Norwegian University of
Science and Technology

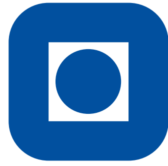
Evolving Behaviour Trees:

Automatic Generation of AI Opponents for Real-Time Strategy Games

Hallvard Jore Christensen
Jonatan Wilhelm Hoff

Master of Science in Informatics
Submission date: June 2016
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science



NTNU – Trondheim
Norwegian University of Science and Technology

Evolving Behaviour Trees:

Automatic Generation of AI Opponents for Real-Time Strategy Games

Jonatan Hoff, Hallvard Jore Christensen

June 2016

MASTER THESIS
Department of Computer and Information Science
Norwegian University of Science and Technology

Supervisor: Professor Keith Downing

Problem Description

We want to look into how bio-inspired methods such as genetic programming can be used in the game industry. Specifically, we will investigate how these methods can be used in conjunction with behaviour trees to increase the quality of AI in games.

abstract

Video games are a source of fun and enjoyment for millions of people across the globe. Artificial Intelligence (AI) is an essential part of many games and there is an increasing demand for ever more realistic computer controlled players. There are many methods and approaches for creating AI for games, with state machines and scripting featured in the majority of projects. Behaviour trees have emerged as a recent competitor, combining features of final-state machines and hierarchical task networks. As the available computational powers increase, the feasibility of using evolutionary computations in the development of game AI rises.

This project explores evolving behaviour trees using bio-inspired methods. This is done by tailoring genetic programming to represent individuals as behaviour trees which control a bot that plays a real-time strategy game. The individuals were evaluated by having them compete against each other, a hand-written behaviour tree and an AI bot implemented using traditional methods. Five experiments were conducted using a variety of parameters in order to explore the suitability of using these techniques conjointly.

The results from this project demonstrate that evolving behaviour trees is an interesting technique for automatically generating AI players which can consistently beat ones produced by humans using the same components, although evolving solutions that are serious competitors of traditional AI bots proved more difficult.

sammendrag

Dataspill er en underholdingskilde for millioner av mennesker verden over. Kunstig intelligens er en vesentlig del av mange spill, og det er en økende etterspørsel for mer realistiske datastyrte spillere. Det finnes mange metoder og teknikker for å lage datastyrte spillere, der state machines og scripting brukes i majoriteten av tilfellene. Behaviour trees er blitt en gradvis mer populær metode som kombinerer egenskapene til final-state machines og hierarchical task networks. Når databehandlingskraft øker, blir det mer nærliggende å bruke evolusjonære metoder i utviklingen av datastyrte spillere.

I dette prosjektet utforsker vi en teknikk for evolusjon av behaviour trees gjennom biologi-inspirerte metoder. Det blir utført ved å skreddersy individ-representasjonen i genetisk programmering til å følge samme notasjon som behaviour trees. Disse individene blir dermed brukt til å kontrollere enkelte aspekter i et sanntidsstrategispill. Deretter blir de evaluert gjennom interne kamper mot hverandre, samt en mer tradisjonell datastyrt motstander, standard motstanderen som følger med det utvalgte spillet og en menneske-designet versjon basert på samme system. Fem eksperimenter ble utført som tok i bruk en variasjon av parametere med hensikt å utforske hvorvidt genetisk programmering er en egnet metode til å utvikle behaviour trees.

Resultatene fra dette prosjektet viser at evolusjon av behaviour trees er en interessant teknikk for å automatisk generere datastyrte spillere. Disse spillerne har konsekvent utkonkurrert håndskrevne behaviour trees. Det har derimot vist seg å være vanskelig å generere spillere med denne teknikken, som slår spillere produsert ved hjelp av mer tradisjonelle metoder.

Acknowledgements

We wish to thank our supervisor Professor Keith Downing, at the Department of Computer and Information Science, Norwegian University of Science and Technology, for his guidance throughout this project.

We wish to thank family and friends for their advice and support.

(JWF, HJC)

Preface

This dissertation has been written as the final part of the authors masters degrees in Artificial Intelligence at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU).

Trondheim, June 1, 2016

Jonatan Hoff

Hallvard Jore Christensen

Contents

Acknowledgements iv
Preface v
Acronyms x

1 Introduction 1
1.1 Motivation 1
1.2 Goal and Research Questions 3
1.3 Research Method 4
1.4 Contributions 4
1.5 Thesis Structure 5

2 Background 6
2.1 Game Intelligence 6
2.1.1 RTS Games 6
2.1.2 Challenges and Common Solutions 7
2.2 Behaviour Trees 8
2.2.1 Structure 9
2.2.2 Flow 10
2.3 Biological Inspiration 13
2.3.1 Evolutionary Algorithms 13
2.4 Genetic Programming 14
2.4.1 Representation 14
2.4.2 Population 15
2.4.3 Selection 15
2.4.4 Elitism 16
2.4.5 Recombination 16
2.4.6 Mutation 17
2.4.7 Evaluation 18
2.4.8 Strongly Typed Genetic Programming 18
2.4.9 Competitive Coevolution 18
2.4.10 Bloat 19
2.5 Summary 20

3 Related Work 21
3.1 Literature Review 21
3.1.1 Identification of Research 21
3.1.2 Selection and Grouping Strategy 22
3.2 Related Systems and Projects 22
3.2.1 Behaviour Trees in Research 23
3.2.2 Genetic Programming for Games 24
3.2.3 Evolving Behaviour Trees 26
3.2.4 Key Takeaways 29

4 Methodology 31
4.1 System Overview 31
4.2 Zero-K 32
4.2.1 Spring RTS Engine 32
4.2.2 Gameplay 32
4.2.3 Choice of Map 35
4.2.4 First Factory 35
4.2.5 Opponent 35
4.3 GameInterface 36
4.3.1 Design Choices 37
4.3.2 Structure 37
4.3.3 Game Flow 38
4.3.4 Economy Manager 38
4.3.5 Recruitment Manager 39
4.3.6 Influence Manager 40
4.3.7 Military Manager 41
4.4 Behaviour Tree 42
4.4.1 ECJ Integration 42
4.4.2 Implementation 44
4.4.3 Variable Abstraction 47
4.4.4 Execution 47
4.5 GP 48
4.5.1 ECJ 49
4.5.2 Fitness 50
4.5.3 SpringProblem 51
4.5.4 CompetitiveSpringProblem 52
4.5.5 Statistics and Logs 52
4.5.6 Selection 53
4.5.7 Genetic Operators 53

4.5.8 Strong Typing 54
4.6 Environment 57
4.6.1 Information Flow 57
4.7 Summary 58

5 Results and Discussion 60
5.1 Experiments 60
5.1.1 Experiment Overview 60
5.1.2 General Parameters 61
5.1.3 Experiments 1: Small Population, no Mutation . 62
5.1.4 Experiment 2: Large Population, no Mutation . . 64
5.1.5 Experiment 3: Small Population, Mutation . . . 67
5.1.6 Experiment 4: Large Population, Mutation . . . 69
5.1.7 Experiment 5: Competitive Coevolution 70
5.1.8 Hand-written BT 73
5.2 Comparative Evaluation 74
5.2.1 Competitive Results 75
5.2.2 Build Order 76
5.2.3 Human Trials 77
5.3 Discussion 78
5.3.1 Fitness Function 78
5.3.2 Stagnation 79
5.3.3 Mutation vs. Reproduction 83
5.3.4 Population Size 84
5.3.5 Bloat 85
5.3.6 Strong Typing 88
5.3.7 Competitive Coevolution 88
5.3.8 Evolutionary Scope 89
5.4 Summary 90

6 Conclusion 92
6.1 Goal Evaluation 93
6.2 Contributions 96
6.3 Limitations & Future Work 96
6.3.1 Extend Evolutionary Scope 96
6.3.2 Technical Improvements 97
6.3.3 Behaviour Tree Library 97
6.4 Closing Remarks 98

A Additional Resources **99**
A.1 Repositories 99
A.2 Hand-written BT 99
Bibliography 101

Acronyms

GP Genetic Programming

BT Behaviour Tree

AI Artificial Intelligence

RTS Real-Time Strategy

FSM Finite State Machine

NPC Non-Player Character

HFSM Hierarchical Finite State Machine

DAG Directed Acyclic Graph

EA Evolutionary Algorithm

GA Genetic Algorithm

EC Evolutionary Computation

STGP Strongly Typed Genetic Programming

GUI Graphical User Interface

Chapter 1

Introduction

This report documents research into the use of Genetic Programming(GP) for evolving Behaviour Trees(BT) in order to create an Artificial Intelligence(AI) player for a real-time strategy(RTS) game.

This chapter introduces the rest of the report. Section 1.1 outlines the motivation and background, section 1.2 defines the research questions, section 1.3 presents the research method, section 1.4 describes the contributions of this research and section 1.5 outlines the thesis structure.

1.1 Motivation

The field of artificial intelligence is a large branch of computer science that contributes to many other fields of science. A common testbed for AI techniques are video games and the reason behind this is described succinctly by Marvin Minsky in the quote below.

> "*It is not that the games and mathematical problems are chosen because they are clear and simple; rather it is that they give us, for the smallest initial structures, the greatest complexity, so that one can engage some really formidable situations after a relatively minimal diversion into programming.*" — Marvin Minsky

AI research for games has mostly focused on games of perfect information like checkers [J. Schae er and Bryant., 1996] or chess, rather than more modern video games. Modern games often involve both long and short-term planning as well as complex decision-making. As Sid Meier put it;

> “*A good game is a series of interesting decisions. The decisions must be both frequent and meaningful.*” — Sid Meier

Generally, AI for games come in the form of AI-bots which refer to automated players that can play a video game in place of a human player. The most common techniques for developing AI-bots are knowledge-intensive techniques like Finite State Machines (FSMs) or rule-based scripting.

An interesting technique in game AI currently gaining popularity is behaviour trees [Champandard et al., 2010]. BTs hold several advantages over traditional methods like FSMs for larger systems, specifically in the areas of design simplicity, scalability, modularity and reusability. Its hierarchical and goal-oriented approach make BTs a powerful method for describing complex behaviours.

Game AI development often revolves around giving the impression of intelligence with simple and cheap techniques rather than developing something more advanced. The downside of this is that human players can easily identify and exploit the weaknesses and adapt their strategy to match it, resulting in a game that is less fun to play through another time. Another trick that game developers use is to give the AI player significantly more information and resources which some players find detrimental to the experience as they want to compete against the AI player on as equal terms as possible.

As the available computational power increases, evolutionary techniques offer potential improvements to the development of game AI. In recent years, there has been much research based on RTS games like Starcraft. These are games where a player consolidates resources, constructs buildings and recruits units in order to destroy other player's assets. An advantage of an approach using evolutionary techniques is that it could produce a large set of AI players, offering the human player a varied selection of opponents to play against. Another advantage is that it requires less domain knowledge from the developer as the AI will effectively learn the game while evolving.

We believe the combination of evolutionary algorithms and behaviour trees will prove a good fit, providing a powerful tool for AI developers to evolve behaviour for autonomous agents in video games. Evolving AI for games in this manner might also be useful for finding bugs and unintended exploits in the game environment. It can exhaust edge-cases and combinations human players might not, seeking an

advantage in the game. Designing behaviour for complex games can be very time consuming as it often involves a lot of manual work and therefore evolving AI could provide a useful alternative.

A few studies have addressed the idea of evolving BTs through evolutionary algorithms and some of the results looks promising. We want to explore these techniques further by applying them to a more complex game. In this project we wish to explore using genetic programming to evolve behaviour trees with the purpose of automatically generating AI players for RTS games.

1.2 Goal and Research Questions

This section defines the goal and the research questions pursued in this thesis.

> **Goal statement:** *Explore the automatic generation of artificial intelligence opponents for a real-time strategy game using genetic programming to evolve behaviour trees*

We will explore the use of genetic programming to evolve behaviour trees with the purpose of generating artificial intelligence players for a real-time strategy game by answering the three research questions defined below.

> **Research Question 1:** How well does genetic programming work in concert with behaviour trees?

Genetic programming revolves around automatically generating computer programs, usually represented in syntax tree form. Behaviour trees are a popular approach for designing AI players for games that use a similar tree notation. We want to investigate whether this coupling of techniques is particularly suited to working conjointly.

> **Research Question 2:** Given the same components, how do behaviour trees evolved using genetic programming compare with ones designed by humans?

Behaviour trees are usually carefully hand-crafted by designers and domain experts. We will investigate if behaviour trees evolved using this technique can produce better results than humans, given the same

amount of environment abstraction in a RTS game. Additionally, we will examine how the evolved BTs are structured, and if any interesting structural patterns or clever uses of the provided nodes/components are evident.

> **Research Question 3:** Can using genetic programming to evolve behaviour trees simplify or improve the development of AI players for games?

Specifically we will evaluate the following questions. How does using genetic programming to evolve behaviour trees compare with more traditional approaches? Can this technique be used by developers to improve or assist in the creation of AI players? Does this method produce better AI opponents than regular approaches?

1.3 Research Method

In order to investigate the outlined research questions, we developed a system that evolves AI players in the form of behaviour trees, using genetic programming, for the real-time strategy game Zero-K. Experiments were then conducted by modifying the parameters governing the evolutionary algorithm. Five such experiments were performed and evaluated by comparing them against each other as well as the default AI player that the game was shipped with and a hand-written behaviour tree. The results were then analysed in respect with the stated research questions. Finally, additional lines of enquiry were outlined in the form of a future work section.

1.4 Contributions

This project contributes to two fields of science; game intelligence and evolutionary computation. For the former, the contribution is in the form of exploring methods for creating AI for games that are potentially faster and results in more complex and dynamic behaviours. For the latter, it is about finding new potential applications for established methods.

1.5 Thesis Structure

This thesis is divided into six chapters. Chapter 2 describes the background theory. Chapter 3 outlines how relevant literature was identified in the form of a structured literature review and describes related systems and projects. Chapter 4 explains how the experiments were conducted and which tools were used and developed as part of that goal. Chapter 5 outlines the experiments, then presents and discusses the results. Chapter 6 contains the the conclusions of this project as well as suggestions for further work.

Chapter 2

Background

In this chapter, we present a selection of information gathered as part of the literary review which will provide a deeper understanding of the subject matters. Section 2.1 contains a brief introduction to real-time strategy games and the methods commonly employed when developing AI for them. Behaviour trees, a mathematical model of plan execution widely used in the games industry is described in section 2.2. In section 2.3 and 2.4, the biological inspiration of evolutionary algorithms and genetic programming are introduced respectively. Section 3.1 outlines how the literature review was conducted. Lastly, section 3.2, describes similar projects that have had an influence on this project.

2.1 Game Intelligence

The purpose of this project is exploring a combination of techniques for generating non-player characters(NPCs) for a real-time strategy game. This section defines what a real-time strategy game is and what the common challenges and solutions are.

2.1.1 RTS Games

Real-time strategy is a genre of computer games where the player positions and manoeuvres units and structures under their control in order to secure areas of the game environment, or map as it is colloquially known, and destroy their opponents assets. Furthermore, this is all done in real-time, contrary to turn-based games. Generally speaking, RTS games consist of gathering resources and using them to create soldiers to engage the enemy. Consequently, the player's attention is divided between the economical and the military aspects of the game. Economy and base-building can be described as long-term

planning strategy whereas the military is more of a tactical challenge consisting of *micro-management*, the task of controlling combat units in detail while in combat. Two of the more famous RTS games are Starcraft and Age of Empires.

2.1.2 Challenges and Common Solutions

There are several challenges with developing AI for RTS games and we will describe some of them in this section. For an in-depth description see [Millington and Funge, 2009].

RTS games require the player to deal with imperfect information and randomness while at the same time formulating a winning strategy. Games of *imperfect information* are games where the players may be unaware of the actions chosen by other players. This aspect makes it significantly harder to predict an opponents actions as opposed to a game of *perfect information* like chess where a player can figure out all of the opponents possible moves in any state. A common way of handling this is simply giving the AI more information and resources than the player, which is sometimes referred to as cheating AI. Some players feel competing against an AI opponent with an unfair advantage detracts from the gaming experience.

An RTS AI often needs to use spatial reasoning to effectively use terrain to their advantage. Most RTS games contain some form of terrain, often in the simple form of varying height which may affect the movement and performance of units. There are also larger terrain obstacles like mountains, seas and forests which may have a great effect on strategy and tactics, for example a narrow valley between two mountains might be a perfect location for fortifications. If the terrain is represented using a height map, it is relatively simple to analyse it depending on the complexity of the game. A common way is to look at the neighbouring values in a height map and based on the variance of values determining the ruggedness of the terrain. For more complex games, one might need to analyse the effect that traversing heavy terrain has on equipment and so on.

A good game AI needs to adapt to its opponents. A human player can quickly form an understanding of the strategy employed by a bot, but it is a considerable task to design an AI that can intelligently adapt to opponents or even give the appearance of doing so. There are many techniques for performing decision making in games; decision

trees, finite state machines, behaviour trees, fuzzy logic, rule-based systems and scripting to name a few. If complex enough, each of these techniques can be used to give the appearance of intelligence. It does however take a considerable amount of work to get to that point, especially with a technique like scripting, where a game designer manually encodes the behaviour or parts of behaviour of an AI entity. In order to reduce the workload a developer can use learning to automatically determine behaviour for aspects of a game. Common techniques are decision tree learning, action prediction, reinforcement learning and evolutionary algorithms.

Lastly, creating game AI is not just about making it as strong as possible, it also needs to be at the right challenge level to entertain the human players. The mapping between challenge and fun varies greatly between people. The most common way of solving this is letting the player choose a difficulty from a list, but there have also been attempts at adjusting this automatically based on the player's behaviour, for example the game Rimworld by Ludeon Studios which uses an *AI storyteller* [Studios, 2016].

Developing artificial intelligence for games is by no means a straightforward task. There are many techniques to choose from and each has advantages and disadvantages. The most important job of game AI is to increase the entertainment value of the game for the player. Both by making the game more entertaining and the challenges more rewarding, but also by increasing the replayability of the game.

2.2 Behaviour Trees

The field of Game AI has a constant need for new solutions to keep up with the demand for the ever increasing realism in modern games. The most popular approaches for writing AI is time consuming and lacking in modularity, especially for the development of Non-Player Characters.

Finite State Machines with their simple layout are extensively used within the game industry. They are very useful for simple action sequences but quickly become illegible as the task grows more complex due to state explosion. *State explosion* is a problem which occurs when the traditional state machine formalism inflicts repetition causing the FSMs to grow much faster than the complexity of the system it describes. This complexity makes it difficult for developers to modify and maintain the behaviour of the autonomous agents. For larger

systems *Hierarchical FSMs* (HFSM) provides reusable transitions between states and a higher level of scalable logic. Their main shortcoming are that they do not provide modularity nor reusability of states, thus demanding a rewrite for different behaviours.

The concept of the *behaviour tree* was originally suggested by R. G. Dromey in [Dromey, 2003] as a method to formally define system design requirements, the framework was later adapted by the computer game industry to control NPCs [Isla, 2005]. They offered a more modular and scalable approach than the methods used at the time. BTs differ by not having transitions to external states; they are simply collections of actions that execute and terminate. In a search through the tree structure the flow of the decision making is controlled. BTs are depth-first, ordered *Directed Acyclic Graph*s (DAGs), $G(V, E)$ with $|V|$ nodes and $|E|$ edges. This means that each edge is directed, leading from one node to another in such a way that starting from the root node there are no execution sequences that lead back to the root. We call the outgoing node of a connected pair the *parent* and the incoming node the *child*. Additionally, child-less nodes are referred to as *leaves* while other non-leaves are described as *control-flow* nodes. The only deviant is the unique parent-less node *root* often labelled $\emptyset$.

2.2.1 Structure

The trees are often structured by having a hierarchical level of goals created by recursively simplifying the goals into subtasks similar to that seen in hierarchical task networks. They can be deep, with nodes calling subtrees, allowing the developer to create whole libraries of behaviours. These can be chained to solve the AI entity's goals through very convincing AI behaviour in a structured and fairly human-readable manner. This gives the developer the opportunity to work in an iterative fashion. Starting with low level behaviour, then providing alternative methods of achieving small goals, see figure 2.2. These goals are ordered according to their desirability and cost, giving the AI entity fallback tactics should it fail. Repeating this process recursively and solving higher level goals for the entity with an ever growing behaviour library, both the development process and results, many will argue, are far superior to other alternatives. By introducing some randomness to the flow in the tree, an AI entity might be perceived as more human and therefore more convincing. Because of

their traits, BTs are a popular approach in the gaming industry having being used in many high profile video games such as Spore [Hecker, 2009], Bioshock [Champandard et al., 2010] and multiple entries in the Halo series [Isla, 2005, 2008].

2.2.2 Flow

The flow of the tree is determined by three states common to all nodes within the structure, which are; success, failure and running. They are used to give the parent abstracted information of the state of their children; what the result of its termination was, or that it has not been determined yet. Consequently individual nodes have very strict contracts of communication, and any tree can be assured that the given set of statuses propagate through its branches. Most behaviour tree implementations will have three different node groups. Composite, decorator and leaf-nodes.

Composite

The composite nodes are the ones that to the largest degree determine the structure and in what way a BT is executed. They have two or more children and will execute them in a particular order. When the processing of the composite node is finished it will pass on either succeed or failure to its parent. While waiting for its children the node will return *running* to the parent. This applies to all nodes in the tree. The most common composite nodes are selector, sequence and parallel.

Sequence might be the simplest commonly used composite node. It will, as the name suggests, process each child in sequential order. If a child fails, the processing stops and the sequence node fails. Only if all children succeed will the sequence return success to its parent, making it analogous to the AND gate. This provides a range of useful applications. The most obvious of them that it can consist of subtasks that must all complete for the main task to succeed. Look at the example in figure 2.1 where the task "enter house" only succeeds if all subtasks do. Sequences may also function as an if-statement when it has conditions followed by actions. The actions will only process if the conditions are met.

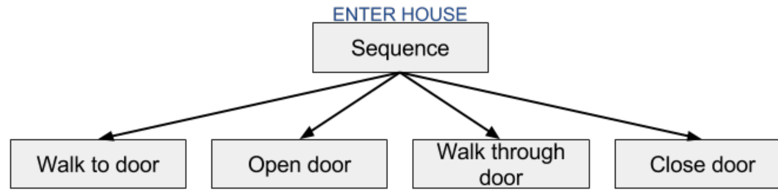


Figure 2.1: Example: enter house problem solution in the form of a behaviour tree.

Selectors are often referred to as the opposite of the sequence node. Where the sequence needs all its children to succeed, selectors terminate as soon as one child succeeds. If all children fail, it too will fail. Again using the analogy of logic gates, selectors can be likened to OR gates. Because of these properties selectors are well suited to create fallbacks for unsuccessful tasks. Supplementing the example in figure 2.1, figure 2.2 shows a deeper behaviour tree providing fallback behaviour using selectors. The *open door* action is replaced with a subtree with alternative routes if simply opening the door fails. This subtree provides multiple ways to open the door sorted based on desirability. If the door is reinforced and even breaking it fails, the *enter house* behaviour may have higher level fallbacks, like *enter window*. Layering tasks and goals with backup routes like this is the core of designing a successful BT. Even by just utilising sequences and selectors, powerful behaviour can be designed. It's important to note that though selectors usually process children from left to right, some implementations might do it in a random order so as not to make the behaviour too predictable. This is the case for the other composite nodes as well. It is not uncommon to both include a standard and a random implementation.

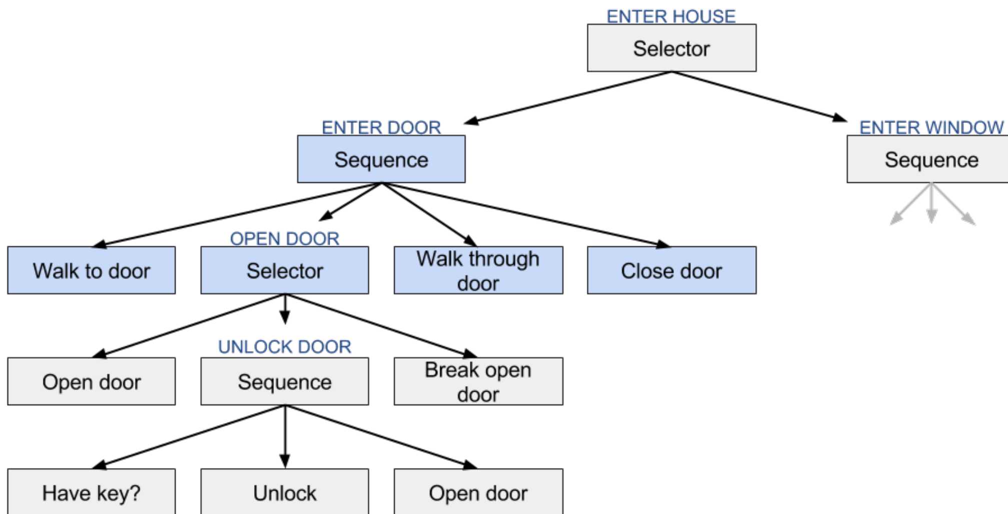


Figure 2.2: Example: enter house BT providing fallbacks options.

Parallel nodes work in a fundamentally different way from the other composite nodes. They handle concurrent behaviour, starting or resuming all its children every time. Some implementations will only simulate concurrency and not actually use multi-threading, but as long as the desired behaviour is achieved the semantics are not that important. Parallel nodes will behave differently based on their policy. The *policy* of the node tells it when it should terminate and what state to return to the parent. Usually the policies are based on sequence and selector termination, where the sequence-policy make it fail as soon as a child fails while the selector-policy succeeds when a child succeeds. This enables more reactive behaviour where tasks can be aborted in favour of reacting to changes in the environment.

Decorator

The second category of nodes are *decorators* which differ from other nodes by only allowing a single child. Decorators do not execute any behaviour directly but still greatly affect the behaviour of the entity. Their function is to either repeatedly run the child until the desired propagating result is received, terminate it, or most commonly, transform the received result from the child before propagating it to its parent. A good example of the latter, is the *Inverter* which as the name suggests inverts the result. If the child succeeds the inverter will return failure to its parent, and vice versa.

Leaf

While the other categories are more general, *leaf nodes* are mainly problem specific and do not have children. There are two types of leaf nodes, actions and conditions. *Actions* will have the entity execute some behaviour and return whether it was successful or not (e.g. `Walk to door` in figure 2.2). *Conditions* run checks on the environment or entity and return the result (e.g. `Have key?` in figure 2.2). These nodes will to a large extent define the perceived environment and the scope in which the entity can operate. An expressive set of leaf nodes supplemented by composite and decorator nodes, allows for layered and complicated behaviours.

2.3 Biological Inspiration

In biology and natural evolution, an individuals ability to reproduce is essential to the success and even survival of its species. This ability is determined by an individuals characteristics which are encoded as genes in a genome. Changes in the genome can occur due to mutation or sexual recombination of the parent's genetic code. These changes can be both beneficial and deleterious, for example by granting the individual an improved resistance to decease. An individuals genome is transmitted to its offspring and tends to propagate into new generations. The offspring's characteristics are partially inherited from their parents and partially generated during the process of reproduction in the form of mutation. The success of an individual can be described as its ability to survive and reproduce in a specific environment, also known as *natural selection* [Darwin and Bynum, 2009]. Natural selection is not synonymous with *survival of the fittest* but instead refers to the individual that is best adapted, rather than in the best condition, with respect to a specific environment.

2.3.1 Evolutionary Algorithms

In 1975 John Holland outlined the genetic algorithm which became the foundation of the field of Evolutionary Computation (EC), defined by the type of algorithms it is concerned with, namely Evolutionary Algorithms(EA) [Holland, 1975]. There are many variants of EAs which are based on the same foundation. Evolutionary pressure is applied to a population of individuals, which causes natural selection

to occur, resulting in a more fit population. Practically, this is accomplished by starting from a high-level statement of what needs to be done in the form of a quality function, then randomly generating a set of candidate solutions to that problem. The quality function can then be applied to the candidate solutions as a fitness measure. The better solutions are chosen using a selection criteria to form the basis of the next generation, which is created by applying one or more of the genetic operators recombination, mutation, reproduction. *Recombination* is the process of taking two individuals and creating one or more new individuals from them that contains parts from both. *Mutation* is a genetic operator that is applied to one individual and gives rise to a single new one. *Reproduction* in this case, refers to making a direct copy of an individual and adding it unchanged to the next population. These operators are applied until a full new, hopefully fitter, population has been created. This process continues until either an acceptable solution is found or the process is stopped based on a previously set criterion.

2.4 Genetic Programming

In 1992, John Koza wrote a book in which he outlined an extension to Hollands genetic algorithm in which the population consists of computer programs rather than bit strings [Koza, 1992]. This extension is known as genetic programming and can be defined as an EC technique that automatically solves problems without requiring the user to know the form or structure of the solution in advance [Poli et al., 2008]. Specifically, problems which require computer programs as solutions. This section aims to give an introduction to genetic programming as well as illustrate how it differs from other EAs.

2.4.1 Representation

In GP, individuals are generally represented as *syntax trees*, which is a way of encoding computer programs in tree form, see figure 2.3. In syntax trees variables and constants are *leaf nodes* or *terminals* as they are also known, while the internal nodes are *functions*, usually arithmetical in nature. The set of allowed functions and terminals in a GP system is referred to as the *primitive set*. It can be implemented as in the provided example where a single number is output from the root

node, alternatively a leaf could be an action such as steering left or accelerating in a racing simulator. Two common terms in genetics are genotype and phenotype. *Genotype* is the set of properties of an individual which can be mutated and altered. *Phenotype* refers to a candidate solution(individual).

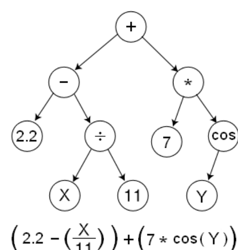


Figure 2.3: Example of a syntax tree generated by GP

2.4.2 Population

There are two main ways of implementing populations in GP; generational and steady-state. Generational GP produces a new population every generation whereas steady-state replaces individuals from a constant, static population. Both methods and most other EAs usually initialise their populations the same way, by generating it randomly. In GP this is generally done by first setting a minimum and maximum size of the initial trees, size being defined as the tree depth, then using a combination of the grow method and the full method known as *ramped half-and-half* [Koza, 1992]. The *full* method generates trees of the maximum depth where all branches end at the same depth. The *grow* method allows branches that end with a terminal before it reaches the maximum depth. Ramped half-and-half is typically a 50/50 mix of grow and full and is used in order to get a population containing as much variation as possible.

2.4.3 Selection

Selection is the way in which individuals are probabilistically chosen based on their fitness to contribute to the next generation. In GP, the three most common selection methods are tournament selection, fitness proportionate selection and rank selection [Mitchell., 1997].

In *tournament selection* a number of individuals k, also known as *tournament size*, are chosen from the population at random and compared against each other in a tournament. The best individual in

the tournament is picked with probability p, the second best with probability $p * (1 - p)$, third with $p * ((1 - p)^2)$ and so on. As two parents are needed for recombination, two tournaments are run for each recombination event. The greater the tournament size the smaller chance an individual with low fitness has of being chosen.

Fitness Proportionate Selection (FPS) is a method where the probability of an individual i, with a fitness f_i, of being chosen is $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, where N is the number of individuals in the population.

In *Rank Selection*, individuals are sorted in a list according to their fitness. An individual's probability of being picked is proportional to its rank in the list, rather than its fitness.

2.4.4 Elitism

Elitism is a technique used to ensure the best individual or individuals are not removed from the population. It works by copying over the best individual from the current generation to the next before selection is applied. This has the effect of always keeping the best individual found so far in the population. Typically this is used for the one or two best individuals depending on total population size.

2.4.5 Recombination

One of the large differences between GP and other classes of EAs is the way in which recombination is performed. A common representation of non-GP EAs are binary strings, where recombination is often performed using the one point crossover technique. *One point crossover* takes two individuals (binary strings) of equal size and chooses a crossover point at random. Then all binary characters on the left side of the crossover point is taken from one parent and the ones on the right are taken from the other parent, see figure 2.4. In GP however, the most common method of recombination is known as subtree crossover. *Subtree crossover* works by choosing a *crossover point* (node) from both parents and replacing the subtree rooted at that node in one parent with the subtree from the other parent, see figure 2.5. The tree we are left with is the offspring resulting from this recombination. It is also possible to configure it in such a way that each recombination yields two children by having the parents switch subtrees.

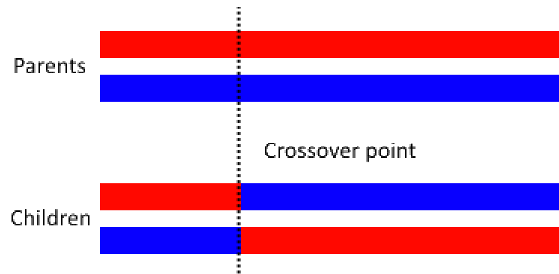


Figure 2.4: Example of 1 point crossover as used with genetic algorithms in binary representation.

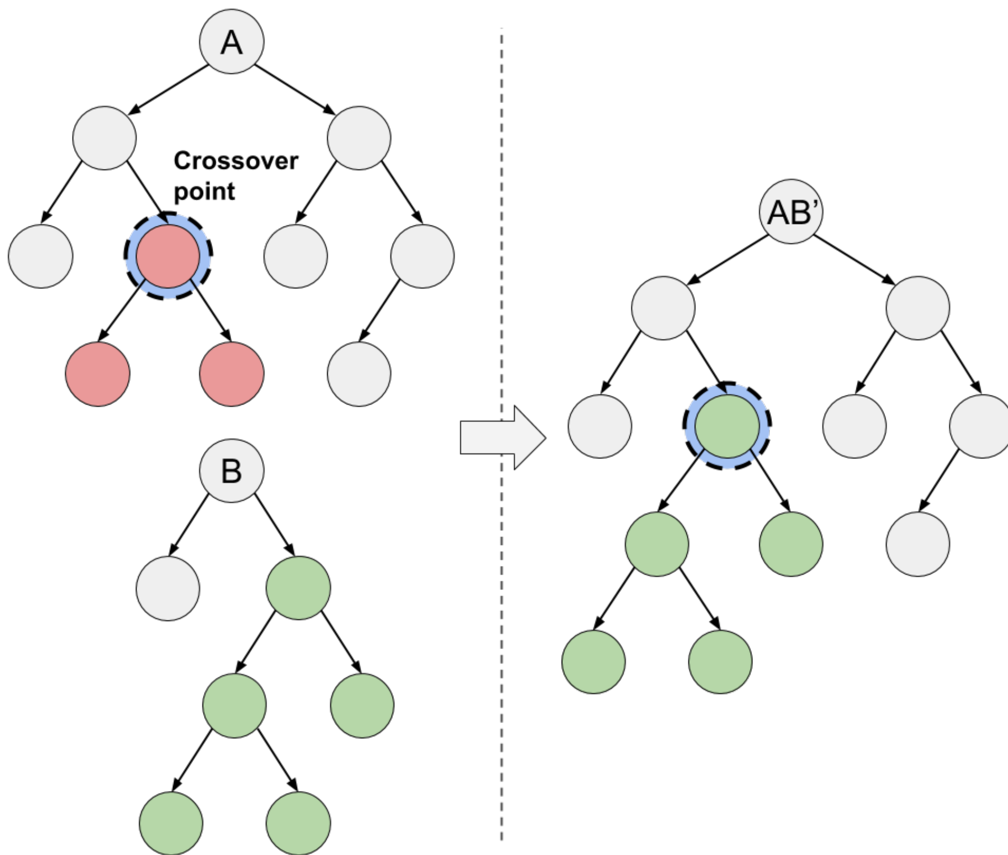


Figure 2.5: Example of subtree crossover, with the parents on the left side and the offspring on the right.

2.4.6 Mutation

In GP there are two main forms of mutation, subtree mutation and point mutation. *Subtree mutation* works by randomly choosing a subtree and performing crossover with a newly generated random tree. *Point mutation* works by selecting a single node and replacing it with a similar compatible node. e.g. replacing a + with a - or a 3 with a 11.

2.4.7 Evaluation

In GP, evaluation can be performed in many ways depending on the problem domain. When solving symbolic regression problems, where the goal is to discover the best model for a given data-set, evaluation is performed by providing input variables in the form of leaf nodes in a syntax tree, and comparing the output from the root node with the known answer. The mean error resulting from this is then used as the fitness, with a low number indicating a fitter individual. In the case of more complex problems requiring some form of simulations, evaluation is generally performed by running a specific scenario in a simulation and scoring the individual based on how well it performs.

2.4.8 Strongly Typed Genetic Programming

In some cases the complexity of the primitive set, such as a mixture of data types, makes it difficult to achieve *closure*, which can be defined as

> the assumption that any non-terminal should be able to handle as an argument any data type and value returned from a terminal or non-terminal[Montana, 1995].

Additionally in some cases it is desirable to enforce a particular structure that is believed or known to be important [Poli et al., 2008]. A solution to both of these problems is *Strongly Typed Genetic Programming (STGP)*. STGP incorporates a type system into GP such that every terminal has a type and every function has types for their arguments as well as a type for the return value. This means that the process that generates the initial trees, as well as the process for crossover and mutation, must be implemented in such a way that it does not violate the defined type constraints.

2.4.9 Competitive Coevolution

Coevolution is a population-based technique which revolves around simultaneously evolving populations of solutions with a coupled fitness. There are two categories of this technique; cooperative and competitive coevolution. In cooperative two or more populations are evolved on separate parts of a greater problem but are cooperatively evaluated to determine fitness. In the case of *competitive coevolution* you pit individuals against each other and fitness is based on how fit an

individual is compared to other individuals rather than how well it satisfies a quality function. This method is particularly suited to evolving AI for games as the great range of potential opponents makes it very difficult to define a fitness function that adequately covers all of them [Rosin and Belew, 1995]. The absence of an advanced fitness function also simplifies the setup of GP considerably as less knowledge of the problem domain is required. A common method for determining fitness in competitive coevolution is *tournament fitness*, defined as a single elimination (see figure 2.6), binary tournament which determines a relative fitness ranking [Angeline and Pollack, 1994]. All individuals are randomly paired off and run against each other, the winners then progress to the next round where the same process is repeated. Using this metric an individual's fitness is the level in the tournament hierarchy they achieve.

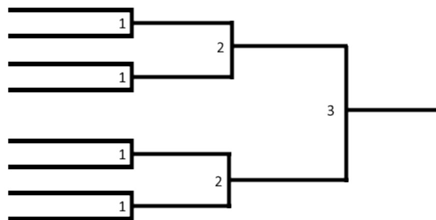


Figure 2.6: Example of a single elimination tournament as used in tournament fitness for competitive coevolution

2.4.10 Bloat

Bloat is a common problem in GP that [Poli et al., 2008] is defined as

> program growth without (significant) return in terms of fitness

While the definition of bloat is agreed upon the explanation as to why it occurs is not. There are three main explanations for this phenomenon. Firstly there is the *replication accuracy theory* [McPhee and Miller, 1995] which states that the success of an individual is determined by its ability to replicate its parents functionality. The consequence of this is ever more accurate replications of individuals in the form of bloated trees.

Another theory is the *removal bias theory* which states that over time the subtrees inserted during crossover will end up being larger than the parts they replace, thus leading to growth in the average program size [Soule and Foster, 1998].

Finally there is the *nature of program search spaces theory* [Langdon and Poli, 1997] which asserts that as there is a greater number of long

programs, there will also be a greater number of long programs of a given fitness than short programs of the same fitness. Because of this GP will end up sampling more and more long programs for the simple reason that there are more of them.

There are numerous proposed techniques for managing bloat, here is a very short overview of three of them. The most obvious technique is setting a depth limit, specifically not accepting new programs above a specified size and choosing new parents when this occurs. This however has the effect over time of creating a population with an average size that is close to the depth limit, resulting in a less varied population [Koza, 1992]. Another method is *size fair crossover* which add constraints during crossover so that the subtree chosen from the second parent is not significantly larger than the first, consequently limiting the average size growth of the population [Crawford-Marks and Spector, 2002]. Lastly, a technique defined by [Poli, 2003] called the *tarpeian method*, which modifies the selection probability of programs by randomly setting the fitness of larger-than-average programs to 0.

2.5 Summary

In this chapter we have presented the background on which this project is based. In section 2.1 we introduced some of the challenges of the field. In section 2.2, the workings of behaviour trees and the motivation for using them was presented. In sections 2.3 and 2.4 we explained the inspiration and basics of genetic programming.

Chapter 3

Related Work

3.1 Literature Review

Getting familiar with existing research is very important. Finding similar research projects and exploring their shortcomings, restrictions and future work often gives a solid base for new projects and a good connection to the field.

3.1.1 Identification of Research

This section outlines the strategy for finding primary studies that was employed in this project. During the first stages of the project we conducted an examination of a set of resources, listed below, in order to find relevant studies which would define the scope of the project and form its foundation. The bibliographies of relevant literature were examined in order to identify further potential sources. With the purpose of forming an in depth understanding of the current state of the relevant fields many combinations of the search terms 3.1 were queried against the resources listed below.

Literature Resources

This is a list of resources which were searched in the process of identifying relevant literature.

- IEEE Xplore
- Springer Link
- ResearchGate
- Google Scholar

- ACM Digital Library

3.1.2 Selection and Grouping Strategy

After collecting a sufficient literary catalogue, pertinent studies and other documentation were selected and grouped based on the combination of terms applicable to the research scope. See table 3.1.

Category	Search terms
EA	Genetic Programming, Grammatical Evolution, Bio-inspired methods, Competitive Coevolution
Game AI	Behaviour Tree
Game	RTS, Zero-K

Table 3.1: Terms in correlation to topics

The search queries resulted in a range of resources. Only a small set of studies turned out to be directly related to our research goals, whereas the bulk was only relevant to specific parts of the project but made good background reading. As our scope covers a range of fields, we followed a selection process to reduce the hundreds of studies into a manageable set. The titles were evaluated, looking for correlations between the study and our research questions, then grouped according the criteria specified in table 3.2. Secondly, their abstracts were evaluated in order to decide which should be examined further, resulting in a manageable pool of studies.

Group	Minimum requirement
1st	Evolutionary algorithms in combination with behaviour trees
2nd	Combination of two of the search term categories
3rd	Exploring a relevant concept within one of the terms
4th	Not directly relevant but containing applicable concepts or relateable information

Table 3.2: Terms and group placement

3.2 Related Systems and Projects

This section contains descriptions of related work identified when performing the literature review. It is divided into four subsections. Firstly, section 3.2.1 which covers research projects where behaviour trees have been used. Following that is section 3.2.2, describing projects where GP or closely related techniques have been applied to

games. Thirdly, the most relevant studies, namely those that relate to evolving behaviour trees, are described, section 3.2.3. Lastly there is a section describing the key takeways from these related project, see 3.2.4.

3.2.1 Behaviour Trees in Research

The current form of behaviour trees was developed within the computer game industry [Isla, 2005; Champandard, 2007] as a more modular and flexible alternative to FSMs. Their recursive structure, usability and readability made them popular in the game industry, which in turn lead to a increasing amount of attention in academia.

Behaviour trees have been postulated as a possible solution to the control of Unmanned Aerial Vehicles (UAV) [Ogren, 2012]. According to Ögren, Hybrid Dynamical Systems (HDS) are a common way of writing controllers for robots which could benefit from being encoded as BTs. HDS are systems that contain both continuous and discrete dynamic behaviour. Large HDS often become quite complex leading to a lack of modularity and scalability. Encoding a HDS as a behaviour tree replaces the explicitly listed state transition functions of a classic HDS with transitions that are implicitly given by the tree structure in BTs. Ogren also lists readability as a great advantage of a behaviour tree approach. On a side note, BTs have received a increased interest from the field of robotics including multiple implementation for soccer bots created for the RoboCup [Abiyev et al., 2013], displaying their applicability to other fields of research.

Tomai and Flores investigated how designer-created BTs could be automatically modified in order to give the user a less repetitive experience in a Massively Multiplayer Online Role-Playing Game (MMORPG) [Tomai and Flores, 2014]. In order to gather data on how players behave in a MMORPG, they created their own simple game and observed the behaviour. Various unpredictable human behaviours were found, including socialising, exploring and idling. Using designed deterministic BTs that follow a simple combat/collect quest structure as base, the discovered behaviour incorporated traits from human players with decorators they call *modifiers*. In many cases this includes not doing the optimal action, but the BT tree is automatically adapted to explain the player's choices and generally performed well.

First described as powerful tools for designing NPC behaviour, later studies have adapted the use of behaviour trees far beyond the gaming industry by proposing having BTs control UAVs [Ogren, 2012], while also having them contribute towards formal verifications of mission plans [Klöckner, 2013]. Following an increase in interest, studies seeking to contribute to this technology are on the rise. Suggestions for new improvements include functionality to enable parameter passing for increased flexibility [Shoulson et al., 2011]. BTs are mostly designer-created and some studies try to achieve less predictable behaviour. Some suggests automated modifications by learning from human players [Tomai and Flores, 2014] and a few others, like ourselves, suggest using BTs in combination with EAs. Other studies take on more formal aspects of the growing field of behaviour trees by suggesting a unified framework [Marzinotto et al., 2014] or compute performance analysis [Colledanchise et al., 2014] to quantify other advantages. Behaviour trees are a prevalent technique in the gaming industry, and show little indication of receding.

3.2.2 Genetic Programming for Games

In this subsection we describe and discuss related research projects that have used genetic programming or other EAs in conjunction with games.

Miles used used coevolution to evolve influence maps in order to create tree based strategy game players [Miles et al., 2007]. *Influence maps* (IM) is a technique used to facilitate decision-making in computer games. It works by mapping the game world to a grid where each square contains a value that describes some aspect of the game. Game objects like units exert influence on the IM which is propagated to nearby areas. The maps can be combined in order to condense more information, for example figure 3.1, where there are two units, friendly F and enemy E that each exert influence that cancels out each other. In a game this could be used to find out where the battle lines are or where there is the most tension. Miles used several IMs to condense large amounts of information to a form more easily interpreted by the AI. For example an IM that produces high values near vulnerable enemies combined with one that produces high negative values near powerful enemies, resulting in an IM where a high value indicates a vulnerable enemy. In order to evolve this system IMs where represented as leaf nodes, with arithmetical operations as the internal

nodes, in a syntax tree. Miles used two-population co-evolution with one population containing attacking strategies and the other defencive strategies. The results were quite good, the evolved bot did well both versus other bots as well as human players. Miles described future work as increasing the evolutionary scope of the task as well as performing more experimentation with the evolution parameters.

.4	.8	1	.8	.4	0	0	0
.8	1.6	2	1.6	.8	0	0	0
1	2	F	2	1	0	0	0
.8	1.6	2	.8	0	-1	-.8	-.4
.4	.8	1	0	-.8	-2	-1.6	-.8
0	0	0	-1	-2	E	-2	-1
0	0	0	-.8	-1.6	-2	-1.6	-.8
0	0	0	-.4	-.8	-1	-.8	-.4

Figure 3.1: Influence map illustration. F stands for friendly unit and E for enemy unit.

Garcia developed a GP system where the individuals can be used to generate C++ classes which control aspects of the RTS game StarCraft [Garcia-Sanchez et al., 2015]. He states that there are two main forms of AI in RTS games, namely strategic and tactical. The former takes decisions affecting the whole set of units in a game and defines the high-level direction of a match whereas the latter controls a specific unit or subset of units. Garcia chose to focus on the strategic part of the game by evolving an AI that controls the construction order of buildings in the game and the unit composition of military squads. Two types of fitness functions were employed in this study; victory-based and report-based. The former runs the evolving bot against a set of bots developed by other researchers and scores it on victories. The latter scores individuals on how well they perform in terms of military, economy, relative destruction, game length and relative economy in a single match. The latter was included to provide a smoother slope towards good solutions. The resulting AI bot was able to defeat hand-written bots 61% of the time. Garcia proposes more experimentation with fitness functions and competitive coevolution as future work.

Agapitos applied two learning methods to the problem of evolving a controller for a simulated racing game [Agapitos et al., 2007]. The first method employed was GP, both with and without automatically defined functions. The GP approach was compared with an evolved neural network approach which used multi-layer perceptrons and Elman-style recurrent neural networks. The sensor variables available to the algorithms were all of the type that could realistically have been retrieved by sensors placed on a car e.g. speed, angle and distance to the next waypoint. The results indicated that GP found good solutions faster whereas neural networks found better ones though after a significantly longer time.

Alhejali investigated using training camps with GP in order to evolve game playing agents [Alhejali and Lucas, 2011]. *Training camp* refers to a technique where a problem is divided into a set of smaller problems that can be solved easily on their own. The solutions to the sub-problems are then combined into a larger solution which covers the whole problem. Practically, this was done by running GP on each sub-problem and then seeding the population of a larger GP run with the best individuals from the sub-problems and letting evolution combine them as it sees fit. The authors found that the training camp method outperformed a pure GP approach. Additionally they found that it made the results of GP less volatile and thus more predictable.

The articles described in this section provide background information on how genetic programming can be used to generate AI for video games. [Miles et al., 2007] described a way of abstracting complex game situations into manageable numbers as well as motivation for using co-evolution. [Garcia-Sanchez et al., 2015] described several ways of affecting the high-level strategic behaviour of an RTS agent. [Alhejali and Lucas, 2011] illustrated a way of splitting up a complex problem in order to improve the results achieved with GP. All in all, these papers show that GP has been used in conjunction with games successfully on several occasions.

3.2.3 Evolving Behaviour Trees

In this subsection we describe and discuss the papers that are most relevant to this project as they use a form of genetic programming to evolve behaviour trees.

In 2013 B.J. Oakes wrote his master dissertation on the practical and theoretical issues of evolving behaviour trees for a turn-based game [Oakes, 2013]. Oakes' goal was to research how evolutionary computing could be applied to behaviour trees in order to evolve AI strategies and how this process could be improved. To this end he developed a system using GP and BT for a simplified version of the open-source game Battle for Wesnoth. The game revolves around moving units around on a tile-based board to capture villages that produce gold which the player needs in order to recruit more units. Oakes implemented BT in such a way that the tree is run once to control a whole set of units rather than a single unit. Furthermore, Oakes uses a blackboard for most interactions between nodes. For example the getUnits node which retrieves and places a list of available units on the blackboard. Individuals were evaluated by playing them against the game's default AI. Oakes conducted three experiments; evolving from a BT containing a single sequence node, evolving from a randomly generated population and evolving from a population seeded with three different hand-written BTs. The results were evaluated against each other as well as the default AI. Oakes consequently found that the evolved BTs outperformed the hand-written ones, additionally, the BTs that were evolved in a more varied environment outperformed the ones that were evolved in a static environment.

Perez et al. and Togelius wrote papers as part of their entries to the Mario AI competition in 2012[Julian Togelius, 2012; Perez et al., 2011]. They evolved behaviour trees using *grammatical evolution* (GA), a type of GP where you evolve grammar rather than code, in order to control the player in a version of the game Super Mario. The results were mixed; the developed agent showed good reactive capabilities, such as avoiding obstacles at close range and neutralising enemies, however, it did not do very well for path planning. They attributed part of their success to the use of grammatical evolution in combination with behaviour trees as well as the use of two point sub-tree crossover in GP.

In 2007 Robin Baumgarten developed an AI framework for the commercial strategy game DEFCON [Baumgarten and Colton, 2007]. DEFCON is a nuclear war simulation strategy game where players control a continent each and attempt to lose the least in a nuclear conflict. The player controls naval fleets, nuclear silos, satellite arrays and airports with the purpose of outwitting opponents. Baumgarten used the developed framework to create a bot using several traditional

AI techniques such as simulated annealing, decision tree learning and case-based reasoning, with some success. Case-based learning was used to find winning strategies from records of earlier games, these were then used together with descision tree learning in order to create an AI bot. The AI-bot used simulated annealing to prioritise targets in the game. The resulting AI managed to win against DEFCON's default AI 77% of the time.

In 2010, Baumgarten along with Chong-U Lim and Simon Colton attempted to create AI for the same game, this time using a combination of behaviour trees and genetic programming [Lim et al., 2010]. The evolutionary scope of the game was restricted in order to simplify the learning task. They used GP with a population of 100 individuals and 100 generations. They used FPS and a mutation rate of 5%. Four BTs were evolved separately and evaluated on different aspects of the game. The four resulting BTs were then stitched together manually to form a larger tree, where the evolved BTs were placed as sub-trees, resulting in a single BT that can play the whole game. The evolved BT beat DEFCONs default AI in 55% of the battles.

Evolving BTs using EAs is an increasingly popular topic in the academia. The research has been dominated by simulations in games, like the ones described above. Researchers at MIT however recently conducted the first application of the behaviour tree framework to a real robotic platform using the evolutionary robotics methodology [Scheper et al., 2015]. Utilising the 20-gram DelFly Micro Air Vehicle as base, with on-board optical cameras using a StereoVision algorithm called LonqSeq which extracts depth-information from the environment. Individuals were given fitness based on their performance at calculating the flight path in a simulation platform called SmartUAV. The fitness function had its main performance metrics in tree size and success rate for the task of navigating a square room searching for a window to fly through.

The behaviour tree implementation consisted of generic sequence and selector nodes with two conditions and one action; greaterThan, lessThan and setRudder, respectively. The implementation revolves around sharing information across the tree with the common blackboard object. The blackboard contains a total of four variables, the first four condition variables and the last is used to set the BT action output. Human designed behaviour tree was used as a baseline to judge the performance of the solution. The best individual was

optimised by pruning the BT and removing redundant nodes, then run with the same validation set as used with the human designed behaviour resulting in a success rate of 88% for the problem beating it by 6%. In evolutionary robotics, when moving from a simulated environment to reality, the simulation will always wary from reality to a degree resulting in artifacts known as the reality gap. When put to practice the reality gap proved significant for multiple metrics and considerable adjustments were needed. The test flights in the real physical environment showed a success rate of 54% for the genetically optimised tree, hence outscoring the user-designed who scored a 46% success rate.

Since these studies have such a high degree of relevancy, they form an important base for our research. Learning from their experiences and taking the described future work into account is important. The authors of both [Oakes, 2013] and [Lim et al., 2010] mention competitive co-evolution as an interesting next step. The former suggest using *training camps* to evolve various aspects of the game separately could be beneficial. In several of the related projects the authors refer to improving the fitness function as future work. [Lim et al., 2010] also mentions the risk of over-fitting which exist in most learning techniques, in his case, by focusing on too specific scenarios when evolving.

3.2.4 Key Takeaways

There were several interesting related systems and projects, some proved more relevant than others. In the case of behaviour trees, there was not any single most relevant project but rather the general impression of the state of the art of behaviour trees, which the study of the described papers provided. The opposite is true for genetic programming; we had a lot of knowledge of genetic programming from before but it was the details of how it has been used in conjunction with games that proved the most useful. The use of influence maps to condense large quantities of information as described in [Miles et al., 2007] has been a part of this project and will be described in the chapter 4. We also took inspiration from the experimentation with fitness functions in [Garcia-Sanchez et al., 2015] and the training camp technique for splitting tasks to be learnt into smaller chunks as used successfully in [Alhejali and Lucas, 2011]. The most inspiration

however, was taken from the projects described in section 3.2.3 as these projects effectively describe the current state of using evolution to evolve behaviour trees for games. [Oakes, 2013] and [Lim et al., 2010], as well as [Garcia-Sanchez et al., 2015], state competitive coevolution as future work and as a consequence, an experiment exploring it was included in this project.

Chapter 4

Methodology

This chapter describes the way in which the research questions were answered and which tools were used, and created to that end. The chapter starts with a description of the system overview in Section 4.1. Following that, Section 4.2 outlines the game-play of Zero-k, the game for which we developed AI, as well as the special considerations employed when working with it. Section 4.3 describes how the game was connected programmatically to the behaviour trees and genetic programming functionality as well as how we implemented a simple traditional AI bot for Zero-k. Section 4.4 contains a description of how behaviour trees were implemented for this project. Section 4.5 outlines how genetic programming was performed using the ECJ library. Following that, Section 4.6 describes how the various components of the system work together. Finally, there is a brief summary of the chapter.

4.1 System Overview

There are three main components in the system developed for this project; EvolutionRunner, the behaviour tree library and a game interface referred to as ZKGPBTAI(Zero-K Genetic Programming Behaviour Tree Artificial Intelligence). EvolutionRunner uses the ECJ framework [Luke et al., 2006] to run genetic programming. It generates an initial population of BTs using the behaviour tree library and then evaluates individuals by running a match of Zero-K using ZKGPBTAI. The behaviour tree library is designed to be a reusable component separate from the problem specific code. In the following sections each component will be described in detail.

4.2 Zero-K

Zero-K is a RTS game developed on the open-source RTS game engine known as *Spring Engine*. The gameplay consists of managing soldiers, consolidating resources, constructing buildings, recruiting units and outwitting your opponents. This game was chosen over others because of its comprehensive AI framework as well as its community, who were happy to answer questions about the source code.

4.2.1 Spring RTS Engine

Spring Engine is an open source 3D RTS game engine. The development of the game engine was inspired by the game Total Annihilation by Cavedog Entertainment, released in 1997. When Cavedog Entertainment went bankrupt, a group of fans got together and began development of a RTS engine that could replace the ageing Total Annihilation engine. In April 2005 Spring was released under a General Public License and has been in constant development since.

4.2.2 Gameplay

This section contains a description of the game mechanics that was considered when evolving a bot to play Zero-K. See figure 4.1 for images of the units described in this section.

There are two resources in Zero-K which the player must gather to produce units and buildings; metal and energy. Metal is gathered using *metal extractors* placed on metal spots, which are spread out on the map. Energy can be made anywhere using a range of buildings, in this project we consider only the *solar panel*. Metal is considered the more important of the two as it can only be gathered from specific spots on the map. Consequently, military expansion generally revolves around capturing and consolidating metal spots. There are infinite amounts of both resources; metal spots never run out of metal and the sun never stops shining. Players can construct the *storage* building to increase resource stockpiling capability, allowing the player to have a large stockpile in case metal or energy production is interrupted. It is important to have more energy than metal as the energy excess will be used to overdrive the metal extractors resulting in greater metal production. An additional way of acquiring metal is to reclaim it from the wrecks of buildings, units as well as some environmental objects.

Construction works a bit differently in Zero-K compared to most RTS games. The common way of doing construction is that the player pays all the necessary resources to construct a building upfront. In Zero-K, the cost of the building is drained from the player's resources while the building is being built. This means that there is a constant inflow and outflow of resources, in addition to the stored resources, both of which must be considered when making decisions. A common beginners mistake is to build too many buildings at once, effectively strangling the economy by having too much outflow and not enough inflow. We consider two types of construction units in this project; conjurer and commander.

An important aspect of Zero-K as well as a recurring feature in many games developed for the Spring Engine is the commander. The *commander* is the single unit the player controls in the beginning of a game. It is a strong all-round unit with a large health pool, offencive and defencive capabilities, and the ability to construct buildings. It is also possible to upgrade the commander during the game, but this feature was not considered in this project.

The *conjurer* is a basic construction unit with the added benefit of becoming cloaked when standing still. We do not consider the aspect of cloaking in this project as it is outside the evolutionary scope. Conjurers can be created using the the cloaky bot factory.

The factory is arguably the most important building in Zero-K. It is in the factory that a player can recruit more construction units as well as soldiers. There are 13 different factories in the game, however, for this project we only consider the *cloaky bot factory*, which contains a representative mix of unit types. As with construction, recruiting units drains the resource cost from the player's resource pool while building the unit.

Zero-K is quite advanced compared to many RTS games when it comes to physics simulations. The maps, the game environments you play in, have full 3D terrain which can be actively terraformed by the player. Where terraforming refers to dynamic modification of the game world. We chose to ignore this aspect in order to simplify the task to be learnt. All bullets fired by units have a chance of missing based on the terrain it traverses. For example, it is fully possible for a unit to hit other allied units while attempting to fire on an enemy. Additionally, units with explosive firepower like artillery, deform the landscape with their shots.

As in most RTS games there is fog of war. *Fog of War* (FOW) is a game concept that simulates the element of limited information in real warfare. Areas of the map covered by FOW are hidden from the player. In many cases, including this one, the player can see the lay of the land but all enemy units and buildings are hidden. Each unit has a *line of sight (LOS)*, an area around them in which they remove the FOW, revealing opponents in that area. In Zero-K there are two main ways of managing this; the first is using units as scouts by sending them to hidden areas of the map in order to reveal it. The second is building *radar* towers. If a player has radar coverage over an area of the map hidden by the fog of war, any enemy units in that area are revealed as coloured dots on the map. The player can see that there are enemies there but not what type. Additionally, radar towers have longer range when placed on high terrain and the signal is blocked by obstacles or heights in the area.

An important aspect of the game is defence. In addition to building soldiers to protect your buildings and to attack enemy buildings, a player can build defencive, stationary units, commonly referred to as turrets. In this project we consider two such units; the Lotus turret and Gauss turret. The *Lotus turret* is a cheap all-round laser turret. The *Gauss turret* is an expensive self-healing turret. The reasoning behind including these two is to offer the bot both a cheap and an expensive option to choose from. Both of the turrets can fire at ground and air forces, which was an additional reason for choosing those two turrets. Whereas we do not use any airborne units, CAI, the bot which we use to evaluate individuals might.

The last building we chose to include is the caretaker. The *caretaker* is a static constructor, meaning that it is a building that functions as a construction worker. The most common way of using it is to place it next to a factory and ordering it to assist the factory with recruitment, effectively doubling the production speed of units. It is also used to repair damaged units. It is important to build this unit in the correct part of the game, because if it is built too early it will consume resources at an accelerated rate by helping the factory construct units faster, potentially strangling the economy.

An important thing to note is that whereas the bot evolved in this project is bound to the limitations described in this section, the AI it is evaluated against, is not. CAI, the AI shipped with the game uses the full range of factories and most of the other buildings and units. For a

full list of the excluded aspects of the game see table 4.2.5

4.2.3 Choice of Map

All the conducted experiments are run on a single map, which is small in size and symmetrical. The reason for this is that a larger map significantly adds to the duration of each battle, greatly increasing the overall run time. A symmetrical map was chosen over an asymmetrical one so that the bot would not become specialised at playing on one side of the map. Additionally, the map does not contain any advanced terrain obstacles like water, mountain ridges or lava. The reasoning behind omitting maps containing these things is that it would require the bot to use specialised units in order to perform well, effectively adding another layer of complexity to the learning task. Ideally the bot should have been evaluated on several maps, thus ensuring that the BT that functions best on the largest variety of landscapes would prevail. Sadly, the length of the run prohibited this in the time-frame of the study.

4.2.4 First Factory

In Zero-K, the first factory is free of cost, and constructing it is usually the first command of the game. All other observed AIs, as well as most human players, start the game with this action. Consequently, it was decided that hard-coding the construction command of the factory into the bot as the default first action was acceptable. If this is not included the BT bots generally lose to CAI very quickly as they are sometimes not able to recruit workers nor soldiers.

4.2.5 Opponent

The development of AI opponents in the game is community driven, and bots are constantly being developed. The most common is CAI. It is a non-cheating bot that is currently the default AI players shipped with Zero-K. CAI is a tough opponent to beat, especially for newer players. It quickly builds a strong economy while scouting and raiding the enemy. Making use of most of the functions of the game it has a big advantage over the AI players developed in this project. Table 4.2.5 shows a list of aspects of the game not considered. CAI does not necessarily make

use of all possible functions of the game either, but it does it to a much greater extent.

Aspects of the game not considered
– Commander upgrades
– Only 13 of 100 mobile units are considered
– Only 1 of 13 factories are considered
– Terraforming
– Only 2 of 23 turrets are considered
– Air and naval warfare
– Power flow mechanic (pylon grids)
– Only 1 of 5 energy generating buildings are considered
– Various specialised buildings
– Nuclear warfare
– Cloaking and shielding
– Unit transportation
– CeaseFires (Alliances) and Restricted Zones
– Resource allocation priority for construction
– Evolution is only run on one map

Table 4.1: A list of game aspects that was not considered in this project.

Figure 4.1: Starting in the upper left corner: **a.** Metal Extractor, **b.** Solar Panel, **c.** Storage, **d.** Conjurer, **e.** Cloaky Bot Factory, **f.** Caretaker, **g.** Radar, **h.** Lotus, **i.** Gauss

4.3 GameInterface

This section will cover how and why the game interface was implemented and how it interacts with the behaviour trees. First there is a section describing some of the design choices made when developing the interface. After that, a description of the structure of the code and the four managers it is divided into.

4.3.1 Design Choices

An important design choice we made was to develop a tradition AI bot for Zero-k while developing the game interface. This was a part of our intention to use training camps, as described in [Alhejali and Lucas, 2011], in order to simplify the evolution task. The way it would work is that the evolved BT behaviour would gradually assume command of more and more aspects of the game. In other words, a training camp would be likened to an aspect of the game. For example the recruiting of units. Evolution of a BT to control the recruitment of units would have access to only nodes governing that specific domain while letting the implemented traditional AI player control the rest. When all major aspects of the game had been evolved as BTs, they could be stitched together to form a large tree that would be able to cover all parts of the game, forming a complete AI player. However, due to time and computation constraints we were only able to evolve a BT for one major aspect of the game, the control of economy using workers and construction. The traditional AI is also used to evaluate the performance of the evolved solutions in the results section and will from now on be referred to as B_1.

4.3.2 Structure

When developing the Spring Engine the developers did not want to spend a large amount of time developing AI opponents for the engine. Instead they left this task to the community and provided interfaces in a range of programming languages to build upon. These relay all events and information from the game and provides a channel to issue game commands. In the Java interface the AI implementation connects and gets access to this functionality by extending the abstract class `AbstractOOAI`. In the rest of the report "the game interface" refers to the provided AI interface, including the implemented AI as a whole system.

The basis of this implementation is the `Main` class. Its purpose is to initialise and load all necessary resources before further assigning all gameplay decisions to the respective managers. The game interface has four managers that each control a specific aspect of the game; Influence Manager, Economy Manager, Military Manager and Recruitment Manager. See figure 4.2. Additionally, it contains an implementation of every behaviour tree node, which retrieve

information from the various managers and executes commands through them, in order to fulfil their function.

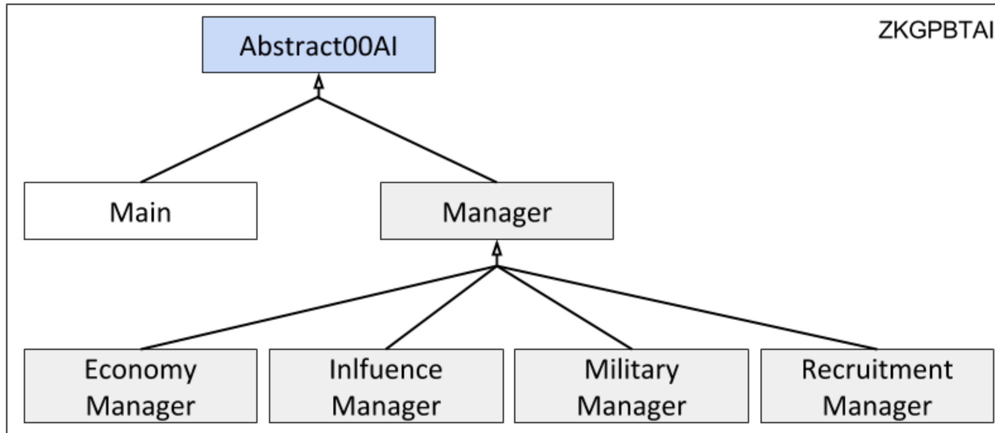


Figure 4.2: Class inheritance

4.3.3 Game Flow

The Zero-k AI framework calls the update function once per frame in the game with a callback object which contains all information about the game state. The update function calls its corresponding update function in each of the four managers. This enables each manager to control how often to execute behaviours from the different trees while keeping the code modular.

4.3.4 Economy Manager

The economy manager handles everything to do with measuring the state of the economy, affecting changes to it and the construction of buildings using workers. The current state of the economy is measured in terms of effective income, see algorithm 4.1, and expenditure, see algorithm 4.2. These values are updated every fifth frame and are used in many calculations throughout the game interface.

In B_1, the decision of what building to construct is done by iterating through a prioritised list of buildings and running a function consisting of pre-conditions for each building. For example, for the bot to decide to build a storage building, the effective income has to be above 15 and the current levels of energy and metal has to be within 100 units of the maximum limit. Whenever a worker finishes a task it is given a new one immediately using this method. The code handling construction and workers is called every 60th frame in order to minimise computation.

An exception is data about the state of the economy which requires practically no computation and is updated every fifth frame.

In the BT bots, all construction and worker movement is driven by the action and leaf nodes in the BT. Note that all BTs are called through the Main class, not the economy manager. They are called with a higher frequency (f_{tick}) than the economy manager in general, ticking the behaviour trees every 20th frame.

When a building type has been chosen for construction, the bot determines where to place it. Building placement is governed by a set of rules; buildings cannot be placed too close together as this can result in units being barricaded in by a wall of buildings, neither can they be placed on top of metal spots as this would limit the amount of metal resource the player could gather, finally factories may not be placed too close to the edge of the map as that may prohibit units from leaving the factory. Additionally the Zero-k AI framework ensures buildings are not placed on impossible locations like cliff edges.

$$effectiveIncome = \min(metalIncome, energyIncome) \tag{4.1}$$

$$effectiveExpenditure = \min(metalExpenditure, energyExpenditure) \tag{4.2}$$

4.3.5 Recruitment Manager

The recruitment manager determines which units are created in factories as well as which factory type will be constructed next. This manager is smaller than the others, seeing as we did not get further than using a single factory type and the units it can produce. An important part of this manager is the code for deciding how many workers the team should have at any one time. To determine if the team needs another worker the equation 4.3 is run, which determines the maximum worker count from the effective income. The left side of the equation represents the cardinality of workers present at the time.

$$|workers| \leq \left\lfloor \frac{effectiveIncome}{4.5} \right\rfloor \tag{4.3}$$

4.3.6 Influence Manager

As part of the game interface, a set of simple influence maps were implemented which control the behaviour of soldiers and some of the leaf nodes used in the BTs. See related work for a definition of influence map.

In this project five influence maps are used; friendly, opponent, standard, tension and vulnerability. Friendly is the influence exerted by friendly units and conversely opponent is the influence exerted by non-friendly units. Standard influence is defined as $friendly - opponent$. Tension is $friendly + opponent$. The vulnerability map is calculated by subtracting the absolute value of a position in the standard influence map from the tension one, $tension - Abs(standardInfluence)$. The standard influence map is used to detect the borders between players, the area where their influence overlaps. The tension influence map is used to steer soldiers toward the areas of the map where there is most conflict, additionally it is used to decide which direction to build defence buildings in. The vulnerability map shows which parts of the map is most vulnerable for either team. It produces high values in locations where there is high tension but the sides are quite equal, and it produces low values where there is high tension but one side is superior to the other. This can used to both decide where to build more defences and where to attack.

A simple GUI was developed in order to make sure the influence manager work properly, see figure 4.3. The figure shows a standard influence map where green represents positive values and red negative values. As may be apparent, some units spread more influence than others, this is because a unit's influence value is determined by a unit power metric which describes a units overall strength. This metric is defined in the Zero-k framework and is based on a number of factors like health, cost, weapon damage and so on. Units like soldiers or defencive turrets which have a weapon range exert full influence within that area. The influence spread from a unit also dissipates at varying distances, this is called *falloff distance* and is determined by the movement speed of a unit.

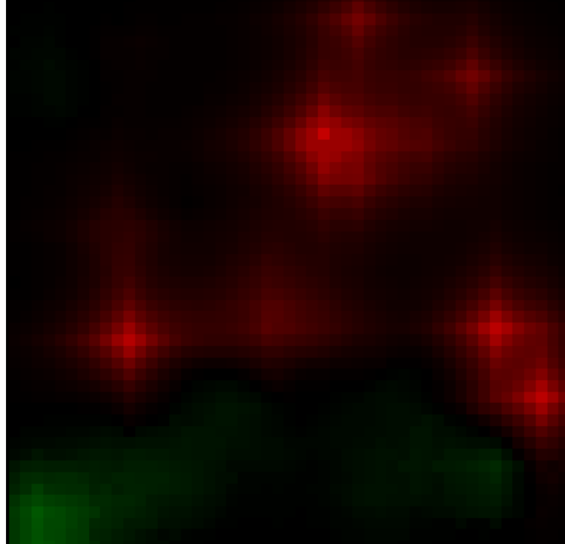


Figure 4.3: Graphical representation of the influence map implemented in ZKGPBTAI.

4.3.7 Military Manager

The military manager controls the movement and actions of soldiers as well as monitoring of enemy units. As described in 4.3.6, the movement of soldiers is determined by an influence map. However, not all soldiers follow the same goal. In order to cover more tactical options, the soldiers are split into squads which pursue different goals. Squad size is determined by taking the metal value of a squad, the sum of the metal value of every soldier in the squad, and if that number is higher than $effectiveIncome \times 60$ then the squad is of appropriate size. The number 60 was determined through experimentation. This has the effect of increasing the size of squads when the economy is doing well. Squads are rallied in a safe location, determined by influence map, until they reach full size at which time they are given orders to move to one of the areas of the map with the most tension. If a squad becomes too small due to loss of soldiers, it will be disbanded and the leftover soldiers transferred to a new squad.

The military manager constantly monitors enemy units. An important function of this monitoring is the identification of unit types. When an enemy unit is in radar range, but not LOS, it is not known what type of unit it is, it could be a building or a worker rather than a soldier. If one of these unknown units enter LOS it is identified and stored so that if it moves into the FOW we will still know what type it is. This also helps the influence manager calculate the threat

that a unit poses.

Though the military manager makes all the military decisions for this project, the modular approach taken in its implementation ensures easy integration for behaviour trees to control squads and make other military decision in the future.

4.4 Behaviour Tree

Most of the time spent on this project revolved around implementation. In order to lower the implementation-heavy threshold for future studies, a library for behaviour trees was implemented with the goal of making it as general and reusable as possible, thus providing a solid framework on which to base further research. There are multiple open-source BT libraries for game development and decision making, written in a variety of programming languages. However, none of the ones we are aware of are tailored to genetic programming by mapping the tree structures like the one developed for this project. Not all of the implemented features in the library were used in this project, and may therefore require additional testing, but the library is open-source and can easily be reused as well as supplemented with additional features, in order to support further projects with more functionality. This section covers the functionality of the library and a description of how it was used in this project.

4.4.1 ECJ Integration

Tailoring the behaviour tree library to support ECJ was fairly simple. `Task`, see figure 4.5, was made to extend the `GPNode`-class provided by ECJ, constraints were defined and provided a few supplementary methods were added. The internal control-flow nodes were implemented as *functions* while the leaf nodes were mapped to *terminals* in the GP syntax-tree.

As most control-flow nodes in behaviour trees behave like basic logic gates, the structural difference between a GP tree representing a multiplexer boolean function, often used for classification, and a fully functional behaviour tree is minimal. Even with GP trees representing regression problems, as in the left tree in figure 4.4 where it describes the function $(29 - \frac{X}{18}) + 7\cos Y$, there are significant similarities with BTs making them a good fit.

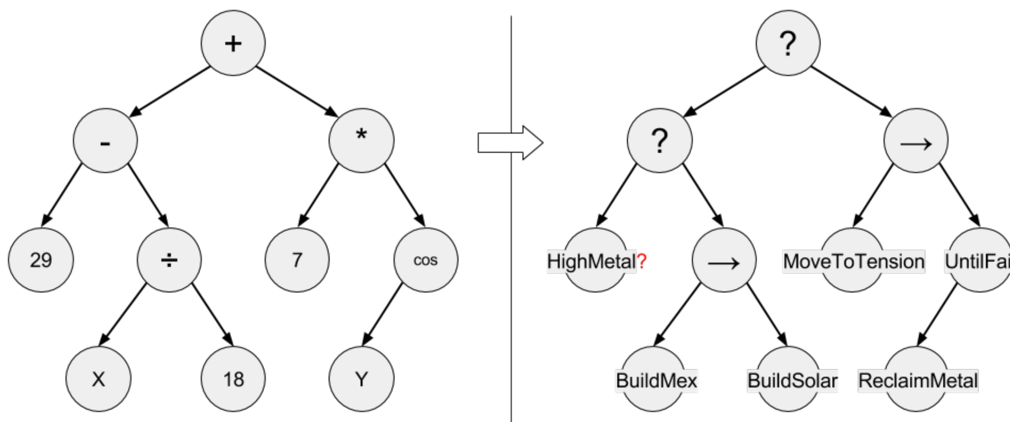


Figure 4.4: GP node mapping

The left tree in figure 4.4 shows a simple behaviour tree representing the subtask of acquiring metal in Zero-K. In the BT notation standard suggested by [Marzinotto et al., 2014], questionmarks and single right pointing arrows represent selector- and sequence nodes respectively. With this in mind, understanding the subtree behaviour is fairly straightforward. If the bot has a large amount of metal, it does nothing, propagating the success state to the top. If not, it will build a metal extractor and a solar panel to give it a power supply before returning success. If there are complications and the construction fails in some way, the top selector node supplies a fallback option, an alternate route which will also increase the metal stock. This route consists of moving to a place with high tension and reclaiming/extracting the metal from dead units in the area. See table 4.4.2 for details.

The most significant logical difference between trees in GP and behaviour trees is the meaning of the data propagating through the structure. In standard GP, the output of the root node is the the main end product of the tree. In the BT representation however, the data sent between nodes are mere guidelines for the order of execution and invocation of the nodes. By traversing the tree the desired result has already been achieved during the execution. Thus the output of the root node is redundant and can be disregarded.

ECJ generates string representations of the individual behaviour trees. A *tree interpreter* was created in order to instantiate valid BT data structures from this representation. The tree interpreter is included in the library and only needs the proper vocabulary, in the

form of a list of all implemented leaf classes, to function.

4.4.2 Implementation

The setup and general structure of the BT implementation draws inspiration from an article by Chris Simpson [Simpson, 2014] in addition to following the standards suggested in [Marzinotto et al., 2014]. Furthermore, some specifics were based on the Java behaviour tree implementation in the libGDX framework [Badlogicgames, 2016].

Using the generics and polymorphic capabilities of Java the library has a high degree of adaptability. The project is built upon layers of inheritance, where all nodes extend a common class with the exception of the GUI, the tree interpreter and a few utility classes. See figure 4.5.

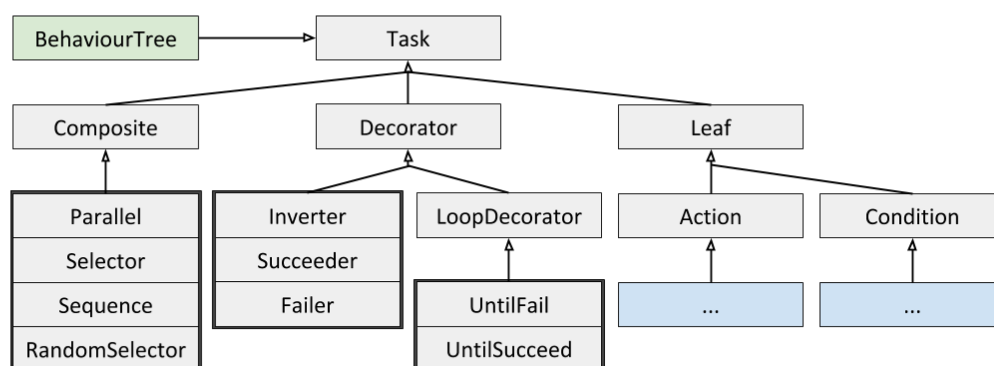


Figure 4.5: Class inheritance

The GUI allows for a graphical representation of the tree traversal by drawing the tree structure and colour-coding the nodes depending on their state in real-time, see figure 5.8. The interface has a tabbed layout for easy navigation between the tree instances.

Composite

This project uses selector and sequence nodes with two to five children each. Additionally a *random selector* was included in order to achieve less predictable behaviours. This node is fixed to three children and executes them in a random order. The selector, rather than the sequence, was chosen as base for this functionality since it often functions as a decider of what goals to pursue due to its properties.

Decorators

As far as behaviour trees go, this implementation uses a fairly standard set of decorator nodes to accompany the composites and terminals. There are three nodes for altering the results; an inverter which inverts the result, *failer* and *succeeder* which return failure and success respectively, no matter their input. Additionally, there are two repeater nodes *untilSucceed* and *untilFail*, both continuously executing until they receive desired results, success and failure respectively, see algorithm 4. When running the evolution many illogical and structurally flawed trees are created. Some of these will cause the repeater decorators to perform an infinitely long execution, causing overflow and excessive memory use. For this reason our implementation deviates slightly from the algorithm in question by not ticking itself when the result does not match the policy. It is rather ticked by the parent with a frequency, f_{tick}, to combat this issue.

Leaf nodes

Table 4.4.2 lists all problem specific leaf nodes created for this project. The nodes extend an abstract leaf class in the BT library, either *Action* or *Condition* marked as "type" in the table. These nodes represent the knowledge of the environment and the action domain for the behaviour tree, and inherently the evolution. Due to limitations in time and resources the number of terminal-nodes was restricted in order to conclude evolution within reasonable time.

Name	Type	Description
buildMex	A	finds the nearest available metal spot and builds metal extractor
buildSolar	A	builds a solar panel
buildRadar	A	builds a radar tower
buildGauss	A	builds a gauss turret towards tension
buildLotus	A	builds a lotus tower towards tension
buildCaretaker	A	builds a caretaker near a factory
buildFactory	A	builds a factory
buildStorage	A	builds a storage building
repairUnit	A	repairs buildings and units in the local area
reclaimMetal	A	reclaims rocks and wrecks in the local area
moveToMapCentre	A	moves half way to the centre of the map
moveToRandom	A	moves half way to a random location
moveToSafe	A	moves to a large concentration of friendly forces
moveToTension	A	moves to where there is most tension as per the tension map
highEnergy	C	over 90% of energy storage full and effective income is positive
highMetal	C	over 90% of metal storage full and effective income is positive
lowMetal	C	less than 10% of metal storage full and effective income is negative
lowEnergy	C	less than 10% of energy storage full and effective income is negative
highTension	C	true if the area around the unit has tension over 50%
closeToFactory	C	true if within line of sight of a friendly factory
enemyBuildingNear	C	true if within line of sight of a enemy factory building
inRadarRange	C	true if in friendly radar range
isAreaControlled	C	true if several towers in range and tension is low
lowHealth	C	true if less than 50% health
majorityOfMapVisible	C	true if more than 50% of map is directly visible
topOfHill	C	true if the surrounding elevation is slightly lower than at the position

Table 4.2: A list of leaf nodes that make up the evolved behaviour trees. A refers to action and C refers to condition

4.4.3 Variable Abstraction

It was a balancing point in this project to keep the tailoring of trees to a minimum, letting BTs evolve freely, while still being rigid enough to maintain a decent structure. The BT should to a high degree be a product of the evolution. This put a few restrictions on the behaviour tree implementation and called for the nodes to be virtually independent. All numeric variables like coordinates and quantities were abstracted away from the leaf nodes, which then only issued higher level commands of tasks to execute. If the node buildMex were to be called, see table 4.4.2, the BT would issue the command to the game interface, which would then provide all the details necessary for the action to be completed. The game interface calculates the coordinates for the nearest available metal spot, issues a move command, as well as a build command for the specified coordinates when the unit reaches it.

4.4.4 Execution

The tree is executed calling the *tick* function with frequency f_{tick}. In the implementation the parent node will always have control of which, if any, child is running. The game interface will only tick the root node which will then propagate through the chain of running children to the currently executing terminal. For all other nodes, the tick is divided into three disjoint methods; start, run and end. *Start* is executed when the node is *fresh*, meaning that it has not been executed before in the current tree traversal. Upon completion of the traversal, all nodes are reset to fresh in preparation for a new behaviour search. Most nodes ignore the start method, though the `action` nodes use it generously. They often have complex prerequisites and calculations that have to be included in the command before actually executing it, e.g. determining coordinates. The *end* method is called when the game interface, observed by the action node, flags the command completed. Some action-implementations will use this to finalise the execution. Apart from this the *run* method behaves like the tick function, by issuing commands, invoking children or checking the condition, and then returning the state of the node.

Algorithm 1: Sequence

```
1  for i ← 0 to |children| do
2      state ← tick(child[i]);
3      if state = Running then
4          return Running;
5      end
6      if state = Failure then
7          return Failure;
8      end
9  end
10 return Success;
```

Algorithm 2: Parallel

```
1  for i ← 0 to |children| do
2      state_i ← tick(child[i]);
3  end
4  if n_Success ≥ S then
5      return Success;
6  end
7  if n_Failure ≥ F then
8      return Failure;
9  end
10 return Running;
```

Algorithm 3: Selector

```
1  for i ← 0 to |children| do
2      state ← tick(child[i]);
3      if state = Running then
4          return Running;
5      end
6      if state = Success then
7          return Success;
8      end
9  end
10 return Failure;
```

Algorithm 4: LoopDecorator

```
1  state ← tick(child);
2  if state = Running then
3      return Running;
4  else
5      if policy ≠ state then
6          tick(child);
7      end
8  end
9  return Success;
```

Algorithm 5: Root

```
1  return tick(child[0]);
```

Algorithm 1-5: In pseudocode, the algorithms for the node execution. The variables S, F and *policy* are node parameters representing the number of required successful child-nodes, maximum limit of failed child-nodes and the node policy respectively.

4.5 GP

This section details how genetic programming was setup using ECJ, how some of the GP specifics were implemented and how it was then connected to the other two components.

4.5.1 ECJ

ECJ is a Java-based evolutionary computation research system mainly developed by Sean Luke[Luke et al., 2006]. It was designed for heavy-weight experiments and provides tools for many of the common EC algorithms, with a particular emphasis on GP.

The state of an evolutionary run is stored in a single instance of a subclass of EvolutionState. There are two versions of EvolutionState; generational and steady-state. Generational produces a new population every generation whereas steady-state replaces individuals from a constant, static population. Figure 4.6 illustrates the top-level loop of the generational version of EvolutionState, which is the one used in this project. The loop iterates between breeding and evaluation, including an optional exchange period after each. Statistics is called before and after each period of breeding, evaluation, exchange as well as prior to and after initialisation of population and when finishing, when a cleanup is performed prior to exiting the program. All of these processes are implemented as singleton objects.

In order to set up a GP run with ECJ, one provides a parameter file and implements a subclass of ec.SimpleProblemForm, which represents the task. The parameter file defines which implementations of the evolutionary processes will be used, for example the choice between SimpleBreeder and MultiBreedingPipeline. Additionally, the parameter file also contains all options to do with the run, options like how many generations are to be used, population size, mutation rate and so on. The purpose of implementing a subclass of SimpleProblemForm is defining how evaluation will be performed for the experiment at hand. In this project the subclass contains code that runs a game of Zero-K by passing an individual to the game-adapter and then listening to the output in order to determine the fitness of the individual. See 5.1 for details on the parameters used in this project.

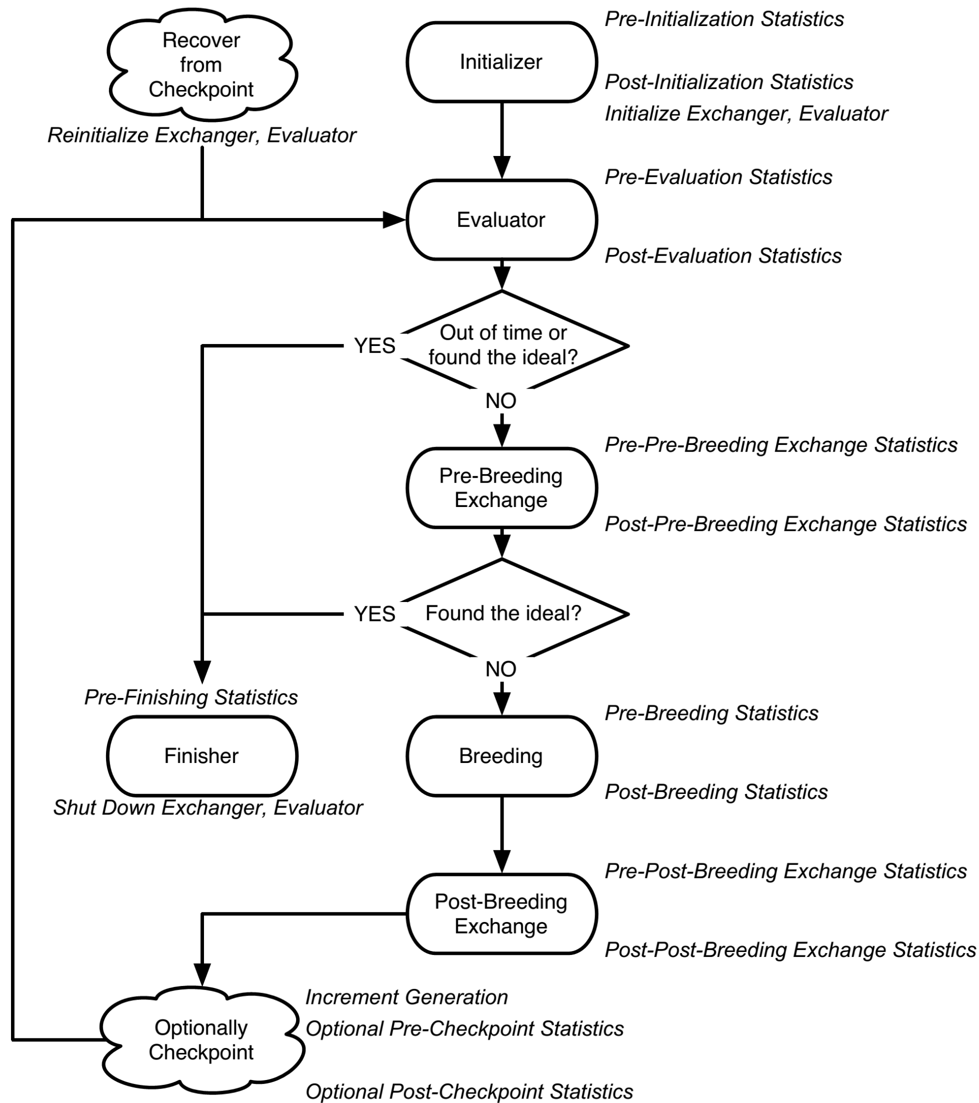


Figure 4.6: An overview illustrating the program-flow of ECJ.

4.5.2 Fitness

As discussed in the background section fitness is a measure of how good a solution is at solving the task at hand. In this project the fitness measure is a number between 0 and 1. In the conducted experiments the goal was to evolve a BT that mainly focused on economy as the military aspects of the game were not directly under its control. This is reflected in the fitness function. Forty percent of the total fitness is directly derived from how well the bot did economically, specifically how high the average income 4.7, the highest income peak 4.5 and how many metal spots on the map was under the bots control on average 4.4. In the two former, the highest observed value was used as the denominator in their respective equation.

The BT also controls the construction of factories and defences and because of this it needs some measure of military success in order to not neglect that aspect of the game. This prompted the addition of 4.6 which measures how much metal resource the opponent has lost in the form of units and buildings divided by how much metal resource the BT has spent. This accounts for ten percent of the fitness. The final part of the fitness function is the reward for winning the game 4.8 which either grants 0.5 fitness for victory or 0 for a loss. The total fitness equation is the weighted sum of the fitnesses mentioned until now 4.9.

$$averageMex = \frac{avgMexSpotsControlledThroughoutGame}{totalMexSpotsOnMap} \tag{4.4}$$

$$peakIncome = \frac{highestIncome}{50} \tag{4.5}$$

$$killVSExpenditureInMetal = \frac{enemiesKilledInMetalValue}{totalExpenditureInMetal} \tag{4.6}$$

$$averageEco = \frac{averageEffectiveIncome}{50} \tag{4.7}$$

$$victoryReward = \begin{cases} 1 & if\ victory\ achieved \\ 0 & otherwise \end{cases} \tag{4.8}$$

$$\begin{aligned} fitness =& averageMex \times 0.10 + averageEco \times 0.25 + peakIncome \times 0.05 \\ &+ killVSExpenditureInMetal \times 0.10 + victoryReward \times 0.50 \end{aligned} \tag{4.9}$$

4.5.3 SpringProblem

The implementation of SimpleProblemForm used in this project has several functions; it writes individuals to files which can be read by the game interface, it starts an evaluation by calling the game interface and it interprets the results retrieved from the output provided by the game interface to determine the fitness of the individual being evaluated. The game interface writes the fitness of individuals to the game chat every 2000th frame which is then be processed by SpringProblem when the match is over.

Because of evaluations potentially taking very long time, as may be the case if neither combatant builds anything military, functionality to stop an evaluation after a specified time was developed. When an evaluation is started, the EvolutionRunner no longer has programmatic access to that process. In order to solve this a script that ends all Zero-K processes is run when the time is up. When this happens, the fitness of the individual being evaluated is set to be the latest value written to the game chat before it was closed down. An unfortunate effect of this is that parallel runs are not possible as this method would kill all Zero-K processes.

4.5.4 CompetitiveSpringProblem

In the case of competitive coevolution CompetitiveSpringProblem is used rather than SpringProblem. It is an implementation of `GroupedProblemForm`, which means it considers more than one individual at the same time. Often, only one individual is evaluated at a time in competitive coevolution, however, due to time constraints we decided that each evaluation will count for both of the competing individuals. This means that in a competition between individual a and b, the result will count towards the fitness of both. For example, if a wins then it will receive a fitness score of 1 whereas b will receive a score of 0.

In this project single-elimination tournament is used to assign fitness to individuals when running competitive coevolution. Each battle of the tournament is run up to three times; if a individual wins two times, it is declared the winner. If there is no definite winner after three games a random winner is picked.

4.5.5 Statistics and Logs

In order to preserve as much information as possible about each run, a checkpoint is created after each generation. A checkpoint is a snapshot of all data relating to a run, which can be used to resume a run. These are used as backup in case a computer should crash, but are also useful to go back and look at an especially interesting time frame of an evolution. The data stored in the checkpoints is not human-readable and because of this we also store all generated individuals, their fitnesses and meta information, for example the best individual in each generation, in a separate file. Additionally, this data is used to generate two graphs

which we use to determine the effectiveness of a set of GP parameters. The first graph plots average fitness, best fitness and normalised size for every generation, see 5.1. The second is a bar chart which illustrates how often the various node types are used, which was very useful to determine if a specific node should be included or not.

4.5.6 Selection

All five experiments use tournament selection. There are three reasons that tournament selection was chosen for this project. Firstly, it is easy to adjust the selection pressure by changing the tournament size. Secondly, the selection pressure on the population remains constant because tournament selection does not take into account how much better an individual is over another, only that it is better, which has the effect of automatically rescaling fitness. Thirdly, because an individual only has to be slightly better than the others in the tournament, small differences are amplified which is important in this project because the individuals are evaluated against CAI which is a very strong opponent, especially early in the run. Through experimentation it was determined that a tournament size of seven was appropriate for this project. Tournament selection is used to decide which individuals we will apply genetic operators to. This project used the implementation of tournament selection provided in ECJ, `ec.select.TournamentSelection`.

4.5.7 Genetic Operators

In this project we employed three forms of genetic operators; subtree crossover, reproduction and mutation. First, selection is used to determine which individuals genetic operators will be applied to, then a genetic operator is chosen using the probabilities described in the parameter file, thirdly, Koza node selector is used to determine which node or nodes the genetic operator will be used on and lastly the genetic operator is applied. *Koza node selector* is a method for picking nodes from trees laid out in [Koza, 1992]. The method divides the nodes that are possible to pick into four probability areas: terminal, non-terminal, root node and random node. In this project only terminal and non-terminal were used, which means that the root node cannot be mutated nor used in crossover.

Mutation is performed using the mutate one node method as outlined in [Chellapilla, 1998] and implemented by Sean Luke in ECJ. It selects a node using Koza Node Selector and replaces it with another randomly chosen node of the same arity. This means that the original topological structure will be the same but that specific node will have changed.

Subtree crossover is performed using ECJ's strongly-typed interpretation of Koza's sub-crossover technique. First, two individuals are selected, then a node is selected from each using Koza Node Selector such that the return type in one tree is type-compatible with the argument type of the parent in the other tree. Then a check is run to see if the derived tree will violate the depth restraints. If it does then the process is repeated. The variable *tries* in the parameter file determines how many times the process will repeat in case of failure.

4.5.8 Strong Typing

We discovered early on that evolution often yielded BTs that contained contradictions that made the trees highly inefficient. For example the placement of a succeeder, which always returns true, as a child to an untilFail node, creates an infinite loop. Using the functionality for strong typing built into ECJ it was possible to remove situations like this. Practically, this was accomplished by defining a list of types as in 4.5.8, then a list of sets like 4.5.8, which are groups of types, which can be used the same way as types. For example *allExceptAction* which, as the name implies, contains all types except action. Following that, we define a list of constraints, which can be better described as definitions of types of nodes. These node constraints contain the return type of the node, how many arguments it takes and what type those arguments have, see 4.5.8. Finally we connect the type of a node with the path to the Java implementation of that node, see 4.5.8. For a complete list of all strong typing employed in this project see 4.5.8.

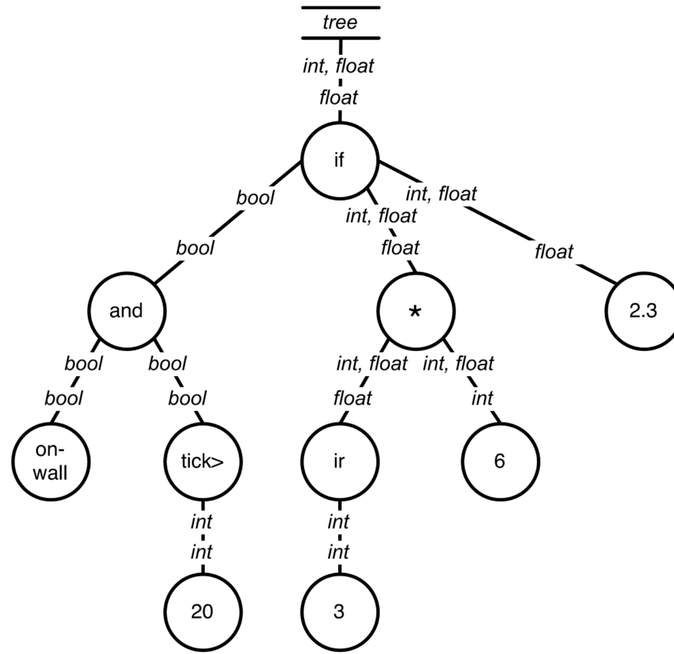


Figure 4.7: An illustration of how strong typing works in ECJ.

Types

Explanation	Code
size	`3`
type0	`action`
type1	`leaf`
type2	`inverter`

Table 4.3: Example of how types are defined in ECJ parameter files.

Sets

Explanation	Code
size	`2`
set0 name	`any`
set0 size	`3`
set0 member0	`action`
set0 member1	`leaf`
set0 member2	`inverter`
set1 name	`allExceptInverter`
set1 size	`2`
set1 member0	`action`
set1 member1	`leaf`

Table 4.4: Example of how sets are defined in ECJ parameter files.

Node Constraints

Explanation	Code
size	`3`

name
return type
size
	`nc0_action`
	`action`
	`0`

name
return type
size
	`nc0_condition`
	`condition`
	`0`

name
return type
Number of children
child0
	`nc1_inverter`
	`inverter`
	`1`
	`allExceptFailer`

Table 4.5: Example of how constraints are defined in ECJ parameter files.

Node Definition

Explanation	Code
implementation type	`bt.leaf.action` `nc0_action`
implementation type	`bt.leaf.condition` `nc0_condition`
implementation type	`bt.decorator.inverter` `nc1_allExceptInverter`

Table 4.6: Example of how node definitions are defined in ECJ parameter files.

Parent	**Child**	*Reason*
UntilSucceed	UntilSucceed	*redundancy*
UntilSucceed	UntilFail	*redundancy*
UntilFail	UntilSucceed	*infinite loop*
Succeeder	Succeeder	*redundancy*
UntilFail	Succeeder	*infinite loop*
UntilSucceed	Failer	*infinite loop*
UntilFail	UntilFail	*infinite loop*
Inverter	Inverter	*cancels each other*
Failer	Failer	*redundancy*

Table 4.7: A list of all the strong typing used in this project.

4.6 Environment

This project requires a fair amount of work to set up and get running. It contains multiple applications that need to communicate and execute in parallel. This section will describe the data flow of the overall system and how all the different parts communicate.

4.6.1 Information Flow

A GP run in the system implemented for this project is started from EvolutionRunner, see figure 4.8. The main class retrieves a specified parameter file and uses it as an argument to start ECJ. Most aspects of the run are handled by ECJ according to the stated parameters. ECJ uses an implementation of `SimpleProblemForm` or `GroupedProblemForm`, in this case *SpringProblem* or *SpringProblemCompetitive*, to evaluate individuals. In order to evaluate an individual, a Zero-k match is run using ZKGPBTAI. The result of the run is stored in a log which is interpreted by SpringProblem to determine fitness.

To run a Zero-k match, SpringProblem first writes a file containing the individual in string form. It then spawns a command line process and executes a script which starts Zero-k with settings designed to minimise game-play time and computation costs. Among these, a setting that enables running Zero-k without graphics, significantly reducing computation. These settings are retrieved from a specified setup file. The setup file also defines which map and which players that will be used for the match. Bots are added to the match; one *ZKGPBTAI* instance and one instance of CAI. In case of competitive coevolution there are two ZKGPBTAI instances. Individuals are read from the file written by SpringProblem and are then used to generate behaviour trees using the BehaviourTreeECJ library. Periodically throughout the game it ticks the BT to make a search through the structure to find an action to execute.

Zero-K is set to a high verbose level with additional information conveyed from the bot though the game to the command line. All this information is gathered by SpringProblem which analyses the game and calculates the fitness of the individual when completed. When multiple games are played, the fitness returned will be the mean across all games. After setting the fitness, ECJ continues this cycle for all individuals in need of a fitness calculation throughout the run. When

the last generation has been evaluated and the GP run is concluded, graphs are generated and detailed information including checkpoints are stored in folder named after the seed used in the run. A lot of tailoring was required to get the different parts of the system to communicate properly. Figure 4.8 displays a very simplified diagram of the different components making up the project.

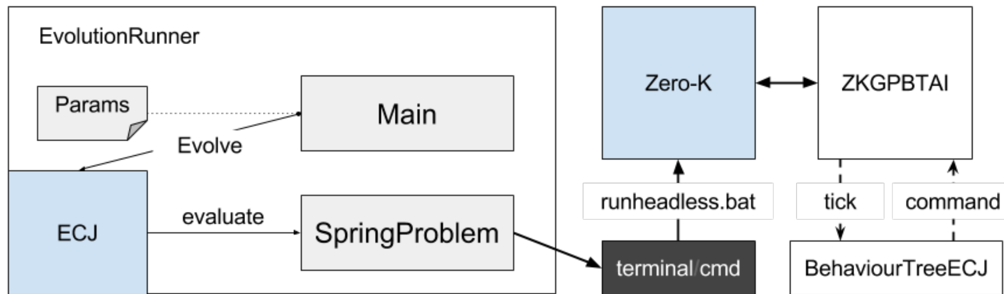


Figure 4.8: System environment overview

4.7 Summary

Throughout this chapter the most essential features of the system have been described while leaving out implementation details. The various components that make up the system are allocated to sections in the chapter, starting with the RTS game Zero-k. Zero-k is built upon the Spring Engine which provides an AI interface for developing bots in different languages. Our implementation is written in Java, section 4.3 describes how it was developed and the techniques that were employed. This game interface functions as the base and connection point for the behaviour trees outlined in section 4.4. The BTs make decisions and issue commands for the game interface to execute. It is behaviour trees which are the subject of the evolution in this project. The BTs are evolved using genetic programming which is covered in section 4.5. Through strong typing and use of the framework provided by ECJ, GP was set up with a variety of settings and parameters for the different experiments. These experiments are described in the subsequent chapter. A large amount of implementation was needed to complete the system, and specifics have been left out of this thesis. Most of this information are available through the public Github repositories found in appendix A.1 and should include nearly everything needed to

recreate the experiments or expand the research with new or extended experiments.

Chapter 5

Results and Discussion

In this chapter we describe and discuss the conducted experiments. Section 5.1 describes the experiments and the results they provided. Section 5.2 outlines tests performed in order to compare the experiments. In section 5.3 we discuss and explain the findings in the previous two sections. Lastly, section 5.4 provides a summary of this chapter.

5.1 Experiments

In this section we describe why and how the experiments were conducted as well as the results obtained. Due to time-constraints, the number of runs for each experiment was limited. This means that the statistical significance of the experiments is fairly low, therefore, hypotheses rather than conclusions can be drawn from the results. In addition to descriptions of the experiments there is a subsection covering a hand-written bot referred to as H_1.

5.1.1 Experiment Overview

In the course of this project five experiments, referred to as E_1 to E_5, were conducted. The first two experiments, E_1 and E_2, used the genetic operators crossover and reproduction whereas E_3, E_4 and E_5 used crossover and mutation. The reasoning behind this was that [Koza, 1992, 1994] argue that mutation is not necessary to obtain good results with GP. However, most of the projects reviewed in section 3.2.3 do use mutation. Therefore, we decided to investigate both approaches.

In addition to exploring the use of different genetic operators, we experimented with population size and the amount of generations. It has

been argued that in some cases small populations with many generations can outperform large populations with fewer generations [Gathercole et al., 1997; Fuchs, 1999]. In order to explore both options, E_1 and E_3 had small populations with many generations whereas E_2 and E_4 had larger populations with fewer generations. All in all, experiment E_1 and E_3 have a total of 2000 evaluations whereas E_2 and E_4 have 2400. See table 5.1 for an overview of the first four experiments. The effect of having different population sizes for the experiments will be discussed in section 5.3.4.

E_5 uses competitive coevolution and the genetic operators crossover and mutation. It was decided to use mutation rather than reproduction because the best individual found after the first four experiments came from an experiment with mutation.

	No Mutation	Mutation
Small Population(20)	E_1	E_3
Large Population(50)	E_2	E_4

Table 5.1: Experiment description matrix for experiments one through four.

5.1.2 General Parameters

Many of the parameters used in the experiments are the same, at least across the first four. Behaviour trees that are generated through initialisation or genetic operators may not be deeper than 17. This is the standard max depth used in ECJ in order to reduce the computational load a deep tree may create. As discussed in section 4.5, tournament selection is used to choose which nodes genetic operators will be applied to. Tournament selection used a tournament size of seven which was determined to be appropriate through experimentation. The initialisation method employed is Koza's ramped-half-and-half which is described in section 2.4. The initial trees may not have a smaller depth than three nor a larger depth than six. These values were determined to yield a good variety of trees by studying the generated initial populations. When choosing which node in a tree to apply a genetic operator to, Koza node selector will choose terminals 20% of the time and non-terminals for the rest. The genetic operators use tournament selection to choose parents and Koza node selector to pick nodes. For a summary of the parameters see table 5.2.

Max Depth of Tree	17 nodes
Selection Method	Tournament
Tournament Size	7
Initialisation Method	Ramped-Half-and-Half M in-Depth: 3 Max-Depth: 6 Node Selection of Terminals: 0.2 Node Selection of non-Terminals: 0.8
Genetic Operators	
Crossover	Selects parents using Tournament Selection Selects nodes using Koza node selector
Reproduction	Selects individual using Tournament Selection
Mutation	Selects individual using Tournament Selection Selects node using Koza node selector Performs mutation using ECJs MutateOneNodePipeline

Table 5.2: General Parameters used for all experiments.

5.1.3 Experiments 1: Small Population, no Mutation

E_1 has a population size of 20 and evolves for 100 generations. It uses crossover 85% of the time and reproduction for the rest. See table 5.1.3 for an overview.

Evaluation	Fitness function and competition vs CAI
Population size	20
Generations	100
Evaluations per individual	2
Time limit	15 minutes
Genetic Operators	
	Crossover: 0.85
	Reproduction: 0.15

Table 5.3: E_1 (Small Population, no Mutation) parameters.

Analysing the nodes and structures of the individuals for the runs performed give clear indication of low genetic diversity. As a run progresses towards generation 100 the evolved individuals become increasingly similar, until there are only minor differences separating them. There is little similarity between the runs, not indicative of any general tendencies.

The best run from the experiment is plotted in figure 5.1. The average fitness of the population, represented in red, rises slightly the

first third of the run then lies quite steady just below 0.2. Best fitness, the green line, indicates the best individual found in each generation. In this run, best fitness rises and falls sporadically, probably because the best individual is subject to deleterious crossover operations. For example, if the good features of an individual are moved to an inactive branch of a tree through crossover the fitness will fall, however, in the next generation the opposite might occur resulting in a good individual. The average size, represented by the blue line, increases steadily until generation 75 when it rises quickly then falls down to a low value. The reason for this peak is that the best individual tripled in size and because it is the best individual it will most likely contribute a lot to the next generation. Additionally, because it is large there is a higher chance for crossover to pick a large subtree meaning that there is a higher probability of large individuals to be generation for the next generation. In generation 79, the best individual had a significantly lower size than the one before, this gradually propagates into the rest of the population and by generation 83 the trend has turned and average population size is on the decline.

The best individual from the run displayed in figure 5.1, has a fairly limited behaviour spectrum. Early in the game it looks promising, building a balanced and strong economy. The problem however is that it focuses too much on economy. It constructs no defencive buildings, factories or caretakers making it a easy target for raids and limits the unit-production. The tree is fairly large, but only a minor part of the full tree is ever invoked with the majority being unreachable in actual game-play. The other top individuals from this experiment suffer from the same problem, namely not progressing after the initial economic build up.

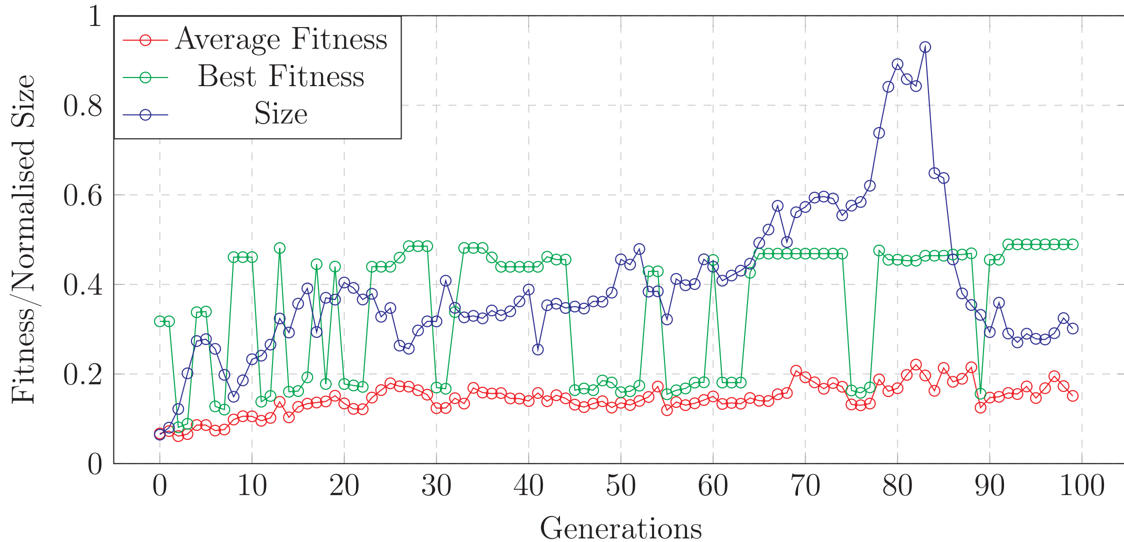


Figure 5.1: The best run from E_1 (Small Population, no Mutation)

5.1.4 Experiment 2: Large Population, no Mutation

Experiment two has a population size of 50 and evolves for 50 generations. It uses crossover 85% of the time and reproduction for the rest. See table 5.4 for an overview.

Evaluation	`Fitness function and competition vs CAI`
Population size	`50`
Generations	`50`
Evaluations per individual	`2`
Time limit	`15 minutes`
Genetic Operators	
	`Crossover: 0.85`
	`Reproduction: 0.15`

Table 5.4: E_2 (Large Population, no Mutation) parameters

E_2 shows some promise after the three runs, achieving a best fitness of 0.89. From observing the results, the larger population seems to provide the crossover algorithm with a more varied gene pool and the decrease in average fitness growth is not as pronounced. The largest area of growth, often including the best fitness of the run, occurs around a third of the way through the runs.

The best run, see figure 5.1.4, produced a interesting graph which contrasts with the one from E_1. Very little happens in terms of fitness before generation 15 when best fitness rises to a high plateau and stays there the rest of the run. Average fitness follows suit and stays around

0.4 which is very good as any number above 0.5 would indicate a victory against CAI. The only significant point to mention about the average size is that it moves from a low value to a high one when the best fitness peak happens and then it mostly stays between 0.6 and 0.8.

The best individual from this run shows promise during gameplay as well. Similarly to the highlighted individual from E_1, it focuses heavily on the economy, but the lack of defencive buildings leave it vulnerable. If the bot does not get harassed by the opponent during the economic boom, it will perform well, building caretakers and produce a large army. However, it does not construct any radars, and locating the enemy becomes an issue. The bot has to rely on the enemy units leading it to their base. This naive strategy does pay off occasionally given the mentioned favourable circumstances, which probably have originated the high fitness, but in the majority of games the opponent will exploit the fragile setup yielding a clear defeat.

fFigure 5.2, shows an individual with a very elegant structure. The BT is divided into two clearly separated branches with fairly different goal-oriented behaviours. What goal to pursue is decided by the node `highMetal`. The up-most right part of the tree has a clear goal for economic growth and is active when `highMetal` is not fulfilled. The other side of this condition generally contains more advanced behaviour, including a high number of expensive buildings and complex strategies. This structure provides more reactive behaviour compared to the handwritten BT which with its more iterative conditions seems to follow an algorithm and does not react to the environment to the same extent. An interesting point is that the evolved BT uses the *highMetal* conditional node to divide between resource providing and resource draining action nodes. This is similar to how we used the *highMetal* node when creating hand-written behaviour trees.

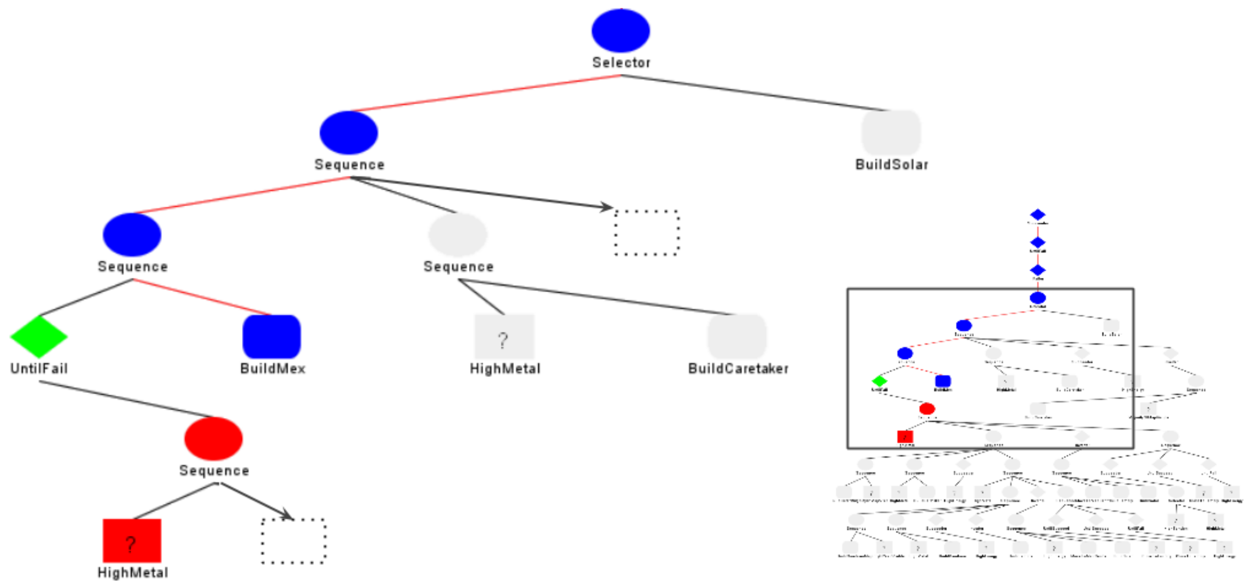


Figure 5.2: The best individual from E_2 (Large Population, no Mutation)

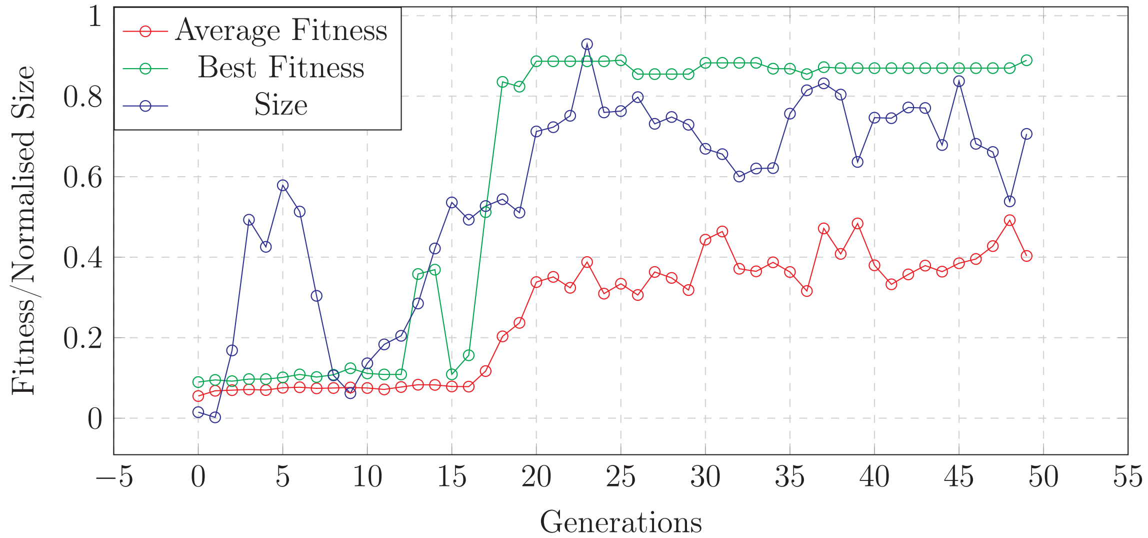


Figure 5.3: The best run from E_2 (Large Population, no Mutation)

5.1.5 Experiment 3: Small Population, Mutation

Experiment three has a population size of 20 and evolves for 100 generations. It uses crossover 98% of the time and mutation for the rest. See table 5.1.5 for an overview.

Evaluation	`Fitness function and competition vs CAI`
Population size	`20`
Generations	`100`
Evaluations per individual	`2`
Time limit	`15 minutes`
Genetic Operators	
	`Crossover: 0.98`
	`Mutation: 0.02`

Table 5.5: E_3(Small Population, Mutation) parameters

Of all experiments, E_3 is the one which shows the clearest signs of stagnation. Similarly to E_1 the genetic diversity stagnates significantly towards the end of the evolution for all three runs. Both the average fitness and best fitness across all three runs are very similar to E_1, displaying the same tendencies, see figure 5.5 and 5.1. The run that produced the best individual was the one with the lowest measured diversity across all experiments. Both program variety and subtree variety, see subsection 5.3.2, reached lows far below the ideal, see figure 5.4.

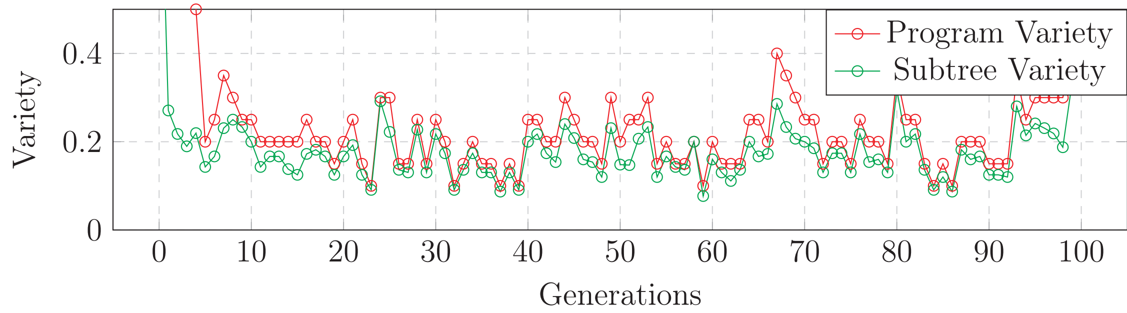


Figure 5.4: Program and subtree variety of E_3(Small Population, Mutation)

The best fitness of each generation in this run rises and falls sporadically, similarly to the highlighted run from E_1. The fitness oscillates even though the best individual remains constant. The likely explanation for these fluctuations is the highly stochastic nature of most aspects of the game environment. Certain decisions might cause the game to take a vastly different direction given the same preconditions, as there are a lot of uncertainties to take into account. This is why each individual should play a higher amount of games each evaluation to get a more accurate and probable fitness value for the individuals. This would also decrease the bad individuals with good fortune dominating the evolution, with this run proving a prime example. The goal is to evolve a versatile AI entity with a high degree of adaptability. Having individuals progressing due to piggybacking on good fortune will not help reach this goal.

For the general game-play, the individuals again lack middle and endgame strategies only focusing on building economy before getting beaten fairly fast with little resistance by the opponent (CAI). Some of the individuals however build more advanced buildings, mainly caretakers, at fairly appropriate times, something not seen in the selected individuals from E1. This may be a coincidence since we have no greater statistical data to draw any meaningful connections from.

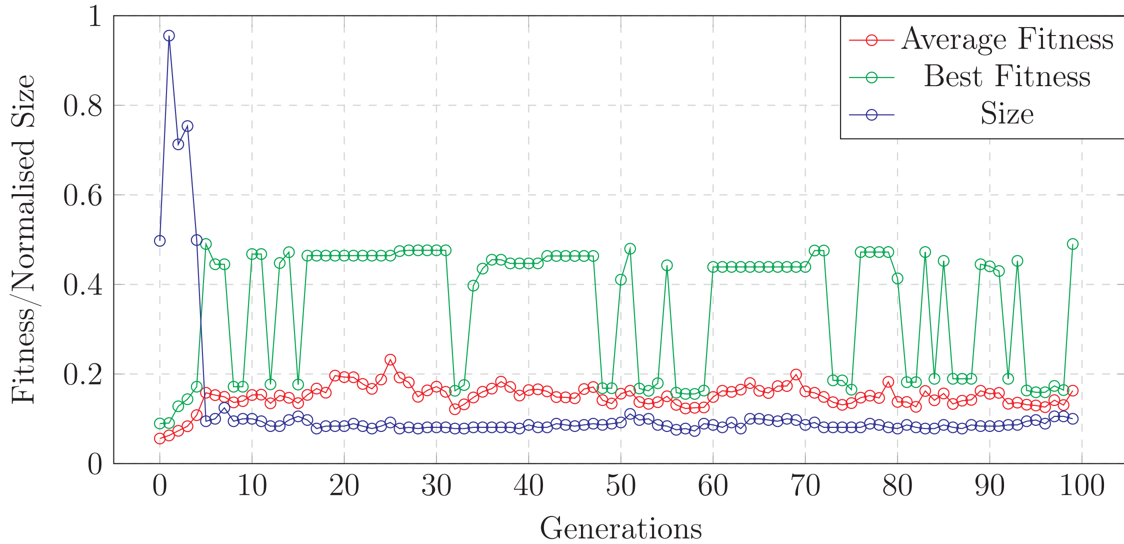


Figure 5.5: The best run from E_3(Small Population, Mutation)

5.1.6 Experiment 4: Large Population, Mutation

E_4 has a population size of 50 and evolves for 50 generations. It uses crossover 98% of the time and mutation for the rest. See table 5.1.6 for an overview.

Evaluation	`Fitness function and competition vs CAI`
Population size	`50`
Generations	`50`
Evaluations per individual	`2`
Time limit	`15 minutes`
Genetic Operators	
	`Crossover: 0.98`
	`Mutation: 0.02`

Table 5.6: E_4(Large Population, Mutation) parameters.

E_4 achieved a top score of 0.84 which, while not as good as E_2, is a very good score. The three runs were quite varied, two of them showed similar plateau behaviour as described in experiment 2, while the third stagnates only a few generations in at around 0.45 with only small oscillation in fitness throughout. The runs generally had a good amount of genetic diversity with few identical individuals and a high amount of unique subtrees each generations.

The best run, see figure 5.1.6, is one of the more interesting ones observed. Average fitness constantly rises throughout the run, evenly in the first half and more sporadically in the second. Best fitness moves

sporadically in the first quarter then stabilises around 0.4 before jumping up to around 0.8 and eventually, stabilises in the last quarter. Average size moves a lot in the first half then rises constantly in the second half.

The best individual from experiment four shows some interesting characteristics. It focuses solely on economy in the first part of the game then begins building caretakers and radars once the *highMetal* and *highEnergy* nodes succeed. The combination of caretakers and radars is quite effective as the caretakers ensure a higher unit production rate and the radars gives the bot more information about enemy locations, which means that the bot has a lot of soldiers and knows where to send them. This bot is however vulnerable to raiding attacks in the beginning of the game as it does not build any defences. Another interesting point is that subsequent to building a caretaker, which is usually built in a safe location, workers active the node *moveToTension* which sends them to the front, where there is the most conflict.

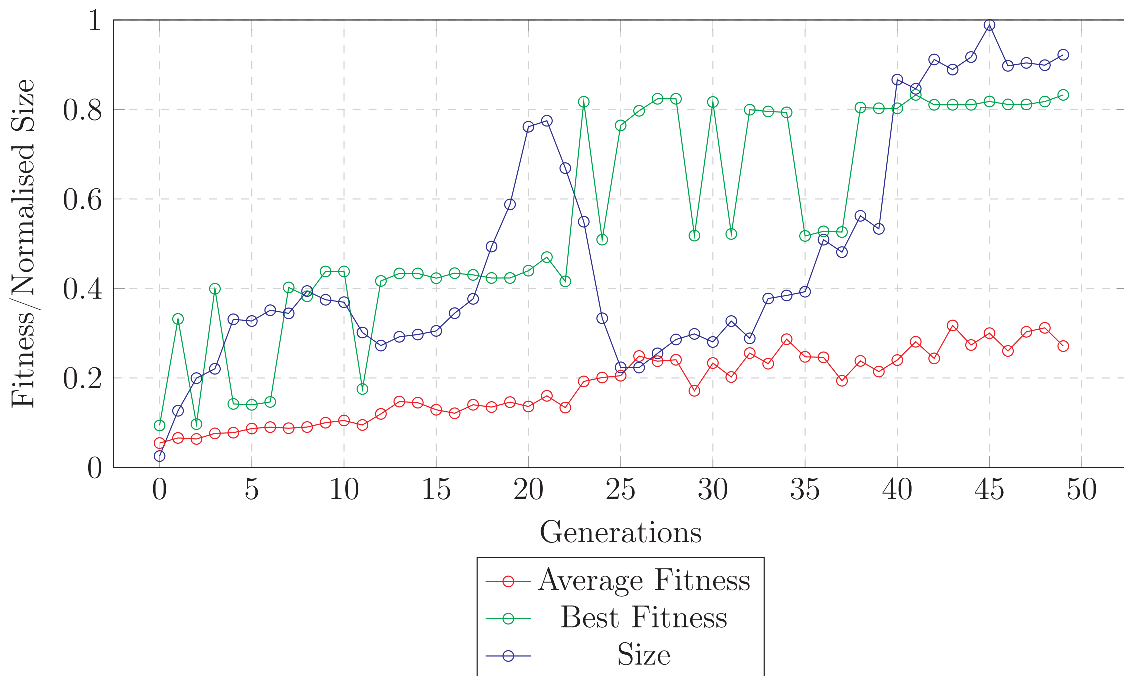


Figure 5.6: Best run from E_4(Large Population, Mutation).

5.1.7 Experiment 5: Competitive Coevolution

Competitive coevolution was included as one of the five experiments because it has been shown to do well for competitive games, see [Rosin

and Belew, 1995], and several of the related projects described it as future work. In the previous four experiments, the fitness of evolving bots was determined by how well it performed in a match against CAI. This may have had a detrimental effect on the evolution as small improvements might not register in the fitness score because CAI is a too strong opponent. As described in section 2.4.9, competitive coevolution works by evaluating individuals against each other rather than against a fixed opponent. This has the effect of making the entire population evolve in smaller increments as every little change might give a single individual an advantage over the others.

The most common way of setting up competitive coevolution is with two populations, where one population is evaluated against the other, resulting in an arms race where both populations try to evolve faster than the other. However, because of computational limits we opted for a single population setup in this experiment. To further reduce computation we chose single elimination tournament, which has a time complexity of $O(n)$, over methods like *round robin*, where all individuals are tested against all individuals, which has a time complexity of $O(n^2)$. Because the first four experiments were conducted with one bot facing off against CAI, a tournament size of two was chosen for E_5. In order to make the tournament as fair as possible, a population size of 16 was chosen, which means the tournament tree forms a perfect binary tree [Zou and Black, 2008], giving all individuals an equal chance. In order to reduce computation further both individuals participating in a battle are evaluated on their performance simultaneously. Each pairing is evaluated up to three times to ensure there is always a clear winner. Because many of the individuals in the beginning of a run are so poor, a time limit of 250 seconds was added, which means that when a battle has run for 250 seconds the individual with the highest fitness is chosen as the winner. In the unlikely case of a tie, the winner is chosen randomly.

Additionally, we sought to remedy some of the problems discovered in earlier experiments. To this end the tries parameter was increased from one to ten, elitism with one individual was added and mutation was run with a higher rate. Explanations of the problems referred to here will be found in the discussion section, see 5.3.

Evaluation	`Single Elimination Tournament`
Population size	`16`
Generations	`50`
Tournament size	`2`
Max evaluations per grouping	`3`
Tie breaks	`Fitness function from the other experiments`
Time limit	`250 seconds`
Tries	`10`
Elitism	`1`
Genetic Operators	
	`Crossover: 0.95`
	`Mutation: 0.05`

Table 5.7: E_5(Competitive Coevolution) parameters.

We logged data from competitive coevolution runs differently compared with the other experiments. The average fitness of a population in a single-elimination system is constant; there are always the same amount of individuals with each ranking due to the nature of the algorithm. This means that it is more difficult to present the results in a meaningful way. It was decided to evaluate the best individual from every second generation against CAI. This is however, not optimal as the BTs were evolved in competition against each other, not CAI, but it simplifies comparisons with the other experiments. Each of the evaluation against CAI were run once.

In figure 5.7, the best individuals from all three runs are plotted. When compared with the plots of the other experiments, these runs do not look very successful. However, when run against other evolved bots rather than CAI, they do a lot better. This will be discussed in further detail in subsections 5.2.1 and 5.3.7.

The best individual from E_5, run 2 in figure 5.7, achieved a low score when evaluated against CAI, however, it had some interesting characteristics. In the beginning of the game it focuses on economy, building metal extractors and solar panels but in addition to this it uses the reclaim action. Furthermore, some workers were observed using reclaim as their first action after being recruited which indicates that there is a conditional node high up in the tree, which when fulfilled calls a reclaim node. Another interesting point is that it is the only observed evolved bot which builds the storage building.

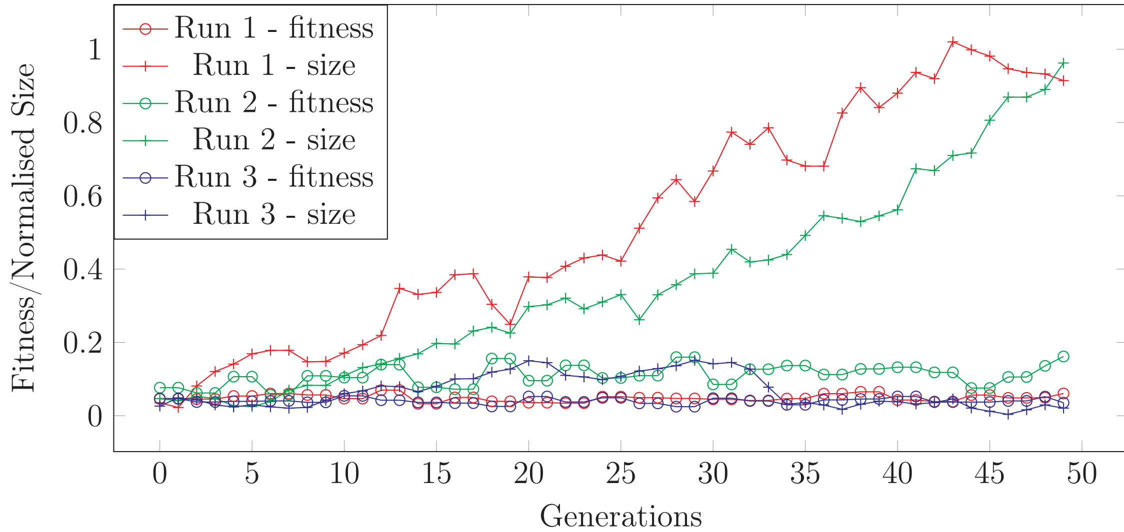


Figure 5.7: Best run from E_5(Competitive Coevolution). Size for all runs has been normalised using 8 as the lowest value and 248 as the highest.

5.1.8 Hand-written BT

With the two-fold goal of evaluating the experiments and testing the project setup, several hand-written behaviour trees were developed. These were evaluated against each other and *CAI* in order to determine which one to use. The best one is used to evaluate the evolved bots and will henceforth be referred to as H_1. See figure 5.8.

H_1 is split into three main branches; economic construction, other construction and movement. When it builds metal extractors it will attempt to fortify the area by building a turret. There is also a chance of the bot building a turret in the next branch which means that the bot might end up building two turrets in a row making it a very defencive bot. The central branch contains all construction not related to economy. Inside it, there is functionality for building turrets, radars, caretakers, factories and storage. The last main branch contains movement related behaviours. If the worker has low health, is close to enemy buildings or in a high tension area it will move to safety. If that is not the case and the area it is in is safe it will instead move to a random new area. All in all, it is a very defencive bot which takes and consolidates metal spots.

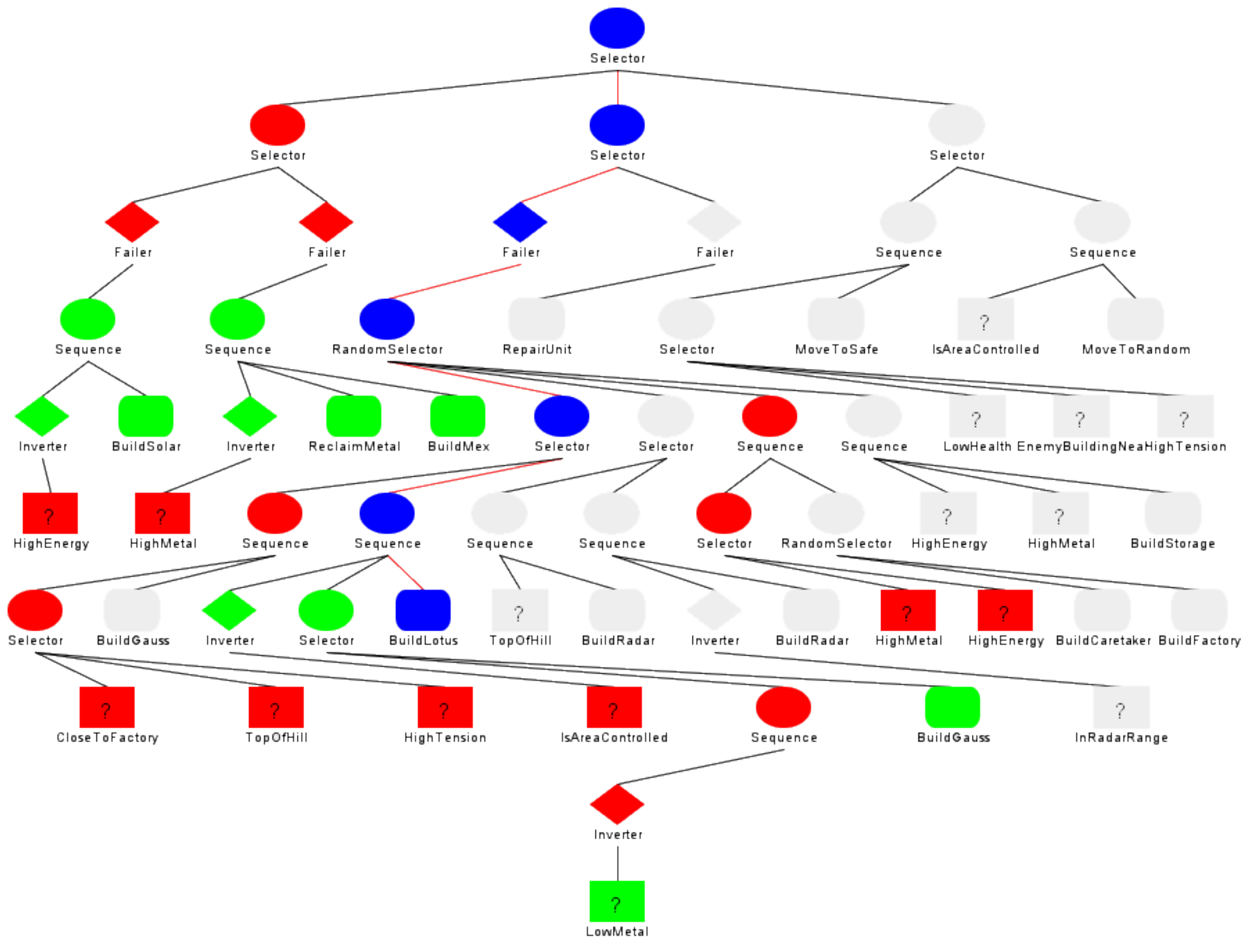


Figure 5.8: The best of the hand-written behaviour trees which is used to evaluate evolved bots. Larger version can be found in appendix A.2.

5.2 Comparative Evaluation

This section describes the comparative evaluations performed in order to determine the quality of the results obtained from the experiments. To analyse the performance of the highlighted individual from each experiment, we conducted three different evaluations. The first and most extensive, described in subsection 5.2.1, involve running the individuals against each other, as well as CAI, the traditional AI and the hand-written BT bot, 100 times for each combination. The second evaluation consists of comparing the first ten buildings constructed by each bot, while the third briefly investigates how the top two evolved bots fair against human players.

5.2.1 Competitive Results

In order to compare the behaviour trees found in the experiments, the BTs were run against each other as well as against H_1, B_1(the bot developed as part of ZKGPBTAI) and CAI. The results are detailed in table 5.8. Each combination in the table represents 100 evaluations between two entities. The table should be read left to right, where the number describes how many battles was won by the entity on the left. To determine which individual from the three runs in each experiment was to be used for this evaluation, the best scoring one from each run was evaluated against CAI ten times and the one with the highest fitness is the ones used in table 5.8. The entities below the double line are the non-evolved bots.

Experiment	E_1	E_2	E_3	E_4	E_5	H_1	B_1	CAI	**Win %**
E_1	N/A	0	0	0	0	0	0	0	**0**
E_2	100	N/A	86	0	57	0	0	8	**36**
E_3	100	14	N/A	0	0	0	0	1	**16**
E_4	100	100	100	N/A	71	100	0	9	**69**
E_5	100	43	100	29	N/A	85	0	1	**51**
H_1	100	100	100	0	15	N/A	0	2	**45**
B_1	100	100	100	100	100	100	N/A	11	**87**
CAI	100	92	99	91	99	98	89	N/A	**95**

Table 5.8: Match statistics from running bots against each other. The bots are Experiments 1-5, H_1 (the hand-written BT bot), B_1(the traditional AI bot described in section 4.3) and CAI
.

Looking at the table 5.8, it is quite clear that the non-evolved bots(H_1, B_1 and CAI) performed better overall than the evolved ones, there are however a few interesting observations to be made. Firstly, B_1 seems to have performed better than CAI on average but lost heavily when played against CAI. The matches were even, but CAI had a good endgame counter against B_1. Secondly, the experiments with larger populations, E_2 and E_4, performed better than E_1 and E_3, which had small populations. Thirdly, E_4 won against H_1 in all of the conducted matches even though H_1 won against all other evolved bots, except E_5, with the same margins. This means that E_4 has a strategy which proved especially useful against H_1.

An interesting point is that E_5 is the second best evolved bot and yet it does badly against CAI, only winning one battle out of 100. The other two good evolved bots, E_2 and E_4, both did eight and nine times

better respectively. This could indicate that E_5 has overfitted or at least adapted to playing against BT bots. Additionally, B_1 beats all the evolved bots but does quite poorly against CAI whereas CAI has lost some of the battles against the evolved bots. This could indicate that the evolved bots have adapted to CAI.

The results in table 5.8 point toward the results not having a transitive property. This is well illustrated by E_2, E_5 and H_1. E_2 beats E_5 57/100 and H_1 0/100. E_5 however, beats H_1 85/100. This could indicate that E_5 has evolved a strategy which is more effective against H_1 than E_2. This intransitivity is common in complex environments like games or sports and makes determining the winner a nontrivial decision. Based on the win percentage E_4 is the most versatile of the evolved bots racking up the largest amount of wins against all opponents.

5.2.2 Build Order

All five experiments focused on evolving a behaviour tree that would be able to handle the economic aspect of Zero-K proficiently. In order to compare how evolution solved this in the various cases, each candidate, including the non-evolved, were run three times and the order in which the first ten buildings were constructed was recorded in table 5.9. The first thing one notices when looking at 5.9 is that there

Bot	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
E_1	mex	mex	solar	mex	solar	solar	solar	solar	mex	mex
E_2	mex	solar	mex	mex	solar	solar	mex	solar	mex	mex
E_3	solar	mex	solar	mex	solar	mex	solar	mex	solar	mex
E_4	mex	solar	mex	mex	solar	solar	mex	mex	solar	mex
E_5	solar	solar	mex	mex	solar	solar	mex	solar	mex	mex
H_1	solar	mex	gauss	solar	mex	solar	radar	gauss	solar	solar
B_1	mex	solar	mex	solar	mex	mex	solar	solar	radar	radar
CAI	solar	mex	radar	mex	wind	lotus	solar	wind	lotus	mex

Table 5.9: This table contains the first ten buildings that each bot constructs. Mex - Metal Extractor, Wind - Wind Generator(similar to solar), solar - Solar Panel.

are a lot of solar and mex entries. In fact, the evolved bots have not been observed constructing anything other than solar panels and metal extractors for their first ten buildings whereas the non-evolved bots use a larger variety of buildings. H_1 and CAI both build two defencive turrets as well as radars in additional to resource generating buildings

like metal extractors and solar panels. All the evolved bots build about the same amount of each building, with a one point difference being the maximum observed.

An important difference between B_1 and the evolved bots is that B_1 builds a lot of radars, two of the first ten buildings in fact. This may contribute to explaining the results in table 5.8 which show that B_1 wins all games against the evolved bots. Building radars is especially important for bots using ZKGPBTAI as military units determine where to attack using information about the location of enemies. Whereas a human can look at a map and determine the likely starting points of the opponent the bot needs to rely on accumulated information of enemy locations. Though the use of radars may have played an important part, it is by itself not enough to explain why B_1 did so well against the evolved bots.

The prevalence of mex and solar in table 5.9 is no coincidence. The most effective strategy for the early game that we have observed for Zero-K revolves around building a strong economy while harassing the opponent using your soldiers as well as protecting against similar attacks. The evolved bots have clearly employed the first point of that strategy as they build a solid economy. Some of them have evolved strategies that perform well in the mid-game, for example E_4 which builds radars and caretakers. A clear difference between the evolved and the non-evolved is the use of turrets, which are important for protecting against enemy advances and early game raids. CAI clearly uses this as evidenced by table 5.9. Unfortunately, implementing action nodes that contain behaviour for harassing the opponent fell outside the project scope. All in all, the evolved bots have learnt how to build a strong economy.

5.2.3 Human Trials

In order to further investigate the two best individuals identified, E_4 and E_5, both authors played a game against each of them. E_4 won one of the games while the rest was won by the human players.

We discovered that E_4 is quite a good bot player but it can be easily countered if harassed constantly from the beginning of the game. If left alone it will become very strong towards the middle of the game when it starts building caretakers and radars. These two buildings have the coupled effect of building a large army and directing said army towards its opponent using the information provided by the radars. Additionally,

it was revealed that E_5 will rebuild a factory if the first one is destroyed which we had not been able to deduce from studying the behaviour tree.

5.3 Discussion

In this section we discuss and explain the findings presented in the previous two sections. The topics are fitness function, stagnation, mutation compared with reproduction, population size, bloat, strong typing and evolutionary scope.

5.3.1 Fitness Function

The fitness function of a GP system has a substantial impact on the results. In this project economical fitness accounts for 40% of the total fitness, 50% is given for victory, which leaves only 10% for military. This has had a clear effect on the results, as the best individuals from all five experiments mostly build *metal extractors* and *solar panels*. Additionally, as illustrated by the build order table 5.9, the evolved bots all construct economic buildings for the first ten buildings whereas the non-evolved use a more varied selection. Because economic buildings increase the total fitness by such a large amount, the effect that building radars and defencive turrets has on the success of the military is overlooked. Some of the evolved bots construct factories and caretakers in addition to resource buildings. An explanation for this is that both of these buildings contribute to producing more soldiers, resulting in a greater chance for victory. Factories and caretakers have a much more explicit effect on the victory chances of a bot than radars or defencive turrets. Another underused building is the storage. Constructing more storage has a very small effect on the economy fitness but may have a significant effect on the behaviour of the *lowMetal*, *highMetal*, *lowEnergy*, *highEnergy* condition nodes. These nodes are implemented in such a way that *highMetal* requires the bot to have 90% of the storage full as well as a positive inflow of resources in order to return success. Because of this, building a new storage leads to *highMetal* no longer be true, which can have unpredictable effects on the traversal of the tree.

Two potential ways of solving these problems are a better fitness function and more strong typing. For the former, an improved fitness function that awards more fitness based on military and especially

defencive turrets, which the evolved AI has greatly underused. For example, a score could be given based on how many enemy units friendly turrets have killed. To increase the use of radar towers, a score could take into account what average percentage of the map is in radar range. All in all, a more complex fitness function with more points being given to aspects relating to specific buildings. For the latter, strong typing could be used to force each BT to include a variety of buildings. For example by requiring every tree to contain both the resource buildings, factory, radar and one of the defencive turrets. This would have the effect of narrowing the search space which can be both positive and negative. It would make the bot always have all the building blocks of success close at hand.

Awarding 50% of the fitness points for winning a game has had a detrimental effect on the runs. In these experiments each individual is evaluated up to three times in a generation to determine fitness. If the evolving bot wins one of the evaluations, its fitness is boosted by a large amount. In order to evaluate a bot, it is run against CAI, which is far from a stable benchmark. CAI chooses between several strategies and will occasionally pick a sub-optimal one, thereby reducing the difficulty of winning against it. This can have the effect of a mediocre individual being awarded a large fitness and because of that having its characteristics propagated into new generations.

5.3.2 Stagnation

Progress in evolution depends on variation in a population [McPhee and Hopper, 1999]. A big problem in EC is loss of diversity which can lead to stagnation. This usually takes the form of the system getting trapped in a local optima which it lacks the genetic diversity to escape. There are many proposed diversity measures, two popular measures are described in [Koza, 1992; Keijzer, 1996]. In the former, Koza uses the term *variety* to indicate the number of different genotypes a population contains. In the latter Keijzer measures program variety as the ratio of the number of unique individuals over population size and subtree variety as the ratio of unique subtrees over total subtrees.

We decided to analyse the experiments using Keijzer's measure for variety. Calculating program variety revealed a big difference in values across the runs, even within the same experiments. In figure 5.10 the mean program variety across all runs for experiments E_1 to E_4 have

been placed on a graph based on the program variety. While program variety will reveal duplicates, the individuals might rely on a very small subset of trees giving the metric little meaning. Subtree variety however takes this into account.

In figure 5.9, the subtree variety of all five experiments has been calculated. Extracting all possible DAGs from the individuals as their subtrees is usually the better base for the measure. The layout and structure of the output files however, made a recursive division by the composite nodes the more practical approach, and was chosen for this reason.

The subtree variety measures the amount of duplication in an evolving population and tracing it is useful as it may reveal a loss of diversity. This metric usually gives very low values, but it is important to watch out that the variety of a generation does not drop to or below $\frac{1}{numberOfPrograms}$, which would indicate that the population is dominated by a small set of subtrees. This value is important since it is the value one would get for a population filled with identical individuals. Individual programs might contain duplicate subtrees allowing the values to drop even further. When dropping below this line it might be difficult for the evolution to explore new regions of the search space possibly causing stagnation. Experiment E_1 and E_3 will have this limit at 0.05 while E_2 and E_4 at 0.02 far below. As the graph in figure 5.9 illustrates the average values were far above this. Only one of the runs came close, the best individual from run 1802 of E_3 with a few generations nearing 0.07 in subtree variety containing only 2 unique programs.

According to Koza [Koza, 1992] the first step in studying the evolution of complex structures is studying the gross size of the individual programs. Even though the subtree variety measurements shows a big and continuing drop throughout the runs for all experiments, this does not necessarily indicate a reduced amount of unique subtrees. As illustrated by figure 5.13 the average size usually increases throughout the run. Figure 5.11 displays the average amount of unique subtrees and the total amount of identified subtrees across all three runs in experiment E_4. It illustrates the correlation between the values making up the subtree variety measure and shows an increase, rather than a decrease, in unique subtrees.

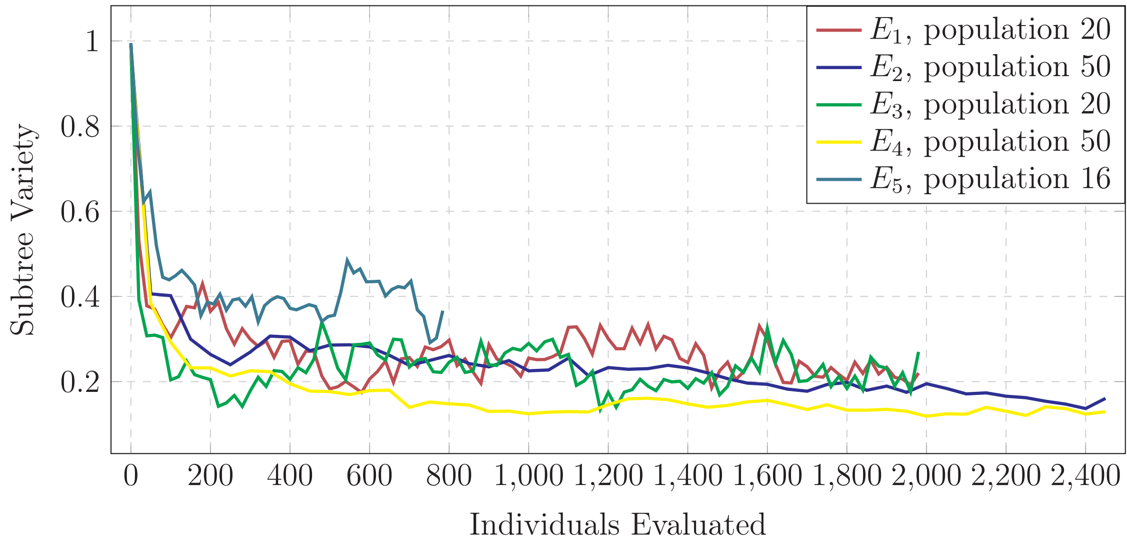


Figure 5.9: Plot of the subtree variety in all five experiments. It is important to take the population size into account when comparing the experiments as the tolerances will be slightly different.

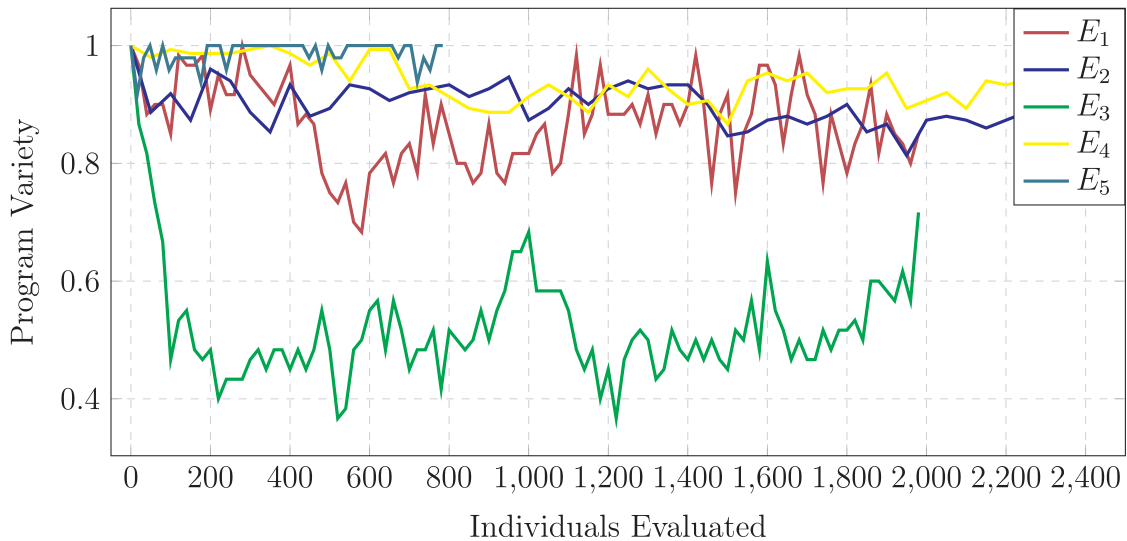


Figure 5.10: Plot of the program variety in all five experiments, where the value one indicates there are no duplicate individuals present after a given number of evaluations.

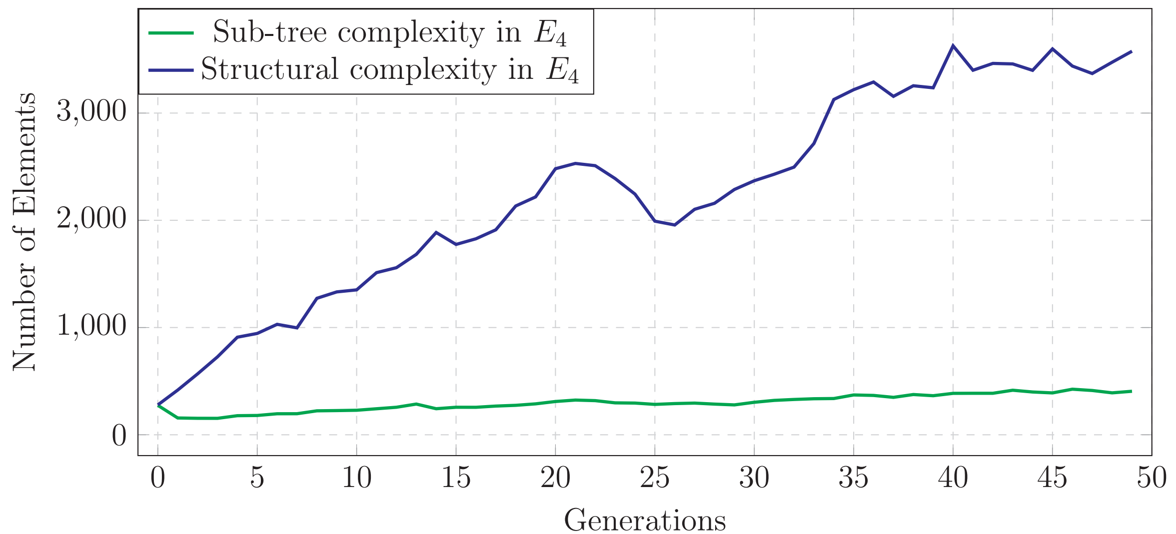


Figure 5.11: Plot of the average sub-tree and structural complexity for all runs in E_4(Large population, mutation).

A problem we discovered when interpreting the results was that in some cases, at the end of a run, many individuals would be almost identical with only minor differences, and then often in leaf nodes that never get activated. In the worst case there were several identical individuals in the population, and in all cases there were several similar individuals. The latter is to be expected as the crossover between trees can, depending on where the crossover is performed, lead to increasing similarity over time. The former however, is a problem as it can lead to stagnation as well as a lack of diversity in the population. We have a threefold explanation for this.

The first explanation is the concept of *hitchhiking*. The main idea is that not all placements of nodes have the same significance. This means that individuals which have a few well placed nodes will do well, this does not however mean that all nodes in that individual are placed well nor beneficial but they will be propagated into the new population nonetheless.

The second explanation is *Genetic drift*, which is a term describing the gradual loss of a node type from the population through crossover. Some nodes and subtrees will do better than others and as an effect of this some nodes might be completely removed from the population. However, if a mutation operator is used it is possible that an excised node might reappear through mutation.

Lastly, a possible explanation is the tries-parameter used in the

ECJ parameter files. *Tries* refers to how many attempts are allowed in case crossover results in an illegal tree. If no legal tree is found when the limit of allowed attempts is reached, an unmodified copy of one of the parents is returned instead, arguably having the same effect as reproduction. In this project we chose to use ECJ's default value for this parameter, which is one. Additionally we employ strong typing which makes it even harder for crossover to generate a 'legal' tree. This in combination with the chance of a mediocre individual propagating its characteristics as described in section 5.3.1 can cause large distortions in the population that can lead to stagnation. In experiment E_5 the tries-parameter was increased to avoid this possible issue. Across all three runs there were fewer cases of duplicate program, see figure 5.10, strengthening our suspicion that the parameter affected experiments E_1 through E_4.

Of the three possible explanations of the observed stagnation, hitchhiking and the tries problem are the likely culprits. Genetic drift can likely be ruled out as it should be negated by using mutation. This is exemplified by there being no major observed differences between E_1 and E_3 nor E_2 and E_4, where E_3 and E_4 used mutation.

Hitchhiking is a plausible explanation for the low subtree variety in nodes often observed at the end of a run. Whereas it does explain loss of diversity, it cannot explain the nearly identical trees observed in some of the runs which is better explained by the tries problem. The prime example of this is E_3 which had just two unique individuals present in some generations, drastically reducing the likelihood of the run yielding a good solution.

5.3.3 Mutation vs. Reproduction

The two most common genetic operators in GP and EC in general are crossover and mutation. Crossover is a convergence operation which pulls the population towards a local minimum/maximum. The role of mutation is to break individuals out of local minimum/maximums in order to discover possible better solutions. In [Koza, 1992] and [Koza, 1994] John Koza argued and demonstrated that mutation was not necessary in order to get good results with GP. While Koza does not view mutation as necessary, he did advice using a low level of mutation in [O'Reilly and Oppacher, 1996]. There have also been papers where pure mutation approaches have been demonstrated to outperform

crossover ones [Harries and Smith, 1997]. In [Luke and Spector, 1997] Sean Luke, the main contributor to ECJ, performed a comparison study between crossover and mutation approaches. He found that crossover was slightly more successful than mutating overall but with a small margin. Additionally, mutation was found to be more effective for smaller populations depending on the problem domain. All in all, it would seem that good solutions can be found with both approaches, with the results largely dependent on the genetic operators suitability to the problem domain.

Figure 5.12 shows the average fitness achieved in each of the experiments, $E1$ and $E3$ clearly follow a very similar trajectory though the former uses no mutation and the latter does. The same can be argued for the best fitness, see figure 5.14. There is a clear disparity between the experiments with different population parameters but mutation would seem to not have had any significant effect neither on average nor best fitness. A possible explanation for this is that mutation is not very suitable to this problem domain. Another explanation is that the mutation rate employed was quite low at 2% and its effect might have been negligible in the small population sizes used. A third explanation is that the tries problem described in subsection 5.3.2 might have interfered with the proper working of the mutation operator. All in all, no single interpretation can be said with certainty to be the right one as the reality of it may well be a combination of several explanations.

5.3.4 Population Size

In this project it was decided to perform two experiments with small populations, many generations and two with large populations, fewer generations. The reasoning behind this is that there are proponents for both approaches. On the one hand, it has been suggested to use as large a population as computationally feasible [Koza, 1992], on the other, it has been argued that for some problems, a smaller population evolving over many generations will produce better results [Gathercole et al., 1997; Fuchs, 1999]. As a compromise we tested both approaches.

It was found that the two experiments with larger populations performed notably better than their smaller counterparts when it comes to both average fitness and best fitness. In figure 5.12 and 5.14, one can clearly see that experiments E_2 and E_4 performed better than

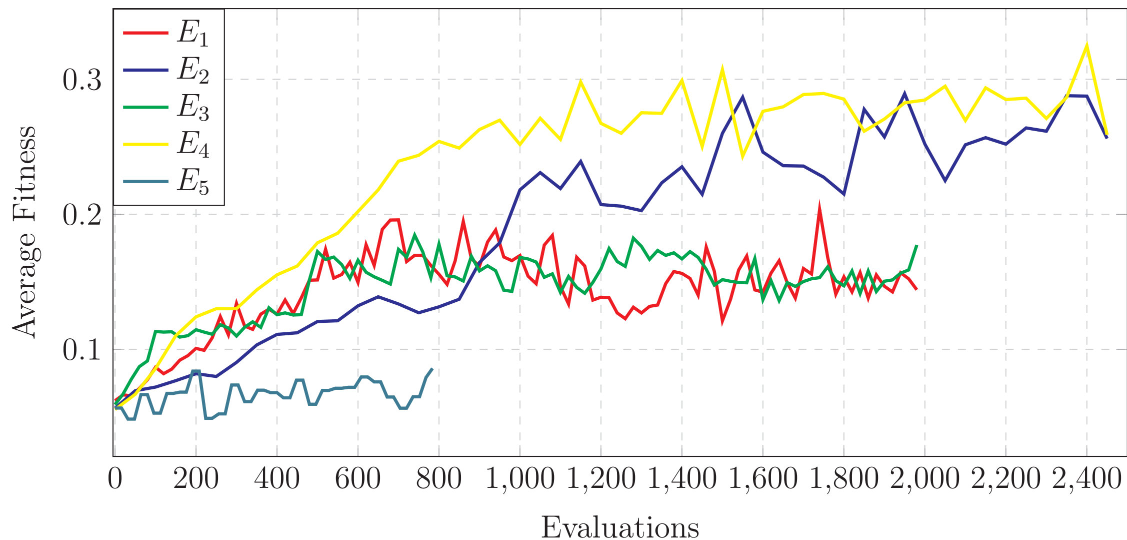


Figure 5.12: Plot of the average fitness across all experiments where each line represents the average of all three runs in a single experiment.

E_1 and E_3. At about 800 evaluations the latter stagnates whereas the former keeps improving. The explanation for this might be rooted in the loss of diversity described in section 5.3.2. Given that hitchhiking and the tries problem may lead to stagnation over time, the experiments with larger populations should take longer to stagnate as they contain a greater variation of BTs to begin with.

Though no statistically valid conclusions can be drawn due to the low amount of runs performed, the larger populations have performed better than smaller ones in this project, both in terms of average fitness and in terms of finding the best possible solution as evidenced when evaluating the individuals against each other. While using a small population, E_5 is not comparable to the first four experiments as it is based on competitive coevolution and was run for a lot fewer evaluations.

5.3.5 Bloat

As described in section 2.4, the most common definition for bloat is "program growth without (significant) return in terms of fitness" [Poli et al., 2008]. This definition is quite vague as "significant return" can be interpreted differently and as a consequence there have been several attempts at defining a metric for bloat. In [Vanneschi et al., 2010], the authors define bloat as an expression between the average growth and the average fitness improvement in generation g compared with generation zero, see equation 5.1. The term $\overline{\delta}(g)$ refers to the average program size,

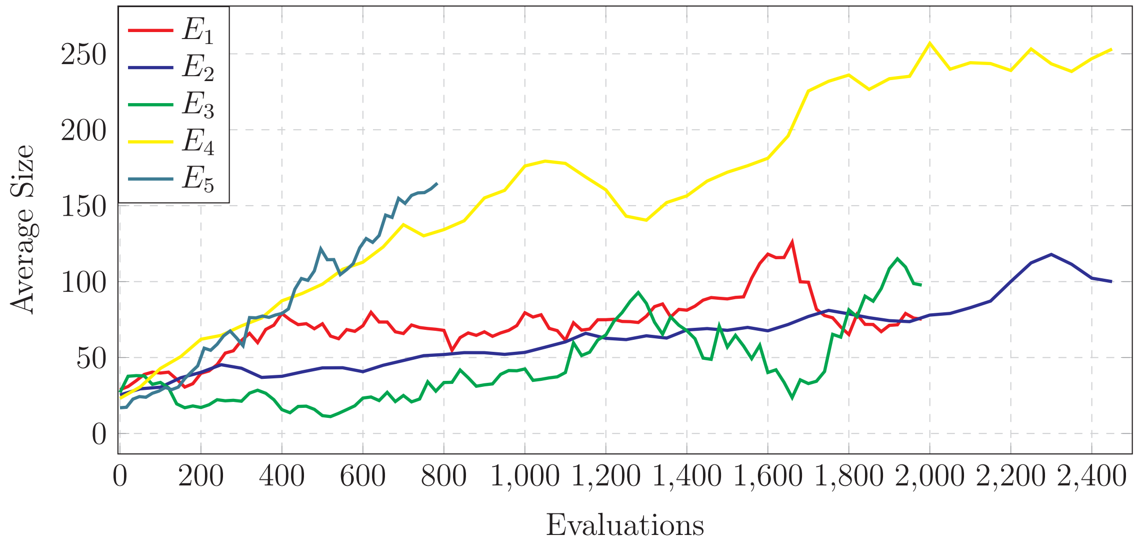


Figure 5.13: Plot of the average size across all experiments where each line represents the average of all three runs in an experiment.

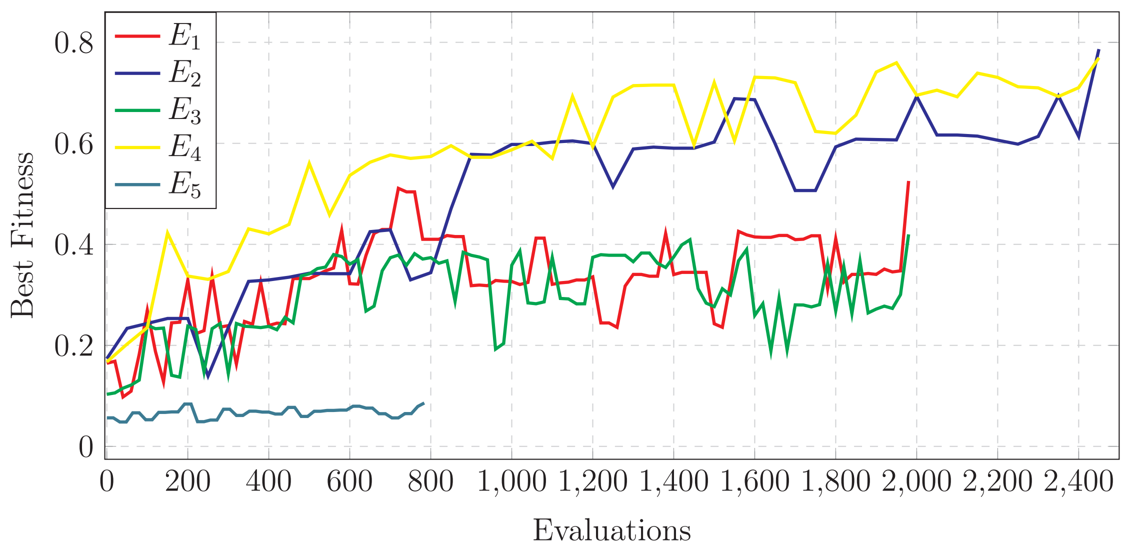


Figure 5.14: Plot of the best fitness across all experiments where each line represents the average of all three runs in an experiment.

while $\overline{f}(g)$ refers to the average fitness at generation g.

$$bloat(g) = \frac{(\overline{\delta}(g) - \overline{\delta}(0))/(\overline{\delta}(0))}{(\overline{f}(0) - \overline{f}(g))/(\overline{f}(0))} \tag{5.1}$$

We decided to measure bloat across all experiments using this technique, this resulted in figure 5.15. There are a few observations to be made here; firstly, there is a large spike in the very beginning of two of the experiments, which is most likely due to selection picking what seems like a good individual in the first generation but after crossover turns out to be undesirable. Secondly, E_3 goes quite far into the negative values in the first 600 evaluations. This means that the average size has decreased while the average fitness has increased which is very good. Thirdly, E_4s bloat seems to grow towards the end of the run which matches the plateauing curve in figure 5.12 and the increase in size seen in figure 5.13.

Avoiding bloat is important when evolving behaviour trees as large BTs will take up more memory. Traversal through behaviour trees is relatively efficient, compared to alternatives like FSMs, though when many large trees are introduced it may cause performance issues. When performing the experiments, those runs with a large average size took noticeably longer time to run than smaller ones. Consequently avoiding bloat is highly preferable and important for the results to be considered successful. Though we have found instances of bloat in this project, it is not prevalent enough to be considered an issue. If more generations were introduced to the experiments it is likely that bloat would become a bigger problem as indicated by the upwards curve of E_4.

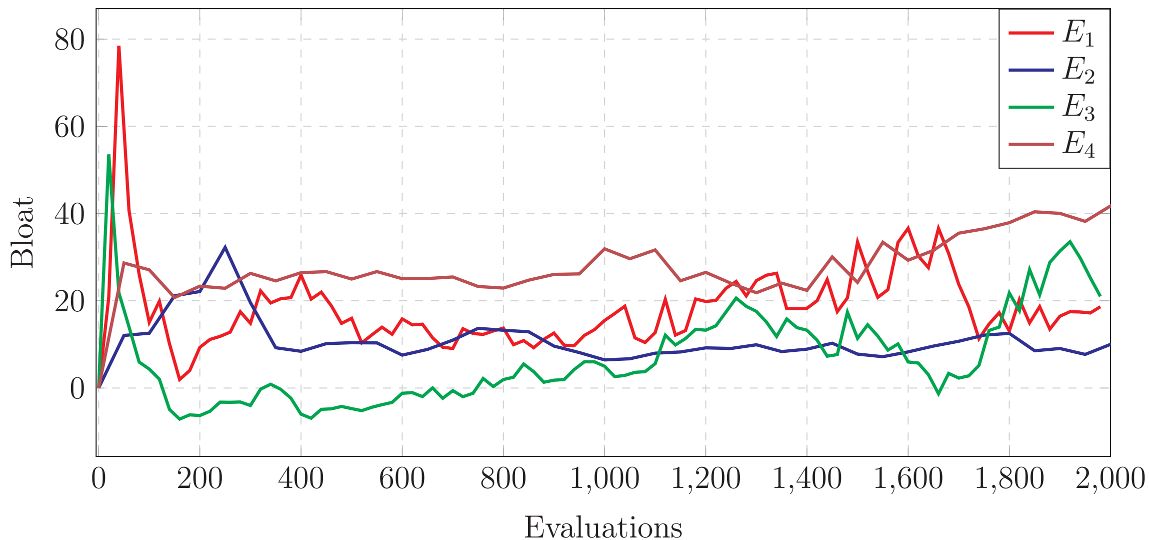


Figure 5.15: A plot of the bloat of the experiments calculated using equation 5.1.

5.3.6 Strong Typing

Strong typing was mainly used to remove situations that caused crashes and to limit redundancy. It proved successful in reducing redundancy by removing situations where a decorator node would have another decorator node as a child, for example the pair inverter→inverter. It was also beneficial for removing situations that could lead to crashes like infinite loops, e.g. the connected pair, untilSucceed → failer. Strong typing could probably have been used more in order to restrict the search space and accelerate evolution, for example by requiring that each individual contains all or a subset of the construction nodes. However, we wanted to keep evolution as general and unrestrained as possible in order to make it transferable to another problem domain. Additionally, when restricting the search space using strong typing, one might inadvertently remove too much and near-optimal solutions might end up outside the scope of the algorithm. In this case we have acquired a significant amount of domain knowledge and can therefore say with certainty that the optimal solution to this problem requires the use of a wider selection of buildings than what the best solutions found use.

5.3.7 Competitive Coevolution

Competitive coevolution resulted in the second best individual found in all the experiments. Experiment five was the last of the experiments

to be run and because of several factors we had to shorten its run time by reducing the population size to 16. Despite this the experiment performed well and was the only one to evolve behaviour that used the *reclaim* node correctly.

As discussed in 5.3.4, larger populations have achieved better results and that may be the case for competitive coevolution as well. With only 16 individuals evolving over 50 generations, the experiment only consisted of about 800 evaluations compared with the 2000 to 2500 in the other experiments. Having achieved second place in so few evaluations could indicate that competitive coevolution converges on good solutions faster than the setup used with the other experiments, though further experimentation is needed to determine if this is the case.

5.3.8 Evolutionary Scope

It was originally our intention to evolve several aspects of the game(economy, military, recruitment) separately, then integrate them gradually into a complete bot where BT would be responsible for all non-trivial decisions. However, due to time constraints only evolution of the first aspect was completed. It is likely that if all aspects were evolved the bot would perform notably better as it would have a more complete perception of the problem domain.

The evolutionary scope has had a large impact on several aspects of this project, most evidently in the fitness function. The fitness function used for all of the experiments was written with the purpose of evolving an AI that would create a good economic base which the military and recruitment behaviour branches could make use of to achieve a military victory. The other major game aspects would have their fitness functions tailored to their respective tasks.

The evolutionary scope had a large effect on which condition and action nodes were included. The condition nodes define what the bot can perceive of the game environment and as such fulfil a very important function. As the the BT controls a unit that moves around in the game world many of the nodes relate to the environment e.g. *topOfHill* and *closeToFactory*. There are a few nodes that relate to enemies in the vicinity of the worker unit, for example *enemyBuildingNear* and *highTension*, but none that relate to the opponents strength in general. This means that the defined military condition nodes only allow for reactive behaviour and not planning

behaviour when it comes to military matters. Additionally, the decision to leave out the parallel node effectively slowed down the reaction as the BT in a worst case scenario would need to complete a whole tree traversal in order to react to the environment.

We have not considered what military actions a worker unit could accomplish and this is reflected in the selection of action nodes. A worker can *moveToSafe*, *repairUnit* or construct a building but there are no nodes that define more explicit military actions such as attacking.

This project uses 26 BT nodes which is far greater than what has been used in similar projects [Perez et al., 2011; Lim et al., 2010]. To include the improvements mentioned above, along with more units, actions and conditions would have exponentially increased the complexity and hence the computation and time requirements. Zero-K has a far more complex environment than what is considered in related projects, requiring great abstractions and reductions of what the bot can perceive of the game world. These modifications have the added effect of putting the evolved bot at a disadvantage compared with traditional AIs and human players.

All in all, this means that evolving bot can only perceive economic aspects of the game and is almost only judged on how it performs in terms of economy. Given these caveats, GP has found a very good solution in E_4 as the best individual both builds a good economy and constructs caretakers and radars in the middle of the game which indirectly has the result of constructing a large army and telling it where to go.

5.4 Summary

This chapter has given an overview of the conducted experiments as well as a detailed discussion of the results and the most important aspects which affected them. Experiment four produced an individual capable of beating our best hand-written BT every time, the best individual does however lose to the two non-BT bots most of the time. The fitness function has had a significant effect on the outcome by outlining what defines a good solution. An unfortunate side effect of this is that it pushed the individuals towards excessive construction of resource buildings effectively restricting the use of other construction nodes. There has been some stagnation of diversity in the experiments but not uncommonly high, except in the case of one run from experiment three where the genetic diversity reached notably low

measures. This was almost certainly due to the the tries problem but hitchhiking likely also played a role. No significant difference in results was found between the experiments using mutation and those that did not. Experiments with large populations was found to have performed better than those with smaller ones, most likely due to having access to more varied subtrees in the form of a larger initial population. Though some instances of bloat were found, it was not prevalent enough to be considered an issue. The large abstractions of the perceived game environment reduced the chances of evolving a bot which could challenge the best traditional AIs and human players. However, despite the circumstances the best two individuals performed quite well by beating the hand-written BT bot in most games.

Chapter 6

Conclusion

This chapter begins with a brief summary of the chapters of this report. Following that, section 6.1 contains a discussion of results of this project in relation to the research questions. In section 6.2 the contributions of this thesis to the fields of evolutionary computation and game development are discussed. In section 6.3, limitations of this project and suggested future work is presented. Lastly, section 6.4 contains a few closing remarks.

In Chapter 2 we gave an overview of the theories and techniques that form the base of this project. Some of the challenges of the game intelligence field were briefly discussed. The behaviour tree technique was outlined in detail. Finally, genetic programming and its roots in biological evolution was explained.

Chapter 3 describes how related work was identified and the most relevant of them were discussed in detail. There were several interesting related systems and projects, some proved more relevant than others. In the case of behaviour trees, there was not any single most relevant project but rather the impression of the state of the art of behaviour trees, which the study of the described papers provided. The opposite is true for genetic programming; we had a lot of knowledge of genetic programming prior to this project but it was the details of how it has been used in conjunction with games that proved the most useful.

Chapter 4 covers how the system on which the experiments were run was developed. Zero-K is built upon the Spring Engine which provides an AI interface for developing bots in different languages. Our implementation is written in Java, section 4.3 describes how it was developed and the techniques that were employed. This game interface functions as the base and connection point for the behaviour trees

outlined in section 4.4. The BTs make decisions and issue commands for the game interface to execute. It is behaviour trees which are the subject of the evolution in this project. The BTs are evolved using genetic programming which is covered in section 4.5. Through strong typing and use of the framework provided by ECJ, GP was set up with a variety of settings and parameters for the different experiments.

In Chapter 5 the conducted experiments were described, compared and discussed. The important factors that affected the results were discussed in detail. The most important factors were the fitness function, stagnation, population size and the evolutionary scope.

6.1 Goal Evaluation

The goal of this project was to explore a method for automatically generating artificial intelligence players for RTS games using genetic programming and behaviour trees. This was accomplished by creating a system for evolving behaviour trees for the RTS game Zero-k, performing a series of experiments while answering the research questions. This section contains a discussion of the answers found to the research questions.

> **Research Question 1:** How well does genetic programming work in concert with behaviour trees?

There are striking similarities in the representation of behaviour trees and GP syntax trees. We encountered few issues when mapping them together by tailoring the GP implementation to represent the programs using the BT syntax.

This technique has been used in similar projects. The studies described in [Baumgarten and Colton, 2007; Lim et al., 2010] also investigated representing GP individuals as behaviour trees and evolving them in game environments. In this project we used a greater amount of nodes to represent a more complex game environment than done previously, in order to further explore the viability of this approach. The individual BTs were tasked with controlling an important aspect of the game effectively determining the strategy as well as making many major decisions. Given a restricted selection of ways of sensing and acting (condition and action nodes) on the the environment, compared with a human or traditional AI player, the

experiments produced behaviour trees with goals clearly matching their relative importance as defined in the fitness function. Some of the examined individuals evolved clever patterns utilising a range of features provided by the behaviour tree framework, as well as developed different behaviours depending on the state of the game. These results indicate that this method holds merit, even when given a task of greater complexity than demonstrated in previous projects.

The coupling of behaviour trees and genetic programming shows promise. We assess the technique to be viable based on the results obtained through this study and our assessment of related research, described in subsection 3.2.3.

> **Research Question 2:** Given the same components, how do behaviour trees evolved using genetic programming compare with ones designed by humans?

The solution generated by the most successful experiment beat the best hand-written tree every time, however, the quality of the results from the five experiments varied greatly. Experiment one produced a solution that was not able to beat any of the best solutions from the other experiments nor the hand-written AI. Due to the exploratory nature of this study and the stochastic environment we cannot statistically conclude what experiment will provide the best results. It does however, provide a useful indicator for future research.

The most successful experiment had a large population and used the genetic operators crossover and mutation. The selected individual from this experiment won against the best hand-written BT every time. It evolved behaviour that creates a strong economy in the beginning of a game, then constructs buildings that have the effect of recruiting and directing a large army that can overwhelm the opponent. This strategy proved quite good, consistently beating the results of the other experiments and even occasionally defeating the AI that came with the Zero-K.

The best individual from experiment two contains an especially interesting behaviour tree structure. It evolved two clearly separated branches with fairly different goal-oriented behaviours. The structure provided the individual with reactive behaviour through clever use of BT nodes. More impressively, it is a composition which we had not conceived of when designing the hand-written BT. This means that this technique is not only applicable as a way of generating an AI

player but also as a tool for finding useful patterns and strategies in the game environment.

All in all, evolving behaviour trees using genetic programming resulted in a much better BT solution than the authors were able to design manually.

> **Research Question 3:** Can using genetic programming to evolve behaviour trees simplify or improve the development of AI players for games?

Whereas the evolved BTs did very well against the hand-written BTs they did not do very well against the traditional AI bot developed in this project (B_1) nor the AI shipped with Zero-k (CAI). All evolved BTs lost every game against B_1 and the best evolved BT managed to win only 11% of the matches against CAI.

Traditionally, creating AI for games has revolved around giving the appearance of intelligence using techniques that are as simple to implement as possible, and that uses the least amount of computational power. This is because developing better AI takes time and does not always yield noticeably better results. Due to the computational power available, it is only in recent years that using techniques like machine learning and evolutionary computation for modern games has become feasible. In this study the development of the traditional AI required drastically less time compared with the implementation required to evolve BTs. Consequently, we cannot argue that replacing traditional AI with the proposed technique would significantly simplify AI development for games.

One of the advantages that this technique could provide is the automatic generation of a larger variety of AI opponents in games. Typically, games are shipped with one AI player which can be made more difficult or easier by giving it access to more computational power or in game resources. Some games contain AI that exhibit different personalities, for example Civilization 5 [2kGames, 2010], where the player can choose to play against AIs that exhibit varying levels of aggressiveness. Evolving BTs could provide similar functionality by tailoring the evaluation of the individuals based on the presence of certain desirable traits.

Based on the results obtained in this study we cannot conclude that evolving behaviour trees is a serious competitor to traditional AI methods. However, we believe that when techniques like evolving

behaviour trees have had time to mature and if commercial tools to simplify their use become available, they might be useful tools in the process of developing AI for games.

6.2 Contributions

The current research on evolving behaviour trees using evolutionary algorithms, as described in section 3.2.3, has shown this method to be applicable to a range of tasks, and has had a large impact on our research. This study builds upon these results and extend its use to a more complex environment in the form of the RTS game Zero-K. Our research has yielded some positive results, though not of statistical significance, indicating that the method is applicable to such a problem domain.

Few publicised studies are researching the possibility of evolving behaviour trees through genetic programming. This thesis will provide additional background in the field and contribute by exploring the possible applications of these methods. By providing a behaviour tree library tailored to ECJ our contributions also include lowering the implementation overhead for similar projects in the future.

6.3 Limitations & Future Work

In this section we outline further lines of enquiry identified in this project. There are three main points we consider as future work; extending the evolutionary scope, technical improvements and the development of a tool for simplifying the application of evolutionary methods to behaviour trees.

6.3.1 Extend Evolutionary Scope

We believe research question one should be investigated further by extending the evolutionary scope to cover more aspects of the gameplay. In most of the identified related work, the authors have had to reduce the evolutionary scope in order for evolving behaviour trees to be feasible with the resources at hand [Oakes, 2013; Lim et al., 2010]. Similarly as stated in subsection 5.3.8, this project only covers one aspect of playing a real-time strategy game. It is our belief that if all aspects were to be evolved, the result would be greatly improved.

Another important aspect to consider in future work is the avoidance of overfitting by evolving individuals in a variety of environments (game maps) and against a more varied selection of opponents, alternatively more competitive coevolution. This would make it possible for the individuals to discover strategies and tendencies that transcend a single environment and therefore become more adaptable to novel challenges. Other researches have had the same experience; Lim et al. "considering other starting positions" as future work and also discusses evolving against a variation of opponents.

6.3.2 Technical Improvements

The biggest way to improve the results of this project and similar projects would be to get access to more computational power so that one could both perform more experimentation and get results with statistical significance.

More computational power would make it possible to run competitive coevolution properly with only wins and losses counting towards an individuals fitness. Running genetic programming like that would mean that the search space would be larger and consequently the algorithm would explore more potential strategies than it currently does. Additionally, the developer would need far less domain knowledge of the game and its strategies, thus simplifying the application of the technique to new games.

6.3.3 Behaviour Tree Library

As mentioned in the goal evaluation, the introduction of a tool for simplifying the application of evolutionary approaches to game AI and behaviour trees might greatly advance the use of techniques like the one explored in this project. As part of this research we created a general behaviour tree framework for evolving BTs with ECJ, however, it does at this time only include the most basic features required for this project, but can provide a solid base for further implementation. Extension of this library to support more advanced features, or the development of new similar tools, would simplify the application to new problems and lower the implementation overhead.

6.4 Closing Remarks

Overall, we feel that this project has been successful when considered as an exploratory research project. We developed a system which automatically generated an AI player that consistently beat the best hand-written AI we were able to produce using the same markup. We were able to achieve these results only using the computational power of two regular desktop computers, far less than what a commercial developer might have access to. However, the results fell short of our expectations in terms of being competitive with traditional AI methods. All in all, we hope this project provides valuable insight for future research into this area and for pioneering game developers attempting to bring more advanced AI into the game industry.

In closing;

> *Evolving behaviour trees is an interesting technique for automatically generating AI players that has shown that it can produce solutions that consistently beat ones produced by humans using the same components. Based on the findings in this study it is at present not a solid competitor of traditional AI methods. However, given well-defined components and sufficient computational power this technique has the potential of being a very useful tool for game AI developers.*

Appendix A

Additional Resources

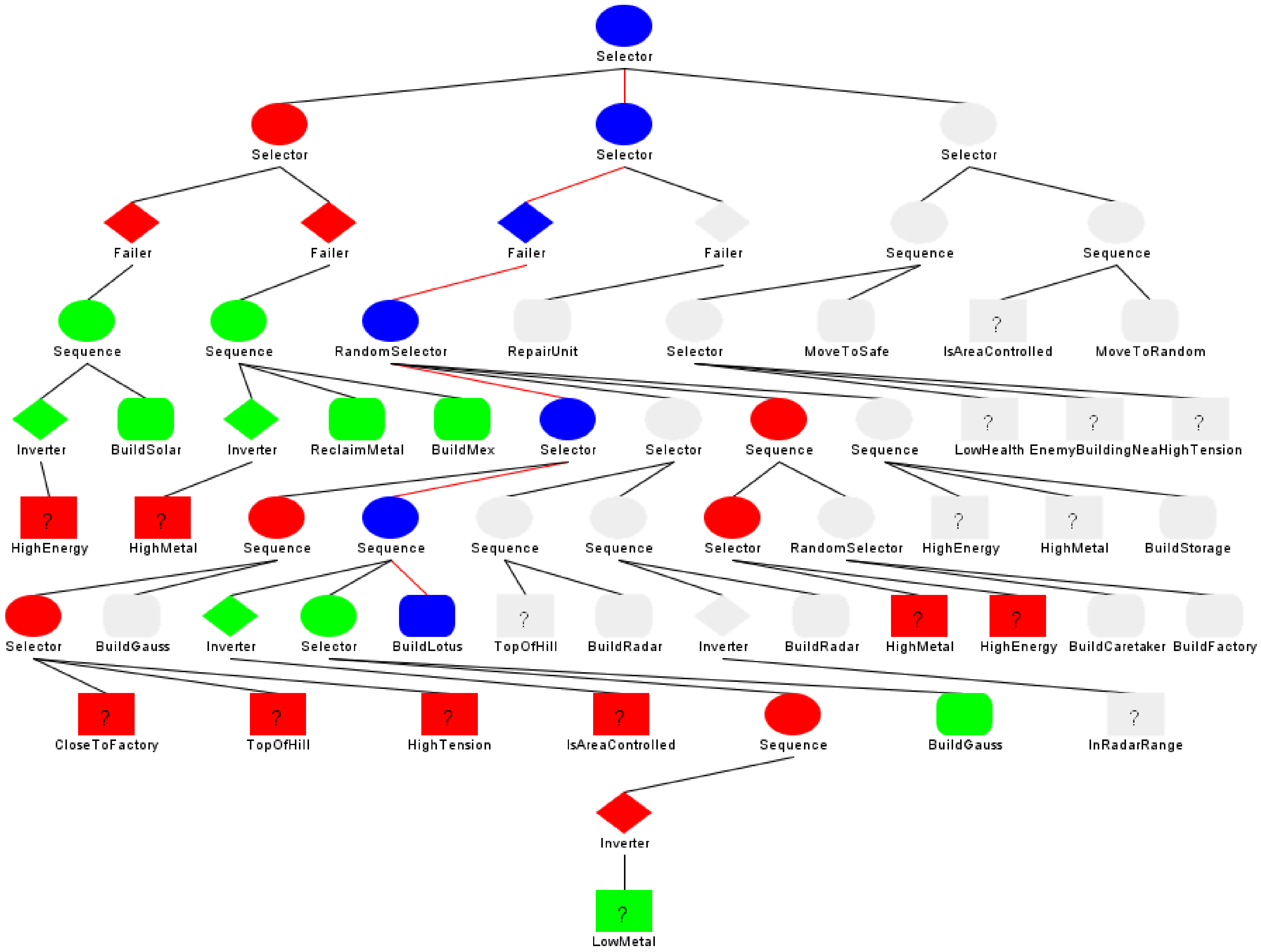
A.1 Repositories

ZKGPBTAI:
https://github.com/elusivedelusive/ZKGPBTAI

EvolutionRunner:
https://github.com/hajoch/EvolutionRunner

BehaviourTreeECJ:
https://github.com/hajoch/BehaviourTreeECJ

A.2 Hand-written BT



Selector
Selector
Selector
Selector
Failer
Failer
Failer
Failer
Sequence
Sequence
Sequence
Sequence
RandomSelector
RepairUnit
Selector
MoveToSafe
IsAreaControlled
MoveToRandom
Inverter
BuildSolar
Inverter
ReclaimMetal
BuildMex
Selector
Selector
Sequence
Sequence
LowHealth
EnemyBuildingNea
HighTension
HighEnergy
HighMetal
Sequence
Sequence
Sequence
Sequence
Selector
RandomSelector
HighEnergy
HighMetal
BuildStorage
Selector
BuildGauss
Inverter
Selector
BuildLotus
TopOfHill
BuildRadar
Inverter
BuildRadar
HighMetal
HighEnergy
BuildCaretaker
BuildFactory
CloseToFactory
TopOfHill
HighTension
IsAreaControlled
Sequence
BuildGauss
InRadarRange
Inverter
LowMetal

Bibliography

P. Lu J. Schae er, R. Lake and M. Bryant. Chinook: The world man-machine checkers champion., 1996.

A Champandard, M Dawe, and DH Cerpa. Behavior trees: Three ways of cultivating strong ai. In *Game Developers Conference, Audio Lecture*, 2010.

Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2009.

Ludeon Studios. Rimworld, 2016. URL `http://rimworldgame.com/`.

R Geoff Dromey. From requirements to design: Formalizing the key steps. In *Software Engineering and Formal Methods, 2003. Proceedings. First International Conference on*. IEEE, 2003.

Damian Isla. Managing complexity in the halo 2 ai system. In *Proceedings of the Game Developers Conference*, volume 63, 2005.

Chris Hecker. My liner notes for spore/spore behavior tree docs, 2009.

Damian Isla. Halo 3-building a better battle. In *Game Developers Conference*, 2008.

Charles Darwin and William F Bynum. *The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life*. AL Burt, 2009.

John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. U Michigan Press, 1975.

John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

David J. Montana. Strongly typed genetic programming. *Evol. Comput.*, 1995. URL http://dx.doi.org/10.1162/evco.1995.3.2.199.

Christopher D Rosin and Richard K Belew. Methods for competitive co-evolution: Finding opponents worth beating. In *ICGA*, 1995.

Angeline and Pollack. Competitive environments evolve better solutions for complex tasks. In *Genetic Algorithms: Proceedings of the Fifth International Conference*, 1994.

Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*. Morgan Kaufmann, 1995.

Terence Soule and James A Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*. IEEE, 1998.

W. B. Langdon and R. Poli. Fitness causes bloat. In *Soft Computing in Engineering Design and Manufacturing*. Springer-Verlag London, 1997. URL http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.bloat_wsc2.ps.gz.

Raphael Crawford-Marks and Lee Spector. Size control via size fair genetic operators in the PushGP genetic programming system. In *GECCO 2002*. Morgan Kaufmann Publishers, 2002.

Riccardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic programming*, pages 204–217, 2003.

Alex J. Champandard. Understanding behavior trees, 2007. http://aigamedev.com/open/article/bt-overview/.

Petter Ogren. Increasing modularity of uav control systems using computer game behavior trees. In *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.

Rahib H Abiyev, Nurullah Akkaya, and Ersin Aytac. Control of soccer robots using behaviour trees. In *Control Conference (ASCC), 2013 9th Asian*. IEEE, 2013.

Emmett Tomai and Roberto Flores. Adapting in-game agent behavior by observation of players using learning behavior trees. In *Proceedings of the 9th International Conference on the Foundations of Digital Games (FDG 2014)*, 2014.

Andreas Klöckner. Interfacing behavior trees with the world using description logic. In *AIAA conference on Guidance, Navigation and Control, Boston*, 2013.

Alexander Shoulson, Francisco M Garcia, Matthew Jones, Robert Mead, and Norman I Badler. Parameterizing behavior trees. In *Motion in Games*, pages 144–155. Springer, 2011.

Alejandro Marzinotto, Michele Colledanchise, Colin Smith, and Petter Ogren. Towards a unified behavior trees framework for robot control. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5420–5427. IEEE, 2014.

Michele Colledanchise, Alejandro Marzinotto, and Petter Ogren. Performance analysis of stochastic behavior trees. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 3265–3272. IEEE, 2014.

Chris Miles, Juan Quiroz, Ryan Leigh, and Sushil J Louis. Co-evolving influence map tree based strategy game players. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*. IEEE, 2007.

Pablo Garcia-Sanchez, Alberto Tonda, Antonio M Mora, Giovanni Squillero, and JJ Merelo. Towards automatic starcraft strategy generation using genetic programming. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 284–291. IEEE, 2015.

Alexandros Agapitos, Julian Togelius, and Simon Mark Lucas. Evolving controllers for simulated car racing using object oriented genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1543–1550. ACM, 2007.

Atif M Alhejali and Simon M Lucas. Using a training camp with genetic programming to evolve ms pac-man agents. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 118–125. IEEE, 2011.

Bentley James Oakes. *PRACTICAL AND THEORETICAL ISSUES OF EVOLVING BEHAVIOUR TREES FOR A TURN-BASED GAME*. PhD thesis, Citeseer, 2013.

Noor Shaker Julian Togelius, Sergey Karakovskiy. Marioai, 2012. URL http://www.marioai.org/.

Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *Applications of evolutionary computation*. Springer, 2011.

Robin Baumgarten and Simon Colton. Case-based player simulation for the commercial strategy game defcon, 2007.

Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game defcon. In *Applications of evolutionary computation*. Springer, 2010.

Kirk YW Scheper, Sjoerd Tijmons, Cornelis C de Visser, and Guido CHE de Croon. Behavior trees for evolutionary robotics. *Artificial life*, 2015.

Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and A Chircop. Ecj: A java-based evolutionary computation research system. *Downloadable versions and documentation can be found at the following url: http://cs. gmu. edu/eclab/projects/ecj*, 2006.

Chris Simpson. Behavior trees for ai: How they work, 2014.

Badlogicgames. libgdx, 2016. URL https://libgdx.badlogicgames.com/.

Kumar Chellapilla. A preliminary investigation into evolving modular programs without subtree crossover. In *Genetic Programming 1998: Proceedings of the Third*, pages 23–31, 1998. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/gp1998/chellapilla_1998_piempwsx.pdf.

John R Koza. Genetic programming ii: Automatic discovery of reusable subprograms. *Cambridge, MA, USA*, 1994.

Chris Gathercole, Peter Ross, et al. Small populations over many generations can beat large populations over few generations in genetic programming. *Genetic programming*, 97, 1997.

Matthias Fuchs. Large populations are not always the best choice in genetic programming. In *GECCO*, pages 1033–1038. Citeseer, 1999.

Yuming Zou and Paul E. Black. perfect binary tree, 2008. URL `http://www.nist.gov/dads/HTML/perfectBinaryTree.html`.

Nicholas Freitag McPhee and Nicholas J Hopper. Analysis of genetic diversity through population history. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112–1120. Citeseer, 1999.

Maarten Keijzer. Efficiently representing populations in genetic programming. In *Advances in genetic programming*, pages 259–278. MIT Press, 1996.

Una-May O'Reilly and Franz Oppacher. *An analysis of genetic programming.* Carleton University, 1996.

Kim Harries and Peter Smith. Exploring alternative operators and search strategies in genetic programming. *Genetic Programming*, 97: 147–155, 1997.

Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. *Genetic Programming*, 97:240–248, 1997.

Leonardo Vanneschi, Mauro Castelli, and Sara Silva. Measuring bloat, overfitting and functional complexity in genetic programming. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 877–884. ACM, 2010.

Firaxis & 2kGames. Civilization v, 2010. URL `http://www.civilization5.com/`.