PRACTICAL AND THEORETICAL ISSUES OF EVOLVING BEHAVIOUR TREES FOR A TURN-BASED GAME

by

Bentley James Oakes

School of Computer Science
McGill University, Montreal, Quebec

August 2013

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Abstract

The concept of evolving components of an artificial intelligence (AI) has seen increased interest in recent years as the power and complexity of AI has grown. In entertainment software, this AI can impact the player's experiences and enjoyment through elements such as the level of difficulty of the player's competition. Therefore AI development is an important research topic, especially as development is considered difficult by the video game industry. This work applies the evolutionary computing paradigm to a turn-based domain by evolving team strategies. These strategies are represented as behaviour trees, a formalism found in the video game industry and well-suited to the evolutionary algorithm due to their flexibility and tree structure. During the evolutionary process, strategies are evaluated in Battle for Wesnoth, an open-source game with a stochastic nature. A fitness function is defined to assign strategies a numerical strength value, along with a second performance metric that is robust to the variance found in the domain. The evolutionary algorithm then recombines strategies with high strength values, using evolutionary operators from the literature such as crossover and mutation. Later experiments focus on evolutionary algorithm parameters, including comparing a variety of fitness functions to provide insights into their use.

Starting from a number of initial states, numerous strategies are evolved using this algorithm. These initial states are a null strategy, randomly-generated strategies, and a number of hand-built strategies. The evolved strategies are then evaluated in-game, and will be shown to be measurably stronger than the initial strategies.

Résumé

L'application de l'informatique évolutive au domaine de l'Intelligence Artificielle (IA) émerge naturellement. Pour l'industrie du divertissement (jeux ludiques), IA fait référence aux différents éléments visuels ou non qui déterminent une partie importante de l'expérience du joueur. Cette responsabilité fait en sorte que développer des système d'IA n'est pas une tâche simple, ce qui en fait un domaine de recherche intéressant. Ce travail applique les concepts d'informatique évolutive a un jeu par tour en lassant évoluer des stratégies d'IA. Les stratégies sont représentées par des arbres de comportement. Nous retrouvons cet automate dans l'industrie des jeux digitales, en général il est flexibles et bien adapté a l'informatique évolutive. Durant le processus d'évolution les stratégies sont évaluées dans le jeu Battle for Wesnoth, un jeu logiciel libre de nature stochastique. Une fonction d'aptitude est assignée aux stratégies afin de déterminer numériquement leur efficacité, de plus l'algorithme utilise des mesures robuste à la variance du processus. L'algorithme d'évolution recombine les stratégies efficaces utilisant des opérateurs évolutionnaires basés sur littérature par exemple, mutation ou transition. De plus amples expériences se concentrent sur différents paramètres et leurs utilités, par exemple les fonctions d'aptitudes.

L'algorithme utilise de multiples états initiales : stratégie nulle, stratégie issue du hasard et stratégies de conception humaine. Les stratégies sont ensuite testées dans le jeu. Il est démontré par la suite que les stratégies évoluées par l'algorithme sont plus efficace que leurs états initiaux.

Acknowledgments

I would like to thank all those who have provided me with support and inspiration over the years, especially my friends and colleagues.

I give special thanks to the Battle for Wesnoth community. They have made a fantastic game through hard-work and dedication, which provides an excellent domain for future research.

The University of Manitoba and McGill University have been sources of encouragement, hard work and passion for me. They have helped me find my path, for which I am always thankful.

At McGill, I would like to specially thank my supervisor Clark Verbrugge for his guidance and support throughout my Master's degree. My incredible office-mates and friends Ben Kybartas and Jonathan Tremblay have helped in so many ways, from interesting conversations to valued advice. I wish them all the best in the future.

My eternal gratitude goes out to my parents and brother for always being there to listen and to give me support in all I do. I will always be there for them.

Finally, I would like to thank Christina Victoria Cecilia Dwight, for being my companion these last few years; from everyday decisions to providing thesis motivation, she is a guiding light. I join others in finding her a smart, funny, and wonderfully cheerful person to be around. Everyday is an adventure with her, and nothing I can write here will ever be enough.

Contents

List of Tables

List of Figures

Chapter 1
Introduction

The complexity of entertainment software has risen along with an increase in hardware power over the last few decades. From graphics to simulated physics to artificial intelligence, all aspects of video game software have seen vast improvement. This has propelled the video game industry to reach millions of consumers, and has created a multi-billion-dollar market. The underlying technology for entertainment software has also found other markets such as in *serious games*, which are used for educational purposes such as training of law enforcement or military activities [35]. Therefore, advancements in entertainment software are of wide interest to a number of different fields and applications.

1.1 Artificial Intelligence

A main component of many video game experiences is that of competition of the player against an adversary or obstacle. For instance, a human player may be battling against an army of dozens of soldiers. In this instance, it would not be practical for every soldier to be controlled by another human being. Therefore, an *artificial intelligence* (AI) is developed to control these virtual soldiers. This control could be on a number of different levels, from an individual soldier's movement, to a squad formation, to an entire army's level of aggressiveness. For the purposes of this

work, artificial intelligence will be considered to drive the behaviours and actions of units on a team.

1.2 AI Impact

The artificial intelligence in a video game can control the behaviour of both the allies and enemies of the player. Therefore, its strength may positively or adversely affect the challenge in the game [61]. For example, a chess AI that does not match the player's skill level may make the player bored or frustrated while playing. The field of *difficulty adjustment* considers how the gameplay experience may be customized to the player's skill for greater enjoyment.

Considering this large impact on gameplay, artificial intelligence is an important part of game development. However, creating a strong artificial intelligence is not a trivial process. In order to make realistic decisions, a virtual soldier must handle changing game conditions, random events, and the existence of a finite computation time to make decisions in. These demanding attributes mean that high-performance AI is an interesting research topic, both within academia and industry.

1.3 AI Techniques

From examining different video game genres, it is clear that they require varied artificial intelligence strategies. The enemies of an arcade game may have a *finite-state machine* to move and chase the player. This simple representation can be sufficient for some games, such as for the simple ghost opponents in *Pacman.*[1] Other AIs may perform some measure of planning and management of resources, such as required in a *real-time strategy* (RTS) game. In the RTS *Warcraft* series,[2] the AI must coordinate building structures, training troops, researching technologies to unlock more powerful abilities, and ordering their units to attack. One way of accomplishing these complex sequential actions is through a ruleset or *case-based reasoning*, which

1. *Namco*, 1980
2. *Blizzard Entertainment*

matches game states to applicable strategies [48]. In board games such as chess, a standard technique is to simulate future moves [53]. All possible moves by the AI and their opponent are simulated for a number of turns forward, and each new board state is evaluated. The AI can then select moves that will lead to a victory, or at least a strong board state. First-person shooters are another highly-complicated domain. Artificial intelligence in these games may have to consider line-of-sight, direction of shooting, taking cover, and other issues in a real-time environment [61].

This heterogeneity of domains means that AI development for one game may not be transferable to another. Developers thus spend a large amount of time building and customizing an AI to fit a new game [59]. While research on re-using AI components between games has been performed, this is not yet a mature field [11]. Even within a game's development, new game mechanics or play-testing feedback may require developer to modify the AI. Therefore, it is valuable to investigate ways to improve this development cycle.

1.4 Problem Statement

Machine learning has been used to develop game-playing AIs with great success. In one case, a *neural net* was trained to play BackGammon [58]. It achieved such a high level of performance that experts began using its moves in tournament-level play. Evolutionary algorithms have also shown strong performance among a number of problems, sometimes producing a solution better than human experts [29].

This work aims to use techniques from machine learning, specifically *evolutionary computing*, in order to improve AI performance through an unsupervised learning process. An evolutionary computing algorithm iteratively evaluates solutions to a problem, and combines the best-performing solutions together. For the studied problem, a solution is a strategy for a team of units that controls their actions. This strategy will be encoded in the *behaviour tree* formalism which has been recently used in both industry and academia, due to being an expressive and human-readable

representation of AI. The behaviour tree formalism, as well as modifications made to suit this domain, are discussed later in this work.

The evolutionary process, as described later, will attempt to evolve strong strategies from a number of initial states, as scored by a robust performance metric. Later results will show limited success as some evolved strategies achieve a stronger performance than numerous hand-built solutions.

While research has been done in the evolution of AI strategies, this work considers the domain of turn-based games. This domain allows for the selection of a wide variety of fitness functions, while the discrete nature of the turn-based game allows this paper to clearly demonstrate the difficulties and successes of the evolutionary algorithm.

The problem statement is thus:

How can evolutionary computing be applied to behaviour trees to evolve novel artificial intelligence strategies for a selected turn-based game? What are potential areas of improvement in this process?

1.5 Thesis Contributions

The main contribution of this work is a fully functional evolutionary computing algorithm that can automatically create and evaluate AI strategies for a turn-based game. This game will be Battle for Wesnoth (abbr. Wesnoth), with the caveat that a few game mechanics have been removed. The particulars are discussed in Chapter 5. As it is turn-based, Wesnoth is somewhat similar to chess. However, an artificial intelligence may not be able to use a look-ahead search as is common in chess AIs. This is due to battles in Wesnoth having a highly random nature, as well as the number of possible movements for each unit. Due to these complications, another artificial intelligence approach is required. This paper proposes the use of behaviour trees in order to represent team strategies.

The algorithm used to produce these strategies must handle the interesting complexity arising from the element of randomness present in Wesnoth. In particular,

this work defines the use of a performance metric for strategies that is less sensitive to the randomness present in the domain. Other contributions include results and conclusions on the selection and use of various *fitness functions* used to evolve strategies. A fitness function is a metric for determining how well each strategy performs in a battle.

The last major contribution of this work is a detailed discussion of the evolutionary process and the performance of the created strategies. In-game actions of the evolved strategies are examined, along with a critical discussion of the generality of these results. Later discussions also focus on performance improvements that could be made to this process, as well as technical details useful to replicate this work.

1.6 Thesis Organization

The next chapter of this paper will provide background on evolutionary computing, as well as present the formalism of behaviour trees and their use in the video game industry. Following this, Chapter 4 will take a comprehensive look at genetic algorithms in the literature, as well as some problems they are well-equipped to solve. Chapter 5 will provide background on the video game selected for this work, named Battle for Wesnoth. A brief overview will be provided, along with a short discussion on the complexity of this domain. Chapter 6 describes the methodology of this work, the components of the evolutionary computing algorithm, and the particulars of how the algorithm integrates with Battle for Wesnoth. The results of the evolutionary process are then shown in Chapter 7, together with a study of the strategies evolved. Finally, a detailed conclusion of the work will be presented along with avenues of future work in Chapter 8.

Chapter 2
Evolutionary Computing

The field of *evolutionary computing* is built on concepts from evolutionary thinking in biology as applied to problems in computing. Other names for this field include *evolutionary programming* or the field of *genetic algorithms*. This chapter will describe the basic concepts of evolutionary computing and provide a reference example of a well-known computing problem. Related work concerning evolutionary computing will be discussed in a later chapter.

2.1 Biological Inspiration

In the biological sciences, the reproduction of an individual is key to the success of that individual's species. This reproduction capability or even the survival of the individual depends on different genes with the individual's genome. For example, genes may have subtle or overt effects on the strength of the immune system or the ability to find food through smell. These differences may arise through mutations in the genetic code or through sexual recombination of the parent's genetic code. Individuals in biology can be seen as solutions for the problem of survival and reproduction. Successful individuals and species are those that can survive in their environment and reproduce. Their genetic code will be passed on to the next generation with some modifications and recombinations. Due to *natural selection*, the

beneficial genes will therefore spread throughout a population, while the deleterious genes will be less prevalent.

2.1.1 Adaptation to Computing

These concepts of evolution and natural selection were formalized by John Holland as the field of evolutionary computation, in which a simple algorithmic process can be employed to automatically produce solutions for a specific problem [21]. However, there may be difficulty in adapting the evolutionary algorithm to the problem domain. A number of elements in the algorithm must be formally defined to suit the domain, such as the problem representation, the evaluation of solutions, and solution modification.

Pseudo-code for a evolutionary algorithm can be seen in Algorithm 1. It will iteratively search a number of points in the solution space, starting from a *population* initialized in line 1. The parameters on lines 5 to 7 control when execution should cease, and are evaluated in the main loop on line 8. The algorithm may be set to stop when a computing time limit is reached. This time limit can be updated after each generation, as shown on line 15. Another criteria may be to run the algorithm for a certain number of iterations, labelled *generations*. This is seen in the G_{curr} parameter, as updated on line 16. Finally, the algorithm may stop when a solution's *fitness* reaches a *fitness target*, which means that a solution has been found of sufficient quality. For each generation, the population is evaluated using a *fitness function*, as on line 9. This gives a numerical *fitness value*, which is a rating of how well each solution solves the stated problem.

Lines 10 to 14 show how a new generation Q is created. Solutions in P with a high fitness value are combined together by genetic operators to create a new set of points in the solution space, denoted Q. Then, P is set to Q and the algorithm continues. Over a number of generations, these points P will settle into areas of the solution space representing strong solutions to the problem.

```
1: P.initialize()
2: P_fitness ← 0
3: G_curr ← 0
4: T_taken ← 0
5: Set fitness target F
6: Set generation limit G
7: Set time limit T
8: while (P_fitness < F) and (G_curr < G) and (T_taken < T) do
9:     P_fitness ← P.evaluate()
10:    Q = ∅
11:    Q.addTreesSelectedFrom(P)
12:    Q.addTreesRecombinedFrom(P)
13:    Q.mutateTrees()
14:    P ← Q
15:    T_taken ← T_taken+ deltaT
16:    G_curr ← G_curr + 1
17: end while
```

Algorithm 1: Evolutionary algorithm pseudo-code

This pseudo-code provides the structure for the following concrete example of evolutionary computing, based on a well-known problem and the works of Goldberg, Holland et al. [15] [21]. Discussion of each point will lead to examination of extension into more difficult problems. A following chapter will present a genetic algorithm for the studied problem domain.

2.2 Sample Problem

The sample problem to be discussed will be the Travelling Salesman Problem, which attempts to find the shortest cycle that will visit all points in a list. This problem's name is modelled on an imagined travelling salesman who must travel the minimum distance to a list of different cities before returning to his home city.

For example, let there be eight cities labelled A to H randomly placed on a two-dimensional plane as in Figure 2.1. Let all cities have a viable connection between them of Euclidean distance.

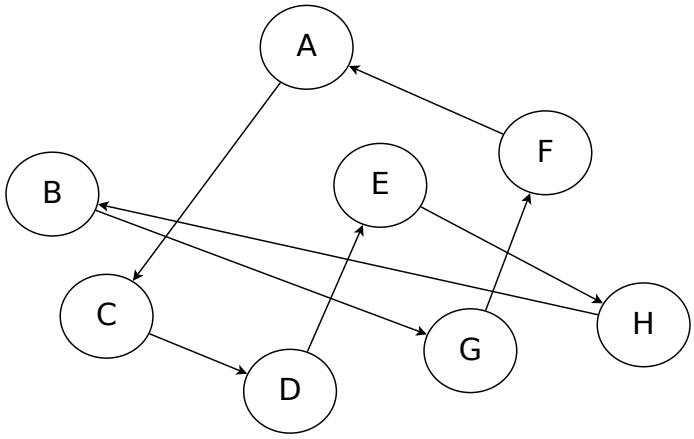


Figure 2.1: A non-optimal cycle in the Travelling Salesman Problem

2.2.1 Complexity

The number of possible tours for an n-city Travelling Salesman Problem example is $(n-1)!/2$. This is calculating by picking an arbitrary city as the first. There are then $n-1$ cities to choose for the second to visit, $n-2$ for the third, and so on. The division by two occurs because it does not matter in which direction the cycle is performed. This factorial search space may be computationally prohibitive for a brute-force search, and hill-climbing techniques may also fail as the search space is non-continuous. However, genetic algorithms can be applied with success to this NP-complete problem [31]. In particular, Banzhaf shows an implementation that produces a solution on a problem with 100 cities that approaches 10% of the global optimum [6]. While the solution given is not optimal, genetic algorithm solutions may approach optimality over a larger number of generations or at least provide an acceptable solution in an extremely high-dimensional search space.

2.3 Representation

A *representation* of a solution to the problem creates the search space that the genetic algorithm will attempt to optimize over. According to Wall [62]:

> For any genetic algorithm, the representation should be a minimal, complete expression of a solution to the problem. A minimal representation contains only the information needed to represent a solution to the problem. A complete representation contains enough information to represent any solution to the problem. If a representation contains more information than is needed to uniquely identify solutions to the problem, the search space will be larger than necessary.

In the sample problem, a solution is defined as a cycle of cities to be visited. For example, one possible cycle may be A, C, D, E, H, B, G, F as in Figure 2.1. This would represent the salesman visiting city A, travelling to city C and so on until city F. Upon leaving city F he would return to city A, completing the cycle. It can be seen that this representation is complete as it can represent any possible cycle. The search space is also limited to the ordering of cities which is the only consideration for this problem.

An effective representation of the solution therefore depends on the domain of the problem, and must be carefully constructed. The representation of solutions directly affects other steps in the algorithm such as evaluation and recombination, as will be shown.

2.4 Population Initialization

Once the solution representation has been decided, the next step is to create an *initial population* of solutions, as on line 1 of the above pseudo-code. As these individuals are points in the solution search space, one approach is to randomly generate them, hopefully covering the space. If hand-built or previously-evolved solutions are available, these may be copied or *seeded* to be the initial population. This is

done in order to guide the search to better-performing solutions. However, as with other randomized optimization algorithms, local maxima may be encountered. This is dependent on the problem domain as well as the random nature of the algorithm.

The choice of *population size* is not a trivial one. Various works have been done to select an appropriate fitness size based on the problem domain and the nature of modifications to the individual [10]. The literature suggests selecting the largest population size that is computationally feasible, as more points will be searched at once [22]. However, as each point takes some time to be evaluated, the time per generation may increase prohibitively.

2.5 Evaluation

The individuals in the population must now be evaluated in order to measure how well they solve the problem. This step in the pseudo-code is line 9. This metric is called the *fitness* or *objective function*. It is a heuristic to discriminate between solutions that solve the problem poorly and those that perform well. This is usually done by forming a maximization or minimization problem, and assigning a real-valued number to each solution.

The objective in the Travelling Salesman Problem is to find a cycle of minimal distance. Let the fitness function $fit(x)$ operate on an individual and also define a minimization problem. Let $fit(x) = dist(C_1, C_2) + dist(C_2, C_3) + \cdots + dist(C_{N-1}, C_N) + dist(C_N, C_1)$, where C_n is the last city in the cycle and N is the number of cities in the cycle. Again, cities are assumed to be fully connected, and the distance is the Euclidean distance between them.

This fitness function is an accurate heuristic to the problem. An x that is a short cycle will be assigned a small distance score by $fit(x)$. For other problems, a formal measurement of the fitness of one solution may be hard or impossible to define, such as defining a 'fit' painting or song. There may also be an issue with optimizing several fitness criteria at once. This problem is studied in game theory as the concept of *Nash equilibria*, and it has been applied to evolutionary computing [45]. In this work

by Périaux et al. multiple criteria are selected in turn as the fitness function for the solution. This continues until there is no improvement under any criteria, in which case the solution has then converged to an optima.

2.6 Creating a New Generation

At this point in the evolutionary algorithm, an initial population of solutions have been created and those solutions have been quantitatively measured. Now the algorithm can use this fitness value to generate new and potentially more fit individuals from the current population. This is seen in the pseudo-code on lines 10 to 14. It should be noted that this new generation is not required to be the same size as the previous generation, and that a variable population size over generations may be beneficial [22]. This work will only consider a fixed population size.

This new generation will be created by *selection*, in which individuals are copied from the current generation, and *crossover*, where two parent individuals are combined to produce a child solution. A *mutation* operator will also act on some individuals in the new generation to add more variation to the population.

2.6.1 Selection

A *selection scheme* aims to ensure that very fit individuals can be chosen, either to be directly copied into the next generation or as parents for the crossover operation. By preferentially selecting fit individuals, the new generation will contain individuals that are equal or fitter than the previous generation. However, it may also be important for low-fitness individuals to propagate as well, as they may contain elements of the optimal solution. Thus, diversity in the population is beneficial. Line 11 refers to the selection process.

In *elite selection*, a small number of the most fit individuals in a generation are copied forward into the next generation. This ensures that the best solutions are retained from generation to generation.

A *roulette or weighted method* is biased towards selected individuals that are more fit. The individuals are weighted based on their fitnesses' proportion of the

population's summed fitness, or in the case of a minimization problem, the reciprocal fitness. This scheme is beneficial as it preserves diversity since low-fitness individuals have a small random chance of being selected for propagation.

In *tournament selection*, there are some number n of tournament rounds. In each round, two candidate individuals are randomly picked. The individual with the higher fitness score is declared the winner of the round and will appear in the next round. With a higher n, it is more likely that a higher scoring fitness individual will be chosen. There is also a random chance that low scoring individuals may be chosen if tournaments happen to occur between less-fit individuals. As with the roulette method, this is beneficial for diversity.

2.6.2 Recombination

Along with copying of individuals to the new generation, other operators combine solutions together. One method is known as *crossover* and involves taking elements of two individuals and combining them to create new children individuals. This is referred to in the pseudo-code algorithm above on line 12

For the Travelling Salesman Problem, the representation of a solution is a list of cities. Assume that two parent individuals have been selected using the methods above. One crossover operator will be described here that combines the order of cities of the parents to create the child. This is done by taking n cities in one parent's cycle and the other $N - n$ cities from the other cycle. This must be done without duplication of cities in order to have a valid solution, as in Figure 2.2. A search in the literature will find multiple operations of higher complexity [31].

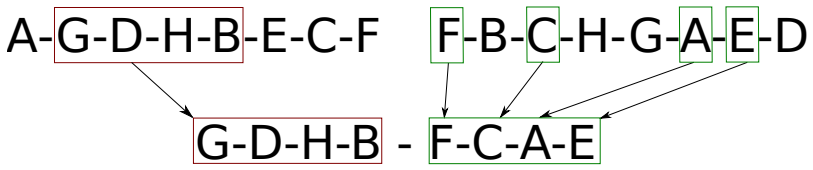


Figure 2.2: One crossover operation for TSP

The point of recombination is to take two existing solutions and select elements from both. Therefore the success of this operation heavily depends upon the representation and the operators used. Early work in genetic algorithms heavily focused on how operators should combine solutions. In particular, the study of *schemata* identifies structural elements of the solution that are important to the solution's fitness [15].

2.6.3 Modification

Modification or *mutation operators* are also typically used in order to introduce further variation into the population, allowing the population to search new points in the search space. Line 13 denotes this in the pseudo-code algorithm. For the representation above, the mutation operator could simply exchange the position of two cities in a cycle. Another operator could randomly pick a city and move it to another point in the cycle.

For other representations this mutation operator may be quite complex. For example, representations that include real-values variables may be modified by a Gaussian deviation. Tree representations like those in this work may also have structure-changing operations, as will be discussed in a future chapter.

2.6.4 Repairing Solutions

The crossover and mutation operations above may drastically modify a solution to the point where the solution is invalid in some way. For example, the crossover operation specified above was created to prevent cities from appearing more than once, as this would be an invalid cycle. An invalid solution is not desirable as computation will be wasted on evaluation. To resolve this issue, the modification operators may be designed in such a way that invalid solutions cannot be produced, or a specialized *repair operation* may be defined to examine solutions and correct them.

2.7 Iteration

Once a new generation has been created by the evolutionary algorithm, those individuals are then evaluated and assigned a fitness value. In turn, yet another generation of individuals are produced through the selection and modification operators, and the algorithm iterates. As mentioned, the stop criteria for this algorithm could be the algorithm running for a specified amount of time or number of generations. A more advanced criteria might consider the fitness of the population. If the average or maximum fitness of the population has not changed in a number of generations, the fitness improvement may have plateaued. This could indicate that the process should be restarted to attempt to find another optima.

It is important to note that the population of the last generation when the evolutionary algorithm stops may not contain the highest-performing individuals. The evolutionary process may have randomly discarded these individuals. For this reason, it is advisable to explicitly record the highest-performing individuals seen throughout the evolution process. These recorded individuals will be the final output of the evolutionary computing algorithm.

Chapter 3
Behaviour Trees

This chapter will introduce behaviour trees as the formalism used to represent artificial intelligence strategies in this work. A brief discussion will also highlight the known use of behaviour trees in the video game industry. A later chapter will discuss the specifics of how behaviour trees were evaluated and modified during the evolutionary process.

3.1 Overview

Behaviour trees have been gaining prominence in the game development industry as a representation and encapsulation of the behaviour of an artificial intelligence [8]. This formalism is similar to a hierarchical state-machine, but offers superior reusability and modularity [39]. Actions in a behaviour tree can be at any level of abstraction, which allows high-level concepts to be expressed in a human-readable manner. This readability attribute allows behaviour trees to be created and modified by non-programmers on a game development team, which can provide improved flexibility and speed in the AI design process [7].

Behaviour trees are queried by a character to order to determine the action to perform. This query flows through various nodes from the root of the tree to the leaves in a post-order depth-first search with pruning. The internal nodes of the tree

control the query flow depending on values returned from their children. The leaf nodes of the tree are usually either *condition nodes* to query the state of the world, or *action nodes* which specify what action the character should perform. An example behaviour tree is shown in Figure 3.1. It encodes the behaviour for a character to open a door and will be discussed in more detail below.

3.2 Node Definition

As with other tree structures, behaviour tree nodes have parent and child nodes. However, when queried these nodes will return a status value. In a simplified model as used in this work, these return values will be binary true/false values indicating whether the node successfully completed its task or not. In a environment where actions may take a number of time-steps to complete, nodes may be given further return values to indicate they are currently active [27].

3.2.1 Composite nodes

Composite nodes impose control on the query flow within the tree. The query will move down different subtrees depending on the type and attribute of the composite node, as well as the values returned by the composite node's children nodes. The two simplest composite nodes are the **Selector** node and the **Sequence** node, both of which are seen in Figure 3.1. One convention is also to represent these nodes as a question mark and arrow respectively.

The **Selector** node is used to pick a single subtree to query out of its children. The **Selector** node queries all of its children nodes in a particular order, usually depicted as left-to-right on a diagram. The **Selector** node will immediately return a true value if any child returns true, otherwise the next child will be queried. The **Selector** node will return false only if all children are false.

Like the **Selector** node, children of a **Sequence** node are evaluated in a deterministic order. If any child returns false, the **Sequence** immediately returns false. If all children return true, the **Sequence** returns true. The **Sequence** node is therefore used to represent a number of actions that must take place in a particular order.

In Figure 3.1, a behaviour tree encodes the unit's strategy to enter a room. In this example, the *DoorLocked* node returns a true/false value. If this value is true, then the **Sequence** node will query the next child, and the unit will perform the *UnlockDoor* action. Otherwise, if *DoorLocked* returns false, the **Selector** node will query the right-hand subtree, so the unit will perform the *EnterRoom* action.

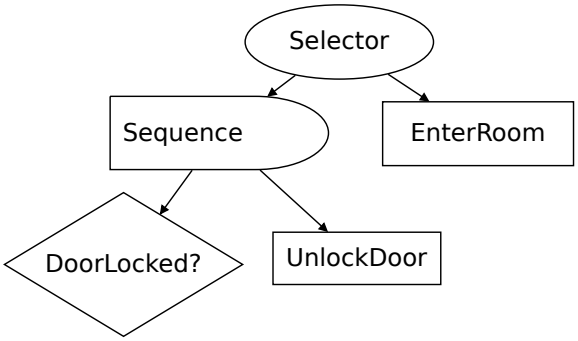


Figure 3.1: Behaviour tree for opening a door

These composite nodes can be modified to offer further flexibility to the AI designer. For example, the order of child querying could be randomized which would make the **Selector** node useful for randomly selecting an action to take. A further modification would be to ensure that all actions have been selected before any repetitions. Temporary adjustments of child order are also possible by adding dynamic priority to children [20].

3.2.2 Non-composite Nodes

Non-composite nodes could be any form of condition node or action node. As the name implies, the condition node returns a true/false answer to a specified test. In combination with the composite nodes, the behaviour tree can then control the query flow through the tree. For example, Figure 3.1 shows a condition node *DoorLocked* that checks if a door in the game world is locked. Complicated tests may be abstracted into a single node, allowing non-programmers to utilize them effectively.

Action nodes in a tree are highly domain-specific and may represent a wide range of actions related to a character or to the game state. For example, an action may be a command for the character to fire a gun, move towards a goal, or play a particular animation or sound. The abstraction provided by these nodes allows for a variety of related tasks to be handled by the same decision structure, in a human-readable manner. Action nodes for the studied domain will be presented in a future chapter.

Decorator Nodes

Decorator nodes are single-child internal nodes that modify the behaviour the behaviour of their child node. This increases the readability and expressiveness of behaviour trees, similar to the well-known *decorator pattern* in programming. For example, a **Negator** node may return the negation of its child's value, while a **Repeater** node could repeatedly query its children until the child returns true, or for some number of iterations. Decorator nodes allow complicated processing to occur while still respecting the behaviour tree formalism.

3.3 Use in Industry and Research

Behaviour trees have been used in a number of high-profile commercial video games [7]. They are cited for their human-readability and high degree of flexibility [24]. Although there are few formal industry and research publications on the technique, there are a growing number of conference slides and short electronic articles. In order to provide a resource for future research, this section will highlight examples of behaviour tree use and innovations within the industry and academia.

For the game *Destroy All Humans*,[1] the behaviour tree formalism above was modified to create a highly modular or 'puzzle-piece' system with great flexibility and ease-of-use [30]. This was accomplished through the ability of parent nodes to pass down arbitrary objects to their children. While this does raise the complexity of the nodes, the characters could then act intelligently in a wider variety of situations, reducing the amount of specialized code.

1. *Pandemic Studios*, 2005

In the first-person shooter game *Crysis*,[2] the artificial intelligence was required to coordinate multiple teammates in a flanking manoeuvre [47]. This was achieved by creating a two-phase process. In the first phase, when a unit queries the **Flanking** action node the unit is marked as able to join the flanking manoeuvre. However, the unit does not begin the action yet. When enough units have signalled they are ready to begin the manoeuvre, then the second phase begins. Now when the **Flanking** action node is queried again, the unit will begin the flanking manoeuvre. This two-phase process prevents the manoeuvre from overriding high-priority actions a unit should perform like dodging a grenade, while still allowing effective coordination between teammates.

Behaviour trees can also be used in *case-based reasoning*, where a database is created with game states correlated to the behaviour trees applicable to those states. This allows the artificial intelligence to perform a look-up of effective strategies with specialized knowledge about the current game conditions [14]. Behaviour trees in particular allow effective similarity metrics to find the appropriate case in the database, as well as facilitating dynamic combination of strategies [43].

A recent implementation of behaviour trees has been in the driving game *Driver: San Francisco*[3] [42]. In this work a metric for the player's gameplay skill classified players into various difficulty levels. This difficulty level was used to control which roads the computer-controlled cars drove on during chase sequences. The intention was for AI cars to turn less often or onto wider roads if players had less skill. The road selection was implemented in a behaviour tree which could be dynamically rearranged or modified based on hints given by the difficulty level.

This dynamic adjustment of behaviours trees has also been studied in the domain of third-player shooter games. In this domain, the player may regard certain combat situations as being overly intense, which is detrimental to their gameplay experience. Artificial intelligence companions of the player may fight alongside them, affecting their combat experience and therefore the *combat intensity*. By dynamically

2. *Crytek Frankfurt*, 2007
3. *Ubisoft Reflections*, 2011

switching between different behaviour trees based on a metric of this intensity, the companion may be able to reduce the combat intensity on the player [61].

Chapter 4
Related Work

This chapter will discuss how evolutionary algorithms have been employed to generate solutions to a variety of problems. Four sample problems from the field of computer science will be examined along with a discussion of evolving evaluation functions for turn-based games. A brief look at evolution of video game strategies, mechanics, and content will follow. The second section of this chapter will discuss the field of genetic programming, which attempts to evolve a program or equation as a solution to a problem. Finally, literature will be presented on the evolution of behaviour trees as in this work.

4.1 Evolutionary Computing

Evolutionary computing has found success in a variety of fields, demonstrating its flexibility and power. The works below are presented as a selection of differing problem environments and various domain-specific modifications to the evolutionary algorithm.

One early application of evolutionary algorithms has been to modify the morphology of virtual creatures [54]. Three-dimensional creatures exist in a full three-dimensional world with simulated physics interactions. Creatures have a graph structure to represent their limbs along with a simulated brain consisting of various neural

nodes and sensors to control their limbs. The objective is for the creature to evolve a strategy in a competitive setting with another creature. A creature's goal in the competition is to reduce its distance to a cube, while keeping the other creature as far away from the cube as possible. Both the neural workings and the morphology of the creatures are modified in the evolutionary algorithm. Over the course of their evolution, creatures developed various strategies such as fully encasing the cube, knocking the cube towards its other arm to carry, or simply knocking the cube away from their opponent. [1]

An informative application of evolutionary algorithms has been to resource-constrained job scheduling, another NP-hard problem [62]. A solution for this problem must consider dependencies between jobs, the length a job takes, and jobs which can only be performed on a particular machine or by a trained operator. The fitness function is the time required for a particular sequence of jobs to complete, and is to be minimized in the optimal solution. Evolutionary algorithms are able to produce optimal or near optimal solutions to a general version of the job shop problem [46]. Their results were comparable to the best known algorithm at the time.

One domain studied with evolutionary algorithms has been placing base stations for wireless network planning in two-dimensional space [23]. In this domain, *base stations* must be placed in candidate locations to serve wireless functionality to a multitude of *subscriber stations* in the space. The objective is to minimize the number of base stations used to serve the subscriber stations while also respecting the maximum bandwidth capacity of each base station. The recombination operators of the algorithm were carefully designed to force the base stations to either carry few or many subscribers, potentially reducing the number of base stations. The authors consider their algorithm to be effective, since in their experiments only half of the candidate locations were selected to be used.

1. BoxCar 2D is a modern real-time adaptation. Available online:
`http://www.boxcar2d.com/index.html`

Evolutionary algorithms can also be used to dynamically evolve bidding strategies [51]. In this case, strategies encoded the amount of power that can be provided by a energy provider and the price to charge for this amount. Therefore strategies must balance the revenue gained by a high price against losing the bids to a competitor. As the algorithm continued, the authors noticed that particular strategies adapted to the changed influence of all other strategies and performed well. Therefore, studying these strategies may provide future economic or bidding insights. Similarly, studying the evolved strategies used in a turn-based game may provide insight into the underlying strategy space.

4.1.1 Game State Evaluation

An artificial intelligence for board games typically examines future board states for some number of moves ahead in order to select the strongest move to perform. For chess an *evaluation function* may take a weighted count of various pieces in order to give a numerical evaluation of a board state [53]. For example, a pawn may be worth one while a queen is worth nine. Historically, piece weights have been selected by human experts. However, recent work has shown that evolutionary algorithms can optimize these weights.

Kendall and Whitwell creatively use the standard deviation of piece weights in their optimization of a chess board evaluation function [26]. A particular piece weight is examined across all individuals in the population, and the standard deviation of these values is recorded. When mutation is applied to a piece weight in one individual, the size of the standard deviation across the population is correlated to the change of this weight. This correlation aids in the convergence of weight values in the population, which may indicate when optima have been reached. This is similar to the concept of *simulated annealing*, where the rate of change in an optimizing process will steadily decrease as the search process continues.

This notion of an evaluation function can be extended further in order to take game strategy into account. For example, the game of four-sided dominoes is played with four players divided in two teams. Each player has imperfect information and

must also consider how their actions may affect the actions of their teammates and opponents [4]. This prediction can be encoded into the evaluation function, where weights are then optimized by a genetic algorithm.

When an evaluation function is tuned by a evolutionary algorithm, it may become over-specialized to specific opponents. Sun et al. combat this by using multiple strong-playing computer programs as opponents in their Othello board game domain [57]. Their results showed that the use of these multiple "coaches" was more effective than with a single opponent.

In the Google Artificial Intelligence Competition, the game *PlanetWars* places AIs in control of fleets of spaceships fighting over a number of planets. The only action available to AIs is to move units from one planet to another, and AIs cannot store any game information from turn to turn. Research shows that a evolutionary algorithm can create a strategy to play this game using a simple evaluation function to select planets for fleet movement [12]. Metrics were devised to compare the sample competition AI with the two AIs in the paper. The first AI had parameters given by the authors, while the second had these values tuned by the evolutionary algorithm. These comparison metrics were the number of games won and the turn length of the games. The turn length consideration was chosen as a proxy to suggest that a stronger AI would be able to win in less turns than a weaker AI. Further work by these authors attempted to minimize the stochastic effects on the evaluation of the parameters [40]. This was done by assigning a score to an AI based on the results of five runs on five different maps, instead of one battle on one map as in the original algorithm. Stochastic effects similar to the *PlanetWars* environment can also be seen in Battle for Wesnoth, as will be discussed in a future chapter.

4.1.2 Strategies

The game of Lemmings tasks players with ensuring that some number of Lemming characters traverse a two-dimensional level from a start position to a given exit point. To overcome obstacles and successfully complete the level, Lemmings are assigned roles at particular times and locations. For example, roles include a "bomber" role to

blast a hole in their immediate surroundings, or a "climber" role which allows them to ascend vertical walls. In their work, Kendall and Spoerer used a evolutionary algorithm to generate a sequence of action triples to guide Lemmings towards the exit [25]. These triples contain three identifiers representing the time-step to assign a role on, the Lemming to receive the role, and the role itself. These sequences were initially seeded randomly and with a null strategy. In an interesting observation, the authors state that it may be preferable to start with a null strategy instead of the randomized strategy. This is because a successful strategy may only require a few actions at specific time-steps. As a small strategy is closer in solution space from a null strategy compared to a randomized strategy, the correct strategy may therefore require less time to evolve. In the work we present here, behaviour trees will be evolved from a null strategy as well.

The NeuroEvolving Robotic Operatives (NERO) system allows players to evolve the neural networks of agents to learn battle tactics in a real-time strategy game [56]. The player provides dynamic weighting to the evolutionary algorithm determining which qualities are important, such as avoiding being hit or damage done to enemies. The system then periodically removes the worst performing agent and replaces it with a new agent created through recombination. Over a period of time, agents evolve strategies such as navigating through a maze and firing their weapons at an attacking enemy while moving backwards. This system is quite interesting in that the evolution of new behaviours takes only minutes. Agents can therefore dynamically evolve to meet player objectives in relative real-time.

4.1.3 Games

The ANGELINA system attempts to create arcade-type games through the evolution of game maps, object placement, and game mechanics [9]. These games are then evaluated with different fitness functions for the different components, such as a metric for map sparseness or the distribution of items to pickup or avoid. The games are then themselves evaluated by being played with simulated players, such as a null player, a randomly moving player, and one that uses a planning algorithm

to maximize score. The authors present two evolved collecting games, which are loosely based on existing game genres. The first is one where the player must avoid wandering enemies and collect items for points as in the arcade game *Pacman*. The second is a "steady-hand" game, where the player must avoid the walls and collect items on a more maze-like map.

Evolutionary computing has also been used within game design. Tozour discusses a process where games are repeatedly played by evolving AIs [60]. The game designer can then periodically examine the evolved AIs, and determine if particular units or strategies are overpowered. Hastings et al. use evolutionary processes in order to procedurally generate weapons for their game [17]. Weapons within the game fire projectiles in a pattern determined by an internal neural-net-like structure. Whenever a player uses that weapon to kill an enemy, the fitness value of the weapon increases. A new weapon is dropped when an enemy is killed. This new weapon is an evolved version of weapons used by the player that have a high fitness value. In this way, players can enjoy infinitely new content, where each new weapon is similar to their established preferences.

4.2 Genetic Programming

In genetic programming, a solution to the problem is encoded into a tree-based structure, which is acted upon by evolutionary operators. For example, a problem may be to build a mathematical expression that calculates π. One representation of this problem would be an *expression tree*. In this representation, constant or variable input nodes would be at the leaves, and mathematical operators such as multiply, divide, or trigonometry functions would be placed in internal nodes. Figure 4.1 shows one such tree which represents the expression $(2 + 10) - (\cos 45 * (45 * 7))$.

During evolution, potential evolutionary operators could recombine the tree by switching subtrees or modifying nodes. Koza's comprehensive work considers problems in a number of domains and provides complete frameworks [28]. These frameworks include the recombination operators, fitness functions, and the domain-appropriate

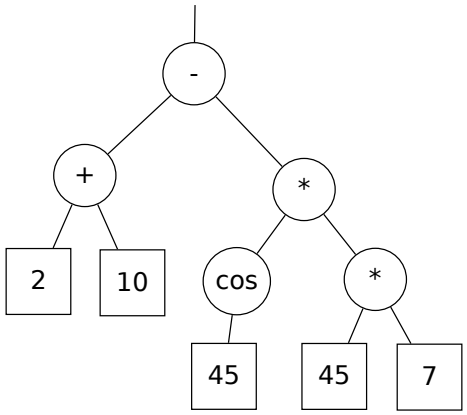


Figure 4.1: A binary expression tree

internal and leaf nodes, denoted as *function nodes* and *terminal symbols* respectively. Our work aims to create a similar framework for the Battle for Wesnoth strategy domain.

As described in an earlier section, many chess-playing programs use a look-forward search to evaluate possible moves. Hauptmann and Sipper employ genetic programming in order to evaluate board positions in the end-game of chess where few pieces remain on the board [18]. The authors make note of the fact that genetic programming allows human knowledge to be represented easily in the functions and terminals used. The AI using the evolved evaluation trees was able to draw against a state-of-the-art chess engine. Later work studied the trees created, as well as identified terminals that were crucial to performance [19]. Behaviour trees were selected for the present work for their similar readability potential. This will be demonstrated in the results chapter where the evolved strategies will be studied.

Gross et al. use genetic programming to evolve weights to determine which board positions should be expanded as well as the weights used to evaluate a board position [16]. A interesting innovation of theirs was to also evolve a program to determine how many turns forward this search should be performed. Their results showed that their evolved programs expanded only six percent of the states that an

Alpha-Beta search would expand. This is a remarkable reduction in computational resources, especially since the algorithm still performed with comparable play quality to Alpha-Beta search.

A real-time strategy game may use weighted map overlays named *influence maps* in order to weight areas of the map for the AI to attack or avoid. One way of creating these maps is through the use of strategy trees to combine circles of various radii and weights. Miles and Louis use this technique to evolve and study AI strategies in a simple game where there are two attackers and one defender of a target [38]. The strategies of both the attackers and the defender were simultaneously evolved. This allowed effective strategies to be countered by the opposing side within a few generations. The authors also note that the use of one tree for each attacker may not allow for team strategies to develop and that a single tree for both attackers may produce a superior result. This idea of one tree for the entire side is present in our work.

A variety of techniques can be employed to create intelligent characters ("bots") for first-person shooters. One approach uses a process very similar to case-based reasoning, where a database of rules are stored linking game conditions with actions that the bot should perform [50]. In this approach, when the bot is to take an action, the area around a bot is first quantized into a grid. As conditions in the database are also grids, the bot's surroundings can be matched to a database row, and the corresponding action can be taken. The actions a bot takes are evaluated by a weighted difference between the damage done to the bot and the damage done to the bot's opponents during the game. The evolutionary algorithm then optimized both the conditions and actions in the database, and produced bots that could defeat the standard game bots by a large margin of health points. Further results showed that only a few rules were integral to the bot's performance, allowing precise insight into the bot's behaviour.

Case-based reasoning has also been used in real-time strategy games. In these domains, conditions in the game world may trigger orders to build buildings, order

troops, or research unit upgrades. These rules are generated by hand or by human players and are placed into the database to be used by the AI. An evolutionary process can then optimize these rules and provide an even stronger knowledge base [49].

A team strategy was submitted to the simulated soccer tournament RoboCup-97 that was created using genetic programming [36]. A soccer game is a highly complicated domain, yet the terminals used were relatively simple and mostly consisted of vectors to/from opponents and the ball. The authors present strategies partway through their evolution, which offers interesting insights into the evolutionary process. An early strategy evolved was so-called "kiddie soccer" in which every teammate tried to head for the ball and kick it. Eventually, the strategy began leaving teammates dispersed throughout the space in order to defend well. The study of strategies from various generations in order to obtain domain knowledge will also be seen in the present work.

4.2.1 Evolving Behaviour Trees

As discussed above, behaviour trees are a tree-based structure with conditions, composite nodes, and actions. Similar to genetic programming trees, behaviour trees also have internal nodes which dictate the flow of execution, as well as action nodes to provide control or sensing information. Thus, behaviour trees are applicable to be evolved using genetic programming techniques as in this work.

An application of evolving behaviour trees is in direct character control [44]. In this work, the behaviour tree directs a character to run and jump throughout a level based on the platforming genre of video games. This character control is achieved by creating a map of the world to query through condition nodes, as well as action nodes which mimic a human player's button presses. A syntax grammar was created with predefined patterns in order to regulate the subtrees created during evolution and avoid invalid trees. The authors make special mention of the fact that the resulting behaviour trees are human-readable, allowing for detailed examination as desired by the video game industry.

In the real-time strategy game *DEFCON*,[2] the player must co-ordinate the placement of missile silos, radars, airbases, and fleets in order to wage global thermonuclear war against their opponents. Lim et al. employed genetic programming on behaviour trees in order to optimize the placement of these offensive and defensive elements [32] [33]. This was achieved by using multiple behaviour trees for various sub-goals of the AI. For example, behaviour trees were created that controlled the placement of radar towers. These towers detect enemy forces within a particular radius, which makes their placement strategically important. The fitness function of the evolutionary algorithm scored each behaviour tree based on how many enemies were detected. Evolutionary operators similar to that in this work could then evolve behaviour trees that detected an increasing number of enemies over the evolutionary run. By combining a number of optimized behaviour trees for different sub-goals, the authors produced a strategy that could defeat the built-in artificial intelligence in fifty-five percent of battles.

2. *Introversion Software*, 2007

Chapter 5
Problem Domain

This chapter will discuss the problem domain, which is a simplified version of Battle for Wesnoth (abbr. Wesnoth), an open-source turn-based game. A brief overview of Wesnoth will be provided, along with the historical context of the war game genre. Special highlight will be given to the difficulties and complexity of this domain which impact the effectiveness of the genetic algorithm. In particular, the randomness of the domain and the removal of the recruitment aspect of Wesnoth will be major issues raised in this chapter.

5.1 Battle for Wesnoth

Battle for Wesnoth is an open-source game, created by David White in 2003. Wesnoth describes itself as a "turn-based tactical strategy game with a high fantasy theme". [1] This game was selected for its high quality of code and visuals, as well as its representation of a wide class of modern computer games. A large community of Wesnoth users and developers also provided this work with valuable game reference material and technical support.

1. `http://wiki.wesnoth.org/Description`

5.1.1 Turn-based Nature

For decades, war games have attempted to simulate the realities of battle in a miniaturized and reproducible fashion [1]. In some versions, players move physical representations of battlefield forces across a large room that symbolizes the battlefield in order to learn tactics and other aspects of war. One defining feature of war games is their use of randomness in their mechanics. This randomness simulates various unknown effects in war and forces the player to adapt their strategy to unforeseen outcomes. In this way, the player learns flexibility and resourcefulness. Battle for Wesnoth is a direct digital successor to these war games and has the same emphasis on tactics and randomness.

Battle for Wesnoth is similar to other war games and chess in that they all simulate war between two or more teams, on a surface that is discretized into a hex or grid pattern. Every turn, a team can move its units and attack other teams' units, according to various unit attributes and mechanics. After the team has moved their units or otherwise passes their turn, control passes to the next team. This continues until the game is over, as signalled by checkmate in chess and by one side losing all their units in Wesnoth.

A section of a Wesnoth map can be seen in Figure 5.1, showing the hex-shaped terrain which is historically common in war games. In this figure the two opposing teams are coloured red and blue, with a red unit attacking a blue unit. One interesting mechanic of Wesnoth is that the different terrain types of tiles will affect the damage taken by units on those tiles. This interesting mechanic will be clarified below.

5.1.2 Unit Actions

Drawing from other fantasy works, the dozens of units in Wesnoth have imaginative type names, races, and attributes. For simplicity, only one particular unit type was selected for the experiments in this work. The relevant attributes of this

Figure 5.1: Units fighting on a Wesnoth map

'Dwarvish Fighter' unit will be explained and related to the various game mechanics of Wesnoth.

Movement

When a unit moves in Wesnoth, it spends movement points to move across tiles. Depending on the terrain type and the unit, tiles take a varying number of movement points. For example, the Dwarvish Fighter has four movement points, and will expend one to cross a Cave tile and three to cross a Swamp tile. There are also impassable tiles such as the Lava tile, which the Dwarvish Fighter cannot move onto or across.

Attacking

Units in Wesnoth begin the battle with a particular number of health points based on their unit type. The Dwarvish Fighter has 38 health points. This health value

will be reduced through the attacking process, and cannot be restored as this work has removed all mechanics to restore health. These simplifications are discussed in Section 5.2.

Attacking is a relatively complicated process in Wesnoth, as it is dependent on the random number generator in Wesnoth, the terrain mechanic, and various unit attributes.

The attacking unit A must be directly adjacent to its target T, and must not have already attacked that turn. A will then directs a number of strikes against T, potentially connecting and doing damage. After each one of A's strikes, whether it connects or not, T will make a counter-strike against A.

Whether a strike or counter-strike will connect or not is based on the terrain type of the tile T is on. Terrain tiles have defence ratings representing how often a strike will fail to connect, as determined by the random number generator within Wesnoth. For example, a Dwarven Fighter has a defence rating of 30 percent on a Forest tile, meaning that a strike has a 30 percent chance of failing to connect. In contrast, a Cave tile has a higher defence rating of 50, forcing strikes to miss 50 percent of the time. Therefore, terrain is a crucial part of any Wesnoth strategy.

Attack Name	Num. Strikes	Potential Damage	Damage Type
Axe	3	7	Blade
Hammer	2	8	Impact

Table 5.1: Two attacks of the Dwarvish Fighter

A unit may have a multitude of different attacks, each with a different number of strikes per attack, and damage potential. The attacks for the Dwarven Fighter are seen in Table 5.1. If A selects the axe attack, A will launch three strikes against T. For each strike that hits T, a maximum of seven damage is taken away from the health of T. The hammer attack will similarily allow two strikes with eight damage maximum. This maximum damage is modified by T's resistance to that damage type, lowering the amount of damage done. For example, a unit may have

10 percent resistance to the *Blade* damage type, which would reduce *Blade* damage taken by 10 percent. Attack type selection is automatically determined in Wesnoth, based on each attack's estimated damage to the target.

This randomized fighting mechanic has caused much discussion and disagreement within the game community about the effects of randomness on Wesnoth's gameplay [63]. This randomness also affects which techniques can be used to create an artificial intelligence for Wesnoth. Unlike the deterministic nature of chess, searching for optimal moves may be infeasible. An attack with three strikes and three counter-strikes creates the possibility of 16 different outcomes depending on how many strikes connect. This *branching factor* may make it impossible to examine a large number of turns ahead in the battle.

5.2 Simplifications Made

To order to manage the complexity of Wesnoth, a number of game features were removed or simplified for this work. It must be re-stated that the objective was not to create viable strategies for the entirety of Battle for Wesnoth, but of a suitably complicated turn-based game. Thus while important mechanics of Wesnoth were removed, the core game mechanics of moving, attacking, and terrain defence remain. These removed topics are mentioned here for completeness and as possible topics for future work.

5.2.1 Unit Recruitment and Economy

A normal multiplayer game in Wesnoth begins with each team receiving a *leader unit* and a quantity of gold. The leader stands on special recruitment tiles and purchases units of differing types to move around the map and attack other teams. In order to recruit more units, gold must be obtained. Two gold is given to a team when they begin their turn, and special 'village' tiles also produce gold every turn when they have been captured. A team captures a village by moving a unit onto it. In this way, players must manage their gold economy by recruiting units to fight and secure villages.

As noted above, this economic aspect of the game was removed. This was done to focus on the particulars of moving and attacking which is perhaps applicable for a wider variety of tile-based games, rather than the Wesnoth-specific actions of capturing villages and making recruitment decisions.

5.2.2 Other Mechanics Removed

Resting

If units do not move or attack for a turn, they are considered as resting and will be healed by two health points at the end of the turn. This mechanic was removed in order for the health of both teams to monotonically decrease.

Healing and Poisoning

Spellcasting units may use special actions to restore health or cause damage every turn through poison effects. These actions were removed to prevent healing as above, and for direct unit-to-unit attacks to be the only damage-dealing mechanism.

Levelling

Units gain experience points when defeating enemies. Upon obtaining enough experience points, units *level up*, which restores their health and increases other attributes. Again, this was removed to ensure that the health points of units only decreased.

Time of day

Wesnoth simulates a time of day for the battle, changing from dusk to afternoon to evening every few turns. This provides a bonus for some races of units and penalties for others. For example, *Undead* units will do 25 percent less damage during the day. The time of day was fixed at afternoon throughout all experiments so that attack damage was consistent throughout the battle.

Random traits

Units receive random traits when recruited or otherwise placed on the map. Examples include *Strong* which gives extra damage per strike and one extra

health point, and *Quick* which gives one extra movement point but removes five percent of the units' maximum health points. Random traits were removed in order to ensure that both teams were equal in strength and ability.

Chapter 6
Methodology

This chapter discusses how the evolutionary computing process was modified for the Battle for Wesnoth domain. In particular, the representation, modification, and evaluation of strategies will be presented. Fitness functions and other metrics will be introduced to measure the strength of strategies within a Wesnoth battle against another strategy.

6.1 Strategy Representation

As mentioned in an earlier chapter, a behaviour tree typically represents the artificial intelligence for a single character. In this work, a behaviour tree has been generalized into a team strategy. On a team's turn, every unit on the team queries the behaviour tree and performs the actions if possible. The choice to use a single tree for the entire team was motivated by the work of Miles and Louis [38].

This section discusses specific modifications that were made to the behaviour tree framework to allow strong strategies to develop through evolution. Domain-specific actions were created to be used in a modular way, ensuring that genetic recombination led to valid strategies.

6.1.1 Node Overview

Three important details of the behaviour tree system are discussed here. The first detail is the use of an attribute table within each node to control node behaviour. The next detail of interest is the use of a *blackboard* structure, which centralizes data for use by all nodes in the behaviour tree. Finally, a list-based data format is used for all data processing within the behaviour tree.

The attributes in a node parametrize the behaviours of that node. For example, two attributes in the **Sort** node define the sorting behaviour of that node. These attributes are described for each node type below. This attribute system was done to reduce the number of nodes to be implemented, as well as provide a simple mechanism for mutating a node as described below.

A *blackboard* is a structure used by an artificial intelligence to share information among components [39]. In this work, the blackboard is used to share data amongst all nodes in a unit's behaviour tree. When a unit queries their behaviour tree at the beginning of each turn, the blackboard is initialized to contain only the unit itself. The implementation of the blackboard is a stack-based structure where nodes in the tree may access the last object in the stack. For example, the **Sort** node will use the last object on the blackboard as input, sort it if applicable, and then add it back to the blackboard.

The data placed on the blackboard and accessed by nodes is list-based, and composed of objects representing Wesnoth units and tiles. By defining a common data structure to be used, each node can robustly handle all input. Consequently, a wider range of behaviour trees are syntactically valid at least, if not semantically valid. This is important to avoid computationally-wasteful invalid strategies.

6.1.2 Composite Nodes

The two composite nodes used in this work were the **Sequence** and **Selector** nodes as described in Section 3.2.1. Both nodes were given an attribute to control whether children would be evaluated in a particular order or in random order.

The **Sequence** node was also given a special attribute *ignoreFailure* that would prevent the node from returning when a child returned false. This property thus forced the **Sequence** node to evaluate all children before returning true. This attribute was created so that hand-built strategies could perform actions despite having subtrees that occasionally fail to perform their actions.

6.1.3 Action Nodes

The selection of appropriate *Action* nodes for the formalism was determined by three influences. The first was by studying sample Wesnoth AIs provided by the community in order to encapsulate them into logical steps. The second influence was by examining what actions were available in the Wesnoth code itself. Finally, the third influence was the blackboard and list-structure, which necessitated creating nodes to manipulate the blackboard structure.

In accordance with these influences, explicit *Condition* nodes were not used. *Action* nodes simply return true if they successfully complete their action, and false if they fail for any reason including invalid input. If applicable, the input to nodes is the last object on the blackboard. As this object may be a list of other objects, and most nodes do not operate on lists, the last object in the list will be taken as input in this case. For example, a **GetUnits** node, when queried, will place a list of units on the blackboard. If an **Attack** node is then queried, the input object will be the last unit in this list. This list-based design was created for modularity, as all data then has a unified format.

A list of the nodes created can be found in Table 6.1. The entry for each node contains the restrictions on the input data, the actions the node performs, and a description of the output object to be placed on the blackboard if any. Nodes may perform multiple actions, which is indicated by a describing name on the action. Note that some nodes are fairly permissive in their input. For example, the **GetUnit** node accepts all objects that have a x and y coordinate. This includes both tile objects as well as unit objects. As with other implementation decisions, this non-discrimination was done to allow as many strategies to be valid as possible. As mentioned above,

if the input is not as specified, or if the action is unable to be taken, the node will return false to its parent. If the action was successful, the node returns true.

Table 6.1: Action nodes created

GetUnit	
INPUT	Accepts an object with x and y coordinates
ACTION	Examines the game map at that x and y for a unit. If there is one, it will be placed on the blackboard
OUTPUT	A unit object if found
GetTiles	
INPUT	Accepts an object with x and y coordinates
GETREACHABLE	If the object is a unit, find all tiles that the unit can move to within one turn. If the object is not a unit, the adjacent action will be performed.
GETADJACENT	Find all tiles directly adjacent to the object's x and y position
OUTPUT	List of tiles
GetUnits	
INPUT	None
ACTION	This action depends on the value of the *side* attribute within the node. This node can either create a list of units that are on the same or opposing team of the querying unit
OUTPUT	List of units
Pop	
INPUT	None
ACTION	Pops the last object off the blackboard
OUTPUT	None

SetOperation	
Input	Takes the two top-most objects on the blackboard, which must be lists. False is returned if either object is null or not a list
Union	Creates a new list of the union of objects from both input lists
Intersection	Creates a new list of objects that are found in both input lists
Output	The created list

Sort	
Input	Accepts any object
Action	Sorts the object depending on a number of sorting functions. These are by health points, movement points, distance to querying unit, and defence rating. The first two are only applicable to units, and the last to tiles. If the object is not of the correct type, false is returned.
Output	The sorted list, or the original object if it was not a list

Attack	
Input	Accepts an object with x and y coordinates
Action	Directs the querying unit to attack the x and y coordinates. This attack will fail if the coordinates are not adjacent to the attacking unit, or if the unit is directed to attack themselves or another teammate. These failure conditions are automatically determined by Wesnoth. The return value of this node will be whether the attack order succeeded or not
Output	None

Move	
INPUT	Accepts an object with x and y coordinates
ACTION	Directs the querying unit to move to these coordinates. If the object is a unit, the move coordinates will be the closest open tile to the object. The return value for this node is whether the move order was successful or not
OUTPUT	None

6.2 Evaluation

In order to assign a fitness to each strategy, strategies were evaluated within an instance of Battle for Wesnoth. First, the strategies are converted into the format used by the Wesnoth AI. Then a number of Wesnoth instances are run, with each strategy in the population battling against the default AI. In this work, instances were run in parallel to decrease the time taken for each generation to about four seconds. The output produced by each instance is then parsed to obtain a fitness value for the strategy that was evaluated by that instance. This fitness value is then used by the genetic operators to create a new generation of strategies. This process then repeats, as shown in Figure 6.1.

6.2.1 File Format

Behaviour trees are represented within the evolutionary algorithm as an XML file, as in Lim's work [32]. The tree-like structure of XML allows genetic operators, such as moving subtrees between individuals, to be easily defined and implemented. Another advantage is that XML is human-readable, allowing easy examination of the behaviour trees produced.

In order to be evaluated within Wesnoth, each strategy must be transformed into a Lua file. This is a straightforward translation operation which re-creates the

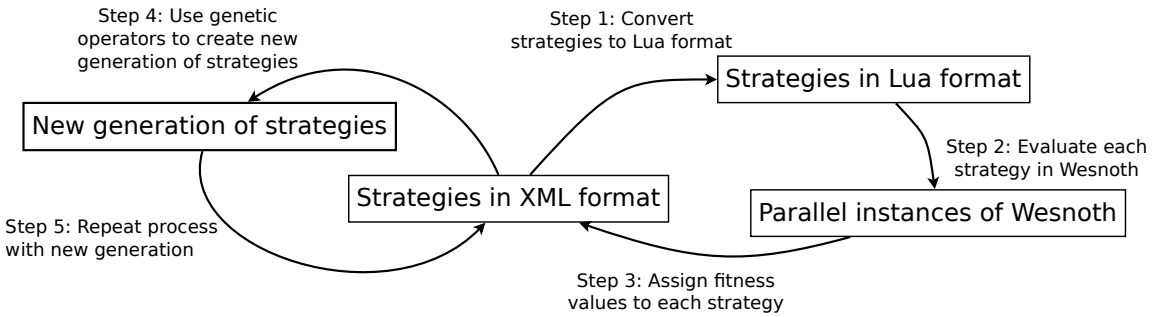


Figure 6.1: Diagram of evolutionary algorithm

```
- <BehaviourTree fitness="103.33">
  - <Sequence name="Sequence0" ignoreFailure="true">
      <GetUnits name="GetUnits00" side="enemies"/>
      <Sort compareFunction="healthPts" name="Sort01" order="highToLow"/>
      <GetTiles name="GetTiles03" tileType="adj"/>
      <Sort compareFunction="defence" name="Sort04" order="lowToHigh"/>
      <Move name="Move06"/>
      <GetUnits name="GetUnits07" side="enemies"/>
      <Sort compareFunction="healthPts" name="Sort08" order="highToLow"/>
      <Attack name="Attack010"/>
    </Sequence>
  </BehaviourTree>
```

(a) XML format

```
return {
    init = function(ai)

        local Sequence0 = Sequence(nil, 'Sequence0', 'ignoreFailure:true ')
        local root = Sequence0
        local GetUnits00 = GetUnits(Sequence0, 'GetUnits00', 'side:enemies ')
        local Sort01 = Sort(Sequence0, 'Sort01', 'compareFunction:healthPts order:highToLow ')
        local GetTiles02 = GetTiles(Sequence0, 'GetTiles02', 'tileType:adj ')
        local Sort03 = Sort(Sequence0, 'Sort03', 'compareFunction:defence order:lowToHigh ')
        local Move04 = Move(Sequence0, 'Move04', '')
        local GetUnits05 = GetUnits(Sequence0, 'GetUnits05', 'side:enemies ')
        local Sort06 = Sort(Sequence0, 'Sort06', 'compareFunction:healthPts order:highToLow ')
        local Attack07 = Attack(Sequence0, 'Attack07', '')
        return root
    end
}
```

(b) Lua format

Figure 6.2: $S_{Defence}$ in two data formats

tree using behaviour tree class nodes defined within Lua. An example of $S_{Defence}$, a hand-built strategy, is seen in both XML and Lua formats in Figure 6.2.

6.2.2 Collecting Output

Once created, each Lua strategy file is then passed to an instance of Wesnoth through a command-line argument. In order to speed up the evaluation process, 16 strategies are evaluated at once. The computer that produced results for all tests was an Intel Core i7-3820 CPU with 8 processors at 3.60 GHz running Ubuntu 13.04.

The map used for the genetic algorithm process was a modification of a built-in multiplayer Wesnoth map, and can be seen in Figure 6.3. It is also suggested as an artificial intelligence testing map by the game's developers. However, as mentioned above, all villages were removed from the map before the experiments were run, so that the default AI was not influenced by their presence.

Figure 6.3: The AI testing map

A turn limit was placed on each battle in the case a winner did not emerge. This occurred occasionally when the default AI would fail to make any actions. The reason for this is unknown, but is potentially related to the removal of villages or the gold economy mechanic. When the turn limit was reached, the winner was selected as the team with the highest health sum. As battles that completed normally took around seven turns on average, the turn limit was set to be twenty turns.

When a Wesnoth instance finishes executing, its textual output is fed as input into a fitness function program. This program then outputs a numerical fitness value, which is placed into the XML file for the strategy which was evaluated in that Wesnoth instance. The genetic operators described above can then operate on these XML files to produce strategies for a new generation. A number of fitness functions may be used, and may have a large impact on the results produced by the

evolutionary algorithm. Therefore, a number of fitness function experiments were conducted, and will be discussed in the following chapter.

6.2.3 Parameter Selection

One issue for studying evolutionary algorithms is the large number of parameters that can be changed. For example, the fitness function, population size, number of generations, and the number of trees created through crossover can all affect the performance of the genetic algorithm. While this author acknowledges that exhaustive tests should be conducted to isolate the effects of every parameter, it was decided that other experiments were more relevant to this domain. Therefore, recommended parameter values will be based on work such as from DeJong [10]. When available, values will be used based on preliminary experiments. Brief notes throughout this section will also discuss rationale for the parameter values selected.

6.2.4 Fitness Function

The winner of a battle is directly determined by the health of units on a team, whether the battle finished normally when all units on a team are defeated, or when an artificial winner is declared when the turn limit is reached. Therefore, the fitness function used was based on the difference in health between the two teams at the end of the battle. This metric has been previously seen in the literature [37][34]. Other fitness functions are explored in the next chapter.

$$D = \sum(U_{health}^{Last} | U \in Team_1) - \sum(U_{health}^{Last} | U \in Team_2)$$

Equation 6.1. Health difference function

Equation 6.1 shows the health difference formula, to be calculated on the last turn of the battle as denoted by *Last* in superscript. The health difference D calculates the difference between the sum of each team's health. Therefore, a positive D means that team 1 had a greater health sum than team 2, and vice versa. This D value will

be used to define a performance gain metric in the next chapter, which determines the increase in performance attributable to evolution.

6.3 Recombination

This section will detail the genetic operators used for the recombination of behaviour trees. As described above, the three main recombination operators are selection of strong solutions, crossover between two solutions, and mutation of solutions. A discussion will follow concerning the lack of a repair operator in this work.

6.3.1 Selection

The **elite**, **tournament**, and **roulette** selection operators described in an earlier chapter were used in this work, and are used to select strong individuals to be copied into the next generation. Each selection operator generates a certain amount of new individuals for the next generation. This amount is controlled by experimentally determined parameters, which are seen in Table 6.3. For instance, the **tournament** selector generates 25 percent of the new generation's individuals, minus one individual, which is selected through the **elite** operator. The remainder of the population is created by using the crossover operator, as explained below. The number of tournament rounds performed within the tournament operator was set to be four. This value was experimentally chosen to select individuals with high fitness.

Table 6.3: Parameters for creation of new generation

Operator	**New Indiv. Created**
Elite	1
Roulette	25% of new pop.
Tournament	25% of new pop. - 1
Crossover	50% of new pop.

6.3.2 Crossover

The purpose of a crossover operator in evolutionary computing is to propagate elements of strong solutions throughout the population. As stated above, these operations are representation-specific. One advantage of using a tree-based representation is that a standard crossover operation is found in the literature [28]. As implemented in our work, two parent trees are selected using the roulette method. Then, a randomly chosen subtree on each tree is swapped to produce a new individual. As mentioned above, fifty percent of the new generation is created through this crossover operator. This value was experimentally selected based on preliminary experiments.

6.3.3 Mutation

Mutation was applied to the behaviour trees in order to search other points in the solution space. Three mutation operators were defined in this work, and will be explained below: adding a new node, removing a subtree, and modifying the attributes of a random node.

In the experiments conducted each behaviour tree had a random one-in-ten chance of being selected for mutation. This value is suggested by the literature based on DeJong's criteria of $value = 1/N$ where N is the population size [10]. In order to control the mutation process, a number of parameters were defined. When a tree is selected for mutation, a random choice is made to determine which one of the above mutation operators takes place. This was done to weight the evolutionary algorithm towards growing the behaviour trees, rather than growing and shrinking equally. This consideration was required for a later experiment, which evolved strategies from a null strategy, as discussed later. Table 6.4 shows the weighting for each operator.

Addition

This mutation operator adds a new child node to a random node in the behaviour tree. This operator is important in order to add complexity and new functionality to trees. This operator is implemented by first choosing a random composite node

Table 6.4: Mutation operator weights

Operator	Chance for Selection (%)
Addition	60
Removal	10
Modification	30

in the tree to add a new child node to. Adding a child to a non-composite would create a syntactically-invalid tree. Therefore, this is avoided.

The second step in the addition process is to assign an *abstract node type* to the child node. This type is selected as a random choice between composite or non-composite. Third, the *concrete node type* is selected. For example, if the new node is composite, there is a fifty-fifty chance of being a **Sequence** node or a **Selector** node. For the non-composite nodes, there are eight node types as defined earlier in this chapter which are equally weighted for selection. Finally, the child node is given random attribute values.

Removal

Another mutation operator is to remove a random subtree. All nodes and their subtrees are available for removal excluding the root.

Modification

The final mutation operator is to modify the attribute of a random node in the tree. This is performed by selecting a random attribute of the node. A new value for the attribute is then randomly chosen among the valid values of that attribute.

6.3.4 Repair

The above crossover and mutation operators have been constructed so that syntactically invalid trees cannot be produced by the evolutionary algorithm. Therefore, an explicit operator to repair the tree is not needed. As mentioned above, syntactically valid trees are desirable to avoid evaluating strategies that will fail or are

otherwise incorrect. Future work may consider the identification and repair of semantically invalid trees, which have nodes in combinations that produce no valuable results. For example, a tree consisting only of composite nodes is syntactically valid, but semantically invalid.

Chapter 7
Experiments and Results

This chapter will demonstrate the results of evolving game-playing strategies through a number of different experiments. The criteria for measuring strategy performance will be defined, providing the ground for measuring the improvement of evolved strategies against the baseline results. Evolved strategies will also be examined in-game, with critical discussion focusing on the generality of these results. Further experiments will consider the use of various fitness functions for the evolutionary algorithm, as well as the effects of randomness on evolution.

7.1 Baseline Strategies

In order to measure any possible improvements obtained from evolving strategies, it is necessary to first measure the performance of *baseline strategies*. Three types of strategies were evaluated for these results. A null strategy was evaluated to determine the effects of randomness on evaluation, hand-built strategies were created to identify strong strategies, and the default game AI was tested as the primary opponent.

7.1.1 Null Strategy - $S_{\textbf{Null}}$

The *Null* strategy is a behaviour tree consisting of only a **Sequence** node, as seen in Figure 7.1. When evaluated, no action will be taken. This strategy was tested

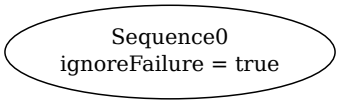


Figure 7.1: S_{Null} behaviour tree

to indicate the effects of randomness on the battle, as any win by this strategy will be solely attributable to randomness. This strategy contains a node solely for the purposes of seeding evolution with the null strategy, as described in a future section.

7.1.2 Hand-built Strategies

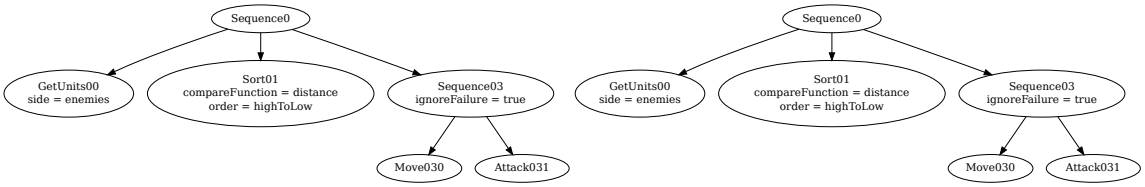
Three behaviour trees were hand-constructed in order to obtain benchmark results, as well as to provide starting trees for later seeding experiments. The first two strategies direct the unit to attack the closest unit or the weakest unit, while the third strategy takes the mechanic of terrain defence into account.

Attack-Closest Strategy - S_{Closest}

The *Attack-Closest* strategy directs the unit to attack the closest unit. The behaviour tree can be seen in Figure 7.2a. The root node is a **Sequence** node that queries its children from left to right. The node also ignores any false values returned from the children, providing increased flexibility to any strategies that evolve from this strategy. This evolution is seen in a later experiment. The **GetUnits** node returns a list of enemies for the **Sort** node to sort by decreasing distance. The second **Sequence** node then directs the unit to **Move** and then **Attack** the last unit in the list. Therefore the closest enemy unit is targeted, moved to, and attacked.

Attack-Weakest Strategy - S_{Weakest}

The *Attack-Weakest* strategy is very similar to the Attack-Closest strategy, as seen in Figure 7.2b. However in the Attack-Weakest strategy the **Sort** node orders the units by health points instead of distance. Therefore, the weakest enemy unit is selected to be attacked.



(a) S_{Closest} (b) S_{Weakest}

Figure 7.2: Behaviour trees for two hand-built strategies

Defence Strategy - S_{Defence}

The *Defence* strategy is slightly more complicated than the other hand-built strategies, as it evaluates the defence rating of map tiles when selecting a tile to move to. It is shown in Figure 7.3. The first two nodes obtain a list of the unit's enemies, and sort them by health in descending order. This list is then placed on the blackboard, as mentioned in Chapter 6. The next node finds the adjacent terrain tiles to the last unit in this blackboard list. These tiles are then sorted by their defence value and then the unit is moved to the tile with the highest defence. Finally, the unit is directed to attack the weakest enemy. Therefore, the unit should move to the tile with the highest defence value that is adjacent to the weakest enemy, and then attack that enemy.

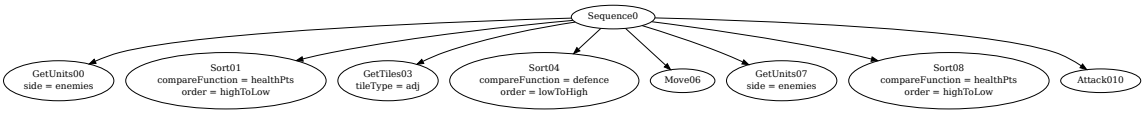


Figure 7.3: S_{Defence} behaviour tree

Default Artificial Intelligence - S_{Default}

Battle for Wesnoth contains its own custom artificial intelligence (AI) built by the Wesnoth community to challenge the player. This AI has the ability to capture villages, recruit units, and some limited coordination of unit movement. As noted, as

the recruitment and gold economy aspect of the game has been removed, the default AI may not be playing at its full strategic potential. However, the default AI does consider such mechanics as a unit being surrounded, the defence values of tiles, the expected damage to be taken from an attack, and a measure of value of a target. Thus, this AI is expected to perform well even with multiple domain mechanics removed. As the default AI is written in the C++ programming language, it is not available to be displayed as a behaviour tree.

7.1.3 Baseline Performance

The initial performance metric examined will be the percentage of games won in a number of battles between two strategies. In Table 7.1, each entry is the win percentage of the strategy in that row versus the strategy in that column. For instance, the top row is the win percentage for S_{Null} against the other strategies, and can be seen to be quite low. On the map that evolution took place on, teams can start on the south side or the north side of the map. Strategies were therefore evaluated on both sides in order to account for any bias in the game map. In Table 7.1a, the entry denotes the win percentage when the strategy in that row was placed on the south side of the map. For example, the top row in this table shows results for when S_{Null} controlled the team on the south side of the map. Conversely Table 7.1b contains results from the battles where the strategy in that row was placed on the north side of the map.

The results were collected by 800 games between the two strategies. For each battle, Wesnoth's random number generator was seeded with a different number. This was done to examine a large number of possible outcomes for each battle.

The diagonal of Table 7.1 is highlighted in bold, and shows the results of strategies evaluated against themselves. The S_{Null} versus S_{Null} entry is empty as neither strategy performed an action. Other entries show that strategies playing against themselves win within 50 percent of the time, which includes a noted variance of about eight percent. This self-play result indicates that neither starting position has

Table 7.1: Performance results for baseline strategies

(a) On south side of map

-	S_{Null}	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Null}	**-**	11	10	3	14
S_{Closest}	87	**43**	38	21	35
S_{Weakest}	90	74	**49**	22	38
S_{Defence}	97	70	77	**46**	58
S_{Default}	84	76	69	58	**53**

(b) On north side of map

-	S_{Null}	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Null}	**-**	13	10	3	16
S_{Closest}	89	**43**	26	30	24
S_{Weakest}	90	62	**49**	23	31
S_{Defence}	97	79	78	**46**	42
S_{Default}	86	65	62	42	**53**

an advantage. Another interesting observation is the disparity between some strategies starting on the south and north sides of the map. For instance, when S_{Closest} battles against S_{Weakest}, S_{Closest} wins 38 percent of the time when it started on the south side of the map in Table 7.1a, but only 26 percent when it started on the north side in Table 7.1b. This disparity does not arise in every result, but seems to arise out of the interactions of strategies with the map.

The results for S_{Null} show it does lose the majority of battles, as expected. Curiously, the null strategy does win more often against the default AI than against the hand-built strategies. This may be due to the aforementioned issue of the default strategy occasionally stopping its actions. If this occurs the null strategy may temporarily have a higher team health depending on the random numbers used in the battles. In this case, the null strategy would be declared the winner of the battle.

The S_{Closest} and S_{Weakest} hand-built strategies perform quite well for being so simple. These strategies defeat the S_{Null} a majority of the time, but lose to the more sophisticated hand-built defence strategy and the default strategy. The S_{Weakest} strategy is also stronger than the S_{Closest} strategy. This was expected as it is a tactically wise choice in this domain to attack the weakest enemy.

S_{Defence} also wins a majority of the time against the null strategy and tends to win against the other hand-built strategies. It also defeats the default strategy 58 percent and 42 percent of the time on the south and north side of the map respectively. This was an unexpected result, as S_{Default} was assumed to be a superior strategy to any hand-built strategy. However, this performance may also be slightly increased due to the issue of the default AI ceasing to issue further orders. As mentioned, it is not known why this occurs, but it is theorized that the removal of the recruitment and gold economy mechanic may be causing this issue.

7.2 Evolving Strategies

This section discusses the evolution of strategies and their comparison to the baseline results. Strategies will be evolved from the null strategy, randomly-created strategies, and from a population seeded with the hand-built strategies. The evolution from the null strategy will be shown in detail, discussing the evolution process as well as metrics of strategy performance. In all evolution experiments, the opponent will be the default Wesnoth AI, denoted as S_{Default}. All evolutionary runs took less than 110 minutes to complete.

In the following evolution experiments the random number seed, used to initialize Wesnoth's random number generator, was the same for every battle. This was done to encourage the strategies to optimize against a relatively fixed target. Later experiments will show the effects of changing these seed every generation of evolution, as well as changing the seed for every battle. The evolved strategy was also fixed to be placed on the south side of the map in each battle. Again, this was done to lower the amount of noise in the evaluation of strategies. The population size for

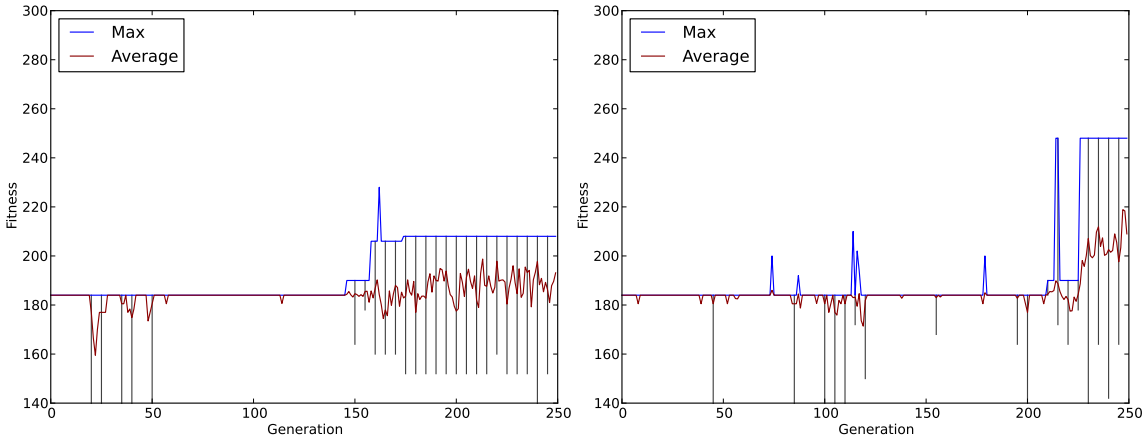
all evolution was 16, as this was a multiple of the number of cores available on the system, reducing the time required per generation. Other evolutionary parameters can be found in the preceding chapter.

7.2.1 From Null Strategies

In this section, the results are presented for the evolutionary algorithm attempting to evolve an population of null strategies. This experiment demonstrates how quickly evolution can produce viable strategies, and also how metrics can be used to verify that evolution was a success.

Evolution

Two evolution runs will be presented here. The first run is where the evolved strategy is always on the south side of the map, while in the second run the evolved strategy randomly switches between sides of the map. This was done in order to examine if restricting the strategy to one side would increase evolutionary performance. Again, the opponent strategy is set as S_{Default}.



(a) Fixed to south side of map (b) Randomly switched sides of map

Figure 7.4: Fitnesses of S_{Null} evolution

Figure 7.4 shows the maximum and average fitness of the population over 250 generations. The gray lines above and below the average fitness demonstrate the

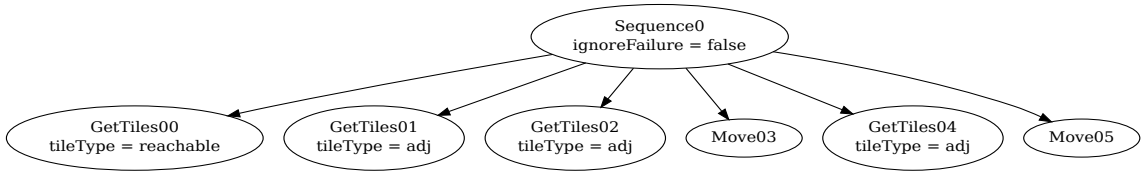
fitness range from the maximum fitness of the population to the minimum fitness. The maximum and average fitness are higher in Figure 7.4b, which suggests that stronger strategies were evolved. Note that in earlier generations, changes caused the average fitness to decrease but these changes were discarded from the population. At different points in the two evolution runs, an individual arose in the population that had a slightly higher fitness. The algorithm could then propagate that change to other individuals. Recombination could then lead to further increases in fitness. It is also interesting to note that fit individuals which had the highest maximum fitness were sometimes lost from the population. Even though elite selection carried that individual forward into the next population, subsequent mutation likely lowered its fitness. Therefore, it is important to save the highest-fitness solutions from the evolution process.

As discussed above, the stopping criteria of the algorithm was 250 generations, which was experimentally selected to facilitate rapid experiments. Although the maximum fitness increased throughout these evolutions, it is unclear if the improvement would continue. However, it is encouraging that the evolution runs shown only took about 96 minutes each, making longer or repeated runs practical.

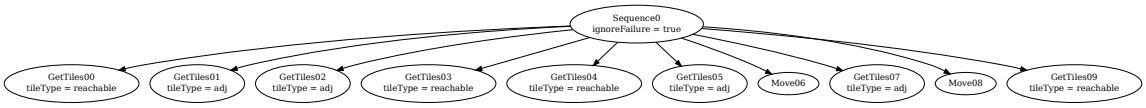
Performance

After evolution, the strategies with the top fitnesses from the evolution process were selected to be evaluated against the hand-built and default strategies. In this work, only one or two strategies were evaluated. In sorting by fitness value, most of the top strategies were almost identical due to the elite evolutionary operator. This may be disadvantageous, as it may be more informative to select varied strategies by comparing their structure or function.

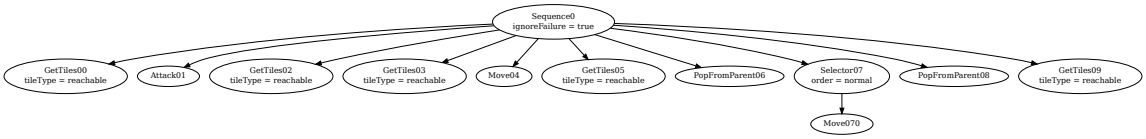
The first two strategies in Figure 7.5 are from the first evolution run, where the evolved strategies were fixed to the south side of the map. Figure 7.5a shows a strategy from generation 162 ($S_{\text{A-Gen162}}$), while Figure 7.5b is a strategy from generation 176 ($S_{\text{A-Gen176}}$). The third strategy S_{B}, in Figure 7.5c, is from the second



(a) $S_{\text{A-Gen162}}$



(b) $S_{\text{A-Gen176}}$



(c) S_{B}

Figure 7.5: Three behaviour trees evolved from S_{Null}

evolution run, where the evolving strategies were placed on a randomly chosen side of the map.

The results of the evaluation process can be seen in Table 7.2. As before, Table 7.2a is the win percentage of a strategy when placed on the south side of the map, compared to Table 7.2b when the strategy was evaluated on the north side of the map. These results show that the evolution was successful, as the evolved strategies perform much better against the other strategies than S_{Null}. As $S_{\text{A-Gen162}}$ and $S_{\text{A-Gen176}}$ had a fixed role on the south side of the map during evolution, it is unsurprising that their performance suffers when they are placed on the north side. Comparing $S_{\text{A-Gen162}}$ and $S_{\text{A-Gen176}}$, it is interesting to note that the strategy from the later generation is stronger against S_{Default}, but is much weaker against the other strategies. This suggests that $S_{\text{A-Gen176}}$ may be over-specialized against S_{Default}.

As S_{B} randomly switched map sides during evolution, it developed a robust strategy that is able to defeat most other strategies. However, it should be noted that S_{B} is still relatively weak against S_{Default} when evaluated on the north side of the

Table 7.2: Performance of strategies evolved from S_{Null}

(a) On south side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Null}	11	10	3	14
$S_{\text{A-Gen162}}$	68	61	33	33
$S_{\text{A-Gen176}}$	83	69	35	86
S_{B}	72	71	64	89

(b) On north side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Null}	13	10	3	14
$S_{\text{A-Gen162}}$	71	61	45	19
$S_{\text{A-Gen176}}$	18	23	11	27
S_{B}	57	60	57	36

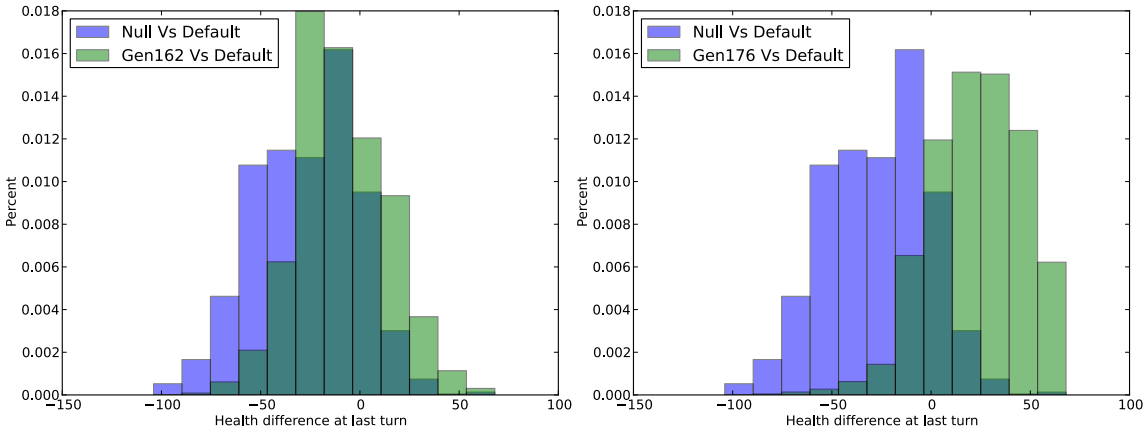
map. The statistical significance of these results will be examined before strategies are discussed in depth.

Statistical Significance

As noted above, there is an element of variance in the win percentages of a strategy. This is because the randomness in a domain directly affects the results of actions within the battles. Therefore, a direct comparison of win percentages may be unsuitable to declaring that evolution has successfully occurred. A performance improvement metric will be defined here that is robust to randomness.

The fitness function used in the evolution algorithm is described in the last chapter. It produces a D value, which is the difference in health between the two teams at the end of a battle. The D values can be recorded from each battle in an evaluation of two strategies. After a large number of such battles, these D values form a normal distribution around some mean with high confidence. If the mean is less than zero, the first team is weaker on average than the second team, and vice versa.

Therefore, the hypothesis of this new metric is that the mean of the distribution represents how strong the strategy is. Strategies with statistically significant changes in mean are meaningfully weaker or stronger. If evolution produces a strategy with a shifted mean compared to the initial strategy, then that evolution run will be declared successful.



(a) D histogram for $S_{\text{A-Gen162}}$ (b) D histogram for $S_{\text{A-Gen176}}$

Figure 7.6: Comparison of D values for evolved strategies

As mentioned above, a distribution of D values is obtained when battling two strategies repeatedly. Figure 7.6a shows the distribution of D values given by the evaluation of S_{Null} versus S_{Default} on the left-hand side of the figure. The D distribution given by the evaluation of $S_{\text{A-Gen162}}$ versus S_{Default} is slightly to the right. Both distributions are normal with high confidence, and with statistical significance the mean moves from -27 to -9. [1] This result means that it can be stated that $S_{\text{A-Gen162}}$ is a stronger strategy than S_{Null}, and that evolution has been successful.

Figure 7.6b shows another histogram with $S_{\text{A-Gen162}}$ versus S_{Default}. The shift in means is even more pronounced here, as the mean has shifted from -27 in the first

1. Statistical significance given by a T-test. $P < 0.05$

distribution to 24 in the other. Again, this shift is statistically significant, showing evolution has been successful in producing a stronger strategy. [2]

Examining the Behaviour Trees

Through the study of the evolved strategies, it may be possible to learn new information about the problem space. This could be in the form of a novel strategy or possibly an unintended or unbalanced design decision.

The first two strategies in Figure 7.5 above are relatively simple strategies that exploit a combination of the **GetTiles** action and the **Move** action. The **GetTiles** node has been previously explained in Table 6.1, and is intended to create a list of tiles around a location or unit that it retrieves from the blackboard. This new list of tiles will be placed on the blackboard as well. Therefore, if two **GetTiles** nodes are evaluated after another, the first node will place a list on the blackboard, which will be input to the second node. Per the specification of the **GetTiles** node as described in the last chapter, the second node will return tiles around the last object in this list. Therefore, a repeated string of **GetTiles** nodes will continually return tiles farther and farther away in a particular direction. The unit will then move to the last position on the blackboard. This movement style was an emergent strategy, unforeseen in the design of the action nodes.

The implementation of the **GetTiles** nodes within Lua and Wesnoth mean that the last tile in the list placed on the blackboard will be the tile directly up and to the left of the input location. Therefore, $S_{\text{A-Gen162}}$ will move the unit approximately two units in the up-left direction while $S_{\text{A-Gen176}}$ will attempt to move the unit approximately six tiles. Impassable terrain tiles may make this movement distance change slightly.

S_{B} is slightly different in its orders. The first two nodes order the querying unit to attack an adjacent tile. This shows the utility of the *ignoreFailure* attribute in the **Sequence** node at the root. As this attack will most likely fail, the strategy may still be viable by ignoring the return value. The strategy then queries two **GetTiles**

2. $P < 0.05$

nodes, placing a number of tiles, centered two tiles away, on the blackboard. The unit is then ordered to move to the last tile in the list, as mentioned above. The rest of the nodes in this strategy have no function. Therefore, this **Attack** node is most likely the crucial action in this strategy.

In-game Strategy

The three evolved strategies can be seen battling the default AI in a Wesnoth screenshot in Figure 7.7. The figures on the left are where the evolved strategy is starting on the south side of the map, and is controlling the units in red. Similarily, the figures on the right are where the evolved strategy controls the units in blue, who have started on the north side of the map. [3]

In Figure 7.7a, the evolved strategy $S_{\text{A-Gen162}}$ orders two units to move to highly defensible territory. The two units in the lower right are on ground and mountain tiles, providing a defence value superior to immediately adjacent tiles. As a reminder, the defence value of a tile is the chance that an attack towards that tile will miss. Therefore, these units have a lower chance than their enemy of being hit by attacks. This leads to an increase in the probability of winning the battle. In Figure 7.7c, $S_{\text{A-Gen176}}$ moves all of the units into a bottlenecked area. The unit defending the bottleneck has a much higher defence value then its attackers, again resulting in improved performance.

When these two strategies are being queried by teams which start in the northern area of the map, these strategies are counter-productive. The bottleneck is found up-left of the southern starting position, and is mirrored to be down-right of the northern starting position. Therefore, these strategies are commanding units to move away from the bottleneck. The results of the strategies can be seen in Figures 7.7b and 7.7d. Strategy $S_{\text{A-Gen162}}$ moves the units up-left to flat land near some shallow water in Figure 7.7b. Strategy $S_{\text{A-Gen176}}$ attempts to move the units even farther, but the movement node incorrectly handles the impassable tiles. This error leaves the unit standing in the shallow water, which has an extremely low defence rating.

3. The opposite team colours were used in Figure 7.7f

Thus, strategies that are optimized for the south side of the map perform extremely poorly on the north side.

S_{B} is a strategy similar to $S_{\text{A-Gen162}}$, and moves units to the same mountain tiles when it is on the south side. This can be seen in Figure 7.7e. However, on the north side of the map, units move to the high defence castle tiles, as well as next to the water tiles. This is seen in Figure 7.7f. Thus, enemy units are at a defence disadvantage when fighting on both sides of the map, making them more likely to miss attacks and not dodge away from incoming attacks. Again, this probabilistic advantage means S_{B} is more likely to win the battle.

(a) $S_{\text{A-Gen162}}$ - South side

(b) $S_{\text{A-Gen162}}$ - North side

(c) $S_{\text{A-Gen176}}$ - South side

(d) $S_{\text{A-Gen176}}$ - North side

(e) S_B - South side

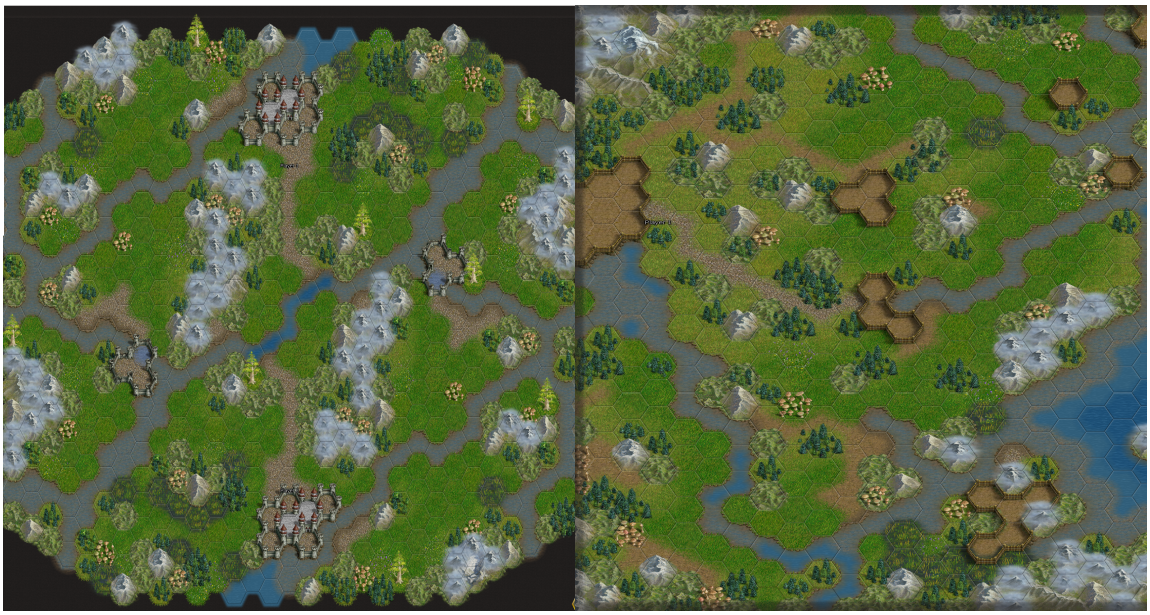
(f) S_B - North side[3]

Figure 7.7: In-game movements of evolved strategies

Alternative Maps

One concern is that the results above may only be valid on the map they were evolved on. Therefore, it is informative to examine the performance of an evolved strategy on different maps, and with differing numbers of units. The strategy selected was S_{B} as evolved above. It was evaluated on two other Wesnoth multiplayer maps, and with two numbers of units on each side. The *Freelands* map is seen in Figure 7.8a and was selected because it has almost no impassable terrain. The *Fallenstar Lake* map was selected because teams start at the mirrored west and east ends of the map, instead of the south and north ends of other maps. The west half of this symmetrical map is displayed in Figure 7.8b.

The results of the evaluation of S_{B} on these two maps are seen in Table 7.3.



(a) Freelands map (b) The west half of the Fallenstar Lake map

Figure 7.8: Two maps in Wesnoth

Table 7.3: S_{B} performance in different environments

(a) On south/west side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
Standard Map - 5 units	72	71	64	89
Fallenstar - 5 units	60	60	63	16
Fallenstar - 10 units	59	62	73	23
Freelands - 5 units	63	54	66	13
Freelands - 10 units	22	33	45	7

(b) On north/east side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
Standard Map - 5 units	57	60	57	36
Fallenstar - 5 units	44	38	36	17
Fallenstar - 10 units	39	41	24	15
Freelands - 5 units	39	43	34	6
Freelands - 10 units	80	70	53	15

The results show that the map does have a large impact on the performance of the strategy, as S_{B} does perform much poorer on maps other than the one it was evolved on. In particular, it was observed that S_{Default} did not have the issue of stopping all actions as often. This could be due to special 'castle' tiles, which are high-defence tiles which S_{Default} was observed ordering its units to move to. This may account for the especially low performance of S_{B} against S_{Default}, and should be removed in further tests. Modifying the number of units on the *Fallenstar* map does not seem to make a significant difference in terms of strategy performance. However, on the *Freelands* map, a large change happens when more units are added. Examination reveals that these extra units happen to move to tiles with high defence, thus gaining an advantage over the enemy team.

Another interesting observation is made from examining the maps themselves, and how strategies are taking advantage of terrain features. A common feature appears on a number of maps examined, including both maps above. Possibly for aesthetic reasons, the boundary of the map is comprised of hill or mountain tiles. These tiles provide a large defence value to units standing on them. As the studied evolved strategies tend to repeatedly move units in a single direction, units often end up on these border tiles. While this is a viable strategy for a team, it may be an unintended consequence of the map design. This simple strategy may also be delaying evolution, as strategies may be falling into a local optima of moving to the edge instead of evolving more robust tactics.

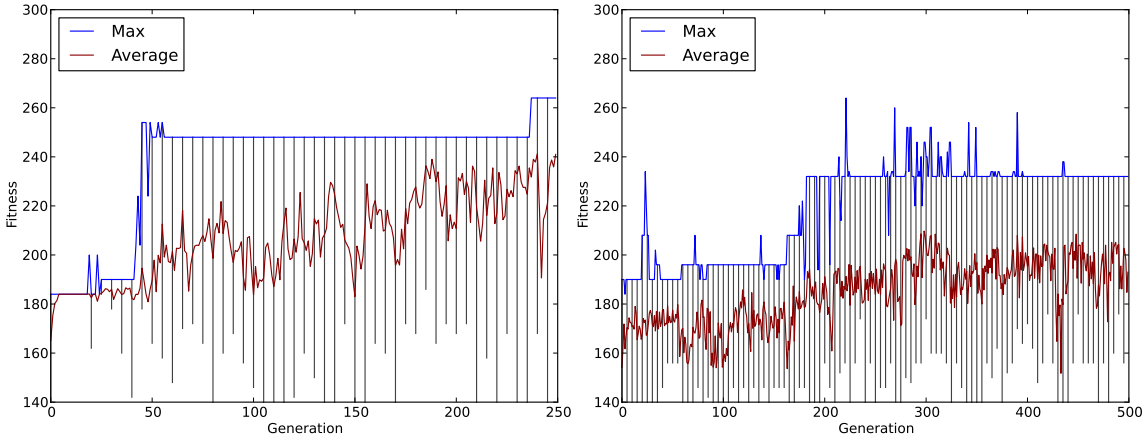
For the rest of this work, evolution and evaluation will take place on the original map as presented in the last chapter.

7.2.2 From Random Strategies

A common strategy in genetic algorithms and genetic programming is to seed the starting population with randomly initialized individuals [28]. This allows the evolutionary algorithm to start from solutions randomly placed in strategy space, as well as create potentially useful subtrees in the initial population. In this work, strategies were randomly created with 11 nodes each. The fitness graphs from two evolution runs will be examined, before the performance of two evolved strategies are discussed.

The fitness improvement for two evolution runs are shown in Figure 7.9. As with the evolution from the null strategy, Figure 7.9a is of the evolution process where the strategies undergoing evolution were fixed to the south side of the map, and Figure 7.9b is where evolving strategies were randomly switching sides. The opposing strategy was again $S_{Default}$.

The average and maximum fitness improve drastically throughout both the evolution runs. However, it should be noted that while the second evolution run was evolving for 500 generations, the average fitness was much lower. This is likely due to the strategy switching sides of the map randomly, which may lead to a strategy



(a) Fixed to south side of map

(b) Randomly switched side of map

Figure 7.9: Fitnesses for evolution from random strategies

being fit for one side of the map, but not the other. This suggests a new evaluation scheme where a strategy is evaluated on both sides before assigning a fitness. This switching may also be creating the fitness 'spikes' seen in the graph.

Another interesting note is that improvements in fitness occur at an earlier generation than for strategies evolved from the null strategy, as seen in Figure 7.4. This is most likely due to the random trees possessing more 'genetic material' to work with. As the random trees are combined together and mutated, there is a greater chance that an improvement in strategy strength will be made.

Table 7.4: Evaluation of evolved random strategies

(a) On south side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{RFS_1}	13	9	4	14
S_{RFS_2}	60	60	19	5
S_{RFS_3}	68	40	26	4
$S_{\text{RFixedSide}}$	84	79	69	91
S_{RRS_1}	24	21	10	14
S_{RRS_2}	54	54	14	6
S_{RRS_3}	14	40	26	4
$S_{\text{RRandomSide}}$	33	22	65	36

(b) On north side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{RFS_1}	14	12	4	14
S_{RFS_2}	22	12	21	11
S_{RFS_3}	33	17	21	11
$S_{\text{RFixedSide}}$	63	66	43	27
S_{RRS_1}	37	25	21	24
S_{RRS_2}	34	29	13	11
S_{RRS_3}	18	53	26	6
$S_{\text{RRandomSide}}$	38	50	32	53

Table 7.4a shows the results for the evolution run where the strategy was fixed to the south side of the map. S_{RFS_N} are three random strategies from the randomly-created population. Surprisingly, these strategies do quite well against the hand-built strategies. However, as expected, they do very poorly against the stronger S_{Defence} and S_{Default}. $S_{\text{RFixedSide}}$ is the strategy with the top fitness after evolution of this population. As can be seen from the results, $S_{\text{RFixedSide}}$ is a stronger strategy than

the initial population.[4] In particular, the 91 percent win rate against S_{Default} is quite remarkable. An examination indicates that this is due to a unit being moved to a mountain tile, where the defence value is quite high. S_{Default} attacks this unit and consequently loses a large amount of health. After this unit is killed, S_{Default} stops ordering units, and the turn limit of the game is reached. This result may not demonstrate a tactically strong strategy being evolved. However, it is a success in terms of the fitness function that the evolutionary process was optimizing for. Therefore, care must be taken to carefully define the fitness function so that only tactically strong strategies will be selected for. This will be further discussed later in this chapter.

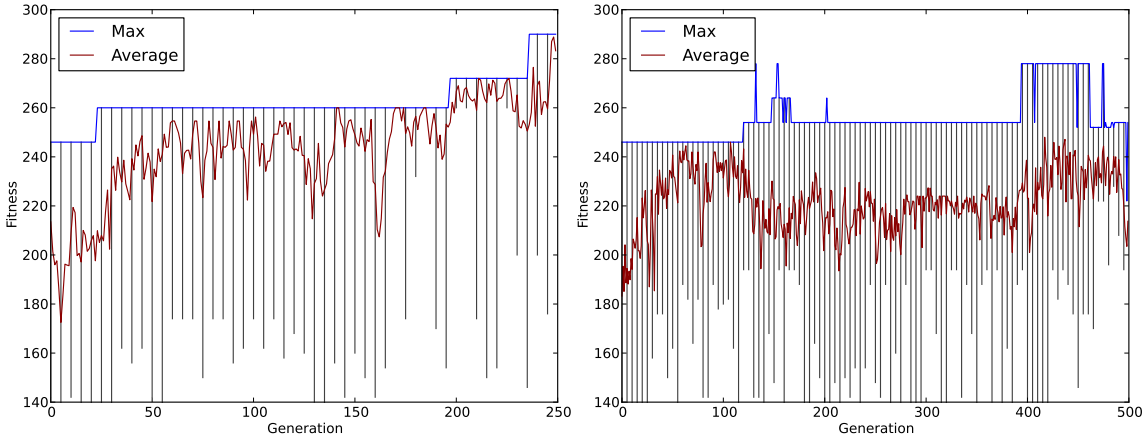
Table 7.4b shows results when the strategy randomly switched sides of the map during evolution. $S_{\text{RRandomSide}}$ does better than the three starting S_{RRS_N} strategies against S_{Default} by quite a significant degree. However, $S_{\text{RRandomSide}}$ does not match the performance of $S_{\text{RFixedSide}}$ against S_{Default}. Again, this may be due to changing sides during the evolutionary run decreasing the speed of the evolution rate. While a performance gain is seen against S_{Default} while on the north side of the map, a similar result is not seen against of the other strategies, which suggests some degree of specialization.

7.2.3 From Seeded Strategies

Another way of initializing the starting population is by seeding it with hand-built strategies [13]. The intention is to provide solutions in the solution-space that already perform well. The evolutionary algorithm will then search near these points, and can combine strong solutions together to potentially find even stronger solutions. As above, the graph of fitness improvement during evolution will be examined before discussion of the performance of the evolved strategies.

Figure 7.10a is the evolution over 250 generations of strategies which were fixed to the south side of the map. Figure 7.10b shows the fitnesses of the population

4. Results are statistically significant except for the improvement from S_{RFS_3} when battling S_{Closest}. The significance is rejected with $P = 0.05355$



(a) Evolution with fixed side (b) Evolution with random sides

Figure 7.10: Fitness results for Seeded evolution

of strategies that randomly switched sides throughout 500 generations of evolution. The overall trend of these graphs is quite interesting. The population was seeded with the hand-built strategies, including the strong $S_{Defence}$ strategy. This accounts for the high maximum fitness at the beginning of both runs. Over the generations, the other strategies in the population become more fit on average as elements from $S_{Defence}$ are combined in.

Even though the evolution run in Figure 7.10b is twice as long as the first, it is interesting to note that the maximum fitness found was lower. The strategy $S_{\text{SFixedSide}}$ evolved in this process has a fitness value of 290, while $S_{\text{SRandomSide}}$ has a fitness of 278. As above, this may be attributable to the effects of switching sides during evolution.

Table 7.5: Evaluation of evolved seeded strategies

(a) On south side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Closest}	43	38	21	35
S_{Weakest}	74	49	22	38
S_{Defence}	70	77	46	58
$S_{\text{SFixedSide}}$	60	66	46	68
$S_{\text{SRandomSide}}$	47	42	33	41

(b) On north side of map

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Closest}	43	26	30	24
S_{Weakest}	62	49	23	31
S_{Defence}	79	78	46	42
$S_{\text{SFixedSide}}$	74	49	57	27
$S_{\text{SRandomSide}}$	48	10	37	3

The results in Table 7.5 show the performance of the evolved strategies from both runs. Unexpectedly, the results of $S_{\text{SFixedSide}}$ are not significantly improved compared to S_{Defence}, one of the strategies $S_{\text{SFixedSide}}$ was evolved from. However, S_{Defence} is quite a strong strategy and is relatively complex. Therefore, it is understandable that random evolution would fail to find a significantly better strategy on a specific run. Further research is required to address this issue.

Examining the performance of $S_{\text{SRandomSide}}$, it is found to be significantly weaker than $S_{\text{SFixedSide}}$. As mentioned above, this may be due to the strategy randomly switching sides during evolution, preventing a true evaluation of a strategy's strength. This issue might prevent continual improvement until a fitness function is used that does takes the performance of the strategy on both sides of the map into account.

7.3 Fitness Function Selection

The choice of a fitness function may have a large impact on the rate of evolution and the resulting strength of the evolved strategy. Therefore, this section will compare four different fitness functions with the fitness function defined above. The fitness functions will be described along with a short rationale before the results are discussed.

In the fitness functions below, N will signify the number of turns in the battle. The sum of team health on the turn i will be denoted as S^i. The team number will be placed in subscript as required.

7.3.1 Health Difference

The health difference fitness function is used above in the main results. It calculates the difference in the sum of team health at the end of the battle. It was previously defined in Section 6.2.4 and is presented again in Equation 7.1.

$$\text{Fitness} = S_1^N - S_2^N \tag{7.1}$$

7.3.2 Average Health

This function was selected for experimentation because of its similarity to the health difference fitness function. The calculation can be seen in Equation 7.2 and is simply the average of S over all turns.

$$\text{Fitness} = \left(\sum_{i=1}^{N} S^i \right) / N \tag{7.2}$$

7.3.3 Competitive

Another fitness function examined also takes the enemy team's health over the entire battle into account. It is calculated as the sum of S for team 1 minus the sum of S for team 2 over all turns, as seen in Equation 7.3.

$$\text{Fitness} = \left(\sum_{i=0}^{N} S_1^i\right) - \left(\sum_{i=0}^{N} S_2^i\right) \tag{7.3}$$

7.3.4 Weighted Competitive

Similar to some fitness functions discussed in past literature, weights can also be placed in the fitness function. This fitness function is a copy of Equation 7.3 with the addition of a doubling of the second term. The aim of this weighting is to encourage strategies to deal damage to their enemy even at the expense of their own health. This fitness function can be seen in Equation 7.4.

$$\text{Fitness} = \left(\sum_{i=0}^{N} S_1^i\right) - \left(2 * \sum_{i=0}^{N} S_2^i\right) \tag{7.4}$$

7.3.5 Win/Lose

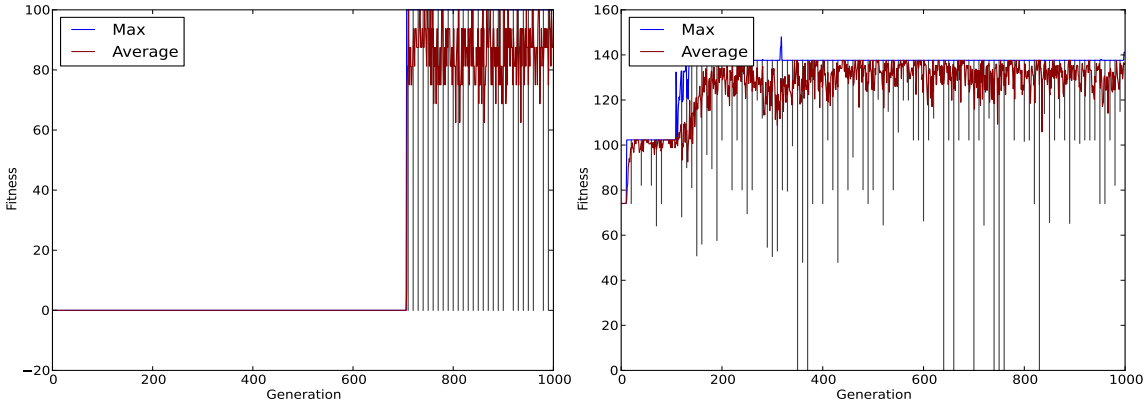
This fitness function simply rewards strategies that won with a fitness value of 100, and strategies that lost with a fitness value of 0. With this fitness function, strategies can no longer be ranked by selection operators, preventing the evolutionary algorithm from performing a guided search around the solution space.

7.3.6 Results

To order to examine these fitness functions, strategies were evolved from S_{Null} for 1000 generations using each fitness function. As a comparison, the health difference function from above was also used as the basis for a control evolution run. Graphs of the average and maximum fitnesses are found in Figure 7.11 for all fitness functions. The different fitness functions have differing scales and thus cannot be normalized. However, the shape of the fitness graph still presents insights. Following a brief discussion of the fitness figures, the performance of some evolved strategies will be presented.

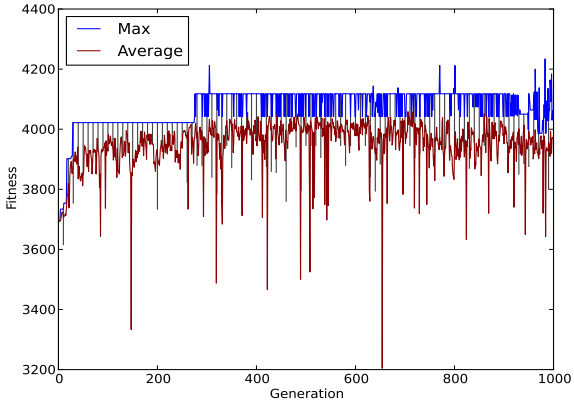
An increase in maximum and average fitness can be seen in all evolution runs. Even without the fitness value guiding the search, the win/lose fitness function in

Figure 7.11a did find a strategy that can win a battle, though over 10400 strategies were evaluated before this occurred. In comparing the figures, the health difference fitness function used in earlier results has a fitness trend that is much flatter than the first three fitness functions. While this may be a consequence of the random search, it may also suggest an insight into fitness function selection. The health difference fitness function only provides a measurement of what has happened on the last turn of the algorithm. In contrast, the average health, competitive, and weighted competitive fitness functions give more information about the battle. A strategy that has performed poorly for most turns will receive a lower fitness value. This may be useful information for the genetic algorithm to rank strategies more effectively, which could speed up evolution.

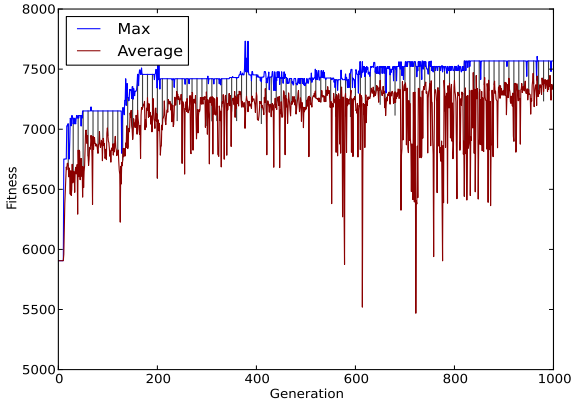


(a) Win/Lose

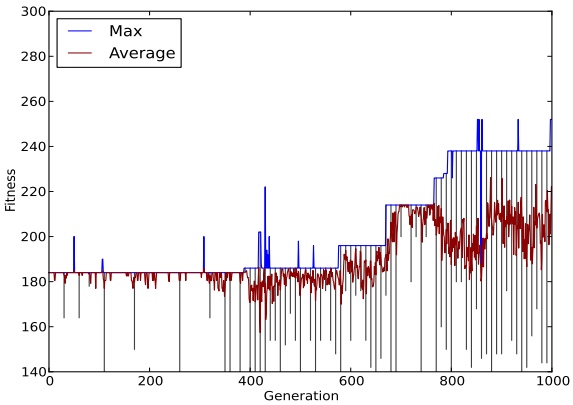
(b) Average Health



(c) Competitive



(d) Weighted Competitive



(e) Health Difference

Figure 7.11: Average and maximum fitness value graphs for five fitness functions

Table 7.6 shows the win percentage of the evolved strategies against the hand-built strategies and the default AI. For simplicity, these results were collected from strategies evaluated alternately between the north and south sides of the map. Thus there is a single win percentage. The first observation to be made is that all evolved strategies performed much better against the hand-built strategies. However, their performance against the default AI is less convincing. In particular, the $S_{\text{Avg. Health}}$ and $S_{\text{Health Diff.}}$ failed to increase their performance against S_{Default} significantly.

Table 7.6: Performance of strategies evolved with different fitness functions

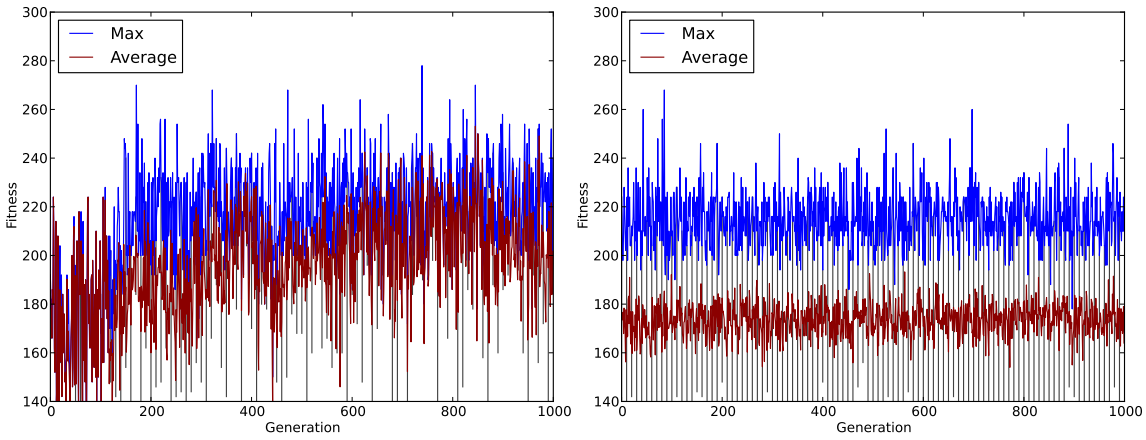
	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Null}	12	10	3	14
$S_{\text{Win/Lose}}$	31	27	34	37
$S_{\text{Avg. Health}}$	40	29	25	16
$S_{\text{Compet.}}$	41	38	33	30
$S_{\text{W. Compet.}}$	50	38	27	21
$S_{\text{Health Diff.}}$	55	42	25	16

Another interesting result is that the $S_{\text{Win/Lose}}$ developed a surprisingly strong strategy, despite an information-poor fitness function. Indeed, its evolved strategy seems to be relatively effective against S_{Default}. Further examination of its performance in-game indicates that it randomly evolved to move onto a tile with high defence value. While unexpected, it is a consequence of a stochastic algorithm that strong solutions can be randomly found.

The above results fail to indicate a superior fitness function to employ. However, it is interesting how the competitive fitness functions seem to quickly improve the population within a few generations, while the health difference function improves its solutions later in the evolution run. Perhaps each fitness function is superior at different points in the evolutionary algorithm. Future work may consider an algorithm that dynamically switches fitness functions during evolution.

7.4 Effect of Randomness

These final results will show the impact of changing the random generator seed provided to Wesnoth during the evolution of strategies. All experiments presented above kept the seed constant during evolution, and only varied it during evaluation of the strategies. This section presents two evolution runs. In the first, the seed was varied every generation of the evolutionary algorithm. In the second, the seed was varied for each battle when assigning a fitness function to a strategy. As before, the average and maximum fitness graphs will be presented before examining the performance of evolved strategies.



(a) Varying random seed per generation (b) Varying random seed per game

Figure 7.12: Evolution fitness when varying random seed

Figure 7.12 shows the fitness graphs for the two evolution runs. From these figures, it is clear that varying the seed makes the evolution process a much noisier process. When varied per generation, evolution does make improvements in both the maximum and average fitness of the population. This suggests that the effects of randomness are not large enough to completely deter the evolutionary process. In contrast, when the random seed is changed every game, it is not clear if any evolution has occurred at all. This is shown in Figure 7.12b with almost flat fitness values over

the evolutionary run. Thus, it can be concluded that this level of randomness is detrimental to the evolutionary algorithm.

Table 7.7: Performance of strategies evolved with varying random seeds

	S_{Closest}	S_{Weakest}	S_{Defence}	S_{Default}
S_{Null}	12	10	3	14
$S_{\text{Varied per Game}}$	12	10	3	14
$S_{\text{Varied per Gen.}}$	27	28	28	47
$S_{\text{A-Gen176}}$	51	46	23	57

The results in Table 7.7 correlate to the graphs in Figure 7.12. $S_{\text{Varied per Game}}$ shows no improvement at all over S_{Null}, while $S_{\text{Varied per Gen.}}$ shows improved results. Results for $S_{\text{A-Gen176}}$ are also present in the table to represent an evolution result where the seed was kept the same throughout evolution. While $S_{\text{Varied per Gen.}}$ has similar performance to $S_{\text{A-Gen176}}$ against S_{Defence} and S_{Default}, $S_{\text{Varied per Gen.}}$ is not as powerful against the other hand-built strategies. Thus, these results suggest that the random seed should be kept constant in order to aid evolution. Any possible benefits of varying the seed per generation are unclear at this time.

Chapter 8
Conclusions and Future Work

This chapter will present concluding remarks, focusing on areas of interest and difficulties encountered. In particular, three key areas of this work will be discussed. The first area is the relatively straightforward adaptation of the behaviour tree formalism to the domain, which highlights the flexibility and power of this formalism. Following this, issues concerning the evolutionary algorithm will be examined. Finally, the strategies produced through evolution will be discussed, along with observations as to their performance in this domain. Potential areas of further study will be presented throughout this chapter.

8.1 Behaviour Tree Formalism

Overall, the behaviour tree formalism worked harmoniously with the domain and the evolutionary algorithm. The formalism is highly flexible, human-readable, and can represent an artificial intelligence at any abstraction level. The usefulness of behaviour trees is reflected in the growing number of adopters in the video game industry and academic literature.

Domain-specific modifications to the formalism, such as the definition of appropriate action nodes and any blackboard-like structures, does require developer effort to be successful. In this work, action nodes were sourced from examining other

strategies and referencing domain knowledge. These action nodes were then used to hand-build strategies, which were surprisingly strong against the default AI. The list-based format of data and the black-board structure were created to provide the evolutionary algorithm the flexibility to recombine and modify strategies without producing an invalid strategy. This was done to avoid wasted computation and prevent invalid subtrees from spreading throughout the population. Another key objective was to maintain the ability to use the genetic operators defined for evolutionary programming. The elegance and modularity of these operators aided in the rapid development of the evolutionary algorithm.

8.1.1 Future Improvements

Future work related to the behaviour tree formalism may consider the actions used. In particular, it would be beneficial to have domain experts create these actions, as well as sample behaviour trees. An open question is whether more node types would be a help or hindrance to the evolutionary algorithm for this domain. For example, a **Sort** node was defined with an attribute to control the sorting behaviour. This node type could be split into separate types such as a **SortHealth** or **SortDistance** node. This may complicate the solution search space, or could potentially increase the success of recombination operators. Explicit condition nodes also were not employed in this work. While they are often used in practice in the behaviour tree formalism, they may also increase the difficulty of evolving strong strategies. This may occur by increasing the number of semantically invalid trees, or by again increasing the number of solutions in the search space.

8.2 Evolutionary Algorithm

The fitness graphs in the last chapter show that the evolutionary algorithm does successfully evolve strategies, according to the fitnesses given by the health difference fitness function. The performance of strategies evolved from the null strategy and random strategies also demonstrate statistically significant improvements. One

result seen is that the maximum and average fitness values tends to increase faster during an evolution run beginning from random strategies over other initializations. As mentioned before, this may be due to a random tree's greater chance of strategy improvement through recombination and mutation, as individuals in the initial population may possess tactically strong subtrees. The results with the evolution run seeded with the hand-built strategies are less encouraging. As S_{Defence} is already a strong performing strategy, it may be a local optima in the solution space. This may be overcome by improving the evolutionary algorithm and the fitness function to help find a stronger strategy. It may also be beneficial to change the search space by defining new nodes that a strategy could employ.

The health difference fitness function used in the results presented may not be the optimal one for this domain. Therefore, a number of evolution experiments were conducted on different fitness functions in order to examine trends. The results suggest that fitness functions which provide information on every turn of a battle may show fitness improvement in evolution at an earlier generation than other fitness functions. Therefore, fitness functions of this type may be more appropriate to use in the beginning of an evolution run. The win/lose fitness function was also examined. A winning strategy was found, despite the low information search performed. An information-theory approach to fitness functions may offer a metric for determining fitness function performance. This would be of use in domains such as the one studied, where there is not a natural or obvious fitness function to choose.

The role of randomness in the evolutionary algorithm was also discussed, as battles in Wesnoth depend heavily on the random number generator. The effects of varying the random seed once per evolutionary generation and for every game were examined. In the case of varying the seed per generation, the fitness improvement was quite noisy but did not appear to impede the evolution process significantly. However, further work is required to determine the full effects. In contrast, varying

the seed per game prevented the evolutionary algorithm from making any fitness improvements at all. [1]

8.2.1 Future Improvements

Fitness functions found in the literature could also be applicable in this domain. For instance, the weighted competitive function defined in the last chapter could be improved by adding a dynamically changing weight [50]. In this work by Priesterjahn et al., the weight decreases during the evolution of strategies in order to steadily deprioritize damage done to the opponent. Another fitness function may measure if each unit on the team was effectively used to deal damage, thereby measuring team coordination [55].

The performance of the evolution algorithm may be significantly increased by the creation of a more robust evaluation process. As discussed in the previous chapter, the performance of the evolved strategies were dependent on the terrain configuration of the map and the starting location of units on a team. Therefore, each strategy should be evaluated multiple times on different maps and different unit placements in order to assign a fitness value. This may be prohibitively computationally expensive unless advanced heuristics are used. For instance, the solution space may be analyzed and separated into distinct regions of performance. This would allow evolution to be guided more intelligently, or to avoid evaluating strategies that fall in a poor region of the space [5, 41].

The use of 'training camps' may also provide a method to evolve strong strategies, where sub-problems in the domain are defined for strategies to solve [2]. In this work, genetic programming trees are evolved for each particular sub-problem. Trees are then combined together to form a larger strategy that can solve the entire problem. Results show that a stronger strategy may be obtained with this method than evolving a whole tree at once.

Another method of evolving strong solutions in a domain is through the use of *co-evolution* [3, 52]. This method involves evolving two populations at once,

1. This issue severely impacted preliminary experiments.

where the populations are in direct competition with each other. As an example, instead of evaluating strategies against the default AI in the evaluation stage of this genetic algorithm, battles would be between two strategy populations being evolved simultaneously. The intention is to use competition between populations to evolve strong strategies as populations may evolve effective strategies and counter-strategies during the evolution run.

8.3 Evolved Strategies

The results in the previous chapter showed that evolution of strategies was occurring, and that the performance of evolved strategies was increased by a statistically significant amount. This was particularly noted in the evolution run starting from the null strategy. However, further examination of the strategies revealed that these performance increases were highly dependent on map terrain features and tactically poor orders. This result is at once encouraging and disappointing. It is encouraging because the evolutionary algorithm did produce strategies that did tend to win against other strategies. This was mostly done by ordering units to repeatedly move in one direction, which is effective for some maps. In particular, it is interesting how this tactic exploits human map designers placing mountains and hills at the edge of the map. This may have been done for visual effect without regard to this emergent behaviour.

Initial expectations for this work were that strategies would develop stronger tactics, possibly involving recognizing high defence terrain. The strategies produced had a poor tendency to generalize, as seen from their application to different maps or a different side of the same map. As discussed above, a more robust evaluation strategy may be needed in order to produce stronger strategies.

8.4 Concluding Remarks

In this work, strategies have successfully been evolved for a turn-based game, with each evolutionary run taking less than 110 minutes each. Many different elements of

the entire evolutionary algorithm have been discussed, from strategy representation to evaluation to modification. This discussion was aided by the presentation of an example problem, as well as references from the literature.

The behaviour tree formalism has been introduced as a representation of artificial intelligence used within the video game industry and cited for its flexibility, power, and human-readability. Modifications made to this formalism to support this domain were discussed, along with a rationale as to why this formalism is well-suited to be used for evolutionary computing experiments.

Results of various evolutionary runs were presented, each focusing on a different parameter of evolution. Strategies were evaluated in the game Battle for Wesnoth in order to be assigned a fitness value. This fitness value was then used in turn to define a performance metric for strategies that is robust to randomness. Strategies were examined in-game and graphically in order to determine their tactics and complexity. A strategy was evaluated in different environments to determine its generality.

Experiments that focused on different aspects of the evolutionary algorithm provided a discussion of potential improvements in a number of parameters such as the fitness function, evaluation regime, and random seed variation, as well as a number of directions for future work.

Bibliography

[1] Movies and War Game as Aids to Our Navy. *The New York Times*, November 5, 1916.

[2] Atif M Alhejali and Simon M Lucas. Using a training camp with Genetic Programming to evolve Ms Pac-Man agents. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 118–125. IEEE, 2011.

[3] Peter J Angeline and Jordan B Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 264–270. San Mateo, California, 1993.

[4] Nirvana S Antonio, MGF Costa, R Padilla, et al. Optimization of an evaluation function of the 4-sided dominoes game using a genetic algorithm. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 24–30. IEEE, 2011.

[5] G Avigad, E Eisenstadt, and M Weiss Cohen. Optimal strategies for multi objective games and their search by evolutionary multi objective optimization. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 166–173. IEEE, 2011.

[6] Wolfgang Banzhaf. The "molecular" traveling salesman. *Biological Cybernetics*, 64(1):7–14, 1990.

[7] Alex Champandard, David Hernandez Cerpa, and Michael Dawe. Behavior Trees: Three Ways of Cultivating Strong AI. In *Game Developers Conference*, 2010.

[8] Alex J. Champandard. Understanding the Second-Generation of Behavior Trees. `http://aigamedev.com/insider/tutorial/second-generation-bt/`, February 2012.

[9] Michael Cook and Simon Colton. Multi-faceted evolution of simple arcade games. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 289–296. IEEE, 2011.

[10] Kenneth Alan De Jong. *Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[11] Christopher Dragert, Jörg Kienzle, and Clark Verbrugge. Reusable components for artificial intelligence in computer games. In *Games and Software Engineering (GAS), 2012 2nd International Workshop on*, pages 35–41. IEEE, 2012.

[12] Antonio Fernández-Ares, Antonio M Mora, Juan J Merelo, Pablo García-Sánchez, and Carlos M Fernandes. Optimizing player behavior in a real-time strategy game using evolutionary algorithms. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 2017–2024. IEEE, 2011.

[13] Gabriel J. Ferrer and Worthy N. Martin. Using genetic programming to evolve board evaluation functions. In *Evolutionary Computation, 1995., IEEE International Conference on*, volume 2, pages 747–752. IEEE, 1995.

[14] Gonzalo Flórez-Puga, Marco Gomez-Martin, Belen Diaz-Agudo, and Pedro Gonzalez-Calero. Dynamic expansion of behaviour trees. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference. AAAI Press*, pages 36–41, 2008.

[15] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional, 1989.

[16] Roderich Gross, Keno Albrecht, Wolfgang Kantschik, and Wolfgang Banzhaf. Evolving chess playing programs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747. Morgan Kaufmann Publishers Inc., 2002.

[17] Erin Jonathan Hastings, Ratan K Guha, and Kenneth O Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245, 2009.

[18] Ami Hauptman and Moshe Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. *Genetic Programming*, pages 142–143, 2005.

[19] Ami Hauptman and Moshe Sipper. Emergence of complex strategies in the evolution of chess endgame players. *Advances in Complex Systems*, 10(supp01):35–59, 2007.

[20] Chris Hecker. Spore Behavior Tree Docs. `http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs`, April 2009.

[21] John H Holland. *Adaptation in natural and artificial systems. an introductory analysis with applications to biology, control and artificial intelligence*, volume 1. Ann Arbor: University of Michigan Press, 1975.

[22] Ting Hu and Wolfgang Banzhaf. The role of population size in rate of evolution in genetic programming. *Genetic Programming*, pages 85–96, 2009.

[23] Ting Hu, Yuanzhu Chen, and Wolfgang Banzhaf. WiMAX network planning using adaptive-population-size genetic algorithm. *Applications of Evolutionary Computation*, pages 31–40, 2010.

[24] Damian Isla. Handling Complexity in the Halo 2 AI. `http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php`, March 2005.

[25] Graham Kendall and Kristian Spoerer. Scripting the game of Lemmings with a genetic algorithm. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 117–124. IEEE, 2004.

[26] Graham Kendall and Glenn Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 995–1002. IEEE, 2001.

[27] Bjrn Knafla. Introduction to Behavior Trees. `http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/`, February 2011.

[28] John R. Koza. Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems. Computer Science Department STAN-CS-90-1314, Stanford University, June 1990. 1990.

[29] John R Koza, Forrest H Bennett III, David Andre, and Martin A Keane. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 1–10. IEEE, 1996.

[30] John Krajewski. Creating All Humans: A Data-Driven AI Framework for Open Game Worlds. `http://www.gamasutra.com/view/feature/130279/creating_all_humans_a_datadriven_.php`, February 2009.

[31] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[32] Chong-U Lim. An A.I. Player for DEFCON: An Evolutionary Approach Using Behavior Trees. Imperial College, London, June 2009.

[33] Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game DEFCON. In *Applications of Evolutionary Computation*, pages 100–110. Springer, 2010.

[34] Sushil J Louis and Chris Miles. Combining case-based memory with genetic algorithm search for competent game AI. In *ICCBR Workshops*, pages 193–205, 2005.

[35] David Luciew, Janet Mulkern, and Ronald Punako. Finding the Truth: Interview and Interrogation Training Simulations. In *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, volume 2011. NTSA, 2011.

[36] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. *RoboCup-97: Robot soccer world cup I*, pages 398–411, 1998.

[37] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Modelling and evaluation of complex scenarios with the Strategy Game Description Language. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 174–181. IEEE, 2011.

[38] Chris Miles and Sushil J Louis. Co-evolving real-time strategy game playing influence map trees with genetic algorithms. In *Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon*, pages 0–999, 2006.

[39] Ian Millington and John Funge. *Artificial Intelligence for Games*, chapter 5, pages 334–371. Morgan Kaufmann, San Francisco, 2006.

[40] Antonio M Mora, Antonio Fernández-Ares, Juan J Merelo, Pablo Garcıa-Sánchez, and Carlos M Fernandes. Effect of Noisy Fitness in Real-Time Strategy Games Player Behaviour Optimisation Using Evolutionary Algorithms. *J. Comput. Sci. & Technol*, 27(5), 2012.

[41] Jason Noble. Finding robust Texas Hold'em poker strategies using Pareto coevolution and deterministic crowding. In M. A. Wani, editor, *Proceedings of the 2002 International Conference on Machine Learning and Applications (ICMLA02)*, pages 233–239. CSREA Press, 2002.

[42] Sergio Ocio. Adapting AI Behaviors To Players in Driver San Francisco - Hinted-Execution Behavior Trees. In *The Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012.

[43] Ricardo Palma, Pedro A González-Calero, Marco A Gómez-Martın, and Pedro P Gómez-Martın. Extending case-based planning with behavior trees. In *Proceedings of FLAIRS*, pages 407–412, 2011.

[44] Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Reactiveness and navigation in computer games: Different needs, different approaches. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 273–280. IEEE, 2011.

[45] Jacques Périaux, HQ Chen, B Mantel, M Sefrioui, and HT Sui. Combining game theory and genetic algorithms with application to DDM-nozzle optimization problems. *Finite Elements in Analysis and Design*, 37(5):417–429, 2001.

[46] F. Pezzella, G. Morganti, and G. Ciaschetti. A genetic algorithm for the Flexible Job-shop Scheduling Problem. *Computers & Operations Research*, 35(10):3202–3212, 2008.

[47] Ricardo Pillosu. Coordinating Agents with Behavior Trees. In *Paris Game AI Conference*, 2009.

[48] Marc Ponsen. Improving adaptive game AI with evolutionary learning. Master's thesis, Delft University of Technology, 2004.

[49] Marc Ponsen, Héctor Muñoz-Avila, Pieter Spronck, and David W Aha. Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27(3):75, 2006.

[50] Steffen Priesterjahn, Oliver Kramer, Alexander Weimer, and Andreas Goebels. Evolution of human-competitive agents in modern computer games. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 777–784. IEEE, 2006.

[51] Charles W Richter, Jr and Gerald B Sheblé. Genetic algorithm evolution of utility bidding strategies for the competitive marketplace. *Power Systems, IEEE Transactions on*, 13(1):256–261, 1998.

[52] Christopher D Rosin and Richard K Belew. Methods for competitive co-evolution: Finding opponents worth beating. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 373–380. San Francisco, CA, 1995.

[53] Claude E Shannon. XXII. Programming a computer for playing chess. *Philosophical magazine*, 41(314):256–275, 1950.

[54] Karl Sims. Evolving 3D morphology and behavior by competition. *Artificial life*, 1(4):353–372, 1994.

[55] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Online adaptation of game opponent AI in simulation and in practice. In *Proceedings of the 4^{th} International Conference on Intelligent Games and Simulation (GAME-ON 2003)*, pages 93–100, 2003.

[56] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. The NERO Real-time Video Games. Technical Report UT-AI-TR-04-312, Department of Computer Sciences, University of Texas at Austin, 2004.

[57] Chuen-Tsai Sun, Ying-Hong Liao, Jing-Yi Lu, and Fu-May Zheng. Genetic algorithm learning in game playing with multiple coaches. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 239–243. IEEE, 1994.

[58] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[59] Paul Tozour. The evolution of game AI. *AI game programming wisdom*, 1:3–15, 2002.

[60] Paul Tozour. Making Designers Obsolete? Evolution in Game Design. `http://aigamedev.com/open/interview/evolution-in-cityconquest/`, February 2012.

[61] Jonathan Tremblay and Clark Verbrugge. Adaptive Companions in FPS Games. In *Proceedings of the 8th Internation Conference on Foundations of Digital Games*, pages 229–236, 2013.

[62] Matthew Bartschi Wall. *A Genetic Algorithm for Resource-Constrained Scheduling*. PhD thesis, Massachusetts Institute of Technology, 1996.

[63] David White. Luck in Wesnoth: Rationale. `http://forums.wesnoth.org/viewtopic.php?f=6&t=21317&start=0&st=0&sk=t&sd=a`, May 2008.