Department of Computing
Imperial College London

An A.I. Player for DEFCON: An Evolutionary Approach Using Behavior Trees

BEng

Final Year Individual Project

Final Report

By: Chong-U Lim

Supervisor: Dr. Simon Colton

2nd Marker: Prof. Ian Hodkinson

June 26, 2009

http://www.doc.ic.ac.uk/~cl2006/project

Abstract

Video games have become technologically more advanced over the years - exhibiting realistic graphic capabilities coupled with engaging game play. Artificial Intelligence (AI) in video games has arisen as an important factor in determining the quality of the game. With increasing demand for more realistic computer controlled players, events and opponents capable of exhibiting human-like characteristics, AI has been a significant area of interest for the games industry and academics alike. Several methodologies exist to approach the modeling, scripting and the design of AI for video games, aiming to make game-controlled entities more realistic and intelligent. Behavior trees attempt to improve upon existing AI techniques by being simple to implement, scalable, able to handle the complexity of games, and modular to improve reusability. This ultimately improves the development process for designing game AI.

In this project, we use behavior trees to design and develop an AI-controlled player for the commercial real-time strategy game DEFCON. We approach the design of the behavior trees using a methodology known as Behavior Oriented Design. We also intend for the AI to adapt and learn to play the game of DEFCON by allowing it to evolve into a competitive player using evolutionary machine learning techniques. The project aims to showcase the feasibility of combining such machine learning techniques together with behavior trees as a practical approach to developing a AI-bots in games.

Acknowledgments

I would like to sincerely thank my supervisor, Dr. Simon Colton, for all the support, guidance and help he has rendered throughout the entire duration of this project. I am extremely grateful for the huge amount of confidence and belief that he placed in me to undertake this project to the best of my ability.

I am also grateful to Professor Ian Hodkinson, for agreeing to be my second marker, and also for providing an alternative perspective to approach my project with.

A huge amount of thanks goes to Robin Baumgarten for all the help he has rendered throughout this project, especially with questions and problems encountered with the DEFCON API. Thanks for being so approachable and a joy to work with.

I am extremely thankful for my extremely loving and supportive parents, who have given me the opportunity to study at a great university, and for always supporting me in everything I do. I also thank my brother, Andrew, for all his encouragement, and for always being an inspirational figure to me.

Finally, very special thanks goes to Adeline Bay for the constant concern, care and support, and for always being there for me.

Contents

Chapter 1

Introduction

1.1 Motivation

An AI-bot is the term used to describe an automated player that is able to play a computer or video game in place of a human player. Designing an AI-bot for computer games is a complex task [10], but a challenging and rewarding area of the application of AI [20]. The ability of AI in games to deliver an engaging experience has become an important aspect of game development in the industry, and as such, numerous techniques and methods have been invented and developed in order to deliver realistic game AI. However, as Jeff Orkin remarked, that if the audience of the Game Developers Conference (2006) were to be polled on the most common A.I techniques applied to games, one of the top answers would be Finite State Machines (FSMs) [28]. As we will cover in Section 2.2, FSMs and its variants have limitations in developing game AI, and a more flexible approach to game AI development is required to handle the complexity of AI in modern games. Another area of interest is the application of learning to game AI, including areas such as learning to play the game, learning about players and optimizing for adaptivity [3]. With this in mind, we outline the motivations of the project in order to address these issues.

1.1.1 Game AI Development using Behavior Trees

Behavior trees have been proposed as a new approach to the designing of game AI. Their advantages over traditional AI approaches are being simple to design and implement, scalability when games get larger and more complex, and modularity to aid reusability and portability. Behavior trees have recently been adopted for controlling AI behaviors in commercial games such as first-person-shooter Halo2 [15] and life-simulation game Spore [14]. In this project, we wish to study the application of behavior trees in the implementation and design of an automated player for DEFCON, a commercial real-time strategy game.

1.1.2 Machine Learning in Commercial Video Games

Despite being applied to computer games almost 30 years ago [31], and an increased awareness of the application of AI as a necessary ingredient to make games more entertaining and challenging over the years, the application of machine learning techniques as a means of developing

AI for commercial games has had less of an impact than expected [3]. With various notable projects succeeding in the application of machine learning techniques to game AI, such as Artificial Neural Networks [35] and Case-Based Reasoning [2] - we would like to investigate the feasibility of applying machine learning to develop competitive AI players in games. Notably, the application of genetic programming has seen positive results in the fields like robotics [22] and computer board games like Chess [13]. An investigation into the feasibility of the application of genetic algorithms in the development of a competitive automated AI player for a current-generation commercial game would exemplify it as a viable means of AI development that may be adopted by the video games industry.

1.2 Contributions

We outline the following contributions made in the development and delivery of this project as follows:

- The development of a simple, but extensible behavior tree framework allowing the development of automated AI players. We demonstrate the ability of the framework to control and dictate the AI for the commercial real-time strategy game DEFCON, by Introversion Software, when used in conjunction with the DEFCON Application Programming Interface (API).

- The application of the Behavior Oriented Design (BOD) methodology to behavior trees in the designing of a goal-oriented AI player for DEFCON, highlighting the practicalities of BOD as an approach to developing an AI player that is able to handle the complexities of a commercial game and run as a fully functional player. We make use of reactive plans in our implementation, to allow the automated AI player to perform dynamic planning to react differently based on game-play conditions using goal-prioritisation.

- Demonstration of the feasibility of incorporating genetic algorithms together with behavior trees as a method of evolving a competitive AI player for the game of DEFCON. Competitiveness, in this context, refers to the evolved AI's ability to beat the standard hand-designed AI written by the programmers at Introversion Software by more than 50% of the time over a large number of games.

- A testing environment that allowed the distribution of processing over multiple computers in order to reduce the time required for the evolution of the AI, dividing the number of required runs of DEFCON between the computers. We also introduced redundancy handling and illustrate the ability to automate the process as much as possible. Due to resource constraints, the approach was implemented over 20 computers, but is scalable to any number of computers, limited by the population size required in each generation of the evolution process.

- Aiding in the improvement and refinement of the DEFCON API by identifying areas in which functionality would be appropriate for AI developers to be able to make full use of the API and to further encourage its adoption by hobbyists and members of academia as a platform for the study and development of AI in games.

1.3 Report Structure

We first provide an overview of the sources of inspiration and background information which pertain to this project (Chapter 2). We then present the design choices for the behaviour trees, how they were used to architect a hand-crafted AI-bot for DEFCON and how we proposed to evolve them (Chapter 3). We then use the design choices made to implement our AI-bot, evolve its behaviour trees and distribute the computation over multiple workstations (Chapter 4). The experiments for which we evolved our behaviour trees are then outlined (Chapter 5), following which, we present their results (Chapter 6) and then proceed to provide an analysis of the results (Chapter 7). Finally, we conclude on the objectives met by this project and proposed areas for future work (Chapter 8).

Chapter 2

Background

In this chapter, we present information from various sources that provide a deeper understanding of the methodologies, applications and technical information which were used over the course of this project. We introduce the commercial real-time strategy game DEFCON in Section 2.1, covering gameplay, game rules and an overview of existing automated AI players that have been developed in the past. We also introduce the Application Programming Interface that was used for this project.

Behaviour Trees, hierarchical structures used to encode AI, are explained in Section 2.2, where we cover definitions, common design approaches and provide examples of their application to AI development. Behaviour-Oriented Design (BOD) is introduced as a programming methodology to approach AI behavioural design. We introduce its motivation, design process and introduce the idea of reactive planning in Section 2.3. We also outline certain correspondences between BOD and behaviour trees.

Genetic Algorithms are introduced in Section 2.4. We outline the general methodology, operators and the concept of fitness functions as they pertain to genetic algorithms in this section. Genetic Programming, where genetic algorithms are applied to computer programs, are also introduced. Finally, we introduce a clustering algorithm known as the k-means algorithm in Section 2.5.

2.1 DEFCON

DEFCON is a multiplayer real-time strategy game that allows players to take the roles of the military commanders of one of six possible world territories. Players are given a set of units and structures at their disposal, and it is up to them to manage these resources and inflict the biggest amount of damage against opposing players. The game is split into 5 discrete intervals, named from DEFCON5 to DEFCON1, and these intervals dictate the movements and manipulation of units that are allowed. DEFCON is described as a *genocide*-em-up, a play on the term *shoot*-em-up and *beat*-em-up often used to classify other game genres, such as first-person-shooters and action games respectively, and since a large element of the game of DEFCON involves the use of nuclear missiles to annihilate large populations of opponent players, it is an appropriate term. Battles are fought in the sea and air, and the game involves strategic planning and decision making in coordinating all these units in order to win. Figure 2.1 shows an in-game screenshot of a coordinated attack in DEFCON, taken from DEFCON's official website[1].

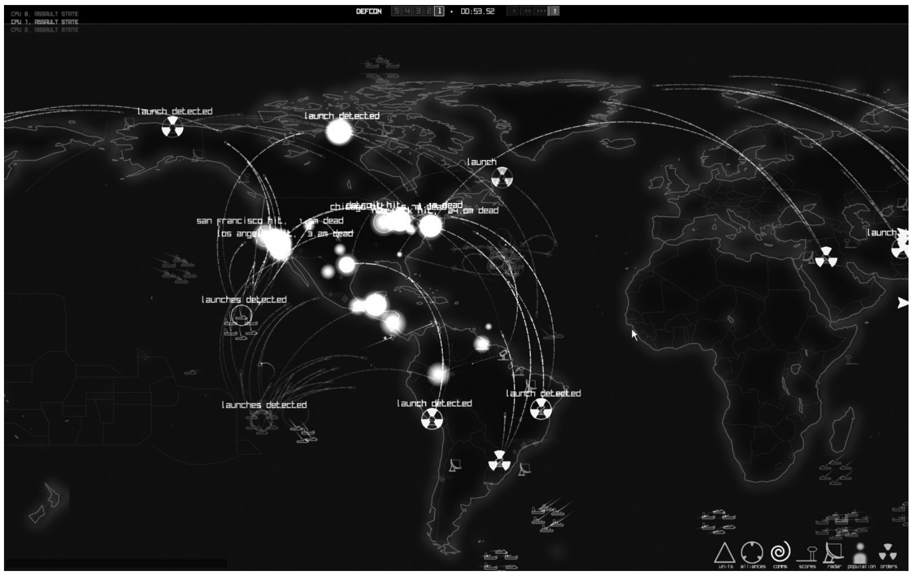


Figure 2.1: Global conflict erupts with a series of co-ordinated strikes in DEFCON

In this section, we begin by providing an overview of the game, covering basic game rules and game play elements. We proceed to present existing automated players, or AI-bots, which are presently available. Finally, we briefly cover the DEFCON API which was used for the purpose of developing an AI-bot for this project. Further information on the game can be obtained from both official DEFCON manual and Robin Baumgarten's MSc Thesis "Automating the Paying of DEFCON" [2].

2.1.1 Overview

Parties & Territories

A **party** represents the player, either human or AI-controlled, in the game of DEFCON. As such, there are at least 2 parties in the game. Each party is assigned a **territory** at the start of a DEFCON match, and this allocation is either done randomly or chosen by the user.

There are 7 available **territories** in each game, and each territory is controlled by up to 1 party. As such, in a game involving two players, two territories will be controlled by one player, and the remaining five territories will remain unassigned to any party. Each controlled territory possesses at least one **city**, and each city consists of a population size. The total sum of the population for each territory is equal between all 7 player territories. The territory designates the points on the map that a player may place **ground installations** and each player may only do so in his or her respective territories. A portion of the sea, **sea-territories**, is associated with each party, into which **navy units** may be placed in.

[1]Introversion Software's Official DEFCON Site: http://www.introversion.co.uk/defcon/

Units

Each party is allocated a fixed quantity of units that it may place and make use of throughout the course of the game. Each unit possesses one or more states, which indicates the type of actions it may execute. The units are classified into 3 groups, namely,

- Ground Installations
- Navy Units
- Aerial Units

Ground Installations

- **RadarStation**: Radars scan a wide range and display enemy units within their range on the map.
- **Silo**: A silo contains 10 Long Range Ballistic Missiles (LRBMs) and can take 3 direct hits before it is destroyed. A Silo has two modes:
  - **Nuclear Launch**: A target on the map may be manually selected to launch a missile. Upon launch, it can not defend itself, and reveals its position to other players after launching its missiles.
  - **Air-Defense**: Enemy air force units and nukes in range are automatically attacked, in the decreasing priority of Nukes over Bombers over Fighters. If there are several of the same class, it chooses the closest to its position.
- **Airbase**: Each airbase starts with 5 bombers, 5 fighters and 10 Short Range Ballistic Missiles (SRBMs) by default. An Airbase has two modes:
  - **Launch Fighters**: A target within range may be targeted, in which Fighter units are launched. A recharge time of 20 seconds activates after the launch of a fighter, which prevents the launch of a subsequent launch of another fighter during that period.
  - **Launch Bombers**: A target within range may be targeted, in which bomber units are launched. A similar recharge time of 20 seconds exists for the launching of bombers.

Air Forces

- **Fighter**: Fighters may attack other Fighters and Bombers, and may be used for scouting to discover enemy installations. A Fighter has limited fuel, and automatically return to any airbase or carrier upon completion of its objective, otherwise it will crash and is lost.
- **Bomber**: Bombers carry a single SRBM, and can be used to fly and deploy nukes at close range. A Fighter has two modes:
  - **Naval Combat**: Attacks visible enemy Navy units.
  - **Missile Launch**: A missile is launched against a selected target.

Navy Units

- **Carrier**: Each carrier tarts with 5 fighters, 2 bombers and 6 SRMBs. Can also launch depth charges against submarines in the vicinity. Each has 3 modes:
  - **Fighter Launch**: Hostile units within range may be targeted and Fighters are launched.
  - **Bomber Launch**: Hostile units within range may be targeted and Bombers are launched.
  - **Anti-Submarine**: A sonar scan is performed, and enemy submarines within range are revealed, where a depth charge is released.

  .
- **Battleship**: Battleships can only attack with conventional weapons, but are extremely effective against other naval units and aircraft.
- **Submarine**: Submarines contain 5 Medium Range Ballistic Missiles (MRBMs) and are either submerged or surfaced. They are invisible to radar while submerged, but must surface to launch. They can be detected by sonar pings from carriers or other subs. Once detected or surfaced they are very vulnerable to attack. Each submarine has 3 modes:
  - **Passive-Sonar**: Submarine remains invisible to radar and can only be detected by carriers in anti submarine status and other submarines in active sonar mode. It can attack hostile naval units if they are visible to the party of the submarine.
  - **Active-Sonar**: Similar to Passive-Sonar, except in addition, the submarine creates a sonar scan, which reveals all naval units within a certain distance from the submarine.
  - **Launch Missile**: A target within range may be attacked, where a missile is launched. In this state, the submarines surfaces and is no longer invisible until all its missiles have been depleted whereby it returns to Passive-Sonar mode.

DEFCON Levels

The course of the game of DEFCON spans over the course of 5 DEFCON phases, starting from DEFCON5. In each DEFCON phase, certain actions and information are permitted whilst others are not. Figure 2.2 provides a table, giving explanations for each of the DEFCON phases and a description of what is involved in each.

Winning Conditions

Winning in the game of DEFCON involves attaining the highest score, and how these scores are calculated differently according to the *Scoring Mode* which is decided at the start of a game. The 3 scoring modes are:

- **Default**: 2 points awarded for every million of the opponent's people killed. -1 point penalty for every million people belonging to the player.

DEFCON Level	Description	Time
5	No hostile actions. Ground installations may be placed. Fleets may be placed and moved within international waters	3
4	Radar coverage will provide information on enemy units if within range	3
3	Units and ground installations may no longer be placed. Naval and Air attacks not involving Missiles are permitted	3
2	This is essentially the same as in DEFCON 3	3
1	No units may be placed, and missile attacks are now allowed. This phase continues until 80% of all missiles in the game have been launched, after which a victory timer counts down until the end of the game.	-

Figure 2.2: Summary of DEFCON levels and permitted actions

- **Genocide**: 1 point is awarded for every million of the opponent's people killed. No penalty for losing a player's own people.
- **Survivor**: 1 point is awarded for every million people surviving in the player's territory. The points start at 100 for both teams and decrease throughout the game.

2.1.2 Automated Players for DEFCON

Several implementations of automated players exist for DEFCON. These implementations of AI-controlled players are often referred to as **AI-bots**[2].

We cover two such bots in this section. Firstly, we describe the default one that ships together with the game which allows human players to pit themselves against an opponent without the need for a network connection with another human player. The second is the result of a MSc project by Robin Baumgarten [2], which was developed using a combination of AI techniques.

Introversion Bot

The default bot[3] that comes with DEFCON is a deterministic, finite-state-machine driven bot [2]. It consists of a set of 5 states and transits from one state to the next in sequence. Upon reaching the final state, it remains in it until the end of the game. This is depicted in Figure 2.3, The states and a brief description, as identified by Baumgarten [2] of what occurs in each are:

[2]For the purpose of this report, the term **AI-bot** is used to refer to any AI-controlled player in DEFCON

[3]Henceforth termed as the Introversion AI-bot

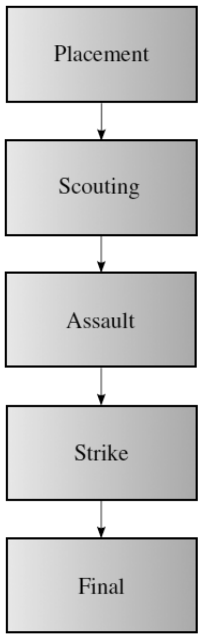


Figure 2.3: The sequence of states in the Introversion AI-bot

- **Placement**: Fleets and structures are placed. The fleet is randomly placed at predefined starting positions. Structures are placed near cities. Once all units are placed, the AI proceeds to the next state.
- **Scouting**: The AI tries to uncover structures of a random opponent by moving fleets towards occupied territories and launching fighters towards them. Once 5 structures have been uncovered, or a predefined assault timer 10 expires, the next state in invoked
- **Assault**: The AI starts to launch missile attacks with bombers and subs on the previously chosen opponent. Once 5 structures have been destroyed or the assault timer expires or the victory timer starts, the strike state is invoked.
- **Strike**: Silos launch their missiles and other missile carrying units continue to attack. After these attacks have been initiated, the system changes into the final state.
- **Final**: In the final state, no more strategic commands are issued. Fleets continue to approach random attack spots.

Robin Baumgarten's AI-bot

In 2007, a bot was developed by Baumgarten [2] using a combination of case-based reasoning, decision tree algorithms and hierarchical planning[4]. For the case-based reasoning system, high-level strategy plans for matches were automatically created by querying a case base of recorded matches and building a plan decision tree. A broad overview of the system design is depicted in Figure 2.4.

The implementation of the bot can be described as a series of steps:

[4]Henceforth referred to as Robin's Bot

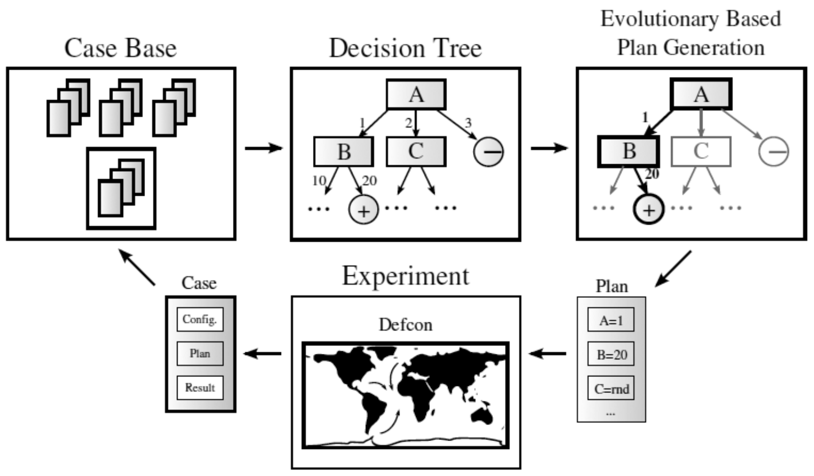


Figure 2.4: Overview of System Design of Robin Baumgarten's Bot (from [2])

1. Upon assignment of territories for both players, the starting configuration is used as similarity measure to retrieve a case from the case-base.
2. Plans from the retrieved case classified into a decision tree with plan attributes.
3. The decision tree determines a high-level plan coordinating groups of fleets, termed metafleets, and are placed as dictated by the plan.
4. The placement of structures is controlled by an algorithm taking recent matches into account by querying the case base.
5. With everything placed, an attack on the opponent's mainland is prepared establishing where and when to attack.
6. A synchronised attack is performed, and the game ends
7. The end-game results, plan used, fleet movement and structure information are extracted into the case and retained in the case base.

The coordination of fleets used in Robin's AI-bot involved the grouping of fleets into *meta-fleets*, where each meta-fleet then followed the plan that was generated for it (Step 3). Each plan described two large scale movements for these meta-fleets, dictating the placement, movement and targets of the fleets. For performing the placement of structures (Step 4), Robin's AI-bot implemented a process to analyse the performance of each silo, radar and airbase placed in each case retrieved and determined an *effectiveness* score. For example, after retrieving a case from its case-base, the system proceeded to note the individual positions that the silos were placed, calculating its effectiveness by using a weighted average of the performance of the silo in aspects such as the number of enemy missles it shot at and the amount of time it survived before being destroyed. The sum of each silo's effectiveness formed the overall score which the silo placements acquired for each case. The best performing case's silo placements were then used for the match.

The synchronised attack performed (Step 6) by Robin's AI-bot was a result of several hand-coded calculations to allow the nukes that were launched by a combination of carriers, submarines, airbases and silos to arrive at the enemy at the same time to inflict maximum damage. The implementation made use of an influence map to determine the best locations to attack and trigonometrical calculations to allow the bombers and fighters to arrive at the same time as the rest of the nukes.

We will soon cover the different approach in which we took in the development of our AI-bot. We approached the fleet coordination using a simple, but effective method (Sections 5.2.4 and 6.3) which allowed allowed it to avoid having to rigidly following a given plan (Section 2.3.2). We adopted an evolutionary approach to allow the AI-bot to learn to improve its performance with as little human intervention as possible. For example, as we will see later, the timings of the attacks made by our AI-bot were not hand-coded to be synchronised, but were a result of evolving individuals that were initially set with random timings. (Section 6.4 and Section 7.4).

Robin's AI-bot was implemented using the source-code of DEFCON, which gave it access to game state information that was required for various calculations, like the effectiveness value mentioned above. Our AI-bot implementation made use of the DEFCON API, and required us to use the API to retrieve similar game state information via an indirect means (Section 4.2.1). Chapters 3 and 4 both cover the design and implementation approaches used in the development of our AI-bot.

2.1.3 DEFCON API

A partnership between Introversion Software and Imperial College has resulted in the development of an Application Programming Interface (API) for DEFCON[5]. The API provides an interface to DEFCON's methods and function calls, and is used to produce a dynamic-link library (.dll) file that contains an AI-bot implementation which can then be included as a module for DEFCON to access to run as an automated player.

The API provides a way for AI developers to build AI-bots without having to work with the source-code of DEFCON directly. At the time of writing, the API is at version 1.51, and provides the base framework upon which this project is based. By making use of the API to develop the AI-bot for our project, we hope to contribute and improve the API, as well as encourage the adoption of the API by others to develop their own AI-bots.

2.2 Behaviour Trees

Behaviour trees provide a way to define intelligent control of characters in video games. Its key characteristics are being simple to define, scalable to exhibit complex AI and modular for reusability [8]. This is achieved by introducing a set of constructs which a behavior tree is composed of. Behaviour trees are goal-oriented, with each tree associated with a distinct, high-level goal which it attempts to achieve. Behaviour trees can be linked together, allowing the implementation of complex behaviours by first defining smaller, sub-behaviours.

In this section, we look at the motivation for behaviour trees, making comparisons to other traditional forms of defining game AI used in practice. We then cover the basic constructs that are used to create behaviour trees with examples. We have adopted a similar convention for the style and design representing the types of nodes in behaviour trees from Alex J. Champandard's

[5]API and documentation available from http://www.doc.ic.ac.uk/~rb1006/projects:api

presentation “Behavior Trees for Next-Generation Game AI” [7].

2.2.1 Motivation

A traditional approach to developing AI-controlled players for games has been to use Finite State Machines (FSM). We shall describe the use of FSMs with an example.

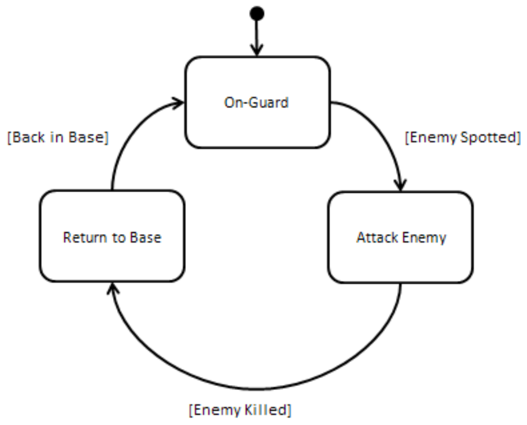


Figure 2.5: Basic FSM of a guard soldier

Figure 2.5 illustrates the FSM used to control the behaviour of a simple guard soldier. The FSM handles its ability to be on guard, attack an enemy when it spots one and return to its base once it has been able to eliminate the enemy. The FSM requires 3 states and 3 transitions in order to dictate this behaviour. Supposing we want to add an additional functionality for this guard soldier, such that whenever it is low on health, it would try to refill its health points by using a health pack. We will thus extend the FSM to include this additional state as shown in Figure 2.6.

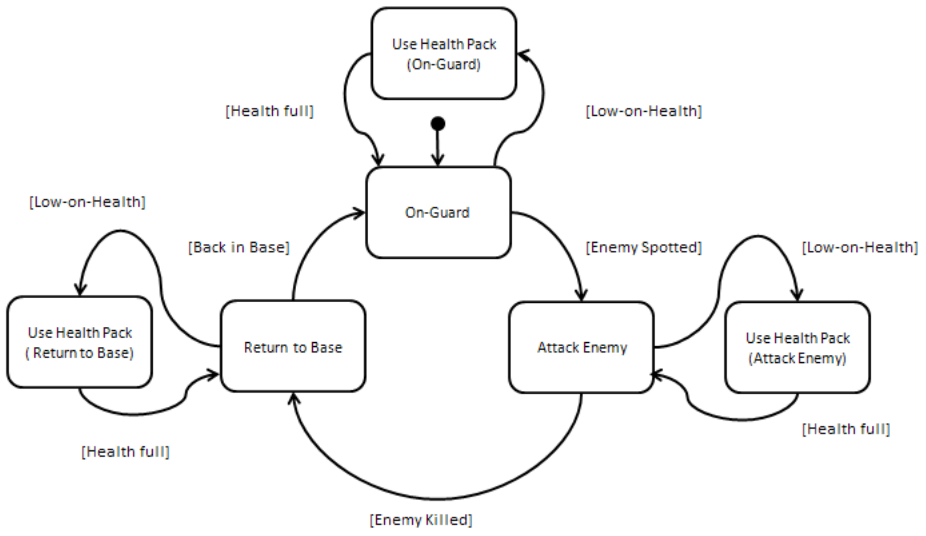


Figure 2.6: Extended FSM of a guard soldier, with an ability to heal itself

It can be seen that the number of states have now doubled in order to accommodate this new functionality. The number of transitions required to link these states has also increased. Thus, it can be seen that if the subject grows in complexity, the number of states that is required to encode the behaviour of the automated player increases, along with the number of transitions between the states. This led to the introduction of Hierarchical Finite State Machines (HSFMS), which were used to overcome these drawbacks and improve scalability by grouping states to share these transitions to develop larger and more complex AI systems. This is illustrated in Figure 2.7.

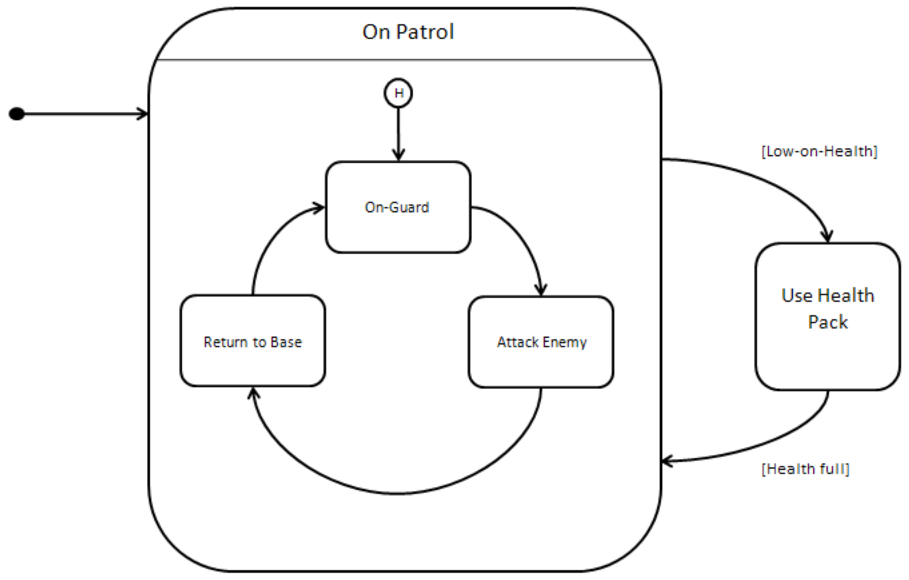


Figure 2.7: Hierarchical FSM of a guard soldier

We now have two high-level states, where the guard soldier can either be on patrol or using a health pack. Within the on-patrol state are several sub states which dictate its behaviour. We have managed to reduce the number of states and transitions that are required. However, these transitions still have to be defined carefully in order for them to be reused because, supposing we wanted to define an additional behaviour, and realise that a portion of this new behaviour consists of states that are in principle, similar to the states that were defined before. Reusing older states would involve a rather tricky task of identifying transitions which are compatible for this new set of behavioural states before providing links to transit between them.

One solution to this problem would be to allow states to be reused easily, without worrying about transitions being invalid when they are reused for different portions of the AI logic - essentially increasing the modularity. A Behaviour Tree enables such modularity by encapsulating logic transparently within the states, making states nested within each other and thus forming a tree-like structure such as in Figure 2.8, and restricting transitions to only these nested states. This exhibits what is termed as **latent computation** [7]. The root node branches down to more nodes until the leaf nodes are reached, and these leaf nodes are the base actions that define the behaviour of the AI from a state beginning at the root. The concept of an AI state is now seen as a high level AI behaviour, or task, whereby the links to nested children nodes define sub-tasks which make up the main behaviour. The leaf nodes are essentially then a group of basic actions that define a behaviour.

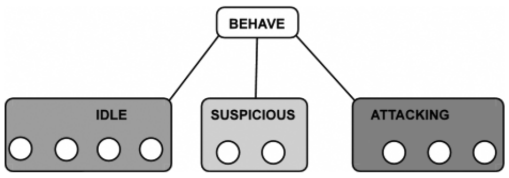


Figure 2.8: A Tree of modular Behaviours (image from [8])

These behaviour tasks are formally constructed out of 2 classes of constructs[6]. The Primitive constructs form the leaves of the tree, and define low level actions which describe the overall behaviour. The composite constructs provide a standard way to describe relationships between child nodes, such as whether all should be executed or only a subset of them. In the next section, we cover these constructs, describing in detail their usage and applicabilities.

2.2.2 Primitive Constructs

Primitive constructs form the leaves of a behaviour tree, and they define low-level basic actions. They form interface between the computations made by AI-controlled player and the main game, and are wrappers over function calls that are used to execute an action within the game world, or query the state of the game.

Actions and Conditions

Two primitive constructs consist of 2 types:

1. `Actions` - which cause an execution of methods or functions on the game world, e.g. move a character, decrease health, ...
2. `Conditions` - which query the state of objects in the game world, e.g. location of character, amount of Health points, ...

These actions and conditions form the definitions of behaviour tasks. For example, in Figure 2.9, we define a behaviour tree task DEDUCT MONEY FROM PLAYER.

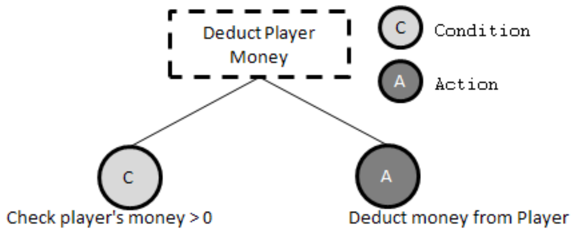


Figure 2.9: A Behaviour Task - Deduct Money From Player

[6]As such, these constructs are also referred to as **tasks** or **task nodes**

The main task is decomposed into two child tasks, the left child being a `Condition` that the player has money to start with, and the right child being an `Action` which deducts the money from the player in the game. Each of these actions and conditions can either succeed or fail, which in turn define whether the high-level behavioural task succeeds or fails.

2.2.3 Composite Constructs

We have introduced the notion of tasks, and how conditions and actions are used as child nodes to define these tasks. However, behaviour tasks of these sorts are limited to very basic tasks. We are interested in increasing the complexity of these behaviour tasks and this is achieved by building branches of the tree to organise the children nodes. These branches are essentially the composite constructs of behaviour trees, and their main purpose involves organising the children nodes of each task. The composite tasks consist of,

1. `Sequences`
2. `Selectors`
3. `Decorators`

Sequences

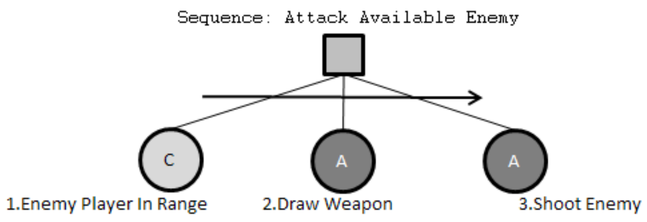


Figure 2.10: Sequence Example - Attack Available Enemy

Figure 2.10 shows a `Sequence` node, consisting of 3 children nodes, each being a primitive type. A `Sequence` essentially imposes an ordering on the execution of its children nodes. In this example, it first checks if there is an enemy in range. Secondly, it draws a weapon before finally attacking the enemy. Note that the order of execution is important, since logically, a weapon has to be drawn before it can be used to fire at an enemy. Thus, in order to determine if the entire `Sequence` has executed successfully, each one of its children nodes has to execute successfully in the order that they are specified.

If in the event that an enemy is present, but the AI player does not have a weapon to draw, the second child task node fails, resulting in an automatic failure of the entire `Sequence`. The third child node, in this case, does not need to be checked or executed. Sequences can be seen to perform implicit pre-condition checking to prevent potentially unexpected behaviour from occuring (i.e. an AI player attempting to shoot before drawing a weapon).

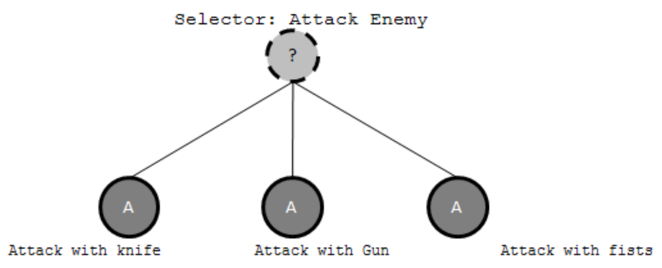


Figure 2.11: Selector - Attack Enemy

Selectors

The second type of composite we introduce is the `Selector`. Figure 2.11 shows a `Selector` node, consisting once again of 3 children nodes. The first thing to notice is that there is no formal ordering on these children nodes. The reason is because, a `Selector` essentially chooses, or picks, one of its children nodes to be executed. This child node can be selected randomly, probabilistically, or using priorities. For this example, we assume that each child node has an equal probability of being selected.

On execution of a child node, the `Selector` succeeds only if at least one of its children nodes succeed. In our example, the game AI selects from 3 alternatives to complete the task ATTACK ENEMY. If it first picks ATTACK WITH GUN, the corresponding `Action` executes. Supposing the `Action` fails, perhaps due to the AI player not possessing a gun at this point, the `Selector` node does not immediately fail, but rather, it picks the next child node to execute.

Continuing with our example, supposing the next `Action` it picks is ATTACK WITH KNIFE. Assuming the AI player possesses a knife at this point and executes the action successfully, the `Selector` node then returns as successful. The `Selector` only fails when none of its children execute successfully. This can be seen as exhibiting an opposite behaviour to that of `Sequences`. Sometimes, we assign priorities to each of the child task nodes. A `Selector` that selects its child task nodes in the order of decreasing priority is termed a `Priority Selector`.

2.2.4 Decorators

We have so far seen primitive and composite tasks which are used to design and define our behaviour trees. `Decorators` are a class of tasks inspired by the software design pattern of the same name. Just as how decorators in software design aim to wrap over an existing class, adding additional features on top of, but without modifying, the existing class, behaviour tree `Decorators` aim to extend existing defined behaviour trees to improve its functionality.

This is best illustrated with an example - consider the behaviour tree in Figure 2.10. Supposing that in order to achieve a more realistic AI, we want the AI player to exhibit this ATTACK AVAILABLE ENEMY behaviour for only a certain duration of time within the game, such that there are points when the AI player is considered off-guard, to mimic how a human player would not always choose to attack an enemy even though one were present.

If we were to try to design a behaviour tree to exhibit this behaviour, we could try modifying our existing behaviour tree to include several more `Condition` tasks linked together by

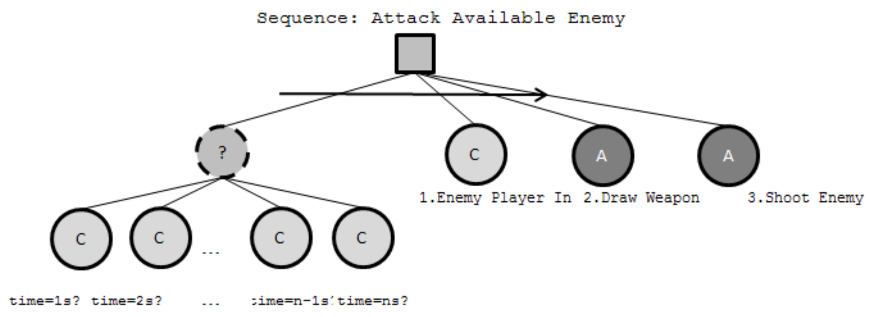


Figure 2.12: An approach to handling timed behaviours

a `Selector` into the existing `Sequence`, with each `Condition` checking for specific points of time within the game, such as in Figure 2.12. This, however,is undesirable on three accounts. Firstly, it breaks the simplicity of designing behaviour trees because of the need to tediously create multiple `Condition` nodes. Next, this is definitely not scalable because if we were to increase the time duration to execute this behaviour, more `Condition` tasks have to be added. Lastly, it breaks the rule of modularity since we are unable to reuse this Behaviour for a different time duration other than that covered by the existing `Conditions`.

`Decorators` are thus the solution to this problem. What occurs is that we define a type of `Decorator - a Timer`, which executes it's child node for a specified duration of time. The existing behaviour does not need to be modified in any way, except that the `Timer Decorator` is linked as the parent to the existing Behaviour Tree, as depicted in Figure 2.13. Now, the Behaviour is simple to create because it only involves the addition of a single `Decorator Timer`, and maintains its modularity since the original behaviour of ATTACK AVAILABLE ENEMY has not been modified in any way. Scalability does not pose a problem since there are no additional nodes to add. This exhibits the usefulness of `Decorators` to extend the functionality of behaviour trees, and more examples of types of `Decorators` include `Counters` which track and execute behaviours a certain number of times or `Debug Decorators` which output information about the corresponding child tasks. A non-exhaustive list of different types of `Decorators` can be found on page 95 of the Appendix.

Lookup Decorators

From the above discussion on `Decorators`, one noticeable point is that `Decorators` never appear as the leaves of a behaviour tree, since their functionality essentially depends on their children's tasks. However, one type of `Decorator` always appears as a leaf node and that is the `Lookup Decorator`, or `Lookups`.

`Lookups` attempt to further increase the modularity and reusability of behaviour trees by using such `Lookups` to search for particular sub-behaviour tasks rather than explicitly stating them in the tree. This can be illustrated by first considering the Behaviour Tree in Figure 2.14, describing a behaviour of an AI player in a game in which it plans to place structures on a given map or terrain.

The Behaviour takes the form of a `Sequence`. First it picks a structure to place, then it decides on a location for the structure and finally, it places it. The first two tasks involve

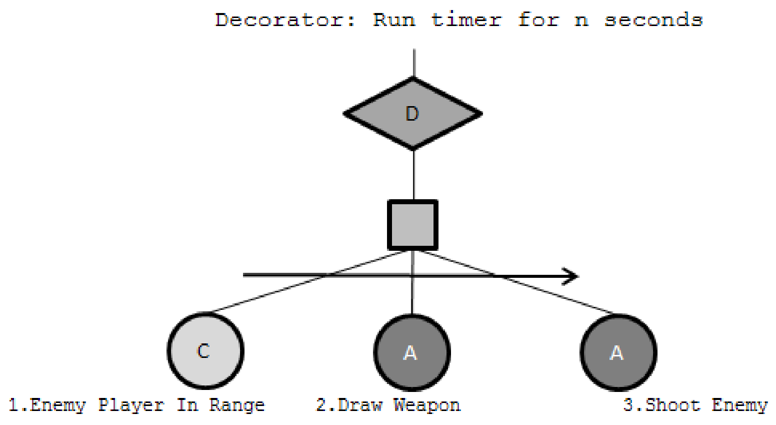


Figure 2.13: Using timer decorators to handle timed behaviours

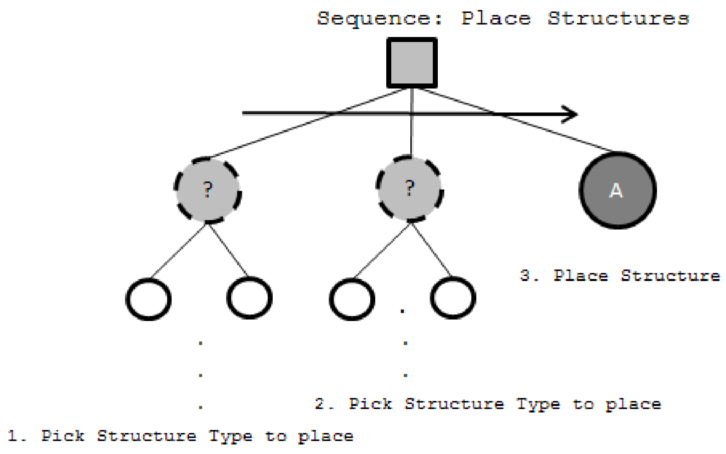

Sequence: Place Structures
?
?
A
3. Place Structure
2. Pick Structure Type to place
1. Pick Structure Type to place

Figure 2.14: Behaviour for AI Placing Structures

`Selectors`, basically because we might design the AI in such a way that it has different implementations of deciding which structure to place. An example would be picking the structure that has the fewest remaining number or an alternative could be picking one with the highest remaining number. Similarly for the second task, we have different implementations from which we wish to select from in deciding the location to place the structure. For example, placing structures nearer the sea areas, closer to cities or closer together.

We are not interested from a high level perspective about the sequence of actions which are required to execute the different implementations of these tasks, we merely want the behaviour tree to decide for us and accomplish the tasks Pick Structure Type to Place and Pick Location to Place. The idea is then to associate these different implementations into a lookup table, grouped by their high-level task which they are accomplishing, and replace the children nodes of the main `Sequence` by `Lookup Decorators`, as shown in Figure 2.15.

In grouping these implementations, into a hierarchy of tables, we begin to move towards a hierarchical goal-oriented architecture. These high level tasks can be viewed as high level goals

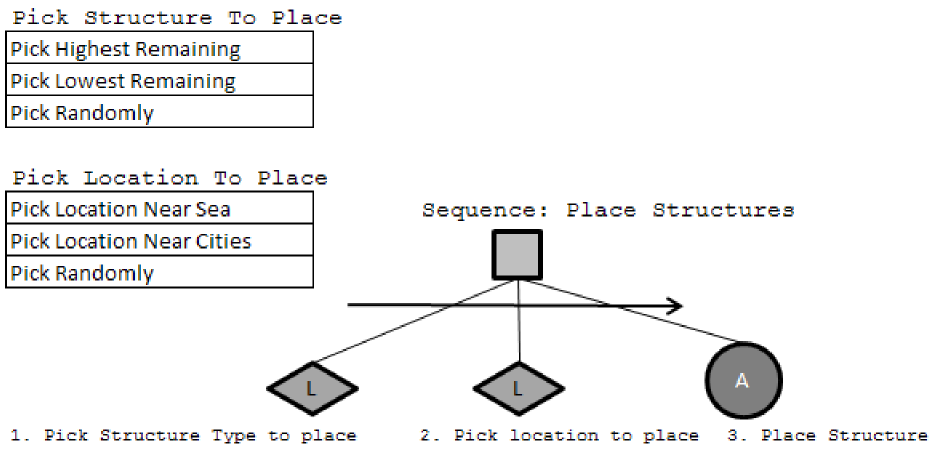


Figure 2.15: Using Lookups to modularise Behaviours

which the Behaviour ultimately wants to achieve, and does so by decomposing down to smaller sub-goals in order to achieve the high level goal. In Section 2.3, we formalise this approach using a theoretical methodogy called **Behaviour Oriented Design**.

2.3 Behaviour Oriented Design

Behaviour Oriented Design (BOD) is a development methodology proposed by Joanna Bryson [6] which aims to design and construct so called *complete complex agents*(CCA) in a modular fashion. The methodology serves as an improvement over current ardhitectures for these CCA. In this section, we provide an overview of BOD, its proposed approach to designing an AI which focuses on iterative rapid prototyping and how such an architecture would allow the AI to exhibit functionality such as reactive planning, which we will show to be desirable when attempting to design automated players.

What interests us most about BOD is its close correspondence with behaviour trees, which we introduced in Section 2.2, such as the similar functionality of BOD's **plan elements** – `Action Patterns` and `Competences`, to the `Sequences` and `Selectors` of behaviour trees respectively. We are thus able to design behaviour trees that are able to perform similar types of reactive planning that BOD allows (Section 2.3.2).

2.3.1 Methodology

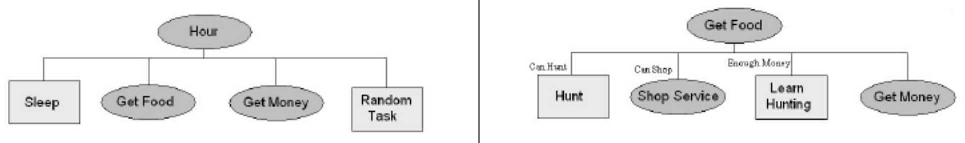
Using BOD, we approach a modular and behaviour-oriented approach to coding the AI. This is achieved by *Behaviour Decomposition* – the approach of breaking down the complex high-level behaviour into smaller sequences or groups of sub-behaviours. The problem that often arises when taking this approach is determining how exactly to decompose a behaviour. For example, we might wonder how much decomposition is necessary, how many sub-behaviours is ideal to define the top-level behaviour and also how complex should they be. BOD approaches this via a cyclic development process involving a set of guidelines for this decomposition.

We present the two main aspects of this development process, namely the *Initial Decomposition* and subsequent *Iterative Development* as proposed by Bryson [6]. This methodlogy is later used to design the hand-crafted behaviour trees for our AI-bot (Chapter 3).

Initial Decomposition

1. Specify at a high level what the agent is intended to do
2. Describe likely activities in terms of sequences of actions. These sequences are the basis of the initial reactive plans
3. Identify an initial list of sensory and action primitives from the previous list of actions
4. Identify the state necessary to enable the described primitives and drives. Cluster related static elements and their dependent primitives into specifications for behaviours. This is the basis of the behaviour library.
5. Identify and prioritize goals or drives that the agent may need to attend to, This describes the initial roots of the reactive plan hierarchy
6. Select a first behaviour to implement.

The initial decomposition process can be seen to be goal-directed, with the initial task being the high-level goal that it intends to achieve, and the sequences of actions representing the execution required to achieve this goal. Several other AI methods bear similarities to this, such as Hierarchical Task Networks (HTN) [18] and Goal Trees [17]. Figure 2.16 shows an example of a HTN, and it can be seen to bear the characteritics of BOD's task decomposition. We shall soon present and see that behaviour trees may also be designed to exhibit such goal-directed behaviours.



(a) Top level task **Hour** decomposed into sequence of subtasks and actions

(b) Sub-task **Get Food** is further decomposed

Figure 2.16: Hierarchical Task Network

The main motivation for such a goal-directed planning architecture arises from the second of two general philosophies of game AI [10]. The first holds the belief that AI has to be strictly controlled using techniques such as finite automata like Finite State Machines and scripting. The reason for this is to give the AI developers a close control over the direction of the AI, what it is enabled to do in each state or situation and for predictable behaviour to ease the process of testing and debugging.

The second philosophy, and the one which BOD pertains to, reasons that it is essentially impossible to adequately dictate and describe every possible every possible action from every possible state. We have shown earlier that this is indeed true for finite state machines. Thus, it proposes that it is better to develop a reasoning AI that understands the basic rules of the game and makes its own decision based on the world as it perceives it, in the hope that the emergent behaviour would be more effective and believable than if specified by hand.

Iterative Development

The development process for the BOD methodology involves the following steps:

1. Select a part of the specification to implement next.
2. Extend the agent with that implementation:
   - code behaviours and reactive plans, and
   - test and debug that code
3. Revise the current specification

2.3.2 Reactive Planning

Planning involves defining the sequence of actions that will achieve a goal [30]. Many of these actions are often potentially executable given a situation or event, although it might not be logically or physically possible for them to execute together. Thus, there exist the requirement to explicitly specify the conditions for which each action may be expressed, which is the case for both the Subsumption Architecture [4] and the Agent Network Architecture [24].

This requirement grows as the complexity of the given domain becomes larger with more and more behaviours added. Reactive planning attempts to address this by describing using sequences of events to describe each behaviour in what are termed as **action-selection sequences**. These sequences may be interrupted, preventing the completion of an intended action sequence, as a result of two categories of events [5]:

- Some combination of alarms, requests or opportunities may make pursuing a different plan more relevant.
- Some combination of opportunities or difficulties may require the current sequence to be reordered.

In order to address the occurrences of such events when defining a sequence in a reactive plan, BOD introduces the use of three types of **plan elements** to approach reactive planning. The three types are:

- **Action Patterns**: things that do not strictly need to be checked at all
- **Competences**: things that only need to be checked in particular context
- **Drive Collections**: things that need to be checked regularly

$$\langle \text{get a banana} \rightarrow \text{peel a banana} \rightarrow \text{eat a banana} \rangle$$

Figure 2.17: Action Pattern (Image from [5])

Plan Elements

Action Pattern: This is a sequence of primitives, which may be either actions or sensory predicates. They are useful for for allowing an agent designer to keep the system as simple as possible and speed optimizations of elements that reliably run in order. In Figure 2.17 above, the action pattern shows the behaviour of an artificial monkey.

$$\text{(have hunger)} \Rightarrow \left\uparrow \left\langle \begin{array}{rl} \text{(full)} \Rightarrow & \textit{goal} \\ \text{(have a peeled banana)} \Rightarrow & \text{eat a banana} \\ \text{(have a banana)} \Rightarrow & \text{peel a banana} \\ \Rightarrow & \text{get a banana} \end{array} \right\rangle \right.$$

Figure 2.18: Competence (Image from [5])

Competence: These encode a prioritisation in place of a temporal ordering as with normal sequences, and this allows an added flexibility in the exhibited behaviour. Figure 2.18 above illustrates a competence for the same artificial monkey, but with a prioritization of actions which increases in the direction of the vertical arrow. In this example plan, the monkey will eat a peeled banana if handed one. If given a banana it will peel it or, given none of the above, it will attempt to get a banana. The competence terminates when either the goal is achieved, or if none can be achieved.

$$\textit{life} \Rightarrow \left\langle\!\!\left\langle \begin{array}{rl} \text{(something looming)} \Rightarrow & \text{avoid} \\ \text{(something loud)} \Rightarrow & \text{attend to threat} \\ \text{(hungry)} \Rightarrow & \text{forage} \\ \Rightarrow & \text{lounge around} \end{array} \right\rangle\!\!\right\rangle$$

Figure 2.19: Drive Collection (Image from [5])

Drive Collection: These provide a means of arbitrating between plan elements or goals and handle contextual changes which occur both in the environment and internally to allow changes between rather than within plans. It is designed never to terminate and executes on every tick of the program cycle, where the highest priority drive-collection element that is triggered passes the control to the next plan element. When initialised, the focus is placed on the top most element of the plan. Figure 2.19 demonstrates this.

Basic Reactive Plans

A Basic Reactive Plan (BRP) has been identified by Bryson as an idiom or pattern found in numerous influential reactive planning architectures [5]. BRPs are based on a few assumptions, including:

- The agent has a goal to be achieved
- A set of actions exists that enables the agent to be capable of executing which can lead to achieving the goal

A prioritised list of the set of actions constitues the BRP, with the action with the highest priority being the one that consummates the goal, and each subsequent action in decreasing priority enabling the action that is more important than it.

1. (fiancé present AND in church) $\Rightarrow$ marry
2. (fiancé present) $\Rightarrow$ goto church
3. (engaged) $\Rightarrow$ goto fiancé
4. (receiving attention) $\Rightarrow$ become engaged
5. () $\Rightarrow$ flirt

Figure 2.20: A Basic Reactive Plan (Image from [5])

Figure 2.20 above shows an example of a BRP, an example covered Bryson [5]. The agent of interest has a highest goal to get married, indicated by the value 1 in the priority list on the left. One can understand the BRP as follows:

- To get married, the agent needs a fiancé and needs to be in church
- If the agent is together with her fiancé, they will go to church (Priority: 2)
- If the agent's fiancé is available,
  - If the agent were engaged, she would look for her fiancé (Priority: 3)
  - Otherwise, if the agent were not engaged,
    - but receiving attention, the agent would attempt to get engaged (Priority: 4)
    - but not receiving attention, the agent would simply flirt (Priority: 5)

The approach outlined above allows the designer to focus on the context and situation where each action is executable, rather than having to recognise the possible transitions between the action states, due to the invariant that no higher priority action would have fired when a current one is being considered. The designer only has to hand-code one situation per action required, stating the minimum conditions for which that action will be executed.

Correspondences with behaviour trees

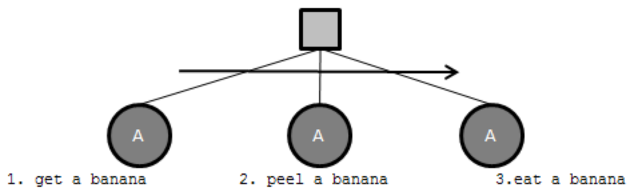


Figure 2.21: Action Pattern as a Behaviour Tree Sequence

We can see that **action patterns** can be represented in behaviour trees as `Sequences`. Figure 2.21 illustrates the corresponding `Sequence` for the **Action Pattern**.

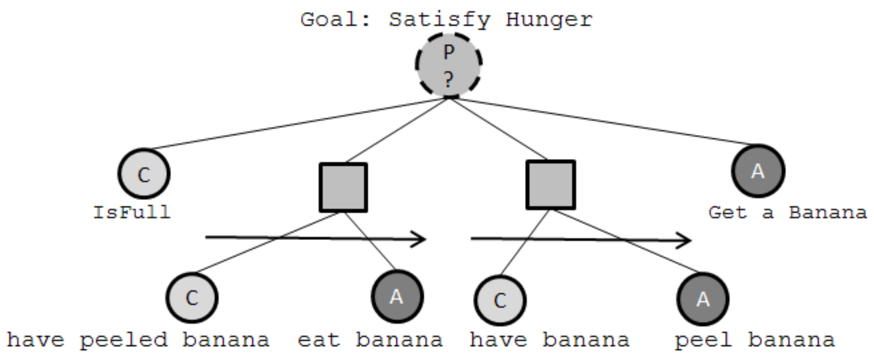


Figure 2.22: Competence as a Behaviour Tree Selector

Figure 2.22 shows the corresponding `Selector` for the **Competence**. It is worthwhile to note that the `Selector` is in fact a `Priority Selector` - where it places a priority values of decreasing value from left to right on its children task nodes. By outlining such correspondences, we identify the applicability of behaviour trees to the BOD methodology. In Section 3.2 of Chapter 3, we outline the design of the behaviour trees using the BOD approach for the automated AI player covered in this project. We also present applications of reactive planning for the purpose of coordinating behaviours for choosing appropriate times for attacking in the AI-bot.

2.4 Genetic Algorithms

Genetic algorithms are a form of machine learning that is based on natural biological evolution. In applications used for classifying data correctly, multiple candidate hypotheses are evolved and generated from preceding populations using a combination of genetic operators such as recombination and mutation which mimic the inheritance of chromosomes in organisms. The process is often repeated over numerous generations until a threshold of optimality is reached. In this section, we introduce genetic algorithms as a a machine learning approach. We begin by formally introducing the terms and concepts of genetic algorithms, including an overview of a prototypical genetic algorithm and common genetic operators. Following this, we introduce Genetic Programming as an application of such genetic algorithms to computer programs.

2.4.1 Methodology

Genetic algorithms were inspired by the biological process of evolution. It serves to find near optimal solutions to complex non-linear problems [21]. The approach usually begins by starting with an initial population of solutions to the problems. These solutions are rated on how optimal they are, termed its *fitness*, and subsequent populations are produced by a set of **genetic operators**. This is allowed to repeat over numerous runs, or **generations**, until a satisfactory performance is achieved. Informally, this can be described as a series of steps [21]:

1. Create a first-generation of the population of random organisms
2. Test them on the problem we are trying to solve, and rank them according to fitness. If the best organisms have reached our performance goals, stop.
3. We produce the offspring organisms for the next generation via any combination of the following methods
   - Carrying forward offspring from the previous generation.
   - Producing new offspring by genetic operators such as crossovers and mutation to the fittest individuals as identified in step 2. (covered in Section 2.4.4)
   - Creating entirely new offspring to introduce variety and prevent local maxima convergence.
4. With this new set of individuals, we then repeat from step 2.

Mitchell [27] covers these steps similarly using Algorithm 1 on page 31.

2.4.2 Fitness

Fitness is a numerical measure of the performance of the organisms concerning the problem at hand [27], and the fitness function is defined as an objective function that quantifies the optimality of a solution to the target problem [29], assigning a fitness value to it.

Often, the fitness function is used in the selection process as outlined in Algorithm 1 above, to provide a means for ranking each candidate in the population based on their fitness. Selection

Algorithm 1 Prototypical Genetic Algorithm

```
1:
2: INPUT : $Fitness$ {A function that assigns an evaluation score, given a hypothesis}
3: INPUT : $Fitness\_threshold$ { A threshold specifying the termination criterion}
4: INPUT : $p$ {The number of hypotheses to be included in the population }
5: INPUT : $r$ {The fraction of the population to be replaced by Crossover at each step}
6: INPUT : $m$ {The mutation rate}
7: {Initialise population}
8: $P \Leftarrow Generate\, p\, hypotheses\, at\, random$
9: {Evaluate}
10: for each $h$ in $P$ do
11:     $Compute)\, Fitness(h)$
12: end for
13: while max($Fitness(h)$ less than $Fitness\_threshold$ do
14:     {Create new generation $P_S$ }
15:     $1. Select :\ Probabilistically\, select\, (1-r)(p)\, members\, of\, P\, to\, add\, to\, P_S$
16:     $2. Crossover :\ Probabilistically\, select\, (\dot{r}p)/2\, pairs\, of\, hypotheses\, from\, P.$
17:         $For\, each\, pair,\, produce\, two\, offspring\, by\, applying\, the\, Crossover\, operator.$
18:         $Add\, all\, offspring\, to\, P_S$
19:     $3. Mutate :\ Choose\, m\, percent\ the\, members\, o\, P_S\, with\, uniform\, probability.$
20:         $For\, each,\ invert\, one\, randomly\, selected\, bit\, in\, its\, representation.$
21:     $4. Update :\ P \leftarrow P_S$
22:     $5. Evaluate :$
23:     for each $h$ in $P$ do
24:         $Compute)\, Fitness(h)$
25:     end for
26: end while
27: return $hypothesis\, with\, highest\, fitness$
```

within a population based on fitness attemps to select “fitter” candidates over others, whether for direct insertion to the next generation, or to generate offsprings with the use of genetic operators prior to insertion. In Section 3.3, we cover the various fitness functions that were used for the evolution of the DEFCON bot used in this project.

2.4.3 Selection

Selection involves picking a subset of the existing population to carry forward to the next generation. Three common forms of selection [27] are:

- Fitness Proportionate Selection
- Tournament Selection
- Rank Selection

In *fitness proportionate selection*, individuals from the population are selected based on the ratio of its fitness to the fitness of the other individuals present in the current population. Thus, the probability of selecting an individual `x` from a population of `n` individuals is described by the equation:

$$P(x) = \frac{Fitness(x)}{\sum_i^n Fitness(x_i)}$$

In *tournament selection*, a number n of individuals are picked to compete. They are ranked according to their fitness, and are assigned a probability of being selected as follows:

$$P(x) = p \times (1-p)^{rank-1}$$

In *rank selection*, the individuals are sorted according to their fitness. The probability of selecting an individual is then proportional to its rank in this list, rather than its fitness.

2.4.4 Genetic Operators

Genetic operators are used to produce offspring for a new generation. In Genetic Algorithms, these genetic operators can be illustrated by operations on bit-string hypotheses, as illustrated by Mitchell [27]. The two common operators are identified as:

- Recombination, consisting of
  - Single-Point Crossover
  - Two-Point Crossover
  - Uniform Crossover
- Mutation

Recombination

In *Recombination*, two parents are selected and two offspring are produced by copying the i'th bit from each of the parent, where the choice of which parent to receive the bit is defined by a **cross-over** mask. By varying the mask, offspring are able to inherit varying portions from the parents, and in doing so, aids in the diversity for the next generation.

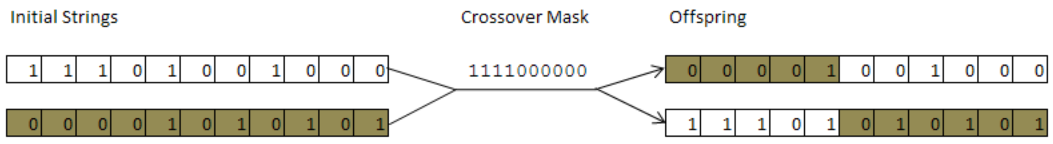


Figure 2.23: Single-point Crossover

In *single-point crossover*, the parent sequence is divided into two portions. The first offspring receives the first 5 bits from the first parent, and the remaining 6 offspring from the second -

illustrated in Figure2.23 above as the underlined bits. The second offspring does the converse, receiving the first 5 bits from the second parent and the last 6 bits from the first.

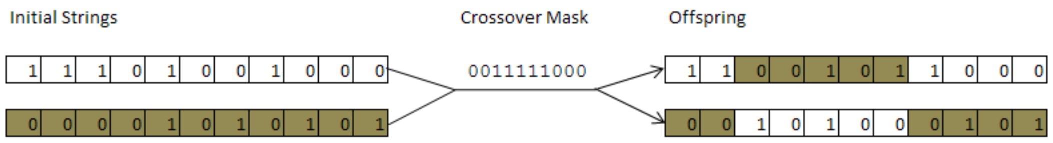


Figure 2.24: Two-point Crossover

In *two-point crossover*, the parent sequence is divided into 3 intermediate portions. The first offspring receives the first 2 bits from the first parent, the following 5 bits from the second parent, with the last 4 bits from the first once again. The second offspring can be seen to do the converse.

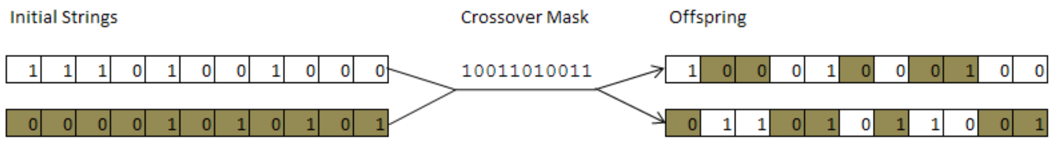


Figure 2.25: Uniform Crossover

In *uniform crossover*, the offspring receive bits uniformly sampled from either parent, usually by randomising the crossover mask. This is illustrated in Figure 2.25

Mutation

The mutation operator produces offspring by modifying just a single parent. In this case, the parent is either selected by probabilistic or uniform selection, and an offspring is produced by a single, random modification to a bit. As shown in Figure 2.26, the seventh bit of the parent is flipped from 0 to 1, and the resulting bit string is the offspring to be used in the next generation of evolution. The occurence of mutation in a population is determined by a **mutation rate**, which determines the probability of a mutation occuring. The selection of the portion of an individual that is mutated is also determined probabilistically.

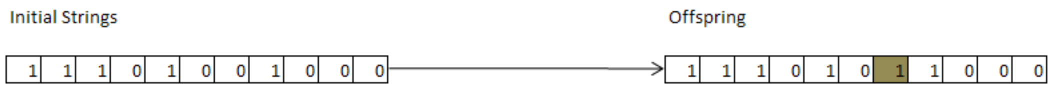


Figure 2.26: Point Mutation

2.4.5 Genetic Programming

When members of the population are no longer bit strings, but are in essence actual computer programs, the application of genetic algorithms to these programs is known as **genetic programming**. These are typically represented by trees, where nodes represent function calls and arguments form the children of these nodes. As an example, consider the trees shown in Figure 2.27.

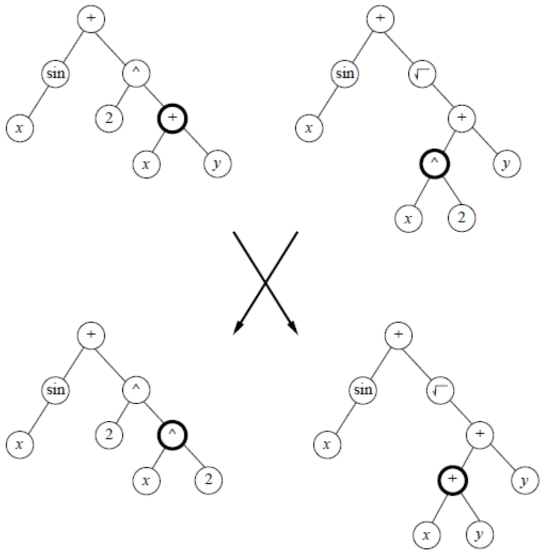


Figure 2.27: Recombination of Genetic Programs

The trees at the top denote the parent programs, and those below are the evolved offspring programs produced as a result of a recombination operations. The parent trees encode the following functions:

$$\sin(x) + 2^{(x+y)}$$

$$\sin(x) + \sqrt{x^2 + y}$$

After recombination, the offspring programs generated are:

$$\sin(x) + 2^{(x^2)}$$

$$\sin(x) + \sqrt{(x + y) + y}$$

2.5 Cluster Analysis

2.5.1 k-means Clustering

The k-means algorithm is used to perform k-means clustering, in which a set of N data points are grouped into k clusters in an I-dimensional space [23]. Each cluster k is parameterised by its mean $m^{(n)}$. For more information, there exists a good numerical example of the application of the k-means algorithm in [36]. The employed technique is the iterative refinement technique, which consists of two steps:

1. **Assignment Step**: Each of our data points is assigned to the cluster $\hat{k}^{(n)}$ with the closest mean.

$$\hat{k}^{(n)} = \underset{k}{\operatorname{argmin}} \left\{ distance(m^{(k)}, x^{(n)}) \right\}$$

2. **Update Step**: Each cluster's parameterising mean is updated to be the centroid of all the data points within it.

$$m^{(n)} = \frac{\sum_{x_i \in \hat{k}^{(n)}}}{\left| \hat{k}^{(n)} \right|}$$

The two steps are repeated until the assignments of the data points no longer change. An application of this clustering algorithm is to the coordinates on a world map, each denoted by a longitude and latitude value specifying a specific location on the map. Given two points $x = (lon_x, lat_y)$ and $y = (lon_y, lat_y)$, the distance between them can be calculated using the equation:

$$distance(x, y) = \sqrt{(lon_x - lon_y)^2 + (lat_x - lat_y)^2}$$

The application of the k-means algorithm was used in the analysis of the placement and movement positions that were resulted from the evolution of our AI-bot in this project (Chapter 7).

2.6 Summary

We have covered the game of DEFCON in this chapter, together with an introduction to behaviour trees as a tool for designing AI-bots. We then introduced the BOD methodology as a formal approach that can be used to approach behaviour tree design. These were used in the designing of the hand-crafted behaviours for the initial development of our AI-bot (Chapter 4). The application of genetic programming forms the basis of the approach to evolving these behaviour trees, which we cover in Chapter 3 (Section 3.3). Implementation details of the areas of evolution, pertaining to the automated AI bot we developed in order to improve its competitiveness as a player, are covered in Chapter 5 (Section 5.2).

Chapter 3

Design

Recall that the purpose of this project was to design and implement behaviour trees to control the behavior of an automated player for DEFCON and together with an evolutionary approach, proceed to evolve these initial hand-crafted behaviours to produce a competitive AI-bot. Therefore, in this chapter, we present the design approaches undertaken in the development of the various components. We give a schematic overview of the system design of the behaviour tree framework that was developed in section 3.1, outlining its implementation and how it was used to interface with the DEFCON API. In section 3.2, we describe the design process which was used to design and hand-craft the behaviour trees for the AI bot using the BOD methodology and a form of design denoted as goal-directed reasoning. Section 3.3 outlines the application of genetic programming to the hand-crafted behaviour trees, using genetic operators and fitness functions, for the purpose of evolving them.

3.1 Behaviour Tree Framework

3.1.1 Behaviour Tree Design

We present the behaviour tree data structure that was designed and implemented for the purpose of this project. Our implementation makes use of the **Composite Design Pattern** and the UML class diagram in Figure 3.1 shows our implementation. We take note of the following:

- The nodes of a behaviour tree are of the *abstract class* `TaskNode`.
- Two derived classes, `Primitive` and `Composite`, inherit from the `TaskNode` class.
  - Concrete classes `Action` and `Condition` are derived from `Primitive`
  - Concrete classes `Selector`, `Sequence` and `Decorator` are derived from `Composite`

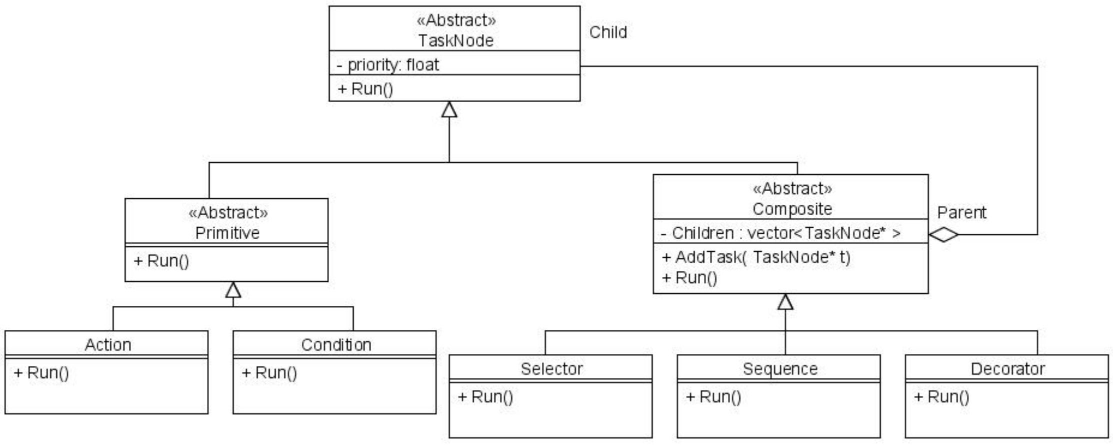


Figure 3.1: UML Diagram for behaviour trees

3.1.2 System Design

The entire system consists of two main components - the DEFCON API and the behaviour trees. As we can see from Figure 3.2, the API's Bot class makes use of the behaviour trees in order to execute AI logic and behavior. At the same time, the behaviour trees need to be able to access the API calls to perform checks on the game state as well as executing commands.

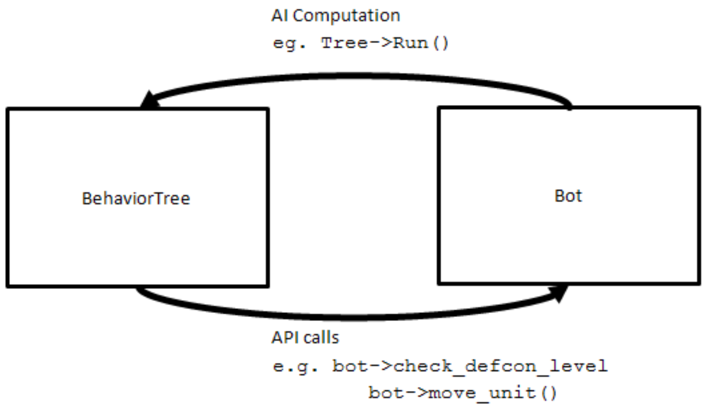


Figure 3.2: Dependence of both the behaviour tree and the Bot's API on each other

Thus, in order to adhere to one of the core dependency principles of having non-cyclic dependencies between classes, we have made use of the *abstract client pattern*, which is illustrated by Figure 3.3.

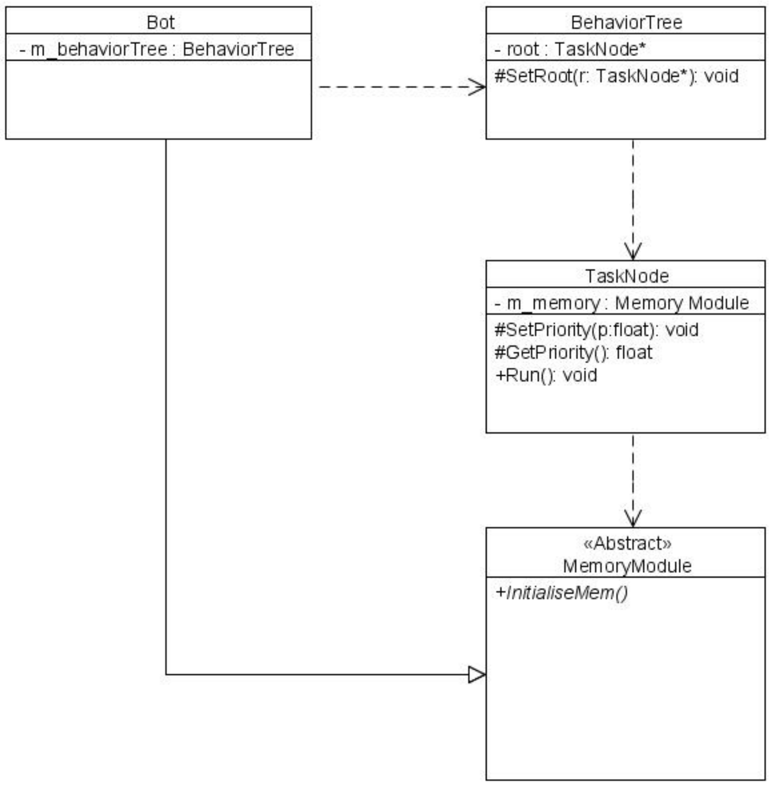

Bot
- m_behaviorTree : BehaviorTree
BehaviorTree
- root : TaskNode*
#SetRoot(r: TaskNode*): void
TaskNode
- m_memory : Memory Module
#SetPriority(p:float): void
#GetPriority(): float
+Run(): void
«Abstract»
MemoryModule
+InitialiseMem()

Figure 3.3: Abstract Client Pattern to prevent cyclic dependcies

In this case, the behaviour trees make use of what we term a **MemoryModule** abstract class, and this leads to modularizing the behaviour trees, making them reusable for any other application that may require it, and not being restricted just for the DEFCON API. An entirely seperate application, or game, which wanted to make use of the behavior trees would simply need to expose its game interface(i.e. via an API) as a derived class of the MemoryModule abstract class. After which, the application's methods and tasks would be encapsulated within `TaskNodes` specific to itself.

3.2 Encoding DEFCON

3.2.1 Goal-Directed Reasoning

To design the behaviour trees for our AI-bot, we employed the Behaviour-Oriented Design methodology introduced in chapter 2(section 2.3). We outline how the approach was applied to behaviour trees in our AI-bot development, detailing the design process for both the topology of the behaviour trees and the leaf task nodes that were used.

Designing the behaviour trees

The first thing to identify is the top-most goal that we want to achieve for our AI. Quite simply put, this top most goal would be to Win the Game. We then identify what are the sequences or groups of actions that this top level goal can be decomposed into, before recursively breaking down these subgoals down to primitive tasks such as `Actions` and `Conditions`.

This is an example of **Goal-Directed Reasoning** – we identify both a target goal state and the set of moves that might generate this goal state. The process is repeated for all predecessor states until the initial state is reached.

In Figure 3.4, we have identified our top most goal of our behaviour tree to be, as we mentioned, WIN THE GAME. This is decomposed into 5 sub-goals which are named DEFCON 5 to DEFCON 1. What this means is that, in order to achieve the top level goal of WIN THE GAME, we have to essentially *Behave Optimally in DEFCON 5*, *Behave Optimally in DEFCON 4*, ... and so on. This covers the first 2 steps of the initial decomposition process introduced in Section 2.3.1. Note that we group these set of DEFCON goals with a `Selector` node rather than a `Sequence` node, because we want the AI to choose the associated DEFCON plan for each DEFCON state the game is in. Using a `Sequence` would make the behaviour tree fail once the game has proceeded to DEFCON 4, and is not the correct behavior we wish to exhibit.

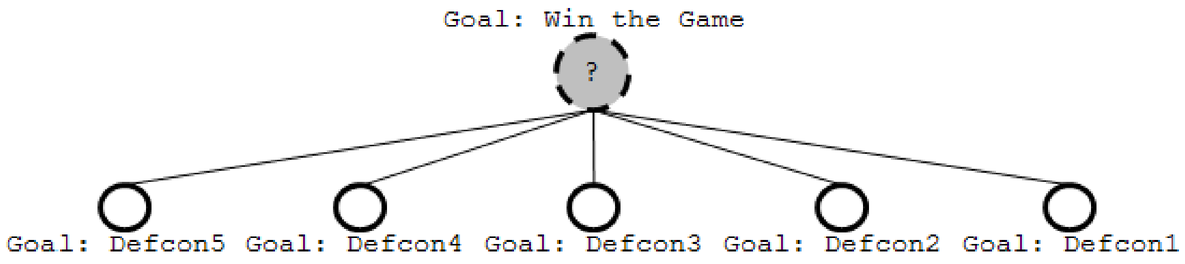


Figure 3.4: The Goal-Oriented behaviour tree with its initial decomposition

Looking at these sub-goals, we identify the need for a sensor primitive, or in behaviour tree terms, a `Condition` task node, that performs a conditional check of the current DEFCON state of the game. From here, we group these different sub-goals by registering them with lookup tables used to group and modularize the behaviour trees into a hierarchy. We may then choose to use `Lookup Decorators` to locate these trees to enable reusability. This covers steps 3 and 4 of the initial decomposition process (from section 2.3.1). Figure 3.5 shows the original tree, now including the `Condition` nodes for each branch, as well as how they might be grouped into a lookup table. Thc collection of lookup tables form the behavior library.

Finally, as we mentioned, the use of Selectors was required to accomplish steps 5 and 6. Priorities are assigned to each branch corresponding to each DEFCON state, and we assign a higher priority to DEFCON 5 and continue in a decreasing order towards DEFCON 1, since at the start, we always begin in DEFCON 5. This concludes the initial decomposition of the task WIN THE GAME.

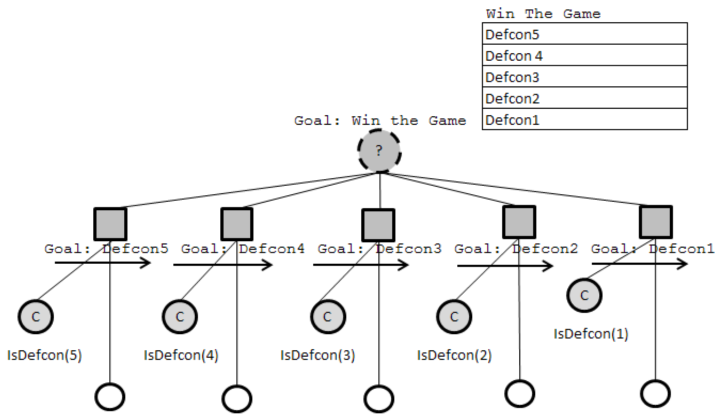


Figure 3.5: Each sub-goal introduces a set of primitives and other subgoals

Subsequent Iterations

With the completion of the initial decomposition, we test to ensure that all new introduced nodes are working correctly before moving on to the next phase of the iteration. A similar process is then replicated for each subgoal, decomposing the task further into subtasks until we reach a point when further decomposition is no longer permissible, which in the case of behaviour trees, is when we have reached the leaves of the tree where only primitive task nodes exist. We now present the complete behavior tree of one of the branches, namely DEFCON 5.

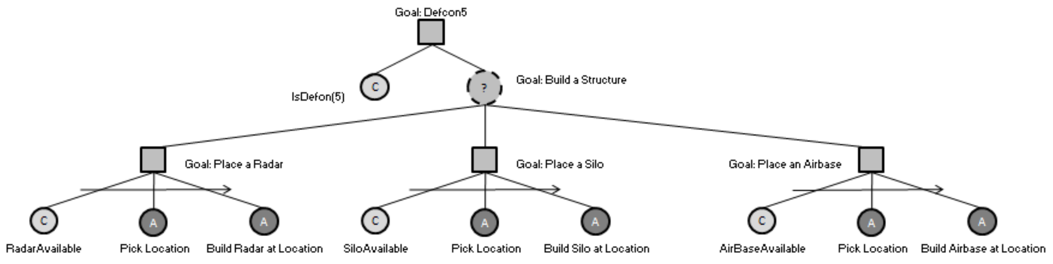


Figure 3.6: Complete behaviour tree for Task: DEFCON5

Figure 3.6 below shows the complete behavior tree for the DEFCON5 task. As we can see, the entire task consist of a sequence of actions. First, it checks whether the current DEFCON state of the game is DEFCON 5 using `Condition` IsDEFCON(5). Next, it picks a structure type to place within the game world, leaving the choice randomly to the `Selector` Build a Structure. This task is broken down into 3 sub-tasks – to place a silo, to place a radar and to place an air-base.

In order to place a structure, what involved is a `Sequence` first checking if there exist any available structures of the chosen type to be placed with a `Condition`. Next, it performs as `Action` to choose random coordinates to place the structure. Finally, upon picking the location, the AI proceeds to place the structure at the designated location with yet another `Action`.

During the designing of the task DEFCON5, we have identified both the composites required, but more importantly, the primitive task nodes that are required to be constructed. These nodes, IsDEFCON, PickLocation and Build Structure at Location were identified as required primitives - which were then constructed and added into the list of actions and primitives for the entire AI. This process was repeated for each of the DEFCON tasks, and resulted in a cumulatively constructed library of primitive `Actions` and `Conditions`.

3.2.2 Reactive Planning in DEFCON

By adopting a goal-directed, behavior-oriented approach to the designing of our behaviour trees, we now present how the application of reactive planning (discussed in Section 2.3.2) was used to allow the developed bot to perform dynamic planning in order to exhibit different behaviors at appropriate times in a simple, modular manner. We shall present an example of one of the behaviour trees which was used to dictate the attacking behavior of the submarines used during the game. Figure 3.7 below is a sub-tree taken from one of the final behavior trees used in our final bot implementation.

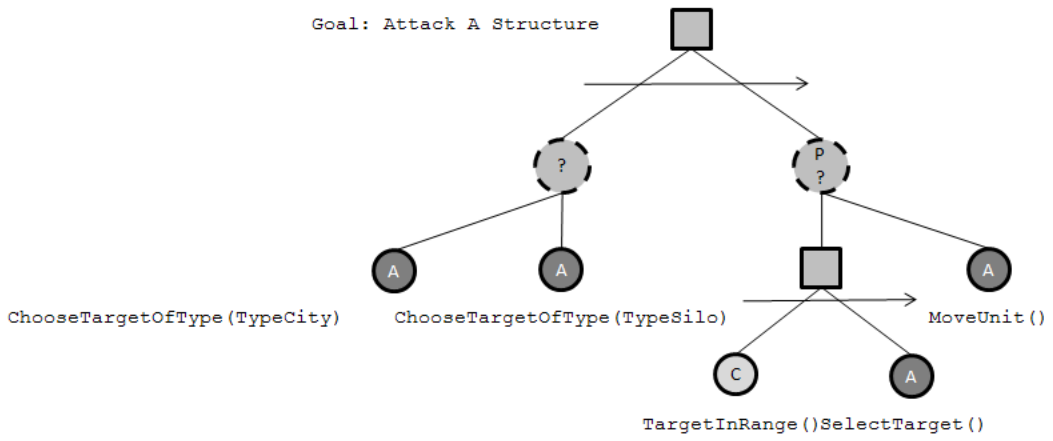

Goal: Attack A Structure
?
P
?
A
A
A
C
A
ChooseTargetOfType(TypeCity)
ChooseTargetOfType(TypeSilo)
MoveUnit()
TargetInRange()SelectTarget()

Figure 3.7: Reactive Planning to Coordinate Submarine Attacks

From this we can see that, our top most goal is to attack an enemy structure. We achieve that by a `Sequence` consisting of two sub-goals. First, we have to choose whether to attack a City or a Silo. Secondly, after making our choice, we proceed to try to attack it, shown in the right hand sub-tree of the figure. However, because the MRBMs carried by the Submarines have limited range, there is a possibility that despite picking a target, we will not be able to select the target to attack due to it being out of range. The condition node `TargetInRange` checks this. So, in order to achieve our goal, we might have to move our unit nearer the chosen target structure from our current position.

This behavior is possibly due to the `PrioritySelector` used in the right sub-tree, and because of such a prioritisation of actions, we demonstrate that this resembles the **Basic Reactive Planning** paradigm (introduced in section 2.3.2), since the action with the lower priority (Moving Submarine Closer) , aids in allowing the action with the higher priority (Attack the Target) to be able to succeed and achieve the overall goal of attacking an enemy structure. We apply this form of reactive planning in various portions of our completed behaviour trees, as we will cover in Chapter 4(section 4.1).

3.2.3 Implementing the Leaf Task Nodes

We shall now focus on the leaves of the trees which were presented which are the concrete implementations of the basic behaviour tree primitives - `Actions` & `Conditions`. We reference the tree in Figure 3.6, and using the node `IsDefcon` to describe how the tree node interfaces with the DEFCON API, it will become clear how the rest of the nodes were similarly implemented. Listing 3.1 exhibits the implementation of the corresponding `Condition` node. We have two local variables, namely the `MemoryModule` and the texttttDefconLevel in lines 5-6. The memory module contains all the game related state information, and more importantly, the pointer to the game API `m_Game`, seen to be initalised in line 18.

```
 1
 2   class IsDefcon : public Condition
 3 {
 4 private:
 5   DefconMemory* MemoryModule;
 6   Funct* Game;
 7   static string CONDITION_NODE_TYPE;
 8
 9   int DefconLevel;
10
11 public:
12   IsDefcon(DefconMemory* module, int defconlevel )
13   virtual Status Run();
14 };
15
16   Status IsDefcon::Run()
17 {
18   Game = MemoryModule->m_Game;
19
20   Status status = STA_FAILED;
21
22   if ( Game->GetDefcon() == DefconLevel )
23   {
24     status = STA_COMPLETED;
25   }
26   return status;
27 }
```

Listing 3.1: Code Implementation of `IsDefcon`

Upon invocation of the `Run()` method on the Task Node, the node performs an API call in line 22, to query the actual state of the game via the API. After this, it compares the actual in-game DEFCON level to the variable in question, namely `DefconLevel`, and then returns whether the node has Succeeded or Failed. This shows how each node in the behaviour tree encapsulates the API calls within itself, and also allows internal computation or processing prior to returning whether a node has Succeeded or not - an example of latent computation.

The remaining leaf nodes that were implemented follow a simple pattern of construction. Each node is a class containing a reference to the game via its `MemoryModule` variable. Then, it possesses a set of local variables which it requires for its computation. For example, for placing a unit, the node for the `Action` - PLACESTRUCTURE would require the type of structure that it will attempt to place in game when its `Run()` function is invoked during the execution of its corresponding behaviour tree.

3.3 Evolutionary Approach

The behaviour trees resulting from the initial phase of design was capable of playing the game of DEFCON automatically. However, we will soon see that the resulting bot did not play optimally (Section 5.2), and was not, in our definition, a competitive player. In order to achieve our goal of attaining a competitive bot, we employed the use of Genetic Programming(Section 2.4) to our resulting behaviour trees, with the intention of evolving the bot into a more competitive player. As we will demonstrate in our evaluation in Chapter 7, this approach was indeed feasible in achieving this.

We now outline the application of genetic programming to behaviour trees as a general approach in this section. The general idea is to have an initial population of games played by the Random bot, collect the information of decisions that the AI made throughout the course of the game, pick the behaviour trees of those games which the bot performed well in, and use them for the next generation of runs by the application of genetic operators. Behaviour trees are suited for this approach because, as we will show, its tree structure allows genetic operators to be performed on them suitably. Also, the trees can be explicitly viewed in order to determine how exactly the AI behaved in a given game - making an analysis of how its behavior and performance are co-related easier.

3.3.1 Genetic Operators

We now demonstrate how genetic operators (Section 2.4.4) are applied on behaviour trees. We consider the sub-tree of the tree PLACESTRUCTURES involving the placement of Silos. For the purpose of this explanation, all `Action` nodes are assumed to be `PlaceSilo(x,y)` nodes, where the arguments are labelled at each node.

Recombination

Figure 3.8 is an example of *single-point crossover*, where the dashed lines on the parent trees identify the point at which the recombination occured. From the diagram, it is clear to see that each offspring has a mixture of nodes from their parents. Figure 3.9 shows offsprings generated when *two-point crossover* was used. In this diagram, recombination occurs at two points, resulting in a greater mixture of nodes being passed from parent to offsprings. Figure 3.10 illustrates *uniform crossover*, where the nodes are combined uniformly from the two parents.

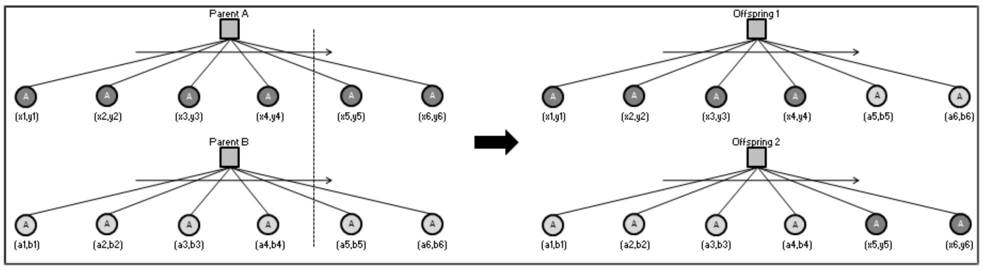

Parent A
(x1,y1) (x2,y2) (x3,y3) (x4,y4) (x5,y5) (x6,y6)
Parent B
(a1,b1) (a2,b2) (a3,b3) (a4,b4) (a5,b5) (a6,b6)
Offspring 1
(x1,y1) (x2,y2) (x3,y3) (x4,y4) (a5,b5) (a6,b6)
Offspring 2
(a1,b1) (a2,b2) (a3,b3) (a4,b4) (x5,y5) (x6,y6)

Figure 3.8: Single-Point Crossover

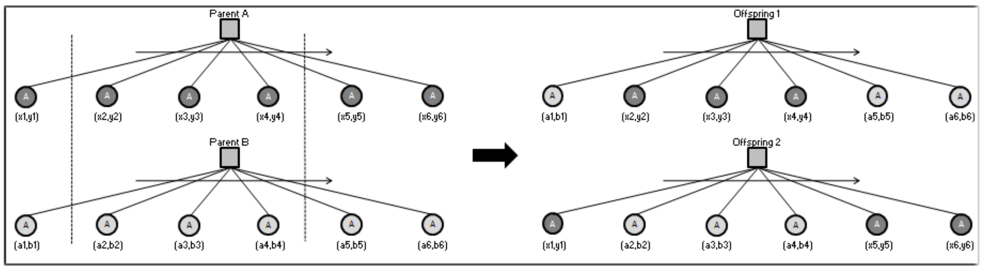

Parent A
(x1,y1) (x2,y2) (x3,y3) (x4,y4) (x5,y5) (x6,y6)
Parent B
(a1,b1) (a2,b2) (a3,b3) (a4,b4) (a5,b5) (a6,b6)
Offspring 1
(a1,b1) (x2,y2) (x3,y3) (x4,y4) (a5,b5) (a6,b6)
Offspring 2
(x1,y1) (a2,b2) (a3,b3) (a4,b4) (x5,y5) (x6,y6)

Figure 3.9: Two-Point Crossover

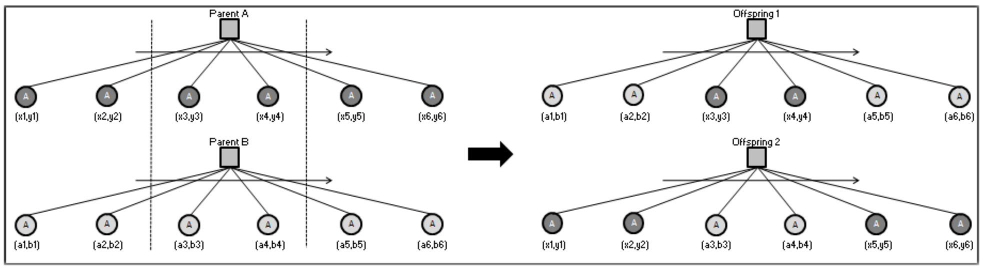
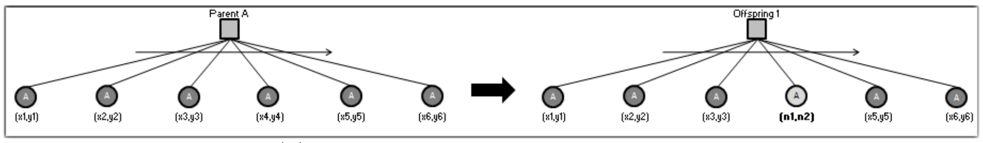

Parent A
(x1,y1) (x2,y2) (x3,y3) (x4,y4) (x5,y5) (x6,y6)
Parent B
(a1,b1) (a2,b2) (a3,b3) (a4,b4) (a5,b5) (a6,b6)
Offspring 1
(a1,b1) (a2,b2) (x3,y3) (x4,y4) (a5,b5) (a6,b6)
Offspring 2
(x1,y1) (x2,y2) (a3,b3) (a4,b4) (x5,y5) (x6,y6)

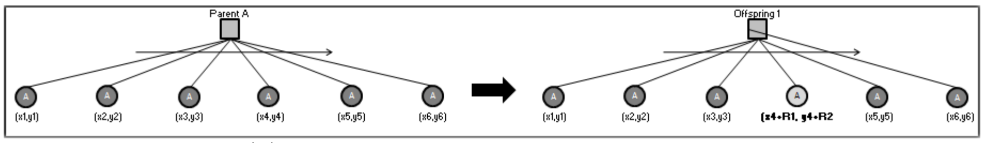
Figure 3.10: Uniform Crossover

Mutation

Mutations are employed to produce unpredictable random changes in individuals, modifying its attributes, so as to allow the individual to vary across generations. In this case, mutations can be useful to reintroduce placement coordinates that might have been lost due to being part of an unfavourable individual early in the evolution process. Mutations are also a good way to prevent against a local convergence, which tends to happen in small populations which very quickly become saturated with individuals that are locally optimal. Once again, a mutation rate would determine that percentage of the population of individuals that would be mutated.



(a) Mutation by inserting new values



(b) Mutation by modifying existing values

Figure 3.11: Mutations in Action nodes of behaviour tree

Figure 3.11 demonstrates mutation occuring for a node in the behavior tree. The coordinate values might be entirely regenerated, or perhaps, modified slightly. Figures 3.11(a) and 3.11(b) illustrate this respectively.

3.3.2 Fitness Function

As we will cover later in our experiments in chapter 5, we identify different aspects of the AI-bot's game-play which we wanted to improve. The notion of how well the bot performs during each game corresponds to the fitness of the bot in each of those game-play aspects. A natural inclination would be to simply use whether the bot won or lost as a measure of fitness for each of the identified areas of which we identified or improvement. However, the reason why we do not employ this shared fitness function is due to the fact that by decomposing the bot into the areas of improvements, the fitness measure of "winning the game" may be too distant a measure to appropriately quantify the fitness of the AI bot's behavior.

We would want to disable the portion of the behaviour tree which begins launching attacks, since we are merely interested in its defence potential. The chances of winning the game would be virtually zero in this case since the AI would not be attempting to attack at all. A different approach could then be proposed - in which all the behaviour trees of the AI are activated. In this case, even though a measure of "winning the game" would be appropriate, there are certain points to consider, including:

- A large population size would be required
- To reach an optimal solution, a large amount of time would be required as the number of generations would predictably be larger

In our approach, we have thus decided to evolve portions of the behaviour trees used for each of the identified areas of improvements. We will then attempt to combine these trees together, with the intention that by performing well in these each individual portions, the overall play of the AI would be improved.

3.4 Summary

We have covered the system design of our behaviour tree implementation which we used, outlining how the it allowed the behaviour trees to work with the DEFCON API for the development of our AI-bot. We then outlined the approach that was undertaken in the designing of the behaviour trees which were required by our AI-bot, using a combination of goal-directed reasoning and reactive planning. We then covered how evolutionary genetic operators could be applied to such behaviour trees. In the following chapter, we make use of the design choices to implement our AI-bot.

Chapter 4

Implementation

With the design of the system and its components covered in the previous chapter, in this chapter, we describe the implementation details undertaken for the development of our AI-bot. We present the completed set of hand-crafted behaviour trees and task nodes AI-bot in section 4.1, and how we made use of the eXtensible Markup Language (XML) format to store these behavior trees. Next, the implementation considerations of applying evolution to the resulting behaviour trees are covered in Section 4.2, covering an example of how a tree might be automatically generated. Finally, we introduce our distributed system which was used to divide the workload of running the DEFCON games over multiple computers in Section 4.3. This was required to reduce the computation time required to evolve the AI-bot.

4.1 Behaviour trees for DEFCON

By application of the BOD and goal-directed design process, we were able to eventually develop and produce a set of behaviour trees that were sufficient to act as an automated player for DEFCON. In summary, our final implementation consisted of:

- 10 behaviour trees
- 20 DEFCON Tasks - categorised into
  - 10 Action Task Nodes
  - 7 Condition Task Nodes
  - 3 Decorator Task Nodes

The resulting set of behaviour trees, together with the task nodes, allowed us to modularise the behavior of the AI into discrete goal-directed components, which enabled us to engage the process of evolution on them in a simple manner. This meant that we were able to disconnect the portions or subset of behaviour trees that were not required for a particular set of experiments in order to have a controlled experiment. We shall present, over the following sub-sections, the final set of behaviour trees and Task Nodes that were developed and the method of storage for the behaviour trees.

4.1.1 Completed Set of Behaviour Trees

- **PlaceStructures**: Responsible for the placement of silos, radars and airbases
- **PlaceFleets**: Responsible for the placement of fleets.
- **MoveFleets**: Responsible for the movement of placed fleets to a target destination
- **CarrierAttack**: Responsible for switching carriers to bomber launch mode and attacking the enemy at a specific time
- **AirBaseDefence**: Responsible for sending fighters to scout random locations or attack any enemy that is within radar coverage
- AirBaseAttack: Responsible for switching airbases to bomber launch mode and attacking the enemy at a specific time
- SubDefence: Responsible for switching the submarine units to active sonar mode
- SubAttack: Responsible for switching submarine units to launch mode and attacking the enemy using nukes at a specific time
- SiloNukeLaunch: Responsible for switching silos to launch mode and attacking the enemy using nukes at a specific time

We shall now go through in detail how the **SiloNukeLaunch** behaviour tree was implemented. The rest of the behaviour trees, including similar explanations can be found in page 97 of the Appendix. Figure 4.1 shows the behaviour tree used to allow the AI-bot to switch to launch mode and attack the enemy with its silos. Looking at the root of this behaviour tree, it consists of a `sequence` with two child nodes. The first checks if the current time corresponds to to the AI-bot's `SILONUKETIME`, a specific timing which the AI-bot uses to denote the time to launch an attack. We shall cover in Section 5.2.5 of Chapter 5 that this was a time which we attempted to evolve in order to improve the AI-bot's performance.

If the time within the game is less than the time specified in the `IsTime` node, then the behaviour tree fails and no further action is taken. If the game time is greater than, or equals to, the specified time, then the behaviour tree proceeds on to the second `sequence` node. It checks that the AI-bot does possess a silo, and then selects one of its silo units if it does. The next step invloves a `priority selector`. It first check the left node (since by our convention, the priority of the children in a priority selector diagram decreases from left to right), and proceeds to check if the silo which it selected is in launch mode. If it isn't, the left hand tree of the `priority selector` fails and the right branch is then run. The AI-bot will now perform the action of setting the state of the silo to its launch mode.

Supposing that the silo was already in launch mode, allowing the condition node to succeed, then following that, the AI-bot would choose an enemy city and select it as a target for a nuke attack. The pattern exhibited by the `priority selector` is another example of a basic reactive plan (Section 2.3.2).

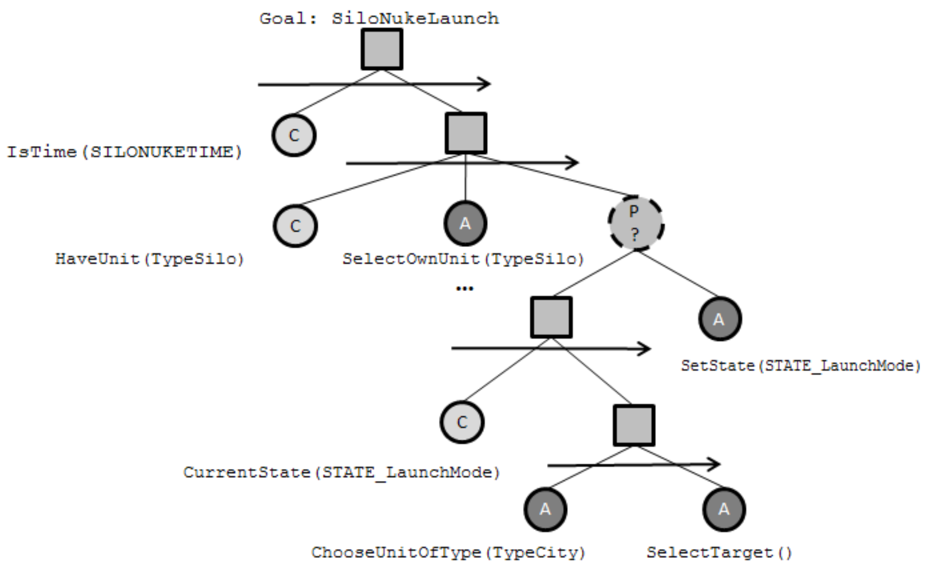


Figure 4.1: Behavior Tree: SiloNukeLaunch

4.1.2 Completed Set of Task Nodes

Actions

- **BuildFleet**: Builds a Fleet of a certain composition and at a specified location
- **ChooseTargetofType**: Randomly picks an enemy unit of specified type as the active target
- **MoveUnit**: Moves a fleet unit to a target location
- **PickFleetComposition**: Picks a fleet composition of a specified fleet unit type and saves it in memory
- **PickCoordinates**: Picks random longitude and latitude coordinates and saves it in memory
- **ChooseTargetofType**: Picks at random an enemy unit of a specified type
- **PlaceStructure**: Builds a structure of specified type at a specified location
- **SelectOwnFleet**: Selects a fleet belonging to the AI player
- **SelectOwnUnit**: Selects a unit belonging to the AI player
- **SelectTarget**: Selects specified target unit as an active target
- **SetState**: Sets the state of specified unit to a specified state

Conditions

- **CurrentState**: Returns successfully if the current unit is in the specified state. Fails otherwise.
- **HaveUnit**: Returns successfully if AI-bot posseses specified unit. Fails otherwise.
- **IsDefcon**: Returns successfully if current DEFCON level in-game is the same as the specified DEFCON level. Fails otherwise.
- **IsTime**: Returns successfully if current game time(in seconds) is the same as the specified time. Fails otherwise.
- **StateCount**: Returns successfully if the number of activations in the specified unit's target state is at least the specified number. Fails otherwise.
- **TargetInRange**: Returns Successful if the specified target is within attack range of specified unit. Fails otherwise.
- **UnitsAvailable**: Returns Successful if the AI-bot has the specified number of ground installation units to be placed. Fails otherwise.

Decorators

- **Counter**: Runs child Behavior Tree the specified number of times.
- **Timer**: Runs child Behavior Tree if in-game time is greater or equals to specified time.

4.1.3 Storage & Retrieval

Saving from behaviour trees to XML

In order to load and save constructed behaviour trees, we used eXtensible Markup Language(XML) files to encode the information of the behaviour trees used, due to its suitability for encoding nested structures such as trees.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Behavior>
 <Sequence>
  <IsTime invokeTime="0.0" enumTime="DTIME_NukeLaunch" timeFromMem="true"> />
   <Sequence>
      <HaveUnit type="TypeSilo"></HaveUnit>
      <SelectOwnUnit type="TypeSilo"></SelectOwnUnit>
      <PrioritySelector>
         <Sequence priority="10">
            <CurrentState state="STATE_SILONUKE" fromMem="true"></CurrentState>
            <Selector>
             <Sequence>
             <ChooseTargetOfType type="TypeCity"></ChooseTargetOfType>
             <SelectTarget type="TARGET_UNIT" coordsFromMem="false"
                          unitFromMem="true" fromMem="true"></SelectTarget>
             </Sequence>
            </Selector>
         </Sequence>
         <SetState state="STATE_SILONUKE" priority="2" fromMem="true"></SetState>
     </PrioritySelector>
      </Sequence>
  </Sequence>
 </Behavior>
```

Listing 4.1: XML File for Behavior Tree of SiloNukeLaunch

Listing 4.1 above shows an example of an XML file used to encode the tree for SiloNuke-Launch. As a convention, each behavior tree is encoded between the tags `<Behavior></Behavior>`. Each type of Behavior Tree node, whether composite or simple, is encoded as an XML entity, with its arguments and parameters as attributes. Some additional attributes, such as `fromMem`, `timeFromMem` and `coordsFromMem` are used to indicate if the values are to be read explicitly from the behaviour trees, or if the value should be read from the Memory Module.

To illustrate its meaning, consider the `ChooseTargetOfType` entity and `SelectTarget` entity on lines 13 and 14 respectively. What the former does is to query the game state for any target enemy city. If successful, the AI-bot requires some way of remembering which enemy city it had picked and this is done by storing the information in its memory. On execution of `SelectTarget` on line 14, note that `unitFromMem` has been set to "true". Thus, there is no need to specifically state which unit has to be targeted in this particular node and instead, the Behavior Tree AI understands that the enemy unit it should be targeting should have already been saved in memory prior to the execution of the current node. Because the entire action of choosing an enemy unit and then actively targeting it is constructed as a `Sequence`, the Behavior Tree implicitly asserts that a unit would have been selected before being targeted and attacked. Failing which, the execution would have ended after the `ChooseTargetOfType` node had failed.

Loading from XML to behaviour trees

In reverse, the process of loading a Behavior Tree from its corresponding XML file consists of a recursive method invoked when building the AI at the beginning when DEFCON is launched. This method, `BuildAndLink` is illustrated below.

Algorithm 2 Method BuildAndLink

```
Require: INPUT : $XML\_root$
1: $root\_node \Leftarrow BuildNodeFromXML(XML\_root)$
2:
3: for each child element $XML\_child$ of $XML\_root$ do
4:     {Recursively call method on child}
5:     $child\_node = BuildAndLink(XML\_child)$
6:     $child\_node.parent = root\_node$
7:     $root\_node.child = child\_node$
8: end for
9: return $root\_node$
```

4.2 Evolution System Implementation

In this section, we cover implementation details pertaining to the data collection and evolution process in order to perform the evolution of behaviour trees.

4.2.1 Data Extraction

In order to keep track of the performance of our AI bot, we needed to be able to keep trace of the state of a DEFCON game, providing information on game objects like units being created or destroyed. We made use of both the `Update()` and `CreateEvent()` methods corresponding to the in-game updates and event system respectively.

Game Update Ticks

In DEFCON, the game engine performs game update *ticks* at regular intervals, and at each of these ticks, the state information on all objects within the game is updated. The engine then proceeds to render all the game related objects based on the information computed from the last tick. As such, our AI-bot's perception of the game state had to be updated whenever such ticks occur, which corresponds together with the `Update()` method of the API.

The three main updates to the AI bot is as follows,

- `UpdateUnits`, used to
  - Keep track of units that were placed by our AI
  - Keep track of enemy units that become visible within our radar coverage
- `UpdateFleets`, maintains information regarding
  - Updated fleet positions
  - Updated fleet compositions
  - Updated fleet active targets
- `UpdateEnemyCities`, used to
  - Keep an updated list of enemy cities

Some information, such as enemy units being destroyed, were not suited to be collected during the game update because the removal of units visible during an update tick can correspond to an enemy unit simply being out of Radar coverage or an enemy unit being destroyed. An elegant solution to this problem was to make use the event system, as described next.

API Event System

The API event system allows us to keep track of in-game events such as when units are destroyed or hit, or when a launch is detected. A full list of the type of events that can be tracked is in page 95 of the Appendix.

```
1   void Bot::AddEvent( int _eventType, int _causeId, int _targetId,
2                       int _unitType, float _longitude, float _latitude)
3   {
4       string sEventType = GetEventString(_eventType);
5       int unitType = m_game->GetType(_targetId);
6       string sUnitType = GetEnumString(unitType);
7       int causeType = m_game->GetType(_causeId);
8       string sCauseType = GetEnumString(causeType);
9       int targetTeamId = m_game->GetTeamId(_targetId);
10      int causeTeamId = m_game->GetTeamId(_causeId);
11  |
```

Listing 4.2: CreateEvent method in the API

Listing 4.2 above shows the method `CreateEvent` within the API. The types used for the event system differ differently to those used within the API, and as such, lines 4-10 includes the code that was included to combined the API method calls together with the event system to collect the required information from each game. The three main events that we needed to keep track of were as follows:

- `EventNukeLaunchSilo`: To identify an enemy Silo structure that was not within our radar coverage and add it to the list of viewable objects.
- `EventNukeLaunchSub`: To identify an enemy Submarine that was not within our Radar coverage and add it to the list of viewable objects.
- `EventDestroyed`: To keep a counter of units belonging to both our AI and the enemy, in order make use of the information for our fitness functions.

4.2.2 Behavior Tree Evolution Example

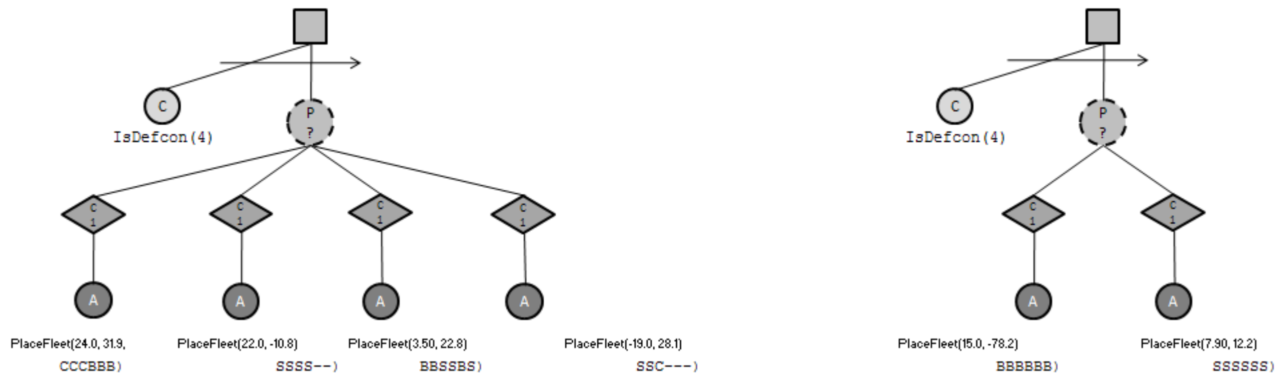

C
IsDefcon(4)
P
?
C
1
A
PlaceFleet(24.0, 31.9, CCCBBB)
PlaceFleet(22.0, -10.8) SSSS--)
PlaceFleet(3.50, 22.8) BBSSBS)
PlaceFleet(-19.0, 28.1) SSC---)
C
IsDefcon(4)
P
?
PlaceFleet(15.0, -78.2) BBBBBB)
PlaceFleet(7.90, 12.2) SSSSSS)

Figure 4.2: Two behaviour trees prior to recombination

We will cover an example of how the behaviour trees might be evolved. Figure 4.2 shows two behaviour trees which are responsible with the same goal – the placement of fleet units.

Using the left behaviour tree as a reference, what occurs is a `sequence` that first checks if the current DEFCON level is 4, and then proceeds to the priority selector if it is so. The priorities of each child node in the `priority selector` follows our convention of decreasing from left to right, and as such, the behaviour will attempt to place fleet units of the specified compositions at each of the specified locations **once**. Looking at the left most `action` node, it will attempt to place a fleet composed of 3 carriers and 3 battleships at longitude 24.0 and latitude 31.9. The parent counter `decorator` ensures that the action is only executed once. We also denote the absence of a fleet unite type with a dash "-" character.

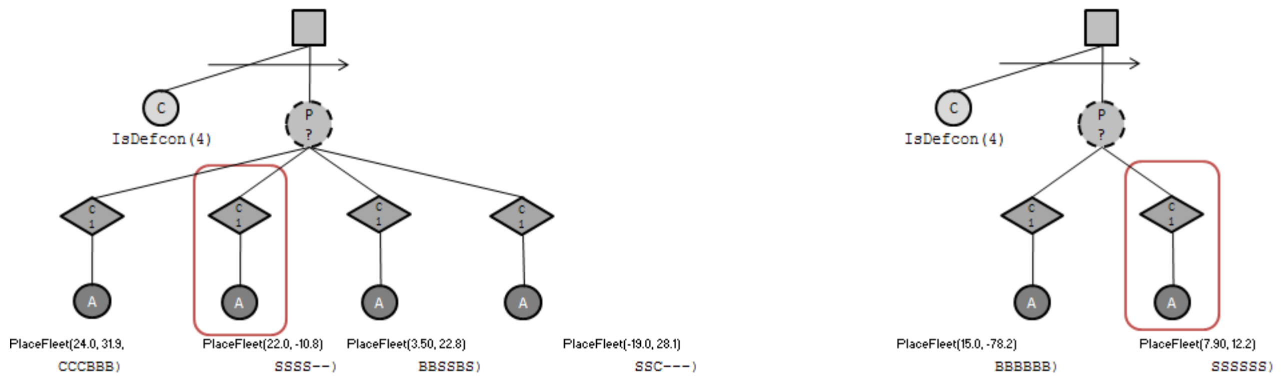

C
IsDefcon(4)
P
?
C
1
A
PlaceFleet(24.0, 31.9, CCCBBB)
PlaceFleet(22.0, -10.8) SSSS--)
PlaceFleet(3.50, 22.8) BBSSBS)
PlaceFleet(-19.0, 28.1) SSC---)
PlaceFleet(15.0, -78.2) BBBBBB)
PlaceFleet(7.90, 12.2) SSSSSS)

Figure 4.3: The same behaviour trees, with branches selected for recombination

Assuming that the two behaviour trees were part of the same population, and were selected for recombination. Figure 4.3 illustrates the two branches that were from each of the behaviour trees. A recombination occurs between the trees, causing the selected portions of the trees to cross-over. Figure 4.4 shows one of the resulting offspring from this recombination process. Do note that now, instead of placing a second fleet at (22.0,-10.8) with a composition of 4 submarines as it did in Figure 4.2, now it places a second fleet at (7.90,12.2) with a composition of 6 submarines.

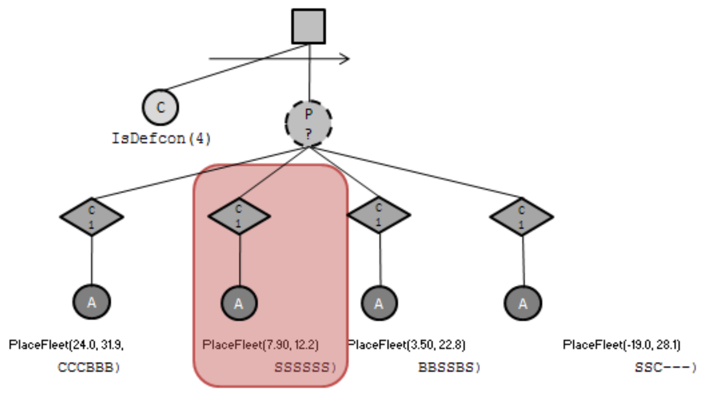

C
IsDefcon(4)
P
?
C
1
A
PlaceFleet(24.0, 31.9, CCCBBB)
PlaceFleet(7.90, 12.2) SSSSSS)
PlaceFleet(3.50, 22.8) BBSSBS)
PlaceFleet(-19.0, 28.1) SSC---)

Figure 4.4: One of the offspring from the recombination between the two parent behaviour trees.

Suppose that the evolved behaviour tree from Figure 4.4 was selected for mutation during the next evolution stage of the subsequent generation. Figure 4.5 illustrates how incremental

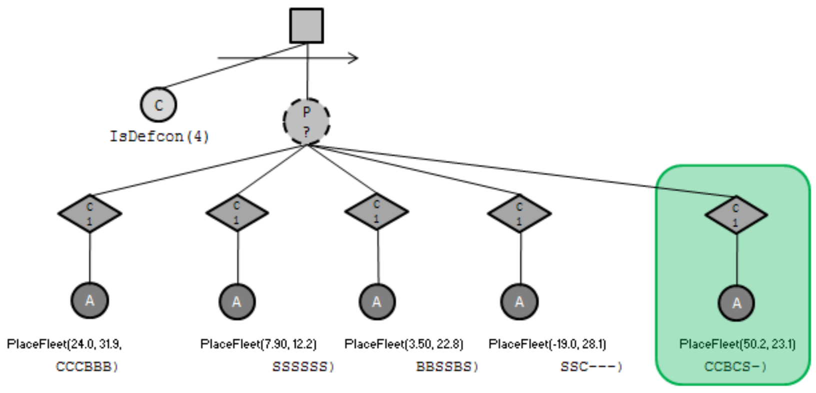

IsDefcon(4)
PlaceFleet(24.0, 31.9, CCCBBB)
PlaceFleet(7.90, 12.2) SSSSSS)
PlaceFleet(3.50, 22.8) BBSSBS)
PlaceFleet(-19.0, 28.1) SSC---)
PlaceFleet(50.2, 23.1) CCBCS-)

Figure 4.5: An incremental mutation occuring in one of the behaviour trees

mutation could occur to the behaviour tree, resulsting in a different topology. Instead of placing 4 fleet units, the behaviour tree will now cause the AI-bot to place 5 fleet units. The longitude, latitude and fleet compositions used for the new branch (highlighted in green) were generated randomly during this mutation process.

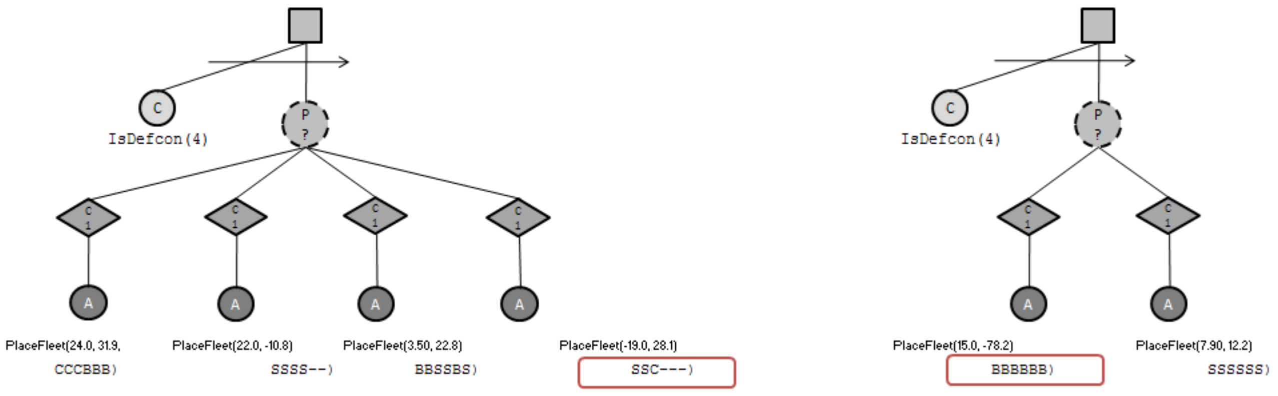

IsDefcon(4)
PlaceFleet(24.0, 31.9, CCCBBB)
PlaceFleet(22.0, -10.8) SSSS--)
PlaceFleet(3.50, 22.8) BBSSBS)
PlaceFleet(-19.0, 28.1) SSC---)
IsDefcon(4)
PlaceFleet(15.0, -78.2) BBBBBB)
PlaceFleet(7.90, 12.2) SSSSSS)

Figure 4.6: Two compositions selected for recombination from parent behaviour trees

Finally, we consider a different example of how recombination could have been applied to the two behaviour trees from Figure 4.2. Consider the scenario where recombination was applied to the `PlaceFleet` nodes in such a way that a crossover of fleet compositions would occur. Figure 4.6 illustrates the two nodes whose fleet compositions have been selected for recombination. A cross over between the fleet compositions might then occur in the manner illustrated in Figure 4.7. Figure 4.8 shows the offspring behaviour trees after the recombination. The fourth fleet placed by the behaviour tree will be at the same location, but will consist of a different composition this time.

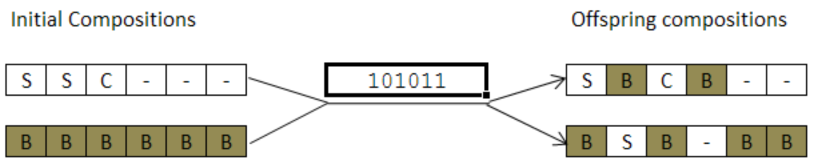


Figure 4.7: The recombination of fleet compositions

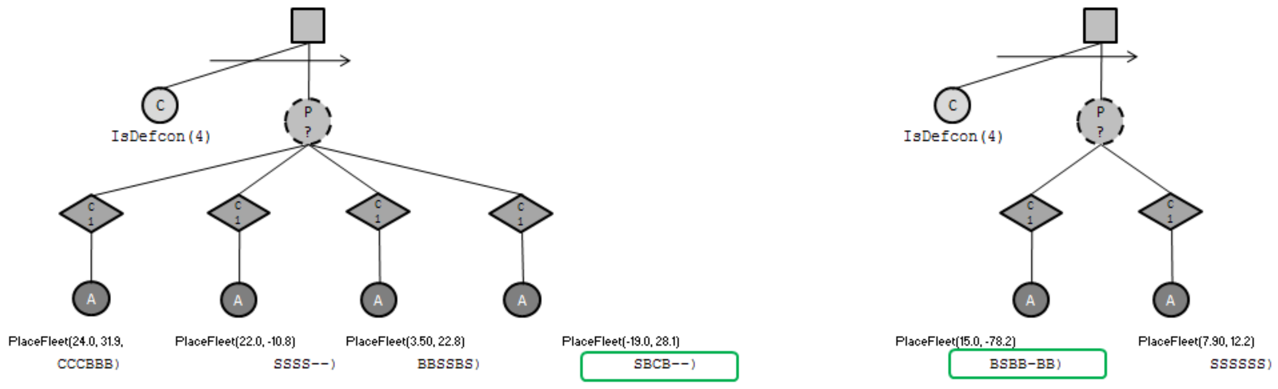

IsDefcon(4)
PlaceFleet(24.0, 31.9, CCCBBB)
PlaceFleet(22.0, -10.8) SSSS--)
PlaceFleet(3.50, 22.8) BBSSBS)
PlaceFleet(-19.0, 28.1) SBCB--)
IsDefcon(4)
PlaceFleet(15.0, -78.2) BSBB-BB)
PlaceFleet(7.90, 12.2) SSSSSS)

Figure 4.8: Resulting offspring behaviour trees after fleet composition

4.3 Distributed Processing and Parallelisation

- We have to repeat the process for all 4 areas of improvements
- 100 generations of 100 games may not be sufficient to reach an optimum performance
- With 6 possible starting locations and 5 corresponding enemy positions, we have to repeat it 30 times to allow the bot to play in every position optimally

In this section, we will outline the method used to distribute the computation over more than just 1 computer, in order to reduce the amount of computation time required to evolve the system.

4.3.1 Overview

We identify two main components of the system,

- The **Server**, responsible for combining the results from each client, performing the evolution of the behaviour trees and dividing the evolved trees for each subsequent generation for the clients. The server resides on a shared folder accessible by all the clients.
- The **Client**, responsible with running the individual games of DEFCON using the behaviour trees received from the server. After which, the results from each game are

collected and sent to the server for the evolution process. Each client resides on a single workstation, each with access to the shared directory of which the server resides.

4.3.2 Client

Each client computer consists of a working copy of DEFCON, a set of behaviour trees for which the AI-bot is to run. The results at the end of the game are recorded by the client before being sent to the server.The client then waits and continues to poll the server's shared space for a unique file containing the next set of behaviour trees to run. Upon detecting that the file exists, it moves it from the shared folder into its own computer and begins the runs of DEFCON. This is illustrated by the pseudo-code in Listing 4.3

```
Initialise NumberOfRuns

:Running
for( 1 to NumberOfRuns )
  Run DEFCON
  Save to Results Log

Tag Results Log with Client number N
Send Results Log to Server's Shared Folder

:Wait
while( next set of trees not yet available )
  wait

Copy next set of trees from Server's Shared Folder

GOTO Running
```

Listing 4.3: Pseudocode for Client N

4.3.3 Server

The Server resides on a shared space accessible by all the clients. The reason for this is to allow the merging of the results from each of the clients into a single file, following which, the performance of each tree is measured and the next generation of trees is evolved. The server keeps track of the number of clients it has, and divides the trees evenly between the clients and tags them with each Client's unique number. The divided files are left on the shared folder and await retrieval from each of the client's. This method highlights the fact that the server does not keep track of the location of each individual client. The reason for this is to make the server's role as independent from the clients' as possible, allowing us to replace client workstations as we needed, especially in the cases when some of them had faults.The pseudo-code in Listing 4.4 below outlines the Server process in more detail.

```
set current_generation
set max_generations
set number_of_clients

:AWAIT_CLIENT_FILES
while ( current_generation < max_generation )
  for ( i:{1..number_of_clients} )
      if ( part_i does not exist )
           wait

  merge the parts into combined results

  evolve the next generation of trees

  split the new generation into number_of_client parts

  current_generation++

  GOTO AWAIT_CLIENT_FILES

:END
```

Listing 4.4: Pseudocode for Server

4.3.4 Redundancy

In our proposed distributed system, the workload of running the games and collecting the results have been placed on the individual client computers, while the combining, evolving and dividing of the results are left to the server. In such an approach, there is a need for some form of redundancy or error handling in the event of a workstation crash or a single run of DEFCON failing.

Workstation Crashes or Reboots

As mentioned earlier, we have separated the roles of the Clients and the Server and as such, in the event of a workstation crashing, we merely need to run the Client process on a new workstation while the failing workstation reboots. It is easy to identify which workstation has failed by looking at the part which the server is still waiting for.

Individual DEFCON runs

On certain occasions, a game of DEFCON might experience errors, resulting in the failure to give the output of results for a particular behavior tree. This would cause the population size of a particular generation to decrease, requiring us to take some form of action to enable us to perform the evolution of the results for its subsequent generation.

Two approaches can be used in order to resolve this situation,namely:

1. Take an average of available results to estimate results for any failed run
2. Fill in results of failed runs with a set of back-up results, produced by:
    - Generating a second set of the current generation's population
    - Run a small number of games
    - Use results from this back up set to replace failed runs

The second approach was implemented in our approach, used in conjunction with the existing client and server system. By implementing such error control and backup measures, we were able to automate as much of the evolution process as possible, allowing us to leave the workstations to work autonomously, without the need of a person to manually perform the tasks required for each run.

4.3.5 Scalability

The system consists of scripts that are theoretically scalable to up to any number required, limited only by the population size in each generation. For example, with a population size of 100, this system is scalable up to 102 workstations, one for each run of DEFCON, the Server and the Backup machine.As we will cover in the section on the experimental setup in Chapter 5, the system was tested on a total of 20 Client Computers, 1 Server and 1 Backup workstation.

4.4 Summary

In this chapter, we have used the design choices from Chapter 3 to implement the hand-crated behaviour trees that were required for our AI-bot. We presented the complete set of behaviour trees and leaf task nodes that were developed and outlined how we used XML files to store these behaviour trees. We outlined the implementation of the evolution system used to evolve the behaviour trees, showing how the game state information was retrieved from the API via two method calls. An example of how a behaviour tree was evolved was presented, showing how the a resultant tree was automatically generated after the application of genetic operators and selection. Finally, we outlined the distributed system which was used in order to perform the multiple runs required for the evolution process. In the next chapter, we cover the experimental design used to evolve various aspects of our AI-bot.

Chapter 5

Experimental Design

In this chapter, we cover the experimental set up which we employed to perform the evolution of the bot. We first provide information about resources that were used, including computer specifications and in-game options that were set for the purpose of our experiments. These are all covered in 5.1. We then cover the main experiments that were performed, with the purpose of improving our AI bot, in Section 5.2. We detail the areas of which the AI bot was evolved, identifying appropriate fitness functions that were used in order to measure the AI-bot's performance as it evolved. The results and graphs that were obtained from the experiments outlined in this chapter are provided in Chapter 6. An analysis into the significance of the results of the experiments are then covered in Chapter 7.

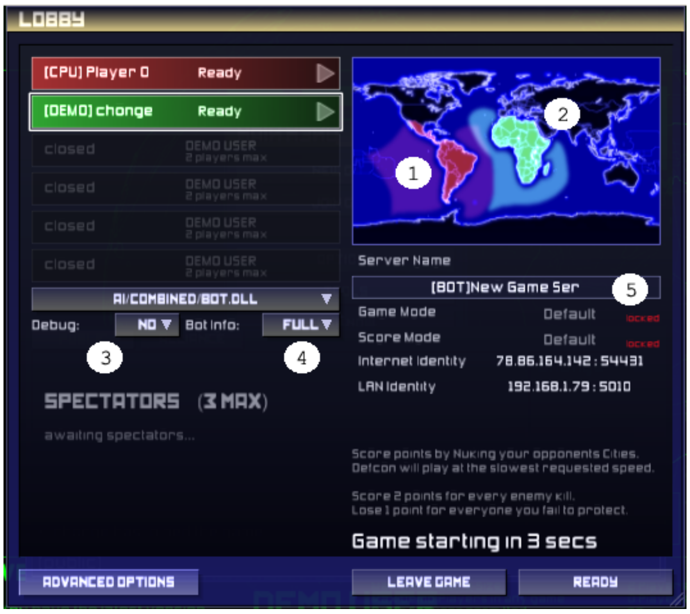
5.1 Setup

5.1.1 Game Options

We made use of Introversion Software's DEFCON (Demo v1.43) together with API (v1.56 beta). The API was continuously revised, beginning at version v1.51, with input from this project. Figure 5.1 shows the main lobby screen highlighting the main game options used for the experiments.

We chose the following game options:

1. The opponent's starting position was fixed as South America
2. Our bot's starting position was fixed as Africa
3. Debug mode was turned off
4. Bot information was turned to Full
5. Default Game and Scoring mode was used
6. API command line parameters (not shown in figure) which were used included
    - `fastserveradvance`: Used to reduce the amount of time between ticks
    - `norender`: Turns off rendering of game objects



[CPU] Player 0 Ready
[DEMO] chonge Ready
closed DEMO USER 2 players max
closed DEMO USER 2 players max
closed DEMO USER 2 players max
closed DEMO USER 2 players max
AI/COMBINED/BOT.DLL
Debug: NO Bot Info: FULL

SPECTATORS (3 MAX)
awaiting spectators...

Server Name
[BOT]New Game Ser
Game Mode Default locked
Score Mode Default locked
Internet Identity 78.86.164.142 : 54431
LAN Identity 192.168.1.79 : 5010

Score points by Nuking your opponents Cities.
Defcon will play at the slowest requested speed.

Score 2 points for every enemy kill.
Lose 1 point for everyone you fail to protect.

Game starting in 3 secs

Figure 5.1: Experimental Setup

These options may be set automatically by inserting them as command line parameters when running the DEFCON executable that is provided with the API. A full list of commands are listed in page 96 of the Appendix.

5.1.2 Distribution

We distributed the running of DEFCON games over 20 computers, connected together via the department's Local Area Network. Each of these computers ran our client scripts used to regulate the number of games that ran. In this case, with our proposed population size of a hundred individuals, each computer ran 5 games.

We used a separate computer, with similar specifications, to act as the server. This computer was responsible for the storage and merging of the results from the client computers. A final computer was used to run the back-up games to cope with games that failed to run or crashes.

5.1.3 Computer Specifications

The specifications of each computer that our experiment was run on is as follows,

- **Operating System**: Microsoft Windows Vista Business (SP1)
- **Processor**: Intel Pentium 4 3.60 GHz
- **Memory**: 1.00 GB
- **Graphics Card**: nVidia GeForce 7600 GS

5.2 Experiments

5.2.1 Introduction

The Problem

The constructed Behavior Trees from the previous section resulted in an AI player that is capable of playing the entire game of DEFCON, making moves and decisions such as placing units, moving units and selecting enemy targets - although its choices are based on random choices and calculations rather than informed decision.

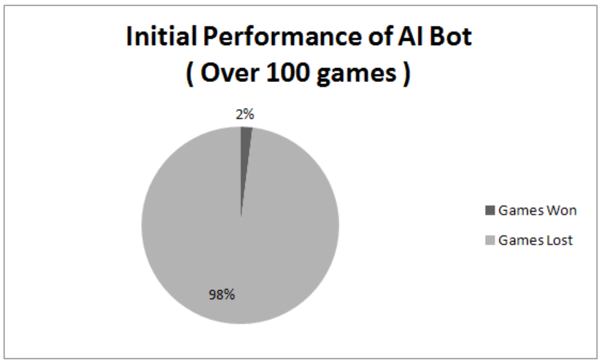


Figure 5.2: Performance of Random bot

From the graph in Figure 5.2, the prototype bot performs relatively poorly. There are several reasons for this,

- The placement of land structures are not maximising the player;s ability to do well. Specifically:
  - Silos are being destroyed easily by enemy units, preventing them from providing defence against air attacks and also preventing the AI from launching LRBMs for its offensive attack.
  - Radar stations placed do not maximise the area of coverage, making the AI unable to detect enemy units in time.
  - The AI is not placing the maximum number of units it is able to, putting itself at a disadvantage by not maximising its resources
- Fleet units are moving to random positions on the map so that:
  - Battleships are unable to attack enemy sea units sufficiently
  - Carriers are not close enough to the enemy territory to send fighters to scout and uncover more enemy units
  - Submarines are too far away from enemy buildings to launch an attack
- The timings of the attacks are randomly assigned, thus reducing the chance of a coordinated attack to be able to inflict maximum damage to the enemy.

Identified Areas to Improve

With the identification of shortfalls in the AI-bot above, we we identified the 4 key areas which we wanted to improve upon, namely:

1. Placement of silos to maximise its defence against enemy air units
2. Placement of radar Stations to maximise the number of enemy units spotted throughout the course of the game
3. The placement and subsequent movement of the player's fleet units, with the intention of improving:
   - The number of enemy sea units destroyed
   - The number of enemy buildings destroyed
   - The number of cities attacked, and populations destroyed
4. The timings of the attacks for the 4 main offensives, namely:
   - Submarine attacks using MRBMs
   - Bomber attacks from Carriers using SRBMs
   - Bomberattacks from Air Bases using SRBMs
   - Silo attacks using LRBMs

As such, we performed four main evolution experiments with the following common properties,

- The population size of games was 100
- The selection ratio was 0.5
- The recombination rate was 0.5
- The mutation rate was 0.05

In each experiment, the number of generations varied between 80 and 250. The numbers outlined in this section were values deemed practical for the experiments which were required, given the resource and time constraints of this project. Ultimately, the obtained results were sufficient to perform an analysis of the performance of the evolved bot, in addition to fulfilling the objectives set out at the beginning of obtaining a competitive bot.

5.2.2 Silo Placement

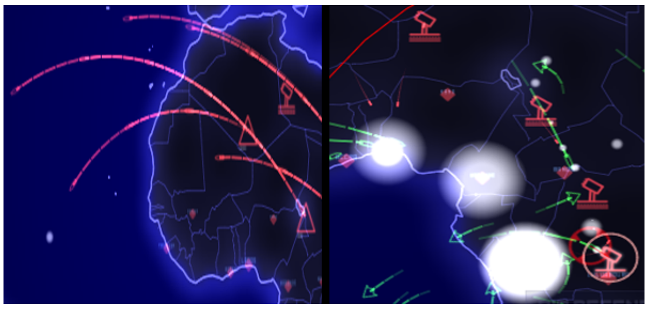


Figure 5.3: Silo Placement, Defence and Attack

Silos are ground installations which play a significant role in both attack and defence. When in *Air-Defence Mode*, silos actively shoot down enemy air units and try to limit the damage on the player's city populations and units. In *Launch Mode*, silos provide the heaviest firepower for attacking enemy units and city populations. Its effectiveness at both defence and attack is dictated not only by each individual silo's absolute position, but also their positions relative to one another. With such a significant role, we are interested in obtaining placement positions that would maximise their collective effectiveness. We have chosen to focus on the defensive aspect of the Silos.

Fitness Function: Defence Potential

In order to measure how well a certain position system worked, we define the **defence potential** of a position by the total number of air units that were successfully destroyed for a given game. These air units consist of

- Enemy Nukes
- Enemy Fighters
- Enemy Bombers

Thus, a behavior tree driven bot placing Silos in a certain position is deemed fitter than another if its defence potential is higher - meaning that a greater number of air units were destroyed.

Evolution Area: Placement Positions

For each run, we keep track of the positions that each of the silos were placed in. The genetic operators are used to recombine the positions, mixing the positions which one Behavior Tree had with that of another in order to produce the next generation's set of placement positions. For the initial run, the placement position for each Silo were assigned randomly.

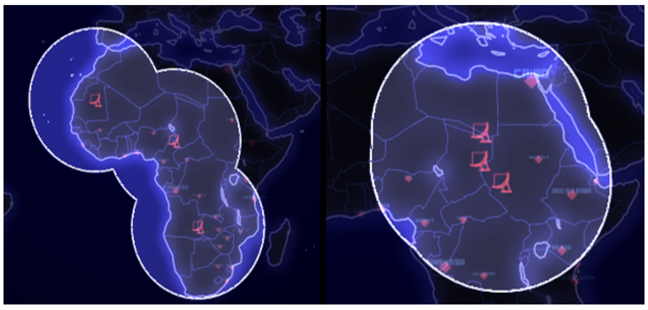


Figure 5.4: Radar Placement & Coverage

5.2.3 Radar Placement

While radars do not have a direct influence on a player's attack or defensive abilities, they provide an indirect means of support by uncovering enemy units. This allows the AI player to then react appropriately to the situation, for example, by sending fighters to enemy sea units or to be able to shoot down nukes earlier. In Figure 5.4 above, we exhibit that coverage of the Radars are determined by their positions. An investigation into a good positioning system for the Radars is thus the focus of this set of runs.

Fitness Function: Uncovered Enemies

Uncovered Enemies refer to the enemy sea units, making a total observable number of 36. A slight difference in the tracking of units is that we only observe the total number of units seen before the start of DEFCON1. The reason to allow ample time for retaliation by our AI-bot to react and behave after detecting the enemy units. For example, when enemy Submarine units attack, they surface and become visible, and are added to the total. However, we deemed this particular type of detection not to be particularly advantageous to our AI-bot. Thus, the fitness of an individual was taken to be the number of uncovered enemies uncovered before the start of DEFCON 1.

Evolution Area: Placement Positions

Similar to the silo placements above, we prepare individuals for each subsequent generation by performing genetic operations on the placement positions of the Radars dictated by the behavior trees, with the intention of evolving towards an optimal position placement of the radar stations.

5.2.4 Fleet Unit Placement & Movement

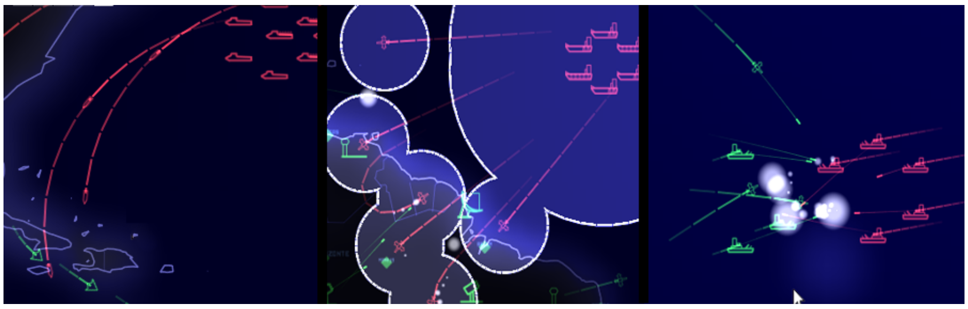


Figure 5.5: Fleets in DEFCON performing various roles

The fleets in DEFCON differ from ground installations and are more complex to evolve due to their ability to move around to different sea locations on the map. Thus, we are concerned not just with the initial positioning of the fleets but also their subsequent movements which increased the AI's effectiveness in their abilities to attack, defend or scout enemy territories. Figure 5.5 above shows from left to right, subs attacking using MRBMS, carriers scouting enemy territories with Fighters and battleships attacking other enemy units.

We model this complexity by making use of two Behavior Trees - one to designate the starting positions of the fleets and the other to designate its target position which it intends to reach. We do not consider explicitly any subsequent movements after a fleet has reached its target position. This may initially seem to be restrictive of fleet movement behavior, but as covered in Section 3.2, reactive planning is used to dictate subsequent movements from an initial target position. Also, as we see in the evaluation presented later, this approach (Chapter 6, Section 6.3) proved to be sufficiently effective. A final area of consideration involves the composition of each fleet that was placed. As the fleets in DEFCON allow any combination of Submarines, Battleships or Carriers, an investigation into the effects of having different fleet compositions to the AI's performance was of particular interest.

Fitness Function: Weighted Average

The fitness of each run consisted of evaluating how well the bot performed in 3 areas, namely,

- The number of enemy buildings uncovered
- The number of enemy buildings destroyed
- The number of enemy sea units destroyed

In order to give an overall aggregate value to how well a bot performed based on its results in each of these three areas, we perform linear normalisation to the result using the following equation:

$$score_i = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Where i here refers to the which area of interest we are evaluating. The min and max values are also specific to the scenario. For example, supposing that in a game, a total of 2 silos, 5 radars and 1 airbase is destroyed:s, 5 radars and 1 airbase is destroyed:

$$score_{destroyed\_buildings} = \frac{(2+5+1)-0}{17-0} = \frac{8}{17} = 0.471$$

Given that the maximum number of buildings that can be destroyed would be 17 since there are a total of 6 placed silos, 4 airbases and 7 radar stations. Naturally, the minimum number destroyed would be if none of them are destroyed. Finally, in order to obtain an aggregated fitness across the three areas, we calculate the fitness by the following equation,

$$fitness = 100 \times \sum weight_i \times score_i$$

We used an equal weighting for each of the scores and made them add up to 1. The resulting fitness value would thus be between 0 and 100.

Evolution Areas: Locations, Composition & Timings

Similar to both silo and radar placements, we vary the placement and target positions of each fleet across generations by the use of genetic operators such as recombination and mutation. Additionally, the genetic operators were applied to the composition of fleets.

5.2.5 Timing & Synchronisation

Finally, we aimed to combine the trees used for the previous 3 sections resulting in a fully functional bot, for comparison with the original Random bot which we introduced earlier. Upon combining the trees, one final aspect involved the study of how the timings of the attacks affect the bots performance in the game. The four key timings involved are

1. Submarine MRBM launch timing
2. Carrier bomber launch timing
3. AirBase bomber launch timing
4. Silo LRBM launch timing

Fitness: Overall Game Performance

By combining the best performing Behavior Trees that perform optimally in their respective areas, an assessment of how well the bot performs in trying to win the game was appropriate. We used the difference between the final end-game scores as an indicator of how well the AI-bot performed. A larger difference would mean that a win was convincing, whereas a smaller difference would mean that the win was close. The fitness function makes use of the similar linear normalisation function:

$$fitness = \frac{difference - k_{min}}{k_{max} - k_{min}}$$
$$difference = score_{AI-bot} - score_{enemy}$$

$$k_{min} = -250$$

s

$$k_{max} = +250$$

The values for *k_min*and *k_max* were determined empirically, and denote the maximum difference in the end-game scores observed when losing and winning respectively. A worked example would be to consider a game where the AI player has obtained a score of 67, and the opponent has a score of 22. Thus, the resulting *difference* would be 45, whilst the denominator would work out to be $250 - (-250) = 500$. Thus, the fitness is worked out as follows,

$$fitness = \frac{45 + 250}{500} = 59$$

Evolution Area: Timings

The genetic operators were applied to the values of the 4 key timings as mentioned above.

5.3 Summary

In this chapter, we have outlined the various experiments that were performed with the purpose of evolving our AI-bot. The different experiments attempted to focus on specific areas of game play for which we wanted our AI-bot to perform well in, and as such, employed the use of different fitness functions to assess its performance in the different areas. The results from these experiments are presented in the next chapter, where we make use of graphs and charts to give a summary of the results over the generations of evolution. In Chapter 7, we provide an analysis into these results in order to provide insight and explain the emergence of certain behaviours by our AI-bot.

Chapter 6

Results

In the previous chapter, we detailed the set of experiments which were conducted in order to evolve the behavior trees of the original, hand-crafted AI-bot (Chapter 4, Section 3.2). In this chapter, we shall present the results from these experiments. Section 6.1 presents the results in optimising silo placement positions, section 6.2 then presents the results in optimising radar placement positions. The performance of the AI-bot in evolving its coordination of fleet movements and compositions are presented in section 6.3. We present the results from attempting to find the appropriate timings of attacks for the AI-bot's units in Section 6.4. Finally, section 6.5 presents the results from combining the best performing behavior trees from each of the individual experiments and producing the final version of our AI-bot. This chapter provides the raw data obtained from the experiments of Chapter 5 using graphs and charts. The analysis of the results presented in this chapter are covered in Chapter 7.

6.1 Silo Placements

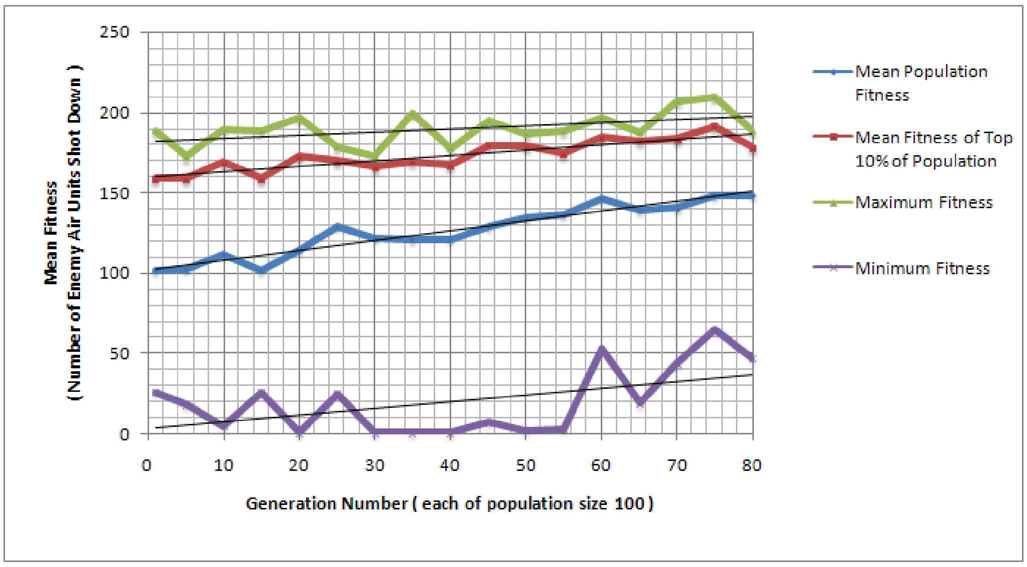


Figure 6.1: Graph of "Enemy Air Units Shot Down" across Generations

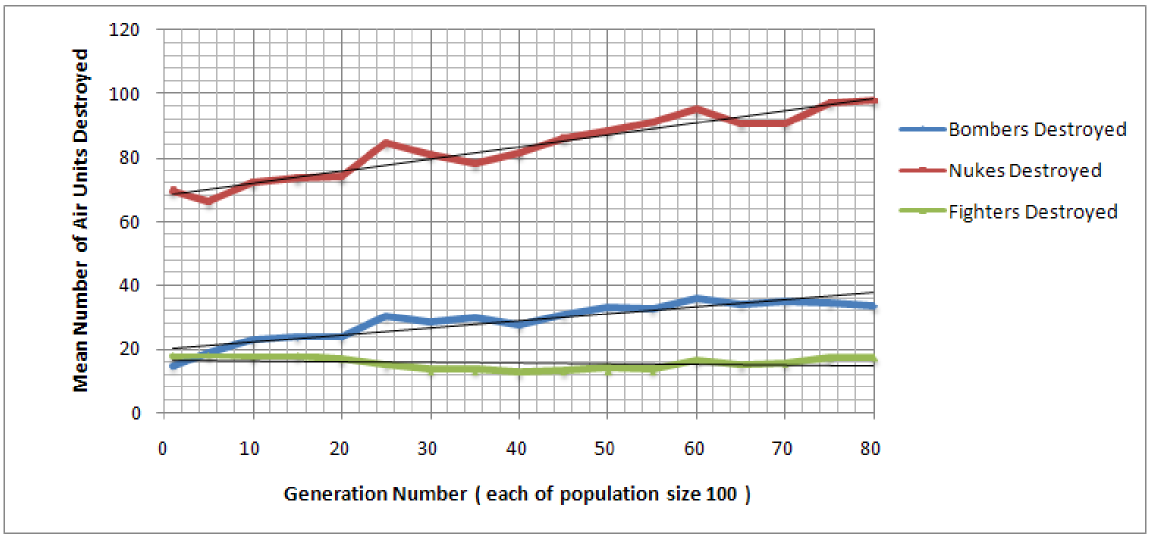


Figure 6.2: Graph of “Enemy Air Units Shot Down” across Generations

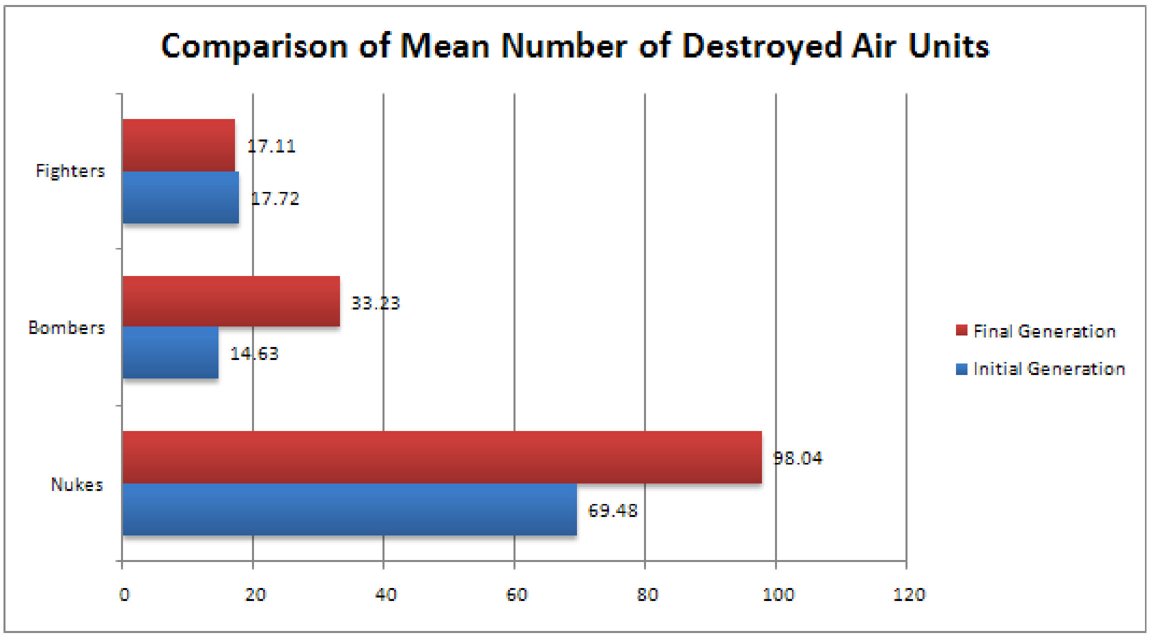


Figure 6.3: Comparison of Distribution of “Enemy Air Units Shot Down” at Initial and Final Populations

In the experiments to discover optimal placement positions for the AI-bot's silos, we used the number of enemy units shot down throughout the course of the game as a fitness measure. We used a population size of 100, and set both the selection ratio and recombination rate to be 50%, along with a mutation rate of 5%. We ran the experiment over 80 generations. Figure 6.1 shows how the mean fitness of the population varied over the generations. To have a better idea of the performance of the AI-bot, the mean counts of each unit type destroyed in each generation is presented in Figure 6.2. Figure 6.3 illustrates the difference in performances of the AI-bot between the initial and final stages of evolution. Table 6.1 depicts the final, best performing placement positions from the behaviour trees.

(13,30)	(43,2)	(37,5)	(33,25)	(34,13)	(22,17)

Table 6.1: Final Silo Placement Positions – (Longitude, Latitude) pairs

6.2 Radar Placements

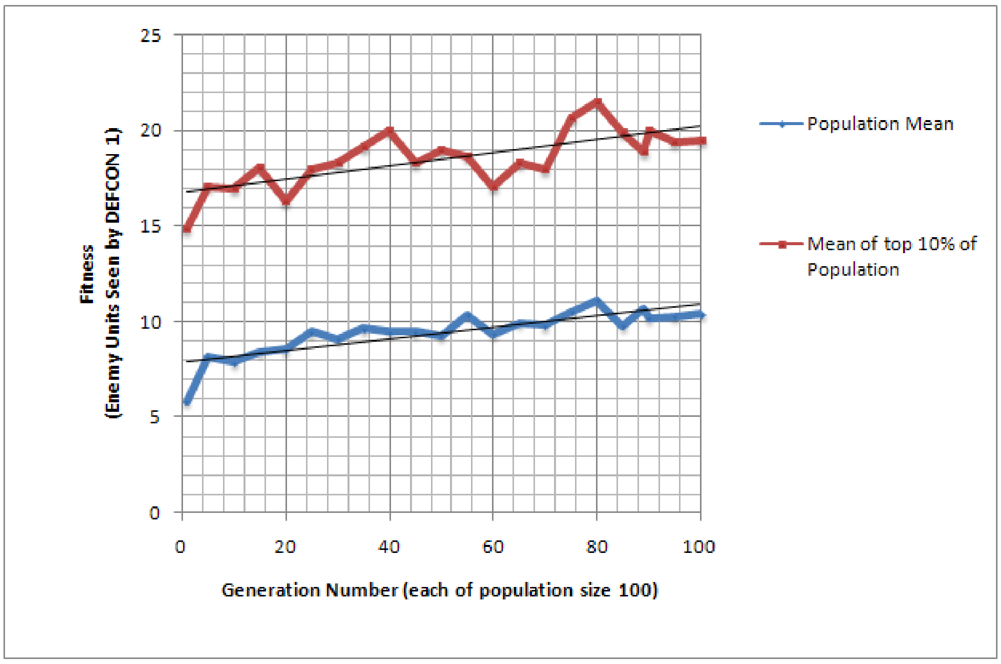


Figure 6.4: Graph of "Enemy Units Seen by DEFCON 1" across Generations

In the experiments to discover optimal placement positions for the AI-bot's radars, we used the number of enemy sea units detected before the phase of DEFCON 1 as a fitness measure. We used a population size of 100 and again set the selection ratio and recombination rate at 50%. The mutation rate was set to 5%. Figure 6.4 shows how the mean fitness of the population varied over the generations. In order to make a comparison between the performance of the AI-bot between the initial and final phases of evolution, we categorised the AI-bot's performance in the following way:

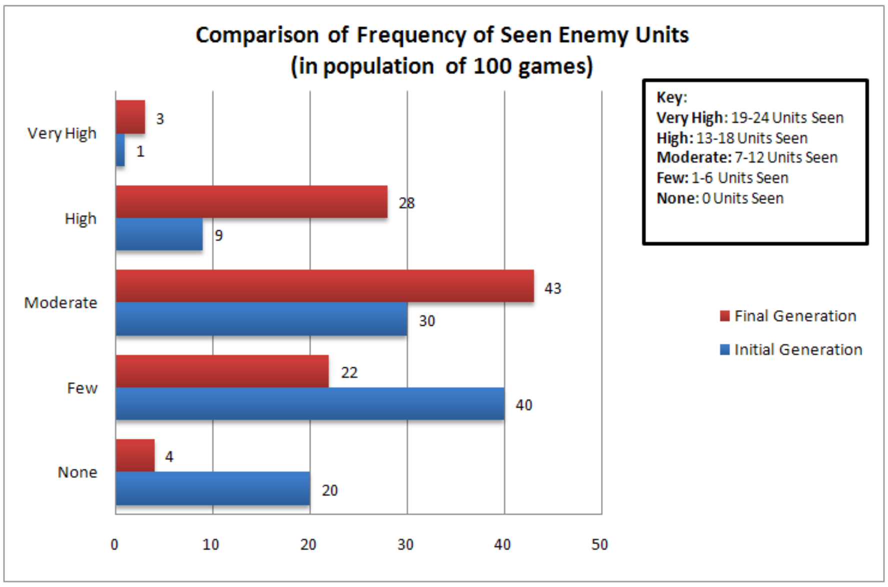


Figure 6.5: Chart showing distribution of Enemy Units Seen at the Initial and Final Populations

- **Very High**: When the AI-bot was able to detect between 19-24 enemy units.
- **High**: When the AI-bot was able to detect between 13-18 enemy units.
- **Moderate**: When the AI-bot was able to detect between 7-12 enemy units.
- **Few**: When the AI-bot was able to detect between 1-6 enemy units.
- **None**: When the AI-bot was unable to detect any units.

The reason for using such a categorisation allows us to determine whether the AI-bot showed an improvement by having a higher percentage of games in which it managed to detect a greater number of enemy units. By possibly having more games in which the AI-bot detected **Very High** and **High** numbers of enemy units and correspondingly less games in which the AI-bot detected **Few** or **None** of the enemy units, we would infer that the AI-bot has improved. Theoretically, the maximum number of observable units is 36. However, based on our experiments, we observed that before the course of DEFCON 1 the default opponent never surfaced its submarine units. And since before DEFCON 1, no nuke attacks are permitted, enemy submarines are thus unable to be detected from launching nuke attacks either. Thus, we have used the value 24 as the theoretical maximum for this experiment, and derived the different levels of performance empirically based on that. Finally, Table 6.2 contains the final, best performing placement positions obtained from the behaviour trees.

(14,-21)	(24,-31)	(-5,5)	(14,-7)	(-15,25)	(-4,23)	(-10,9)

Table 6.2: Final Radar Placement Positions – (Longitude, Latitude) pairs

6.3 Fleet Placement & Movement

In the experiments to discover optimal placement and movement positions, along with ideal compositions for DEFCON fleets, we used the weighted average of the number of enemy structures uncovered, enemy structures destroyed and enemy sea units destroyed as a fitness measure. In this experiment, an equal weighting was assigned to each of the three individual scores. We used a selection ratio of 50% and a 50% recombination rate. The mutation rate was set at 5%. The experiment was evolved over 200 generations using the distributed system from Chapter 4 (Section 4.3).

Figure 6.6 shows how the mean fitness of the population varied over the generations. We then make comparisons between the initial and final populations, presenting the performance of the AI-bot in each of the three individual areas in Figure 6.7. Finally, Figure 6.8 shows the best performing set of fleet positions and compositions obtained from the behaviour trees. A total of 6 fleets were placed, and using the top-left box as an example, it means that one of the fleets was composed of 3 submarines, 3 battleships and 0 carriers. It was placed at the location (-20.00, 40.60) and its target ocation was (-32.00,22.88). The rest of the fleets can be inferred from the diagram in a similar way. Figure 6.13 shows the results from the best performing behaviour trees.

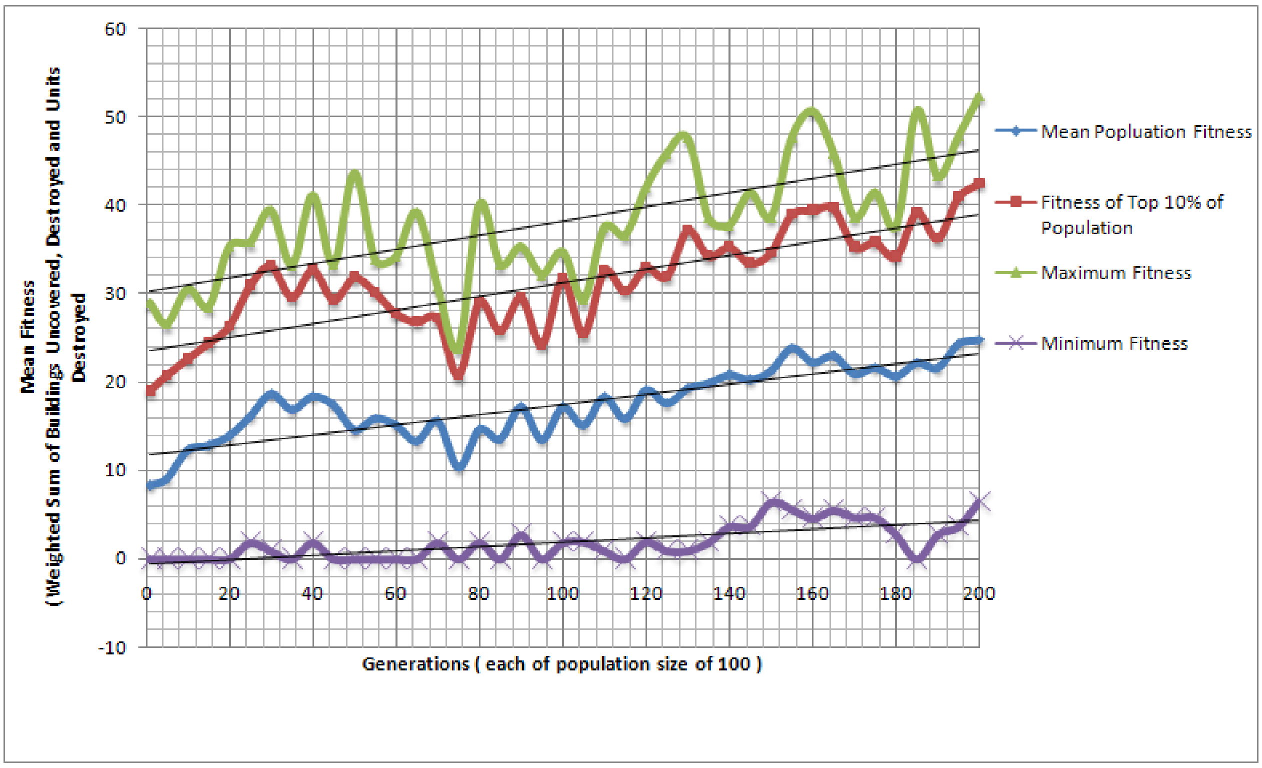
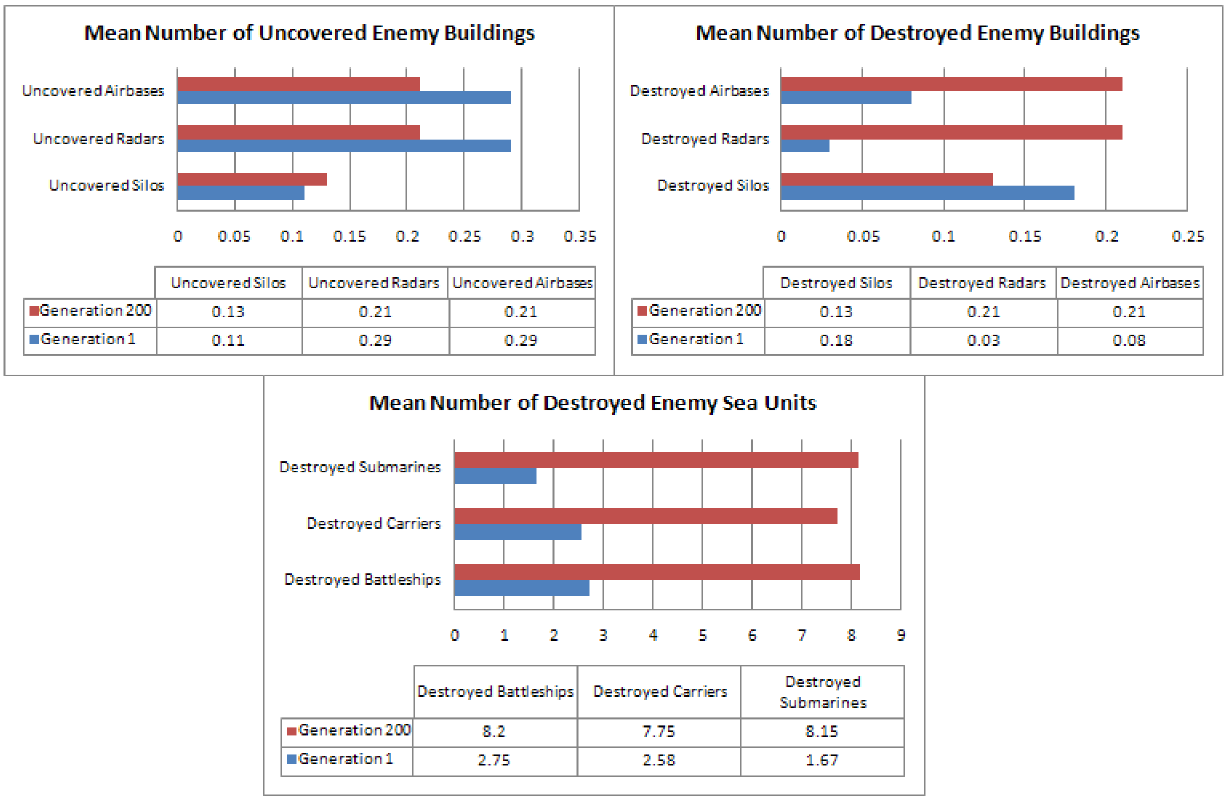


Figure 6.6: Graph of Fitness across Generations

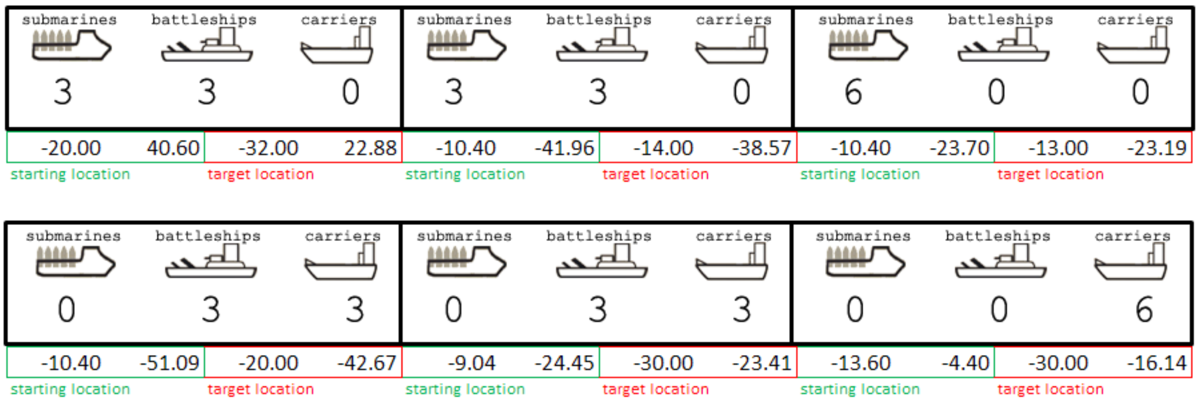


	Uncovered Silos	Uncovered Radars	Uncovered Airbases
🟥Generation 200	0.13	0.21	0.21
🟦Generation 1	0.11	0.29	0.29

	Destroyed Silos	Destroyed Radars	Destroyed Airbases
🟥Generation 200	0.13	0.21	0.21
🟦Generation 1	0.18	0.03	0.08

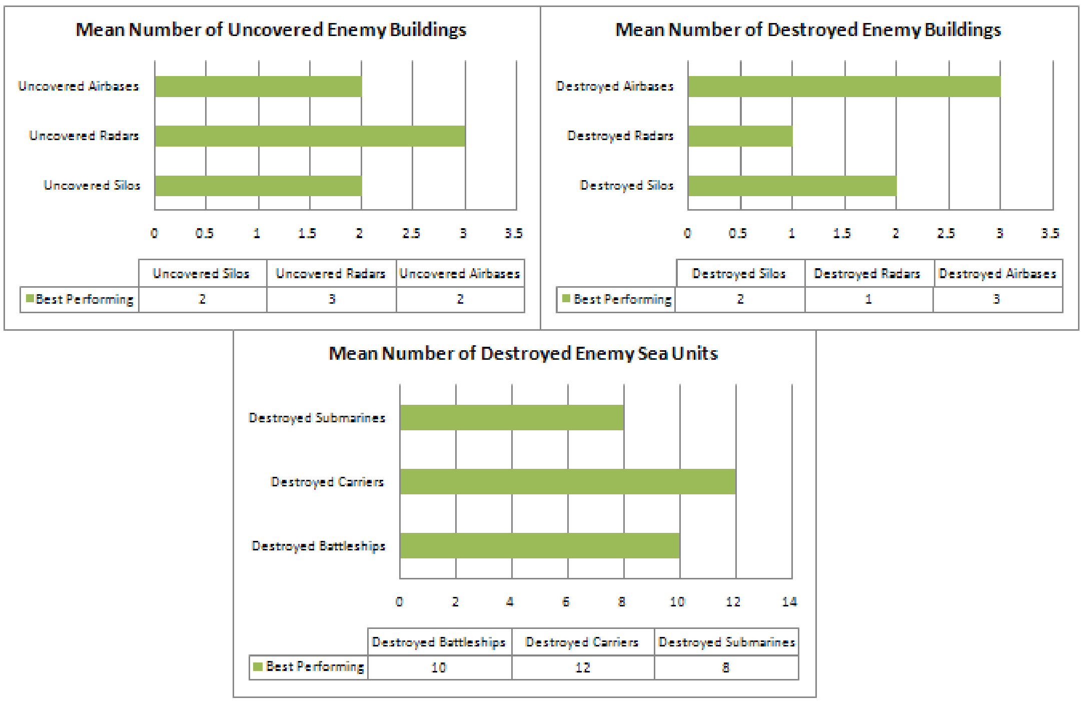
	Destroyed Battleships	Destroyed Carriers	Destroyed Submarines
🟥Generation 200	8.2	7.75	8.15
🟦Generation 1	2.75	2.58	1.67

Figure 6.7: Graph of Fitness across Generations



submarines	battleships	carriers	starting location		target location	
3	3	0	-20.00	40.60	-32.00	22.88
3	3	0	-10.40	-41.96	-14.00	-38.57
6	0	0	-10.40	-23.70	-13.00	-23.19
0	3	3	-10.40	-51.09	-20.00	-42.67
0	3	3	-9.04	-24.45	-30.00	-23.41
0	0	6	-13.60	-4.40	-30.00	-16.14

Figure 6.8: Optimal Fleet Compositions and Movement Positions



	Uncovered Silos	Uncovered Radars	Uncovered Airbases
🟩 Best Performing	2	3	2

	Destroyed Silos	Destroyed Radars	Destroyed Airbases
🟩 Best Performing	2	1	3

	Destroyed Battleships	Destroyed Carriers	Destroyed Submarines
🟩 Best Performing	10	12	8

Figure 6.9: Results from the Best Performing Behaviour Trees for Fleet Movements

6.4 Timing & Synchronisation of Attacks

In the experiments to discover optimal timings for performing attacks using submarines, bombers and nukes, we used the normalised value of the difference between the AI-bot's and the opponent's score as a fitness measure. We used a selection ratio of 50% together with a 50% recombination rate. The mutation rate was set at 5%. The experiment was evolved over 100 generations using the distributed system. We made use of the optimal silo and radar placement positions from Section 6.1 and Section 6.2 respectively, together with the fleet placement, movement and compositions from Section 6.3. We fixed the airbase placement positions by randomly generating the following positions, shown in Table 6.3. We mention the evolution of airbase placements as a potential area of improvement in Section 8.2 of Chapter 8.

(5.5,22.3)	(23.0,8.8)	(25.3,-0.9)	(23.4,-19.8)

Table 6.3: Fixed Airbase Placement Positions – (Longitude, Latitude) pairs

Figure 6.10 shows the variation of means of both the AI-bot's score and the opponent's score over the generations, and Figure 6.11 presents the variation of the mean difference between the scores as the AI-bot evolved. The mean fitness was measured across the generations, and this is shown in Figure 6.12, and the mean number of games won by our AI-bot over the generations is shown in Figure 6.13. Finally, the final timings that were obtained from the behaviour trees are presented in table 6.4

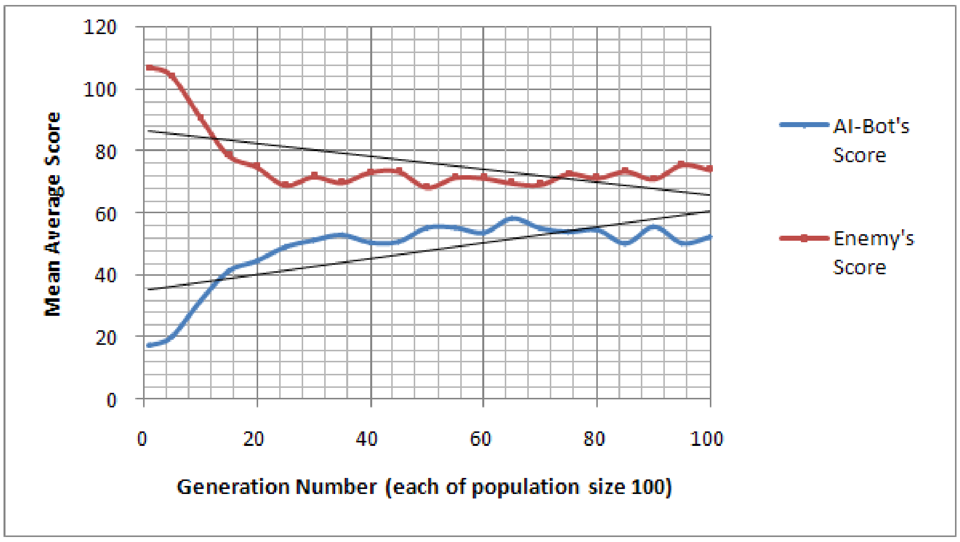


Figure 6.10: Graph showing End-game Scores across Generations

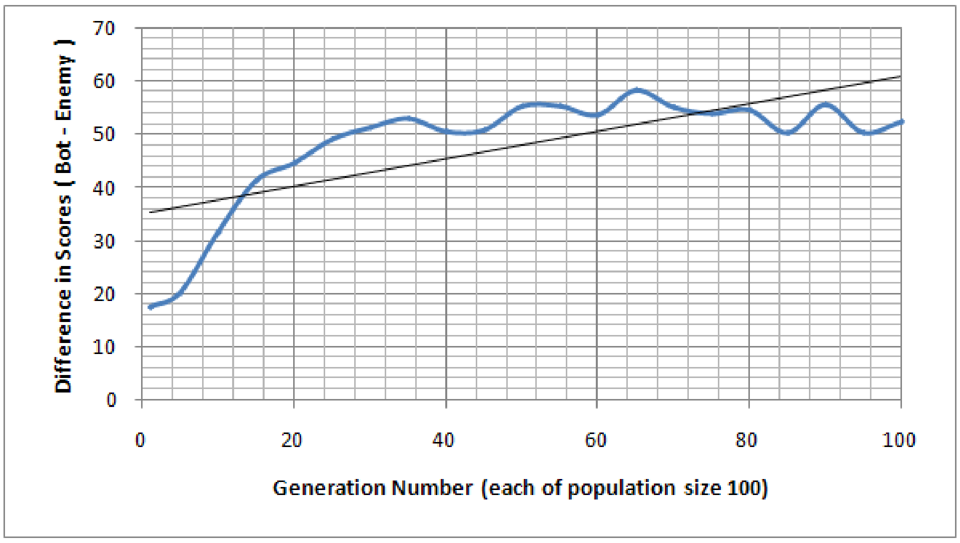


Figure 6.11: Graph showing Score Difference across Generations

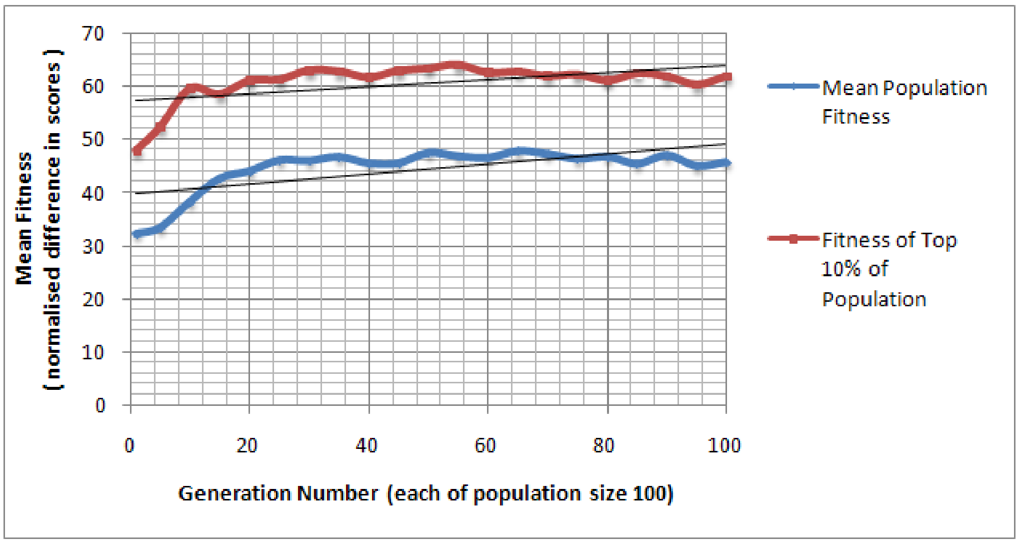


Figure 6.12: Graph showing Mean Fitness across Generations

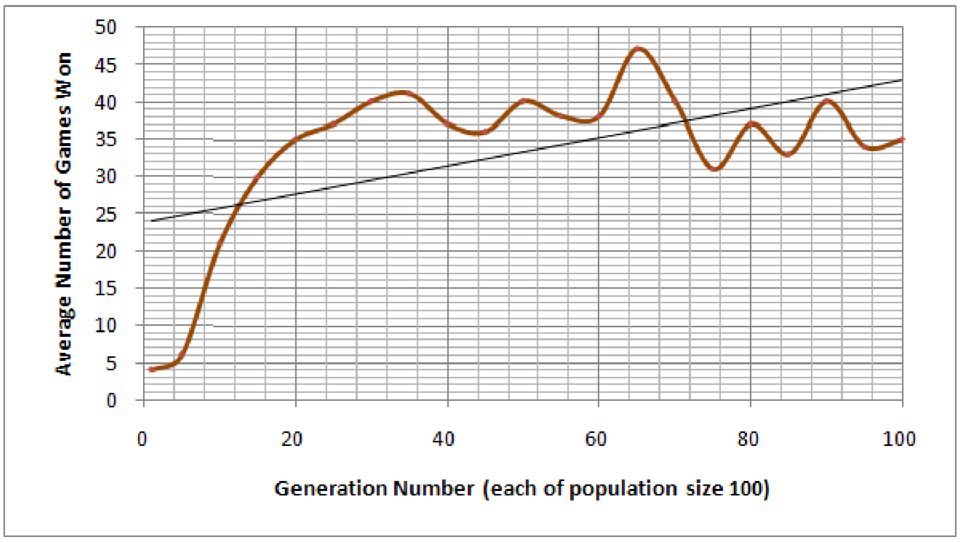


Figure 6.13: Graph showing Average Game Wins across Generations

Silo Nuke Time	Carrier Nuke Time	Airbase Nuke Time	Silo Nuke Time
2767s	3312s	4460s	4331s

Table 6.4: Mean Results of Final Combined AI-bot in across 100 Games

6.5 Overall A.I. Behavior

Mean AI-bot Score	Mean Enemy Score	Mean Difference	Mean Fitness
66.14	61.91	+4.23	50.846

Table 6.5: Mean Results of Final Combined AI-bot across 200 Games

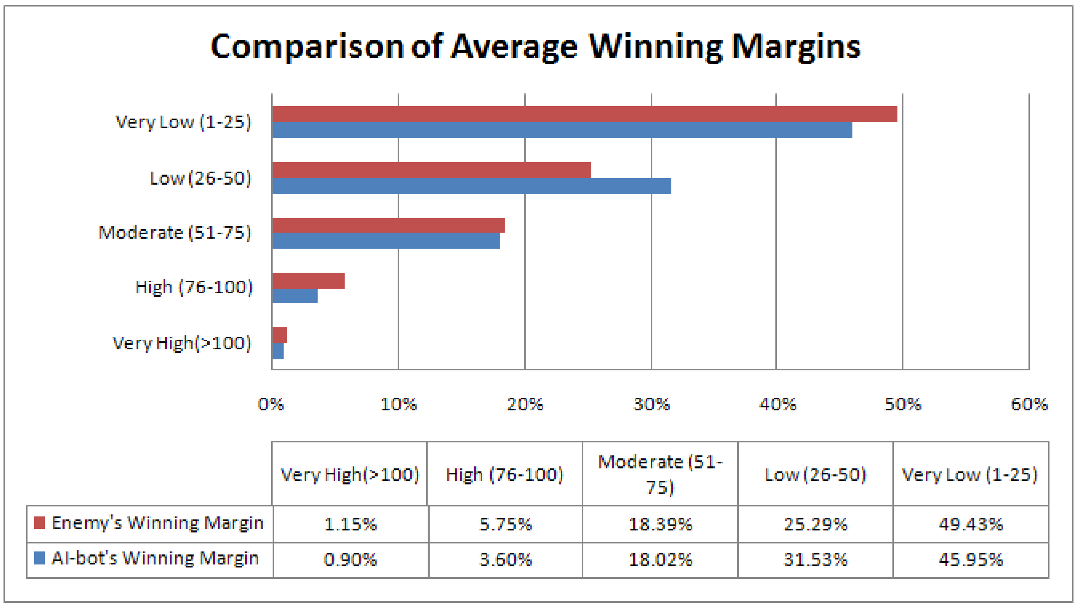
From the evolved behaviour trees from the previous 4 sections, we are now in place to combine the best performing behaviour trees from each experiment to produce a final AI-bot. We ran a total of 200 games using this final, evolved AI-bot in order to assess its performance in the overall game of DEFCON. The difference between the scores obtained by both the AI-bot and the opponent was used as an indicator of how well the AI-bot had performed. This difference is termed as the **winning margin**, and we categorise it into 5 different groups:

- **Very Low**: This is the category when the winning margin is between 1-25.
- **Low**: This is the category when the winning margin is between 26-50.
- **Moderate**: This is the category when the winning margin is between 51-75.
- **High**: This is the category when the winning margin is between 76-100.
- **Very High**: This is the category when the winning margin is greater than 100.

In this set of experiments, we make note of the frequency in which either the AI-bot or the default opponent won and classified each win into one of the above categories. These results are shown in Figure 6.14. The overall performance of the AI-bot in terms of the percentage of games won is shown in Figure 6.15. Finally, Table 6.5 gives the mean fitness of the AI-bot in this final set of experiments, using the same fitness measure as used in Section 6.4.

6.6 Summary

We have presented the performance of the AI-bot as it evolved in the various areas of its game-play in the game of DEFCON. We have successfully evolved the AI-bot's behaviour trees in the areas which we set out to – silo placement positions, radar placement positions, fleet movements and compositions and timing of attacks. We made use of the best performing trees and combined them to produce a final AI-bot and subsequently ran tests, using this combined implementation, against the default opponent once again in order to collect statistics on its overall performance. In the next chapter, we provide an analysis of the results from this chapter in order to provide insight into the performance of the AI-bot as it evolved in the different experiments, providing proposed explanations for the exhibition of certain behaviours.



	Very High(>100)	High (76-100)	Moderate (51-75)	Low (26-50)	Very Low (1-25)
🟥 Enemy's Winning Margin	1.15%	5.75%	18.39%	25.29%	49.43%
🟦 AI-bot's Winning Margin	0.90%	3.60%	18.02%	31.53%	45.95%

Figure 6.14: Comparison of Average Winning Margins

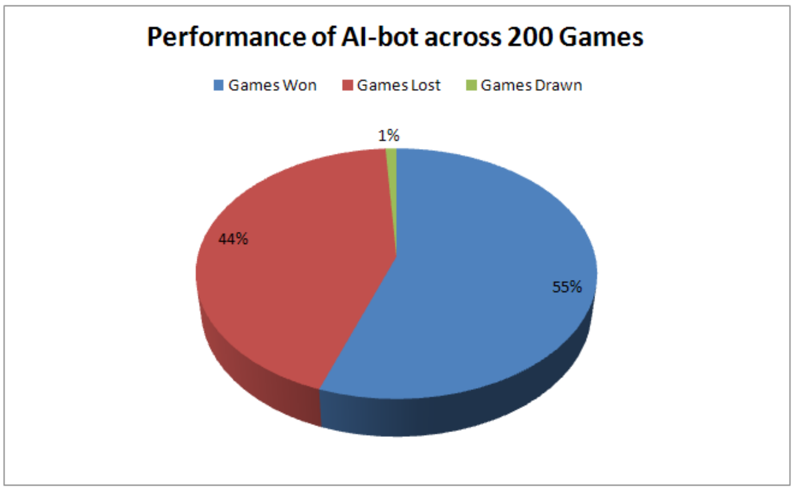


Figure 6.15: Overall Performance of Final AI-bot

Chapter 7

Analysis of Results

From the previous chapter, we presented the results from the experiments conducted in Chapter 5, for which we chose to evolve our existing AI-bot. In this chapter, we aim to perform an analysis of the significance of the results that have been collected, making use of the k-means clustering algorithm (Section 2.5) to perform cluster analysis on the various evolved positions. We also provide insight into the relationship between the results and the behaviours exhibited in our AI-bot. Sections 7.1 and 7.2 cover the analysis of the silo and radar placement experiments respectively. Section 7.3 analyses the results from the fleet placement, movement and composition experiment. Section 7.4 then provides an analysis on the timings of attacks experiment. The analysis of the combined overall performance of the AI-bot is covered in section 7.5. In each of the sections from 7.1 to 7.4, we first analyse the average results from the entire set of evolved poplulations. Following which, we analyse the best-performing behaviour trees in each experiment.

7.1 Silo Placements

By evolving the behaviour trees determining silo placement positions, we were able to see a general increase in fitness over the 80 generations for which the behaviour trees were evolved, as shown in the graph in Figure 6.1 (Section 6.1). This means that the AI-bot was able to shoot down and destroy a greater number of enemy air units as it evolved its silo placement positions. This is further exemplified in Figure 6.3 (Section 6.1), where it can be seen, by comparing the performance of the AI-bot between its initial and final stages of evolution, that on average, the AI-bot showed a 41.1% increase in the number of nukes shot down together with a 129.5% increase in the number of bombers destroyed. The reason a decrease in the number of fighters was observed could be due to the silos being placed further away from the enemy as it evolved, and given that fighters do not have large flying range, most of them failed to reach the areas which our AI-bot's silos were being placed. An interesting point is that in terms of the number of fighters destroyed, only a relatively small 3.5% increase was observed. Figure 7.1 shows an in-game screenshot of the best performing silo placement positions that were obtained from the final evolved set of behaviour trees.

In order to give some insight into the placement positions, Figure 7.2 shows that the silos can be grouped into 3 clusters. The final iteration of the application of the k-means algorithm which determined these clusters in showed in Figure 7.3. Page 103 of the Appendix contains the graphs for each iteration of the algorithm. Intuitively, we can also observe that the silos have

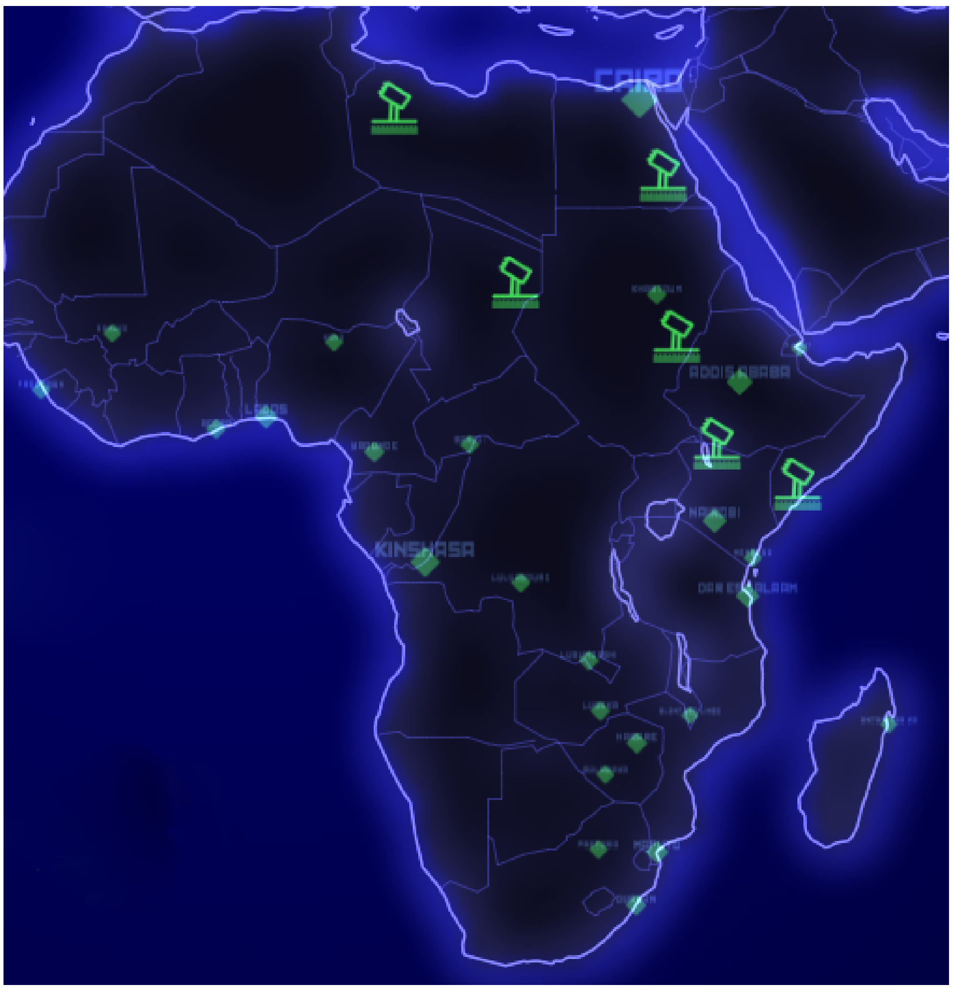


Figure 7.1: In-game view of the final silo placement positions

been placed at the edge of the continent furthest away from the enemy, reducing the chance of detection during the default enemy opponent's scouting stage (Chapter 2, Section 2.1.2).

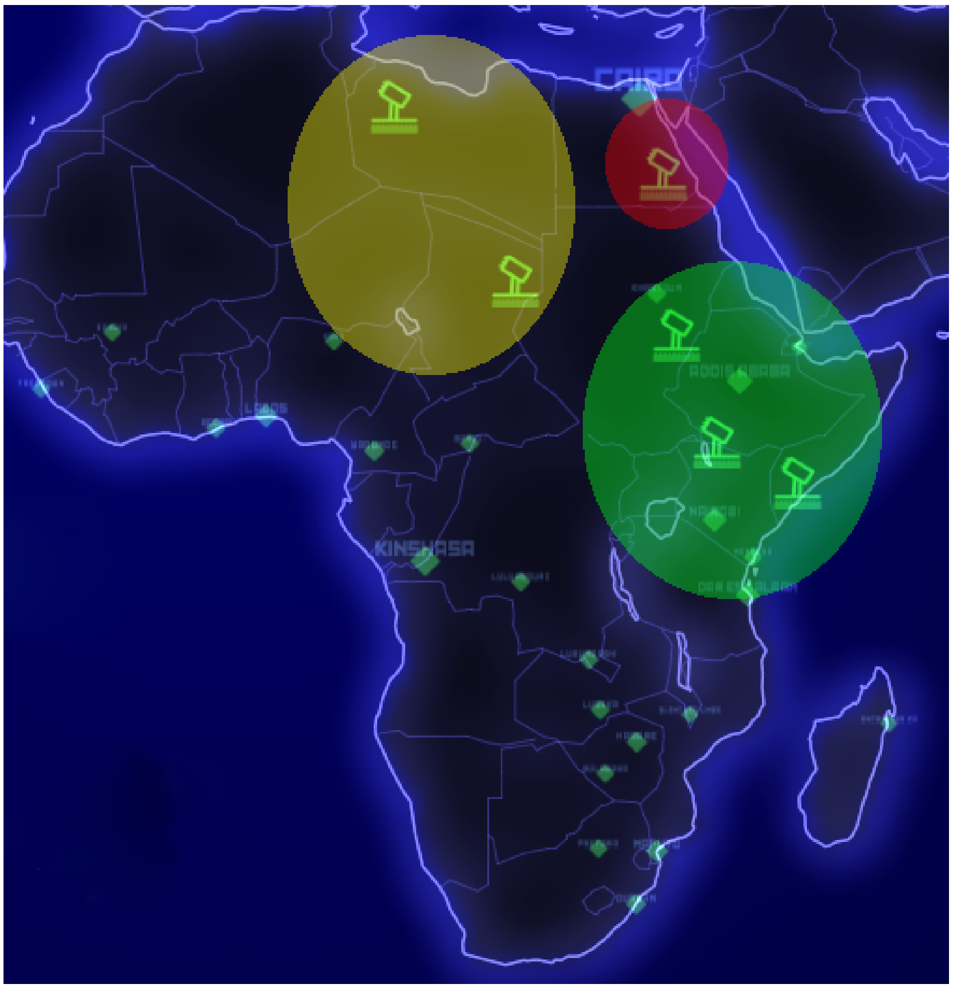


Figure 7.2: The silos can be clustered into 3 groups as shown

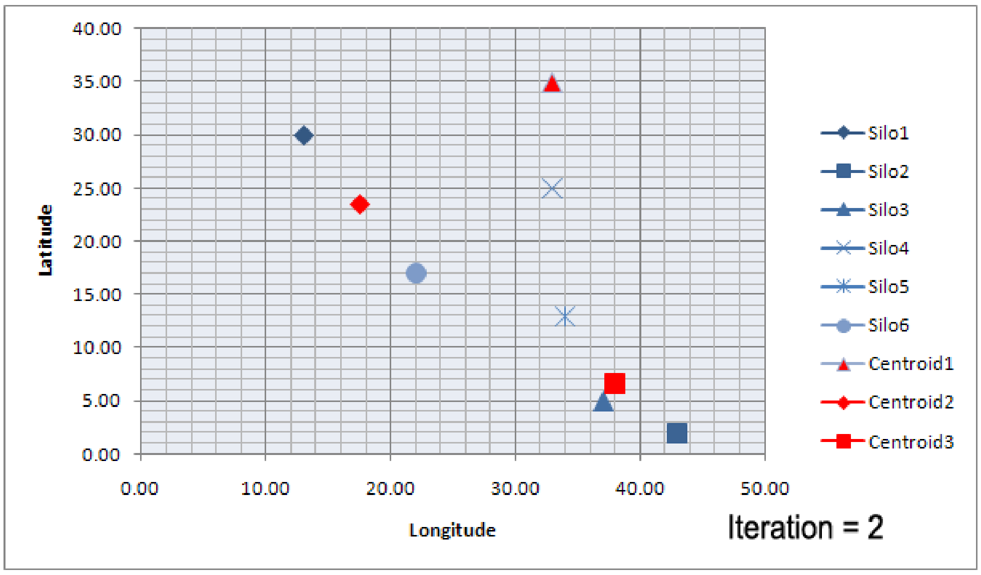


Figure 7.3: Final iteration of the k-means algorithm for silo placements. The red markers denote the means of each cluster, and the blue markers indicate the silo placement positions

7.2 Radar Placements

In evolving the behaviour trees for positioning a player's radars, there was a increase in mean fitness over the 100 generations as shown in Figure 6.4 (Section 6.2), which meant that the AI-bot's ability to position its radars in order to maximise the detection of enemy units showed improvement. Based on the chart in Figure 6.5, we can see that compared to the first population prior to being evolved, the population in the final generation was able to, on average, detect a greater number of enemy units. Only in 4% of the games in the final generation was the AI-bot unable to detect any units at all, compared to 20% at the beginning. Figure 7.4 shows an in-game screenshot of the radar placement positions that were obtained from the best performing evolved behaviour trees.

In order to give some insight into the placement positions, Figure 7.5 shows that the radars can be grouped into 2 clusters, from the results shown in the final iteration of k-means algorithm shown in Figure 7.7. The radars were placed on the edge of the continent closer to the enemy, contrary to the silo placement positions (Section 7.1 above), due to the enemy being located to its west. The two clusters seem to coincide with the positioning and movement of units as performed by the enemy opponent, which were observed to occur throughout the experiments. The default opponent generally places its units in such a way that it approaches the AI-bot's territory via 2 directions. Figure 7.6 shows the general movement directions of the opponent. Thus, the 2 clusters resulting from the positioning of the radars exhibited by the best performing behaviour trees suggest they maximised the chance of detecting the enemy's sea units as they approached from the 2 directions.

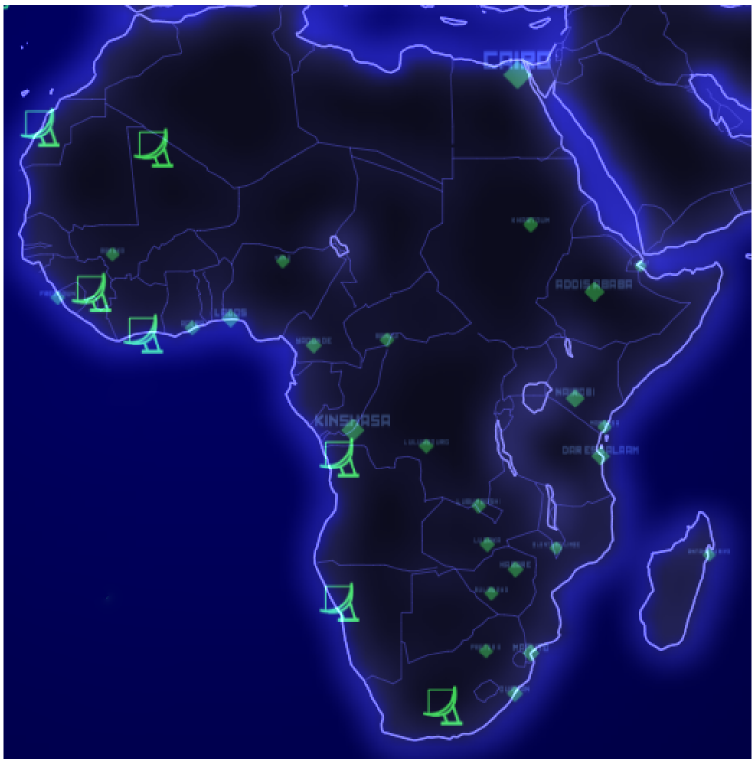


Figure 7.4: In-game view of the final radar placement positions

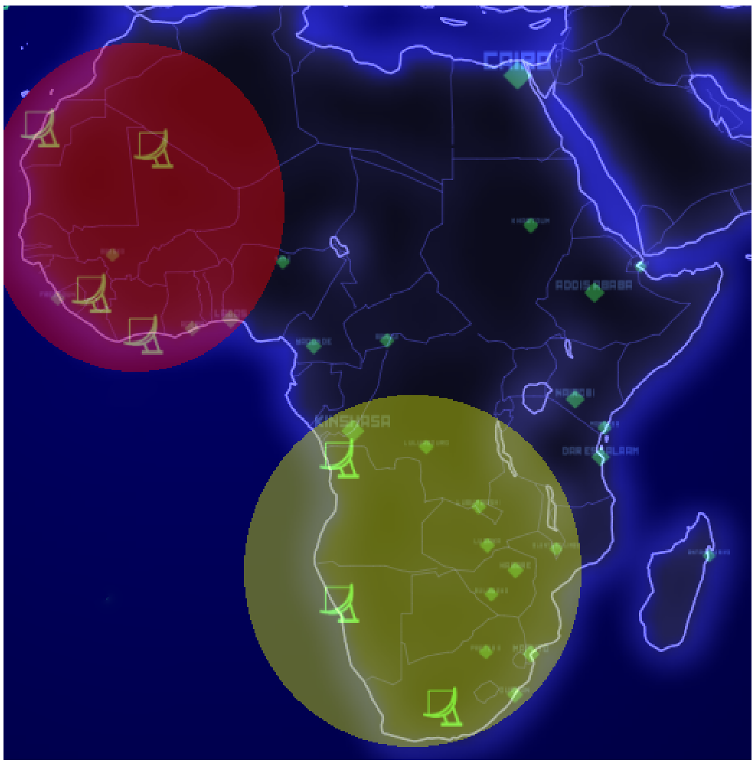


Figure 7.5: The radars can be clustered into 2 groups as shown

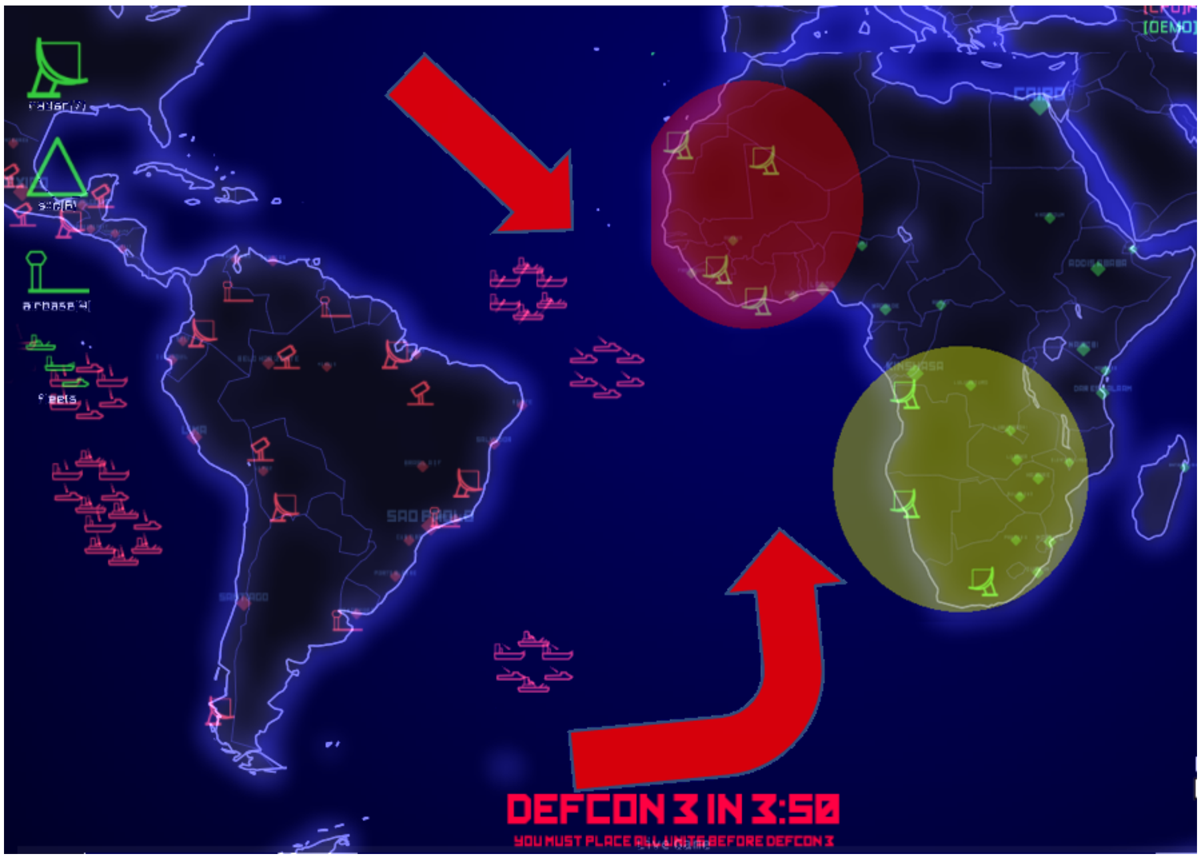


Figure 7.6: General movement direction made by opponent

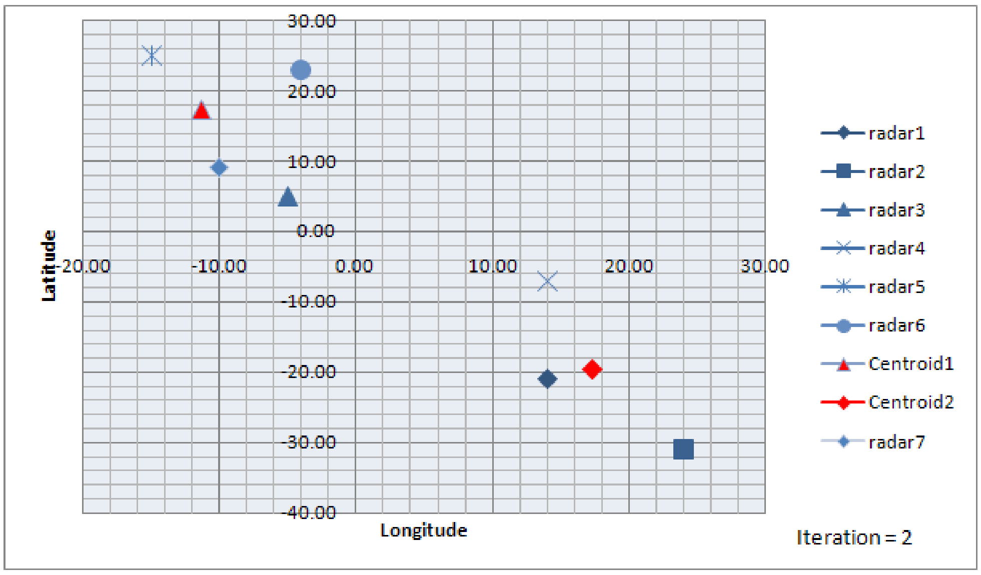


Figure 7.7: Final iteration of the k-means algorithm for radar placements. The red markers denote the means of each cluster, and the blue markers indicate the radar placement positions

7.3 Fleet Placement & Movement

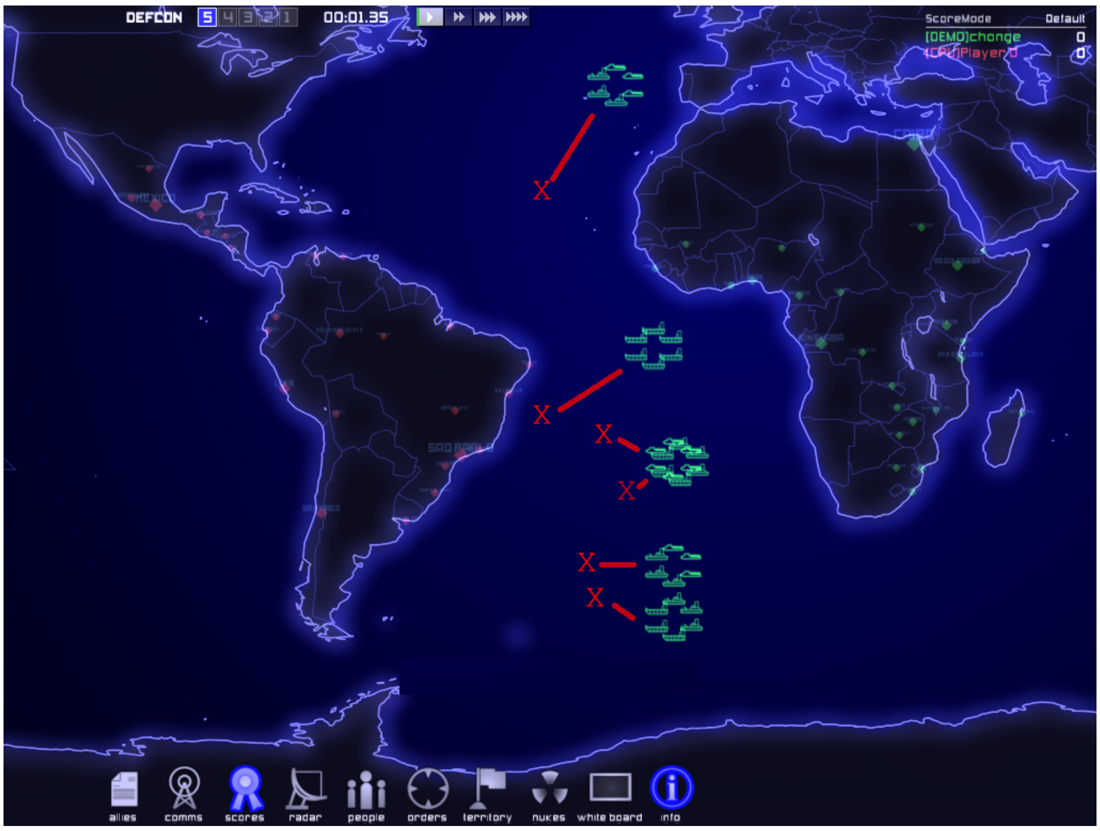


Figure 7.8: In-game view showing Fleet placement and target movement positions

In evolving the positions of fleets for which they were placed and moved, the AI-bot was shown to have improved in performance, with an increase in fitness over the 200 generations for which it was evolved, as shown in Figure 6.6 (Section 6.3). It can be seen that between 0 to 50 generations, there was an increase in fitness, but between 60-100, a drop in fitness was observed. However, by performing the evolution over an additional 100 generations, we were able to observe that the fitness did increase significantly. From the results showing how well the AI-bot performed in the three areas contributing to the overall fitness measure in Figure 6.7 (Section 6.3), it is interesting to note that the most significant increase in performance was for the destruction of enemy sea units. We believe that the individuals performing well in sea-units destruction began to dominate the populations in each generation, thus resulting in a relatively smaller increase in the mean number of detroyed and uncovered silos, and even a decrease in the mean number of uncovered and destroyed airbases and radars, between the first and last generation of individuals.5

We shall attempt to explain this behaviour by providing insight on the way the fleets were positioned and moved based on the final, best-performing behaviour trees. The positions are shown in Figure 7.8 and the red lines show the approximate path the fleets would take in moving towards their respective target positions (denoted with red crosses). By using the k-means clustering algorithm, we note that when placed, the fleets can be categorised into 3 different clusters (Figure 7.9), but on reaching the target positions, the fleets are then categorised into 2 clusters (Figure 7.10). Figures 7.11 and 7.12 show the final iteration of the k-means algorithm as it was applied to the placement and target locations respectively.

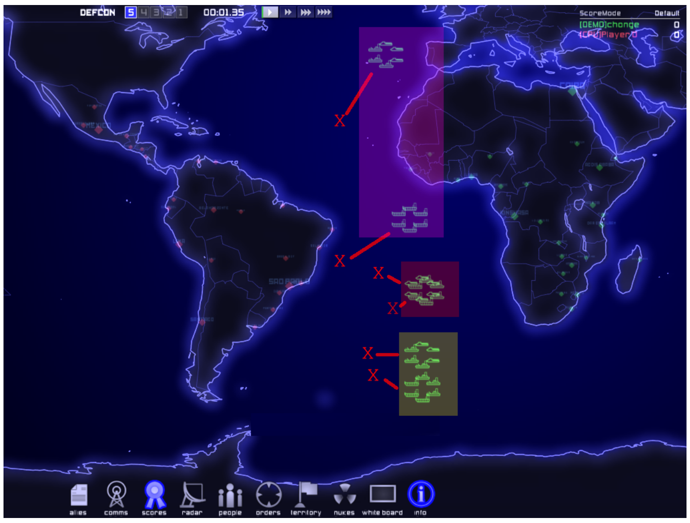


Figure 7.9: When placed, the fleets are clustered into 3 groups

In placing the fleets in three clusters, we observe that in each cluster, both submarine and carrier units are present. Battleships are only present in the top-most and bottom-most of the clusters, and never placed in such a way that a fleet was composed entirely of them. Upon moving to the target positions, the 2 clusters possess all three types of fleet units in them. The presence of 2 clusters suggests that in order to maximise the number of enemy sea units destroyed, the fleets would need to move towards the 2 general directions of which the enemy approaches (Section 7.2 above). The AI-bot was thus able to have a balanced set of offences as it encountered the enemy.

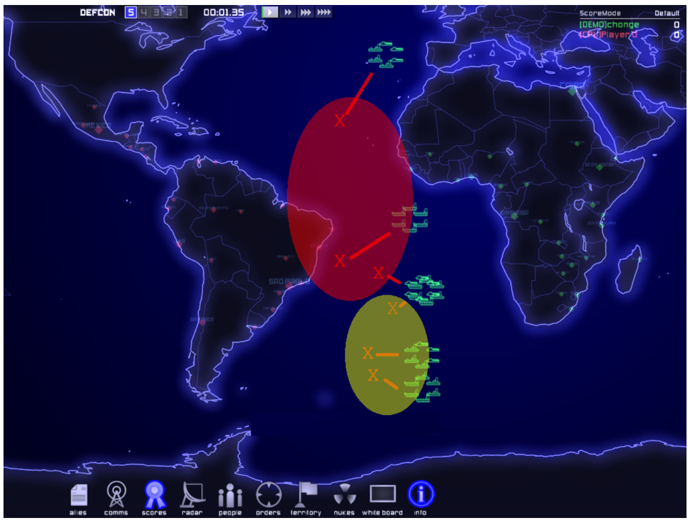


Figure 7.10: When moved, the fleets are clustered into 2 groups

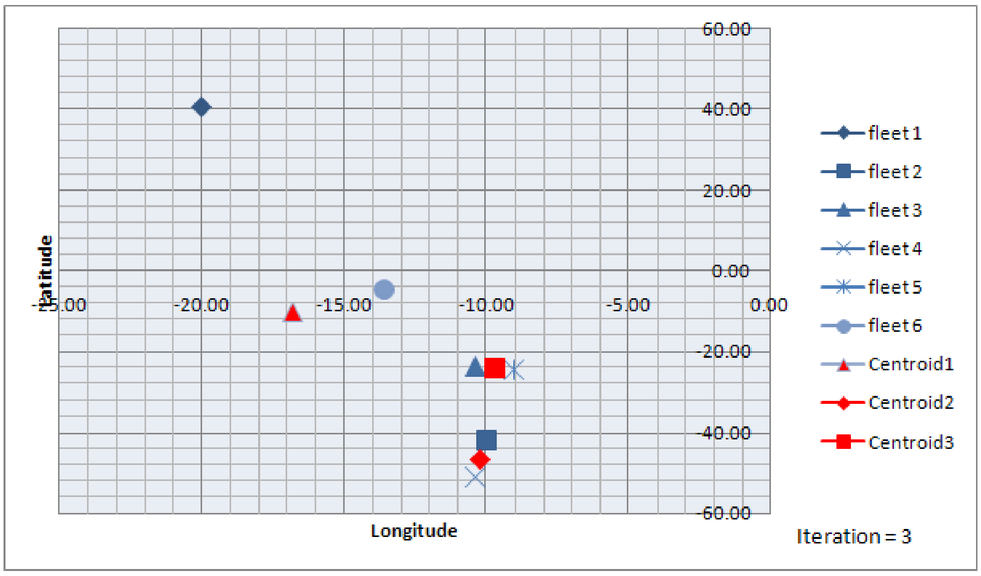


Figure 7.11: Final iteration of the k-means algorithm for fleet starting positions. The red markers denote the means of each cluster, and the blue markers indicate the fleet starting positions.

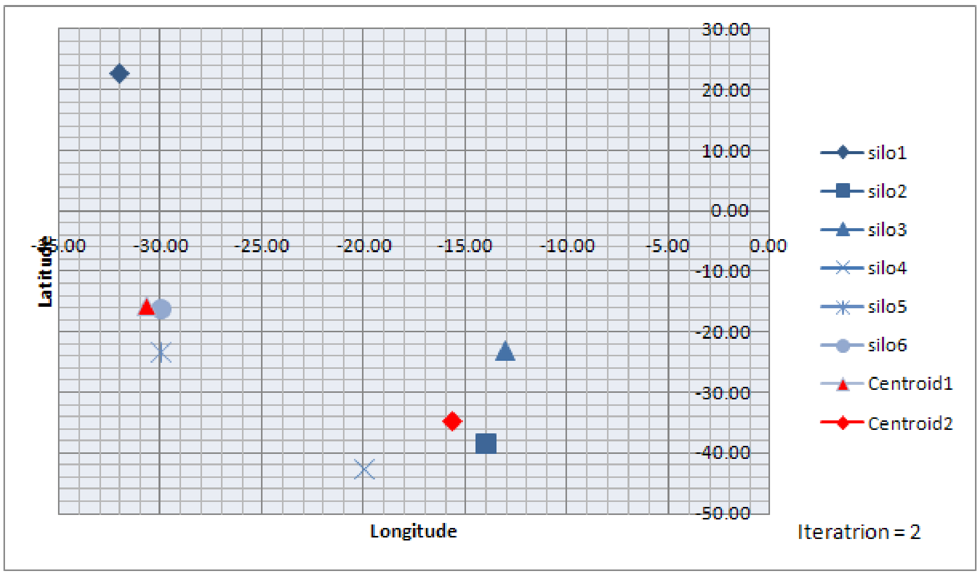


Figure 7.12: Final iteration of the k-means algorithm for fleet target positions. The red markers denote the means of each cluster, and the blue markers indicate the fleet target positions.

7.4 Timing & Synchronisation

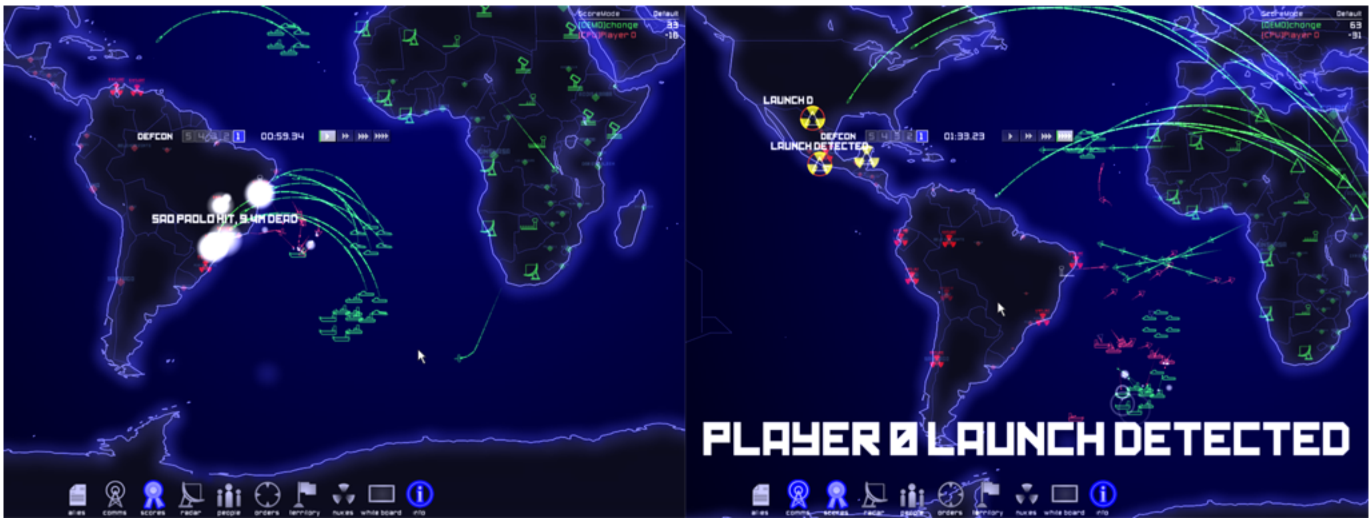


Figure 7.13: Two stages of attack from the evolved timings

Combining the best performing behaviour trees from the previous experiments, we evolved the times in which the AI-bot launched nukes in an offensive attack for 4 key units – submarines, carriers, airbases and silos. From the graph in Figure 6.13 (Section 6.4), we see that there was an increase in the percentage of games won by the AI-bot over 100 generations as the AI-bot evolved. This is further exmpelified in the graph in Figure 6.10 and Figure 6.11 (both from Section 6.4) as the mean difference between the AI-bot's end-game score and the default opponent's end-game score reduced as it evolved. This showed the significance of the timing of attacks, since the same behaviour trees were used to dicate the silo placements, radar placements and fleet movements.

In studying the timings obtained by the best-performing behaviour trees, we observe that the timings defined two stages of attack – one using sea units (carriers and submarines), and the other using ground installations (silos and airbases). This is shown in Figure 7.13. Often, the enemy's sea units would engage in combat with the AI-bot's units at around the hour mark (3600s), and cause a reduction in the number of submarines and carriers left for our attack. Thus, the AI-bot chose to perform its nuke attacks by sea-units before this occurs, so as to be able to inflict the maximum possible damage by having its full strength of fleet units.

For the second stage of attack, we observed that the enemy usually performs its full-scale nuclear attacks at around 5400s. This corresponds to the expiration of its internal attack timer (Section 2.1.2). This is the period of time in which the enemy is unable to defend itself against incoming nuke attacks, and as such, the evolved timings of our AI-bot seem to take advantage of this. The silo and airbase attack times of 4460s and 4331s (Section 6.4) are effective because it considers the time it would take for the nukes and bombers to reach the enemy territory at approximately 5400s, since the airbases are located slightly nearer to the enemies than the silos are.

7.5 Overall Behaviour

Combining the best performing behaviour trees, we were able to produce an AI-bot that was capable of beating the default opponent 55% of the time (Figure 6.15, Section 6.5). However, we are also interested in how convincing our victories, or defeats, were and thus made use of the winning margin to analyse that. From Figure 6.14 (Section 6.5), we observe that there was a high percentage of games (49.43%) that our AI-bot lost where the margin was **very low**. Conversely, our AI-bot beat the default opponent with a **low** margin approximately 6.24% more often than the default AI opponent did.

However, the opponent managed to win by both a **high** and **very high** margin a larger number of times. This would also suggest that our AI-bot did not beat the default opponent by an extremely convincing margin as often as the opponent beat our AI-bot. Overall, both our AI-bot and default opponent managed to beat each other by a moderate margin a similar amount of times. Table 6.5 summarises the performance of our AI-bot, showing that on average, our AI-bot beat the default opponent by a margin of approximately +4.23 points.

7.6 Summary

In this chapter, we have analysed the results from the previous chapter, drawing certain conclusions and insight into the performance of the AI-bot as it evolved. In general, the AI-bot was able to perform better in each of the experiments and areas which it was evolved in. Combining the best performing behaviour trees from each of these experiments, we were able to produce an AI-bot that was capable of beating the default AI opponent approximately 55% of the time. In terms of placement and movement positions, we employed the k-means clustering algorithm to group the set of positions and provided an explanation of why those clusters emerged. We also observed the significance of the timing of the attacks to winning the game. In the next chapter, we provide an overview of the targets met by this project, and the significance of the findings which we have made from the experiments and results that were obtained. We will also provide areas for improvements and lessons learned from this project.

Chapter 8

Conclusion

8.1 Conclusion

We have shown that, by combining the use of behaviour trees and employing an evolutionary approach, we have been able to successfully produce a competitive AI-bot that was capable of beating the original, hand-coded default AI-bot in the game of DEFCON more than 50% of the time (Section 6.5). This would signify that employing such an approach is indeed feasible in the development of automated players for commercial games, which were the key motivations of this project (Section 1.1). This had implied that the time that would have been spent by both designers and programmers in trying to design and tweak intelligent automated players to be focused on other aspects of game development.

While we are not suggesting that there would no longer a need for AI programmers or designers, theoretically, less time would be required of them to spend on fixing and tweaking their programs in order to produce competitive automated players in their games. An approach would instead involve designing simple automated players that are merely able to play the game, and leave the evolution process to evolve the automated player into one that was competitive by running matches against a default opponent, if available, or against human players. The project exemplified the first approach as we first hand-crafted the AI-bot using a set of behaviour trees which allowed it to play the game of DEFCON, making decisions and choices at random (Section 4.1). After which, the AI-bot was evolved into one that was able to play competitively in various aspects of the game (Chapter 6), exhibiting intelligent decisions in the way that it approached the game in order to become a better player (Chapter 7). In the absence of a default opponent or human players, there is the possibility of running matches of games between two different sets of hand-crafted AI-bots, or even against itself [33].

We have also observed that in order to perform the experiments, a large amount of time and resources were required to run the games in order to evolve the AI-bot. In several of the experiments, fluctuations in performance occurred between generations, and usually only after a larger number of generations were we able to conclude that there was a general increase in performance. With services offering application processing for rent, such as Amazon's Elastic Computer Cloud[1], the feasibility of performing even larger numbers of evolution at higher speeds is possible, making it possible for commercial game companies to evolve better and more competitive automated players than what this project could achieve, given its time, resource

[1]For more information , visit http://aws.amazon.com/ec2/

and economical constraints.

8.2 Limitations & Future Work

In this section, we outline several limitations and constraints identified in the development of our AI-bot. We also identify the areas where improvements and further work would be applicable.

8.2.1 Considering Other Starting Positions

The approach that we took allowed us to evolve an AI-bot to be able to play the game of DEFCON competitively using the given set of starting positions for both the AI-bot and the default opponent. Due to the resource and time constraints, we were unable to perform the evolution for each of the different combinations of starting positions. However, the results from our experiments have shown that it is possible to extend this implementation and repeat them for the various starting locations, allowing for the evolution to proceed for the AI-bot to be able to play competitively in all permutations of starting positions possible, thus no longer restricting its optimal performance to the fixed set of starting positions.

8.2.2 Opponents

In this project, we used the default Introversion opponent throughout the entire set of experiments. Even though some variation exists, as the default opponent alters its game play randomly, by placing units at different locations and attacking at different times, it still follows a fixed finite state machine that dictates its behaviour throughout the course of the game. There is the problem that evolving against a single opponent would make our evolved AI-bot competitive only against the default opponent. It would have been good to be able to perform experiments against other AI-bots, and against human players. This would allow the AI-bot to evolve into a competitive player against all different types of opponents. The DedCon project[2] attempts to store the logs from matches between players, allowing replaying of matches where the players follow the exact same moves, and this would be useful in both having a larger opponent base to evolve against as well as for controlling the environment which the AI-bot is evolving in. Given that so far, no other AI-bot has been produced using the DEFCON API, we were unable to have the luxury of testing against other AI-bot implementations. Robin Baumgarten's bot was implemented directly using the source code of DEFCON, and not using the DEFCON API, which also prevented us from competing against it.

8.2.3 Other Gameplay Aspects

We have evolved four areas of our AI-bot's game play and have managed to produce positive results, but there exist several aspects of DEFCON that we have not considered. Firstly, we did not attempt to evolve optimal airbase placement locations, which play the roles of both performing a nuke attack, as well as re-filling the carriers with bombers should they have been

[2]Dedicated Server for DEFCON - http://dedcon.homelinux.net/

destroyed. Given that the demo version of DEFCON was used, we were unable to consider games involving more than 2 players. Such games involve forming alliances and cooperation with other players (or AI-bots), and would definitely be interesting for the application of AI techniques that would allow AI-bots to be able to co-operate and decide when to break and form alliances. Finally, there is the consideration of the different game and scoring modes which DEFCON permits, which we were unable to explore too, with the demo version.

8.2.4 Further Applications of Behaviour Trees

Our behaviour tree implementation was developed by us, with the aims of being a simple, but sufficiently powerful, framework for us to develop our AI-bot for DEFCON. We have not covered all the possible elements of the use of behaviour trees, including the use of `parallel` nodes, which are used to constantly check for game-conditions whilst performing some other actions. We were able to make up for this using `sequence` to performing the checking of conditions because the amount of time required to traverse the behaviour tree was small enough to fit into a single game tick. However, when dealing with a larger number of trees, the traversal may be required to span over several ticks, and as such, there are some applications and games where the use of `parallel` nodes are useful in designing intelligent behaviours.

The second area involves the evolution of sub-trees such that each sub behaviour tree dictates a behaviour which matches a certain style of play – examples being more defensive, attack-orientated, cooperative or merely making random decisions at appropriate points. Our implementation only considered the transition between two stages, from being defensive to launching an attack, which had to occur in that order. It would be interesting to see the application of the evolution of sub-trees using `lookup` decorators to allow the AI-bot to exhibit behaviours which are more complex and allow it to exhibit adaptive game play styles to match opponents. An application of evolution to make further use of the lookup decorators to evolve the higher-level branches of the trees and explicitly performing genetic operators on them would be an interesting approach.

8.2.5 DEFCON Competition

The IEEE Symposium on Computational Intelligence and Games 2009 (IEEE CIG09) event consists of a DEFCON competition[3] in which AI-bots that have been implemented using the DEFCON API would be able to compete against each other in order to determine the best performing one. We believe that with further refinement, that our AI-bot would be a suitable competitor at this event, with the added hope that it might emerge victorious in the competition.

8.3 Closing Remarks

Overall, we believe that this project has been successful, not only in developing a competitive AI-bot, but also exhibiting the use of a different approach to designing competitive automated players in games – an evolutionary approach. We were able to work under the resource, economical and time constraints which commercial game companies would possibly not be as restricted

[3]Competition page at http://www.ieee-cig.org/

by, to show that the evolutionary approach using behaviour trees was feasible and effective. The fact that this project was applied to a commercial game such as DEFCON would possibly allow the game industry to consider such an approach seriously. We are definitely excited to see the progress of automated player development using AI techniques that is to come, and hope to see this project to be an inspiration to the development of other novel and interesting techniques that would be able to further push the boundaries of AI in games.

Appendix

List of Decorators

This presents a non-exhaustive list of proposed **Decorators** types, as suggested by Alex J. Champandard [9].

- **Filters**: These prevent a behavior from activating under certain circumstances.
  - Limit the number of times a behavior can be run
  - Prevent a behavior from firing too often with a timer
  - Temporarily deactivate a behavior by script
  - Restrict the number of behaviors running simultaneously
- **Managers & Handlers**: These are decorators that are responsible for managing whole subtrees rather than single behaviors.
  - Deal with the error status code and restart planning or execution
  - Store information for multiple child nodes to access as a blackboard
- **Control Modifiers**: These decorators are used to pretend that the execution of the child node happened differently
  - Force a return status, e.g. always fail or always succeed
  - Fake a certain behavior, i.e. keep running instead of failing or succeeding
- **Meta Operations**: Such decorators are useful during development, and can be inserted into a tree automatically when needed.
  - Debug breakpoint; pause execution and prompt the user
  - Logging; track this node execution and print to the console

List of `EventTypes`

The following presents the list of `EventTypes` that are detected by the DEFCON API event system:

- **EventPingSub**: Event when a submarine uses its sonar to detect enemies.
- **EventPingCarrier**: Event when a carrier uses its sonar to detect enemies.
- **EventNukeLaunchSilo**: Event when a silo nuke launch is detected.
- **EventNukeLaunchSub**: Event when a submarine nuke launch is detected.
- **EventHit**: Event when a unit is hit by gunfire from battleships.
- **EventDestroyed**: Event when a unit is destroyed.
- **EventPingDetection**: Event when a sonar detection of an enemy is detected.
- **EventCeasedFire**: Event when a ceasefire is called.
- **EventUnceasedFire**: Event when a ceasefire no longer holds.
- **EventSharedRadar**: Event when parties begin to share radar coverage.
- **EventUnsharedRadar**: Event when parties no longer share radar coverage.
- **EventNewVote**: Event when a new vote is cast by a party.
- **EventTeamVoted**: Event when a vote is cast by a party or alliance.
- **EventTeamRetractedVote**: Event when a vote retracted by a party of alliance.
- **EventVoteFinishedYes**: Event when voting has ended
- **EventVoteFinishedNo**: Event when voting has not yet ended.

Command Line Parameters

The following are command line arguments that may be used when running the executable with the DEFCON API:

- **maxgametime=n**: Sets the maximum in-game time to n seconds before the victory timer starts.
- **internalais=n**: Sets the number of internal ais used.
- **internalaiterritory=n**: Sets the territory of an internal default opponent, and repeatable for the number of default opponents required. The `internalais` from above is automatically set to the correct size.
- **repeat=n**: Repeats the game n times. If set, the current game will end immediately after the victory timer ends and return to the host screen. DEFCON will exit when all the games have been completed.
- **fastserveradvance**: Enables a quicker server update.

Complete Set of Behaviour Trees Used

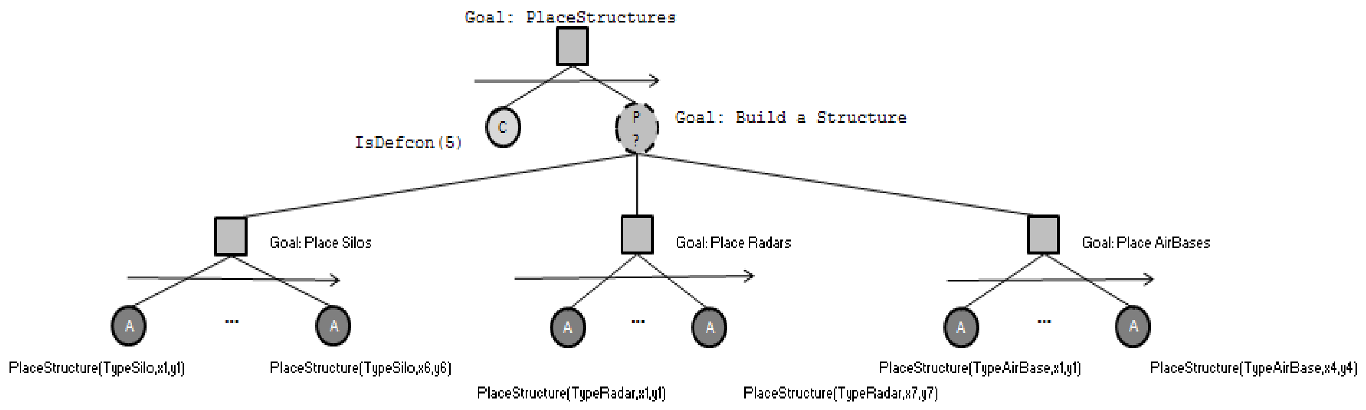

Goal: PlaceStructures
IsDefcon(5)
C
P ?
Goal: Build a Structure
Goal: Place Silos
Goal: Place Radars
Goal: Place AirBases
A
...
PlaceStructure(TypeSilo,x1,y1)
PlaceStructure(TypeSilo,x6,y6)
PlaceStructure(TypeRadar,x1,y1)
PlaceStructure(TypeRadar,x7,y7)
PlaceStructure(TypeAirBase,x1,y1)
PlaceStructure(TypeAirBase,x4,y4)

Figure 8.1: Behavior Tree: PlaceStructures

Figure 8.1 (PlaceStructures): The behaviour tree executes by first checking if the current DEFCON level in the game is 5. If so, it will proceed to place silos, radars and airbases in that order using a `priority selector`. The left most `sequence` places 6 silos at their specified locations, the middle `sequence` places 7 radars at specified locations and the right most `sequence` places 4 radar stations at specified locations.

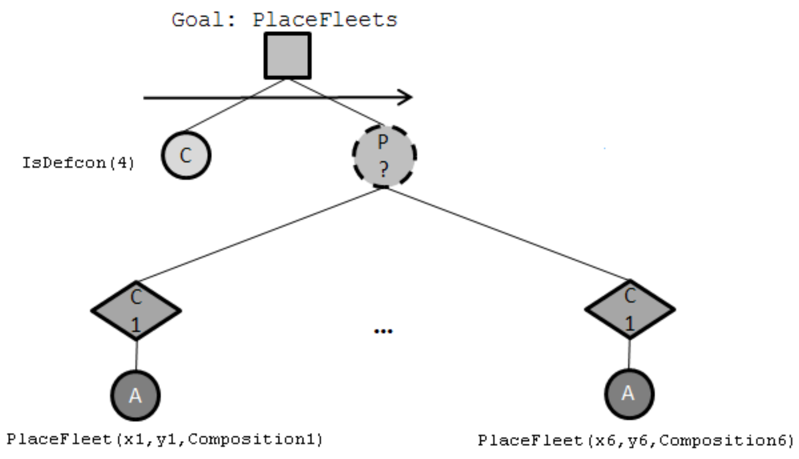

Goal: PlaceFleets
IsDefcon(4)
C
P ?
C 1
...
A
PlaceFleet(x1,y1,Composition1)
PlaceFleet(x6,y6,Composition6)

Figure 8.2: Behavior Tree: PlaceFleets

Figure 8.2 (PlaceFleet): The behaviour tree executes by first checking if the current DEFCON level in the game is 4 and if so, it will proceed to place the fleet units using the `priority selector`. The counter `decorator` node prevents the AI-bot from placing the same fleet twice by restricting the number of times its child is allowed to execute to 1. The behaviour tree copes with the failure to place fleets because of the `priority selector`, which ensures that if it fails to place a fleet unit from once branch, it would proceed on to try to place the next fleet as specified by the next branch.

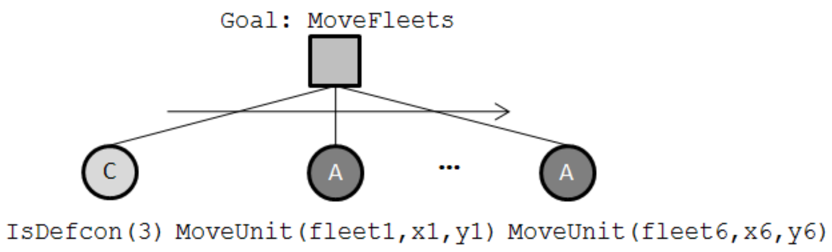


Figure 8.3: Behavior Tree: MoveFleets

Figure 8.3 (MoveFleets): The behaviour tree executes by first checking if the current DEFCON level in the game is 3, and then it proceeds to move all of its fleets to their specified target locations.

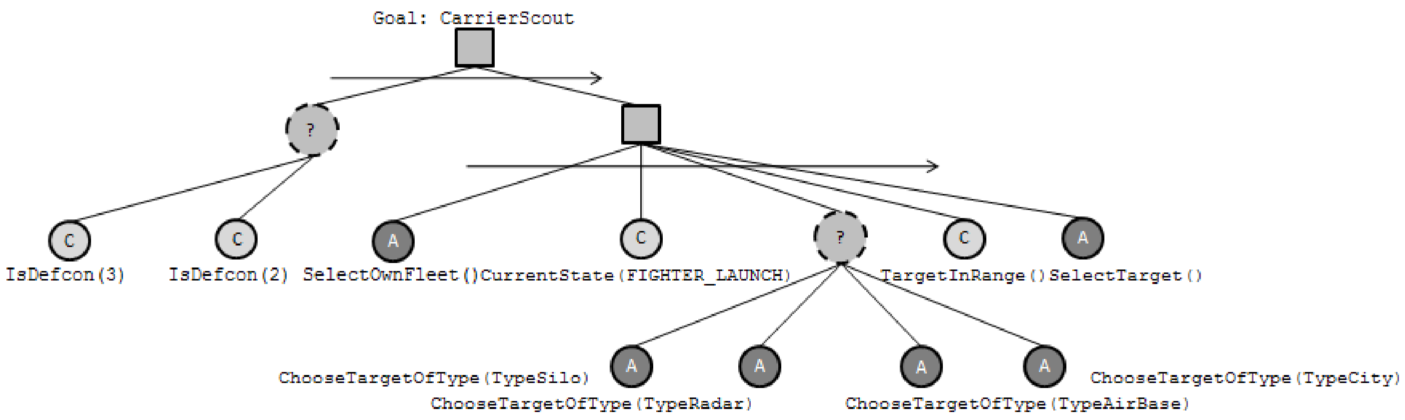

Goal: CarrierScout
?
C
C
A
C
?
C
A
IsDefcon(3)
IsDefcon(2)
SelectOwnFleet()
CurrentState(FIGHTER_LAUNCH)
TargetInRange()
SelectTarget()
A
A
A
A
ChooseTargetOfType(TypeSilo)
ChooseTargetOfType(TypeRadar)
ChooseTargetOfType(TypeAirBase)
ChooseTargetOfType(TypeCity)

Figure 8.4: Behavior Tree: CarrierScout

Figure 8.4 (CarrierScout): The behaviour tree excutes whenever the current DEFCON level within the game is either 3 or 2. If so, it proceeds to select a carrier unit belong to the AI-bot. It then checks that the carrier is in “fighter launch” mode. It then chooses any visible enemy silo, radar, airbase or city as a point for it to launch its fighters at. It proceeds to check if the selected building or city is within range before setting it as its active target.

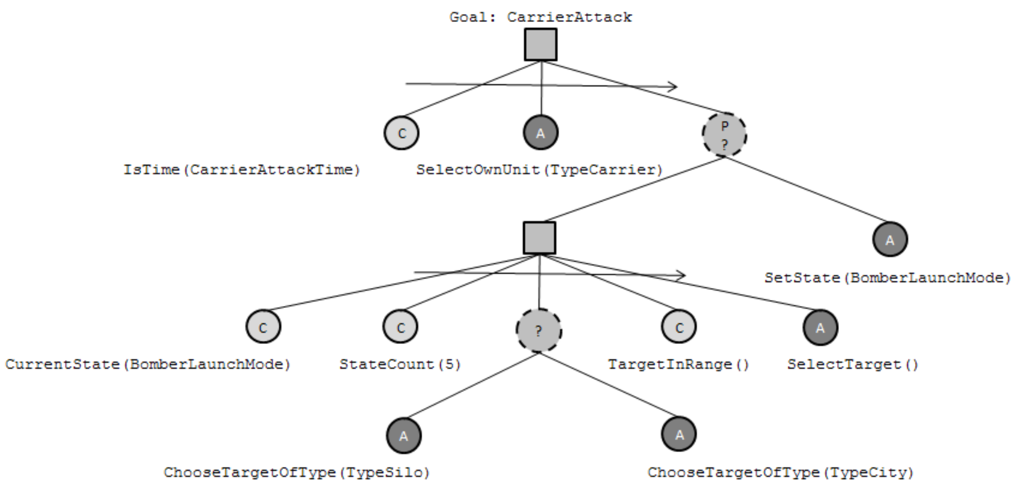

Goal: CarrierAttack
C
A
P ?
IsTime(CarrierAttackTime)
SelectOwnUnit(TypeCarrier)
A
SetState(BomberLaunchMode)
C
C
?
C
A
CurrentState(BomberLaunchMode)
StateCount(5)
TargetInRange()
SelectTarget()
A
A
ChooseTargetOfType(TypeSilo)
ChooseTargetOfType(TypeCity)

Figure 8.5: Behavior Tree: CarrierAttack

Figure 8.5 (CarrierAttack): The behaviour tree executes by first checking if the in-game time corresponds to its predefined `CARRIERATTACKTIME` – denoting the time at which the AI-bot would launch bombers from its carrier units against the enemy. When it is past this specified time, the AI-bot will select a carrier which belongs to it. It will then check the carrier it selected is in bomber launch mode, failing which, the `priority selector` cause the left hand branch to fail, leading to the right hand branch which sets the state of the carrier to the required bomber launch mode. In the event the carrier was in the required mode, it would check if it possessed 5 bombers (state activations) using the `StateCount` node. It then chooses either an enemy silo or city to attack. If the chosen enemy building is in range, it then selects it as its target for a bomber launch.

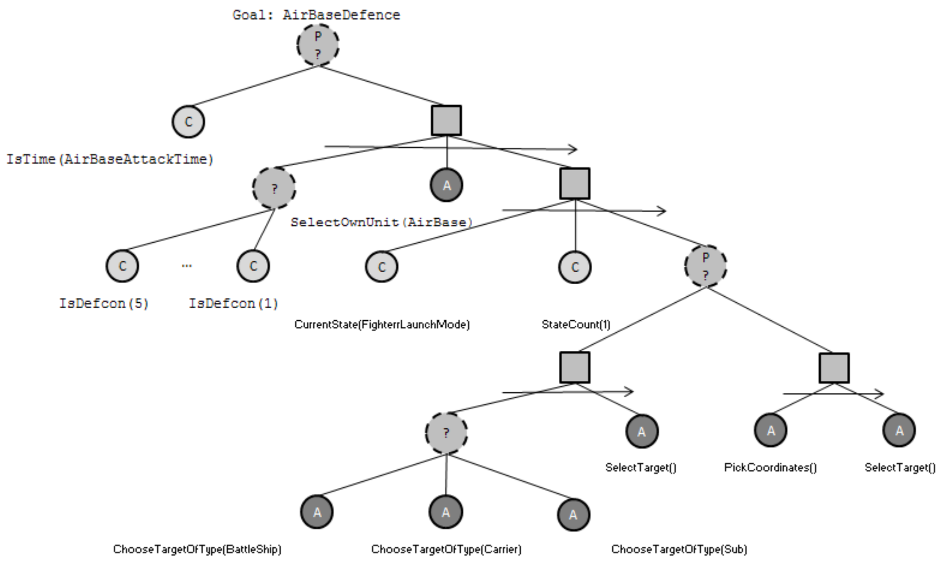


Figure 8.6: Behavior Tree: AirBaseDefence

Figure 8.6 (AirBaseDefence): The behaviour tree executes by first checking if the in-game time corresponds to its predefined `AIRBASEATTACKTIME` – denoting the time at which the AI-bot would launch bombers from its airbases against the enemy. It it is, the behaviour tree does not proceed on due to the `priority selector` succeeding at its first child node. This prevents the air bases from switching back into a defensive mode when its `AIRBASEATTACKTIME` has past. In the situation that the in-game time is still earlier than the specified time, the behaviour tree would continue to check if it was in any DEFCON level, before selecting an air base unit belonging to the AI-bot. It then proceeds to the `sequence` checking if the air base was in fighter launch mode, and that it had at least 1 fighter to launch, using the `StateCount` node. It will then if any enemy sea units have been detected and send fighters to attack them. Otherwise, it randomly picks coordinates to launch its fighters to scout.

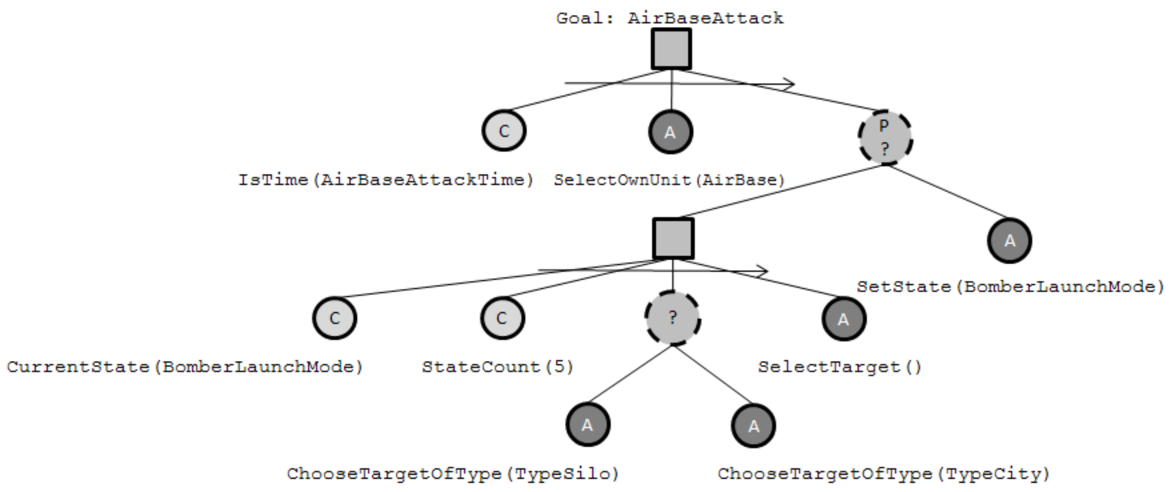

Goal: AirBaseAttack
C
A
P ?
IsTime(AirBaseAttackTime)
SelectOwnUnit(AirBase)
A
SetState(BomberLaunchMode)
C
C
?
A
CurrentState(BomberLaunchMode)
StateCount(5)
SelectTarget()
A
A
ChooseTargetOfType(TypeSilo)
ChooseTargetOfType(TypeCity)

Figure 8.7: BehaviorTree: AirBaseAttack

Figure 8.7 (AirBaseAttack): The behaviour tree executes by first checking if the current in-game time has reached its pre-selected `AirBaseAttackTime`. If so, it selects an air base unit and proceeds on to the `priority selector`. It checks that the selected air base is in bomber launch mode, and that it has 5 bombers for launch using the `StateCount` node. It then chooses either an enemy silo or city to attack before selecting it as a target for its bombers. We do not check whether the target is in range in this instance since bombers are able to cover the entire map. On the occasion that the air base was not in its bomber launch mode, the left hand sub tree of the `priority selector` would have failed, leading to it changing the state of its air base to the required mode.

Figure 8.8 (SubDefence): The behaviour executes by first checking if the in-game time has reached its pre-selected `SubAttackTime`. If so, the behaviour tree does not proceed on further because the `priority selector` would have succeeded based on its first child node. If the in-game time was less than the specified `SubAttackTime`, then the behaviour tree would proceed to select a submarine unit belonging to the AI-bot. It would then switch its current state to the active sonar mode if it wasn't already in it.

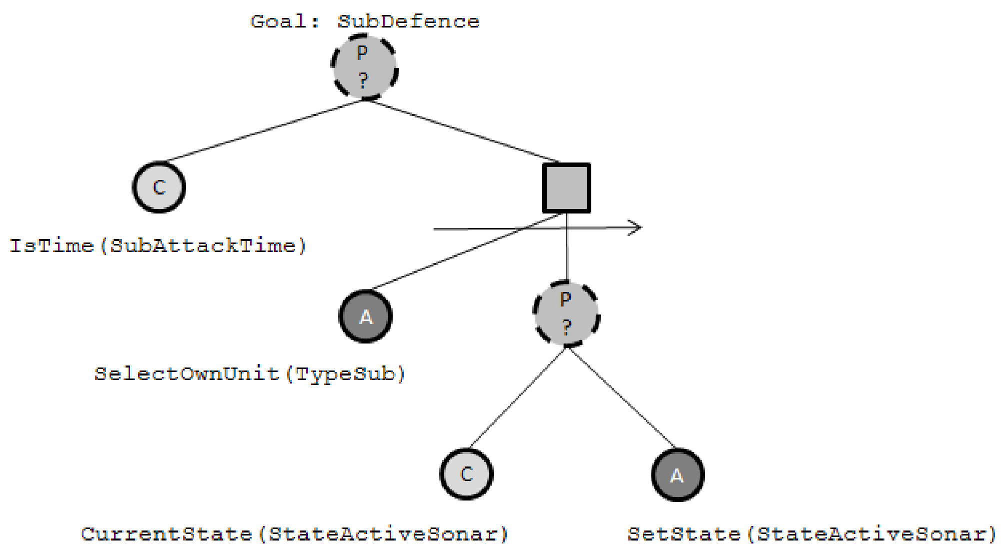


Figure 8.8: Behavior Tree: SubDefence

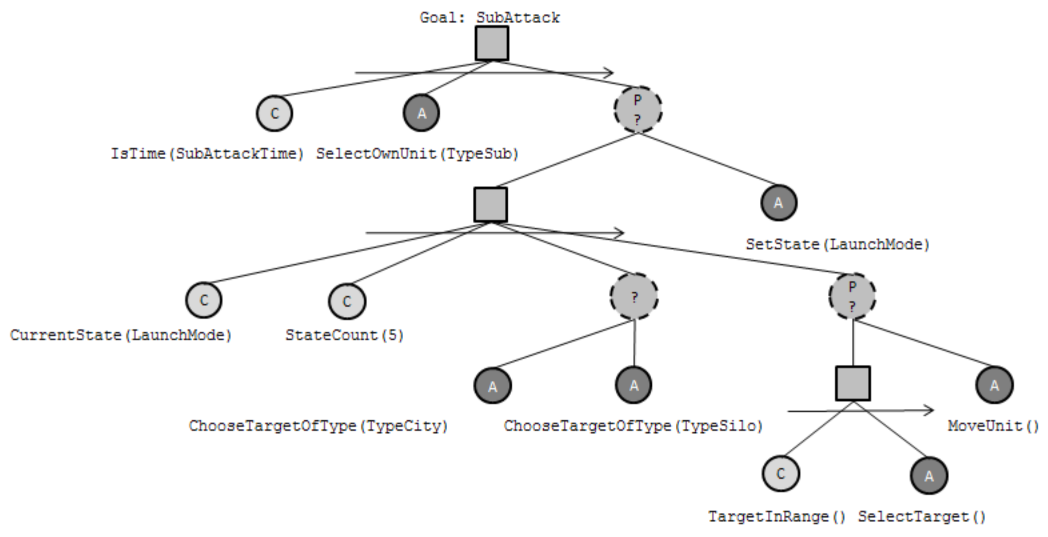


Figure 8.9: Behavior Tree: SubAttack

Figure 8.9 (SubAttack): The behaviour tree executes by first checking if the in-game time has reached its pre-selected `SubAttackTime`. If so, it proceeds to select a submarine belonging to the AI-bot. It then proceeds to the `priority selector`. It checks if the selected submarine is in its launch mode, and that it has 5 available nukes for launch, using the `StateCount` node. It then chooses an enemy city or silo to attack. It proceeds to check if the selected city or silo is within range, and proceeds to launch its nukes against it. If not, it moves the submarine unit towards it. On the occasion that the submarine was not in its launch mode, the left hand subtree of the first `priority selector` from the top would have proceeded to its right hand sub tree in order to set the submarine to the required state.

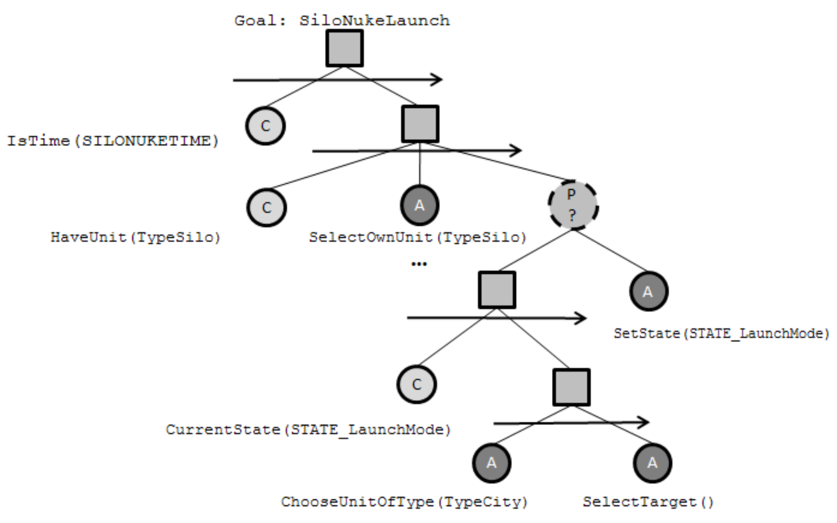


Figure 8.10: Behavior Tree: SiloNukeLaunch

Figure 8.10 (SiloNukeLaunch): The explanation for this tree was covered in page 49 of the report.

Application of the k-means algorithms to the various experiments

We present the following figures:

- **Figure 8.11**: This shows the iterations as the k-means algorithm was applied to the evolved silo positions (Section 7.1.
- **Figure 8.12**: This shows the iterations as the k-means algorithm was applied to the evolved radar positions (Section 7.2.
- **Figure 8.13**: This shows the iterations as the k-means algorithm was applied to the evolved fleet placement positions (Section 7.3.
- **Figure 8.14**: This shows the iterations as the k-means algorithm was applied to the evolved fleet target positions (Section 7.3.

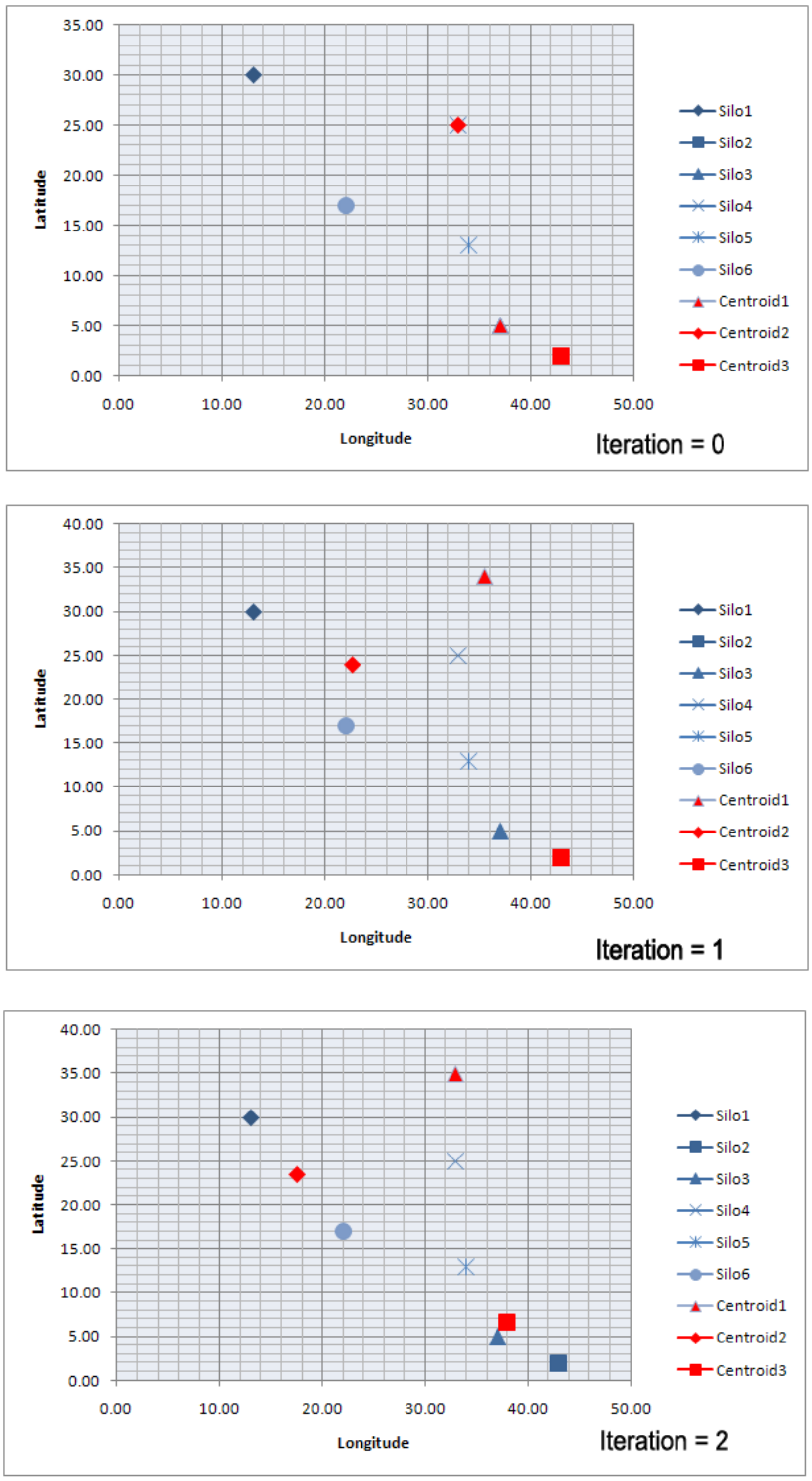


Figure 8.11: Application of k-means Algorithm over several iterations. The red markers indicate the mean values of each of the clusters, and the blue markers show the positions of each silo.

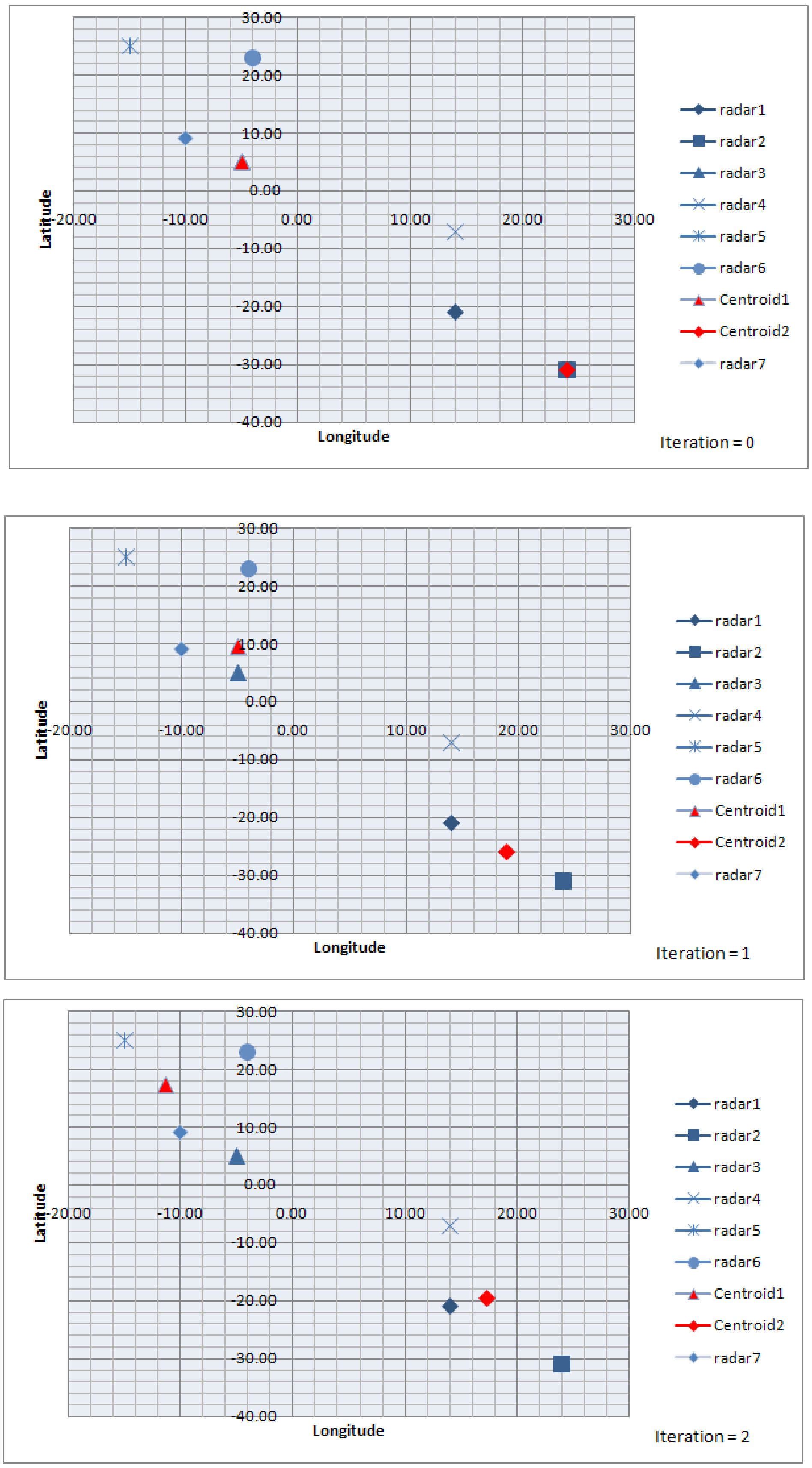


Figure 8.12: Application of k-means Algorithm over several iterations. The red markers indicate the mean values of each of the clusters, and the blue markers show the positions of each radar.

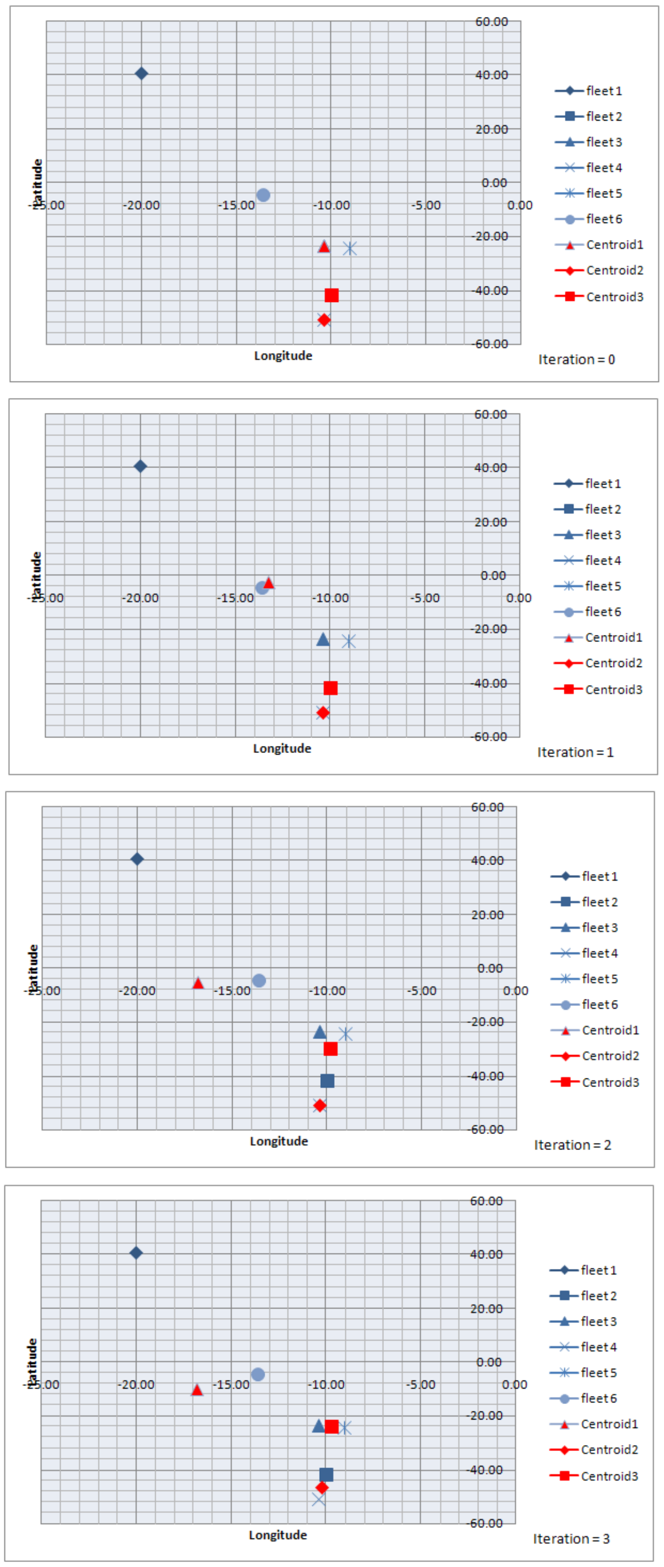


Figure 8.13: Application of k-means Algorithm over several iterations. The red markers indicate the mean values of each of the clusters, and the blue markers show the positions of each fleet's starting position.

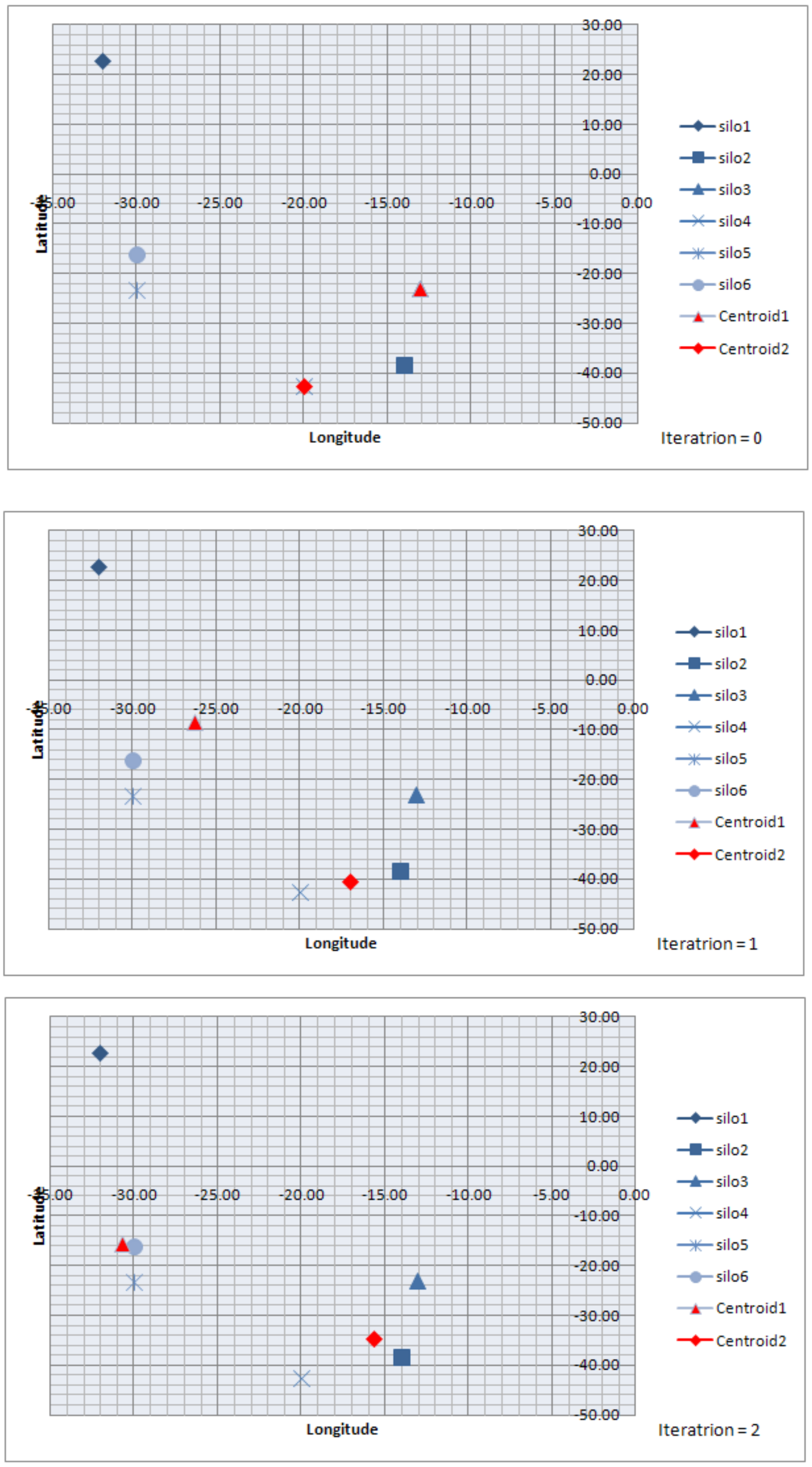


Figure 8.14: Application of k-means Algorithm over several iterations. The red markers indicate the mean values of each of the clusters, and the blue markers show the positions of each fleet's target position.

Bibliography

[1] Christian Bauckhage and Christian Thurau. Exploiting the fascination: Video games in machine learning research and education. 2003.

[2] Robin Baumgarten and Simon Colton. Case-based player simulation for the commercial strategy game defcon. *CGames, 2007*, 2007.

[3] Michael Bowling, Johannes Furnkranz, Thore Graepel, and Ron Musick. Machine learning and games. 2006.

[4] R. Brooks. A robust layered control system for a mobile robot. *IEEE journal of robotics and automation*, 2(1):14–23, 1986.

[5] J.J. Bryson. Action selection and individuation in agent based modelling. In *Proceedings of Agent*, pages 317–330, 2003.

[6] Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. *Lecture Notes in Computer Science*, 2592, January 2003.

[7] Alex J. Champandard. Behavior trees for next-gen game ai part 1. `http://aigamedev.com/videos/behavior-trees-part1`, 2007.

[8] Alex J. Champandard. Understanding behavior trees. `http://aigamedev.com/hierarchical-logic/bt-overview`, 2007.

[9] Alex J. Champandard. Using decorators to improve behaviors. `http://aigamedev.com/hierarchical-logic/decorator`, 2007.

[10] Kevin Dill. Prioritizing actions in a goal-based rts ai. *AI Game Programming Wisdom*, 3, 2006.

[11] Sergio Garces. Extending Simple Weighted-Sum Systems. *AI Game Programming Wisdom 3*, 2006.

[12] Vernon Harmon. An economic apporach to goal-directed reasoning in an rts. *AI Game Programming Wisdom*, 3, 2006.

[13] A. Hauptman and M. Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447, pages 120–131. Springer.

[14] Chris Hecker, Lauren McHugh, Marc-Antoine Argenton, and Max Dyckhoff. Three approaches to halo-style behavior tree ai. GDC 2007 Audio Talk.

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

[16] Yaochu Jin, M. Olhofer, and B. Sendhoff. A framework for evolutionary optimization with approximate fitnessfunctions. *Evolutionary Computation, IEEE Transactions*, 6, 2002.

[17] Geraint Johnson. Goal trees. *AI Game Programming Wisdom*, 3, 2006.

[18] John-Paul Kelly, Adi Botea, and Sven Koenig. Planning with hierarchical task networks in video games. 2007.

[19] Jurgen Kreuziger. Application of machine learning to robotics: An analysis. *ICARCV '92*, 1992.

[20] J.E. Laird and M. Van Lent. Human-level AIs killer application. *AI magazine*, 22(2), 2001.

[21] Francois Dominic Laramee. Genetic algorithms: Evolving the perfect troll. *AI Game Programming Wisdom*, 1, 2002.

[22] S. Luke et al. Genetic programming produced competitive soccer softbot teams for robocup97. *Genetic Programming*, pages 214–222, 1998.

[23] D.J.C. MacKay. *Information theory, inference and learning algorithms.* Cambridge University Press, 2003.

[24] P. Maes. Situated agents can have goals. *Designing autonomous agents: Theory and practice from biology to engineering and back*, pages 49–70, 1990.

[25] M. Mateas and A. Stern. Facade: An experiment in building a fully-realized interactive drama.

[26] Ian Millington. *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

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

[28] J. Orkin. Three states and a plan: the AI of FEAR. In *Proceedings of the 2006 Game Developers Conference.* Citeseer, 2006.

[29] Maja Pantic. Lecture 9-10: Genetic algorithms, December 2009.

[30] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Aproach.* Prentice Hall, 2nd edition, 2008.

[31] A.L. Samuel and Stanford Artificial Intelligence Laboratory. *Some studies in machine learning using the game of checkers II-recent progress.* Pergamon Press, 1969.

[32] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518, 2007.

[33] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[34] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[35] R. S. Sutton and A. G. Barto. Course notes: Reinforcement learning i: An introduction, 1998.

[36] Kardi Teknomo and Zlatan Aki Mur. Numerical example of k-means clustering.

[37] Dale Thomas. Encoding schemes and fitness functions for genetic algorithms. *AI Game Programming Wisdom*, 3, 2006.