NTNU
Norwegian University of
Science and Technology

Balancing Real-Time Strategy Games

Exploring the feasibility of using Artificial
Intelligence Techniques to evaluate Balance
in Real-Time Strategy Games

Arne Mæhlum

Master of Science in Computer Science
Submission date: February 2017
Supervisor: Anders Kofod-Petersen, IDI

Norwegian University of Science and Technology
Department of Computer Science

Arne Mæhlum

Balancing Real-Time Strategy Games

Master Project, Spring 2017

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

NTNU

Abstract

Of the most important things when creating a good Real-Time Strategy game (RTS) is ensuring that it is balanced. When a game is *imbalanced*, some strategies are given far more weight due to their disproportionate viability. Without a variation of strategies that are viable in different situations, the entire aspect of strategic planning starts to crumble.

In this paper I present a theoretical system capable of balancing an RTS, and take a small step towards the realization of such a system.

I have explored a coevolutionary approach to producing sets of viable, high-level strategies and counter-strategies for RTS games. These high-level strategies are often dubbed build orders, meaning the order in which units, structures and upgrades are produced as part of a bigger strategy. I have also explored how varying certain characteristics of a game affects the complexity of this task, and whether this approach can give insight to a game's level of balance.

The high-level strategies are evaluated through simulations in Decnalab, a minimal RTS game simulator developed specifically for this purpose.

Sammendrag

Blandt de viktigste problemene å løse når skal utvikle et godt sanntidsstrategispill, eller Real-Time Strategy game (RTS) som det heter på engelsk, er å sørge for at spillet er balansert. Når et spill er ubalansert vil noen strategier tillegges langt større vekt enn andre, fordi de er uproporsjonalt godt egnet. Uten et utvalg av strategier som er egnet til forskjellige situasjoner kan hele strategiaspektet falle bort.

I denne oppgaven presenterer jeg et teoretisk system som er istand til å balansere et RTS, og tar et lite skritt på veien mot å kunne virkeliggjøre et sånt system.

Jeg har utforsket en koevolusjonær tilnærming til å produsere sett av brukbare høynivå-strategier og mot-strategier for RTS-spill. Disse høynivå-strategiene kalles ofte "build orders", som betyr at de beskriver i hviklken rekkefølge man skal produsere forskjellige bygninger, "units" eller kontrollerbare tropper, og oppgraderinger som ledd i en større strategi. Jeg har også undersøkt hvordan denne oppgaven varierer i kompleksitet når man endrer på enkelte karakteristikker ved spillet, og hvorvidt denne tilnærmingen kan fortelle noe om hvor balansert spillet er.

Strategiene er evaluert gjennom simuleringer i Decnalab, en minimalistisk RTS-simulator utviklet for dette øyemed.

Acknowledgements

I would like to thank my supervisor, Anders Kofod-Petersen for listening to my ideas, for his invaluable feedback and for providing me a direction in my work.

I would also like to thank my girlfriend who has patiently motivated me during my work with this thesis project.

Arne Mæhlum
Oslo, March 12, 2017

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper concludes my thesis project, where my aim was to explore the feasibility of balancing RTS games using AI techniques. I decided to focus on a coevolutionary approach, in hopes that it could emulate the process of human players who must choose appropriate strategies, considering the strategies their opponents are likely to use.

1.1 Background and Motivation

When developing a new Real-Time Strategy game (RTS) one of the major tasks is making it balanced. Balancing is an iterative process of creating a configuration of the game, evaluating whether it is balanced, and proposing changes that can make the game more balanced. Evaluating balance of an RTS is typically done by letting human players play the game and identifying possible strategies. If there exists a strategy which is so good that no other strategies can defeat it, the game loses its main aspect which is choosing the right strategy for the given situation. When an RTS game is balanced, no such strategies exist.

A system that can explore and exploit the possibilities of an RTS configuration could reduce the time needed to evaluate its balance, and cut months or years of development time.

An RTS game poses several complex problems, all of which need to be solved in a sufficient way to in order to play it well. In this thesis I focus on the subproblem of finding build orders, which are plans for when and what to produce out of the available units, structures and upgrades that exist in the game.

1.2 Goals and Research Questions

The overall goal of this research is to take a small step in exploring the feasibility of a system for improving balance in RTS games.

With knowledge of which strategies people use in an RTS game it is possible to evaluate the game's balance. In theory, evaluating balance could be done as an automated process. Taking the most prominent strategies and the games configuration into account, one can make changes to the game in an attempt to make it more balanced. This too could, in theory, be done as an automated process. If there was an automated process for predicting which strategies people would use given the new configuration of the game, these three processes could be repeated until a more balanced configuration of the game is found.

I have focused on the problem of finding viable strategies and counter-strategies for an RTS game, in order to predict what kinds of strategies people would use.

Goal *Explore the feasibility of using coevolutionary algorithms to predict the set of viable strategies in RTS games*

Rather than requiring human players to play a large number of games to identify good strategies and counter-strategies, I will explore the feasibility identifying such strategies using a coevolutionary algorithm (CA).

Research question *Can a coevolutionary algorithm find a representative set of viable build orders and counter-build-orders for an RTS game?*

What I want to answer is how well the set of solutions found by a coevolutionary algorithm represents the set of viable build orders that exists in a game.

To answer the research question I will use the RTS game I developed for simulating games between computer controlled players, develop a coevolutionary algorithm, develop a method for evaluating viability of a set of strategies, and finally evaluate the results of this study.

1.3 Experimental Setup

For the experiments I have created and used a simple RTS game engine called Decnalab to evaluate the build orders generated by the coevolutionary algorithm.

In the experiments, the set of instructions provided to the computer controlled players consists of (1) a build order and (2) a timing for when to attack. The timing is defined as a minimum number of used supply, where used supply is a number related to how many live units the team currently has. These instructions are solutions generated by the coevolutionary algorithm.

The players do not have any behaviour for prioritizing which targets to attack or where to go depending on situation, but rather wait until they either see the enemy or reach their attack timing. When the latter occurs, they will *attack-move* their army units in a straight line towards the enemy base, meaning that the game engine will automatically switch a units current command from moving to attacking when it comes close enough to an enemy to attack.

Because the players only perform a subset of the tasks that a RTS game normally consists of, some simplifications of the game can be made without affecting the ability to evaluate the solutions.

1.4 Thesis Structure

This thesis is organized into the following six chapters

Chapter 1 This chapter explains the motivation behind the work that has gone into this thesis project, describes the goals of the research and briefly introduces the fundamental concepts.

Chapter 2 Chapter two provides an introduction to the theoretical concepts related to RTS games and Coevolutionary algorithms, and elaborates on the motivation behind the research. It also describes the structured literature review protocol for finding related work.

Chapter 3 In this chapter I present related work in the field of AI research in RTS games, with a primary focus on work related to build order generation.

Chapter 4 This chapter provides a further description of the RTS game engine Decnalab, its goals and features, and explains how it facilitates automated exploration of build orders. It also describes how the strategies played by the bots in Decnalab are defined and encoded, a set of handmade strategies used to facilitate learning and evaluation in the coevolutionary algorithm. Finally it presents the coevolutionary algorithm I have used and the method for evaluating the solutions it finds.

Chapter 5 The contents of this chapter includes a detailed description of the experiments that were conducted and presents the experiment results.

Chapter 6 In this chapter I interpret the results of the experiments, evaluate how they may answer the research questions, and propose possible improvements.

Chapter 7 In this chapter I will provide a short conclusion of my findings in this study, and provide some views on the possibilities of future work.

Chapter 2

Background Theory and Motivation

This chapter provides background theory, presents the structured literature review protocol, and details the motivation behind this thesis.

2.1 Background Theory

In this section we explain the different concepts needed to understand the work presented throughout thesis. They are separated into to main parts: RTS concepts and coevolutionary algorithms.

2.1.1 Concepts within RTS games

This subsection explains some key concepts which are particular to RTS games.

Resources

In an RTS game, part of the player's goal is to gather resources that can then be used to produce units, structures and upgrades. In addition to gathering resources, most RTS games have a concept of *supply*. One of the requirements of building units is to have enough available supply, and available supply can thus be though of as another resource. Available supply is typically increased by producing certain units or structures.

Units, Structures and Upgrades

Units and structures can be commanded to do different things in the game. Successful strategies involve producing unit and structures in order to gain an advantage, either by just having more than the opponent or by having specialized units that the opponent is unprepared to properly deal with. Some examples of traits that make units hard to deal with is high defense (health or armor), high attack damage, high maneuverability or even invisibility.

Technology tree

A technology tree, or tech tree, is a representation of the prerequisites for producing the various units, structures and upgrades that exist in the game. Defining prerequisites makes the game much more complex, as it creates multiple possible paths. This makes it more difficult to predict which path is ideal for a given situation.

Typically, at the beginning of the game only a few units and structures are available to the player. By producing new types of structures, upgrades and possibly even units, the player unlocks new parts of the tech tree, making other parts of the tech tree available.

Therefore, a way to change the dynamic of an RTS game, without adding or removing units etc., is to change the structure of the tech tree, potentially making e.g. a unit easier or harder to obtain.

Build Order

A build order is a plan for which units, structures and upgrades to produce in what order. A good build order should at some point(s) in the game maximize the players potential to do damage to the enemy, by outnumbering him or by advancing further through the tech tree.

Strategy

A strategy is a high-level plan for how to win the game.The build order is part of this strategy. A strategy also includes when and how to attack, which should be at points in the game when the potential for dealing damage is maximized with the build order that is chosen. A strategy can also include plans for how to deal with potential challenges presented by the opponent. For any strategy there should be counter-strategies which, if used, are likely to defeat it. Robust strategies are typically equipped to deal with most of the strategies it is likely to meet. Planning a build order so that the player can choose to adapt which part of the tech tree to advance through, without too much of a penalty, is an important part of creating a robust build order.

Balance

In a balanced RTS game the players have to constantly gather information about the opponent and decide on the on the best strategy for the situation. With a mistake from one player, the game can be tipped in the favor of the other player. In an unbalanced RTS game there are some strategies with disproportionate pobability of success. Players can then use those strategies to tip the game in their favor without considering the actions of the opponent. As this weakens the aspect of strategic planning in the, balance is important when creating a good RTS game.

2.1.2 Coevolutionary Algorithms

Coevolutionary algorithms is a class of genetic algorithms. Genetic algorithms search for solutions using darwinian evolution. Genetic algorithms maintain a population of individuals, where the genome of each individual represents a solution to the problem. The individuals fitness is a measure of the quality of the solutions it represents.

Through stochastic selection mechanisms, fit individuals are favoured for survival and reproduction. Since the selection mechanisms are stochastic, there is still a chance for less fit individuals to survive and reproduce. That can enable genetic algorithms to find solution which are difficult to search for using classical search techniques, because the best solutions may be just a small step away from a seemingly bad solution.

Coevolution is based on the idea that evolution of one population will drive the evolution of another. As exemplified by Rosin and Belew [1997] parasites and their hosts are in an evolutionary arms race, where the hosts evolve to resist the parasites and the parasites evolve to circumvent the resistance of the hosts.

A characteristic of coevolutionary algorithms that distinghish them from standard genetic algorithsm is that they can gradually improve by solving more and more difficult problems. By starting off easy and gradually increasing the challenge, the selection pressure is kept at a level which allows seemlingly poor solutions to evolve into good solutions that may otherwise be too hard to discover.

Some of the mechanisms that enables this to happen is a teach set, shared fitness, shared sampling, case-injection and hall-of-fame (Rosin and Belew [1997]).

For problems such as strategies in RTS games, where we only have one kind of individuals, a teach set can be used in place of a parasite population. The teach set represents a set of challenges for the main population to be evaluated against. The teach set it renewed on each generation by evaluating candidates against the old teach set. For performance, the teach set should ideally be a small population of fit yet highly diverse individuals. A good way to ensure that is through shared sampling, described below.

Before continuing to shared sampling, shared fitness is a mechanism where the fitness of each individual in a population is *not only* dependent on how well it solves a problem, but also on how well *the other* individuals solve the problems. Problems that the overall population struggles to solve are given more weight. To that fact, specialized individuals are more likely to survive and reproduce.

In shared sampling the individuals are also evaluated using shared fitness, but rather than applying a randomized selection method, the best addition to the population is iteratively selected. This ensures good diversity in the population. Shared sampling is typically used when selecting the next teach set.

A hall-of-fame is a list of individuals that have performed well in the past. They are archived in order to retain good solitions. When using a hall-of-fame in a genetic algorithm, it can be used as a source of candidates for the teach set, in addition to the main population.

Case-injection is the technique of taking known, good solutions and injecting them into some population. For coevolutionary algorithms, they are injected into either the main population, the hall-of-fame (on initialization), the teach set or a combination.

Because the teach set, which acts as an evaluation mechanism, varies from generation to generation, the fitness of the population may vary with the difficulty of the teach set, as well as with the quality of the solutions in the population. A set of *baseline solutions* can act as a fixed measurement for the population. When analyzing the statistical data, cross referencing with the evaluation aginst the baseline makes it possible to decide if fitness fluctuations stem from the teach set or the main population.

Ballinger et al. [2016] used these techniques for a coevolutionary algorithm and compared its performance to a hill-climber, a genetic algorithm and a brute force algorithm. They found that a coevolutionary algorithm produces the most robust build orders, robust meaning that they berform well against a wide range of opponents, but the coevolutoinary algorithm was not able to find the very best solutions.

2.2 Motivation

This section outlines the motivation behind the work that went into this thesis project. As a passionate RTS player, I know that making an RTS game balanced is a both necessary and difficult task.

My initial focus going into the structured litterature review was on finding any work exploring AI techniques for evaluating balance in RTS games.

Sinces that material was relatively scarce, I expanded to search for applications of various AI techniques in RTS games in the hopes that some of it would

be relevant to balancing, regardless of the aim of the authors. After guidance from my supervisor I decided to focus on work related to developing build orders.

Planning the build order is the most high-level part of a strategy, and determines what units and unit compositions that will be available at different stages in the game. Evaluating different compositions units against eachother can be interesting, but without calculating when they may be obtainable at throughout the course of a game, this cannot give any indication of the level of balance.

Admittably, to properly evaluate balance one needs not only to find viable build orders, but also to use the units that are produced in the best way possible, both tactically and with good micro management. Previous work typically focuses on only one of these problems, so I was forced to focus on just one of them. I decided to start at the top, with developing buld orders.

Most of the previous work has focused on separating the evaluation of build order into two parts: (1) generating build orders in one system and (2) evaluating them in another, such as Ballinger et al. [2016]. This makes it impossible to calculate loss of structures and workers, as only army units take part in the evaluation step. Losing structures and workers can have a big impact on how the build order is executed. For that reason I attempted to create an RTS game engine that could support both build order execution and battles at the same time.

When evaluating balance it is important not only to find some good build orders, but to find a representative set of build orders that could be viable against eachother. In StarCraft, the set of currently popular strategies, known as the *meta*, is continuously changing as people discover new strategies. This development is also driven by changes made to the game in attempts to keep it balanced. Knowing the current meta is important for understanding which strategies are currently viable. In the same way, an AI approach should be able to keep a current meta, where the viability of strategies change as the meta changes.

For this reason I believe coevolutionary algorithms to be a good approach, since it allows a solution, in this a bulid order, to be evaluated with respect to the solutionas it coexists with.

2.3 Structured Literature Review Protocol

To identify related work I have done a structured literature review. Search engines used: *Springer, ScienceDirect, IEE EXplore, Wiley, ACM, ISI Web of Knowledge.*

2.3.1 Specifying research questions

In order to define the SLR research questions, first we need to recognize which problem we want to solve, relevant ways of solving it, and relevant types of solutions.

Original Problem *Using AI to play an RTS and discover ways to play it.*

Our original problem is both broad and extremely complex, so it must be narrowed down to something manageable.

Although making AI for RTS can be thought of as a single problem, and some holistic approaches have been attempted, we choose to look at it as a set of problems, where by combining solutions one may be able to create a fully functional AI, with capabilities comparable to those of a human player.

Problem (P) *Any problem related to AI in RTS*

Since we can only solve a small part of our original problem, we needed to choose which problem(s) to solve based on what we learned through the SLR.

Constraints, methods and/or approaches (C) Any of: dynamic scripting, evolutionary algorithms (or more generally genetic algorithms), artificial neural networks, flocking, swarm intelligence or particle swarm optimization.

The AI techniques deemed relevant were chosen iteratively while developing a review protocol and conducting the review. This may be contrary to how an SLR is supposed to be conducted, but in the initial phase we did not have sufficient knowledge on which problems – and adjoining approaches – existed within P.

Solution (S) *AI system capable of managing some task being part of playing an RTS, or a system producing such an AI.*

Relevant types of solutions do not only include AI systems used during during a game of RTS, but potentially system used to develop such AI systems. To illustrate the difference, a person which is good at football solves the problem of playing football, while a person who is a good football trainer solves the problem of developing solutions to the problem.

Following the identification of P, C and S, we should be able to formulate our research questions.

RQ1 *What are the existing solutions to problems related to AI in RTS?*

RQ2 *In the existing solutions discovered, what methods are used to handle different problems related to AI in RTS?*

RQ3 *What is the strength of the evidence in support of the different solutions?*

RQ4 *What implications will these findings have when developing AI(s) for RTS games?*

Chapter 3

Related Work

In this chapter we will introduce the most relevant related work, mostly found in the structured literature review.

3.1 Build Order Generation

Garćıa-Sánchez et al. [2015] describes the problem of AI in RTS games as being separated into two levels: a strategic level for overall strategy, and a tactical level for controlling subsets of units. In their system, they use a bot based on the open-source *OpprimoBot*, and they use OpprimoBot's default behaviour to handle what they call the tactical level. They used Genetic Programming, a sub-category of Evolutionary Algorithms, to generate strategies for the modified OpprimoBot to carry out. The fitness of each strategy was decided based on how well the bot did against a 'properly selected opponent', taking into account the player score at the end of the game (provided by StarCraft) and a game report.

3.2 Unit grouping and cooperation

In Preuss et al. [2010] address the issue that RTS AIs are traditionally bad at making their units fight well together. This problem is twofold, it involves (1) distributing the available units among coinciding encounters with opponent(s), and (2) organizing the units fighting together in an encounter to produce the best outcome. For the first problem they use a learning self organized map (SOM) to distribute units, while using an Evolutionary Algorithm (EA) to adapt it to the game. For the second problem they use flocking and influence map-based path finding to organize the units amongst themselves.

Preuss et al. [2010] is based on the work from Danielsiek et al. [2008] where Danielsiek use flocking to make units fight more effectively together. They also show how using Influence Maps (IM) can mend cases where flocking alone fails by getting the units trapped.

Preuss et al. [2010] also derives from the article of Beume Beume et al. [2008] where they use self-organizing maps – trained *offline* by an EA – to distribute available units into appropriate groups.

Ballinger Ballinger et al. [2016] developed a coevolutionary approach for finding build orders. They compared build orders created by three different techniques: coevolutionary algorithm, genetic algorithm, and hill-climber to exhaustive search. The generated build orders are evaluated with respect to strength and robustness.

They consider a strategy as strong if it defeats opponents quickly, and robust if it is able to defeat many different opponents. Hence, a more robust build order has a higher chance of winning against a randomly drawn opponent.

They found that 'Genetic algorithms find the strongest build orders, while coevolution produces more robust build orders than a genetic algorithm or hill-climber'.

In order to properly evaluate build orders, they argue that the search space must be limited so that it can be searched exhaustively, so that each generated build orders can be evaluated against all possible build orders. They chose to limit the build orders to maximum five actions.

Search space: They build on their previous work, where the generated build orders were encoded as lists of actions. In this paper they use lists with branches and iterative loops. Logical conditions control which branch of the build order to take, as well as when to repeat the loops.

Environment:

They developed a new RTS game called WaterCraft, modeled after StarCraft and designed to be better suited for evolutionary approaches, specifically in terms of simulation speed. Build in to the game itself is a scoring algorithm, that rewards producing units and structures, and killing/destroying enemy units and structures. This score is used when calculating the score of a build order against another.

WaterCarft receives build orders to carry out. For each action in the build order, any dependencies are carried out first (e.g. you need a Barracks to produce a marine).

Speeding up evaluation:

To further speed up evaluation, which takes on average 20 seconds in WaterCraft, they use Build-Order Simulation Software (BOSS), that they created, which 'takes a minimalist approach to evaluating build orders'.

They combine BOSS and SparCraft, using BOSS to simulate build orders and

SparCraft to simulate battles whenever a player intends to attack.

With this techinque, evaluating a build order only takes around one tenth of a second.

Simulation and evaluation:

In their simulation there is one worker unit, the SCV, and three army units, Marine, Firebat and Vulture (from weakest and most easily obtainable to strongest and most advanced).

Park Park et al. [2008] improves a scripted AI for the game *Age of Mythology: The Titans*. This shows up in the search because they work on an AI, but they do not employ any AI techniques to do so, just scripting. It is therefor not if interest.

Chapter 4

Architecture/Model

This chapter presents the software developed and used in the research, and the method for evaluating successfulness of a search for viable strategies.

4.1 Decnalab – an RTS Game for AI reserach

To simulate the execution of simple strategies, I developed Decnalab, an RTS game engine written in JavaScript, designed for use in AI research. Through NodeJS it is runnable on most Operating Systems, and it can be downloaded and installed in a matter of minutes. In addition to being easily available, Decnalab aims to be deliver at a performance level that is comparable to similar systems.

The runnable source code of Decnalab is available at `https://github.com/arnemahl/decnalab`.

4.1.1 Features

As a measure to boost performance during simulations, Decnalab calculates the timing of future events and fast forwards to the next event that will occur. All events essentially result from commands given by the computer controlled players. For most commands the players are only interested in changes that occur when the command is fully completed. By only progressing commands at the time step (tick) they complete, the game engine only need to do a few operations each tick. However, the most significant performance boost comes from skipping past ticks where no events occur.

The only types of command whose progress needs to be updated on each tick are the *move* and *attack-move* commands. Updating the positions of moving

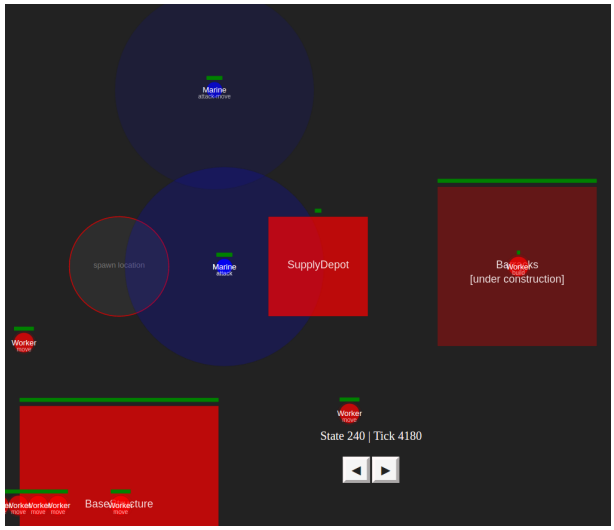

Figure 4.1: Screenshot from Decnalab

units on each eventful tick enables army units to attack other units when they come into range.

In StarCraft Brood War, which via BWAPI has previously been used for AI research, it is possible to speed up games by skipping most of the frames that are rendered. Since Decnalab is developed primarily for simulations involving computer controlled players (bots) rather than humans, the simulation process is completely decoupled from the graphichal presentation process.

Simulations in Decnalab are deterministic, so that if both players do exactly the same in two different games, the result will also be the same. This is useful, and though not uncommon in games designed for the purpose of AI research, RTS games designed for humans to play typically have some randomness.

The way Decnalab enables analysing games between bots with specific instructions, is by first simulating a new game and storing each state throughout the game. Since the bots have specific instructions and Decnalab is deterministic, the game will play out the same way it did previously. The NodeJS server that simulated the game then serves the game states to a simple front-end script that runs in a web browser, visualizing what happened. By animating moving units between their current and next position, the user also gets a visual representation of the units theorectical position at all times, and can easily track the unit movements.

In most RTS games each player gains vision in a radius around each of it's

units and structures. Taking each unit and structure on both teams and calculating which ones of the enemys units and structures are in vision has a runtime complexity of $\mathcal{O}(N * M)$, where N and M is the number of units and structures for the respective teams. Instead, Decnalab only divides the map into sectors, and when a player has a unit or structure in the sector they gain vision of any enemy units or structures also in that sector. This only has a runtime complexity of $\mathcal{O}((N + M) * S)$ where and S is the number of sectors.

Lastly, collision detection is completely unsupported in Decnalab. Since units can collide with units and structures on the same team, collision detection has a runtime complexity of $\mathcal{O}((N + M)^2)$, where N and M are the total number of unit and structures for the respective teams. However, for experiments that do require collision detection it can be implemented in the game engine with relatively little effort.

Units, Structures and Upgrades

Because Decnalab is written in JavaScript rather than in a compiled library, the set of available units, structures and upgrades, which can be though of as the *game configuration*, can easily be modified. This makes it easy to experiment with different game configurations and evaluate their level of balance.

4.2 Bots and Strategies

For all of the experiments I used a coevolutionary algorithm to generate strategies executed by the computer controlled players (bots) in Decnalab.

A strategy contains (1) a *build order* and (2) an *attack timing* defined as a number of used supply. Before this *attack timing* the player will leave the army units where they are, unless the enemy comes into vision. When having enough supply to attack the player will *attack-move* all army units toward the enemy base.

The player starts with five *Workers* and one *BaseStructure*, and zero of all other units and structures. The *build order* is defined as a sequence of types of units and structures, and how many of these to add beyond the previous count. A simple 3-step build order is shown in figure 4.2. The first target in the build order is to add one more *Barracks*. When the player has enough resources to start the barracks it will continue to the next target, which is to add one *Supply Depot*. Lastly, the player will produce a *Marine*, and since that is the last item in the build order the player will continue to produce more marines whenever it has enough resources and a structure to produce them from, in this case the one barracks.

```
buildOrder: [
    {
        specName: 'Barracks',
        addCount: 1
    },
    {
        specName: 'SupplyDepot',
        addCount: 1
    },
    {
        specName: 'Marine',
        addCount: 1
    }
]
```

Figure 4.2: A three-step build order

4.3 Handmade strategies

In this section I present the handmade strategies that were used in most runs of the CA. These handmade strategies were usually used both as baseline solutions and as case-injected solutions.

The set of handmade solutions are designed so that all of them win against at least one of the others, and they are designed to range from very aggressive to very economical.

More aggressive strategies focus more of the resources early in the game towards producing an army, while more economical strategies focus more of the early game resources towards producing more workers. To illustrate, if a player using an aggressive strategy fails to capitalize on the advantage in army size early in the game, the opponent using a more economical strategy eventually gets a higher production capacity and in turn a larger army. However, if the economical strategy of the opponent produces too few army units in order to defend, the aggressive player will win before the economical advantage comes into play.

The figure showing each strategy is labeled with an id, such as B-01. The B stands for baseline, because these handmade strategies are used as the set of baselines. When evaluating the final set of solutions from the CA, the baselines are assigned an id with "B-" and the index, while the solutions are assigned a value with "S-" and an index.

Id/Id	B-00	B-01	B-02	B-03
B-00	tie	loss	win	win
B-01	win	tie	loss	loss
B-02	loss	win	tie	loss
B-03	loss	win	win	tie

Table 4.1: Game results for the baselines (row) when playing against other baselines (column)

Table 4.1 shows which strategy will win in a game between two baselines. The result, win, tie or loss, is the result of the baseline whose id is displayed on the left column, when played against he opponent whose id is displayed on the topmost row.

Each strategy beats the strategy which is slightly less economical. This is in part because they successfully utilize the defenders advantage. The defenders advantage applies when a battle is happening closer to one team's base, where their units are produced. Having to move the units a shorter distance before they come into battle, and thus for a shorter time, means they can begin to be produced later.

B-00, the most offensive handmade strategy, shown in figure 4.3, does not produce a single worker. It focuses solely on making marines from a single barracks, and having only the five workers it starts with that is pretty much all it can afford.

With an *attackAtSupply* of 6, this strategy involves sending the marines to attack as soon as they are produced. This increases the chances of overpowering a more economically oriented opponent before they get an army. However, faced with an aggressive opponent this strategy would benefit from waiting a little bit longer, grouping up more units before launching an attack.

```
{
    "attackAtSupply": 6,
    "buildOrder": [
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "Marine" }
    ]
}
```

Figure 4.3: B-00, the most offensive handmade strategy

B-01, shown in figure 4.4 produces three workers before focusing all resources towards making an army. Because all three workers are produced at the beginning of the game, this strategy quickly provides an income-boost that provides an advantage against less economical strategies. This strategy, B-01, beats B-00 because it, just barely, manages to get marines out to defend before the B-00 can kill any units or destroy any structures. Eventually the economical advantage allows B-01 to get ahead and win the game.

```
{
    "attackAtSupply": 20,
    "buildOrder": [
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "Marine" }
    ]
}
```

Figure 4.4: B-01, a quite offensive handmade strategey

B-02, shown in figure 4.5, is a much more complex strategy, with a build order length of 25 as opposed to 8 in the two first ones. It produces seven workers, stopping at 12 when counting the five it starts with.

This strategy runs a higher risk of being attacked before it has produced enough marines to defend (or any at all). If the game goes on long enough, it will eventually have a massive production capacity, with three barracks and enough income to produce from them almost continuously.

B-02 beats B-01 after getting an economical advantage, while it loses to B-00 because it does not begin to produce marines early enough to defend.

```
{
    "attackAtSupply": 30,
    "buildOrder": [
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "Marine" }
    ]
}
```

Figure 4.5: B-02, a very economical handmade strategy

B-03, the most economical of the handmade strategies, goes up to a total of 15 workers before even preparing to produce an army. At that point it has such a large income that it can produce four structures at a time, after which it suddenly has a formidable capacity to produce marines.

B-03 is able to beat B-01 and B-02 because it gets an economical advantage, while at the same time it manages to produce enough marines to defend their first attack. The reason for this is that B-03 has the defenders advantage in the first battle.

B-03 loses to B-00, because the latter arrives with the first marines long before the first barracks of B-03 is even started. B-00 then starts to kill B-03's workers and structures, which prevents B-03 from advancing through the build order. B-03 eventually loses without having managed to produce a single marine.

```
{
    "attackAtSupply": 35,
    "buildOrder": [
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Worker" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "SupplyDepot" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Marine" },
        { "addCount": 1, "specName": "Barracks" },
        { "addCount": 1, "specName": "Marine" }
    ]
}
```

Figure 4.6: B-03, an extremely economical handmade strategy

4.4 Coevolutionary algorithm

The coevolutionary algorithm (CA) that I used was inspired by the one used by Ballinger et al. [2016]. Here I will present my CA implementation, the configuration parameters it uses, the values I chose to use, and why I utilized those parameters and values.

4.4.1 Implementation

A simplified version of the CA I implemented is shown in algorithm 1. It does not include the tracking of statistics etc.

Algorithm 1 Coevolutionary algorithm

1: *hallOfFame* ← *caseInjected* solutions
2: *population* ← generate *nofChildrenPerGeneration* random individuals
3: *teachSet* ← use shared sampling to select *teachSetSize*/2 from each of *hallOfFame* and *population*
4: Evaluate *shared fitness* of *population* against *teachSet*
5: *population* ← select *popSize* unique individuals from *population* using *scaledFitnessSelection*
6: **while** *currentGeneration* <= *maxGenerations* **do**
7: *parents* ← select *nofChildrenPerGeneration* from population
8: *children* ← pair the *parents* and let each pair produce a pair of children
9: Evaluate *shared fitness* of *children* against *teachSet*
10: *teachSet* ← use shared sampling to select *teachSetSize*/2 from each of *hallOfFame* and *children*
11: *population* ← select *popSize* unique individuals from children using *scaledFitnessSelection*
12: *hallOfFame* ← use shared sampling to select one individual from *population*, add to *hallOfFame*

When initializing, the algorithm takes the case injected solutions and adds all of them to the hall of fame. It then generates random individuals and adds them to the population. To initialize the teach set, it selects up to four of the *first* individuals from the hall of fame, then applies shared sampling to select the next four from the population. (Here, I use "four" in place of $teachSetSize/2$.)

Later on, the teach set is updated by using shared sampling to select both from the hall of fame and the population. We then use the previous teach set to evaluate the individuals, and shared sampling involves repeatedly picking the one with highest shared fitness.

At the end of each generation, the best individual from the population is added to the hall of fame. We select individuals from the hall of fame, as well as the population, to create the teach set. Thus, solutions from each generation will be rewarded for performing well against both the hall of fame and the current population. This prevents the CA from losing the selection pressure and thus the solutions that perform well against past solutions.

Because the hall of fame also contains the case injected solutions, the case injected solutions are likely to impact the selection of solutions. This should

drive the CA to produce solutions that perform well against the case injected solutions.

A quirk in my CA implementation is that I begin by creating *nofChildrenPerGeneration* individuals in the population, and then remove some of them to be left with a population of size *popSize*. The reasoning behind this is that it increases the chances of selecting good individuals for the teach set. For the same reason, for each generation, the teach set is selected from the newly created children, before removing some of them to be left with *popSize* solutions for the next generation.

4.4.2 Available parameters

An example showing the full configuration used for a run of the CA is shown in table 5.1. In addition to the values shown in the table, the set of case injected solutions have an impact on the selection of solutions in the CA, and the set of baselines have an impact on how the performance of the CA is tracked and evaluated.

The population size, or *popSize* seen in the table, is the size of the population at the start of each generation. The number of children to be created per generation, *nofChildrenPerGeneration*, decides how many parents to select for reproduction, each couple of parents producing two children. If the number of children created exceeds the specified population size, a subset of the children are selected to survive and become the population for the next generation.

The size of the teach set is decided by *teachSetSize*. Every individual generated, either initially by random generation or subsequently by crossover and mutation, have to play against all the members of the current teach set. Thus, this parameter greatly impacts the run-time of the CA.

Two other parameters with influence on the CA run-time are *maxLoopsPerGame* and to some extent *maxTicksPerGame*. The former of the two defines how many loops the game in Decnalab can simulate before calling a tie. The latter is similar, but sets a limit to ticks, rather than loops. Remember that for each loop, Decnalab skips to the next eventful tick, so it typically skips some ticks per loop.

The fitness scaling factor (*fitnessScalingFactor*) is used to define how much fitness is scaled. The fitness of an individual is scaled by the function

$$f_{scaled}(fitness, maxFitness) = maxFitness + fitness * fitnessScalingFactor \quad (4.1)$$

where *fitness* is the fitness of the individual and *maxFitness* is the fitness of the fittest individual in the concurrent population. Thus, a fitness scaling factor of

0 causes all individuals to have the same fitness. As the fitness scaling factor approaches infinity, the scaling effect will approach none.

The individuals are selected using scaled fitness selection, which is identical to roulette wheel selection apart from the fact that it applies fitness scaling. Thus, a fitness scaling factor of 1.5 means that an individual with $fitness = maxFitness$ has a 60% bigger allocated section of the roulette wheel than an individual with $fitness = 0$, and thus a 60% higher probability of being selected.

Crossover is controlled by three parameters. *crossoverRatio* defines the probability that a pair of parents will produce offspring through crossover, to which the alternative is to simply clone the parents.

The type of crossover it specified through the parameter *cossoverFunction*. The first alternative I have implemented here include *bitwiseUniformCrossover* which is a standard, uniform crossover that allows crossover points to be placed anywhere in the genome. The other alternative is to use *sequenceWiseUniformCrossover*, which prevents crossover points from being placed in the middle of a *specName* in a target in a build order. To explain the difference, here is an example.

Using bit-wise crossover, if, at the same point in the build order, one parent contains a Marine, encoded as "01", the other contains a SupplyDepot, encoded as "10", and a crossover point is placed in the middle of these *specName*s, one child would inherit Worker, encoded as "00", and the other would inherit Barracks, encoded as "11". Using sequence-wise crossover a crossover point could not be placed in the middle of these *specName*s, and the children would be guaranteed to inherit Marine and SupplyDepot.

Choosing between these two crossover functions it is possible to control the CA's degree of exploration vs. exploitation. The bit-wise crossover is more explorative and less exploitative than the sequence-wise crossover.

Lastly for crossover, *crossoverPointRatio* defines the ratio of crossover points. When using a value of 0.5, on average, equally many crossover points will be selected as it would in standard *bitwiseUniformCrossover* if this parameter didn't exist. That is because, for each bit there is a 50% chance of selecting a bit from the same parent as the previous bit, and a 50% chance of selecting a bit from the other parent. This is equvalent to a 50% chance of having a crossover point being placed between the previous and the current bit.

Using a ratio for crossover points that is closer to zero, the average number of crossover points will be reduced. As with crossover function, the ratio of crossover points enables controlling the degree of exploration vs. exploitation.

Mutation is controlled through *perBitMutationRatio*, which is the probability for each bit in the genome to flip it, i.e. 0 into 1 or 1 into 0.

Some parameters control the range of solutions that the CA can search through. These include *initialBuildOrderLength* which controls the length of the build order, *producableThings* which control which units or structure the build orders may contain and *possibleAddCounts* which controls the range of possible "add counts" each target in the build order may have. An add count of one means that the target is to produce one additional of whichever thing should be produced. Finally, the range of possible attack timings in the strategy is controlled by *minAttackTiming* and maxAttackTiming.

The remaining parameter for the CA only affects the way its found solutions are evaluated. *nofBestFoundSolutionsToselect* is used when selecting the best found solutions and evaluating the CA performance, as described in section 4.5.

4.4.3 Selecting parameter values

My starting point was to take inspiration from Ballinger et al. [2016], and I ended up using the same values for the CA parameters they described.

They did not mention how many children they produced per generation, so I assume they only produced how many they needed to fill the population. I decided to produce a few more, as that would increase the probability per generation of finding good solutions, at the cost of run-time. The value I ended up using was 70 children per generation.

Controlling exploration

Initially, my CA encoded genomes as JSON-objects, but at some point I changed the encoding into strings that should simulate bits. The reason behind this was that the the genetic distance within the population would fall dramatically towards the end of the CA run, and there would be only few unique solutions, i.e. the population would converge towards a very small set of solutions.

Using a genome encoded as bits allowed for a much bigger range of changes to occur during reproduction, both regarding crossover and mutation. It greatly reduced the convergence, and caused many more unique solutions to be found.

However, I saw that using bit-genomes caused the quality of the solutions in the CA to fluctuate a great deal more than it had when using JSON-genome. For this reason I added the sequence-wise crossover and the crossover-point-ratio, so that I could reduce the degree of exploration and thus the fluctuation in solution quality. While this was not part of my structured experiments, I decided on using a crossover-point-ratio of 0.25.

Solutions space

After creating the handmade strategies presented in section 4.3, I discovered the need for changing the values for some of the CA parameters. Firstly, I found it was not possible to encode the two most economical strategies with less than 24/25 targets in the build order, given "add-counts" of maximum 1. Increasing the maximum add-count to 7 it was possible to encode the same build orders with eight and nine targets.

I attempted to make the CA find these economical build orders in both ways (1) build order length of 26 and add-count of 0 or 1, and (2) build order length of 9 and add-count of [0, 7]. In both cases, the CA was unable to find the strategies any equally economical strategies, but it found better strategies when using a longer build order and a smaller possible range of add-counts.

With the previous parameters the CA was also unable to consistently get a good score when evaluating the found solutions. I therefore decided to reduce the build order length and thus the search space. It would no longer be possible to find the most economical build orders, but perhaps this could increase the probability of finding good solutions.

Choosing limit of game duration

Another insight I got after creating the handmade build orders, is that a higher number of workers causes the number of skipped ticks per loop to decrease, as more workers means more eventful ticks. Thus, games with more economical strategies would last for a lower number of ticks, or shorter in-game time if you will. In order to give the economical build orders a fair chance of defeating their opponents I doubled the limit for maximum number of loops per game to 2000, and added a limit for maximum number of ticks per game at 20000. In games between two strategies that produce few things the tick limit will be reached be fore the loop limit.

4.5 Method for evaluating sets of found solutions

In order to evaluate the performance of a search method for a given problem, one must first decide on a method of evaluation. In context of a clearly defined evaluation method, it is possible to determine the successfulness of a single run of the search method. Through repeated runs of different search methods – or different configurations of the same search method – it is also possible to argue about the methods' ability to produce good solutions to the problem in context of the evaluation method.

Ballinger et al. [2016] did an exhaustive search to find the best build orders with a maximum build order length of 8. They could then use those build orders as a comparison when evaluating solutions found by different search methods. As I wish to find longer build orders, and because I also have an *attack-timing* as part of the strategy which can take on a considerable range of values, using an exhaustive search would not be feasible.

Instead I developed a method in order to answer the research question, that asks whether a coevolutionary algorithm can be used to find a set of viable strategies and counter-strategies for an RTS game. This method takes the solutions that are found in a single run of the CA, and evaluates their *viability*, *robustness* and *specialization*. Note that in this context, solutions are strategies played by the computer players in Decnalab.

Selecting some of the best strategies found by the CA, we can compare their average values to those of the baseline strategies. If the average viability is higher for the CA solutions than the baselines, the CA is considered to have found a better set of strategies and counter-strategies.

4.5.1 Introduction of concepts

A strategy's *robustness* is a measure of its ability to perform well against a range of strategies, while its *specialization* is a measure of its ability to perform well against particularly difficult strategies.

Viability is a measure of how viable a strategy is with respect to certain sets of strategies, and is determined via the strategy's robustness and specialization. A strategy that is highly specialized may be viable although it is not very robust, while a robust strategy may be viable although it is not very specialized.

4.5.2 Set of opponents and challengers

In order to evaluate specialization, robustness and viability for strategies, we need to define the set of *opponent* strategies they should play against, as well as what set of *challenger* strategies that also play against these opponents.

I have defined the set of *challengers* as the joint set of (1) the best found solutions – i.e. the population after the final generation of the CA – and (2) the baseline solutions.

The set of *opponents* can be defined in three ways: (1) only the baselines, (2) only the best found solutions, or (3) the joint set of baselines and found solutions. The CA may find solutions that are good with respect to one set of opponents, but less good with respect to another. Considering each set of opponents, we get more information about how the found solutions relate to the baselines, and possibly why they are better or worse.

We regard the baseline solutions to be the best known set of solutions prior to the CA, and the goal of the CA is to find a better set of solutions. A good set of solutions should be a set of viable strategies and counter-strategies. Therefore, each strategy in the set should be viable with respect to the set. Additionally, if strategies in the set are also viable against strategies not found in the set, there is less reason for them to be replaced by other strategies. If a set is less prone to having its strategies replaced, it is more stable, and thus represents a better set of strategies and counter-strategies.

4.5.3 Defining strategy performance

When evaluating specialization, robustness and viability I have chosen to consider only the outcome of a game: i.e. win, tie or loss. This is unlike the evaluation of shared fitness, where the score of a game is also taken into account in order to reward intermediate solutions that are likely better over those that are likely worse.

4.5.4 Calculating Robustness

The robustness of an opponent strategy is calculated as shown in the equations 4.2 and 4.3, where c is a challenger in the set of challengers C, o is an opponent in the set of opponents O, and $performance_R(c, o)$ calculates performance in terms of robustness for a game between c and o.

$$robustness(c, O) = \sum_{o \in O} \frac{performance_R(c, o)}{|O|} \tag{4.2}$$

$$performance_R(c, o) = \begin{cases} 1, & \text{if } c \text{ won against } o \\ 0, & \text{if } c \text{ and } o \text{ tied} \\ -1, & \text{if } o \text{ won against } c \end{cases} \tag{4.3}$$

4.5.5 Calculating Specialization

The specialization of an opponent strategy is calculated as shown in equations 4.4, 4.5, 4.6 and 4.7, where, again, c is a challenger in the set of challengers C and o is an opponent in the set of opponents O. $realativePerformance_S(c, C, o)$ calculates performance in terms of specialization, considering the outcome of a game between c and o in relation to the number of challengers that beat o.

$$specialization(c, C, O) = \sum_{o \in O} \frac{realativePerformance_S(c, C, o)}{|O|} \tag{4.4}$$

$$realativePerformance_S(c, C, o) = \begin{cases} \frac{|C|}{wins(C, o)}, & \text{if } c \text{ won against } o \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

$$wins(C, o) = \sum_{c_n \in C} didWin(c, o) \tag{4.6}$$

$$didWin(c, o) = \begin{cases} 1, & \text{if } c \text{ won against } o \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

4.5.6 Calculating Viability

Viability of a strategy is calculated as the sum of robustness and specialization, as shown in equation 4.8.

$$viability(c, C, O) = robustness(c, O) + specialization(c, C, O) \tag{4.8}$$

4.5.7 Selecting best found solutions

To evaluate the results from the CA we take into account some of the *best* found solutions, rather than the entire population after the final generation. As part of the CA configuration, the variable *nofBestFoundSolutionsToSelect* specifies how many of the found solutions we select when finding the best ones.

The following process is repeated for each set of opponents. Consider the set of found solutions to be the best found solutions. Then, evaluate their viability with respect to the set of opponents. While the number of best found solutions remaining is bigger than *nofBestFoundSolutionsToSelect*, we remove the least viable one and reevaluate.

The reason why we must reevaluate every time we remove a solution is that the set of challengers C affects the calculation of specialization, and thus viability. Imagine there are only two strategies c_A and c_B that can beat an opponent strategy o. After removing c_B, c_A will be considered more specialized, as it is now the only strategy in C that can beat o. Thus, c_A will also become more viable.

4.5.8 Evaluating overall results

Having evaluated and selected the best found solutions S_O with respect to each set of opponents O, the respective sets of challengers C_O is now the joint set of S_O and baselines (B).

We can evaluate the overall success of the CA by comparing the average robustness, specialization and viability of S_O to the averages of B. We do that by finding the difference, rather than the relation, because the values of specialization, robustness and viability may be zero (the latter to even negative). If the difference in average viability is above zero, S_O is a better set of solutions than B, considering O.

We would expect the set of opponents for which the difference is lowest (smallest or most negative), is when we let $O = B$. This is because the strategies in B are designed to be viable with respect to B, while the strategies in S – the population after final generation of the CA – are evolved to be viable against S. On the other end of the spectrum, we would expect the difference to be highest when letting $O \subseteq S$. (Note that O becomes a subset of S as we select only the most viable solutions from $S_O \subset S$.)

Chapter 5

Results and Analysis

For answering the research question I performed multiple experimental runs of the CA, simulating games in Decnalab and using the evaluation method I developed to evaluate the found solutions. In this chapter I will present and analyze the results from a selected experiment.

5.1 Finding viable strategies using Coevolution

Finding a CA configuration that could reliably produce good results according to my evaluation method proved more difficult than anticipated. Here I present the results from one run of the CA where I used the configuration shown in table 5.1.

Property	Value
popSize	50
nofChildrenPerGeneration	70
teachSetSize	8
maxGenerations	50
maxLoopsPerGame	2000
maxTicksPerGame	20000
fitnessScalingFactor	1.5
crossoverRatio	0.95
crossoverPointRatio	0.25
crossoverFunction	bitwiseUniformCrossover
perBitMutationRatio	0.01
producableThings	Worker, Marine, SupplyDepot, Barracks
possibleAddCounts	1
minAttackTiming	6
maxAttackTiming	42
initialBuildOrderLength	13
nofBestSolutionsToSelect	8

Figure 5.1: CA configuration

Most notably, I used a build order length of 13 (defined by *initialBuildOrderLength*), allowed producing workers, marines, supply-depots and barracks (defined by *producableThings*), and specified that each target in the build order should involve producing exactly one more unit or structure (defined by *possibleAddCounts*).

For the case injected solutions I used the handmade strategies presented in section 4.3, and I used the same strategies as baselines.

5.1.1 Results

This subsection shows the results from the first run of the CA with the specified configuration. The results are divided into two parts: (1) the evaluation of the found solutions and (2) the statistical data of how the solutions developed throughout the run of the CA.

Evaluation of found solutions

Presenting the evaluation of found solutions, I will show step-by-step how the values are calculated.

The evaluation is performed with respect to three different sets of opponents. The following values were found when letting the set of opponents be the set of

baselines.

Table 5.1 shows the outcomes of games between the baseline solutions. On each row, we see the id of the baseline and the outcome of the game for that baseline against each of the baselines (including itself, against which it ties).

Id/Id	B-00	B-01	B-02	B-03
B-00	tie	loss	win	win
B-01	win	tie	loss	loss
B-02	loss	win	tie	loss
B-03	loss	win	win	tie

Table 5.1: Game results for the baselines (row) when playing against other baselines (column)

Table 5.2 shows the outcomes of games between the eight best found solutions and the baselines. It shows that almost all of the found solutions win against baseline B-00, most of them win against B-02, while almost none win against B-01 and B-03.

Id/Id	B-00	B-01	B-02	B-03
S-08	win	win	win	loss
S-04	win	loss	win	loss
S-17	win	loss	win	loss
S-09	win	loss	win	loss
S-18	win	loss	win	loss
S-41	tie	loss	loss	win
S-22	tie	loss	loss	win
S-07	win	loss	tie	loss

Table 5.2: Game results for best found solutions when playing against the baselines

Table 5.3 and 5.4 shows the evaluation of the baselines and the eight best found solutions, in terms of specialization, robustness and viability.

S-08, which is the only found solutions to beat B-01, gets a high specialization. It also gets a high robustness, as it manages to beat three of the baselines.

Half of the found solutions have a robustness of 0, which means that they win and loose against equally many of the opponents.

Comparing the specialization values of the found solutions and the baselines, we see that all but one of the found solutions have a higher specialization than even the best baseline.

Id	Specialization	Robustness	Viability
B-00	0.48	0.25	0.73
B-03	0.48	0.25	0.73
B-02	0.33	-0.25	0.08
B-01	0.14	-0.25	-0.11

Table 5.3: Evaluation of baselines, using baselines as opponents

Id	Specialization	Robustness	Viability
S-08	1.24	0.50	1.74
S-04	0.57	0.00	0.57
S-17	0.57	0.00	0.57
S-09	0.57	0.00	0.57
S-18	0.57	0.00	0.57
S-41	0.67	-0.25	0.42
S-22	0.67	-0.25	0.42
S-07	0.29	-0.25	0.04

Table 5.4: Evaluation of best found solutions, using baselines as opponents

Finally, I present the overall evaluation of the CA performance, shown in table 5.5. The table shows the difference between the average values for the best found solutions and the average values for the baseline solutions, with respect to each set of opponents. The values from which the difference is calculated are shown in parentheses.

As described in section 4.5, when the difference is above zero the CA is considered to have found a set of solutions that is better than the baseline set, with respect to that set of opponents.

	vs baselines	vs solutions	vs all
Viability	0.25 $(0.61 - 0.36)$	-0.14 $(0.54 - 0.68)$	0.02 $(0.53 - 0.51)$
Robustness	-0.03 $(-0.03 - 0.00)$	-0.22 $(0.00 - 0.22)$	-0.12 $(-0.04 - 0.08)$
Specialization	0.29 $(0.64 - 0.36)$	0.08 $(0.54 - 0.46)$	0.15 $(0.57 - 0.43)$

Table 5.5: Overall evaluation of best found solutions

In this run the CA found a set of solutions that is more viable (0.61) than the baselines (0.36) with respect to the baselines. We see that the baselines are marginally more robust, but the found solutions are considerably more specialized. This corresponds with what we saw from table 5.3 and table 5.4.

Interestingly, the viability of the best found solutions is higher when evaluated with respect to baselines than they are when evaluated with respect to themselves. As described in section 4.5.8, this is the opposite of what I would expect.

Development throughout CA run

Figure 5.2 shows an overview of the most important statistical data from the run of the CA.

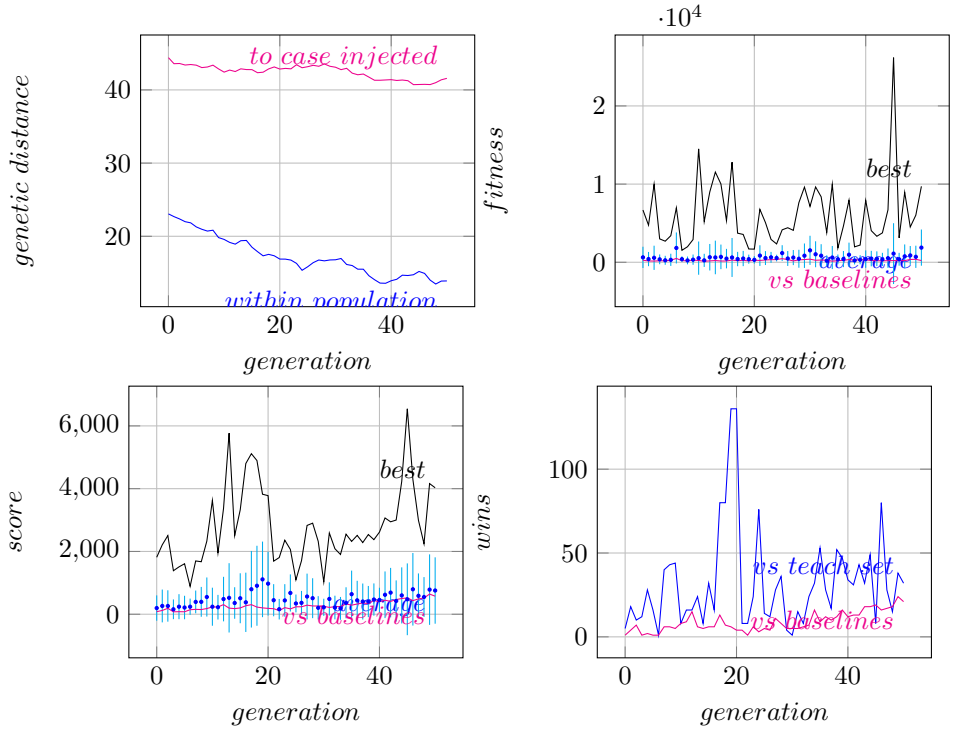

Figure 5.2: Overview.

The top-left graph shows the genetic distance within the population in blue, and the genetic distance from the population to the case injected solutions in magenta. As you can see the genetic distance within the population is declining throughout the CA, ending up at approximately half the genetic distance with which it started. This is not unexpected behaviour, as the initial, randomly generated solutions are possibly more different than any set of viable solutions.

Compared to other runs of the CA using parameters that facilitated a lower exploration, the final genetic distance is here quite high.

The top right graph shows the shared fitness of the population when evaluated against the teach set. The upper, black line shows the fitness of the best individual per generation, while the blue dots with cyan bars show the average fitness with standard deviation. In magenta we see the average fitness of the population against the baselines.

The bottom left graph shows the score of the population against the teach set, with max, average and standard deviation like in the previous graph. The average score against the baselines is shown in magenta.

Lastly, the bottom right graph shows the total number of wins for the population against the teach set, in blue, and against the baselines, in magenta.

Figure 5.3 shows more detailed results from the CA. In the three graphs on the left we see the fitness, score and number of wins of the population against the *teach set*, some of which was also shown in figure 5.2. On the right we see the same values for the population when playing against the *baselines*.

All graphs show the corresponding maximum in solid black, the average in dots and the standard deviation as bars. What we can clearly see from this set of graphs is that the performance of the solutions are increasing against both sets, but slightly more, and more steadily against the baselines. This means that the (1) difficulty of the teach set is evolving as the CA finds new and better solutions, but (2) the difficulty of the teach set is not strictly increasing, because the CA some times fail to produce equally good solutions for the next generation.

In figure 5.4 we see the performance of the *teach set* when playing against the *baselines*. This confirms our belief that the quality of the solutions in the teach set flucutates over the generations.

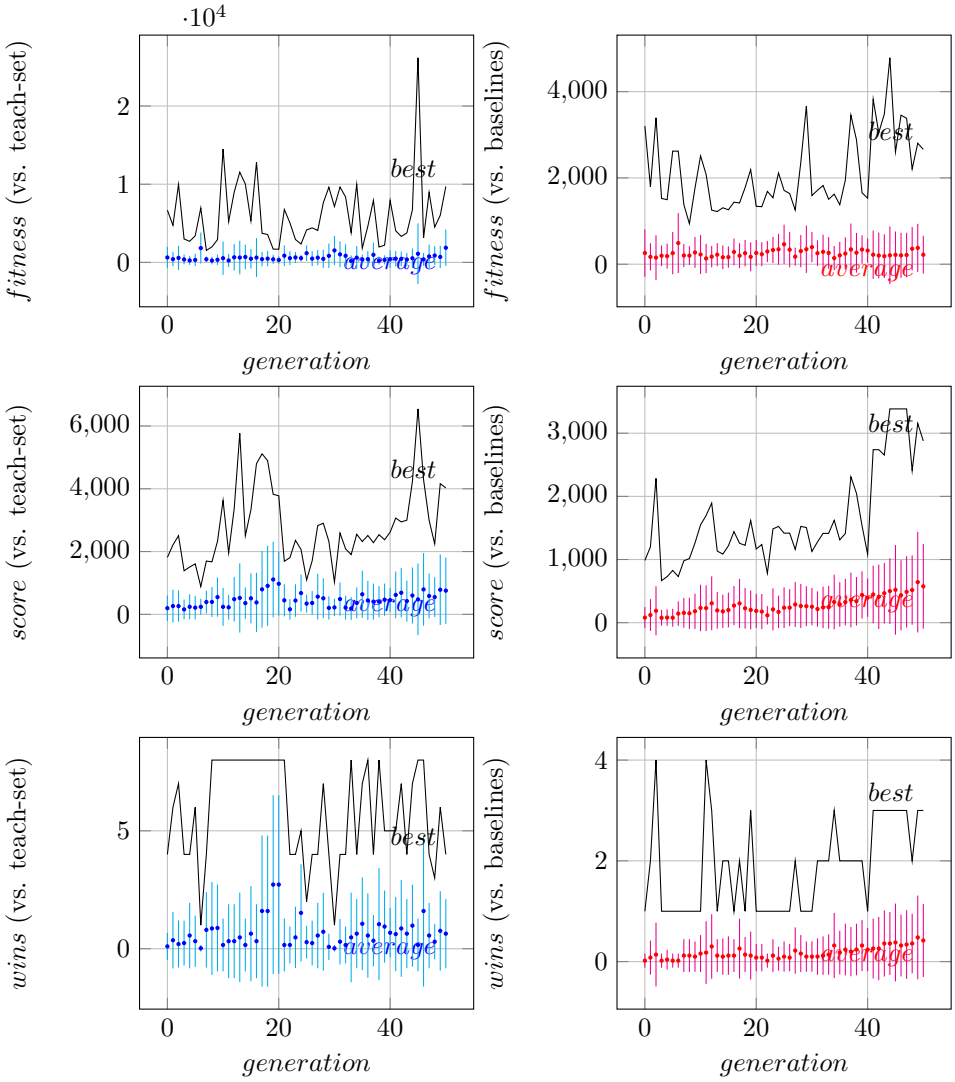

Figure 5.3: Detailed results for *population* vs teach-set and baselines

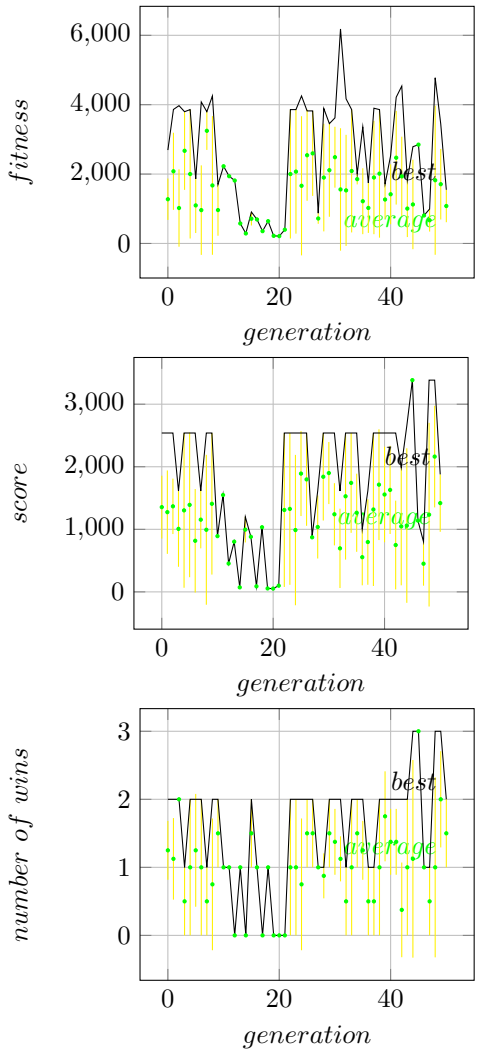

Figure 5.4: Detailed results for *teach-set* vs baselines

Chapter 6

Discussion

In this chapter I discuss the research question in light of my findings.

6.1 Can a CA find a representative set of viable strategies?

Through the experimental runs using different parameters for the CA, I found that there are some big challenges to creating viable sets of strategies. These problems mainly involve convergence towards a small subset of solutions and fluctuation in performance.

6.2 Convergence and fluctuation

Convergence is seen as a steady decrease in genetic distance throughout the generations and a low number of unique solutions at the end of the CA. Convergence in genetic algorithms, such as CAs, should be avoided because it can cause the algorithm to narrow in on suboptimal solutions, and makes it less likely to find the best solutions in the search space.

Throughout the generations, the performance of the solutions – measured in terms of fitness, score and number of wins against both the teach set and the baselines – would vary significantly from generation to generation. This *fluctuation* is caused by loss of good solutions, as explained in section 5.1.1.

In order for a CA to consistently produce viable sets of build orders for an RTS, these problems must be tackled.

6.3 Preventing convergence

Encoding the genomes of the individuals as bit-like strings – which in turn could be translated into JSON-objects representing the strategies – made it possible to change the genomes in more ways. Each atomic part of the JSON-structure requires multiple bits to be encoded, and thus encoding it as bits increases the number of parts of which it is composed.

This enables a much higher degree of recombination during crossover, and also increases the number of ways in which a genome can be mutated.

6.4 Preventing fluctuation

The longer the encoding of the genome of a single solution, and the more each part of the genome or solution relies on another, the less likely it is that a solution's children will inherit it's favorable traits. I believe this is the cause of the fluctuation in performance.

6.4.1 Slowing the rate of change

To mitigate this problem we could reduce the mutation ratio and the degree of recombination that occurs during crossover. The latter may be achieved through e.g. reducing the crossover ratio or crossover point ratio, or by using a specialized form of crossover such as sequence-wise crossover, as described in section 4.4.

Although these measures could potentially mend the problem, they do so only by slowing down the rate of change in the search. This means that given the same size of the solution space, it may take longer to find equally good solutions.

6.4.2 Reducing the solution space

An alternative approach to preventing fluctuation is to simply reduce the solutions is to simply reduce the solution space. If the solution space can be reduced without reducing the capabilities of the solutions, this may make it easier to search for a solution to the problem.

Reducing responsibilities of the solutions

As described in chapter 1, real time strategy games pose a very large set of complex problems, and the success of a strategy depends on the way in which each of these problems are solved.

I decided search for strategies consisting of (1) a build order and (2) and attack-timing, and let all other things be equal for both teams in a game. Since

neither team tries to solve e.g. the problem of micro-managing their units to any degree, it should be fair to both teams.

Although I had narrowed it down to a subset of only two problems – finding a build order and finding an attack-timing – the solutions space was still perhaps a bit too large.

Giving the computer controlled players a default behaviour for deciding when to attack, the only problem we would need to tackle is to find build orders.

Variable length genomes

Using genomes of variable length would make it possible to start with smaller build orders while still making the CA capable of producing long, complex build orders.

6.5 Redefining the solutions

A final thing that could simplify the search for solutions is to redefine the solutions and thus the solution space.

Making the the solution space more connected might decrease the distance between good solutions, which in turn could improve the probability that the CA could produce a better solution from an already good one. Because better solutions are more likely to be selected, this would both (1) reduce the fluctuation in solution quality, and (2) increase the CAs capacity for improvement per generation.

6.5.1 Ordering of targets

I encoded the build orders as lists of targets, and the order of the targets is defined simply as the order in which they occur. One artifact of that is that there is no single operation that can swap the order of two targets. If there was such an operation, the solution space would become.

6.5.2 Entirely different ways of encoding build orders

Lastly, I want to point out the possibility of encoding build orders in entirely different ways, such as Artificial Neural Networks (ANN).

Chapter 7

Conclusion

In this chapter I will provide a short conclusion to this study.

Research question *Can a coevolutionary algorithm find a representative set of viable build orders and counter-build-orders for an RTS game?*

From my research and analysis I have found that, to some extent, a coevolutionary algorithm can find sets of viable build orders and counter-build-orders for an RTS game. Further advancements are needed in order to perform this task in a sufficient way that it can be used for evaluating balance in RTS games.

I also identified some challenges. These challenges mainly involve avoiding convergence towards a small subset of solutions and preventing fluctuation in performance by preventing the loss of good solutions.

7.1 Future work

Future work may apply techniques to reduce fluctuation in performance and convergence, such as variable length genome, multiple populations that allow evolving more different types of solutions, and different ways to encode build orders, such as Artificial Neural Networks.

Bibliography

Ballinger, C., Louis, S., and Liu, S. (2016). Coevolving robust build-order iterative lists for real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1.

Beume, N., Hein, T., Naujoks, B., Piatkowski, N., Preuss, M., and Wessing, S. (2008). Intelligent anti-grouping in real-time strategy games. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 63–70.

Danielsiek, H., Stuer, R., Thom, A., Beume, N., Naujoks, B., and Preuss, M. (2008). Intelligent moving of groups in real-time strategy games. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 71–78.

Garćıa-Sánchez, P., Tonda, A., Mora, A. M., Squillero, G., and Merelo, J. J. (2015). Towards automatic starcraft strategy generation using genetic programming. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 284–291.

Park, J., Zhang, D., and Lu, M. (2008). An intelligent agent for the game of age of mythology: the titans. In *Information Reuse and Integration, 2008. IRI 2008. IEEE International Conference on*, pages 92–97.

Preuss, M., Beume, N., Danielsiek, H., Hein, T., Naujoks, B., Piatkowski, N., Stuer, R., Thom, A., and Wessing, S. (2010). Towards intelligent team composition and maneuvering in real-time strategy games. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):82–98.

Rosin, C. D. and Belew, R. K. (1997). New methods for competitive coevolution. *Evolutionary computation*, 5(1):1–29.