Student report from
Aalborg University

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø
Telephone (+45) 9940 9940
http://www.cs.aau.dk

Title:
Evolving Strategies for a Real-Time Strategy Games Using Genetic Algorithms

Project Theme:
Machine Learning

Project period:
Spring Semester 2013,
February 1st to
June 7 th

Project group:
SW1013F12

Authors:

Steffan Bo Pallesen

Nikolaj Dam Larsen

Mikkel Graarup Jensen

Supervisor:
Nicolaj Søndberg-Jeppesen

Pages:
88

Abstract:

The demand for challenging AI in commercial Real-Time Strategy games is increasing by each generation. This project explores how genetic algorithms might be used to evolve more challenging AIs, by utilizing the power of natural selection. The popular strategy game Starcraft is used a testing platform throughout the project, due to its maturity as a research platform. Two approaches to evolving a Starcraft AI is examined. One is to use a traditional genetic algorithm, and the other is to use a Estimation of Distribution Genetic Algorithm. However, only the traditional genetic algorithm is implemented and tested. The results show that it is in fact possible to evolve an AI that can successfully beat an opponent playing the same static strategy. More research is needed in order to determine if genetic algorithms can be used to evolve complete commercial grade AIs.

PREFACE

This report is a master thesis made for the Department of Computer Science at Aalborg University in spring 2013.

The topic of this project is Adaptive Artificial Intelligence (AI), specifically the development of an AI, using a genetic algorithm evolving strategies for the Real-Time Strategy (RTS) game Starcraft.

This project is divided into five parts:

- **Introduction**
  Introduces the project statement and the focus of this thesis, i.e. our hypothesis and how it is validated. Then moves on to describe the Starcraft domain and at last investigates related work - including our own pre-specialization project.
- **Evolutionary Algorithms**
  Describes the theory of genetic algorithms, which forms the basis for the later chapter about developing our AI using genetic algorithm. This part also presents the topic of Estimation of Distribution Algorithms (EDAs) which is a possibly better alternative to a standard genetic algorithm.
- **Evolving Starcraft Strategies**
  Presents our implementation of a genetic algorithm, such as how to encode a chromosome representing a Starcraft strategy. This part also introduces the development of our AI bot called Cromartie, which is able to play Starcraft and both use static strategies and strategies represented by chromosomes.
- **Results and Discussion**
  Describes the results gathered from testing our hypothesis using the genetic algorithm version of Cromartie and discusses the perspective of our results and how they may be used in practice.
- **Conclusion and Reflection**
  Concludes the work done it this thesis and reflects on what topics might be interesting for future work.

After these parts come the appendix and the bibliography.

CONTENTS

Part I

Introduction

CHAPTER 1

INTRODUCTION

Today, strategy games are an integrated part of human culture. In the western world, Chess has for hundreds of years been one of the most popular strategy game. Likewise, the Chinese game Go plays a similar role in many Asian cultures. The two games share properties such as being turn-based and being played by two players. The main differences between the games are the strategic goals. In Chess, a player must defeat his opponent through attrition[1], until he is no longer capable of protecting his king. In order to win in Go, a player must conquer more territory than his opponent.

Attrition from Chess, and control of territory from Go is the basis for many modern strategy games. These two strategic elements are often combined, requiring even more complex strategic thinking from the players. A new strategy concept found only in computer strategy games, is the idea of real-time gameplay. Instead of having each player wait until it becomes his turn, like in Chess or Go, Real-Time Strategy (RTS) games allow both players to make decisions and perform actions simultaneously. Additionally, the game-time keeps progressing, forcing the players to make quick decisions.

Starcraft is one of the most popular RTS games ever made. It includes all three strategic elements from Chess and Go, and being a RTS, is played in real-time. In fall 2012 our group wrote a pre-specialization project[1], about performing data mining on Starcraft replays in order to find common strategies in the set of mined replays. This project provided the insight into using Starcraft as a research platform. For the duration of this thesis, as with the pre-specialization project, Starcraft Broodwar is used as our domain/testing platform. The Starcraft modding community have developed open-source tools for injecting code into the game at runtime and an API for developing AI agents/bots for the game. More on the Starcraft domain is found in Chapter 2.

The focus of this project is on adaptive AI, which is Artificial Intelligence that is able to adapt to its current environment or to the player(s). Adaptive AI is essentially applying machine learning techniques to improve an AI, for example by letting the AI learn from mistakes (reinforcement learning) or adapt to new previously unknown tactics. Much of the academic literature about this topic revolves around Pieter Spronck who also wrote the book *Adaptive*

[1] Attrition means gradually reducing the enemy forces

Game AI[2]. This book, as well as the article *Automatically Acquiring Domain Knowledge*[3], inspired us to define the problem statement presented in Section 1.1.

The report starts out by introducing Starcraft and related work followed by a detailed description of the theory, development, tests and results from implementing an adaptive AI using Genetic Algorithms (GAs). This adaptive AI is used to validate the correctness of Hypothesis 1 set up in the problem statement. First the theoretical background on GAs and similar techniques are established, as this is the base of our own implementation of a Genetic Algorithm (GA). The standard GA is shown to be difficult to configure, as the configuration influences the correctness and convergence rate of the solution. EDAs are introduced as a possibly better alternative to the standard GA, although trying this technique is left for future work.

To validate the hypothesis described in Section 1.1 the test setup required us to implement two AI agents/bots able to play Starcraft. One of the bots is an AI enhanced with our GA and the other is simply able to play a static strategy given as input at the beginning of a game.

When testing our genetic algorithm AI, we tested using four different configurations, different on the population size and the configuration of the selection mechanism (see Chapter 4). Eventually our results showed that at least two different configurations of our GA was able to evolve 4 winning chromosomes against an AI with a static strategy. It was also found that a population size of 100 appears to outperform a population size of 50.

1.1 Problem Statement

The goal of this master thesis project is to investigate how adaptive AI can be used to enhance gaming experience, specifically in RTS games. One of the challenges of Artificial Intelligence is to always be able to beat the player or play at the same level as the player, without using obvious cheats.

In this project this challenge is addressed, as we attempt to find answers to a hypothesis regarding adaptive AI. Four hypotheses were initially considered related to how adaptive AI can be applied to Starcraft. The first of these hypotheses were selected for further investigation, while the rest was saved for future work (see Chapter 11). Our goal is to setup tests and an environment, and use it to validate the accuracy of this first hypothesis:

Hypothesis 1. *It is possible to use a Genetic Algorithm (GA) to evolve a Starcraft strategy that can* consistently *win against a single other strategy.*

As a Genetic Algorithm will evolve a population of strategies (described in Chapter 4), the term *winning consistently* is defined as at least one chromosome in the population must win 50% of the time over 100 games.

1.1.1 Setting up the test

In order to test the hypothesis two Starcraft AIs are developed; one AI that incorporates the Genetic Algorithm and one AI to play a single strategy given as a parameter to it before the game begins. The Genetic Algorithm AI will use the strategy defined by a given chromosome, while the AI to play a single strategy will take as input some external strategy. gathered from an expert source[4]. The internal workings on the AI agents will primarily differ on the components related to managing strategy, while tasks such as gathering resources and micro-management of units is handled by exactly the same components for each AI. The implementation details of each of these AI agents is described in Chapter 7.

The hypothesis is tested by repeatedly making the Genetic Algorithm AI play against a single AI, using the exact same strategy, on the same map, in each game instance. The Genetic Algorithm AI will use a different chromosome for each game instance, and thus a different strategy. The state of the Genetic Algorithm AI will be checked regularly, in order to monitor whether the win-rate or the average fitness of the population increases - and of course to validate whether a chromosome with at least 50% win-rate have been evolved.

There is no limit on the number of repeated games played when testing this hypothesis. This is because the GA does not provide a guarantee on the time taken to converge towards a winning strategy. This means that it is hard to tell whether the test is a failure. Instead the information repeatedly reported regarding the state of the GA, is used to monitor whether the GA converges. If the GA does not converge, the hypothesis could be deemed incorrect, or alternatively the configuration of the GA might have to be adjusted and tested once again. The subproblems which is described next, considers this.

1.1.2 Subproblems of the hypothesis

The Hypothesis 1 established for our problem statement raises additional questions involving the configuration of the tests and the testing environment. These questions are described here as subproblems of our hypothesis, and the results of testing their influence is summarized in the next section and described in Chapter 8.

First subproblem

In Section 4.3 it is stated that differently sized populations, impacts how near-optimal a solution can be evolved, as well as how fast it converges. We want to test a couple of different population sizes and see their influence on how our GA converges.

This subproblem is tested in the same manner as the overall hypothesis, but using different configurations of the GA to compare the results.

CHAPTER 2

DOMAIN: STARCRAFT

This chapter describes the RTS game Starcraft, which is the problem domain for this project. A more detailed description of Starcraft can be found in our pre-specialization report from fall 2012.

2.1 Mechanics of Starcraft

Starcraft is an RTS game in which 2-8 players play against each other. The game is won by the player that destroys all his enemies forces or if all his enemies give up by leaving the game. Players can play as three different races: Protoss, Zerg and Terran. The chosen race will dictate which buildings, combat units and technology upgrades are available to the player, as each race is unique. In the beginning of a Starcraft game, only a few buildings and combat units are available to the player. More units and buildings can be unlocked by constructing technology buildings. Figure 2.1 illustrates which buildings unlocks which, for the protoss race [1].

Because a Protoss player always starts with the Nexus building, the player only has five different buildings available for construction at the start of the game (Pylon, Nexus, Assimilator, Gateway and Forge). If the player then decides to build the Gateway, then once the construction is finished, the Cybernetics Core is available for construction.

Another important mechanic of Starcraft is the concept of supply cap (short for capacity). The supply cap limits the number of units a player can train, as each unit consumes, or reserves, an amount of supply, e.g. the Protoss Zealot consumes 2 supplies. Once the unit is destroyed the consumed supply is released and may be reused. At the beginning of a game the supply cap is only 8, but can be increased by constructing the pylon building. Units usually require 1 or 2 supplies, and the maximum value for the supply cap is 200.

There are two resources in the game that players must collect in order to train units, construct buildings and research upgrades. Minerals and gas are scattered all over the map for players to find and mine. Worker units must be trained to collect these resources, and assimilators must be constructed on top of gas resources in order to enable extraction.

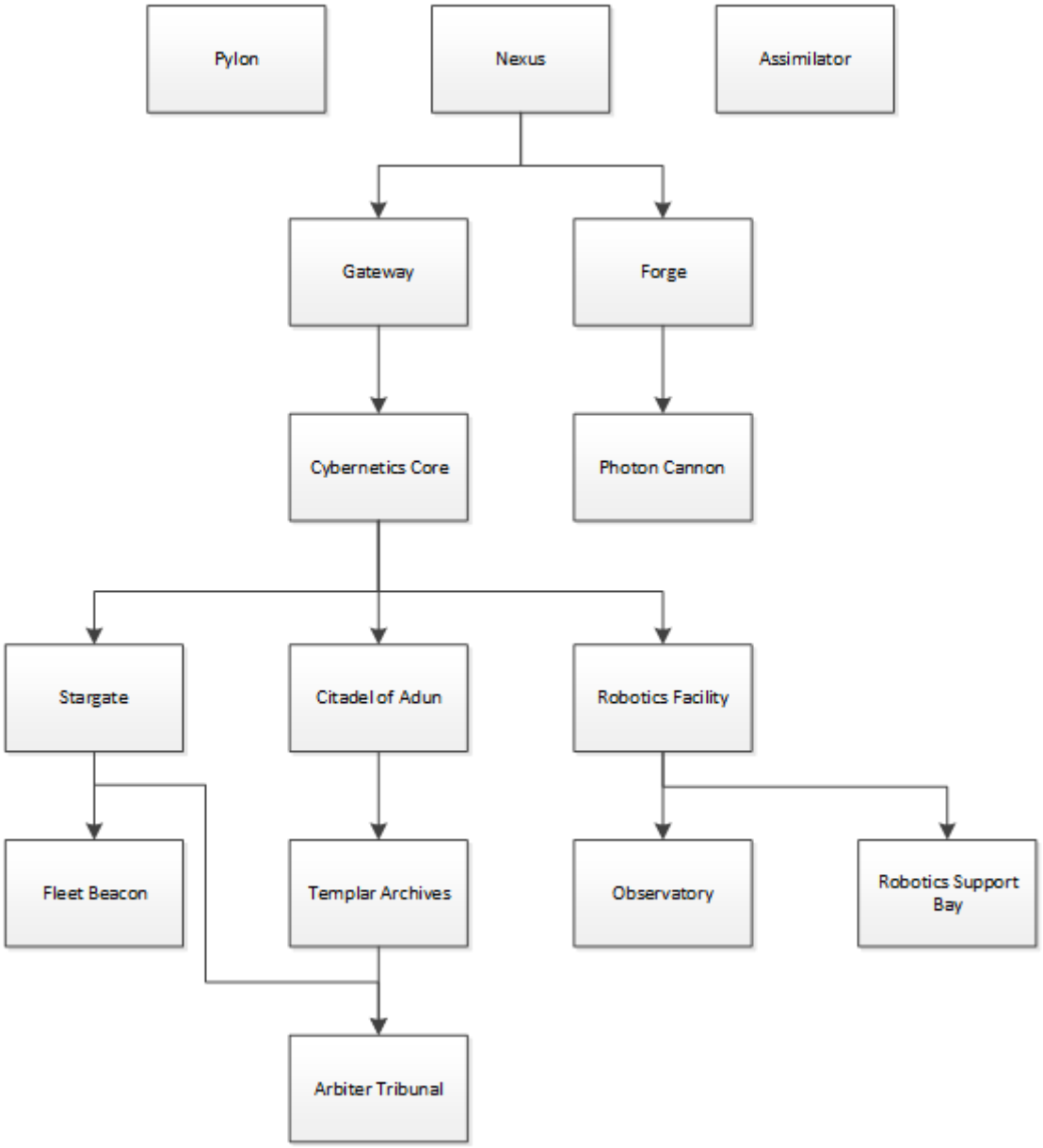

Figure 2.1: This figure illustrates the technological dependencies for the race Protoss. Each square represents a building and an arrow represents a technological dependency. For instance, the Cybernetics Core building requires that the Stargate building has been constructed.

When a Starcraft game is started, a map has to be chosen. A map is a 2-dimensional grid of plains, hills, rivers and other objects, with resources scattered around for the players to exploit. The various features of the terrain can be used when micro-managing combat units. During a Starcraft game, only a fraction of the map will be visible to the player. This is

because of the game concept called Fog of War. Fog of War hides areas of the map that cannot be seen by the players units. Each unit has a visibility radius that allows it to see parts of the map that is close to it. Because intelligence is highly important in strategy games, scouting tactics are usually employed to lift important parts of the Fog of War.

Buildings can be divided into three categories: Production buildings, research buildings and defense buildings. Production buildings can be used to train combat units, while research buildings can be used to research technology upgrades. Defense buildings serve as static defense structures that can damage enemy units. [1]

2.2 Our use of Starcraft

We use Starcraft as the test-bed for our hypothesis as it is highly complex and a real commercially successful game, in contrast to "'research"' games such as Wargus[5], as we wanted the domain to reflect the use of adaptive AI in practice. Our pre-specialization project, was focused on one of the races of Starcraft - the Protoss - the same is done in this thesis. This decision is based on our existing knowledge of the race and common tactics, as well as the basis for our AI agent/bot, Skynet which is described later in Section 7.2, being primarily a Protoss bot.

2.3 Development Platform

In the introduction in Chapter 1, it is stated that Starcraft is a common domain for use as a research platform. This is in part because of a large community of modders, that have created open-source tools and APIs that enable anyone to inject code - such as an AI - into Starcraft. The most used API is called BWAPI[6] (BroodWar Application Programming Interface), which is used to development AI agents/bots that can be injected by a tool called Chaosloader (bundled with BWAPI) into an instance of a Starcraft game. The BWAPI enables any program or AI using it, to take any action in the game available to the human player, e.g. moving units, constructing buildings, researching, attacking, etc. BWAPI is much too large to cover here, but a full documentation can be found at the BWAPI Manual [7].

CHAPTER 3

RELATED WORK

Starcraft has previously been used as a platform for testing a range of different machine intelligence and machine learning hypotheses. GA have also previously been applied to RTS games. This chapter presents a few topics of related work of interest to this thesis as well as a list of popular Starcraft bots and the techniques they use.

3.1 Pre-specialization project

This project is a continuation of our pre-specialization project created in fall 2012. The pre-specialization project [1] explored the use of unsupervised learning to label Starcraft replays. The goal was to identify Starcraft strategies from these replays and then create a bot that could execute learned strategies. K-means clustering was used for the purpose of learning strategies from these replays, but unfortunately the results were not promising. We were able to identify only two overall strategies, but were unable to identify more finely grained strategies. The reason for this was either that Starcraft was too complex for the clustering algorithm, that the feature selection was less than ideal or that the choice of clustering algorithm was poor for the problem domain. Because of these disappointing results, we decided to not explore strategy identification further in this thesis.

3.2 Automatically Acquiring Domain Knowledge

Automatically Acquiring Domain Knowledge For Adaptive Game AI Using Evolutionary Learning is an article that describes research in the field of adaptive AI. Specifically, the article aims at developing an adaptive AI for the RTS game Wargus[5] by combining Genetic Algorithms (GAs) with dynamic scripting. Dynamic scripting [8] is a simple but effective way of creating an adaptive AI. The most important part of dynamic scripting is the rule base. The rule base is a collection of condition based rules each with probability weight assigned to it. When an AI for a game agent is needed, it can be created by sampling the rule base. A rule with a high probability weight is more likely to be included in the script, while rules with

low probability weights are less likely to be included. When an agent with a dynamically generated AI script has been engaged in combat with the human player, the rule base is updated. Rules that worked well against the enemy will receive a higher probability weight, while rules that did not work well, will receive a lower probability weight. Over time the scripts generated from the rule base will be adapted to work well against the human player, since successful rules will be more likely to be included in the dynamically generated script [2]. The main contribution of the paper is the use of a GA to evolve a rule base, which is normally created manually using domain knowledge. This approach managed to evolve rule bases that are capable of defeating opponents that uses a specific strategy. The practical applications of using a GA as a means for evolving a competent rule base, is that AI developers can automate some parts of the process of developing AI. This article has been one of the inspirations to this writing thesis.

3.3 Existing bots

The community revolving around BWAPI (See Section 2.3) have created a large number of bots with different goals and uses. This section will list some of the most well-known bots - a subset of those regularly competing in the annual Starcraft AI Competition [9] and the techniques they use. The purpose of this section is to both present some of the bots inspiring the creation of our GA bot, but also to credit the ingenuity of some of the bots competing.

Skynet is one of the most important AI projects to this thesis as it is used as the basis for common tasks required from our GA bot (more in 7). Skynet is mostly interesting for its complexity, size and the stability. The bot is highly scripted and does not make use of interesting AI or machine learning techniques. [10]

UAlbertaBot is another interesting bot developed by the University of Alberta hosting the competition. The bot uses both dynamic AI systems and scripted rule-based decision making for its different components. It includes a heuristic search based build order planning system, that dynamically tries to find near-optimal build orders as well as a real-time combat simulation system, used to determine whether it is feasible for a group of units to engage in combat. It utilizes machine learning to select strategies for opponents, based on previous matches. [11]

NOVA is a bot playing the Terran race that uses various AI techniques. The NOVA bot is a multi-agent system that assigns the tasks of micro- and macro management to multiple sub-agents. It boasts a long list of interesting techniques, such as: opponent strategy prediction, potential fields and score board for combat target selection. [12]

SCAIL is an ingenious AI that uses multiple modern AI techniques. An article by the developers describe the techniques and systems used by the AI to compete the Starcraft AI

competition. Some of the sophisticated AI presented in their article and used by the bot is: particle filtering, online machine learning, drive based motivational systems and virtual emotions. [13]

Part II

Evolutionary Algorithms

CHAPTER 4

GENETIC ALGORITHMS

A genetic algorithm simulates the process of biological evolution in order to produce solutions to an optimization problem. Genetic algorithms are the most popular technique in the class of algorithms called Evolutionary Algorithms. Other techniques in this class include: Genetic Programming, in which the evolved solution is in the form of a computer program and Evolutionary Programming, which is similar to Genetic Programming, but allows for the numerical parameters (population size etc.) to evolve.

Genetic algorithms have been used to solve multiple optimization problems. For instance, computer-automated design [14], optimizing solutions for Traveling Salesman Problem (TSP) [15], and training neural networks [16]. In each of these cases, the genetic algorithm has been tailored to fit the domain in order to produce good results.

A genetic algorithm works by first randomly creating a set of solutions to the targeted problem. This set is called the population, and has a fixed size that can vary depending on the problem domain. Each solution is called a chromosome or an individual. These initial chromosomes are unlikely to be effective solutions to the problem as they are randomly generated. They will however serve as building blocks for creating better chromosomes. A fitness function is used to evaluate the quality of a chromosome, and a selection mechanism selects which chromosomes are allowed to breed. A set of genetic operators are used to spawn child chromosomes from one or more parents. The newly spawned child chromosomes replace old chromosomes in the population. In many selection mechanisms, chromosomes with a high fitness value are chosen for breeding while chromosomes with a low fitness value are destroyed. A genetic algorithm is terminated when some max fitness threshold is reached, or after an arbitrary number of iterations.

In order to explain how a genetic algorithm may be used to optimize a solution to a problem, an example is given. Imagine that we want to find the shortest route through all the nodes in an undirected graph (also known as the TSP). This is not possible with a genetic algorithm, since there is no way of knowing if the solution produced is the shortest possible. However, it is possible to optimize a solution to the problem that might be close to the shortest route. In order to do this a chromosome encoding for a route must be defined. A chromosome could be a set of state transitions, as seen in Table 4.1.

A -> C
C -> D
D -> B
B -> E

Table 4.1: This is how a chromosome might be formatted when applying a genetic algorithm to the TSP.

The fitness function for evaluating the quality of a route could be a function that calculates the total distance of a route. The shorter the distance, the better the route. With a chromosome encoding and a fitness function we can now simulate natural selection to optimize a solution as shown in Listing 4.1.

```
1  Generate the initial population randomly
2  Calculate the fitness for each chromosome in the population
3  while (!stopcriterionIsMet())
4  {
5      Select a set of promising chromosomes
6      Breed these chromosomes using genetic operators to produce children
7      Evaluate fitness of each child
8      Add the children to the population
9      Remove the chromosomes with the lowest fitness
10 }
```

Listing 4.1: Pseudo code showing how a genetic algorithm works.

The algorithm starts by generating a random population of routes. They are not completely random since they must be valid, meaning that a chromosome must visit all nodes in the graph once, and only once. The fitness value for each route is calculated using the fitness function. The best routes are then taken from the population and used to breed new routes using genetic operators (how chromosomes are chosen for breeding can vary, as there exist multiple selection mechanisms. See Section 4.1). A genetic operator could for instance breed a new route by combining two parent routes. The resulting route would be a route that is the combination of some part of parent one and some part of parent two. A genetic operator could also mutate a parent to produce a child. When a number of children have been produced, an equivalent number of chromosomes are removed from the population. Over time, long routes will be removed and shorter routes will survive increasing the average fitness value of the population, and (hopefully) the maximum fitness value.

4.1 Selection Mechanisms

There are many different mechanisms for selecting individuals for breeding. This section will cover some of them, including the ones used in this project.

4.1.1 Tournament Selection

Tournament selction has three parameters:

- SampleSize
- WinnersSize
- LoosersSize

A random sample of size SampleSize is taken from the population. The best individuals in the sample are selected for breeding while the worst individuals in the sample are destroyed. The amount of individuals chosen for breeding is determined by the WinnersSize parameter while the amount chosen for destruction depends on the LoosersSize parameter [17]. Some individual might not belong to either the winners or the losers. For instance, if sampleSize = 10, winnersSize = 2 and loosersSize = 2, 6 chromosomes will neither belong to the winners or losers. These chromosomes will not be chosen for breeding, but neither will they be chosen for destruction. They will simply remain in the population. The chosen parameters for the tournament selection have a high influence on how it will perform. For instance, if the sample size is set to the same value as the population size, only the absolute elite will be chosen for breeding. This may seem like a good thing, since the whole point of a genetic algorithm is to breed the good chromosomes and discard bad ones. But by doing this, a lot of building blocks are lost, since mediocre chromosomes will not be chosen for breeding. Therefore, the consequence of having a high sample size is that the genetic algorithm might converge faster, but is more likely to get stuck at a local optimum.

4.1.2 Fitness Proportionate Selection

Fitness Proportionate Selection has one parameter:

- SampleSize

Each individual in the population is assigned a probability based on its fitness value. The higher the fitness value, the higher the probability it has of being selected for breeding. The probability of individual i is calculated using Equation 4.1:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{4.1}$$

Where p_i is the probability that individual i is chosen for breeding, f_i is the fitness value for individual i, and N is the population size. To keep the population size constant, the individuals with the lowest fitness in the population is destroyed.

4.1.3 Truncate Selection

Truncate selection is a very basic selection mechanism in which the population is sorted by fitness, and the X best performing chromosomes are selected for breeding.

4.2 Genetic Operators

When a selection mechanism has chosen at set of chromosomes to breed, genetic operators are used to perform the actual breeding. Genetic operators are divided into two classes: Mutation operators and crossover operators. Mutation operators works on only one parent. A child is produced by taking the parent chromosome and the applying mutations to its genes. For instance, a mutation operator might replace existing genes with different types of genes. Alternatively a mutation operator might change the parameters of the genes.
Crossover operators take two parent chromosomes and produces one or two child chromosomes, by combining different parts of the parents. There are four common types of crossover operators.

4.2.1 One Point Crossover

The first is the One Point crossover. The chromosomes are divided as illustrated in Figure 4.1. Two children will be produced, each containing one part of parent one and one part of parent two [18].

4.2.2 Two Point Crossover

The Two Point crossover works similar to the One Point crossover operator. Instead of choosing one separation point, two points are chosen as illustrated in Figure 4.2. Two children are then produced as shown in the figure.

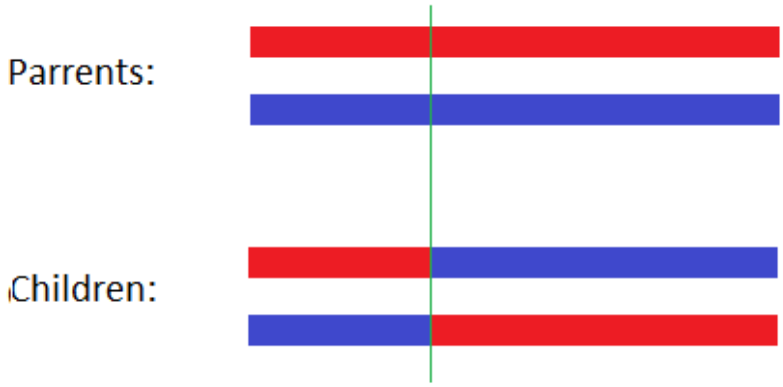

Figure 4.1: This image shows how the One Point crossover operator can be used to split two parents, and produce two children.

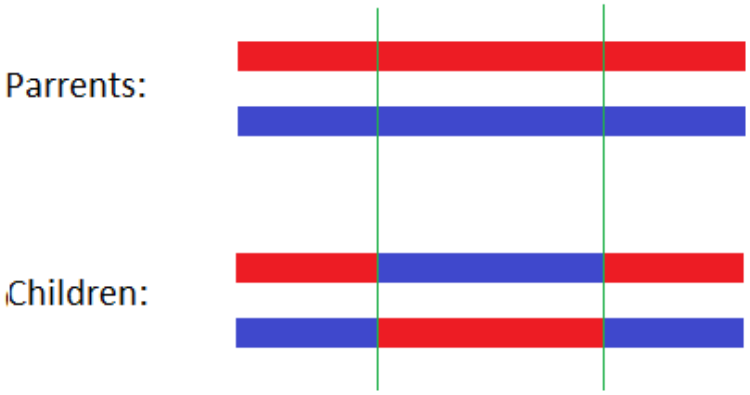

Figure 4.2: This image shows how the Two Point crossover operator can be used to split two parents, and produce two children.

4.2.3 Population-wise Uniform Crossover

The Population-wise Uniform crossover is also similar to the One Point- and Two Point Crossover operators, but it applies a much more powerful mixing as it alternates every gene between the two parents as shown in Figure 4.3. It is given the name "'Population-wise"' to distinguish it from the probabilistic uniform crossover described in Section 5.3.

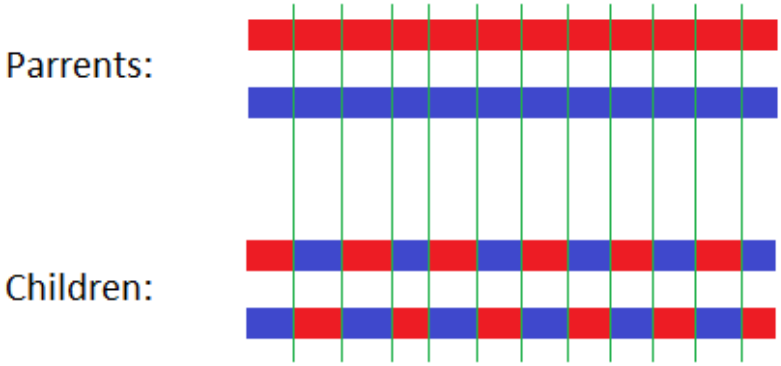

Figure 4.3: This image shows how the Population-wise Uniform crossover operator can be used to split two parents, and produce two children.

4.2.4 Cut and Slice crossover

The Cut and Slice crossover operator randomly chooses two different cut points on each parent. Two children can then be produced by combining the different parts of the parents, as seen in Figure 4.4. It is important to note that since the cutting points are not the same on each parent, the children produced will not necessarily have the same chromosome length [18].

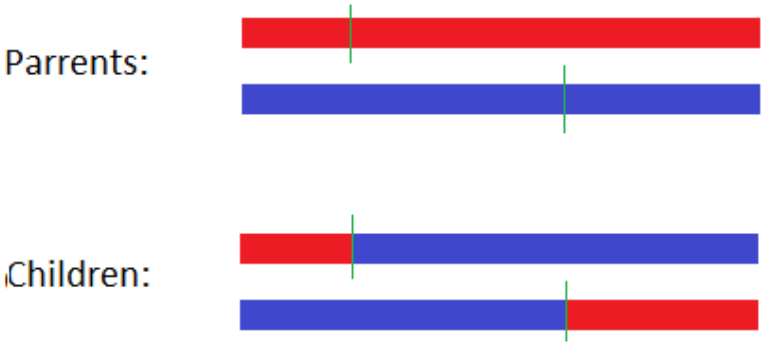

Figure 4.4: This image shows how the Cut and Splice crossover operator can be used to split two parents, and produce two children.

The designers of genetic algorithms must construct their genetic operators in a way so they do not yield invalid chromosomes [18] by discarding invalid children generated, and not

letting them enter the population, or by using a chromosome encoding which can never be invalid (often a difficult challenge).

4.3 Choice of Population Size

When optimizing a solution with a GA, choosing the right population size is important. Unfortunately, little research has been done in this area [19]. Research in this area focuses on finding the optimal population sizes for well-known problems. According to [19] one must balance two tradeoffs when choosing the population size for a lesser known problem: a large population size yields a more accurate result, whereas a small population size yields less accurate results. However, a GA with a large population size require more generations to converge than that of a small population size. One must balance convergence time against accuracy in order to choose the best population size. Unfortunately, the usual approach to determining the best population size is by trying different sizes, and then settling on one that works best [19].

CHAPTER 5

ESTIMATION OF DISTRIBUTION ALGORITHMS

Estimation of Distribution Algorithms (Also known as Probabilistic Model-Building Genetic Algorithms or Iterated Density Estimation Algorithms) are very similar to the GA approach in that they are both loosely based on biological evolution. However, where a GA uses recombination (crossover) and mutation to generate the population of the next generation, Estimation of Distribution Algorithm (EDA)s builds a probabilistic model based on the statistics of the selected set of promising solutions, *discards the population* and generates a new population based on the probabilistic model (Some EDAs advocate only replacing part of the population). In this chapter, the details of an EDA are described.

5.1 Motivation

GAs are excellent for many problems, one being the Onemax problem (See 5.3). Many different combinations of parameters and operators can be used, and it still works. In practice, though, for more complex problems a significant amount of time is spent tuning the GA, choosing the best parameters and operators. Ideally, this work should be incorporated into the algorithm itself, such that it adapts to the specific problem on its own. EDA combines machine learning and Genetic and Evolutionary Computing (GEC) to achieve just that. As an additional benefit, EDAs also provides the practitioner with a series of probabilistic models which may give additional insight into the problem at hand [20].

5.2 General EDA

Just like a general GA, the general EDA starts by generating an initial population randomly. Each chromosome is evaluated, and a set of promising chromosomes are chosen. Based on the promising chromosomes, a probabilistic model is built, and the model is sampled to generate a new population. Listing 5.1 shows in pseudo code the general EDA

```
1 t = 0
2 randomly generate initial population, P(0)
3 while(!StopCriteriaIsMet())
4   select set of promising chromosomes, S(t) from P(t)
5   update the probabilistic model based on the estimated distribution in S(t)
6   create a new population, P(t+1), by sampling the probabilistic model
7   t = t+1
```

Listing 5.1: Pseudo code of a general EDA.

5.3 Onemax Problem

The Onemax problem is a simple linear problem which will be used as an example to illustrate the concepts of EDA. It is defined in Equation 5.1

$$f_{Onemax}(X_1, ..., X_n) = \sum_{i=1}^{n} X_i \tag{5.1}$$

Where $X_1, ..., X_n$ is a string of bits. Clearly, the global optimum of this problem is the bit-string containing a 1 in every position. A chromosome for this problem can be defined as a collection of genes, each representing a single bit.

The data shown in Figure 5.2, Figure 5.3 and Figure 5.4 using a GA and EDA uses identical population sizes and selection mechanisms. The GA uses population-wide uniform crossover (See Section 4.2.3) and no mutation, while the EDA uses probabilistic uniform crossover, defined as:

- Compute the probability of a bit being 1, for every bit in the chromosome, based on its relative marginal frequency in the selected set of promising chromosomes. The set of probabilities computed constitutes the probabilistic model, and is referred to as the probability vector.
- Generate a new population by sampling the probability vector.

According to [21] there is theoretical evidence that the probabilistic uniform crossover approximates the behavior of population-wide uniform crossover. Figure 5.1 illustrates the process of the probabilistic uniform crossover.

Figure 5.2 illustrates how the probability vector correctly converges at the global optimum, where the probability of a bit being one is (very near to being) 1, for all bits in the bit string.

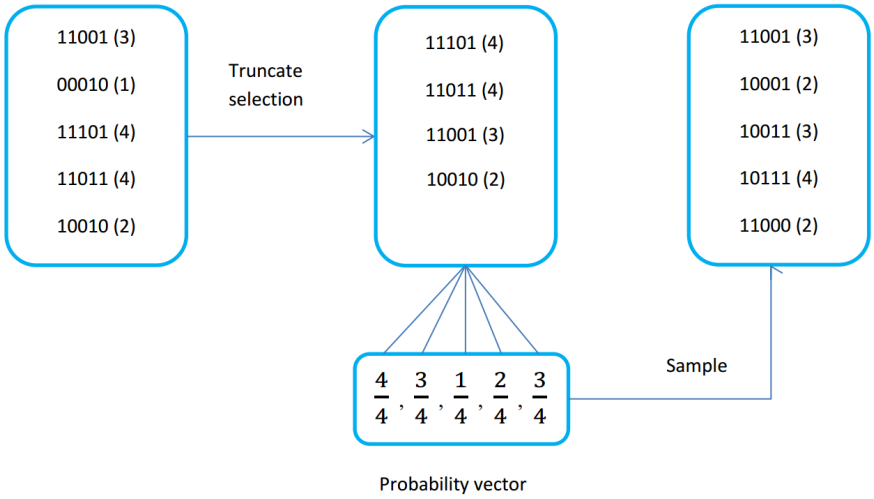

Figure 5.1: Example of a single EDA iteration.

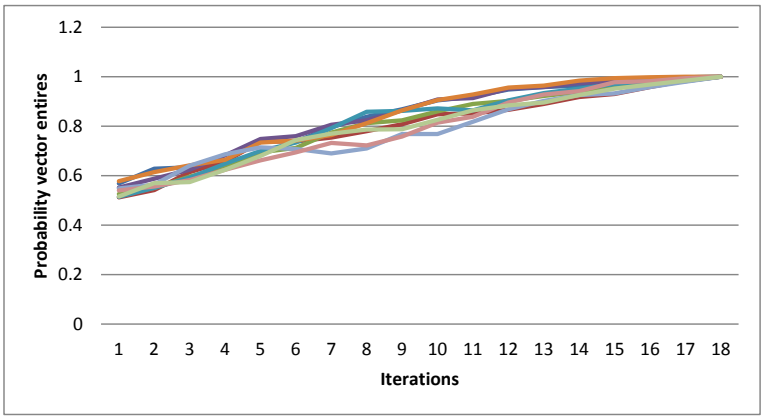

Figure 5.2: Probability vector entries in relation to iterations for the Onemax problem.

Figure 5.3 and Figure 5.4 shows the results of an experiment in which a GA and an EDA is applied to the Onemax problem, and shows that the GA performs much worse than the EDA (as expected, according to [20]).

5.4 Trap Problem

While Section 5.3 shows promising results for the EDA approach, there is a significant aspect which has been left out so far, which will become apparent when the general EDA is applied

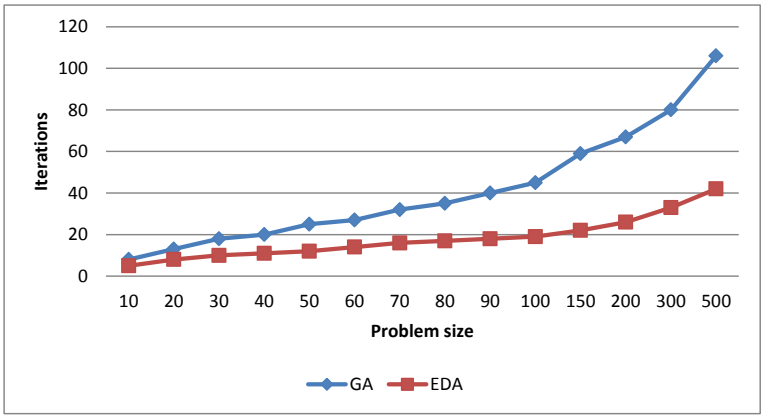

Figure 5.3: Number of iterations required to reach the global optimum in relation to problem size for the GA and the EDA, respectively.

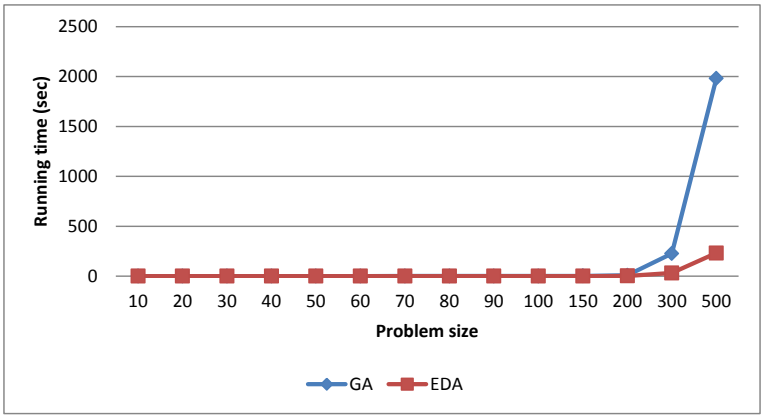

Figure 5.4: Running time measured in seconds in relation to problem size for the GA and the EDA, respectively.

to a more difficult problem, the 5-Trap problem, Equation 5.2

$$f_{trap}(n) = \begin{cases} 5 & \text{if } n = 5 \\ 4 - n & \text{otherwise} \end{cases} \tag{5.2}$$

Where n is the sum of the bits in the bit string. The trap problem contains two optimums, one local for the chromosome with the bit-string 00000 and a global for the bit-string 11111 as shown in Figure 5.5. Figure 5.6 shows the probability vector entries over time as the

algorithm runs.

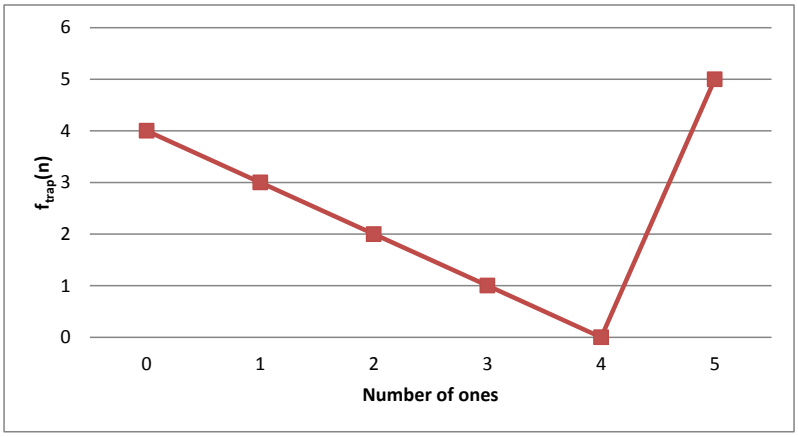

Figure 5.5: The value of the trap5 function depends on amount of ones in the input string.

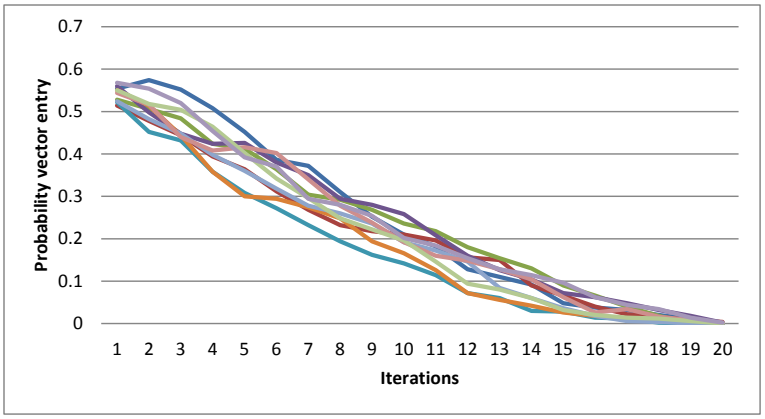

Figure 5.6: Probability vector entries in relation to iterations for the Trap problem.

As seen in Figure 5.6, the EDA converges at the local optimum of 00000. In fact, it will always converge to the local optimum given a reasonable population size, as having a 0 is statistically always preferred to having a 1, in any position of the bit-string. The mean fitness of all chromosomes starting with a 1 is 1.375, while the mean fitness for chromosomes starting with a 0 is 2 (See Appendix .1 and Appendix .2) which leads the algorithm away from the global optimum. Clearly, a single bit is an insufficient building block representation for the Trap problem. It is tempting to assume that pairs of bits would perform better as a building block, but it turns out to be a false assumption as 00 has a higher mean fitness than both 01

and 11. In fact, it is necessary for the probability vector to consider the entire 5-bit chromosome as a building block for the EDA to be successful. Where the original probability vector model had probability entries for individual bits ($p(X_i = 1)$ and $p(X_i = 0)$ for all i bits), the correct model should instead contain entries for every possible chromosome ($p(X = 00000)$, $p(X = 00001)$, $p(X = 000011)$, ...).

As seen in Figure 5.7 the EDA with the correct model is able to identify and converge at the global optimum. However, to reach this result it was necessary to manually modify the model based on prior knowledge about the problem. This leads us to the concept of *Linkage Learning* which attempts to design methods capable of automatically identifying building blocks and efficiently processing these.

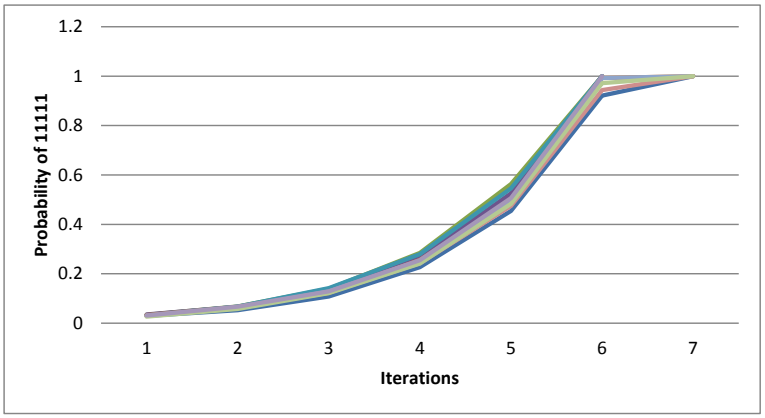

Figure 5.7: Probability of a chromosome sampled containing only 1's ($p(chromosome = 11111)$) in relation to iterations.

5.5 Probabilistic Linkage Learning

As shown in Section 5.4 much care must be taken when decomposing a problem into appropriate subproblems (building blocks). Much research in the area of EDA focuses on learning probabilistic models for proper decomposition. Many models have been proposed, some of which will be discussed in this section.

When tasked with choosing a probabilistic model there are two major concerns to be aware of:

- Independence assumptions

- Complexity of learning algorithms

The concern of independence assumption deals with how the individual parts of a chromosome are dependent on each other. For the Onemax problem (See Section 5.3) it was sufficient to consider every bit to be completely independent of every other bit. It is, however, insufficient for more deceptive problems such as the Trap problem (See Section 5.4) where every bit is dependent on all other bits (a 1 is only preferred if all other bits are 1).

Of course, with more complex probabilistic models comes higher complexity in the algorithms needed to learn the model. A model may be able to perfectly capture all interdependencies of the problem, but if its complexity is too great a large portion of the motivation for using an EDA is lost.

Existing EDAs can for the most part be categorized based on their independence assumptions.
Univariate models assume no interaction between the individual parts. Algorithms in this category include BSC ([22]), PBIL ([23]), Compact GA ([24]), UMDA ([25]) and DEUM ([26]). The examples shown in Section 5.3 and Section 5.4 applies UMDA. These algorithms solely does parametric learning of the model while its structure remains fixed throughout.
Bivariate models which deals with interactions between pairs of building blocks. This category adds structural learning to the EDA approach, and includes algorithms such as MIMIC ([27]), COMIT ([28]) and BMDA ([29]).
Multivariate models consider multiple interactions between the individual parts. Applications of this approach is found in the EBNA ([30]), FDA ([31]), ECGA([32]) and BOA ([20]), with BOA appearing to be an interesting approach to apply to Starcraft.
In BOA, the Bayesian network is initially constructed by generating a node for every gene in the chromosome and builds the network using a simple greedy algorithm with 1 operation: adding a directed edge between nodes in the network. The "'fitness"' of the network can be evaluated with any chosen evaluation technique but suggests using the Bayesian-Dirichlet metric (BD) mainly because it allows for using prior knowledge about the problem, by taking an optional pre-made network as input. New chromosomes are generated by sampling the built network [20].

Part III

Evolving Starcraft Strategies

CHAPTER 6

GENETIC ALGORITHM FOR STARCRAFT

A crucial step when using genetic algorithms Chapter 4 to optimize or solve problems, is to tailor it to the problem domain. This chapter covers how a genetic algorithm can be tailored to fit the Starcraft environment and describes how the chromosome encoding is designed.

6.1 Chromosome Encoding

Pieter Spronck et al (See [3]) proposes a chromosome encoding for another RTS game by the name of Wargus in which chromosomes are divided into states and genes. A state can be seen as a container of genes, and genes are constructs that represents some action taken by the agent. For instance, a build gene will cause the agent to build a building when it is executed. This chromosome encoding is well suited for most RTS games, although we propose modifications more fitting for Starcraft. This section describes our encoding in detail.

There are four kinds of genes:

- Build gene: Builds a building when executed
- Research gene: Researches an upgrade when executed
- Combat gene: Constructs a number of combat units when executed.
- Attack gene: Initiates an attack on the enemy when executed.

The execution of a chromosome entails executing each state in chronological order. If a game has just started the bot will start by executing first state of the chromosome. In order to execute a state, each gene in that state is executed. If a build gene is executed, a specific building will be constructed. If a research gene is executed, a specific upgrade will be researched. If a combat gene is executed, a certain number of combat units will be trained. And if an attack gene is executed the chromosome will initiate an attack on the enemy. When the building built by the build gene has been constructed, the chromosome proceeds by executing the next state. This continues until the chromosome no longer contains any states. A possible chromosome is shown as an example in Figure 6.1.

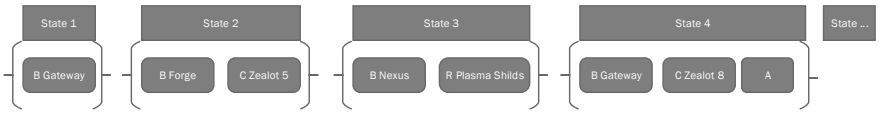

Figure 6.1: This image shows an example of a chromosome. The first state contains a build gene that will build a Gateway building. State two contains two genes. One that constructs a Forge building, and one that trains 5 Zealot combat units. State three constructs a Nexus building and researches the Plasma Shield upgrade. State four constructs another Gateway building, trains 8 Zealot combat units and initiates an attack on the enemy.

6.1.1 Creation of Chromsomes

When random chromosomes are created by the GA, they must be constructed with the rules of Starcraft in mind as to avoid the creation of invalid chromosomes that cannot be executed in the environment. As mentioned in Chapter 2 most buildings and units are not available at the start of a Starcraft match. Buildings and units must be unlocked by constructing tech buildings. This can cause problems for the chromosomes if these constraints are not accounted for. For instance, if state one of a chromosome specifies that the building Photon Cannon must be constructed, then that chromosome would be invalid since Photon Cannon requires that the Forge building has been constructed. Likewise, some units can only be trained if certain buildings have been constructed. For instance, the Zealot combat unit can only be trained if a Gateway has been constructed. In order to remedy this problem, the random generation of chromosomes in the GA is guided by a Starcraft Rule Base. This is done by simply calculating all the legal buildings, upgrades and combat unit based on the previous states in the chromosome. The following is an example of how a random chromosome might be created.

The first step of creating a random chromosome is to create the first state. At this point in the game no buildings have been created, so only a few buildings can legally be built. No combat units or upgrades are valid, since no research or training buildings are available. The buildings available are:

- Nexus
- Forge
- Gateway
- Assimilator

The first state is then simply created by choosing a random building from the list of valid buildings. The generated state can be seen in Figure 6.2. In this case, the Gateway building was chosen.

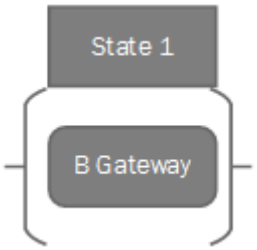

Figure 6.2

The next step is to generate state 2. Since state 2 is executed after state 1, we know that a gateway building exists. This open up for more options in the gene creation. For instance, having a Gateway building unlocks the Cybernetics Core building. It is now also possible to train Zealot combat units. We can now construct the following buildings:

- Nexus
- Forge
- Gateway
- Assimilator
- Cybernetics Core

And we can train the following units:

- Zealot

Remember that each state must contain a build gene, since a state transition is triggered by the completed construction of a building. Other genes, however, are optional. Build and research genes both have a chance of 1/4 for being included in a state, while an attack gene has a 1/8 chance of being included. Since it is now possible to train combat units, state 2 has a 1/4 chance of containing a combat gene. In this case, a combat gene was added to state 2, that trains 5 zealot units. The number of units trained is a random number between 1 and 10. The random build gene chosen for state 2 is a Forge building. State 2 can be seen in Figure 6.3.

The next step is to generate state 3. With the previous construction of a Forge building, research upgrades are now available. The following research upgrades are available:

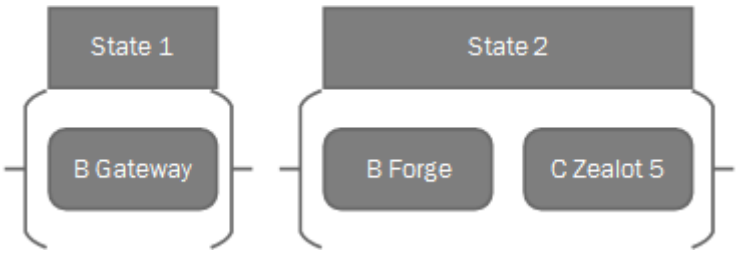

Figure 6.3

- Ground Weapons
- Ground Armor
- Plasma Shields

The following buildings are available:

- Nexus
- Forge
- Gateway
- Assimilator
- Cybernetics Core
- Photon Cannon

And we can train the following units:

- Zealot

Creating state 3 is done much like the other states. Whether state 3 will contain combat, research or attack genes will be randomly chosen. In the case of Figure 6.4, a research gene upgrading the Protoss plasma s was chosen.

Ensuring that randomly generated chromosomes obey the rules of Starcraft makes it easier to guarantee that the population does not contain invalid chromosomes.

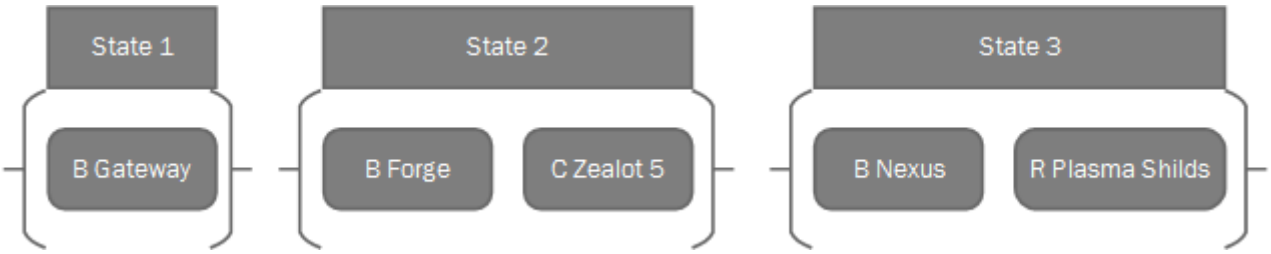

Figure 6.4

6.2 Selection Mechanism

The selection mechanism chosen for the genetic algorithm is the Tournament Selection mechanism described in Section 4.1.1. The reason for this is that it has previously been successfully applied to a problem domain similar to Starcraft [8]. Furthermore, tournament selection is often used in practice instead of Fitness Proportionate Selection (Section 4.1.2), since tournament selection tends to outperform it [33].

6.3 Fitness Function

To measure the quality of a chromosome, a Starcraft game is played where the chromosome is pitted against an enemy AI. The game is ended when one of the players is victories, or when the draw timeout of 60 minutes is reached. At the end of a game the following fitness function is used to calculate the fitness of the chromosome:

$$f = \begin{cases} \frac{S_p}{S_o} & \text{if winner} \\ \frac{ET}{MT} * \frac{S_p}{S_o} & \text{if looser} \end{cases} \tag{6.1}$$

Where S_p is the score of the agent, S_o is the score of the opponent, ET is the elapsed game time and MT is the maximum time a game can take (60 minutes). The reason for dividing S_p and S_o is that we want to favor chromosomes that achieves a high game score, and prevents the enemy from achieving a high game score. The higher the score of the chromosome, and the lower the score of the enemy, the better the fitness value. If the chromosome loses against the AI, the fitness still depends on game score achieved by the chromosome and the enemy AI. However, we multiply with $\frac{ET}{MT}$ in order to favor chromosomes that were able to last longer against the enemy. If the chromosome was defeated quickly, $\frac{ET}{MT}$ will make the fitness value small. However, if the chromosome was able to hold out for a long period of time, the reduction to the fitness value caused by $\frac{ET}{MT}$ will be smaller. This part of the fitness function is very important since most of the chromosomes will lose to the enemy. It

is therefore necessary have some form of reward for losing chromosomes who were able to at least stay alive longer than losing chromosomes who were defeated quickly.
The enemy game score and chromosome game score are not available to us by default. The score is calculated adding several score values provided by BWAPI. BWAPI (described in Chapter 2) provides three different scores to be considered. Unit score, which is determined by the units the player has trained. Kill score, which is determined by the units the player has killed, and building score which is determined by the buildings the player has constructed. The score for player p is determined by:

$$S_p = unitScore_p + killScore_p + buildingScore_p \tag{6.2}$$

6.4 Genetic Operators

The choice of genetic operators is influenced by the choices advocated in [3] and are as follows: Three Point Crossover, Rule Replace Mutation, Rule Biased Mutation and Random Chromosome Creation. Each has a 30% probability of being applied except for Random Chromosome Creation which has a 10% probability.

6.4.1 Rule Replace Mutation

Every research, combat and attack gene in the parent chromosome has a 25% probability of being replaced by a random new gene. Building genes are excluded from this process since replacing a build gene might result in an invalid chromosome. For example, if state 5 of a chromosome contains a build gene that builds a Cybernetics Core, and that gene is replaced with a build gene that builds a Nexus, then the subsequent states might have built genes that requires the Cybernetics Core, rendering the chromosome invalid.

6.4.2 Rule Biased Mutation

Each combat gene and attack gene in the chromosome has a 50% probability of having its parameters mutated. For instance, a combat gene might produce 7 Zealot combat units. The parameter that will be mutated is the number of zealot units produced. Research and build genes are excluded from the rule biased mutation process. This is because mutating these genes might produce invalid chromosomes.

6.4.3 Random Chromosome Creation

The random chromosome creation does not require any parent to produce a new child. A child is simply randomly created (See Section 6.1.1). The reason for having this genetic operator is to introduce new genetic variety into the population in an attempt to avoid local optimums.

6.5 Search space approximation

To get a better understanding of the complexity of the problem at hand, the size of the search space is investigated in this section. While in reality the search space is infinite, as a state can contain an unlimited amount of genes, the calculations in this section assume that each state has one gene of every type.
Ideally the search space should be calculated precisely, taking into account the limitations imposed by technology tree (See Figure 2.1) and the rules of StarCraft in general.
As a starting point, we only consider the build genes of the chromosome. In the first state, the build gene can, according to the technology tree, take one of four different values: Nexus, Assimilator, Forge or Gateway (Pylons are built automatically by the bot) (See Figure 6.5).

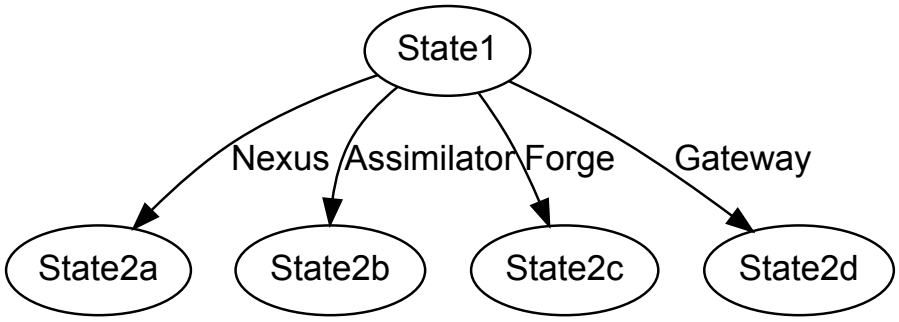

Figure 6.5: Search space tree for 2 states.

Depending on the value of the build gene in state 1, the build gene in state 2 can take either 4 or 5 different values. Building a Forge in state 1 opens the door to build a Photon Cannon in state 2. Likewise, building a Gateway opens up the possibility of building a Cybernetics Core, as shown in Appendix .3.

If we count the leafs of the tree we see that 18 different chromosomes can be created if we consider only 3 states and only include build genes in the chromosome. As the chromosome encoding used in this project has 50 states as well as attack, research and combat genes in addition to build genes, it quickly becomes impractical to precisely calculate the search space.

Instead, the search space is approximated. When the chromosome is grown to 9 states, it is possible for a chromosome to have unlocked all 14 buildings (The search space is at this point at 8694640). The approximation then considers all subsequent states to have access to all 14 buildings. Of course, this approximation will give us a larger search space than reality, and the error will accumulate and grow exponentially as the chromosome is grown larger to include all 50 states. Nonetheless, it is considered useful to provide a rough idea of the complexity of the problem at hand.

$$50states = 8694640 * (41 * 14) \approx 5billion \tag{6.3}$$

Of course units and research should be handled in a similar fashion. After 12 states, all 11 units and all 24 researches are potentially unlocked. Unlocking units and upgrades relies entirely on buildings including them in the precise calculation becomes impractical. Instead, as an approximation, no upgrades or units are available until state 12 at which point they all become available. Note that every combat gene can build between 0 and 10 of a unit:

$$8694640 * (41 * 14) * (38 * 24) * (38 * 11 * 10) \approx 19quadrillion \tag{6.4}$$

Finally, each state can contain either an attacking or nonattacking gene:

$$8694640 * (41 * 14) * (38 * 24) * (38 * 11 * 10) * (2 * 50) \approx 1,9 * 10^{18} = 1quintillion \tag{6.5}$$

CHAPTER 7

CROMARTIE IMPLEMENTATION

This chapter describes the implementation of our two AI agents, from here on referred to as bots or GA Cromartie and Static Strategy Cromartie. The purpose of the bots is to test the validity of Hypothesis 1 from the problem statement in 1.1. The two bots are similar in many areas, due to much of the code being shared - thus when talking about the shared components or both bots, they are simply referred to as Cromartie, and not their individual names. The function of the bots are essentially to play Starcraft Brood War, but each Cromartie bot is developed with a different sub-purpose. For instance, the GA Cromartie is used to playback chromosomes as directed by the Genetic Algorithm described in Chapter 4, while the Static Strategy Cromartie play Starcraft by following a static strategy given as a parameter before game begins. This chapter first describes the common components and functionality of the two bots, where-after the components that are different between the bots are presented. The bots are based on the bot called Skynet[10] and is built using the application framework called BWAPI (see Section 2.3).

The first section presents the tasks of the bot as well as the reason for using a different bot, such as Skynet, as the base of development. The first section also makes a short introduction to a few of the Starcraft bots that were considered a suitable base for Cromartie.

The second section covers Skynet, the basis of Cromartie. It presents the most important components from Skynet that is used for developing Cromartie.

The third section describes the additions and modifications of Skynet, made during this project.

The fourth section describes the Genetic Algorithm Cromartie, and how the Genetic Algorithm is implemented as a component within Cromartie.

The last section is a short presentation of the Static Strategy Cromartie and the External Build Order component, unique to this bot. It also introduces the Nexus14 strategy/build order and, using Nexus14 as an example, how strategies are constructed and given as a parameter to the bot.

The name Cromartie comes from the fictional universe of the Terminator franchise. A bot called Skynet that serves as the base of the bot developed during this project is named after the self-aware AI antagonist of the Terminator franchise. Thus it seemed appropriate to find the name for our bot, within that same universe. The choice fell on Cromartie, a Terminator agent sent back in time by Skynet, and the main antagonist in the spin-off TV series, Sarah Connor Chronicles. [34][35]

7.1 Tasks of Cromartie

In order for the bot to be able to play Starcraft effectively, a set of common tasks is to be identified in order to carve out the required functionality of the bot. As described in Chapter 2 Starcraft is a game of gathering resources, effective use of these resources and managing units, in order to eventually defeat an enemy with the same prerequisites as one self. From this the following tasks can be identified:

- Gather resources
- Expand to new base locations
- Construct buildings
- Train units
- Upgrade technology
- Micro-manage combat units
- Scout map and enemy bases
- Launch attacks

All of these tasks are important and must be handled by the bot in order for it to play effectively, but only a few of the tasks are essential for this project. This is because the purpose of the bot is to test the hypotheses described in Section 1.1, which are concerned with strategy, but not all of the listed tasks are used for managing the strategy of the bot. Thus these tasks essential for managing strategy, should have the highest focus for the development of Cromartie, while the remaining tasks should, if possible, be handled by third-party code or libraries.

Name	# Games	Crash %	Build order	Scouting	Code	Micro	Resources	Supply
UAlbertaBot	11421	0.11	8	7	4	6	7	8
SCAIL	1656	0.32	4	8	8	4	10	5
Nova	5995	0.15	10	5	8	10	10	5
Skynet	11888	0.05	7	10	2	8	10	10

Table 7.1: This table show the potential bot bases investigated, along with the evaluation criteria and scores. Scores with underlines is the highest/best in the given criteria.

The tasks related to managing strategy are: Expansion, Constructing buildings, training units, upgrading technology and launching attacks. The tasks less essential to this project, although equally important to be handled, are: Gather resources, micro-management of units and scouting.

By suggestion of the BWAPI [6] project website, a library called BWSAL was initially investigated, as it provided all of the aforementioned low priority tasks, along with a range of helping tools such as terrain analysis and building placement. This library was found to be out-of-date and faulty, requiring us to use a different approach. Instead of using BWSAL, a number of open-source Starcraft bots, all competing in the 2012 Starcraft AI Competition[9], was investigated in order to find a code base to build Cromartie from. A number of criteria was defined, which were used to evaluate these different bots. The bots chosen for evaluation and the different criteria is shown in Table 7.1, where each criteria except *# Games* and *crash %* were given a score between one and ten. From this evaluation of bots *Skynet* was chosen, as it was both very stable and managed the low priority tasks very well. One difficulty with Skynet was the sheer size of the code and the way much of its modules were strictly coupled, which would make the process of building upon it harder. This difficulty was in part conquered, by exchanging much of the strict coupling with an event/messaging system, which is described later in Section 7.3.

In the next section we describe the components and code provided by Skynet.

7.2 Skynet

The Skynet bot is a bot competing in the 2012 Starcraft AI Competition and is created and maintained by Andrew Smith [10]. In this project the different versions of Cromartie is developed as an extension and modification of Skynet, such that the development of Cromartie was focused on the tasks related to managing strategy.

Skynet is a large and complicated bot (22868 Lines of Code (LOC) at the time of writing this report), so covering all of its components and functionality would be infeasible. Instead the most important components are introduced. These are:

Terrain Analysis Manager is responsible for analyzing the map and create a graph structure by dividing the map into regions as nodes, connected with choke-points[1] as edges. The Terrain Analysis Manager also uses the map information to create a list of potential base locations and determining initially where buildings can be placed.

Unit Tracker is responsible to keep track of active units in the game, both friendly and enemy units. Units in BWAPI includes workers, troops, buildings and other actors like minerals and geysers. The Unit Tracker informs other subsystems of Skynet when a new unit is observed e.g. a new worker is trained, a building is built or a mineral patch is observed through the fog of war. It also informs the subsystems of the destruction of units e.g. when a troop is killed in combat or a mineral patch is depleted.

Task Manager is the component that manages the tasks given to Skynet from other components. A task in this sense could be to construct a building, train a unit or gather resources. One or more units may be assigned to a task and a task can have preconditions. Thus if a task is given to the manager from a component, but its precondition is not yet fulfilled, the task is put on hold until it is ready. E.g. a construction task may have the precondition of having a worker assigned and enough minerals to perform the construction, and will stay on hold until enough resources have been gathered.

Squad Manager is responsible for the micromanagement of combat units. Trained troops are placed in one or more squads, where each squad is associated with a high-level responsibility, e.g. defense, attack, drops, etc. Each squad then controls the micro-actions of each individual unit, whether in combat or idle.

Scout Manager is the component that uses any available units, e.g. workers and observers, to gather information of the state of the game. This is done by moving the scouting unit around on the map and try to cover as much ground as possible. Whenever the Scout Manager observes something interesting, e.g. an enemy technology building, it passes on this information to any component interested - the Unit Tracker for instance.

Building Placer is a component used to help position buildings on the surface of the map. When a component sends a construction task to the Task Manager, the task includes information about the type of building to construct and a high-level position, i.e. whether to place it at a base, a choke-point, a proxy (a somewhat hidden location near the enemy base), etc. It is then the Building Placers responsibility to determine exactly at which map-tile to start building. This done by first trying at the center of the proposed location, e.g. the center of the base, and if this location is unavailable (from other buildings or structures) then tries again at a different location spiraling the center, in the case of building at the base, while increasing the radius, until a valid location is found.

[1] A choke-point is another term for a bottleneck between regions. Whether they actually "choke"' or not is a point of discussion

Skynet includes a lot of different components, which differ much in complexity, coupling with other components and size. One commonality with the Skynet code is that there is a lot cross-cutting concerns between the components, and it is very difficult to extend and modify the bot as was the purpose of selecting Skynet. In Figure 7.1 the public function call dependencies of some of the components in Skynet is presented. The diagram shows a web of inter-dependencies and illustrates the difficulty of extending Skynet. The first challenge of developing Cromartie based on Skynet was thus to change the structure of Skynet to be easier to extend. The solution to this challenge was to add an event-/messaging component, which is addressed in Section 7.3.1.

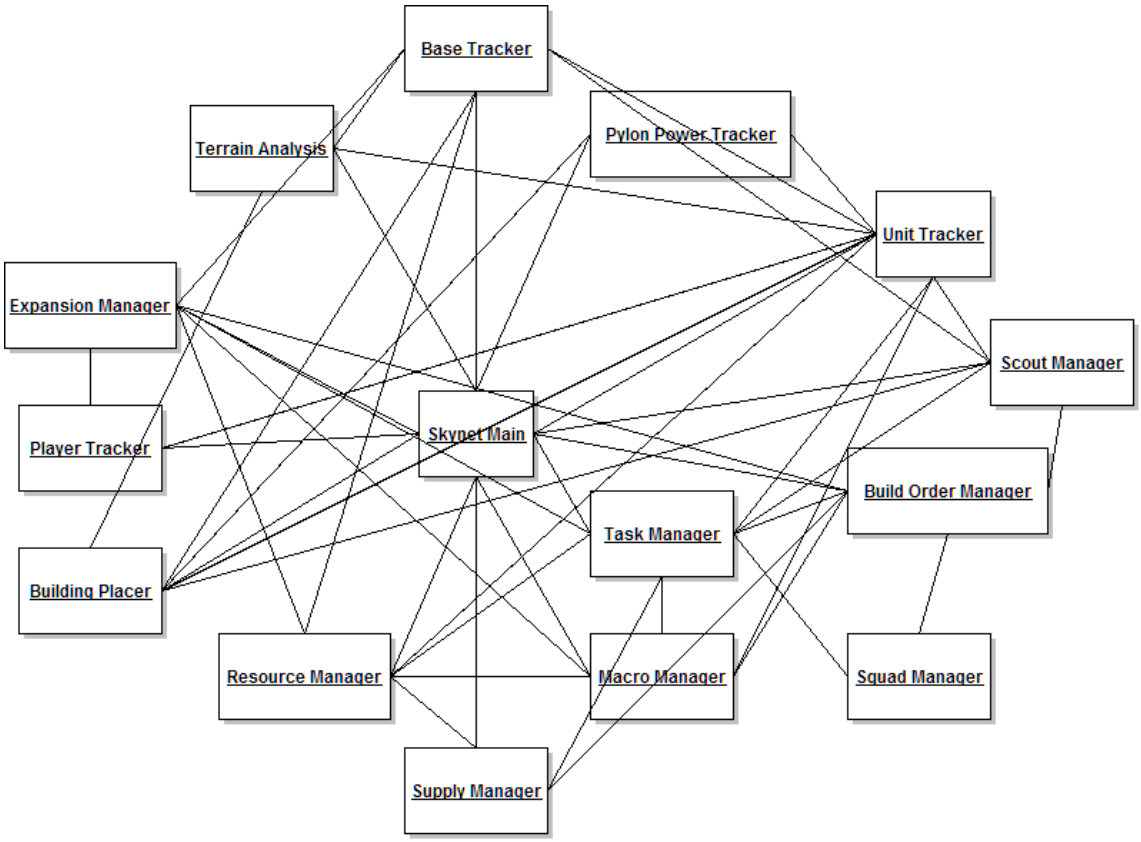

Figure 7.1: This figure shows some of the components of the Skynet bot and the dependencies between them. Note that the dependencies of this diagram only show public function calls and not class associations, aggregations and inheritance dependencies. Also, the diagram is not meant to be understood, it is merely a visualization of one of the difficulties with having selected Skynet as a base.

7.3 Modifications and Additions

One of the first development issues of this project was to mold the Skynet bot into Cromartie by extending and modifying the Skynet code. The purpose of this was to enable and make it easier to extend the bot with the Genetic Algorithm component or the External Build Order component in order to build the different versions of Cromartie.

7.3.1 Event Management Component

The first modification was made to solve the issue of the large amount of cross-cutting dependencies of Skynet. One common way to solve this issue is to apply an Observer Pattern[36], specifically by using an event/messaging system. Thus an Event Manager was created for Cromartie, that each component would call to queue events, instead of calling each other, such that the Event Manager in turn could pass on these events to any "'listening"' components. The goal was to turn the highly coupled architecture in Figure 7.1 into Figure 7.2. In Figure 7.3 the class diagram of the Cromartie Event Management component is shown and in Figure 7.4 a sequence diagram for the Event Manager is shown.

In the class diagram of Figure 7.3 there is a class called `EventManager` - this is the main class of the component. When the bot begins executing, some listeners (the observers of the Observer Pattern) are added to the EventManager, associated by an `EventType`. A listener is a delegate, a function pointer that handles a given type of event. For example, the Unit Tracker component has a function called `UnitDiscoveredEventHandler` which serves as the event listener for UnitDiscovered events.

In the sequence diagram of Figure 7.4 an example of a possible sequence of calls in Cromartie is visualized. First Starcraft - or BWAPI - calls the `onStart` function of Cromartie, which in turn calls the private `registerListeners` function, that is responsible for adding all the required listeners to the EventManager. For each game frame in Starcraft after the `onStart` function have been called, Starcraft calls the `onFrame` function of Cromartie. In this `onFrame` function a list of game events from BWAPI is collected and sorted into the event queue of the event system. For instance will any unit discover events from BWAPI, be queued as a `UnitDiscoveredEvent` in the `EventManager`. Also in each frame the event `OnUpdateEvent` is queued, before the `onFrame` function finally calls `Update` of the `EventManager`. The Update function is responsible for taking each event which has been queued, and execute any relevant listener delegates associated with each given EventType, and sending the data of the event as a parameter to the delegate. For example in the sequence diagram a `UnitDiscoveredEvent` was queued, and when the `EventManager` is updated the listener function `UnitDiscoveredEventHandler` in the Unit Tracker component is called.

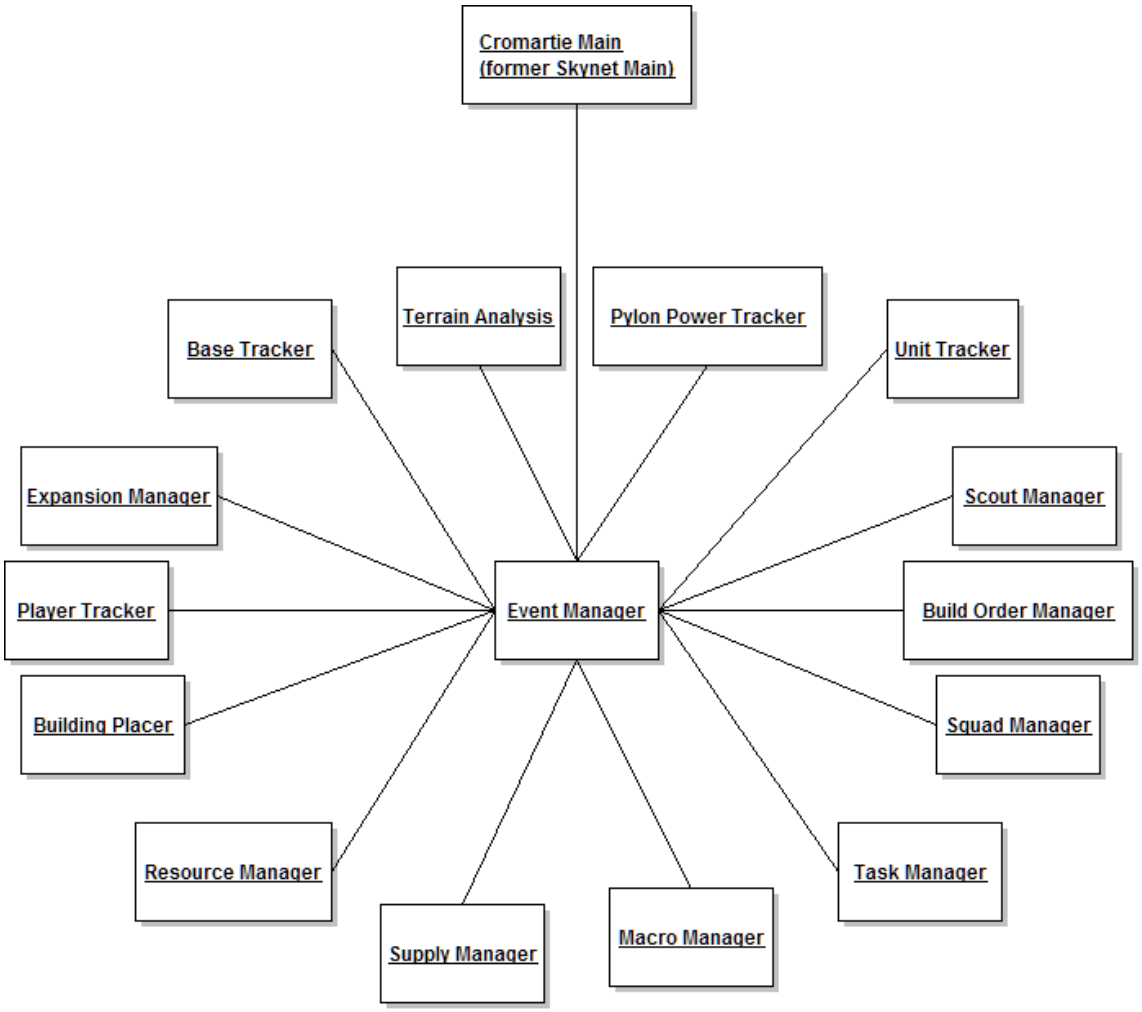

Figure 7.2: This figure show how the components communicate after applying an Observer Pattern.

The event management system was inspired by the Observer Pattern[36] and the game event management system proposed in the book Game Coding Complete[37]. The event system developed in this project uses delegates rather than the usual observer/listener classes which usually, according to the Gang of Four Observer Pattern, implements some IObservable interface. Specifically the framework fastdelegate[38] is used to create easy listener delegates and thus avoid using function pointers directly, for faster and easier development.

7.3.2 Decoupling Skynet Specific Components

Some of the components provided by Skynet is replaced by the components specific to Cromartie, such as the Genetic Algorithm component and the External Build Order component.

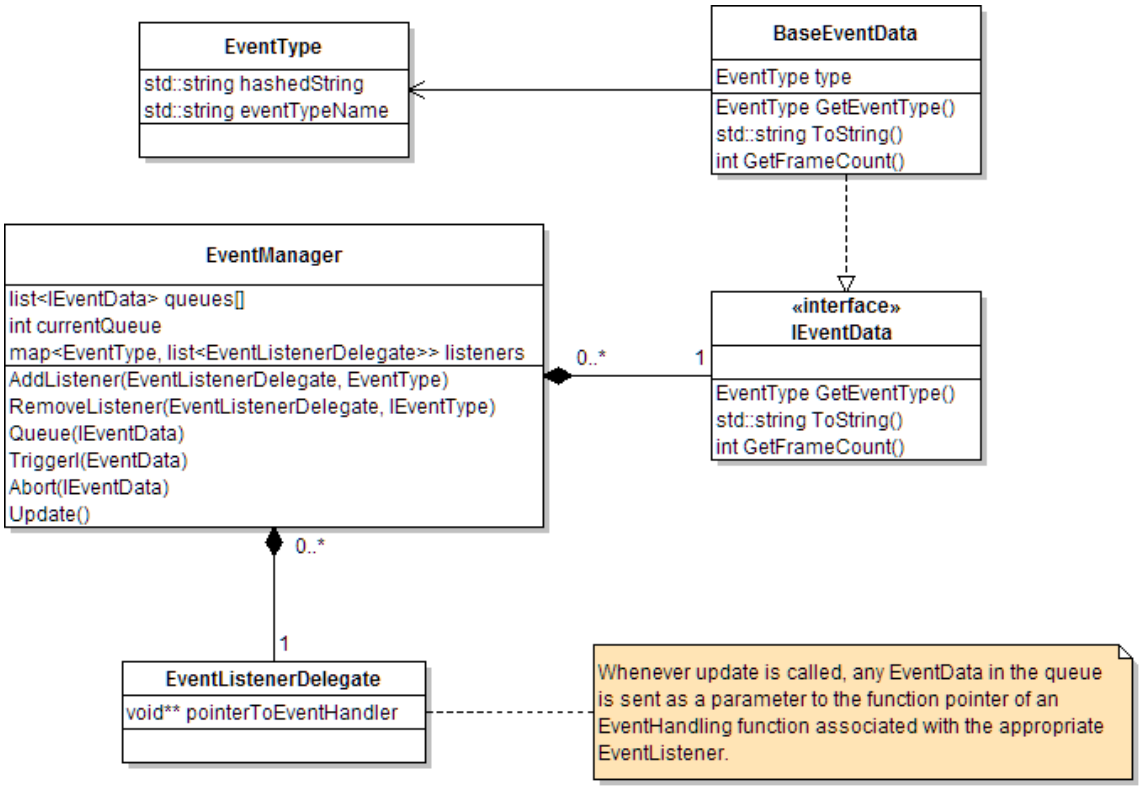

Figure 7.3: This figure show the class diagram of the Event Manager component of Cromartie.

The second development task was thus to identify, remove and decouple those Skynet specific systems from the code, while still keeping the rest of the functionality provided by Skynet intact.

Build Order Manager

Skynet contains a component responsible for deciding what build order to use against the enemy. This system conflicts with both our Genetic Algorithm component, where the generated chromosomes controls which buildings are built at any given time, and our External Build Order component, where we want to impose a specific build order right when the game starts. Thus for our purpose, parts of this component was rendered inactive in Cromartie, although some of it was still kept, as it was used as a base for the External Build Order component, which is described later in Section 7.5.

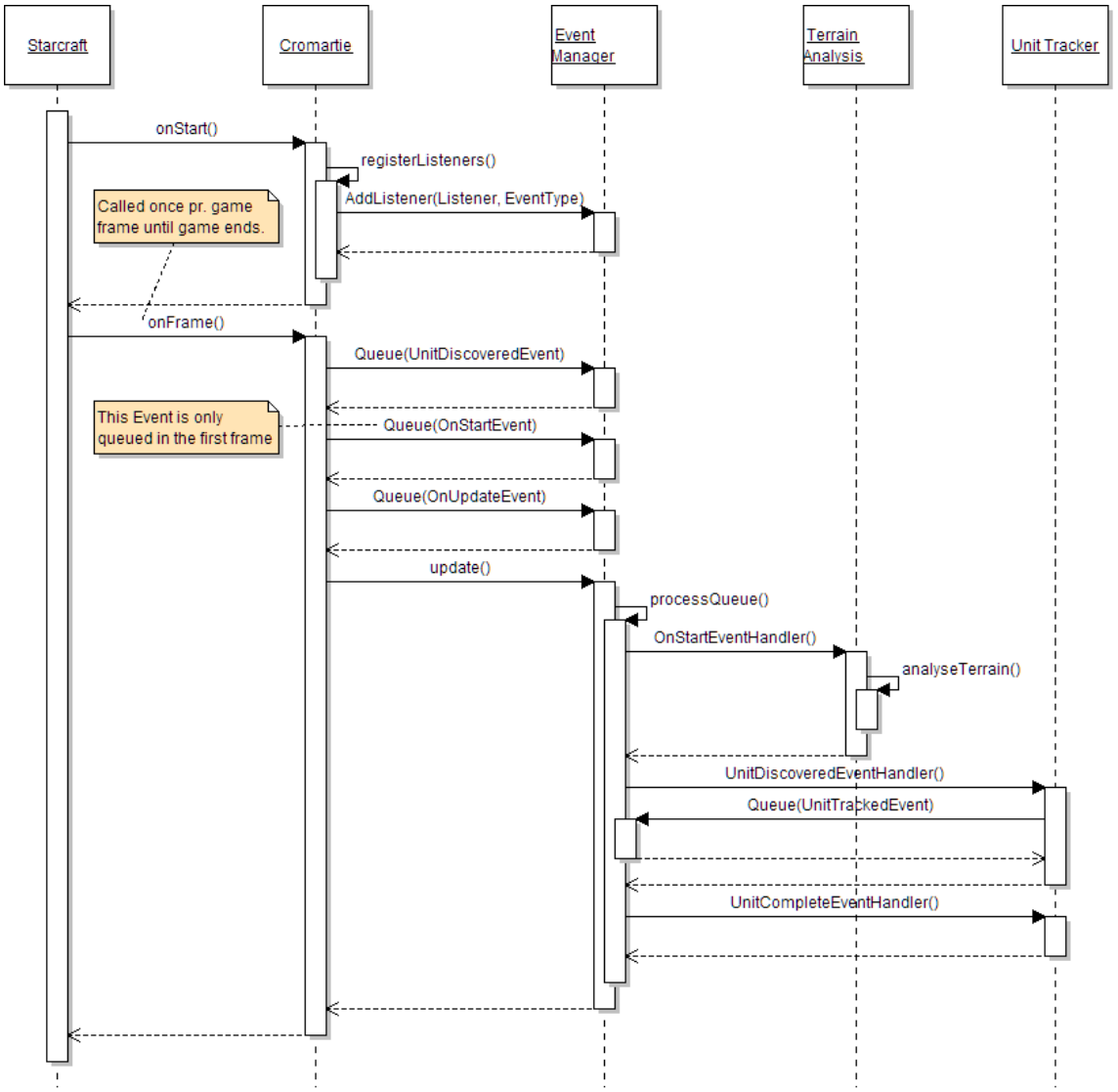

Figure 7.4: This figure show a partial sequence of how the event system works in Cromartie.

Squad Manager

In our Genetic Algorithm component Chapter 6 the chromosomes have the power to decide when Cromartie should attack. Skynet though, uses the Squad Manager to figure out when to attack and what. Skynet does this using a Default Squad that is associated with all units of the bot, and then at each frame performs calculations related to the decision on when to attack. This decision is taken based on an army behavior flag (aggressive, defensive, all-in, etc.) defined as a part of the build order, as well as the known and guessed information about the enemy. This Default Squad was of no use to Cromartie, so a different Squad - called Attack Squad - was developed, with the sole purpose of attacking when commanded to. Combining

this squad with the event system, the Genetic Algorithm component was granted the power to attack whenever a chromosome was generated with one or more attack genes.

7.4 Genetic Algorithm Cromartie

The GA part of Cromartie, is an implementation of the bot where the tasks related to managing strategy is handled by a genetic algorithm. The purpose of this part of Cromartie is to test the accuracy of the first three hypotheses from Section 1.1, by observing that our genetic algorithm, converges towards a strategy that consistently wins against both a bot with a single static strategy, and a bot that switches between three strategies. This section described the implementation of the Genetic Algorithm component, specific to this implementation of Cromartie. The tasks assigned to the genetic algorithm are:

- Construct buildings
- Train units
- Upgrade technology
- Launch attacks

The reason for this division of tasks is that we want Skynet to handle the mundane micro management, while macro management is left to the chromosomes evolved by the GA. The rest of this section will describe how the genetic algorithm was implemented on top of Skynet.

Figure Figure 7.5 shows the main classes of the GA implementation.

All gene classes inherit from the abstract Gene class so that we can maintain a list of genes in each State object. The Chromosome class maintains a list of states associated with it. The maximum number of states a chromosome can hold is 50. This value was chosen since it is unrealistic for a Starcraft game to go on for so long that more than 50 states are executed.

The main class of the GA is the GA class. Its responsibility is to manage the population of chromosomes, performing tasks such as generating the initial population, and breeding new children. How these classes interacts with each other and Starcraft can be seen in the sequence diagram in Figure 7.6.

When a Starcraft game starts onStart in Cromartie is called, which then calls the onStart function in GA. If this is the very first Starcraft game, GA generates an initial random population. If this is not the first Starcraft game, the population is loaded from a file. The reason for

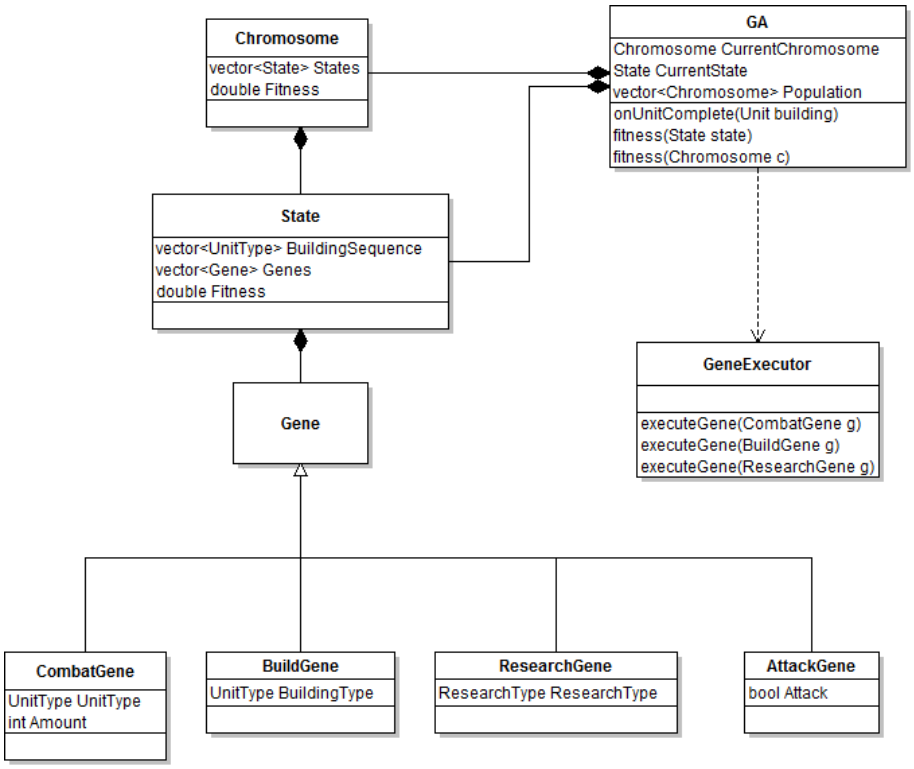

Figure 7.5: This figure shows the main classes of the genetic algorithm.

having the population in a file is that when a Starcraft game ends, Cromartie is destroyed and everything in RAM is deleted. Therefore we need to save the population when a game ends, and load it again when a game starts. When the population has been loaded, a chromosome that has not been evaluated is chosen for execution by calling setActiveChromosome. After that, the first state in the chromosome is executed. When the build gene has finished constructing a building, onUnitComplete is called. This triggers the GA to fetch the next state in the chromosome and execute it. This is repeated until the game ends. When that happens, onGameEnd is called and the fitness of the chromosome that played the game is calculated. Finally, the GA saves the entire population so that it is not lost when Cromartie is destroyed.

7.5 Static Strategy Cromartie

The Static Strategy Cromartie bot is a version of Cromartie where the Genetic Algorithm component described in Section 7.4 is replaced by the External Build Order component. The term build order in Starcraft is basically a synonym for strategy. The purpose of the Static Strategy Cromartie is to provide an opponent for the GA Cromartie bot, that uses exactly the same strategy in each game played against it, such that the GA evolves to win against this

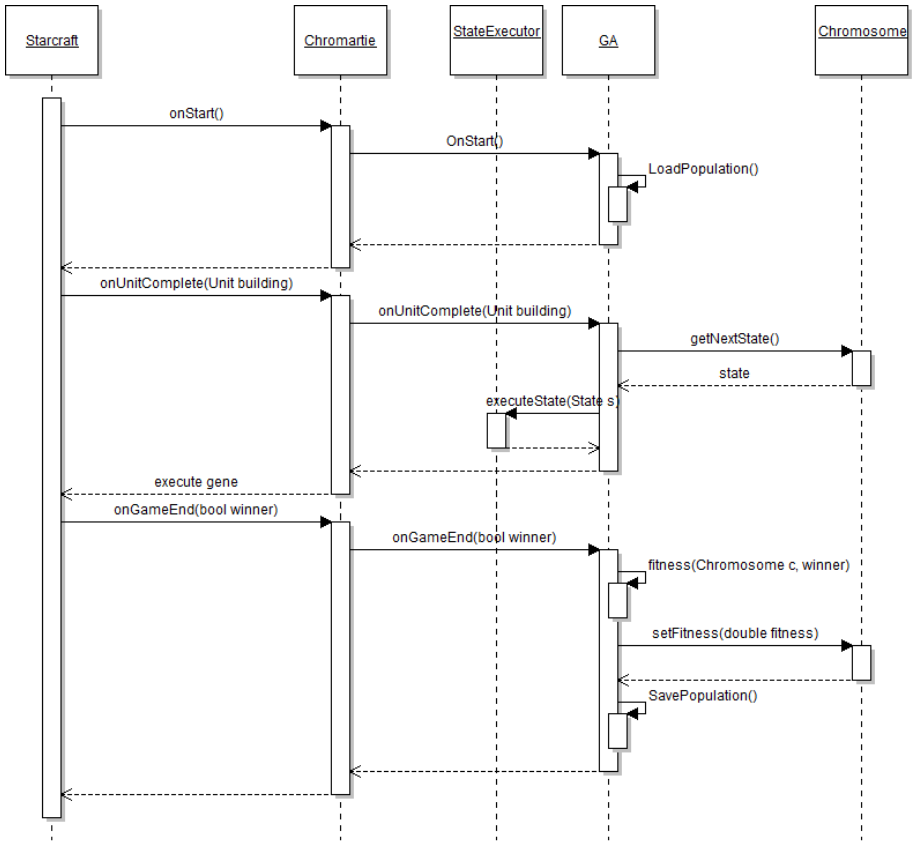

Figure 7.6: This sequence diagram shows the process of the genetic algorithm.

given strategy. As stated earlier the bots are made up of components, handling the different tasks of the bot. The components related to managing the strategy, e.g. the GA component and the External Build Order component, is different in the two bots, whereas the rest of the components, those not related to managing strategy, is exactly the same in the two bots. The reason for this is to minimize the amount of influence the components not related to strategy, e.g. the component controlling micro-management of units, have on the result of the GA. For instance, if one bot was better at micro-controlling units in combat, this would give that bot an unfair advantage, not related to the strategy, thereby bringing more uncertainty into the eventual results from testing the GA.

Skynet contains a Build Order Manager component, which selects build orders depending on the state of the game, such as the race of the enemy, etc. This component is also responsible for executing build orders. In GA Cromartie, this component from Skynet is turned off, but in Static Strategy Cromartie, the component is instead modified. The modification consists of inhibiting the build order selection mechanism, as this is not needed due to the build order being provided by the input parameter. Instead only the execution of build orders in the Build Order Manager is used to execute the build order given as a parameter.

The construction of the build order Nexus14[39] can be seen in Listing 7.1. Nexus14 is the build order used to play against GA Cromartie while testing the hypothesis in Chapter 8. Nexus14 is arbitrarily selected from the expert Protoss builds found on the Liquidpedia website[4]. The way the construction works, is by first instantiating the `BuildOrder` object with the race, name and ID of the build order. Then the actions of the build is added in sequential order, as this is the order the actions will be executed in by Static Strategy Cromartie. For instance, in Nexus14 first 4 Probes (worker units) is added to the build order, followed by a Pylon. The name Nexus14 comes from the goal of getting a second Nexus building once the players supply consummation is at 14 - for this build order it means that Cromartie should begin building the second Nexus when its supply is at 13. It is the common convention to use current supply consummation, as an identifier for when to execute each order, instead of game time. This is because that if game time was used, and the player using the build order started building the Nexus a few seconds late, he would have to recalculate the times of the build order in order to follow it through the rest of the game. Thus using supply is much easier to follow, especially for new players.

```
1  BuildOrder fourteenNexus(BWAPI::Race::Protoss, BuildOrderID::FourteenNexus, "14 Nexus");
2
3  // Build Order - every order is executed sequentially
4  fourteenNexus.addItem(Protoss_Probe, 4);
5  fourteenNexus.addItem(Protoss_Pylon); //Pylon on 8
6  fourteenNexus.addItem(Protoss_Probe, 5);
7  fourteenNexus.addItem(Protoss_Nexus, 1, BuildingLocation::Expansion); //Nexus on 13
8  fourteenNexus.addOrder(Order::Scout); // Scout
9  fourteenNexus.addItem(Protoss_Probe);
10 fourteenNexus.addItem(Protoss_Gateway); //Gateway on 14
11 fourteenNexus.addItem(Protoss_Probe);
12 fourteenNexus.addItem(Protoss_Assimilator); //Gas on 15
13 fourteenNexus.addItem(Protoss_Probe, 2);
14 fourteenNexus.addItem(Protoss_Cybernetics_Core); //core on 17
15 fourteenNexus.addItem(Protoss_Gateway); //gate on 17
16 fourteenNexus.addItem(Protoss_Zealot); //zealot on 17
17 fourteenNexus.addItem(Protoss_Probe, 2);
18 fourteenNexus.addItem(Protoss_Pylon); //pylon on 21
19 fourteenNexus.addItem(Protoss_Dragoon, 2);//2 Dragoon on 21
20 fourteenNexus.addItem(Singularity_Charge);//range on 25
21 fourteenNexus.addItem(Protoss_Probe, 2);
22 fourteenNexus.addItem(Protoss_Pylon); //pylon on 27
23 fourteenNexus.addItem(Protoss_Dragoon, 2);//2 Dragoon on 27
24 fourteenNexus.addItem(Protoss_Probe, 2);
25 fourteenNexus.addItem(Protoss_Pylon); //pylon on 33
26 fourteenNexus.addItem(Protoss_Dragoon, 2);//2 Dragoon on 35
27
28 // Define build order to transition to, 2 min after the opening strategy is complete.
29 fourteenNexus.addNextBuild(BuildOrderID::CitadelFirst, 24*60*2);
```

Listing 7.1: The Nexus14[39] build order used as the opponent of GA Cromartie.

Part IV

Results and Discussion

CHAPTER 8

RESULTS

This chapter describes the results acquired from testing the hypothesis described in Section 1.1:

Hypothesis 1 *It is possible to use a genetic algorithm to evolve a Starcraft strategy that can consistently win against a single other strategy.*

This chapter includes a description of the physical test setup, a test plan describing which and how the tests were performed, the results gathered and finally a discussion of the results. The opponent that the GA is evaluated against, which is a modified version of Skynet (See Section 3.3) executing the Nexus14 strategy presented in Section 7.5.

8.1 Test Setup

The fitness function of the GA used to evolve Starcraft strategies requires the chromosome under evaluation to be executed in the game. The chromosomes must be evolved against a single other strategy. The only way to restrict the opponent to always play only a single strategy is to create a bot with this behavior. Playing against a bot is only possible using the multi-player mode(s) of Starcraft, causing a bottle-neck in which the evaluation speed is severely limited by the game speed. The evaluation of a single chromosome takes anywhere from 5 to 60 minutes.
In order to evaluate a population at an acceptable and practical speed, the chromosomes are executed in parallel. The test setup consists of multiple PC's running multiple virtual machines with each pair of virtual machines being connected by an internal virtual network. In every pair of virtual machines one is considered the host (executing the GA) and one is the client (executing the Nexus14 strategy). A Java program is responsible for starting Starcraft and synchronizing the client and the host. Menu automation is handled partly by Chaoslauncher and partly by AutoHotKey scripts. All chromosomes are stored in a central database. Figure 8.1 shows the test setup.

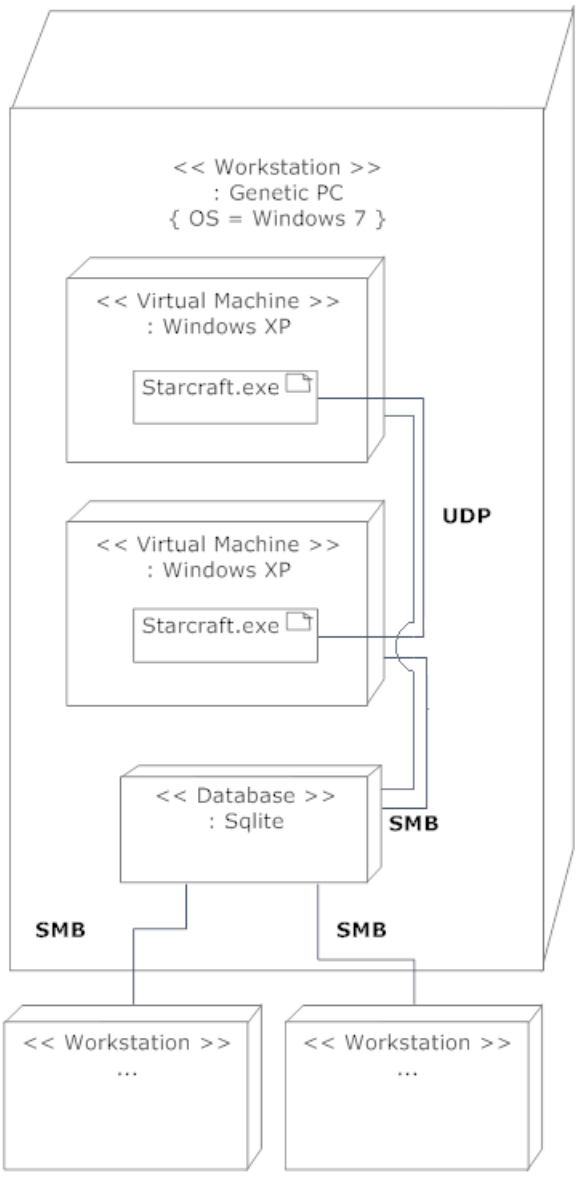

Figure 8.1: Deployment diagram of the test setup. The GA is executed by the Starcraft.exe process and stores its results in a SQLite database accessed through the network over the SMB protocol (Windows filesharing) using a C++ API. The Starcraft.exe procceses play multiplayer against eachother over UDP. Additional nodes (Workstations) can be added to the network at will.

8.2 Test Plan

In order to test the hypothesis, two tests were performed. The first aims to tune the parameters of the GA, while the second aims to produce the most effective chromosome possible.

8.2.1 Test 1: GA parameter tuning

Empirical evidence suggests that a population size of 50 is sufficient to generate effective chromosomes in a RTS problem domain using a similar chromosome encoding as the one used in this project ([3]). However, considering the significant size of the problem at hand we make the claim that it could benefit from a larger population and thus (potentially) having access to more building blocks. As such, to tune the parameters of the GA a test is performed with a population size of 50 and 100.
A valid argument could be made that the population size should be even larger (significantly larger, even tens of thousands), but a defining aspect of the problem is the time consuming evaluation. As we know, a larger population size requires more evaluations to reach convergence [19] which makes a large population impractical.

The purpose of Test 1 is two-fold. First, it is to determine the ideal population size. In an attempt to reduce the impact of the other parameters of the GA (Crossover sample size, winner size and loser size), each test was performed twice with different crossover parameters. Ideally each test should be performed many more times and more combinations of operators and parameters should be used, but the evaluation speed of Starcraft is an heart-breakingly limiting factor that trumphs the ideal approach. Two sets of parameters were chosen based on initial tests which will not be covered further.
Second, the test could produce winners. This would be a valuable result, as it would make the belief in our chromosome encoding stronger.

8.2.2 Test 2: Evolve a powerful chromosome

The goal of this test is to produce a winning chromosome and directly validate or invalidate the hypothesis mentioned in Chapter 8 by running the GA algorithm with the parameters deemed promising in Section 8.2.1. Any winning chromosomes gathered in Section 8.2.1 are inserted into the initial population to refine already successful chromosomes.

8.3 Test Results

In this section the results acquired from executing the test plans is presented.

8.3.1 Test 1 results

While the parameters may not yet be finely tuned, no less than 4 winning chromosomes were found. This is considered a great success, as it indicates that our chromosome encoding is sound. It appears to be able to capture enough building blocks of the Starcraft domain to encode a bot behavior capable of winning.

Figure 8.2 and Figure 8.3 shows the result for a population size of 50, while Figure 8.4 and Figure 8.5 illustrates the results gathered for a population size of 100.

It could appear that a population size of 50 is insufficient for covering the problem space effectively. Both Figure 8.2 and Figure 8.3 shows that the GA appears to converge at an optimum in less than 100 generations. Observing Figure 8.4 and Figure 8.5 it appears that having a population size of 100 does not suffer from the same problem - New better performing chromosomes are continuously found, even at the later stages of the tests. Of course, it is difficult to determine with much certainty if this is truly an optimum, or instead just an effect of a relatively small number of generations.
Worth noting, though, is that Figure 8.2 seems to have converged at a better performing optimum than any other test. This makes it difficult to make assumptions regarding the amount of building blocks contained in the population as the population in this test could, theoretically but unlikely, contain all building blocks. If the global optimum is indeed what was found in this test, the maximum fitness of the population would, of course, never increase regardless of the amount of remaining building blocks in the population.

8.3.2 Test 2 results

The results of test 2 looks promising. A total of 6 winning chromosomes were evolved, with the best chromosome having a fitness of 1.6504. The test was run for 223 generations, and appears to be close to converging as can be seen in Figure 8.6.

The results of test 2 indicates that it is possible to evolve strategies that can defeat the Nexus14 strategy, played by the opponent. However, more research is needed in order to confirm if genetic algorithms can be used to evolve commercial grade AIs.

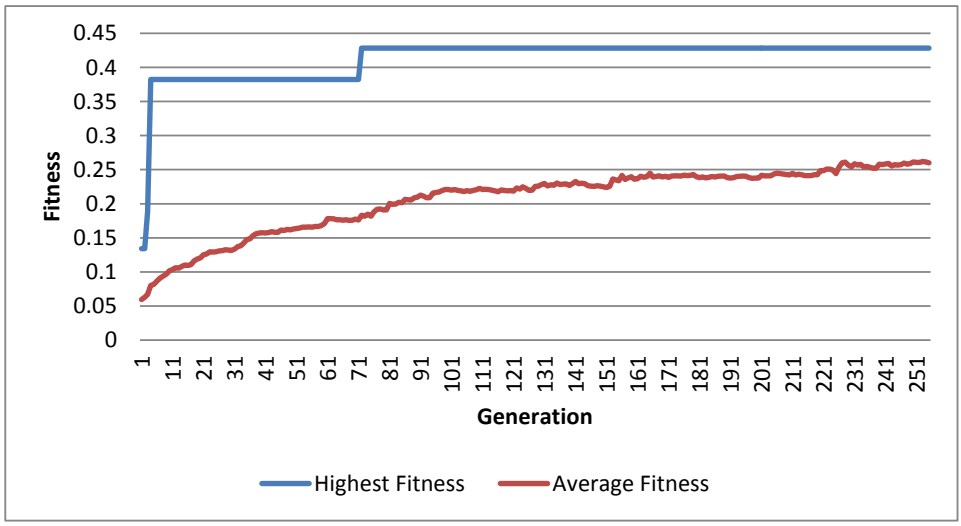

Figure 8.2: The results of Test 1, using a sample size of 10, winners size 2 and losers size 2. The population size is 50. The final average fitness for test 1 is 0.259100754 , and the fitness for the best chromosome in the population is 0.428488.

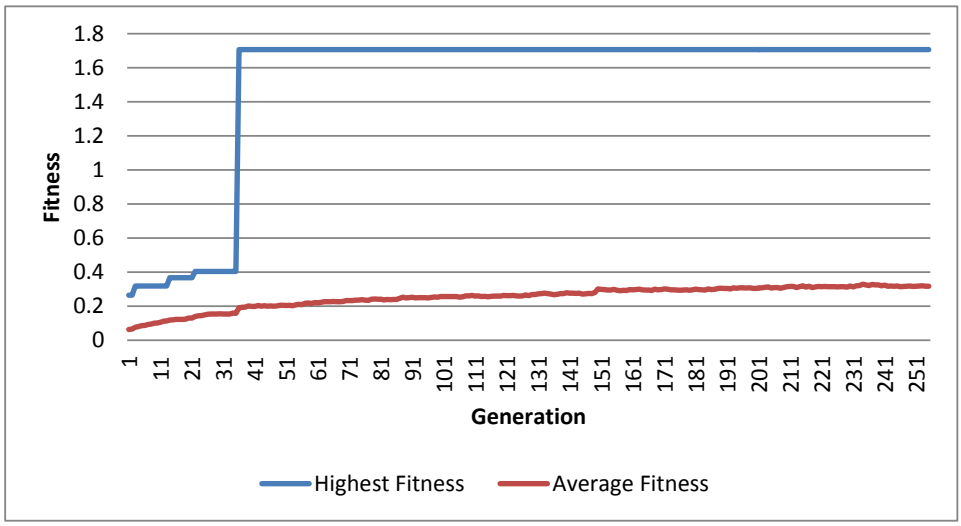

Figure 8.3: The results of Test 1, using a sample size of 20, winners size 5 and losers size 5. The population size is 50. The final average fitness for test 1 is 0.316675386, and the fitness for the best chromosome in the population is 1.70661.

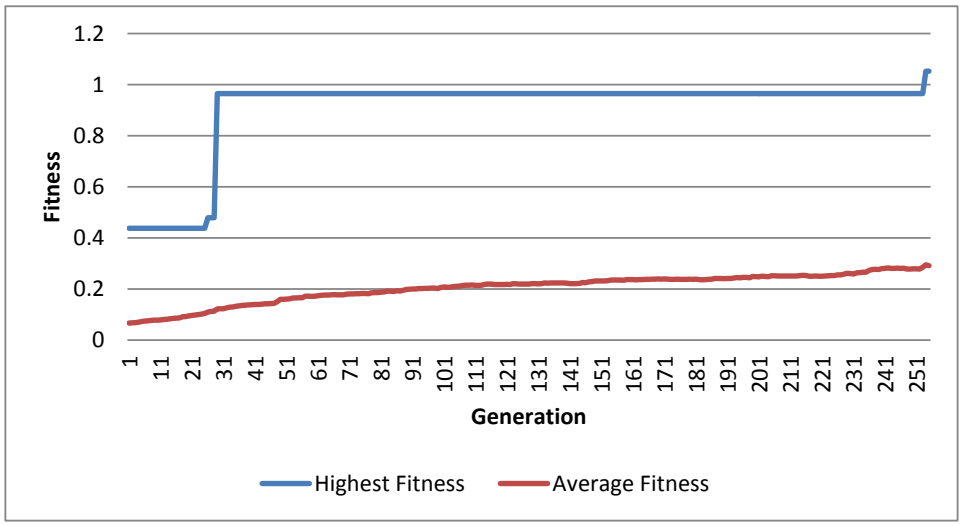

Figure 8.4: The results of Test 1, using a sample size of 10, winners size 2 and losers size 2. The population size is 50. The final average fitness for test 1 is 0.28967961, and the fitness for the best chromosome in the population is 1.05297.

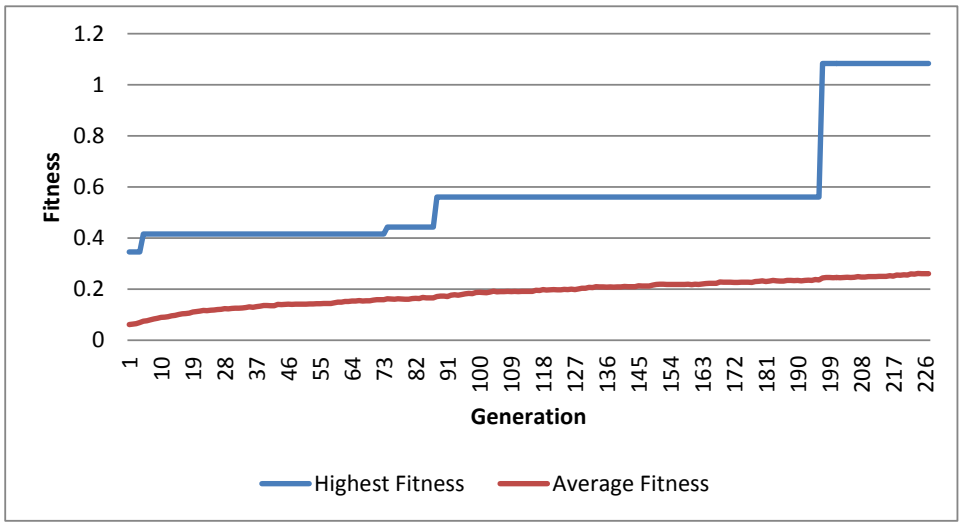

Figure 8.5: The results of Test 1, using a sample size of 20, winners size 5 and losers size 5. The population size is 100. The final average fitness for test 1 is 0.26040578, and the fitness for the best chromosome in the population is 1.08333.

The Winning Chromosomes

This section contains a qualitative analysis of the two strongest chromosomes. All referred listings in this section are the raw database entries which can be found in Appendix .4.

The first chromosome, Listing 1, appears to be heavily invested in producing units as quickly as possible, as all its buildings constructed can produce units (except for the Photon cannon, which is static defence). Listing 2 confirms this as a relative large number of early Zealots and Reaver units are made.

The second best performing chromosome shown in Listing 3 and Listing 4 rushes relatively quickly for a strong air unit, the Protoss Carrier (and less powerful, but cheaper, Scout). Once the Carrier is unlocked, the chromosome continues to build 2 more Stargates allowing for the construction of 3 Carriers simultaniously. The Nexus14 strategy is a strategy which goes for a quick economic expansion and has a very weak (if any) army early on. As such the chromosome can survive with just a few early Zealots while preparing to build higher technology units.

8.4 Conclusion

While many more experiments should be done to confirm the the observations made so far, the results are promising. The biggest success is the fact that the GA was able to produce winning chromosomes. Whether or not the winners are caused by clever GA parameter selection or not is difficult to say, but it does indicate that the chromosome encoding chosen is sound.
The GA was only evolved against a single strategy, but the results suggests that a GA may also be successful in winning against other strategies. Indeed, a very competitive bot may be created by evolving powerful chromosomes against the most popular strategies used by opponents. Of course this approach entails a classification problem in which the strategy of the opponent must be identified before a specific counter-strategy chromosome can be executed. This classification is left as future work.

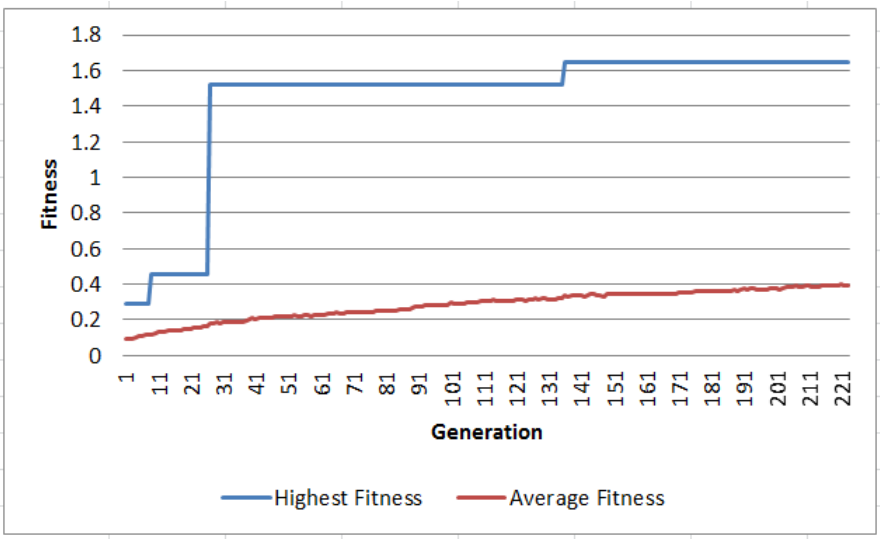

Figure 8.6: The results of test 2, using a sample size of 20, winners size 5 and losers size 5. The population size is 100. The final average fitness for test 2 is 0.3942, and the fitness for the best chromosome in the population is 1.6504.

CHAPTER 9

PERSPECTIVE

This chapter attempts to put the results of our tests from Chapter 8 into perspective, by discussing how this may be used in practice. During this project, the practical application of our thesis has always been the main motivation, thus it is important to document the vision which motivated taking this path for our thesis.

9.1 Our vision with Adaptive AI

When the topic Adaptive AI first came up, our vision was improve the game experience, from the point of view from the player, when playing against an AI.

We decided to focus on Genetic Algorithms (GAs) and Real-Time Strategy (RTS) games, and found that GAs can satisfy this vision in multiple ways. The way we imagined GAs could be used for improving game experience in RTS games, was that if we could create an AI that could always beat the player, we could forever provide a challenge to this player - at least until the player reaches a skill level equal to the global optimum in the game. The AI difficulty would then have to be scaled down to meet the level of the player, such that the player would not lose confidence from always loosing.

In a practical scenario, e.g. when developing an AI for a brand new RTS game, this usage of GAs require that the GA would have found the global optimum before the game is released. Because the GAs require players to play against, in order to evolve, that approach is unlikely. Instead the GA could be run over time, combining the results from every player playing against the AI, thus the GA would always attempt to follow, and beat, the trends of the player community, thereby adapting to the players. Our hope was that this adaption would make it more interesting for the player to play against an AI, and by that make the game more valuable.

9.2 Using our results

For obvious reasons the whole vision of Section 9.1 could not be covered in one thesis, as it raises so many different research questions. Instead we concentrated on developing an AI that utilizes a GA in the domain Starcraft, and by that tries to satisfy a small part of this vision. For this reason, the practical usage of our results is related to the efficiency of using a genetic algorithm as basis for an AI.

The tests of running different configurations of the genetic algorithm, offers promising results as shown in Chapter 8. The genetic algorithm successfully evolved chromosomes (strategies) that could win against the AI with a static strategy, and although the AI did not run for more than a week, the tendency of the fitness value was increasing for each configuration, which might mean that over time the win-rate of the genetic AI would also increase. Thus our results can be used to show that the usage of genetic algorithms, for evolving AIs that plays a commercial grade real-time strategy game, has potential.

Part V

Conclusion and Reflection

CHAPTER 10

CONCLUSION

Two major contributions were made during this project. Not only did we do an exploration of the feasibility of using genetic algorithms to evolve challenging AIs for commercial grade RTS games, we also produced a contribution to the The Brood War Application Programming Interface (BWAPI) community by releasing the source code of Cromartie. Cromartie is expected to replace Standard Add-on Library for BWAPI (BWSAL) as the de-facto standard for newly developed bots, as it provides a stable and modular implementation of frequently requested functionality such as building placement, scouting, micro management, database support, etc.

Starcraft has been used as our domain as it is a good representation of a commercial grade game, and is a frequently used platform for doing research. Two different approaches for evolving challenging AIs has been proposed. The first approach is to use a traditional genetic algorithm, an the second approach is to use an EDA based genetic algorithm. Only the first approach has been tested in this project, leaving the second approach to future work.

In order to test whether it is possible to use the GA to evolve a challenging AI, two tests where performed. One test with the purporse of determining the best parameters for the GA, and another test using the best parameters found in the previus test. The first test showed that a population size of 100, a sampleSize of 20 and a winner/loser size of 5 yielded the best results. However, due to the stochastic nature of the GA and the limited time available to run the tests, it is not certain that these settings are the best.
In the last test, the GA was able to evolve multiple winning chromosomes, indicating that the chromosome encoding used is reasonable - It is indeed able to represent a powerful strategy. The best chromosome scored a fitness value of 1.6504, which indicates the chromosome scored significantly higher than its opponent.

These results shows that it is possible to use a GA to evolve strategies that can beat an opponent playing the same static strategy used in the project. However additional research is required to determine if more flexible bots can be developed using the GA approach. Future work in this direction is covered in Chapter 11.
In addition to this, the area of EDA was explored as a promising alternative to GA. Unfor-

tunately a significant amount of development time would have to be dedicated to explore the effectiveness of using an EDA for evolving Starcraft strategies and as such this is left as future work (See Chapter 11).

CHAPTER 11

FUTURE WORK

In the beginning of and during this project, a lot of interesting topics and questions arose. We were not able to cover every one of these topics during our master thesis thus it is left for future work. This chapter describes these topics and where to go from here.

11.1 More genetic algorithm hypotheses

Early in this project, two additional hypotheses regarding genetic evolution of StarCraft strategies were considered. The first one considers whether it is possible to evolve a strategy that can win against multiple strategies, in contrast to evolving just against one single strategy. The second one is about the relationship between an evolved strategy against a single strategy and an evolved strategy against multiple strategies. The two additional genetic algorithm hypotheses are:

Hypothesis 2. *It is possible to use a genetic algorithm to evolve a Starcraft strategy, that can consistently win against three other strategies.*

Hypothesis 3. *The strategy evolved against a single other strategy has a higher win ratio than the strategy evolved against multiple strategies, when playing against the same single strategy.*

11.1.1 Testing the first hypothesis

The first hypothesis is tested similarly to Hypothesis 1 from Section 1.1, by repeatedly making the genetic algorithm AI play against three strategies, alternating between playing against each of them, on the same map in each game instance. The state of the genetic algorithm AI will be checked regularly, in order to monitor whether the win-rate or the average fitness of the population increases.

11.1.2 Testing the second hypothesis

For the purpose of this section, let the chromosome with the highest win-rate, evolved while testing Hypothesis 1 from Section 1.1, be referred to as EvolSingle and let the chromosome with the highest win-rate evolved while testing Hypothesis 2, be referred to as EvolMulti. There should be evolved a EvolSingle chromosome for each of the three AIs used to evolve the EvolMulti. The second additional hypothesis is then tested by letting each of the three EvolSingle chromosomes play a predefined number of games against the single AI that is used when evolving them. Then letting EvolMulti play three rounds of the same number of games against the three strategies. So if we defined the number of games to be 100, then the AIs will play games as follows:

Player 1	**...evolved against**	**Player 2**	**Number of games**
EvolSingle	Strategy 1	Strategy 1	100
EvolSingle	Strategy 2	Strategy 2	100
EvolSingle	Strategy 3	Strategy 3	100
EvolMulti	All Three	Strategy 1	100
EvolMulti	All Three	Strategy 2	100
EvolMulti	All Three	Strategy 3	100

If the average win-rate λ_S of the games played by each of the EvolSingle chromosomes, is higher than the average win-rate λ_M of the games played by EvolMulti against the three static strategies, the hypothesis is confirmed in this scenario. Formally this is defined by:

$$WL(x) = \begin{cases} 1 & x \textit{ is a winning game} \\ 0 & x \textit{ is a losing game} \end{cases} \tag{11.1}$$

$$\text{win-rate} = \frac{1}{n}\sum_{i=1}^{n} WL(i) \tag{11.2}$$

where $WL(x)$ is a win-lose function that returns 1 if the game x is won, and 0 if the game x is lost. The win-rate describes the fraction (or percentage) of winning games out of n games.

$$S_i = \textit{ win-rate for EvolSingle against strategy } i \tag{11.3}$$

$$M_i = \textit{ win-rate for EvolMulti against strategy } i \tag{11.4}$$

$$\lambda_S = \frac{1}{n}\sum_{i=1}^{n} S_i \tag{11.5}$$

$$\lambda_M = \frac{1}{n}\sum_{i=1}^{n} M_i \tag{11.6}$$

Hypothesis 3: $\lambda_M < \lambda_S$ (11.7)

where n is the number of strategies, and i correspond to a single strategy.

11.1.3 How to test

The setup and environment used for testing the hypothesis from Section 1.1 (See Section 8.1) can directly be applied to testing these two hypotheses. It is only a matter of configuring the tests, running the algorithm and waiting for a result.

11.2 Improve the genetic algorithm by using EDA

In Chapter 5 the theory about Estimation of Distribution Algorithms (EDAs) where covered. During this project, these techniques were not tested as our focus was with the standard genetic algorithm. By using these techniques though, it might be possible to improve how close to being optimal, an evolved strategy can get, while lowering the convergence time in the process.

One approach to improving our AI using EDAs would be to use the Bayesian Optimization Algorithm (BOA) introduced by Martin Pelikan et al[20]. This algorithm uses techniques for modelling multi-variate data using Bayesian networks, to create an estimate of a probability distribution of promising solutions. The challenge is then to incorporate this algorithm into our AI and into using the chromosomes encoding used for our genetic algorithm.

11.3 Player Modelling and Adaptive AI

This thesis project started by considering adaptive AI in general, which eventually led to investigating genetic algorithms. Before that an early topic was investigated, which considered how the AI's (evolved) strategies would be used in a real life scenario involving human players. This led to the following assumption:

Hypothesis 4. *It is possible to learn a model of a player, such that the strategy of that player can be consistently guessed correctly, before the game begins. This guess can then be used by the AI to select one strategy, that consistently wins against the guessed strategy.*

The reason for wanting to guess the strategy of the player, before starting the game, is that we want to know which strategy the AI should use as a counter, as early as possible.

11.3.1 Testing Hypothesis 4

In order to test this hypothesis, a few additional assumptions about the case is made. It is assumed that the player only knows the strategies that the genetic algorithm has evolved a counter strategy against. This is in order for the AI to select the appropriate evolved strategy, once the guess about the player strategy is made. We also assume that the player does not switch strategy during the game. This is in order to only being required to the guess of the player's strategy once per game, right before the game begins, and not while the game is running. This means that it is possible to build a player profile, solely from replay data, thus making it easier to test the hypothesis.

The guess about the opening strategy of the player is made based on the model of the player. Our thought on how to create this model, is to use a probability distribution, over the probability of the player selecting each of the strategies known by the AI. This player model could initially (before the player plays his first game) be built by performing data mining on the latest community replays and building the probability distribution on that. Then after each game played by the player, the probability distribution would be updated to reflect the strategy used in that game. This means that the player model would initially model the community, but over time (games played) converge towards being primarily a distribution of the strategies used by the player.

Another consideration is how the AI selects the counter strategy once the player strategy is guessed. One approach is to simply select the strategy evolved against the guessed strategy. Another approach would be to use the player model to store how well the player does against certain counter strategies, and use this information to select the counter strategy most likely to win, statistically.

Finally a player model could contain a list of tactics and a probability for each tactic, for each player. Against a player with a tendency to prefer a specific tactic, the bot could prepare an optimal defense ahead of time.

11.3.2 Difficulty scaling

This hypothesis considers how to get the AI to win as much as possible, but in order for an AI to be fun, it should be able for the player to win against it. Thus another future work to consider is how scale the difficulty of the AI, to suit the skill and experience of the player.

BIBLIOGRAPHY

[1] Steffan Bo Pallesen, Nikolaj Dam Larsen, Mikkel Graarup Jensen. *Labelling Starcraft Replays Using Cluster Analysis*. 2012 (see pp. 9, 13, 15, 16).

[2] Pieter Hubert Marie Spronck. *Adaptive game AI*. UPM, Universitaire Pers Maastricht, 2005 (see pp. 10, 17).

[3] Marc JV Ponsen, Héctor Mu~noz-Avila, Pieter Spronck, and David W Aha. "Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning". In: *Proceedings Of The National Conference On Artificial Intelligence*. Vol. 20. 3. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2005, p. 1535 (see pp. 10, 37, 42, 63).

[4] *Team Liquid Liquipedia: Protoss Build Orders*. Url: `http://wiki.teamliquid.net/starcraft/Category:Protoss_Build_Orders` (visited on 06/06/2013) (see pp. 11, 57).

[5] The Wargus Team. *Wargus*. Url: `http://wargus.sourceforge.net/index.shtml` (visited on 06/06/2013) (see pp. 15, 16).

[6] *BWAPI: An API for interacting with Starcraft: Broodwar (1.16.1)*. Url: `http://code.google.com/p/bwapi` (visited on 06/06/2013) (see pp. 15, 47).

[7] *BWAPI: Manual*. Url: `http://code.google.com/p/bwapi/wiki/BWAPIManual` (visited on 06/06/2013) (see p. 15).

[8] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper, and Eric Postma. "Adaptive game AI with dynamic scripting". In: *Machine Learning* 63.3 (2006), pp. 217–248 (see pp. 16, 41).

[9] *StarCraft AI Competition 2012*. Url: `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/media.shtml` (visited on 06/06/2013) (see pp. 17, 47).

[10] Andrew Smith. *Skynet: A Starcraft: Broodwar Bot using BWAPI*. Url: `http://code.google.com/p/skynetbot` (visited on 06/06/2013) (see pp. 17, 45, 47).

[11] Dave Churchill. *StarCraft AI Competition UAlbertaBot*. Url: `https://code.google.com/p/ualbertabot/` (visited on 06/06/2013) (see p. 17).

[12] Alberto Uriarte. *NOVA, a Starcraft bot*. Url: `http://nova.wolfwork.com/` (visited on 06/06/2013) (see p. 17).

[13] Jay Young, Fran Smith, Christopher Atkinson, Ken Poyner, and Tom Chothia. "SCAIL: An integrated Starcraft AI system". In: *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE. 2012, pp. 438–445 (see p. 18).

[14] Peter Bentley. "Generic evolutionary design of solid objects using a genetic algorithm". PhD thesis. 1996 (see p. 21).

[15] Michael LaLena. *Traveling Salesman Problem Using Genetic Algorithms*. Url: `http://www.lalena.com/AI/Tsp/` (visited on 06/06/2013) (see p. 21).

[16] David J Montana and Lawrence Davis. "Training feedforward neural networks using genetic algorithms". In: *Proceedings of the eleventh international joint conference on artificial Intelligence*. Vol. 1. San Mateo, CA. 1989, pp. 762–767 (see p. 21).

[17] Brad L Miller and David E Goldberg. "Genetic algorithms, selection schemes, and the varying effects of noise". In: *Evolutionary Computation* 4.2 (1996), pp. 113–131 (see p. 23).

[18] Pravir Chawdhry, Rajkumar Roy, and Raj Pant. *Soft computing in engineering design and manufacturing*. Springer Verlag, 1998 (see pp. 24, 26).

[19] Stanley Gotshall and Bart Rylander. "Optimal population size and the Genetic Algorithm". In: *Population* 100.400 (2008), p. 900 (see pp. 27, 63).

[20] Martin Pelikan. "Bayesian Optimization Algorithm". In: *Hierarchical Bayesian Optimization Algorithm*. Springer, 2005, pp. 31–48 (see pp. 28, 30, 34, 77).

[21] Heinz Mühlenbein. "The equation for response to selection and its use for prediction". In: *Evolutionary Computation* 5.3 (1997), pp. 303–346 (see p. 29).

[22] Gilbert Syswerda. "Simulated crossover in genetic algorithms". In: *foundations of Genetic Algorithms* 2 (1993), pp. 239–255 (see p. 34).

[23] Shumeet Baluja. *Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning*. Tech. rep. DTIC Document, 1994 (see p. 34).

[24] Georges R. Harik, Fernando G Lobo, and David E. Goldberg. "The compact genetic algorithm". In: *Evolutionary Computation, IEEE Transactions on* 3.4 (1999), pp. 287–297 (see p. 34).

[25] Heinz Mühlenbein, Jürgen Bendisch, and H-M Voigt. "From recombination of genes to the estimation of distributions II. Continuous parameters". In: *Parallel Problem Solving from Nature—PPSN IV*. Springer, 1996, pp. 188–197 (see p. 34).

[26] SK Shakya, JAW McCall, and DF Brown. "Updating the probability vector using MRF technique for a Univariate EDA". In: *Proceedings of the Second Starting AI Researchers' Symposium*. Vol. 109. 2004, pp. 15–25 (see p. 34).

[27] Jeremy S De Bonet, Ch L Isbell, Paul Viola, et al. "MIMIC: Finding optima by estimating probability densities". In: *Advances in neural information processing systems* (1997), pp. 424–430 (see p. 34).

[28] Shumeet Baluja and Scott Davies. “Fast probabilistic modeling for combinatorial optimization”. In: *Proceedings of the National Conference on Artificial Intelligence*. JOHN WILEY & SONS LTD. 1998, pp. 469–476 (see p. 34).

[29] Martin Pelikan and Heinz Mühlenbein. “The bivariate marginal distribution algorithm”. In: *Advances in Soft Computing*. Springer, 1999, pp. 521–535 (see p. 34).

[30] P Larranaga, R Etxeberria, JA Lozano, JM Pena, JM Pe, et al. “Optimization by learning and simulation of Bayesian and Gaussian networks”. In: (1999) (see p. 34).

[31] Heinz Muhlenbein and Thilo Mahnig. “The factorized distribution algorithm for additively decomposed functions”. In: *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*. Vol. 1. IEEE. 1999 (see p. 34).

[32] Georges Harik. “Linkage learning via probabilistic modeling in the ECGA”. In: *Urbana* 51.61 (1999), p. 801 (see p. 34).

[33] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press on Demand, 1996 (see p. 41).

[34] *The Terminator*. Url: `http://www.imdb.com/title/tt008824` (visited on 06/06/2013) (see p. 46).

[35] *Terminator: The Sarah Connor Chronicles*. Url: `http://www.imdb.com/title/tt0851851` (visited on 06/06/2013) (see p. 46).

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (see pp. 50, 51).

[37] Mike McShaffry and David Graham. *Game coding complete*. Charles River Media, 2009 (see p. 51).

[38] Don Clugston. *Member Function Pointers and the Fastest Possible C++ Delegates*. Url: `http://www.codeproject.com/Articles/7150/Member-Function-Pointers-and-the-Fastest-Possible` (visited on 06/06/2013) (see p. 51).

[39] *Team Liquid Liquipedia: Nexus14*. Url: `http://wiki.teamliquid.net/starcraft/14_Nexus_(vs._Terran)` (visited on 06/06/2013) (see p. 57).

Part VI

Appendix

.1 Trap data 1

```
10101 = 1
10011 = 1
11000 = 2
11001 = 1
11101 = 0
10100 = 2
11011 = 0
10110 = 1
11100 = 1
11010 = 1
10010 = 2
11110 = 0
10001 = 2
10111 = 0
11111 = 5
10000 = 3

Average = 1.375
```

.2 Trap data 2

```
01011 = 1
01001 = 2
00000 = 4
00001 = 3
00010 = 3
01111 = 0
01000 = 3
01101 = 1
00110 = 2
01100 = 2
00011 = 2
00111 = 1
01010 = 2
00100 = 3
01110 = 1
00101 = 2
```

```
Average = 2.0
```

.3 Search space

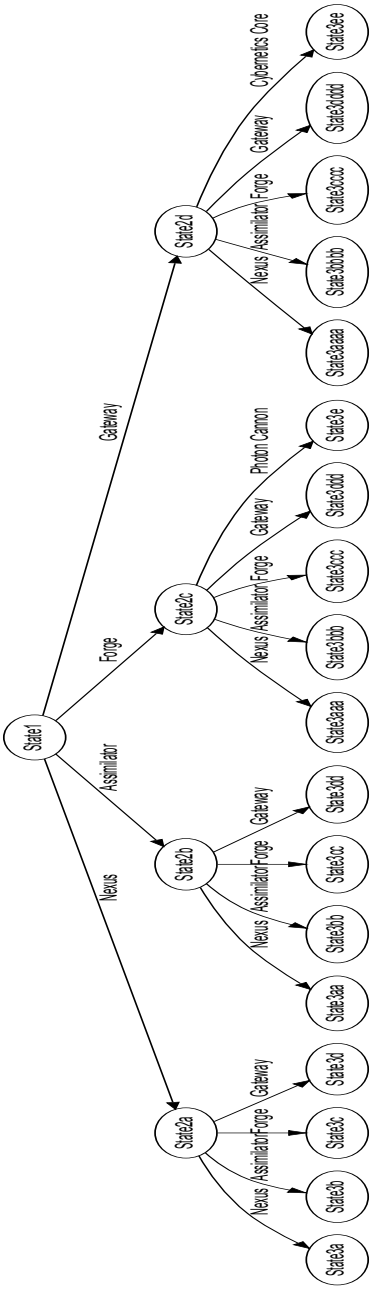

Figure 1: Search space tree for 3 states.

.4 Strong chromosomes data

```
ChromosomeID| ChromosomeFitness | Building name
-----------------------------------------------
68|16.504|Protoss Assimilator
68|16.504|Protoss Gateway
68|16.504|Protoss Cybernetics Core
68|16.504|Protoss Robotics Facility
68|16.504|Protoss Gateway
68|16.504|Protoss Forge
68|16.504|Protoss Robotics Facility
68|16.504|Protoss Stargate
68|16.504|Protoss Photon Cannon
68|16.504|Protoss Robotics Facility
```

Listing 1: Building gene order of strong chromosome 1

```
ChromosomeID| ChromosomeFitness | UnitName | UnitCount
-----------------------------------------------
68|16.504|Protoss Zealot|8
68|16.504|Protoss Zealot|4
68|16.504|Protoss Reaver|7
68|16.504|Protoss Corsair|9
68|16.504|Protoss Scout|4
68|16.504|Protoss Zealot|8
68|16.504|Protoss Shuttle|1
68|16.504|Protoss Shuttle|1
68|16.504|Protoss Corsair|4
```

Listing 2: Combat gene order of strong chromosome 1

```
ChromosomeID| ChromosomeFitness | Building name
-----------------------------------------------
14|15.193|Protoss Gateway
14|15.193|Protoss Cybernetics Core
14|15.193|Protoss Assimilator
14|15.193|Protoss Robotics Facility
14|15.193|Protoss Stargate
14|15.193|Protoss Nexus
14|15.193|Protoss Citadel of Adun
14|15.193|Protoss Fleet Beacon
14|15.193|Protoss Assimilator
14|15.193|Protoss Stargate
```

Listing 3: Building gene order of strong chromosome 2

```
ChromosomeID| ChromosomeFitness | UnitName | UnitCount
-----------------------------------------------
14|15.193|Protoss Zealot|1
14|15.193|Protoss Zealot|5
14|15.193|Protoss Carrier|3
14|15.193|Protoss Observer|7
14|15.193|Protoss Scout|7
```

```
14|15.193|Protoss Scout|9
14|15.193|Protoss Shuttle|1
14|15.193|Protoss Scout|9
14|15.193|Protoss Scout|4
```

Listing 4: Combat gene order of strong chromosome 2