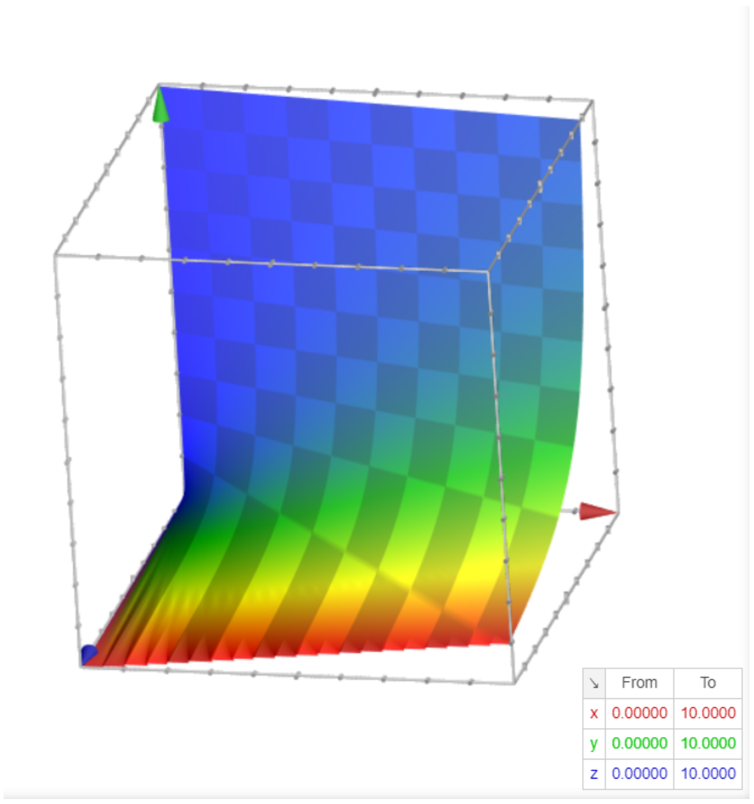
3.3 Data Treatment

As the first enemy unit is seen, the algorithm will record data about which race the opponent is using and will initialize the unit and cost maps as well as the counter multi-map. The unit map is responsible for literally mapping the possible units the agent can produce into integers, this mechanism is used to facilitate the information treatment. The cost map, as the name implies, stores information about the production cost of each unit the agent can possibly produce. The counter multi-map, on the other hand, is responsible for storing information about the relationship between the units the agent can produce and the units the enemy can produce, keeping track of which units are strong, or weak, versus which.

After initializing the structures mentioned above, the algorithm will calculate a counter value for each of the units the agent 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 and the units with non-negative counter values are stored, along with the calculated value, on the counter map. Figure 3.1 shows, in the Data Representation, how these structures link to one another.

3.4 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 3.5 and 3.6 show the graph generated by the non-linear (x/y) and linear ($x - y$) functions, respectively.



↘	From	To
x	0.00000	10.0000
y	0.00000	10.0000
z	0.00000	10.0000

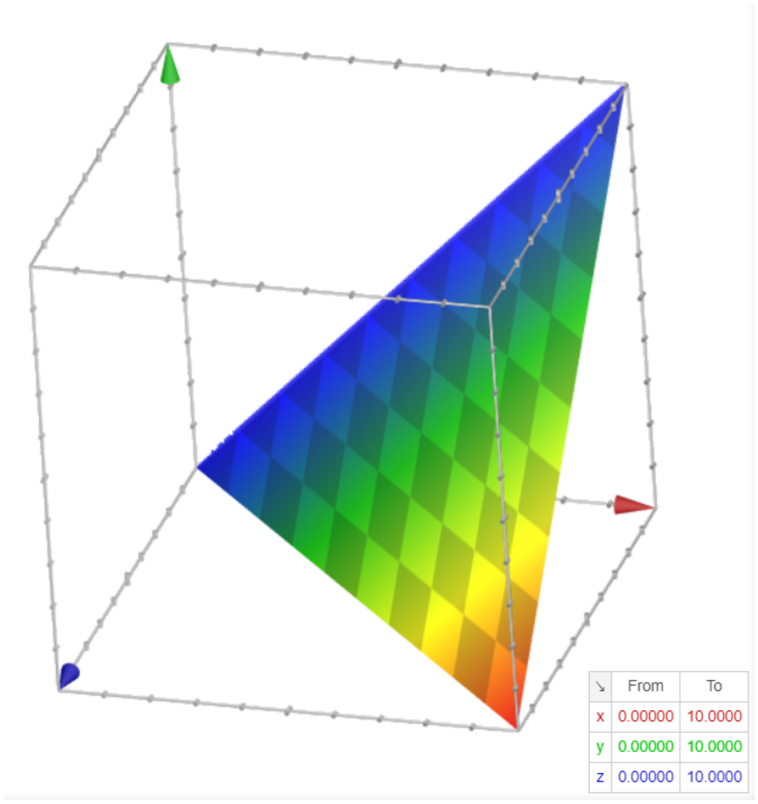
Figure 3.5: Hyperbole graph, generated by the function x/y.

$$Fitness_{(individual[i])} = Counter_{(individual[i])} - Cost_{(individual[i])} \tag{3.1}$$

To calculate the counters value ($Counter_{(individual[i])}$) it is only needed to sum each unit counter value, stored in the counter map (3.1), from each unit in the current individual being evaluated in the fitness function.

$$Counter_{(individual[i])} = \sum_{n=0}^{individual.size()} CounterMap_{(individual[i],gene[n])} \tag{3.2}$$

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



↘	From	To
x	0.00000	10.0000
y	0.00000	10.0000
z	0.00000	10.0000

Figure 3.6: Plane graph, generated by the function $x - y$.

cost map (3.1).

$$Cost_{(individual[i])} = \sum_{n=0}^{individual.size()} CostMap_{(individual[i],gene[n])} \tag{3.3}$$

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

$$normalization = \frac{individual.size() \times enemyUnits.size()}{MaxCost_{(race)}} \tag{3.4}$$

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.

$$weight = max[min[\frac{individual.size()}{200}, 0.3], 0.7] \tag{3.5}$$

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.

$$maxCounter = individual.size() \times enemyUnits.size() \tag{3.6}$$

$$\begin{aligned} Fitness_{(individual[i])} = & \\ & maxCounter + (Counter_{(individual[i])} \times weight) \\ & - \\ & (Cost_{(individual[i])} \times (1 - weight) \times normalization) \end{aligned} \tag{3.7}$$

3.5 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.

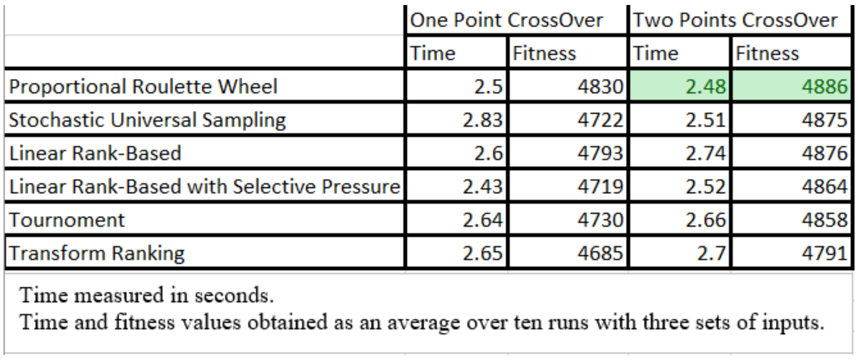
Initially the algorithm used two stopping conditions, a maximum number of generations, set at two hundred, and a verification that if the best solution was the close to the previous best solution, the algorithm would stop. Firstly, it was verified that in order to avoid local maximums the second stop 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 run-time, 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 size of the individuals, and as such, the number of units to be produced, is set as the size of the enemy army, this way we can ensure that the agent 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 gene 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. Figure 3.7 shows the configurations tested, as well as all algorithms used and the results obtained, justifying the genetic algorithm's current configuration.



	One Point CrossOver		Two Points CrossOver	
	Time	Fitness	Time	Fitness
Proportional Roulette Wheel	2.5	4830	2.48	4886
Stochastic Universal Sampling	2.83	4722	2.51	4875
Linear Rank-Based	2.6	4793	2.74	4876
Linear Rank-Based with Selective Pressure	2.43	4719	2.52	4864
Tournoment	2.64	4730	2.66	4858
Transform Ranking	2.65	4685	2.7	4791

Time measured in seconds.
Time and fitness values obtained as an average over ten runs with three sets of inputs.

Figure 3.7: Average results from testing the different configurations for the genetic algorithm in debug mode.

3.6 Integration with *UAlbertaBot*

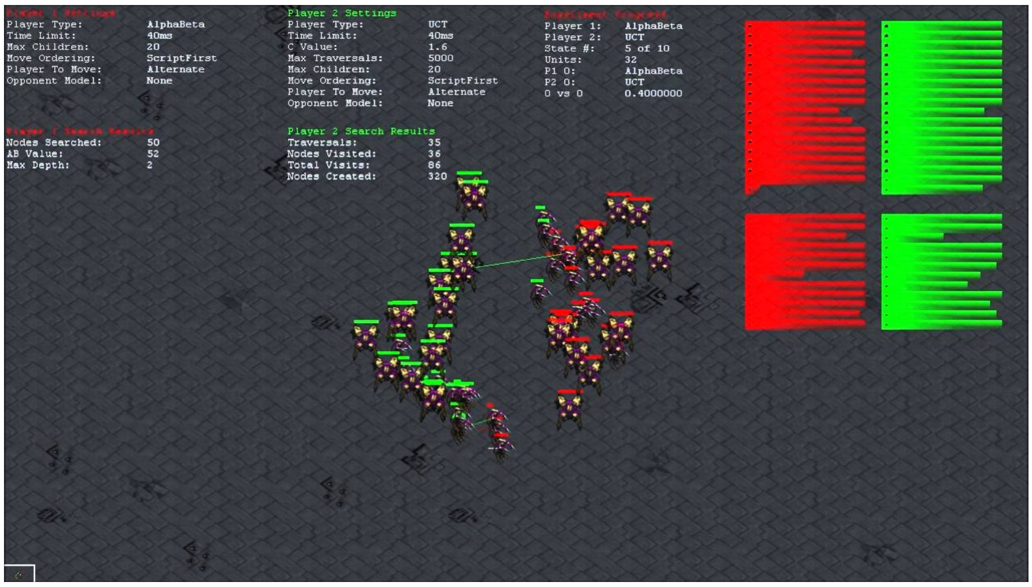
UAlbertaBot is one of the first agents to ever enter the competition. It was built in a modular way so that it can be easily modified and, as such, it served as a base for most of the competing agents nowadays.

3.6.1 Composition

UAlbertaBot is comprised of three major modules described below.

The *UAlbertaBot*, which is the main module (has the same name as the agent), responsible for the strategic management, from which units should be built to how the worker are assigned to the different existing resources.

The *Sparcraft*, responsible for combat aspect of the game, simulates each combat before deciding which action to take based on the simulation's outcome. In case of victory, the agent will engage the enemy army, as opposed to defeat, where the agent will then retreat. Figure 3.8 shows an example of this simulations.



```
Player 1 Settings                      Player 2 Settings                    Experiment Progress
Player Type:        AlphaBeta          Player Type:         UCT             Player 1:   AlphaBeta
Time Limit:         40ms               Time Limit:          40ms            Player 2:   UCT
Max Children:       20                 C Value:             1.6             State #:    5 of 10
Move Ordering:      ScriptFirst        Max Traversals:      5000            Units:      32
Player To Move:     Alternate          Max Children:        20              P1 O:       AlphaBeta
Opponent Model:     None               Move Ordering:       ScriptFirst     P2 O:       UCT
                                       Player To Move:      Alternate       0 vs 0      0.4000000
                                       Opponent Model:      None

Player 1 Search Results                Player 2 Search Results
Nodes Searched:     50                 Traversals:          35
AB Value:           52                 Nodes Visited:       36
Max Depth:          2                  Total Visits:        86
                                       Nodes Created:       320
```

Figure 3.8: *Sparcraft's* instance running.

The BOSS module is responsible for, given a build order, determine the best way to produce units and buildings. Since the proposed algorithm only determines which units to build, the BOSS module was one of the main reasons *UAlbertaBot* was chosen as the base agent to test the algorithm with.

3.6.2 How it was Integrated

The proposed algorithm was added as another module in *UAlbertaBot*, overriding every call the main module would do to its strategy manager component. After running, the algorithm feeds its result to the BOSS module, which will then define the order in which the units and buildings should be produced. Figure 3.9 shows the differences between the original agent and the agent with the new module and how the modules interact with each other.

3.6.3 Integration Problems

The first verified problem in the integration with *UAlbertaBot* was the fact that the agent will constantly make calls to its strategic module, verifying which strategy to take. Since originally *UAlbertaBot* would only change strategies in specific situations this has no negative repercussion in the agent's performance, but since this calls were overridden by the algorithm here proposed the agent became too slow to run effectively.

To solve this issue a function was created that verified if the enemy army composition had changed or grown bigger, as a smaller army with the same composition can only mean that the agent is at an advantage over its opponent.

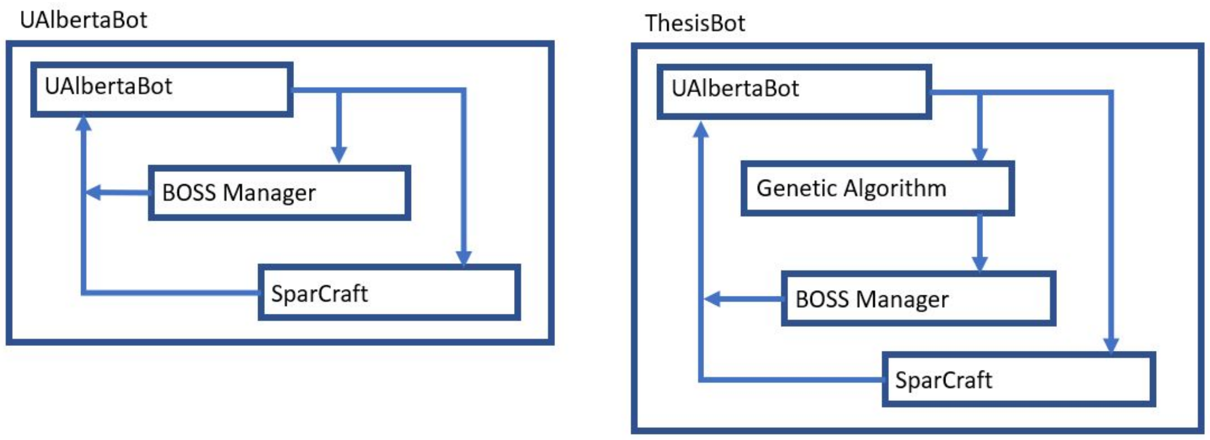


Figure 3.9: *UAlbertaBot* with and without the genetic algorithm module

The second issue verified was with the BOSS module of the agent. This module will only determine the best order in which to build a set of units and buildings provided to it, in a way that fits the agent's original intention (early aggression). This meant that every time the algorithm returned a strategy, the BOSS manager would prioritize producing units it was not yet capable of building, neglecting units that could provide defenses to stall for this more expensive units. This happens because the original agent would only change strategies if strictly necessary.

Since the proposed algorithm only returns which units to build, another algorithm was created. This algorithm verified and added to the build order the necessary buildings to produce the units intended.

Another issue verified was the fact that *UAlbertaBot* focuses too much on offensive, which works well with the original agent as it mainly rushes a few strategies, changing strategies only when very specific conditions are met, or the game reaches a stalemate state.

This may not always be the best course of action, as defending with few units facilitates stalling for more complex strategies and is usually the best way to fend off early aggressions.

4

Results and Discussion

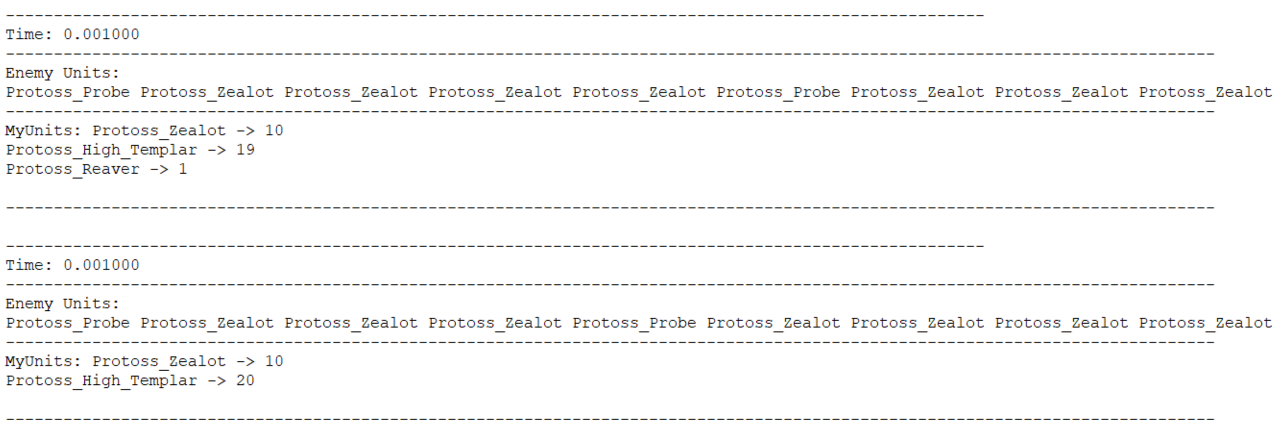
Contents

In this section, we analyze the results of applying the proposed algorithm to *UAlbertaBot*. The results here discussed were obtained through several matches against different agents using different races and opening build orders. Three different kinds of information were recorded in every single match while testing the algorithm. The first was how long it took for the algorithm to run every time it was called (less than one second average), the second was how many times the algorithm was called and if it corresponded to a change in the enemy army, finally the third aspect was the enemy units seen and the algorithm's output, in order to analyze how good was this output versus the enemy units.

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.

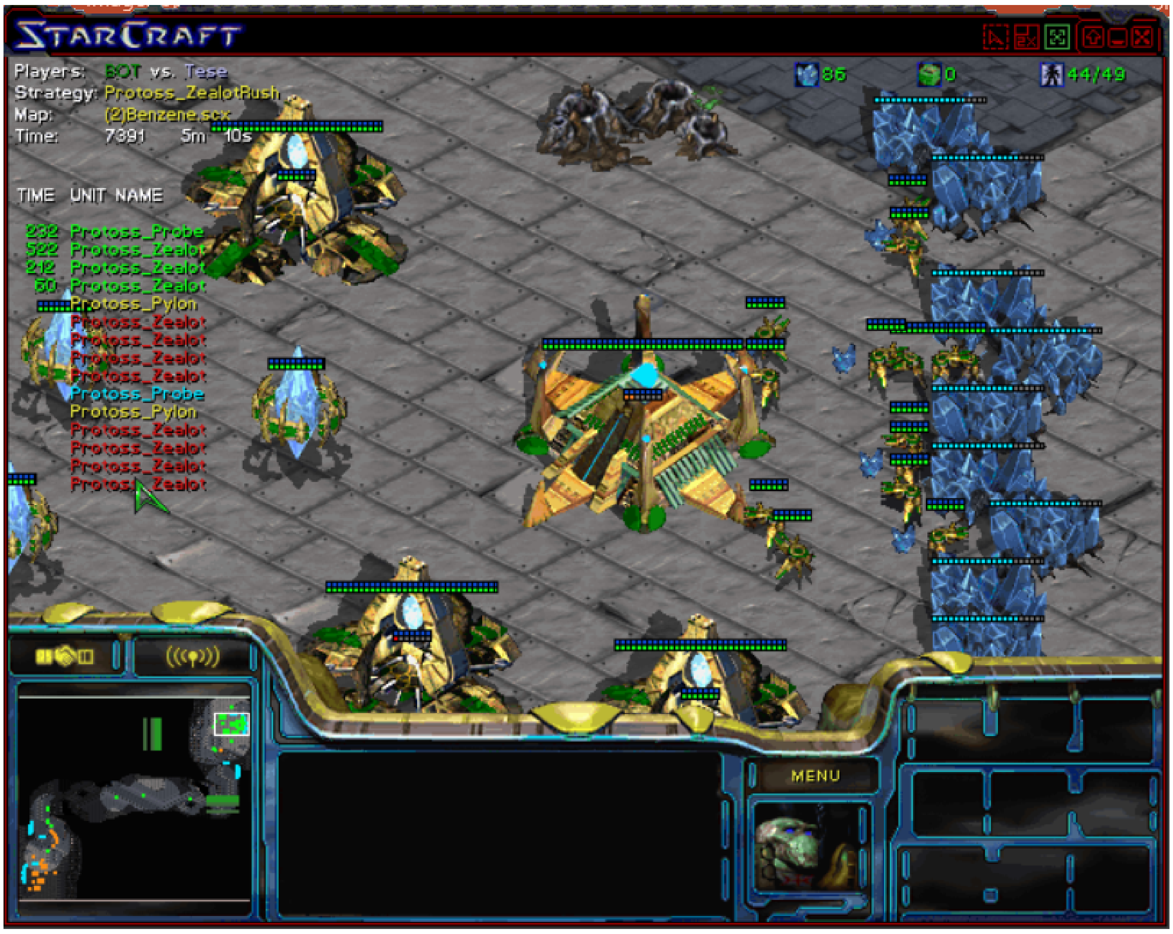
The most challenging aspect to test, was the algorithm's output. Military victory does not necessarily mean that the army had an advantage over the opponent. Depending on the agent's efficiency in micromanaging units, or how fast it can produce them, a battle outcome can be unforeseen. Because of this, 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.



```
------------------------------------------------------------------------------------------------------
Time: 0.001000
------------------------------------------------------------------------------------------------------------------------------
Enemy Units:
Protoss_Probe Protoss_Zealot Protoss_Zealot Protoss_Zealot Protoss_Zealot Protoss_Probe Protoss_Zealot Protoss_Zealot Protoss_Zealot
------------------------------------------------------------------------------------------------------------------------------
MyUnits: Protoss_Zealot -> 10
Protoss_High_Templar -> 19
Protoss_Reaver -> 1

------------------------------------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------------
Time: 0.001000
------------------------------------------------------------------------------------------------------------------------------
Enemy Units:
Protoss_Probe Protoss_Zealot Protoss_Zealot Protoss_Zealot Protoss_Probe Protoss_Zealot Protoss_Zealot Protoss_Zealot Protoss_Zealot
------------------------------------------------------------------------------------------------------------------------------
MyUnits: Protoss_Zealot -> 10
Protoss_High_Templar -> 20

------------------------------------------------------------------------------------------------------------------------------
```

Figure 4.1: Sample of a test output file, showing the enemy units seen, units intended to build and execution time.

Figure 4.1 shows a sample of the algorithm's output in a match. This information can also be verified (partially) in figures 4.2 and 4.3, that represent two different perspectives between a match featuring *UAlbertaBot* with (figure 4.3) and without (figure 4.2) the proposed algorithm.



```
Players:   BOT vs. Tese
Strategy:  Protoss_ZealotRush
Map:       (2)Benzene.scx
Time:      7391     5m  10s

TIME  UNIT NAME

 232  Protoss_Probe
 522  Protoss_Zealot
 212  Protoss_Zealot
  60  Protoss_Zealot
      Protoss_Pylon
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Probe
      Protoss_Pylon
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Zealot
```

Figure 4.2: *UAlbertaBot* running without the proposed algorithm

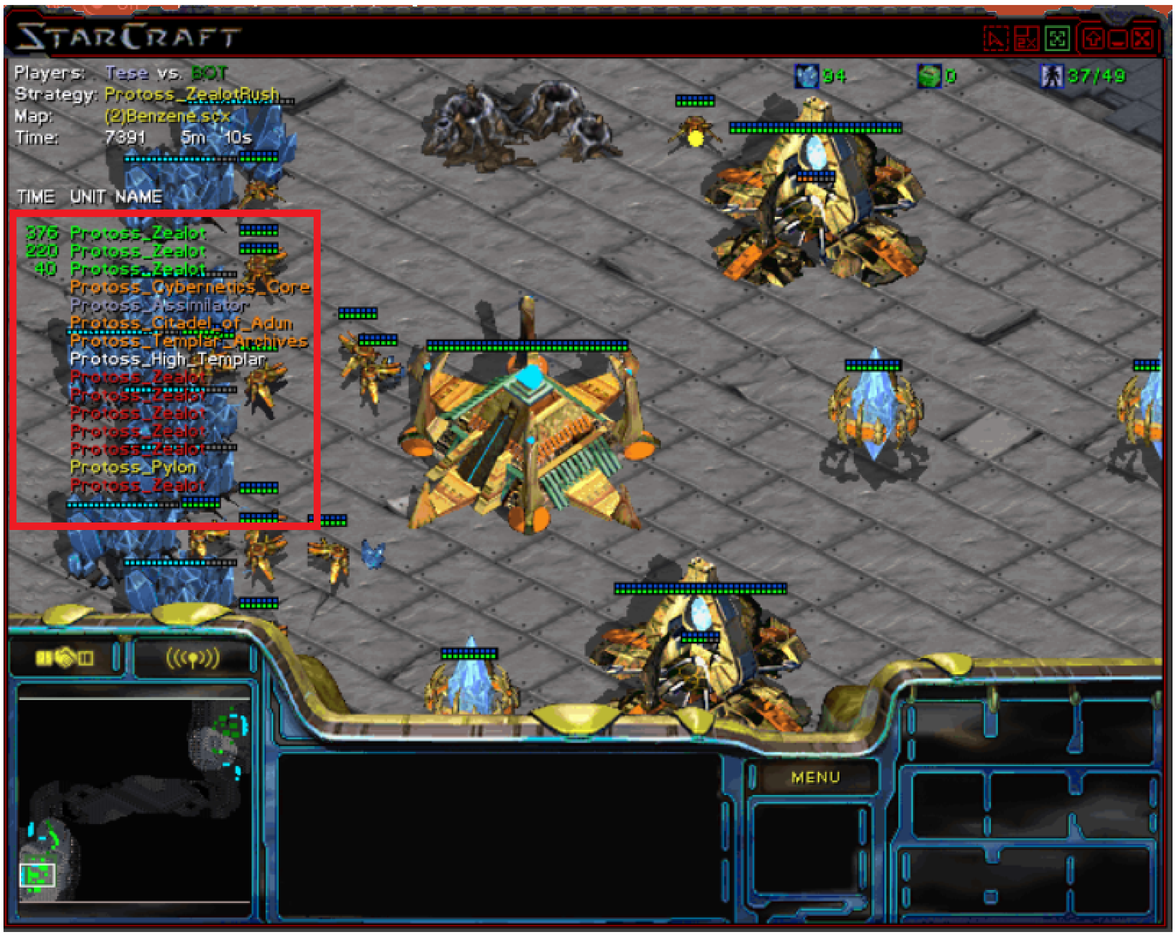
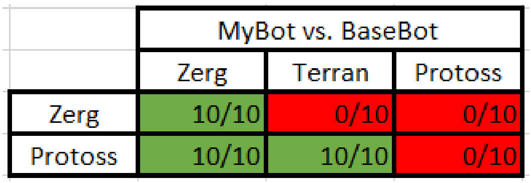

```
Players:  Tese vs. BOT
Strategy: Protoss_ZealotRush
Map:      (2)Benzene.scx
Time:     7391     5m  10s

TIME  UNIT NAME
 376  Protoss_Zealot
 220  Protoss_Zealot
  40  Protoss_Zealot
      Protoss_Cybernetics_Core
      Protoss_Assimilator
      Protoss_Citadel_of_Adun
      Protoss_Templar_Archives
      Protoss_High_Templar
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Zealot
      Protoss_Pylon
      Protoss_Zealot
```

Figure 4.3: *UAlbertaBot* running with the proposed algorithm, with the difference in build order highlighted.

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

4.2 Discussion

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



	MyBot vs. BaseBot		
	Zerg	Terran	Protoss
Zerg	10/10	0/10	0/10
Protoss	10/10	10/10	0/10

Figure 4.4: Match results against different races upon testing.

The *Zerg* race has a unique aspect, all their buildings are living beings and workers mutate into their buildings. This means that the workers have to be replenished constantly, something that was not verified in the agent's base behaviour, meaning that the agent was constantly behind when playing against the other races. *UAlbertaBot* managed their workers better when playing other races, as they work in a more generic way, since their workers are not spent as each building is created.

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

Another aspect that hindered the agent's performance, 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, making it difficult to stop the opponents military pressure due to the difference in size of the armies.

With all this behavioural divergences between *UAlbertaBot* and how the proposed algorithm intended it to react, a conclusion was reached. The algorithm alone is not sufficient to upgrade *UAlbertaBot* into a better version of itself, a better resource gathering management necessary as well as a upgrade to the BOSS module, making it capable of creating defensive buildings and a better build order.

5 Conclusion

Contents

Assuming that a strategy in *StarCraft* is defined by the choice of units and buildings to build, and that the buildings to build can be derived from the units chosen to produce, this thesis presents an algorithm capable of generating strategies on run-time. This means that the agent will be able to adapt to every single strategy the opponent may try, always seeking economic and military advantage, without the need of coding specific strategies in order to react.

With this in mind, the algorithm here presented tackles the problem of creating an adaptive model for strategy generation in *Starcraft* that is capable of countering an opponent's strategy.

The algorithm is based on genetic methodologies and empiric knowledge of the game, generating economically viable strategies in order to obtain military advantage over its opponents.

5.1 Conclusions

Although proven that a neural network algorithm is capable of defeating professional human players, this type of algorithm requires huge amounts of data and has a high implementation complexity, something that is not as imperative in genetic algorithms.

Not only genetic algorithms are less complex to implement, but in order to use the proposed algorithm in different games, only the counter multi-map and cost hash-map structures need to be updated, as they are the only part of the algorithm dependent on which game they are being used on.

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.

5.2 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.

Bibliography

[1] C. F. d. Oliveira and C. A. G. Madeira, “Creating Efficient Walls Using Potential Field in Real-Time Strategy Games,” 2015.

[2] I.-S. Oh and K.-J. Kim, “Testing Reliability of Replay-based Imitation for StarCraft,” 2015.

[3] H. Huang, “Skynet meets the Swarm: how the Berkeley Overmind won the 2010 StarCraft AI competition,” 2011.

[4] S. Ontanon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, “A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, no. 4, pp. 293–311, 2013.

[5] Google’s DeepMind, “AlphaStar.” [Online]. Available: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/

[6] Jay Scott, “Steamhammer.” [Online]. Available: http://satirist.org/ai/starcraft/steamhammer/

[7] S. J. Russell and P. Norvig, “Artificial Intelligence: A modern approach,” 2009.

[8] S. Samothrakis, S. A. Roberts, D. Perez, and S. M. Lucas, “Rolling Horizon methods for Games with Continuous States and Actions,” 2014.

[9] G. M. J.-B. Chaslot, *Monte-Carlo tree search*, 2010, vol. 174, no. 11.

[10] S. M. Lucas, S. Samothrakis, and D. Perez, “Fast Evolutionary Adaptation for Monte Carlo Tree Search.”

[11] D. Churchill and M. Buro, “Build Order Optimization in StarCraft,” pp. 14–19, 2007.

[12] J. G. Nielsen, “Strategy Prediction in StarCraft : Brood War using Multilayer Perceptrons,” *Strategy*, 2011.

[13] I. Zelinka and L. Sikora, “StarCraft: Brood War - Strategy Powered by the SOMA Swarm Algorithm,” 2015.

[14] P. e. a. Garcıa-Sanchez, “Towards Automatic StarCraft Strategy Generation Using Genetic Programming,” 2015.

[15] D. Churchill, “UAlbertaBot,” 2011. [Online]. Available: https://github.com/davechurchill/ualbertabot

[16] “TeamLiquid.” [Online]. Available: https://strategywiki.org/wiki/StarCraft/Counters

[17] M. G. Bellemare, G. Ostrovski, A. Guez, and P. S. Thomas, “Increasing the Action Gap : New Operators for Reinforcement Learning,” 2012.

[18] H. V. Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning.”

[19] D. Perez and S. M. Lucas, “Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games,” 2013.

[20] U. Jaidee and H. Muñoz-avila, “CLASS Q-L : A Q-Learning Algorithm for Adversarial Real-Time Strategy Games Q-Learning,” pp. 8–13, 2012.

[21] D. Harabor and A. Grastien, “The JPS Pathfinding System,” pp. 207–208, 2012.

[22] J. Hostetler, E. Dereszynski, T. Dietterich, and A. Fern, “Inferring Strategies from Limited Reconnaissance in Real-time Strategy Games.”

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

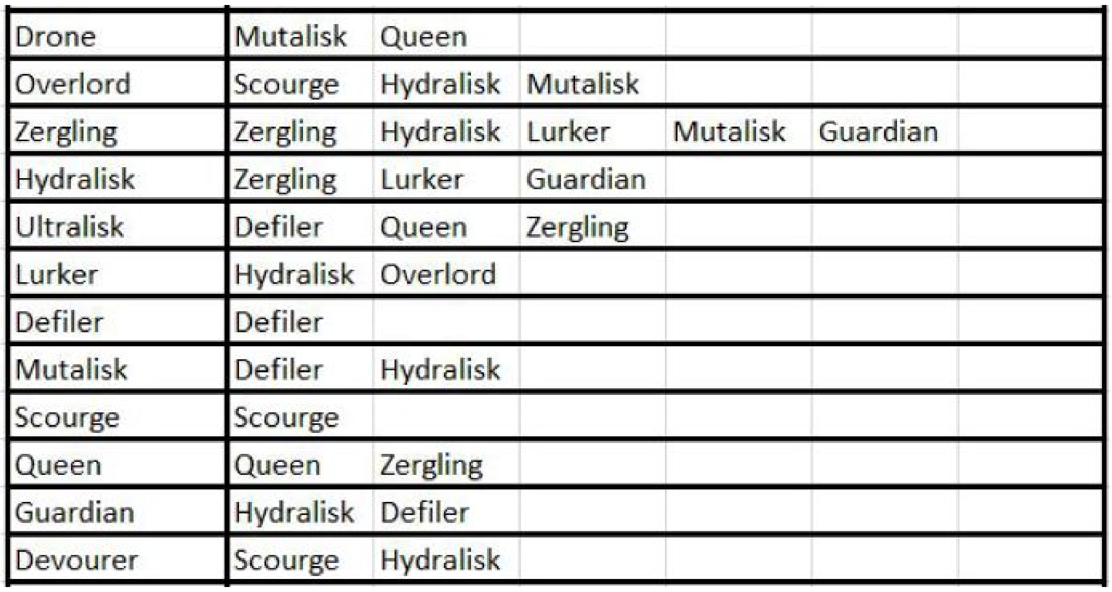
[24] I. Cig, *2015 IEEE Conference on Computational Intelligence and Games*, 2015.

[25] K. Nguyen, “Potential flows for controlling scout units in StarCraft,” *The Proceedings of the IEEE Conference on Computational Intelligence in Games*, vol. 2012, no. Sscai 2012, pp. 344 –350, 2013.

[26] C. M. C. C. M. Bishop, “Pattern recognition and machine learning,” *Pattern Recognition*, vol. 4, no. 4, p. 738, 2006.

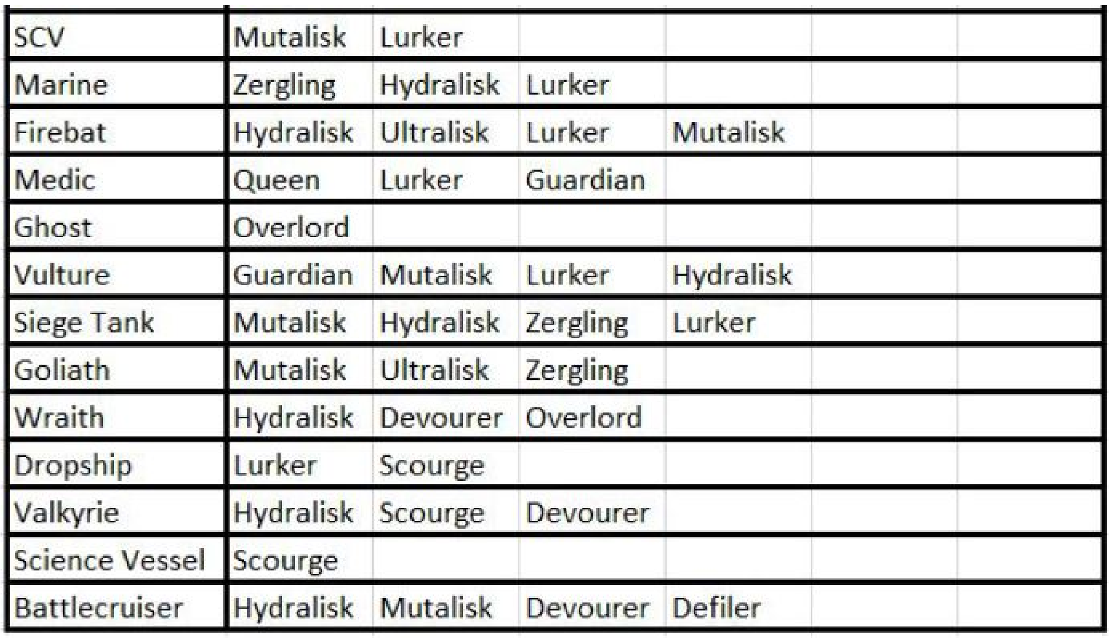
6 Counter Tables

The *Zerg* counter tables are shown below. In the same fashion this tables were created for the *Protoss* and *Terran* races.



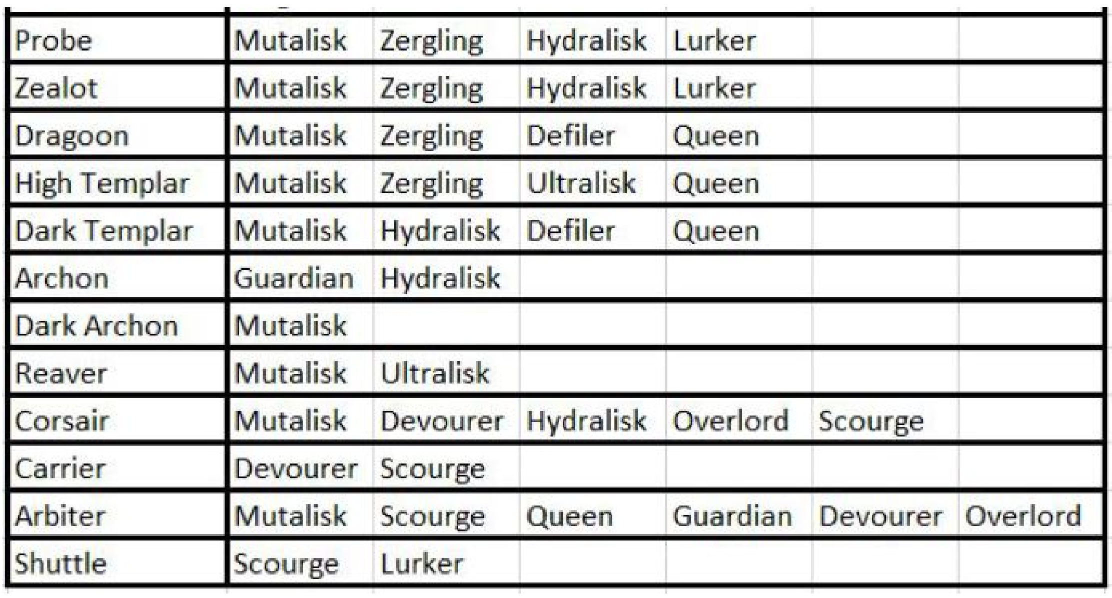
Drone	Mutalisk	Queen				
Overlord	Scourge	Hydralisk	Mutalisk			
Zergling	Zergling	Hydralisk	Lurker	Mutalisk	Guardian	
Hydralisk	Zergling	Lurker	Guardian			
Ultralisk	Defiler	Queen	Zergling			
Lurker	Hydralisk	Overlord				
Defiler	Defiler					
Mutalisk	Defiler	Hydralisk				
Scourge	Scourge					
Queen	Queen	Zergling				
Guardian	Hydralisk	Defiler				
Devourer	Scourge	Hydralisk				

Figure 6.1: Example of a *Zerg vs Zerg* counter table. The units the enemy can produce on the left, my units that are good against them on the right.



SCV	Mutalisk	Lurker				
Marine	Zergling	Hydralisk	Lurker			
Firebat	Hydralisk	Ultralisk	Lurker	Mutalisk		
Medic	Queen	Lurker	Guardian			
Ghost	Overlord					
Vulture	Guardian	Mutalisk	Lurker	Hydralisk		
Siege Tank	Mutalisk	Hydralisk	Zergling	Lurker		
Goliath	Mutalisk	Ultralisk	Zergling			
Wraith	Hydralisk	Devourer	Overlord			
Dropship	Lurker	Scourge				
Valkyrie	Hydralisk	Scourge	Devourer			
Science Vessel	Scourge					
Battlecruiser	Hydralisk	Mutalisk	Devourer	Defiler		

Figure 6.2: Example of a *Zerg vs Terran* counter table. The units the enemy can produce on the left, my units that are good against them on the right.



Probe	Mutalisk	Zergling	Hydralisk	Lurker		
Zealot	Mutalisk	Zergling	Hydralisk	Lurker		
Dragoon	Mutalisk	Zergling	Defiler	Queen		
High Templar	Mutalisk	Zergling	Ultralisk	Queen		
Dark Templar	Mutalisk	Hydralisk	Defiler	Queen		
Archon	Guardian	Hydralisk				
Dark Archon	Mutalisk					
Reaver	Mutalisk	Ultralisk				
Corsair	Mutalisk	Devourer	Hydralisk	Overlord	Scourge	
Carrier	Devourer	Scourge				
Arbiter	Mutalisk	Scourge	Queen	Guardian	Devourer	Overlord
Shuttle	Scourge	Lurker				

Figure 6.3: Example of a *Zerg vs Protoss* counter table. The units the enemy can produce on the left, my units that are good against them on the right.