Automatic Game Parameter Tuning using General Video Game Agents

Kamolwan KUNANUSONT

A dissertation submitted for the degree of Master by Dissertation

in Computer Science

School of Computer Science and Electronic Engineering

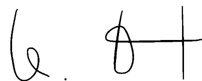
University of Essex

April 2018

Declaration of Authorship

I, Kamolwan KUNANUSONT, declare that this dissertation titled, "Automatic Game Parameter Tuning using General Video Game Agents" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this dissertation has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.
- I have acknowledged all main sources of help.
- Where the dissertation is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 19 April 2018

UNIVERSITY OF ESSEX

Abstract

School of Computer Science and Electronic Engineering

Master by Dissertation

Automatic Game Parameter Tuning using General Video Game Agents

by Kamolwan KUNANUSONT

Automatic Game Design is a subfield of Game Artificial Intelligence that aims to study the usage of AI algorithms for assisting in game design tasks. This dissertation presents a research work in this field, focusing on applying an evolutionary algorithm to video game parameterization. The task we are interested in is player experience. N-Tuple Bandit Evolutionary Algorithm (NTBEA) is an evolutionary algorithm that was recently proposed and successfully applied in game parameterization in a simple domain, which is the first experiment included in this project. To further investigating its ability in evolving game parameters, We applied NTBEA to evolve parameter sets for three General Video Game AI (GVGAI) games, because GVGAI has variety supplies of video games in different types and the framework has already been prepared for parameterization. 9 positive increasing functions were picked as target functions as representations of the player expected score trends. Our initial assumption was that the evolved games should provide the game environments that allow players to obtain score in the same trend as one of these functions. The experiment results confirm this for some functions, and prove that the NTBEA is very much capable of evolving GVGAI games to satisfy this task.

Acknowledgements

I believe it is not an exaggeration to state that many persons would be delighted once I have announced that this long-journey (very true, as it was beginning in the UK and finishing in Thailand) of reading, experimenting and writing is 'finally' finished. I have received a lot of supports from many individuals, ranging from those who are the experts of this field, to those who does not have any slightest ideas of what this dissertation is about.

I would like to give the best honourable thanks to Dr. Diego Pérez Liébana, whose helps and supports were, and have been, far beyond the formal technical supervision. I do not have enough expertise in English language to express how much I appreciate all the helps I have received from him, throughout all of my lows (often) and highs (rarely) moments during the degree. The best I can say here is that I will not hesitate to stubbornly repeatedly recommend him to anyone who is choosing a supervisor with him being one of their candidates, because he has been not just a supervisor in my viewpoint, but also a big-brother-like figure.

Special thanks would be given to Prof. Simon Mark Lucas, who was my main supervisor from January to July 2017, as he was the one who initiated the work that has become the first experiment of this research. Also, it was him who had secured me the funding for 2017 academic year and a part-time work so that I could support my daily life expenses during my study.

As we have mentioned Simon and the funding, I would like to send a sincerely thanks to the Visteon Corporation UK, for supporting my study fees for the 2017 academic year. Additionally, an extraordinary thanks would be to Claire Lewis, for her understanding of my complicated academic situation and to her impressive ability to sort out the best options for both her company and myself.

Staffs in CSEE have also been very supportive, especially Claire Harvey and Dr. Steve Sangwine. Claire Harvey always be patient and kindly helpful for all

complicated situations of myself that required her advices. Dr. Steve Sangwine generosity in offering to name himself as my main supervisor after Diego has left, to assist me through the completion state, has earned a sincere thanks here. I would like to also thanks Dr. Luca Citi and Dr. Michael Fairbank who had agreed to hold a quick board meeting for changing my degree status on that day, as it was an important step leading to this completion.

Family and partner have been heart-warmly supportive. They always encouraged me to keep on with the works every time I felt discourage. I would like to dedicate my deepest thanks to them as I would unable to finish this work without their backing. For Summer, who always be there for me despite the 6 (sometimes 7) hours difference and a number of countries (though Russia takes up the most part) between us, I would like to give her my widest and happiest smile, in the hope that it would bring hers out as well.

Last and definitely not least, two persons that had been tolerate with my unstable emotional status during this writing the most were Raluca and Jon. If they were annoyed by that (in which I am sure they were), they never showed a slightest sign of it, instead they kept on encouraging me and complimenting me for every few hundred words I had added into the text. I would like to thank them quietly here without telling them in persons. For that we Thai people believe that chanting people behind their backs is the sincerest action you can ever express to honour them.

If your name has not been mentioned earlier, I would like to thank you here for being interested in this research work. Please accept my pseudo-handshaking via this text since I could not offer it in person at the moment you are reading this. I am very much hoping that once the opportunities present, I will be able to do it to you in person.

Many Thanks,

Kamolwan KUNANUSONT

Contents

Declaration of Authorship **iii**

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

2 Literature Review **7**

2.1 Automatic Game Design 7

2.1.1 Early Attempts: Proof of Concept 8

2.1.2 Auto-Generate Full Game 9

2.1.3 Auto-Generating Maps or Levels for Games 11

2.1.4 Auto-tuning Game Parameters 17

2.2 General Video Game Artificial Intelligence (GVGAI) 19

2.2.1 Motivation and Origin 19

2.2.2 Competition Tracks 21

2.2.3 GVGAI for Game Design 22

2.3 Evolutionary Algorithms 23

2.3.1 General Improvements 25

2.3.2 Applications in the Games Domain 31

3 Background **39**

3.1 Space Battle 39

3.2 GVGAI Framework 41

3.2.1 Video Game Description Language 42
3.2.2 Selected Games 46
3.2.3 Controllers 53
3.3 Evolutionary Algorithms 60
3.3.1 Random Mutation Hill Climber (RMHC) 60
3.3.2 Biased Mutation RMHC 61
3.3.3 N-Tuple Bandit Evolutionary Algorithm (NTBEA) 62

4 Approaches 67
4.1 Space Battle Evolved 67
4.1.1 Game Rules & Space 67
4.1.2 Fitness Calculation 68
4.2 GVGAI Game Rules & Space 70
4.2.1 Seaquest 70
4.2.2 Waves 72
4.2.3 Defender 76
4.2.4 Fitness Calculation 79
Target Score Functions 80
Loss Calculation & Fitness Value 86
Fitness Calculation Summary 89

5 Experiments & Results 91
5.1 Space Battle Evolved 91
5.2 GVGAI Games 95
5.2.1 Evolving Game Parameters 97
Different Games 97
Biasing Loss Calculation 102
Functions and Parameters 107
5.2.2 Validating Evolved Games 112
Best Individual Selection Criteria 113

Different Games . . . 115
Normal NRMSE and Biased NRMSE . . . 119
Functions and Parameters . . . 122
5.2.3 Evolved Games . . . 127

6 Summary & Conclusion **129**
6.1 Summary . . . 129
6.2 Conclusion . . . 133

7 Future Works **139**
7.1 Further Analysis . . . 139
7.2 Further Experiment . . . 140
7.3 Extensions and Improvements . . . 141

References **143**

List of Figures

2.1 A game evolved by [1], taken from page 4 of the publication . . . 9
2.2 Yavalath puzzle, taken from page 12 of [2] 10
2.3 Example of mazes generated by [3] taken from page 8 of the publication 12
2.4 Comparison between the original and clone version of Cut the Rope game used in [20], taken from page 2 of the publication . . . 16
2.5 Flappy bird parameter space, taken from page 6 of [23] 18
2.6 Front (top left), back (top right), middle (bottom left) areas and neural network structure (bottom right) as applied in [76]. The image is taken from page 2 of the publication 32
2.7 4 Pacman maps with increasing complexity from left to right, employed in [79]. The image is taken from page 4 of the publication 34

3.1 Space Battle Game Screenshot 40
3.2 Space Battle Evolved 42
3.3 Single-player Aliens level description language and screenshots, level 0 and 4 44
3.4 Seaquest *SpriteSet* game description 48
3.5 Seaquest *InteractionSet* game description 48
3.6 Seaquest *TerminationSet* game description 48
3.7 Waves *SpriteSet* game description 49
3.8 Waves *InstructionSet* game description 50
3.9 Waves *TerminationSet* game description 51

3.10 Defender *SpriteSet* game description 52
3.11 Defender *InteractionSet* game description 53
3.12 Defender *TerminationSet* game description 53
3.13 MCTS searching steps, taken from page 6 of [41] 60

4.1 Modified Seaquest *SpriteSet* game description 72
4.2 Modified Seaquest *InstructionSet* game description 73
4.3 Seaquest *ParameterSet* game description 74
4.4 Modified Waves *SpriteSet* game description 75
4.5 Modified Waves *InteractionSet* game description 76
4.6 Waves *ParameterSet* game description 77
4.7 Modified Defender *SpriteSet* game description 78
4.8 Modified Defender *InteractionSet* game description 79
4.9 Defender *ParameterSet* game description 80
4.10 Linear target functions 81
4.11 Linear piecewise target functions 83
4.12 Sigmoid function 84
4.13 Shifted sigmoid target functions 85
4.14 Logarithm and exponential target functions 86
4.15 All target function slope 87

5.2 Screenshots of the 6 designed games evaluated by human players and their feedback. 95
5.3 Average fitness throughout evolutions for $y = 0.2x$ on the games of this study 99
5.4 Average score trend and score difference throughout evolutions for $y = 0.2x$ on the games of this study. Each plot shows different trends, averages taken at different generation ranges through evolution. 100

5.5 Average fitness throughout evolutions for $y = 15\log_2(x)$ for the games of this study 101
5.6 Average score trend and score difference throughout evolutions for $y = 15\log_2(x)$, for the games of this study. 103
5.7 Average fitness throughout evolutions for $y = 15\log_2(x)$, normal and biased loss function comparison 104
5.8 Average score trend and score difference throughout evolutions for $y = 15\log_2(x)$, different loss function comparison. 105
5.9 Average score trend and score difference throughout evolutions for left-sigmoid, for NRMSE and B-NRMSE loss calculation. . . . 106
5.10 Average fitness throughout evolutions for $y = \frac{150}{1+\exp(-\frac{x}{20}+12)}$, different loss function comparison 107
5.11 Average score trend and score difference throughout evolutions for Defender, linear function comparison. 108
5.12 Average score trend and score difference throughout evolutions for Waves, B-NRMSE, logarithm and exponential function comparison 110
5.13 Average score trend and score difference throughout evolutions for Waves, normal NRMSE, shifted sigmoid function comparison 111
5.14 Average score trend and score difference throughout evolutions for Waves, B-NRMSE, 2-part linear piecewise function comparison 112
5.15 Average score trend in validations for $y = 2^{\frac{x}{70}}$, different best parameter selection comparison 114
5.16 Average score difference in validations for $y = 2^{\frac{x}{70}}$, different best parameter selection comparison 115
5.17 Average score trend in validations for $y = x$, different best parameter selection comparison 116
5.18 Average score difference trend in validations for $y = x$, different best parameter selection comparison 117

5.19 Average score trend and score difference during validation for $y = x$ best UCB, on the game of this study 118
5.20 Average score trend and score difference during validation for normal NRMSE right-sigmoid best UCB, on the game of this study 119
5.21 Average score trend and score difference during validation for Seaquest left-sigmoid best UCB, different loss calculation comparison 120
5.22 Average score trend and score difference during validation for Seaquest with, $y = 15\log_2(x)$ best UCB, different loss calculation comparison 121
5.23 Average score trend and score difference during validation for normal NRMSE Defender, best UCB, linear function comparison 123
5.24 Average score trend and score difference during validations for normal NRMSE Defender, logarithm and exponential function comparison 124
5.25 Average score trend and score difference during validation for normal NRMSE Defender best UCB, shifted sigmoid function comparison 125
5.26 Average score trend and score difference during validations for normal NRMSE Defender, 2-part linear piecewise function comparison 126

List of Tables

4.1 *Space Battle Evolved* evolvable parameters, description, their value ranges and step. . . . 69
4.2 An example of the score in T_1, $T2$ and T_3 70
4.3 Seaquest's parameter set search space 74
4.4 Waves' parameter set search space 75
4.5 Defender's parameter set search space 78

5.1 Optimized parameters of game instances with the highest or lowest average fitness, designed by three algorithms. 93
5.2 Experiment parameters 97

Dedicated to . . .

My dear little brother O.H.

For I could not be with you on that day. . .

Because I had to work on this important mission from there. . .

And once I am able to come back. . .
It is too late. . . . . .

Chapter 1

Introduction

Automatic Game Design [1] is a subfield of Game Artificial Intelligence that aims to apply AI techniques to assist in game design tasks. The simplest games should consist of at least game rules, which indicate the procedures/constraints that players should follow or the conditions they should satisfy to win. More complicate games may also have other components such as game maps/levels, and/or certain in-game parameters. Designing a game includes deciding the look-and-feel and functionalities of these components. In many scenarios, these tasks require a number of playtests to value and determine the most suitable composition. Manually playing games repeatedly for this purpose is usually time consuming and inefficient for a few reasons. The first one is that human testers tend to be inconsistent and unable capture many possible playing behaviours. Another reason is that most of the gameplays with human-playable speed are significantly slower than the fastest processable speed. This is adequate to point out the needs of a tool that could repeatedly play the games automatically for all settings and evaluate each with the same criteria human testers do. Attempts to develop such systems have been proposed using Evolutionary Algorithms [2] [3] [4] mainly because of their generality and suitability in optimization problems.

Automatic Game Design research in the early days mainly focused on the proof-of-concept [1][5], which means experimenting the possibility of developing such systems, while more recent works targeted specific-domain tasks.

These are either auto-generating the whole game, for instance the Ludi system [2] and ANGELINA [6], or only a certain component such as map/level [3][4][7][8]. Another game design related task that would be benefited from the similar auto-evolution approach is parameter tuning, which has not gained as much popularity as map or rule generation. An initial work was proposed by Isaksen et. al. [9] for Flappy Bird[1]. This work shows that evolutionary approaches are applicable with this task, and also underlines the importance of the task in game development process.

Recently, General Video Game AI competition (GVGAI) [10] has just introduced a new *Game Design* feature. GVGAI is a video game AI competition that, in contrast to most of other competitions, encourages the development of a single AI game player controller algorithm that is general enough to play any video games instead of just one or a few. Currently there are more than 100 games in the framework, both single-player and 2-player. The game design feature of GVGAI competition does not focus on game playing algorithm development, but on providing a 'general' parameterization algorithm instead.

This dissertation summarizes the results of two research experiments done in video game auto-parameterization using an evolutionary algorithm called *N-Tuple Bandit Evolutionary algorithm (NTBEA)* [11]. NTBEA is an evolutionary algorithm decided specifically to deal with noisy environment (e.g. video games). Instead of evaluating the fitness of each individual by its crisp values (in each position), NTBEA stores the statistical information of the evaluated fitness value of such individual. These information are being stored using N-Tuple structure, therefore they are real-time accessible. Given these characteristics, NTBEA is robust to noise and fast in individual optimizing. The task we have chosen is to evolve games that provide specific player experience, using in-game score as the measurement. The first task that we have done with a variant of *Space Battle* game is to develop games that can distinguish players based on how good

[1] http://flappybird.io/

they were playing, which we called *skill depth*. We divided the player skill depth into three levels: *skilful, intermediate and amateur*. Three *General Video Game Playing (GVGP)* controllers were selected to represent human players of each skill depth. Each parameter set fitness function was determined by playing these controllers against each other in such setting, then the final score was used in fitness calculation. Although we use score to measure player enjoyment, any other metrics apart could be employed as well with our proposed evolutionary scheme.

For the second experiment, we selected three GVGAI games, and focused on another aspect of player experience which is the game's score trends. 9 functions were selected as target score trends so that the score progression achieved by an agent that plays this game approximates the curve. Since player experience is difficult to measure due to the uncertainty of player behaviour, we decided to use 2 GVGAI agents that employ score-based heuristic functions in action selection. Therefore we, to some extent, could assume that these AI players would always look to score when the opportunities present, unless such scoreable option leads to to worse foreseeable outcomes such as losing the game.

The objective of the research experiments presented in this dissertation is to investigate the possibility of applying NTBEA to parameterize game parameters using General Video Game Playing controllers as human-player substitutions. A few hypotheses were formed as the basis of the experiment setup:

- NTBEA is more robust to noise in noisy environments than standard hill-climbing evolutionary algorithms.
- NTBEA can be applied to tune game parameters to provide specific pre-defined player score trend, for any players playing the game.
- General Video Game Playing controllers can be used as substitutions for human players in automatic game parameterization.

- The games evolved by NTBEA satisfy human preferences more compared to both RMHC-based algorithms.
- NTBEA can evolve game parameters to fit the same target functions for any games.
- Biasing the loss calculation helps NTBEA in our optimization task

These hypotheses were validated in the experiments presented in Chapter 5. The knowledge gain from this research can be beneficial for numbers of fields, such as Noisy Optimization, and Automatic Game Design which can also benefit game industry as a basis for a tool for auto-tuning game parameters.

The contents of both experiments presented in this dissertation have been re-assembled into two separate academic papers, one of which was accepted in a conference and published, while another is submitted and currently under review. The contents and contributions of the authors of these papers can be summarized as follows:

- **K. Kunanusont, R. D. Gaina, J. Liu, D. Pérez-Liébana and S. M. Lucas, "The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement", in *IEEE Proceedings of the Congress on Evolutionary Computation (CEC), 2017***

  Contents: Introduction of N-Tuple Bandit EA (NTBEA) usage in game research; Space Battle Evolved game; Applying NTBEA to Space Battle Evolved to evolve game parameters that can best distinguish players with different skill-depth

  Contributions: My contributions are game implementations, experiments and writings. The second and third authors helped with writings and experiments, also the second author provided the implementation of one algorithm used and assisted in game implementation. The fifth author implemented NTBEA and wrote its technical section. All authors helped in discussions and proof-readings.

- **K. Kunanusont, S. M. Lucas and D. Pérez-Liébana, "Modeling Player Experience with the N-Tuple Bandit Evolutionary Algorithm", submitted to** ***the 2018 Artificial Intelligence and Interactive Digital Entertainment (AIIDE) Conference***

  Contents: Applying NTBEA to parameterize three GVGAI games so that the players that play in such evolved games retrieve the score in the same trend as the specified functions.

  Contributions: Game space setting, fitness function, experiments and writings. The second author provided NTBEA implementation. The third author helped with game selection, discussions and writings.

This document consists of further 6 Chapters. Related published literatures are reviewed in Chapter 2. This is followed by essential background knowledge, summarized in Chapter 3. Next, Chapter 4 gives detailed descriptions of the approach proposing. Then, the experiment setup, procedures, and results are discussed in Chapter 5. After that, summary and findings of the research, along with its conclusion are stated in Chapter 6. Finally, the last Chapter describes possible future works of this project.

Chapter 2

Literature Review

In this chapter, previously proposed related works are described. This is to give an overview to the readers about the works that have been done prior to our research. Each of these works falls into either *Automatic Game Design*, *General Video Game AI* or *Evolutionary Algorithm* categories.

2.1 Automatic Game Design

Automatic Game Design refers to systems that are capable of generating game components automatically and efficiently from a given design constraint. Research done in this field aims to close the gap between crude machine-like design and human design which in general is more sensible and playable. An alternative objective is to explore some human-overlooked designs that may be interesting. Early works (before 2010) were done mainly for exploring the possibility of automatic game design, while more recent works focus on developing algorithms and applying them to certain games. These automatic systems can be categorized into three broad areas based on the components/tasks they were developed to fulfil, either generating the whole game, generating game maps/levels and game parameter tuning. Both the automatic game design early works and recent related works are reviewed next.

2.1.1 Early Attempts: Proof of Concept

Nelson and Mateas [5] tried to define automatic game design as a problem-solving task by declaring 4 design factors needed to create a game. This includes *game mechanics*, *game representation*, *game thematic content* and *control mapping*. Game mechanics refer to how a game changes its state when it is being played, while representation means how those states are being presented to the players (visual/audio format). Thematic contents is the domain/world that the game takes place, and control mapping is how a specific physical action of a player is being mapped to affect the game. To clarify these factors with an example, a well-known Pac-man game mechanics contain, for instance, the states that the avatar collects a power pill, which would then effect the enemy ghosts to transform. The game is represented in 2 dimensional grids with the avatar is a yellow circle, and the enemies are in comical ghost-shaped. The game takes place in a big narrow corridor maze full of collectable pills and a player (for a computer keyboard version) uses arrow keys to control the avatar directions. Nelson and Mateas developed a method to auto generate a game by designing all of these components using a set of common sense composition rules acquired from *WordNet* [12] and *ConceptNet* [13]. The generated game was in a similar style of a Nintendo *WarioWare* game. Later, as they had realized that an ability to easily add and remove some mechanics from the game is necessary for automatic modification of a generated game, they proposed another design architecture based on event calculus [14] and reckoned that this new ability profited the automatic game design as it could generate playable games by modifying some current components that were blindly generated earlier.

In the same year, Togelius and Schmidhuber [1] proposed their work in using evolutionary approach to evolve a 15×15 2-dimensional grid game rules from scratch. In their game space, each grid cell can contain either empty space, a wall, or a circular-shaped object that could be in either red, green or blue color. Their proposed game generator (evolver) had to evolve a set of game rules that

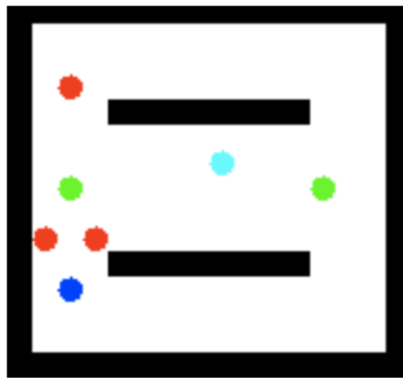


FIGURE 2.1: A game evolved by [1], taken from page 4 of the publication

define the game representation, i.e. how many objects, where are they located and which colors are they, which effects will be triggered after each pair of objects collide. After a game rule set has been generated, it would be evaluated using two random-action controllers. A game was judged as too easy if both controllers performed well. Although most of the game found in this game space were unplayable, they were some playable games generated by the generator. An example of their evolved game can be seen in Figure 2.1, which is taken from page 4 of [1]. As claimed by the authors, this work was the first attempt to do single-player game rule evolution, also the first attempt to apply evolutionary algorithm into non-board game design, inspiring later works in this field including ours.

Research works after this focused on developing and improving auto generation tools for specific domains. Some of the proposed literatures for full game generation are reviewed next.

2.1.2 Auto-Generate Full Game

In 2010, there was an introduction of *Ludi* [2] developed by Cameron Browne and Frederic Maire. Ludi was the first, and still the only one, combinatorial automatic game generator framework with a commercially published game (Yavalath[1]).

[1] http://www.cameronius.com/games/yavalath/

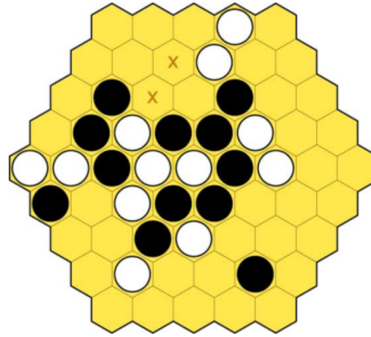


FIGURE 2.2: Yavalath puzzle, taken from page 12 of [2]

Figure 2.2 shows Yavalath game state, taken from page 12 of [2] As defined in their paper, combinatorial games are two-player turn-based deterministic games with no hidden information and the outcome states are finite. Ludi generates games by combining a set of *ludemes*, which is a piece of information for a game component, together. They integrated an evolutionary algorithm into the system to generate game ludemes population and offspring, but keep all survived offspring instead of keeping only a few. Four criteria were applied to judge the generated game quality: depth, clarity, drama and decisiveness, as defined by Thompson [15]. Ludi is the first widely accepted successful application of using evolutionary algorithms in full game design, although the combinatorial game space is small comparing to video games, this framework is still a breakthrough of automatic game generation research field. The game Yavalath has an interesting winning condition that is quite beyond human common sense (create a four-in-a-row without three-in-a-row). This points out that a computer program can find a rule set that humans are interested in but could not think of.

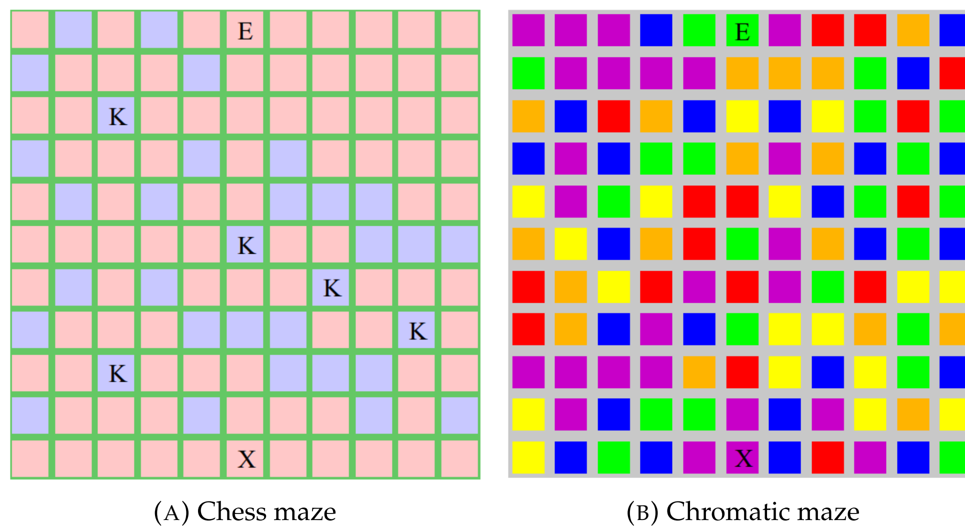
A year later, Michael Cook and Simon Colton proposed an automatic arcade game generator named ANGELINA [6]. Using similar game representation with the work done by Togelius and Schmidhuber in 2008 [1], Cook and Colton implemented generators for three game components: rule sets, character layouts and maps. Evolutionary approach was applied to evolve these components separately, while also sending information of the fittest individuals into the central

point to share with others. Fitness value of a generated map is calculated from *fragmentation* - number of isolated tiles - and *domination* - number of dominated tiles. A tile is isolated if it is disconnected from the walls. Two tiles are being dominated by a third tile if for all paths that connected between the first two tiles included this third tile. For each ruleset (individual), some unnecessary rules would be filtered out first, along with those rules which lead to an unplayable game. Then a bot player with a specific assigned behaviour would play in the game with such ruleset. The fitness value would be then determined based on the bot player score. Game layout fitness value was calculated by how sparse the game objects were placed, how many of them, and also whether the layout is consistency with the fittest map at that time. ANGELINA was the first entire game generator using *multi-faceted evolutionary algorithm*.

As can be seen in ANGELINA system, generating a full game requires different generators for each component. There were research works that were proposed for automatic generator for a specific game component, which is also known as procedural content generation. The most popular content to generate is maps or levels. Some of the proposed works for map/level generation are reviewed next.

2.1.3 Auto-Generating Maps or Levels for Games

In 2010, Daniel Ashlock proposed a wide range puzzle generators using an evolutionary algorithm [3]. Two types of puzzles were tested, including *chess maze* and *chromatic puzzle*. Chess maze is a maze-based puzzle that the player movement is restricted to be the same as his/her assigned chess piece, and the goal is to traverse through the maze to the exit using only such movement. Chromatic puzzle is, again, a maze-based puzzle that the player could only move to the adjacent tile with the same color of his/her current tile, or the color that is adjacent to his/her current tile color in the color wheel. Figure 2.3 shows a chess maze (2.3a) and a chromatic maze (2.3b) generated using Ashlock's approach,



(A) Chess maze (B) Chromatic maze

FIGURE 2.3: Example of mazes generated by [3] taken from page 8 of the publication

taken from Figure 7 (page 8) of his paper. Ashlock's aim then was to explore the possibility of employing an evolutionary algorithm to evolve such puzzle mazes with pre-defined difficulty levels, using dynamic programming to search for the minimum number of moves needed. This move number would be then used to determine the fitness value of the maze map. He reckoned that these puzzles were not intend to be a stand alone game but rather be a mini-game in other bigger games, hence this work is considered as a game content level generation.

Sorenson and Pasquier [4] developed a level creator that was applicable with Super-Mario Bros. and Legend of Zelda. They raised a point that using bottom-up approach to generate game levels (rule-based) restricting the level generators to obey the constraints, hence occasionally causing most time spent in the level construction rather than creation. Based on this argument, they proposed a top-down approach that only specify how the 'preferred' levels should be, instead of how it should be assembled. Using that with a genetic algorithm named *Feasible-Infeasible Two-Population (FI-2Pop)* as a level evolution method, they were able to produce challenging and playable levels for both games. The fitness function they employed was carefully designed to represent enjoyable rate of a level, with an assumption that a player enjoys a game that is neither too easy nor too

difficult, therefore the final equation was to measure how dynamic a level was because the dynamic rate of a map is directly related to how difficult it is.

Another work done for an infinite version of Super Mario Bros. level generation was proposed by Shaker et. al. [7]. They applied *Grammatical Evolution (GE)* to evolve levels that offer the best experience for players with different emotional states from the pre-designed grammar that represents level structures. A player experience model that they used as the input to calculate fitness values was obtained from their previous study on modelling the relationship between player preferences and game content characteristics [16]. The levels were generated aiming to lead human players into feeling either engagement, frustration and challenge. Each of these levels were played by two selected AI controllers with different general behaviours, with the first one (A*) often performing jump action while the second one (simple heuristic-based) only jumps when needed. Their gameplays were recorded and analyzed with the player experience model to see if the levels satisfy their emotional state optimization objectives. They concluded that it is easier to develop levels that make skilful players feel engaged or challenged. This work shares a similar aspect to our work in term of optimizing a pre-defined player experience, with an obvious difference that our work employed parameterization instead of level generation. A less obvious dissimilarity is that we evaluated player experience by pure score or score trend, while this work did with their own extracted features. Furthermore, our objective does not involve measuring a subjective and stochastic concept like emotional states, but focused mainly on a more straightforward numerically measurable statistical data.

A work that was also related to player preferences was done in 2013 by Liapis et. al. [17]. They proposed a novel evolutionary algorithm called *Rank-based Interactive Evolution (RIE)* that was adaptable with player preferences during the gameplays. Their testbed was a 2-dimentional grid map with size 64×64 that consists of either an empty tile, a blocked tile, a resource or a player's base. They

claimed that this was a simple representation for multi-player strategy games. Employing knowledge about aesthetic features from some citations, 10 fitness functions were designed. This ranged from optimizing the map navigational rate, resource locations to visual representation. A computer program were implemented as a human user simulator in preferred map selection, which the RIE would evolve new maps with the same fitness trends in each aspect. They found that the proposed RIE were able to rapidly adjust map generation to satisfy user preferences. Our work also employed computer program agents in place of human testers in fitness evaluation, but the agents we were using were mostly general agents, not domain-specific ones like in this work. Therefore there are no need in re-implementing agents for each new game testing.

Satisfying the players was one of the main objectives in developing video games. Lara-Cabrela et. al. [18] proposed a method to evolve maps for *Planet Wars*[2] that are 'balance' and 'dynamic', as the authors claimed that those were the factors for this game to be considered as 'fun'. They proposed a set of fitness functions to guide the evolutionary algorithm to each, and both objectives in different runs. These fitness values were then combined with fuzzy-logic rules to determine the level of balance and dynamic for a map. Planet Wars game shares some similarity with the *Space Battle* game that we employed in our first experiment as they both are two-player competitive games that take place in the space, though the details and rules are different. The game space we defined also has some parameters that defines the map look, and we have our definition of 'fun' game as a game that better players can score more, which may related to this work as usually better players react better to more dynamic maps.

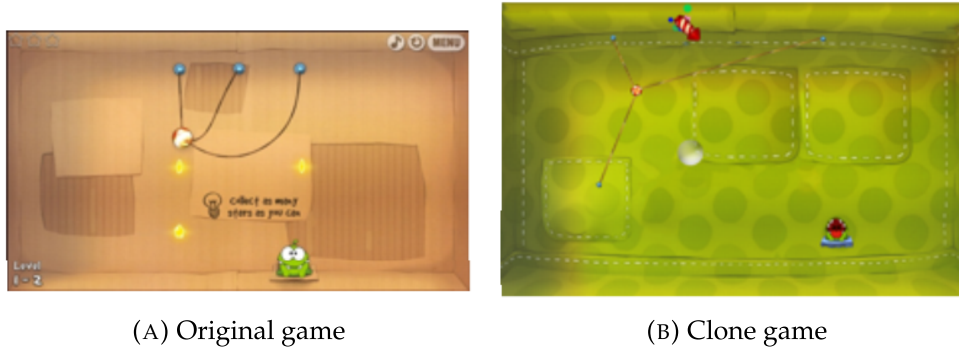
Not only the game player preferences were taken into account in automatic game level generation, but also the game level designers'. Liapis et. al. [19] proposed a *Map Sketch* tool that was adaptable with the designer preferences. After studying various strategy game map design patterns, they introduced six

[2] http://planetwars.aichallenge.org/

quality measurements and defined a fitness evaluator based on those. This fitness function was then applied to evaluate the fitness of the feasible group in the FI-2Pop (as in [4]). They started the experiment by optimizing each aspect of the six, and attempted to combine all aspects together later on. Unsurprisingly, they found that optimizing multiple objectives were much more challenging that single aspect. Nevertheless, they reckoned that the tool would be beneficial to the level designer as it could adapt to their choice of design and assist them by auto-generating the similar patterns in higher resolution.

Mobile games have been as well gained interests in automatic level generation. An example is a commercially well-known *Cut the Rope*[3] developed by Zeptolab. This game is a physics-based puzzle game that the player has to perform a correct sequence of actions (e.g. cut the ropes in a specific order) to win a level. The goal is to deliver the round-shaped candy item to the frog-liked creature avatar. Shaker et. al. [20] used Grammatical Evolution to evolve a level description based on a designed context free grammar. Each level grammar defines positions of the candy, the avatar, and other game components such as ropes or bubbles, along with their specific information (e.g. length for ropes). They created a clone version of the game specifically for the experiment. The screenshots of the original game and their clone version are given in Figure 2.4 As their aim then was to generate at least playable levels, they designed a fitness function that was a combination of various constraints, some of which were the frog avatar placement, the candy placement and other component placements and orientations. Using this scheme, they were able to generate 100 playable variants of levels with 20 generations of individuals and population size $\bar{1}$00, all of which were validated by automatic playtests using a simple random action AI controller. However, this validation step consumed most of the experiment time, hence a follow-up work [21] that aimed to improve the AI controller was proposed. A prolog-based agent that employed a strict rule-based approach

[3] https://www.zeptolab.com/games/cut_the_rope



(A) Original game (B) Clone game

FIGURE 2.4: Comparison between the original and clone version of Cut the Rope game used in [20], taken from page 2 of the publication

to search for a 'sensible' action was developed and tested with the framework. With this agent controller, they were able to detect playable levels much faster as the agent was more efficient in finding winning action sequences.

Another work on map generation for a physics-based environment problem was explored by Pérez et. al. [8]. They considered a variant of a well-known NP-Hard Travelling Salesman Problem [22] that takes physics components (speed and orientation) into account, called *Physical Travelling Salesman Problem (PTSP)*, and implemented a game-like framework to represent this problem. In their paper, they explored on using three different AI bots to assist in fitness calculation of map evolution using an evolutionary algorithm. Three AI controllers have their own route planners embedded, the simplest one was *Nearest-First* planner that selects the closest visiting point to go first, the second planner *Distance* plans the waypoint order by using branch-and-bound with A* search and selects the lowest cost plan, lastly the *Physics* planner that employs the same scheme with Distance planner, but also measuring the cost from physical factors such as the speed and orientation of the vehicles, and the direction it is travelling. Their aim was to evolve maps that Physics planner would achieve better performance than Distance, and better than Nearest-First respectively. The objective of this work was similar to the first experiment done in this dissertation, and the fitness functions employed were similar. Although since *Space Battle* is a two-player game,

we used the score differences instead of pure score as in this work.

Level generation has dominated interests in automatic game design so far, comparing to other aspects such as game parameter tuning. This is also important as manually tuning game parameters could be time consumption, and as can be seen from Yavalath, humans might miss out some interesting settings. I review next the literatures of game parameterization.

2.1.4 Auto-tuning Game Parameters

To the best of my knowledge, the first work that tackled this problem was done by Isaksen et. al. [9]. They defined a parameter space of a well-known game *Flappy Bird*[4] and tweaked those parameters with evolutionary algorithms to see a different variant of games while keeping the rule fixed. Some of these parameters included the pipe width, the gap between pipes, the size of the player avatar and the gravity force. Every unique game variants were played repeatedly 1000 times by a simple AI controller and the time spent were averaged to estimate difficulty levels of each variant. Then they clustered generated games using game parameters as features to select a representative game from each cluster. Four unique settings were discovered, all are different in pipe lengths, gaps, player size and gravity, which led to a unique gameplay experience for each. This shows that changing parameters alone can lead to probably new game sets, even with the rules are exactly the same. A follow-up work was done then for generating game variants with different difficulty levels [23]. They started by implementing a perfect player model that would never lost a playable games, then introducing some noise into it to create various controllers that representing human players with different skill depths. Using playtest from these AI controllers, they categorized the game variants into three disjoint groups: *Impossible games* in which all players lost, *Playable games* that some players were able to survive for a time and *Trivial games* that all players survived for long

[4] http://flappybird.io/

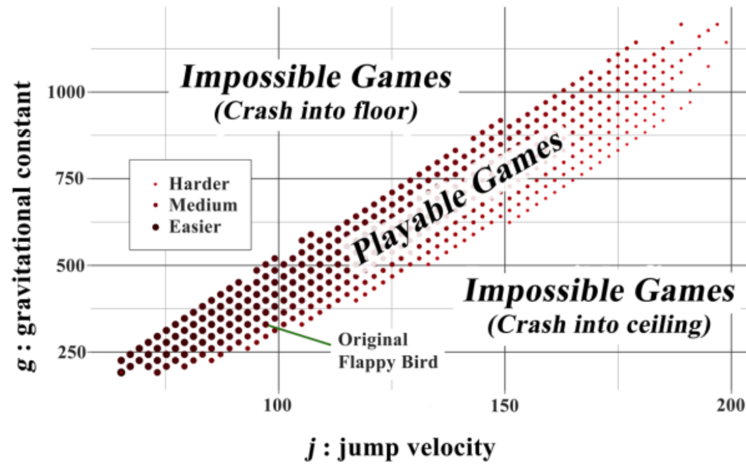


FIGURE 2.5: Flappy bird parameter space, taken from page 6 of [23]

time. They found that all playable games are located together connectedly in the game parameter space, as given in Figure 2.5 which is taken from page 6 of [23]. These two works inspired the importance of auto-parameterize in video game development as adjusting the parameters alone can result in completely different games.

Another automatic game parameterization was proposed by Liu et. al. [24] for a two-player game *Space Battle*. The authors evolve game parameters using *Random Mutation Hill Climber (RMHC)* evolutionary algorithm and its improved version called *Multi-Armed Bandit Mutation RMHC*, which introduced UCB equation in selecting a next mutated parameter and value. A *General Video Game AI* [25] controller named *Open-loop Monte Carlo Tree Search (OLMCTS)* was used as an AI controller to compete with another strategy-based *Rotate-and-shoot (RAS)* controller. Their objective was to evolve the game parameter sets that favor OLMCTS over RAS, hence giving more fitness value if the game was won by OLMCTS. Multi-Armed Bandit Mutation RMHC performed better as it explored more and converged faster. They put some of high fitness parameter set on visual gameplay and found that most of the parameter sets penalized RAS with high cost missiles. This means that RAS, which is just rotate around and

keep shooting, would constantly lose points. Nevertheless, they found an unexpected high fitness pattern with slow missiles and fast movements, which does not benefit RAS but does help OLMCTS. Space Battle is first introduced in this work, along with the idea of using GVGAI agents in playtest.

In this section, I have reviewed in details the related literatures in automatic game design. The same is done for General Video Game Artificial Intelligence (GVGAI) next.

2.2 General Video Game Artificial Intelligence (GVGAI)

2.2.1 Motivation and Origin

General Video Game Artificial Intelligence, or *GVGAI* [25], is a framework for *General Video Game Playing (GVGP)* [26], that aims to fulfil a part of the goal of a popular research field in AI called *Artificial General Intelligence (AGI)* [27] but focuses only on video game domain. As the name suggested, research in AGI involves developing AI that is general, which means capable of solving problems with different levels of difficulties and characteristics.

The concept of GVGP is extended from *General Game Playing (GGP)* [28], which was the first attempt to do AGI in game research. GGP was introduced by a group of researchers at the Stanford University in 2005 as a competition for computer game playing agent. In contrast to other such competitions, games and rules were unknown to both the agents and their developers prior to agent submission. These rules would instead be given to the agents right before they start playing. This, as claimed by Genesereth et al. [29], is to enforce the computer players to employ their own intelligence to make use of the given rules the most, and not rely on their developers. Therefore this should encourage the researchers to develop algorithms for adapting into given rules rather than analyzing the rules, inventing solutions and injecting to the agents (as usually done in all domain-specific AIs). This adaptation ability is considered in AGI field

as more 'intelligent' than narrow artificial intelligence since it is more similar to human intelligence, which is adaptable with various problems.

Although the word 'game' could cover wide variety of activities, games in GGP competition are all turn-taking board games. This means the framework itself does not cover (and from the game definition language they are using, could not cover) video games. GVGP, proposed by Levine et. al. [26], concept was introduce as a complement of this by focusing on developing AGI for playing video games. In contrast to board games, video games require real-time interactions with the game environments, and most of the (human) players do not read the rules before playing but instead observing the games and reacting. Therefore, in GVGP the agents would not be given the rules at the beginning of the game, but would receive the environment information at that particular time, and should apply this information to provide an action to the game within a small amount of time.

Apart from GVGAI, another well-known framework for GVGP is *Arcade Learning Environment (ALE)* [30]. ALE assembles all Atari 2600 arcade games together as testbeds for evaluating AGI algorithms, providing screen-capture tool as a choice of input. The framework has been widely applied in many works, with the most popular one (at the time this dissertation is written) be *Deep Q-learning (DQN)* by Mnih et. al. [31] from Google DeepMind[5]. They proposed a learning agent that receives only screen information and output actions to the game. Using deep *convolution neural network* [32] in combination with *Q-learning* [33] and other neural network-related techniques, their agent was capable of learning to play all games in ALE at human level.

Although ALE has a huge amount (2600) of games in the framework, it is a finite set. It is arguable that an agent that has conquered all games is general enough to solve other video games outside this domain. That is, the framework lacks the ability to constantly update its testing task, which is one important

[5] https://deepmind.com/

nature of a good AGI evaluator framework. Moreover, all of the input screens for ALE are of the same size, which seems to be a hole for many algorithms to take advantage of. Ideally, a framework for GVGP (AGI in video games) should have, or at least potentially have, infinite number of game supplies, and not restricted to a fixed-size screen. General Video Game Artificial Intelligence (GVGAI) has these characteristics.

2.2.2 Competition Tracks

General Video Game Artificial Intelligence or GVGAI is a GVGP framework and competition, first introduced in 2014 [10] by a group of researchers at University of Essex. It has gained interested since then with many research works have been done using its games. In 2017, GVGAI framework featured 5 competition tracks, which are *single-player planning track*, *2-player planning track* [34], *single-player learning track* [35], *procedural generation track* [36] and *rule generation track* [37]. To participate, a competitor has to first register on the website[6] and obtain the framework source code. Then he or she should develop a controller (a generator for generation tracks) and submit to the website as suggested by the instructions given for each track.

Although winning all games with one 'general' algorithm is the main objective here, it still remains unsolvable. The best controller in the 2017 single-player planning competition (YOLOBOT) has managed to win only 106/250 (42.4 %) gameplays[7] (playing 5 levels of 10 games 5 times each). This indicates that there are still improvements to work on in developing general agents, also pointing out that this framework is reflecting how challenging the GVGP (and also the AGI) problem really is. GVGAI games are of different types (although all are represented in 2-dimensional rectangular map with up to 5 non-nil in-game available actions) which require different strategies to solve. This may raise a

[6] gvgai.net
[7] http://gvgai.net/gvg_rankings.php?rg=12

question of whether such general algorithm exists, as each of them would perform well in some problems and bad in others. Fortunately, this concern has been analyzed by Ashlock et. al. [38] with a conclusion that GVGAI would not suffer from the *No Free Lunch theorem* [39]. They claimed that, since GVGAI game corpus consists of only the games that humans are interested in, which is a small subset of the whole game universe, the framework itself is under the line of the no free lunch theorem by a well margin. This also means, in theory, there should exist that general algorithm to solve all games (in which we all know there is, as humans are capable of winning all games in this framework easily. This means the algorithm must be composed of something less than or equally complex with human intelligence).

2.2.3 GVGAI for Game Design

Although Procedural Content Generation (PCG) is one of the most famous game-related research topic, general generator is a new and much more challenging problem. To the best of my knowledge, there was only one publication directly related to GVGAI game design aspect, specifically level generation track. This was done by Neufeld et. al. [40]. They first translated *Video Game Description Language (VGDL)* (the game description used in GVGAI framework) into *Answer Set Programming (ASP)* by analyzing some key constraints of the game that can be referred into ASP rules. Then, using such ASP rules, they applied EA-like approach to generate levels by maximizing the score difference between the sample *Monte Carlo Tree Search (MCTS)* [41] and random controllers. This was based on the assumption that a good level should facilitate good players more than poor players, which is the same assumption we made in our first experiment.

In a tutorial session for the 2017 IEEE Conference on Computational Intelligence in Games (CIG 2017)[8], there was a mention about an ongoing work in

[8] https://www.youtube.com/playlist?list=PLsz4FpDoCTzyRJqDV7VpT2hOO2gHOZwA5

applying GVGAI in game design. The authors had modified GVGAI framework to support game parameterization, which is to consider game parameters as a (huge) search space and try to find a set of parameters that satisfy demands. Initial work was done with the game Aliens. The second experiment done in this dissertation is the continuation of such initial work, by extending it into other games, and to a different task.

2.3 Evolutionary Algorithms

Evolutionary algorithms were originated since 1950s by the idea of modelling the natural selection process. There were three proposed mainstreams in early days that share some common concepts, yet differ in details. These three include *Evolution Strategies (ESs)*, *Evolutionary Programming (EPs)* and *Genetic Programming (GPs)*. Bäck and Schwefel [42] attempted to distinguish them using mathematical notations based on each of the concept, while Melanie Mitchell summarized the brief history, along with their similarities and differences, of all variants in the first chapter of her book written in 1998 [43]. Both of these literatures present the same fact about the originality of each EAs, which are summarized in the next few paragraphs.

The *Evolutionary Strategies* or ESs concept was first investigated and proposed in 1965 by Ingo Rechenberg [44] who also continued his work on this and presented the clearer concept of his nature-inspired model in 1973 [45]. This method was employed then as a real value parameter optimization. He later expanded the concept from only a single next generation individual (offspring) creation per generation (called (1+1)-ES) to (μ+1)-ES that μ individuals genes were re-combined up to produce a new offspring [45]. This method was further generalized into ($\mu + \lambda$)-ES that the recombination resulted in more than one offsprings [46]. This has become the state-of-the-art ES employed in many applications later on. The ESs general steps including first randomly generating

the first generation population and evaluating each individual. Then producing λ offsprings from μ parents (re-combination) and then applies a mutation process to these new offsprings. After that, evaluating each offspring and retaining only some of the best to continue. These steps would be repeated until the termination condition is satisfied.

Evolutionary Programming or EPs was introduced in 1966 by Lawrence J. Fogel, Alvin J. Owens and Michael J. Walsh [47] as a natural-selection-based simulation using finite state machines (FSM). They evolved these FSMs by applying mutation process to the state transition tables, which would be used to predict the next alphabetical character based on the given input. The more accurate the prediction of an FSM is, the better the evaluation score (fitness), hence higher chance to survive and breed future offsprings. For each parent FSM, only one offspring is generated by mutation, and half of the best individuals of the combined population set (parents and offsprings) would be retained for the next generation. Later on, David Fogel studied this and extended the concept to support real value evolution apart from only FSM [48]. Fogel and his colleagues were aware that the practical implementation of their EPs required manually-tuning of high level parameters, and came up with a method to evolve these optimal parameter settings while the evolution is taking place. This technique is called *meta-evolutionary programming* [49]. The EPs general evolution steps are similar with ESs without any re-combination process, and always select half of the best individuals in the combined set, which is equivalent to the $(\mu + \mu)$-ES in ES concept.

Finally, *Genetic Algorithms* or GAs were proposed by John Holland [50] in 1962 as a guideline model for auto-environmental adaptation system, again inspired by the nature. Initially, unlike ESs and EPs, this idea was not designed to solve any specific tasks but rather as a suggestion of importing the evolution concept into computer field. This 'import' refers to applying three evolution

operators called mutation, crossover (similar to re-combination in ESs) and inversion. According to the timeline, this was the first attempt to do so as ESs and EPs were proposed after this GAs. Holland later discussed the possible applications of GAs in his book [51] which included some still well-known tasks such as parameter optimization (which is also the main task of my experiments). General steps of the original GAs (also known as canonical GAs) are similar to ESs, with more priority given to crossover rather than mutation.

The emerging of those three algorithms several decades ago has shaped one of nowadays the most popular research field in computer science: evolutionary algorithms. There has been enormous proposed works in related topics, either improving the methods themselves, or applying them to solve specific tasks. I first review some of the general improvements that are relevant to this thesis research work, then some of the applications in game domain are addressed next.

2.3.1 General Improvements

EAs in general are composed of three main components: initialization, evaluation and evolution. Initialization refers to the selection of the very first population at the beginning of the whole process. Evaluation means to justify the quality of each individual, usually indicating by a real number. Evolution is composed of all nature-inspired operators, such as mutation, re-combination (or crossover) or any other processes done to an individual to produce different individuals.

Initialization step was often accomplished by randomly selecting genes to every individuals in the first population, as it is the simplest, least effort, quick and usually produces at least acceptable results. However, in some domains, careless initialization lead to the local minima convergence after the evolution is finished. There has been a number of works proposed to tackle this issue.

The review done by Kazimipour et. al. in 2014 [52] categorized these improvements based on three criteria: the randomness of the algorithm, whether it is composed by more than one non-separable components, and whether it is general or domain-specific. Using their criteria, the initialization employed in our N-Tuple Bandit EA (NTBEA) is considered as stochastic, non-composition and general.

During evolution, ideal EAs should explore the space as much as possible to ensure that it does not fall into a local minima, which is also known as *premature convergence*. Therefore, maintaining diversity in each evolving population is crucial. An early attempt was proposed in 1984 by Mauldin [53]. He suggested that the premature convergence happened because the individuals are too alike during evolution. Therefore in his paper, all individuals were ensured to be different from others at least one bit (all individuals were sequences of bit strings). A new individual would be compared to all existing individuals in the current population first before it is added. If it is identical to one, a randomly selected bit in the sequence was selected and flipped. Then the checking is performed on this mutated individual again, and repeats until the individual is completely new. Years later there has been many works proposed to overcome this issue, some of these were reviewed in 2012 by Gupta and Ghafir [54]. The first method is called *Niching*, which is to allow converging to more than one solutions during evolution, invented by De Jong [55]. This could alleviate the chance of premature convergence as it allows EAs to monitor more than one local minima, increasing its chance to find the global minima of the space. Another method called *crowding*, also proposed by De Jong [55], takes place at the selection step. That is, before a new individual is about to be included into the population, by finding the most similar individual in the population first then removes it. It seems that De Jong shares the similar idea with Mauldin [53] but Mauldin chose to modify the new individual and preserve the antecedent instead. There also be other criteria to select individuals out, for example considering only offsprings

and their parents [56]. An individual may also be restricted from mating with another too similar individual [57] to preserve diversity, or being forced to share their fitness values [58] with neighbours. In some works *Multi-Objective Evolutionary Algorithms (MOEAs)* techniques were employed considering diversity maintenance as a secondary task [59].

This dissertation presents a work about game parameterization using an EA with some stochastic GVGP agents. The second part of the experiment were for GVGAI framework. Most GVGAI gameplays are non-deterministic, therefore one play-through of a game is not sufficient to justify its true fitness value correctly. This means that the environment we use is noisy. A state-of-the-art guidelines with published literatures prior to 1998 was written by Beyer [60]. In his paper, he confirmed that the presence of noise leads to uncertain location of global minima and there were three methods to reduce noise: *resampling*, *sizing the population size* and *inheritance of rescaled mutation*. With a simple mathematical reduction, he showed that resampling n times reduces the noise strength by $\sqrt{n}$, although that would be traded off with higher evaluation cost. He also presented an equation for suitable population size by reducing it from the previous literature. The last method is to perform large mutations to the parent individuals, selects the mutation that produced the best offspring and scales down the length before adding new offsprings into the population. The idea behind this is that highly mutate the individuals may result in the significant differences between the offsprings and parents, which should be higher than the noise level.

Years later, Jin and Branke did a survey on evolutionary optimization in uncertain environments in 2005 [61]. They categorized these uncertainty into four categories based on the causes: *noisy fitness function*, *post-optimizing parameter deviation*, *fitness approximation instead of evaluation* and *dynamic optimal point location*. Noisy fitness function refers to the situation that evaluating the same individual twice gives different values. This is certainly our case as both our

game framework (GVGAI) and our AI players (RHEA and MCTS) are stochastic. Therefore performing the same action sequence in the same game might give different outcomes, and we were using the AI agents that may not perform the same action in the same situation. Three types of approaches were proposed to deal with this type of noise. The first two were *resampling* and *increaing the population size*, which Jin and Branke called them *explicit averaging* and *implicit averaging* respectively. Another approach is to modify the selection step. For instance, introducing a threshold value that a new individual has to be fitter than the parent at least by this threshold [62]. Although the presence of noise seems to effect slow convergence in optimization, some works reported that a certain level of noise actually assisted EAs [63] [64].

The second cause is that some parameters of the environment might change after the optimization is finished, causing the solution to be non-optimal. Jin and Branke stated that the EAs should be able to find a more robust solution that tolerate this changes, similarly with the over-fitted issue in Machine Learning that a good learning model should overcome it. They reckoned that a robust solution should not be affected by a small alteration in either design variables or environment variables. To accomplish this, one may attempt to define an *expected fitness function* that evaluates the fitness in probability format instead of crisp values. Additionally, one may try calculating both expected fitness value and normal fitness value, and then employing MOEAs techniques to optimize both [65].

In some domains, evaluating the actual fitness might be impractical or unaffordable. Some obvious examples related to games are the measurement of players' fun and enjoyable level. Most of the time the fitness function of these were manually defined by the researchers. Jin and Branke [61] provided guidelines on how to properly approximate the values in noisy environments. The last source of noise is caused by *dynamic environment* that the optimal point may change with time. To tackle this, maintaining diversity during the evolution [66],

keeping track of some previous population [67], or multi-population evolution are some possible approaches [68].

For more recent works in handling uncertainty in EAs, Hansen et. al. [69] proposed an uncertainty measurement that was based on the changing in ranking of the population during the evolution. They integrated this into a combustion control system with an uncertainty treatment module that aimed to prevent system failure. *Covariance Metric Adaptation Evolutionary Strategy (CMA-ES)* was applied as an EA to do parameter optimization. They claimed that the effect of uncertainty in ranking-based EAs would affect the algorithm performance only if the individual ranking was changed after re-evaluation. After the uncertainty level was measured, the treatment step would take place if the measured value is above the threshold. Treatment methods can be either increasing the evaluation time or the population variance. Another work done later by Syberfeldt et. al. [70] employed iterative resampling to prevent bad solution to survive and breed due to noise. They tested that technique with two real world multi-objective optimization problems. Their overall procedures were of 5 steps. First, a pair of individuals to be compared are sampled (evaluation) n times to estimate noise level. Next, each of to-be-optimized objective mean and standard deviation values are then computed. Then the confidence level of the individual relation (how probable one has better, worse, or equally in quality compared with another) is calculated. After that, they used *Welsh confidence interval (WCI)* to indicate whether the differences between two individuals are significant from the confidence value. Finally, another solution were resampled and all steps are repeated. Last year (2017) Friedrich et. al. [71] showed that a type of GA called *Compact Genetic Algorithm (cGA)* was robust to noise without the need of any additional handling method. In contrast to normal GA, cGA does not focus on mutating the population to find better individuals, but instead trying to 'picture' each gene frequencies that the ideal population would have. This can be done by

starting with random frequency values, picking two individuals from this population distribution, comparing them and updating the value accordingly using tournament selection. In their paper, Friedrich et. al. presented a mathematical reduction to prove that the simple hill climber is susceptible to noise even with low amount. Then they further proved that population-based EA with mutation could not rely on increasing population size to alleviate noise level when the intensities is high. In contrast to both, they showed that cGA was capable of converging into optimal even with the presence of noise. The practical experiment the did in comparing cGA with hill-climbing also verified that. Lucas et. al. [72] proposed two other versions of cGA and observed their performance in noisy tasks. The first improvement is to sample more than two individuals per iteration. That is selecting n individuals and compare them pair by pair, which would be $\frac{n(n-1)}{2}$ comparisons in total. The second variant is called *sliding window* cGA, which aims to update the gene probability vector more frequent. That is, once an individual is sampled, it would be evaluated and compared with other previously drawn individuals. The probability vector is updated after each comparison. The authors of this work stated that the main objective of their work is to find an EA that is robust to noise for applications in Game AI. Therefore applying their cGA variants into our parameter tuning might be a possible future work.

I have briefly reviewed some works for general improvements of EAs in different aspects in this subsection. Next, some proposed literatures of applying EAs in game domain would be reviewed.

2.3.2 Applications in the Games Domain

Games have been widely accepted as efficient real-world problem testbeds for long time, especially for newly developed Artificial Intelligence algorithm testing. This is mainly because a game usually is a combination of tasks or problems for the players, and solving them often requires intelligent-driven procedures. In early days, EAs were applied to develop board game players in allied with neural networks, with the idea of using them to evolve a suitable network topology and structure. This is called *Neuroevolution*, which its applications in games and advantages were reviewed by Risi and Togelius [73]. David B. Fogel had been working on applying this to develop variety of board game computer players such as Tic-Tac-Toe [74], Checkers [75] and Chess [76]. The network for each game was designed to receive board representation as input and evaluate all position values for output. Therefore in Tic-Tac-Toe a network received 9 inputs and produced 9 outputs. Then the next move was decided by picking the position with highest value. All neural networks in his Tic-Tac-Toe paper were *Multilayer Perceptron (MLP)* with one hidden layer. The number of neurons in the hidden layers and connection weights between layers were evolved using an EA. He started with 50 networks, playing them against a slightly noisy close-to-optimal rule-based player and recorded the results as fitness values. Then each network fitness were compared with other 10 randomly selected networks in the population, and only some of the best survived for the next iteration. This approach was extended to create a checkers computer player *Blondie24*. Obviously the complexity of checkers is more than Tic-Tac-Toe with 8 by 8 board size, though only half were actually in gameplay. E. Harley stated in a review of Fogel's book [75] that Fogel's objective was to develop a player that could just learn to play without being told how to (e.g. rule-based). Therefore *Blondie24* was not given any information about the game rules or how the score is calculated, and the only signal given was the reward corresponding to the gameplay result. For Chess, Fogel realized that the game complexity is significantly higher

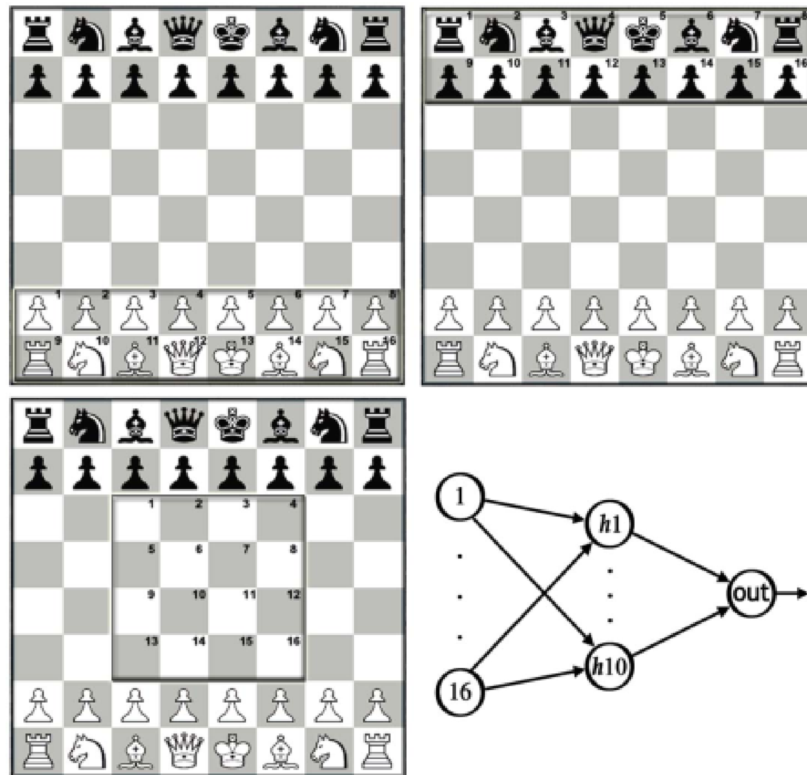


FIGURE 2.6: Front (top left), back (top right), middle (bottom left) areas and neural network structure (bottom right) as applied in [76]. The image is taken from page 2 of the publication

than Checkers, he employed three neural networks to create a player instead of one. Each network was responsible for a 16-tile specific area on the board: the *front*, *back* and *middle*. The front area is the first two rows on player side, while the back area is the two rows on the opponent's side and the middle area is the 16-tile square area at the middle of the board, as illustrated in Figure 2.6. All three networks produce one output each, called *Positional Value Table (PVT)* which each acted as a board position value evaluator. In this setup the hidden layer node size was fixed at 10 and the EA was adopted to evolve only connection weights.

Another board game with neuroevolution-based player is Backgammon, from the work done by Pollack et. al. [77]. They used a two-layer feedforward neural network to evaluate next move value, with hill-climbing to evolve the network topology. Their initial method started with setting all weights of the network to

zero, applied mutation to create a mutant network. Then these parent network and this mutant played the games for a number of times. The mutant would be selected as the next parent if it won more than half of the games. However, they found that sometimes a lucky inferior mutant happened to won more, therefore instead of neglecting the current parent, they mutated the parent to be a bit more similar to the mutant. For Othello, a neuroevolution using *Marker-based encoding* to encode network topology was proposed by Moriarty and Miikkulainen [78]. In this work, the network architecture and connection weights were translated into a sequence of integers in range $[-128, 127]$ and evolved with genetic algorithms. During the evolution, the networks played with a random move player and the results were stored. Then some of the best network individuals performed crossover with another randomly selected individuals, producing two offsprings for each mating. Then the usual selection procedure took place and the evolution was repeated. Their evolved networks were mostly able to learn the positional strategy (high score area falls around the corner) quite fast in the beginning, but more difficult to learn the mobility strategy (conquer as much areas as possible) that the experienced human players usually employed.

Apart from board games, neuroevolution has also been applied in many video games. An example of this is a well-known Pac-Man that Yannakakis and Hallam [79] used neuroevolution approach to generate ghost strategies. First they created a Pac-Man-like simulator with a simpler graphical layout. Four mazes, with different complexities (given in Figure 2.7, taken from page 4 of [79]), were designed for the experiment. Pac-man avatar was controlled by either of three bots that would attempt to avoid ghosts or collect power pills, depends on their embedded strategies and heuristic functions. For ghosts, they were being controlled by a fully-connected feed-forward neural network that receives state information and produces each ghost's next action from it. Their

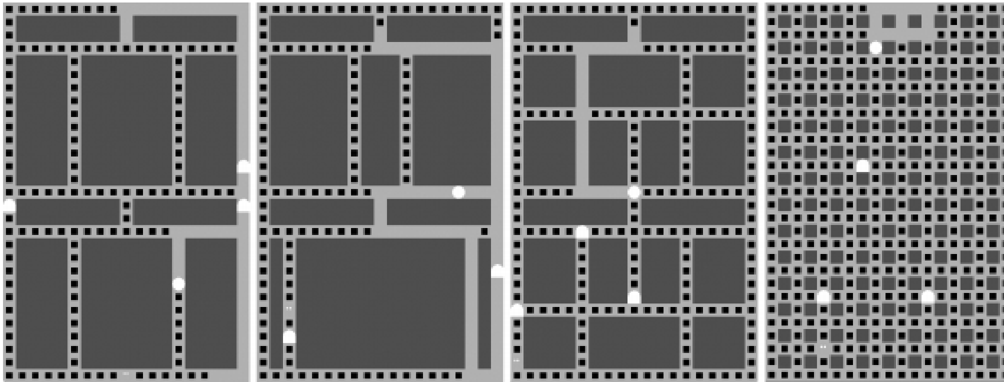


FIGURE 2.7: 4 Pacman maps with increasing complexity from left to right, employed in [79]. The image is taken from page 4 of the publication

objective was to evolve ghost behaviours that make the game interesting, measured by 3 criteria: game difficulty, diversity in opponent behaviour and opponent aggression level.

Togelius et. al. applied neuroevolution to Super Mario Bros. [80] to create an efficient avatar controller. Their experiment was done using either multilayer perceptron or recurrent network, with $(\lambda + \mu)$-ES at first and changed to a GP called *HyperGP* [81] after an initial experiment. The controllers evolved were able to clear the stages they were training on, but struggled in other stages, pointing out their lack of generalization. HyperGP is a GP version of *hyperNEAT*, which is a variant of *Neuroevolution of Augmented Topology (NEAT)* that the networks were used to generate weights. NEAT is a neuroevolution method that starts with the simplest topology and increase complexity during evolution. It also provides inexpensive methods to tackle different network structure recombination and mutation. Unlike hyperNEAT, hyperGP starts with more complex network structure. The fitness function of this Mario evolution was calculated from how far the avatar able to traverse to the right from the initial location. Stanley et. al. [82] invented a real time version of NEAT [83] and experimented its performance with their own developed game name NERO, which is a competitive robot fighting game. The game next moves were evolved online during

the gameplay, while carefully compromise between evolving the strategy individual and hastily reacting to the game. NEAT was also applied in other game domain such as racing, as done by Cardamone et. al. [84]. They focused on developing a car controller for *The Open Car Racing Simulator (TORCS)* by optimizing the performance on two tasks: fast and safety driving component and opponent overtaking & collision avoidance component. Input of the safety driving component network were taken from the TORCS API track and speed sensors while the opponent component received input from the opponent sensor. NEAT was used to evolve a good driving component first on the empty tracks, then it was extended to cover opponent avoidance and overtaking skill. They compared the results with other controllers included the manually-designed one and found that their method outperformed the manually designed controller in both component tasks.

Other Non-neuroevolution EAs can also be applied to create board game computer players. For Checkers, Hughes [85] presented an online co-evolution player that can select a suitable tree-like traversal path from the current state. Although this is similar to MCTS, it instead evaluated the best move without random rollouts simulation, but using real lookup because the branching is not too high. Each individual is a sequence of 100 real values in range $[0, 1]$ that each represent a move to be executed in order. A real number can be decoded to move by multiplying it with the size of the current legal move set, round the value up into an integer, then pick the move from the set with that index. Two populations were evolved simultaneously with each of them randomly selects a 'representative' to play against another from the opponent population. To clarify, the player in this case is the whole population, not just an individual. The game starts with each population picks the best-so-far individual, which is a sequence of moves, to compete. Then after one player makes a move, it will perform a 'look-up' simulation and applying evolution steps to the moves according to the current board state, then execute the next move from the best

individual so far. The fitness function favours the individuals that wins earlier, and loses later. This method is similar to the later-invented *Rolling Horizon Evolutionary Algorithm (RHEA)* [86].

RHEA is an online planning evolutionary algorithm developed by Perez et. al. [86]. It has been applied into game playing in various domains, for example general video game player in GVGAI [87] [88]. Samothrakis et. al. studied and compared the performance of using *Truncated Hierarchical Open-Loop Planing (T-HOLOP)* with another hierarchical open-loop planing called *EVO-P* as EAs in RHEA in three game-like continuous tasks. The first two tasks were benchmarking problems known as *Inverted Pendulum* and *Double Integrator*, while the third task was based on an arcade game *Lunar Lander*. T-HOLOP is a variant of *Truncated Hierarchical Optimistic Optimization (T-HOO)* which is designed for optimization in noisy environments. They found that it is possible to solve the mentioned tasks in rolling horizon setting, although the performance highly relied on the heuristic functions, which are domain-specific. For multiplayer games, Liu et. al. [89] applied RHEA to a simple two-player competitive game. Two variants of RHEA called *Rolling Horizon Genetic Algorithm (RHGA)* and *Rolling Horizon Coevolutionary Algorithm (RHCA)* were proposed as controllers and compared with Open-loop MCTS, *One-Step Look Ahead (1SLA)*, *Rotate-And-Shoot (RAS)* and *Random* controllers. RHCA performed the best among all mentioned controllers in their task. We employed RHEA as the representative of human player during parameter evolution in our second experiment.

RHEA is not the only online planning EA that can be used to develop a game playing controller. Justesen et. al. [90] presented a method called *Online Evolution* for multi-action adversarial games, which is a type of competitive turn-based games that each player perform a number of actions in each turn. As they claimed, Rolling Horizon techniques were designed to consider only the best possible actions of the opponents while planning instead of the actual actions, in opposite to their proposed method. Their approach was to

evolve a set of actions for only one turn, making use of all of the current state information and evaluate each action set at the end of the turn with the pre-defined heuristic function. They experimented this Online Evolution with three tree search methods including *Monte-Carlo Tree Search (MCTS)* with a two-player adversarial game named *Hero Academy*[9]. Some domain-specific modifications were necessary because blindly evolving the individuals is likely to cause non-legal moves. The mentioned four tested algorithms were put to play in all-pair head-to-head fashion for 100 matches in each pair, to remove the advantages of being the first move player. All controllers have 6 seconds to select actions for their turn. They found that Online Evolution performed best with the highest winning percentage at 80.5%. This Online Evolution was later combined with *Portfolio Greedy Search (PGS)* [91] to develop a controller for a micro version of *StarCraft*, by Wang et. al. [92]. The idea of PGS is to generate each unit move from a set of pre-defined scripts (portfolio), improving the scripts by using hill-climbing greedy search after a playout of the generate moves were executed. In Wang et. al. work, Online Evolution was used to evolve these scripts instead of hill-climbing. They tested this proposed method in a Java micro version of the game called *JarCraft*, compared with other three algorithms and found that theirs performed the best.

In this Chapter, previously proposed literatures that are related to our research were surveyed. There are some essential knowledge that the readers should understand in details for the best understanding of our work. These are written in the next chapter.

[9] `http://www.robotentertainment.com/games/heroacademy/`

Chapter 3

Background

In this chapter, all necessary background knowledge are described in details. This includes *Space Battle* along with its variant that was used in this project, GVGAI framework and Evolutionary Algorithms used in the experiments.

3.1 Space Battle

Space Battle is a two-player competitive game written in Java, earlier used in [89] to analyze co-ovolution approach for two-player games. Each player controls a convex-quadrilateral-shape spaceship that can steer (rotating the direction of the ship), thrust (moving forward by applying force in the front direction of the ship), and shoot missiles. The front side of a ship is indicated by a single-acute angle point. The first player controls the blue ship while the second player's ship is green. The game objective is to destroy another ship by shooting missiles to it, while also avoid being hit. Both players act simultaneously and have the same range of actions available, these include: turn clockwise, turn anticlockwise, thrust (move forward) and shoot a missile. Rotation, acceleration and shooting actions can be performed together in one game tick. Turn actions simply rotate the ship into the corresponding directions, without moving it from its current position. Therefore, the ship can only move forward when the thrust action is operated.

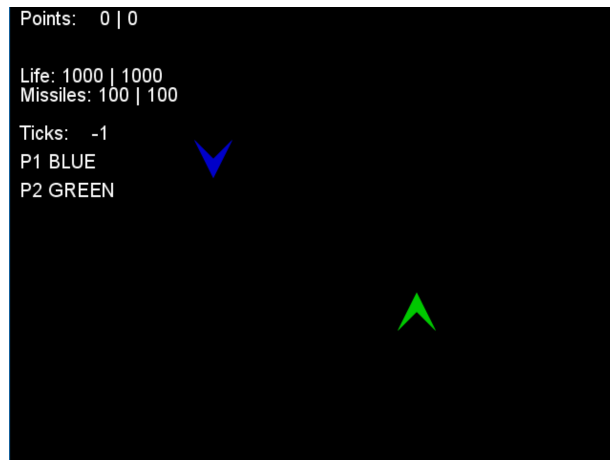


FIGURE 3.1: Space Battle Game Screenshot

When a player chooses to shoot, a round-shaped missile appears at the player's ship location and moves into the ship's forward direction with a specific velocity. Each player starts the game with 1000 lives that will be decreased by one if being hit by an opponent missile, which also give +1 score to the opponent. The game can end in either of two scenarios: one player has lost all 1000 lives or the maximum allowed time step is reached. The player who has more lives after the game ends is the winner.

The framework uses the same interface as the Two-Player General Video Game AI (GVGAI) framework [10], [34], therefore it easy to plug in GVGAI agents and use them for game testing. All controllers have access to a Forward Model (FM), which allows the agents to simulate possible future game states by providing an in-game action. Additionally, as the game is real-time, the agents only have 40ms for making a decision during a game step, with 1s for initialization. Figure 3.1 depicts Space Battle game screenshot at the beginning of the game.

Space Battle Evolved is a variant of *Space Battle* was designed specifically for this project. It introduces three new mechanics to the game:

1. **Black holes**

   Black holes in general refer to places in the space outside planets or stars

that have very high gravity and can pull everything, including light, into their centre. We applied its concept to create a game object with similar mechanism. A black hole in our game is a grey-colour circular area that has gravity-similar force that pulls some other game objects that enter the area into its centre point. In some situation, a player loses score eventually if staying inside a black hole penalty area.

2. **New Missiles**

   In the original Space Battle, there is only one missile type shot out directly from the front side of the ship. We have introduced two new missiles: *twin missile* and *bomb missile*. Twin missile consists of two normal missiles that can be shot in angle -45 and 45 degree from the front of the ship using only one 'shoot' action, and bomb missile is similar to the normal missile excepts that it can explode, causing damages in its area of explosion, if colliding with an opponent's game object.

3. **Missile Supply**

   In the original game, there are only 100 missiles available for each player. We introduced a new collectible game object called missile supply that contains 20 missiles which will be added into the player's available missiles once collected. This missile supply is shown as a small yellow star-shape in the game.

Figure 3.2 shows Space Battle Evolved with two black holes, twin missiles and a missile supply.

3.2 GVGAI Framework

GVGAI is a java-based framework for general video game playing (GVGP). It was first introduced in 2014 to support such research and has since become widely famous among the researchers in Game Artificial Intelligence field. The

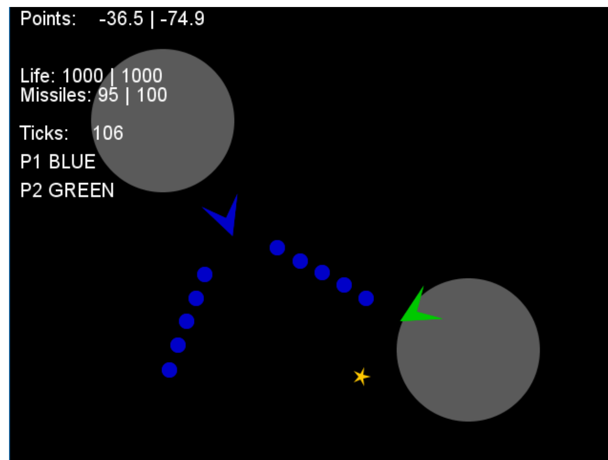


FIGURE 3.2: Space Battle Evolved

framework source code is available online[1] to download and execute. To fully understand how to read the game rules from their description language, Java-VGDL structure is explained next.

3.2.1 Video Game Description Language

All games in GVGAI framework are written in VGDL, which is first introduced by Ebner et. al. [93] and implemented by Tom Schaul [94]. The description language only supports non-physics 2D grid-based game interface, which means currently all games in GVGAI are subjected to that constraint. Nevertheless, the generality of VGDL concepts allows implementation of games in various categories, ranging from puzzles, mazes, to shooting games. To design a game with VGDL, at least two description files are needed. This includes *game description* and *level description*. Both description files are structured in tree-like hierarchy, to facilitate translation and de-serialization into Java game object classes.

Game description defines game rules, events that can happen and all available mechanics. In VGDL, game description has its game class name written on the first line, which can be either *BasicGame* or *GameSpace*. Game-specific parameter such as the number of players, input key handling mode and grid-cell square

[1] https://github.com/EssexUniversityMCTS/gvgai

size (in visualize mode) are defined next to the game class name. Following the game class in the first line, there are five possible instruction blocks in a game description:

- **LevelMapping**: defines which ascii character is used to represent which sprite in the game.
- **SpriteSet**: defines all sprite types, which sprites in the same category can also be defined in hierarchical format. A sprite can either be an *avatar*, a *resource*, a *portal*, *static*, spawn *from avatar* or *movable*.
- **InteractionSet**: defines the events to happen if two sprites collide (move into the same grid tile at the same time step. A sprite always collide with *TIME* in every time step. Therefore this can also be used to define time-based events.
- **TerminationSet**: defines all conditions that leads to game termination, along with the winning result in each case.
- **ParameterSet**: defines all game parameters that can be parameterize in game design track and experiment.

Both *BasicGame* and *GameSpace* always contain the first four instruction blocks, while only *GameSpace* has *ParameterSet* block.

Level description in VGDL defines a game level layout, which are initial positions of game object sprites. Each level file is a text file that each line has the same length, as seen in Figure 3.3 a and b, resulted in the actual game layout in Figure 3.3c and 3.3d respectively. VGDL game must have one game description and at least one level description, in GVGAI there are 5 level description files for each game.

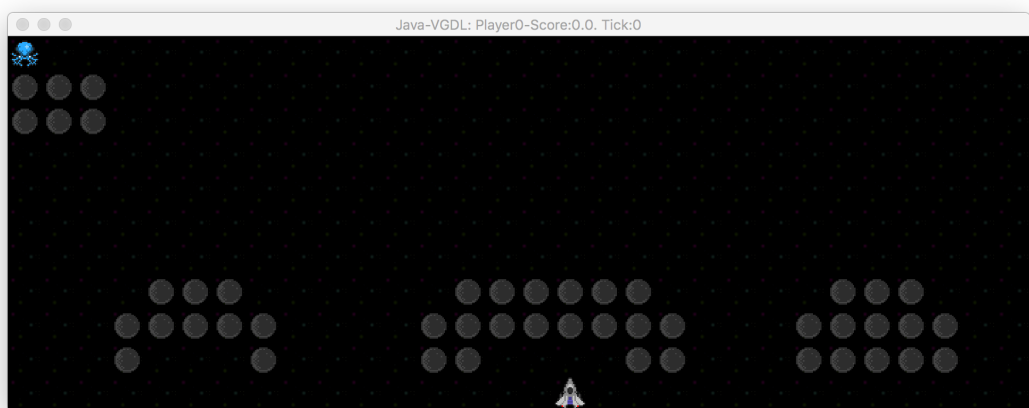
A GVGAI avatar can have up to 5 in-game actions, which are 4 directional movement (*ACTION_UP*, *ACTION_DOWN*, *ACTION_LEFT* and *ACTION_RIGHT*) that will move the avatar into the adjacent grid tile in such direction. Some

```
1.............................
000...........................
000...........................
..............................
..............................
..............................
..............................
....000......000000.....000...
...00000....00000000...00000..
...0...0....00....00...00000..
................A.............
```

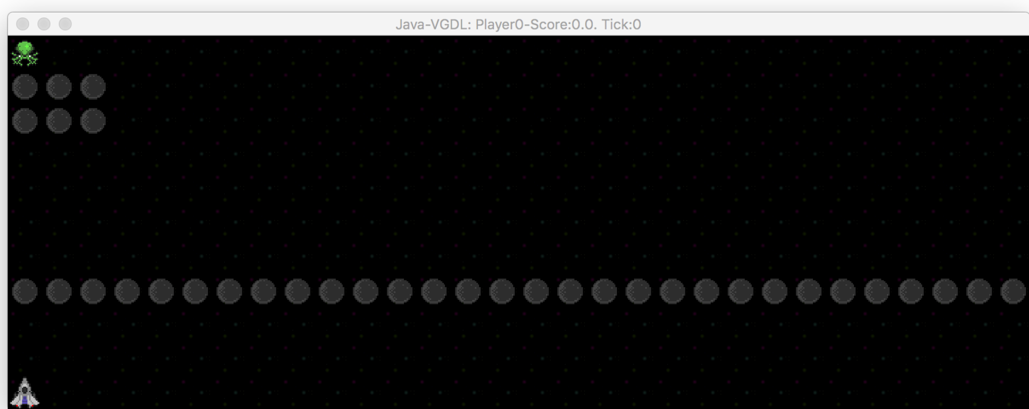
(A) Level 0 description

```
2.............................
000...........................
000...........................
..............................
..............................
..............................
..............................
000000000000000000000000000000
..............................
..............................
A.............................
```

(B) Level 4 description



(C) Level 0 screenshot



(D) Level 4 screenshot

FIGURE 3.3: Single-player Aliens level description language and screenshots, level 0 and 4

movement action are not available in some games. The fifth action is *ACTION_USE*, which will act differently according to avatar type.

GVGAI provides both single-player and multi-player (two-player at the moment) games, with two physics systems: grid-based and continuous. In grid-based physics system games, the game objects (avatars, npcs, push-able objects, etc.) are movable (or pushable) in 4 directions to a non-occupied grid cell if the corresponding action is performed, while in continuous physics system, the movement actions would be affected by game-specific physics forces, which for instance could be gravity force, steering or thrusting force. A game gives reward signal to the players based on one of two score systems: *Binary* and *Incremental*. Binary score system only gives a positive score when the player has won the game, and in some occasion negative score if the player avatar died during the game to distinguish this from losing after time out. Incremental score system provides a change in score (either positive or negative) from some events that happened throughout the gameplay. In the original 2014 GVGAI framework document they suggested another system called *discontinuous* in which certain events lead to significantly higher changes in score. However, after more games has been added into the framework, it is unclear whether how much 'higher changes' of score should be considered as discontinuous system, I would therefore categorize every games with events leading to the changing of score before the end of the game as using incremental system.

All three games we selected for our second experiment are single-player shooting games with grid-based physics and incremental score system. This makes them be good testbeds for the experiment, as the agents play in these game are most likely to obtain score trends in the same style as the functions we picked. The games are described next.

3.2.2 Selected Games

Seaquest, Waves and Defender were chosen to do parameterization from predefined score trend function.

1. **Seaquest**

   Seaquest is a single-player GVGAI game that was inspired by an arcade game with the same name in the Atari 2600 framework[2]. In this game, a player controls a submarine aiming to rescue underwater divers while defending itself from the aggressive fish-shaped underwater animals by shooting them with torpedoes. The submarine has an oxygen level bar that will continuously decrease with the time it is underwater, and can be refilled once it moves to the surface. In the Atari version, the player wins a level if 3 divers are rescued, loses one (out of the initial three) life if being hit by the fish-shaped animals (as well as their bullets) or runs out of oxygen. In GVGAI version, the player wins if the avatar survived until the maximum timestep is reached.

   Seaquest VGDL game description *SpriteSet* instruction block is shown in Figure 3.4. The description is in format *Name* > *Type* [*Optional parameters*]. From the description, there are 6 main sprite classes: *sky*, *water*, *saved*, *holes*, *moving* and *crew*. *Hole* has three sub-classes: *sharkhole* that will spawn *shark* sprites, *whalehole* that will spawn *whale* sprites and *diverhole* that will spawn *diver* sprites. *Diverhole* has two sub-classes, *normaldiverhole* and *oftendiverhole* that will spawn *diver* sprites with different probability. *Moving* sprite has 4 sub-classes: *avatar*, *torpedo*, *fish* and *diver*. There are three types of *fish* sprites: *shark*, *whale* and *pirana* (which is piranha). Grouping sprite types makes it easier to define an interaction set, which is shown in Figure 3.5. The *InteractionSet* block is in format *Sprite1 Sprite2* > *Events*, which means the events *Events* will happen if *Sprite1* collides with *Sprite2*. It can

[2] http://www.retrogames.cz/play_221-Atari2600.php?language=EN

be seen that, grouping *shark*, *whale* and *pirana* as *fish* helps reducing the interaction sets, as the events will occur to the all sprites under a parent class if it has been explicitly defined in the interactions. Each line can be translated as follows.

- Every 26 time step, *avatar* health (in this case oxygen level) will decrease by one.
- If *EOS* (a global sprite type means end of screen) is collided by the *avatar*, or if a *diver* collides with *sky*, the *avatar* /*diver* will be pushed back to the cell they were on the previous time step. This is to prevent the *avatar* from moving out of the screen, and to prevent *diver* from going to the surface by themselves.
- If a *fish* leaves the screen, they will disappear from the game.
- If a *fish* collides with a *torpedo*, they both will disappear from the game, and the player receives +1 score.
- If the *avatar* collides with a *fish*, the avatar is killed.
- If the *avatar* moves to the *sky*, it gets 1 more oxygen level (health point), and the *crew* will be changed to *saved*, with limit crew size = 4. The player will get 1000 points for each *saved* he/she has.
- If the *avatar* collides with a *diver*, the *avatar* has one more *crew* resource and the *diver* is removed. This is equivalent with changing the *diver* into *crew*, but since *crew* is considered as a resource here, the *changeResource* command is used and the *diver* is explicitly killed off.

The *TerminationSet* of Seaquest is given in Figure 3.6, which describes that the player loses if the *avatar* was killed, and wins if he/she has survived until time step 1000.

2. **Waves**

Waves is an alien fighting GVGAI game that the player controls a spaceship trying to survive from a big alien attacking wave. Aliens are spawned

```
SpriteSet

      sky > Immovable img=oryx/backLBlue
      water > Immovable img=newset/water2
      saved > Immovable color=LIGHTGREEN
      holes > SpawnPoint color=LIGHTGRAY img=newset/whirlpool2
portal=True
            sharkhole > stype=shark prob=0.01
            whalehole > stype=whale prob=0.005
            diverhole > stype=diver
                  normaldiverhole > prob=0.005
                  oftendiverhole > prob=0.025

      moving >
            avatar > ShootAvatar color=YELLOW stype=torpedo
            img=newset/submarine healthPoints=18 limitHealthPoints=20
            torpedo > Missile color=YELLOW img=oryx/bullet1
            fish >
                  shark > Missile orientation=LEFT speed=0.25
                  color=ORANGE img=newset/shark2
                  whale > Bomber orientation=RIGHT speed=0.1 color=BROWN
                  stype=pirana prob=0.02 img=newset/whale
                  pirana > Missile orientation=RIGHT speed=0.25 color=RED
                  shrinkfactor=0.6 img=newset/piranha2
            diver > RandomNPC color=GREEN speed=0.5 img=newset/diver1
cons=2
      crew > Resource color=GREEN limit=4
```

FIGURE 3.4: Seaquest *SpriteSet* game description

```
InteractionSet

      avatar TIME > subtractHealthPoints timer=26 repeating=True
      EOS avatar diver sky > stepBack
      fish EOS > killSprite
      fish torpedo > killBoth scoreChange=1

      avatar fish > killSprite
      avatar sky > addHealthPoints value=1
      avatar sky > spawnIfHasMore resource=crew stype=saved limit=4
spend=4
      saved sky > killSprite scoreChange=1000

      avatar diver > changeResource resource=crew
      diver avatar > killSprite
```

FIGURE 3.5: Seaquest *InteractionSet* game description

```
TerminationSet

      SpriteCounter stype=avatar limit=0 win=False
      Timeout limit=1000 win=True
```

FIGURE 3.6: Seaquest *TerminationSet* game description

```
SpriteSet

    background > Immovable img=oryx/space1 hidden=True
    asteroid > Immovable img=oryx/planet
    missile > Missile
        rock > orientation=LEFT speed=0.95 color=BLUE img=oryx/orb3
        sam > orientation=RIGHT color=BLUE speed=1.0 img=oryx/orb1
        shrinkfactor=0.5
        laser > orientation=LEFT speed=0.3 color=RED
        shrinkfactor=0.75 img=newset/laser2_1
    portal >
        portalSlow > SpawnPoint stype=alien cooldown=10 prob=0.05
        img=newset/whirlpool2 portal=True
        rockPortal > SpawnPoint stype=rock cooldown=10 prob=0.2
        img=newset/whirlpool1 portal=True
    shield > Resource color=GOLD limit=4 img=oryx/shield2
    avatar > ShootAvatar color=YELLOW stype=sam speed=1.0
    img=oryx/spaceship1 rotateInPlace=False

    alien > Bomber color=BROWN img=oryx/alien3 speed=0.1
    orientation=LEFT stype=laser prob=0.01
```

FIGURE 3.7: Waves *SpriteSet* game description

from their spawn points and moving considerably fast to the avatar direction. Alien liars also constantly shoot missiles and laser to the player, in which the player will lose one health if colliding with each of them. The avatar can regain health by collecting shields that dropped from destroyed lasers, after shooting missiles at them.

Wave game description *SpriteSet* is given in Figure 3.7. There are 7 main sprites in this game: *background*, *asteroid*, *missile*, *portal*, *shield*, *avatar* and *alien*. *Missile* can be either *rock*, *sam* or *laser* while *portal* has 2 sub-classes: *portalSlow* and *rockPortal* that will spawn *alien* and *rock* respectively.

Waves *InstructionSet* can be found in Figure 3.8. It can be translated as follows:

- The *avatar* are not allowed to leave the screen, while *alien*s and *missile*s will be destroyed once they have left the screen.

```
InteractionSet
     avatar EOS > stepBack
     alien EOS > killSprite
     missile EOS > killSprite
     alien sam > killBoth scoreChange=2
     sam laser > transformTo stype=shield killSecond=True

     avatar shield > changeResource resource=shield value=1
     killResource=True

     avatar rock > killIfHasLess resource=shield limit=0
     avatar rock > changeResource resource=shield value=-1
     killResource=True

     avatar alien > killIfHasLess resource=shield limit=0
     avatar alien > changeResource resource=shield value=-1
     killResource=True

     avatar laser > killIfHasLess resource=shield limit=0
     avatar laser > changeResource resource=shield value=-1
     killResource=True

     asteroid sam laser > killSprite
     rock asteroid > killSprite
     alien asteroid > killSprite
     laser asteroid > killSprite
     avatar asteroid > stepBack
```

FIGURE 3.8: Waves *InstructionSet* game description

- If an *alien* collides with a *sam*, both will be destroyed and the player score increases by 2.
- If a *sam* hits a *laser*, spawns a *shield* (by changing the *sam* to a *shield*), and destroy the *laser*.
- Collecting a *shield* will increase the *avatar* resource by 1.
- The *avatar* loses one resource if hit by a *rock*, an *alien* or a *laser*, and is killed if no resource left.
- An *asteroid* is destroyed if hit by a *sam* or a *laser*.
- A *rock*, an *alien* and a *laser* will be killed once collides with an *asteroid*, while the *avatar* cannot move to any cells with an *asteroid*.

```
TerminationSet
    SpriteCounter stype=avatar limit=0 win=False
    Timeout limit=1000 win=True
```

FIGURE 3.9: Waves *TerminationSet* game description

Waves game termination condition (Figure 3.9) is the same as Seaquest, which the player wins if the *avatar* is still alive until 1000 time step, and loses if the *avatar* is killed.

3. **Defender**

Similar with Seaquest, Defender is also inspired by the game with the same name in Atari 2600 framework[3]. In this game, the player plays as an armed aircraft trying to protect cities from alien assault. Aliens occasionally drop bombs to destroy the cities below. The aircraft can shoot missiles at aliens to kill them before they successfully bombarding all cities. In GVGAI version, aliens move from their spawn points horizontally in one direction and is harmless to the avatar aircraft. The avatar has a limit amount of missile resource, which can be reloaded by collecting a supply pack that is constantly falling down from above.

GVGAI Defender *SpriteSet* game description is given in Figure 3.10. There are 10 main sprite classes in this game: *floor*, *city*, *avatar*, *missile*, *alien*, *portal*, *portalAmmo*, *supply*, *bullet* and *wall*. *Missile* has 2 sub-classes: *sam* and *bomb*, while *portal* can be either *portalSlow* or *portalFast*.

The *InstructionSet*, depicted in Figure 3.11, can be translated as follows:

- The *avatar* cannot leave the screen, nor move to a *city* or a *wall*.
- Once left the screen, an *alien* and a *missile* will be destroyed.
- A *city* can be destroyed by a *bomb*, which will also decrease the player score by 1.

[3] http://www.free80sarcade.com/2600_Defender.php

```
SpriteSet
    floor > Immovable img=oryx/space1 hidden=True
    city > Immovable color=WHITE img=newset/city1
    avatar > ShootAvatar color=YELLOW ammo=bullet stype=sam
    img=oryx/spaceship1 rotateInPlace=False

    missile > Missile
        sam > orientation=RIGHT color=BLUE img=oryx/bullet1
        bomb > orientation=DOWN color=RED speed=0.5 img=newset/bomb
        shrinkfactor=0.6
    alien > Bomber orientation=LEFT stype=bomb prob=0.04 cooldown=4
    speed=0.6 img=oryx/alien3

    portal > SpawnPoint stype=alien cooldown=10 invisible=True
    hidden=True
        portalSlow > prob=0.2
        portalFast > prob=0.5

    portalAmmo > SpawnPoint stype=supply cooldown=10 prob=0.15
    invisible=True

    supply > Missile orientation=DOWN speed=0.25 img=oryx/goldsack
    bullet > Resource limit=20
    wall > Immovable img=oryx/wall1
```

FIGURE 3.10: Defender *SpriteSet* game description

- A *city* can also be destroyed once hit by a *sam* or an *alien*, but the score will stay the same in this case.
- A *supply* is destroyed if collides with an *alien*.
- Once an *alien* was shot by a *sam*, they both will be destroyed, and the player score will increase by 2.
- A *supply* cannot pass through a *city* or a *wall*, or another *supply*, instead it will be stacked up.
- The *avatar* resource (*sam*) will be increased by 5 if it collects a *supply*.

Defender game termination condition is the same as Waves and Seaquest, with an additional case of losing if all *city* has been destroyed. This *TerminationSet* is shown in Figure 3.12.

```
InteractionSet
    avatar EOS city wall > stepBack
    alien EOS > killSprite
    missile EOS city > killSprite
    city bomb > killSprite scoreChange=-1
    city sam alien > killSprite
    supply alien > killSprite
    alien sam > killBoth scoreChange=2
    supply supply > stepBack
    supply wall city > stepBack pixelPerfect=True
    avatar supply > changeResource resource=bullet value=5
    killResource=True
```

FIGURE 3.11: Defender *InteractionSet* game description

```
TerminationSet
    SpriteCounter stype=avatar limit=0 win=False
    SpriteCounter stype=city limit=0 win=False
    Timeout limit=1000 win=True
```

FIGURE 3.12: Defender *TerminationSet* game description

3.2.3 Controllers

In our first experiment, we applied six controllers, all excepts *Rotate and Shoot* taken from the GVGAI framework to use as either player-substitutions or enemy players. This includes *Do Nothing*, *Random*, *One Step Look Ahead (1SLA)*, *Rotate and Shoot (RAS)*, *Monte Carlo Tree Search (MCTS)* and *Microbial Evolutionary Algorithm (MEA)*. In the second experiment, the chosen games were played and evolved using a GVGAI controller called *Rolling-Horizon Evolutionary Algorithm (RHEA)* , and the evolved game parameters were validated using MCTS. RHEA and MCTS controllers were chosen to do this task because they always perform well compared with the other sample controllers in the framework, and also because of their 'score hunger' nature. Specifically, these two controllers are using heuristics that taking the game score into account as the second priority, with only the game winning result is more important.

To develop a GVGAI controller, a Java class named *Agent* is needed, and it must extends a Java class named *AbstractPlayer*. Any non game-specific setup

should be implemented in the agent's constructor as the agent would be created before it is assigned any games to play. After entering a gameplay state, the *act* method would be called for the agent to return an action for that current time step. Both agent constructor and *act* method receive two parameters, a *StateObservation* class which encapsulates all accessible information and a forward model of that game state, and a *ElapsedCpuTimer* that the agent can query time-related information. An agent in non-learning track is allowed to 'simulate' the possible future outcomes of the game by passing an action to the *forward model* (by calling StateObservation.advance(Action) method). With the information obtained from this simulation the agent should plan the action carefully, as it is given no chance to replay the game.

GVGAI allows 1 second CPU time for agent's creation (constructor) and 40 milliseconds for returning an action in the *act* method. Violating any of these time constraints will result in the agent being disqualified for that game play, and receive a huge negative score. All controllers' algorithm description and, for advance controllers only, their implementation in GVGAI framework are explained in details next.

1. **Do Nothing**

   As the name suggests, the controller always return no action in every time step. It was used as an enemy player parameter to be evolved in our variant of Space Battle.

2. **Random**

   This controller returns random action in every game step. It was used as an enemy player parameter to be evolved in our variant of Space Battle.

3. **One Step Look Ahead (1SLA)**

   This is the simplest controller that accesses the forward model. It simulates possible next steps by passing each action to the forward model, and apply the action that leads to the best outcome, which can be determined from its

internal heuristic. In this case the heuristic considers only the game score. It was used to portray amateur player in Space Battle Evolved experiment, because it is powerful only if the enemy is right in front of the ship.

4. **Rotate and Shoot (RAS)**

   Rotate and Shoot is the only non-general agent using in our experiment. It employs a simple-but-powerful strategy of rotating clockwise and shooting missiles around repeatedly. In the Space Battle game with unlimited missile, this controller maybe unbeatable. However, in Space Battle Evolved, only 100 missiles are provided in the beginning. RAS was used to represent intermediate player since it is invincible in at least the first 100 game steps.

5. **Microbial Evolutionary Algorithm (MEA)**

   This controller is known under the name *sampleGA* in GVGAI competition. It first randomly generate a population of individuals encoded as sequences of in-game actions, then performs selection using a microbial tournament, from which an offspring is generated by crossover operation. This newly generated individual is then mutated, and is evaluated its fitness value. Then some of the best individuals are retained as a part of the next generation. This process is reiterated until the budget limit is reached (e.g. time, memory or specific number of iterations). This controller was applied as an enemy player parameter to be evolved in our variant of Space Battle.

6. **Rolling Horizon Evolutionary Algorithm (RHEA)**

   RHEA was first introduced by Perez et. al. [86] as an online planning game player for Physical Travelling Salesman (PTSP) problem. Rolling horizon (also known as 'Model Predictive') refers to a planning behaviour that plans a sequence of actions from the current point to the limit foreseeable future, then performs the first action on of the sequence and then

replan. Perez et al. considered this sequence of actions as an individual in evolutionary algorithm domain. It was implemented as a sample controller for GVGAI framework and has achieved promising performance (although often ranked below MCTS - as claimed in [87]) in average.

RHEA algorithm can be summarized as follows:

(a) Randomly picks a set of individuals (in this case a sequence of actions, each randomly assigned from a set of available actions in the game)

(b) Evaluates fitness values of each individual and sorts them in descending order

(c) Performs crossover and mutation to all individuals excepts the best one.

(d) Evaluate every new population individuals, if the fitness value of a newly created individual is higher than the best one, replace the best one with this.

(e) Repeat step (c) and (d) until the time almost runs out.

(f) Return the first action of the best individual.

(g) Repeat (a)-(f) until the game ends.

Algorithm 1-3 shows how RHEA controller implements GVGAI *act* method. The *evaluate* statement means simulating each action in the individual sequentially using the forward model and return the fitness value according to the heuristic function it is using. *Crossover* and *mutate* methods refer to two well-known techniques for individual evolution in genetic algorithm.

7. **Monte Carlo Tree Search (MCTS)**

Monte Carlo Tree Search (MCTS) is a tree-based planning algorithm that was designed to tackle problems with huge branching factors. Its main strength is the ability to explore in only some sections of the space that are most likely to give promising outcomes, while neglecting other non (or

Algorithm 1 RHEA *act* method

INPUT: StateObservation *stateObs*, **ElapsedCpuTimer** *timer*
OUTPUT: an available action

1: $pop \leftarrow init_pop(stateObs)$
2: **while** enough time left from *timer* **do**
3: $pop \leftarrow runIteration(pop, stateObs)$
4: **end while**
5: **return** first action of the best individual in pop

Algorithm 2 init_pop

INPUT: StateObservation *stateObs*
OUTPUT: a population of action sequences

1: $pop \leftarrow \emptyset$
2: $POP_SIZE \leftarrow$ maximum allowed population size
3: $INDIV_LEN \leftarrow$ longest simulation allowed
4: **while** enough time left **and** $length(pop) < POP_SIZE$ **do**
5: $individual \leftarrow$array of $INDIV_LEN$ random action indices
6: $individual[fitness] \leftarrow evaluate(individual)$
7: add *individual* to *pop*
8: **end while**
9: $sort(pop)$ in descending order of $fitness$ value
10: **return** pop

almost non) informative branches. Generally in planning-based domains with lookahead ability, the most challenging task is to balance between *exploration* and *exploitation* while selecting next area to simulate. Exploration means spending time to search more in unseen areas, while exploitation is to search more in the known areas to gain more or validate the knowledge of such areas (especially in stochastic domains such as GVGAI, one-time simulation of a state does not give all possible outcomes). MCTS tackles this by doing one-level breadth-first-search at a time (expanding all children nodes for this state only) for a new state, and do one-part depth-first-search with a selected children node (repeatedly simulate future steps by selecting random actions until the maximum depth, or the termination state, is reached). This is called *rollout*, then the final outcome of this rollout is back-propagated through expanded nodes. Then the process of selecting

Algorithm 3 runIteration

INPUT: Population *pop*, **StateObservation** *stateObs*
OUTPUT: an evolved population of action sequences

1: $INDIV_LEN \leftarrow$ longest simulation allowed
2: $RE_EVALUATE \leftarrow$ whether re-evaluation should be done
3: **if** $RE_EVALUATE$ **then**
4:     **for all** *individual* in *pop* **do**
5:         **if** enough time **then**
6:             $pop[individual] \leftarrow evaluate(individual)$
7:         **else**
8:             stop
9:         **end if**
10:     **end for**
11: **end if**
12: **if** $length(pop) > 1$ **then**
13:     **for all** non preserved individuals at position i in *pop* **do**
14:         **if** enough time **then**
15:             $newIndiv \leftarrow crossover()$
16:             $newIndiv \leftarrow mutate(newIndiv)$
17:             $newIndiv[fitness] \leftarrow evaluate(newIndiv)$
18:             $pop[i] \leftarrow newIndiv$
19:         **else**
20:             stop
21:         **end if**
22:     **end for**
23:     $sort(pop)$ in descending order of $fitness$ value
24: **else if** $length(pop) = 1$ **then**
25:     $newIndiv \leftarrow$array of $INDIV_LEN$ random action indices
26:     $newIndiv[fitness] \leftarrow evaluate(newIndiv)$
27:     **if** $newIndiv[fitness] > pop[0][fitness]$ **then**
28:         $pop[0] \leftarrow newIndiv$
29:     **end if**
30: **end if**
31: **return** *pop*

to-explore-next area is done by computing UCB values, using UCB1 algorithm, of each children node at a time and pick the one with the highest value to explore. UCB1 equation is given in equation 3.1.

$$a^* = \arg\max_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{3.1}$$

While $(a*)$ is the best action, which would be selected as the one that maximizes the UCB1 equation, $Q(s,a)$ is the estimated reward of taking action a from state s, $N(s)$ is the number of times s has been visited, $N(s,a)$ represents how many times action a has been chosen from s and$C$ is a constant that balances between exploration and exploitation, with a value typically set as$\sqrt{2}$ when rewards are bounded in $[0,1]$.

The mentioned process of MCTS planning can be divided into 4 phases:

- **Selection**: Repeatedly selecting a known child node from root using UCB1 equation (equation 3.1) until a node with some unexpanded children is found.
- **Expansion**: A new unexplored child node is added to the tree.
- **Simulation**: A rollout is done from the newly added child node until the termination condition is reached.
- **Back-propagation**: The outcome of the rollout is repeatedly back-up through the parental paths from the newly added node to the root node.

Figure 3.13 depicted these 4 phases, the image was taken from page 6 of an MCTS survey paper by Browne et. al. [41]. Algorithm 4 summarizes how MCTS is implemented in GVGAI *act* method. *SelectAndExpand* refers to **Selection** and **Expansion** phase of MCTS, returning the newly expanded child node, while *rollOut* and *backup* are **Simulation** and **Back-propagation** phase respectively. In our work, MCTS was used as a representation of skilful player in Space Battle Evolved experiment, and used in validation phase of the GVGAI experiment.

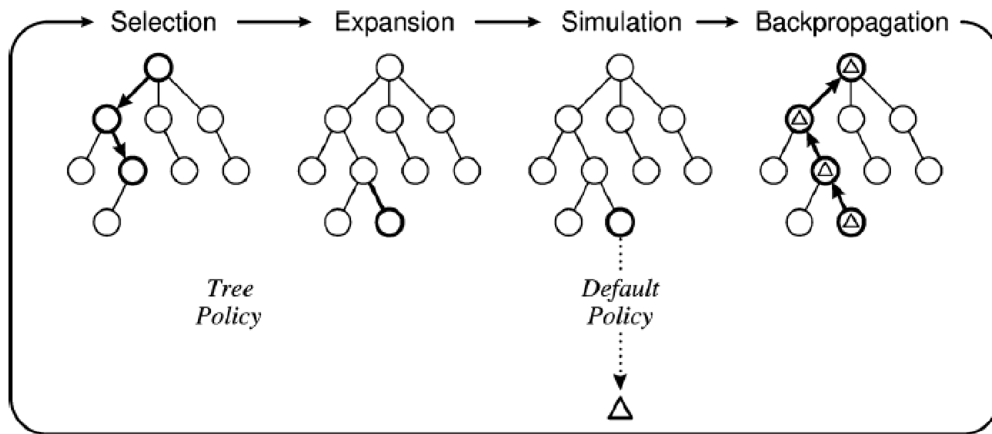


FIGURE 3.13: MCTS searching steps, taken from page 6 of [41]

Algorithm 4 MCTS *act* method

INPUT: StateObservation *stateObs*, **ElapsedCpuTimer** *timer*
OUTPUT: an available action

1: $mctsPlayer \leftarrow$ a new tree with root is the state *stateObs*
2: $rootState \leftarrow mctsPlayer[root]$
3: **while** enough time **do**
4: $\quad selectedNode \leftarrow selectAndExpand(rootState)$
5: $\quad value \leftarrow rollOut(selectedNode, rootState)$
6: $\quad backup(value)$
7: **end while**
8: **return** most visited action in $rootState$

3.3 Evolutionary Algorithms

There were three evolutionary algorithms applied in our two experiments. The first one *Random Mutation Hill Climber (RMHC)* along with its variant *Biased Mutation RMHC* were used in the Space Battle Evolved experiment only as baseline algorithms to compare with *N-Tuple Bandit Evolutionary Algorithm (NTBEA)*. The second experiment with GVGAI games employ only NTBEA as the EA to parameterize games.

3.3.1 Random Mutation Hill Climber (RMHC)

RMHC is the simplest version of evolutionary algorithms that has only one individual in the population. It starts with random individuals, then randomly

selects one gene for mutation uniformly. The fitness value of this new individual is calculated and compared with the previous one. The better individual is kept for the next iteration of the algorithm. These steps are repeated until the termination condition is reached. In the implementation used for the Space Battle Evolved experiment, both the parent and the offspring are evaluated in each generation.

3.3.2 Biased Mutation RMHC

Biased Mutation RMHC (B-RMHC) was inspired by the idea that different parameters affect the change in fitness values at different rates. That is, modifying one parameter might significantly affect the fitness value more than others. Therefore, a biased mutation towards more interesting parameters was used to obtain more diverse games and speed up evolution. The algorithm is made up of two parts: parameter pre-processing and actual evolution.

- **Pre-processing**

  In this part, the parameters were divided into two groups, with the black hole cell parameters (Group B) being separated from the rest (Group A). For Group A's pre-processing, the parameters were first randomly assigned. Then, for each of them, the *importance* metric was calculated using standard deviation from the fitness in N tests, where N is the total number of values the parameter tested can take. For each value, the game taking the new parameter list was evaluated using the same fitness function employed during evolution. This assessment is based on the assumption that larger differences in fitness lead to larger standard deviation values.

  For Group B, the parameters were analyzed separately for each possible grid size value, starting from all the black hole cells being empty and evaluating the effect of enabling a black hole in each cell. Similarly to Group A, the standard deviation of the fitness values resulted from each cell's evaluation was used to rank these parameters.

Algorithm 5 Biased Mutation Pre-processing (MutPrep)

```
1:  Input: game parameter list params
2:  Output: sorted lists of important parameters
3:  BEGIN
4:      PriorityQParams ← ∅
5:      PriorityQBH ← ∅
6:      ParamsN ← GroupA parameters
7:      FOR EACH p in paramsN
8:          value ← ∅
9:          rand ← randomly assign other parameter values
10:         FOR EACH possible value v of p
11:             rand[p] ← v
12:             add fitness(rand) to value
13:         PriorityQParams[p] ← SD(value)
14:     rand← randomly assign other parameter values
15:     rand← disable all black holes
16:     FOR gSize ∈ {0, 1, 2, 3, 4}
17:         bhpriorityQ ← ∅
18:         fitnessOff ←fitness(rand)
19:         FOR b = 1 to gSize^2
20:             enable black hole at position b in rand
21:             bhpriorityQ[b] ← fitnessOff − fitness(rand)
22:         PriorityQBH[gSize] ← bhpriorityQ
23:     RETURN PriorityQParams, PriorityQBH
24: END
```

Pre-processing step outputs two list of parameters sorted by how much they affect the game fitness value. Details of this algorithm can be seen in Algorithm 5. This ordering was then used in the evolution step.

- **Evolution**

  A softmax function was used to bias parameter selection at the beginning of the evolution process. This ensures that more important parameters are more likely to be selected to mutate. After that, the algorithm follows the the similar steps of simple RMHC.

3.3.3 N-Tuple Bandit Evolutionary Algorithm (NTBEA)

N-Tuple Bandit Evolutionary Algorithm (NTBEA), first applied in Game AI by Simon Mark Lucas, is an evolutionary algorithm that was designed for a noisy domain task, e.g. one fitness evaluation does not give the correct value of the

individual, instead the appropriate approach is to find average value from multiple evaluations. NTBEA stores the statistical information of the evaluated fitness value of such individual. These information are being stored using N-Tuple structure, therefore they are real-time accessible. Given these characteristics, NTBEA is robust to noise and fast in individual optimizing.

All of our selected GVGAI games, and also both controllers, are stochastic. Therefore to do game parameter tuning by evolutionary approach, a fast and noise-robust evolutionary algorithm is necessary.

NTBEA is composed of two important ideas. Firstly, it uses N-Tuple structure to store statistical information of the evaluated parameter sets. This N-Tuple structure application literature can be found in [95]. In this implementation, N-Tuple is a set of lookup tables which using integer parameter as keys to access the stored statistics. Another important component of our N-Tuple EA is that it uses bandit approach (UCB1, equation 3.1) to select the best parameter set found so far. N-Tuple Bandit EA operates as follows:

1. Randomly selects a parameter set as the initial individual.
2. Evaluates the fitness value (and other statistical information in later iterations) of that individual, stores it in the lookup table.
3. Mutates the individual to generate a set of 'neighbour' (the individuals that shares all but one parameter values of such individual).
4. Calculates the UCB value of each neighbour individuals from the information stored in their lookup tables.
5. Selects the neighbour with the highest UCB value as the next individual.
6. Repeats 2-5 with the selected next individual until the termination condition is reached.

To clarify how N-Tuple Bandit EA evolution steps work, See the following example.

Example 3.3.1. Suppose we would like to evolve a game parameter set of size 4 with, with all of them have the same set of possible values equals to $1, 2, ...10$. The state space size is $10^4 = 10,000$, and the game is stochastic. Suppose that the fitness value of each individual is defined by the score after the game ends proportional to 100 (e.g score 58 will be $\frac{58}{100} = 0.58$), N-Tuple Bandit EA steps are as follow:

- Selects a random individual value 2487, plays a game with this setup and obtains score = 30, hence fitness = 0.3.
- Creates a lookup table *table* and set
  - $table[2487][average] = 0.3$ and $table[2487][count] = 1$.
  - $table[2xxx][average] = 0.3$ and $table[2xxx][count] = 1$, while [2xxx] means keeping the statistics of all parameter values that start with 2.
  - $table[x4xx][average] = 0.3$ and $table[x4xx][count] = 1$.
  - $table[xx8x][average] = 0.3$ and $table[xx8x][count] = 1$.
  - $table[xxx7][average] = 0.3$ and $table[xxx7][count] = 1$.

  It is possible to also set the values of $table[24xx]$ and $table[248x]$ or other combinations of them, but it will consume more computational resources. In the experiment we only use the 1-Tuple (with one value specified, such as $x4xx$, $xx8x$ and $xxx7$ in this example) and the N-Tuple (with all value specified, such as 2487) and omitted 2, 3, ... N-1 Tuple.
- Standard error and other statistical information are also stored, but it is neglected to keep this example concise.
- Mutates 2487, obtains 7487, 2987, 2407 and 2481 as neighbours.
- Calculates UCB values of all, which in this case they are all the same as none of them has been sampled, therefore randomly pick 2987 as the next individual.

Algorithm 6 N-Tuple Bandit EA *evolution* method

INPUT: Search space *space*, Maximum generation allowed *budget* **OUTPUT:** the best parameter set

1: $ntupleSystem \leftarrow$ setup the lookup tables from *space*
2: $individual \leftarrow$ random a set of parameters from *space*
3: $count \leftarrow 0$
4: **while** $count < budget$ **do**
5: $fitness \leftarrow evaluate(individual)$
6: $neighbours \leftarrow mutateAFew(individual)$
7: update $ntupleSystem$
8: $individual \leftarrow$ neighbours with the best UCB value
9: **end while**
10: **return** best parameter set in *ntupleSystem* based on defined criteria

- Plays a game with parameter set 2987, suppose the fitness value is 0.7, set
  - $table[2987][average] = 0.7$ and $table[2987][count] = 1$.
  - $table[2xxx][average] = \frac{0.3+0.7}{2} = 0.5$ and $table[2xxx][count] = 2$.
  - $table[x9xx][average] = 0.7$ and $table[x9xx][count] = 1$.
  - $table[xx8x][average] = \frac{0.3+0.7}{2} = 0.5$ and $table[xx8x][count] = 2$.
  - $table[xxx7][average] = \frac{0.3+0.7}{2} = 0.5$ and $table[xxx7][count] = 2$.
- Repeats the same mutation and evolution procedure, suppose that after a while the value 2487 was selected again and the fitness after this gameplay is 0.9, we set $table[2487][average] = \frac{0.3+0.9}{2} = 0.6$ and $table[2487][count] = 2$. Then set the rest of 1-Tuple value the same way as explained earlier.

After the evolution has finished, NTBEA gives sets of the best parameters based on each of four selection criteria: best sampled individual, a combination of best parameters, best sampled individual along with its neighbours and best UCB value individual. We have compared the results of these selection criteria of our experiments in 5.2.2. The NTBEA in [11] was implemented in Java, and is already compatible with GVGAI framework. Therefore we re-applied this version in our second experiment. Its implementation details is given in Algorithm 6.

In this chapter, I have given the details of GVGAI framework and all related aspects that were used in this dissertation, along with describing N-Tuple Bandit EA algorithm. Next, the game space and fitness calculation function are explained in chapter 4.

Chapter 4

Approaches

This dissertation gives a summary of two experiments that applying N-Tuple Bandit Evolutionary Algorithm (NTBEA - 3.3.3) to evolve game parameter set of the selected games. For the first game (Space Battle Evolved) we aimed to evolve games that best distinguish players skill-depth, while for the rest (GVGAI) we aim to evolve games that provide game environment for the player to obtain score in the similar, or ideally the same, trend with the pre-defined functions. In this chapter, approaches of preparing the experiment are explained in details. First, the selected game rules and space of Space Battle Evolved are described and followed by the game parameter fitness measurement function. Then the game rule and space of the selected GVGAI games are explained.Finally, the fitness evaluation procedures for GVGAI experiment are described.

4.1 Space Battle Evolved

4.1.1 Game Rules & Space

We selected 30 in-game parameters from Space Battle Evolved to evolve using NTBEA. These parameters can be categorized into 4 groups:

1. **Missile Related**

   There are 6 missile-related parameters. This includes maximum speed, cooldown (time allowed between each shot), type of missile (normal, twin

or bomb), size, maximum time allowed in the game and explosion radius of the bomb.

2. **Black Hole Related**

   There are 21 black hole-related parameters. 4 of these define black hole characteristics, such as size, how strong the dragging force is, penalty score for each time step a player stays inside and how big the non-penalty area is. The rest are for determining the number and position of each black hole. To do this, we first divide the map into n^2 regions where $n \in \{1, 2, 3, 4\}$. Then there to-be-evolved parameters that specify whether such region will have a black hole in the centre or not. This means a game can have up to 16 black holes. Figure 3.2 shows a game with $n = 2$ that have black holes in only region 1 and region 4.

3. **Resource Related**

   There are two resource (missile supply) related parameters: number of time step a resource is allowed on the screen and between each resource spawning.

4. **Enemy ID**

   This parameter determines which controller will be used as player 2 in the game. The controllers can be either *Do Nothing*, *Random*, *1LSA*, *RAS*, *MCTS* and *MEA*.

Table 4.1 describes all parameter names, descriptions, possible values and space size. Notice that even though the space size of *BLACKHOLE_CELL* is written as 2^{16}, we implemented the EAs to ignore other cell parameters that are unnecessary when the *GRID_SIZE* is less than 4.

4.1.2 Fitness Calculation

The fitness value of each game was evaluated with 3 gameplays, using three GVGAI controller 1SLA, RAS and MCTS as players. After a gameplay is finished,

TABLE 4.1: *Space Battle Evolved* evolvable parameters, description, their value ranges and step.

Parameter	Description	Value Range	Step	Space Size
MISSILE_MAX_SPEED	Missile's maximum speed	1 - 10	1	10
MISSILE_COOLDOWN	Missile's cooldown	1 - 9	1	10
MISSILE_TYPE	Missile's type	0 - 2	1	3
MISSILE_RADIUS	Missile's radius	2 - 10	2	5
MISSILE_MAX_TTL	Missile's maximum time to live	40-160	20	7
BOMB_RADIUS	Bomb's radius	10 - 50	10	5
BLACKHOLE_CELL x 16	Whether this cell has a black hole	0 or 1	1	2^{16}
BLACKHOLE_RADIUS	Black hole's radius	25 - 200	25	8
BLACKHOLE_FORCE	Black hole's drag force	0 - 3	1	4
BLACKHOLE_PENALTY	Score lost when in a black hole	0 - 9	1	10
GRID_SIZE	Square root of number of areas	1 - 4	1	4
SAFE_ZONE	Radius of area without black hole penalty	0 - 20	10	3
RESOURCE_TTL	Missile's pack time to live	400 - 600	100	3
RESOURCE_COOLDOWN	Missile's pack spawn cooldown	200 - 300	50	3
ENEMY_ID	Opponent controller ID	0 - 5	1	6
Total search space size		7.134×10^{14}		

player 1 and player 2 scores would be divided by 100 to lower the scale, then a 1000 bonus points would be given to the winner to prioritize winning result in producing the final score. The fitness value of this game is determined by the difference in the final score between player 1 and player 2. Equation 4.1 shows the final score calculation for each gameplay. $W_k = 1000$ if the player k won the game and 0 otherwise.

$$T_g = (\frac{S_1}{100} + W_1) - (\frac{S_2}{100} + W_2) \tag{4.1}$$

This total score calculation is done for three gameplays: player 1SLA vs RAS, RAS vs MCTS and 1SLA vs MCTS. After the total score T_g for every game g is computed, it was brought into the final fitness calculation using Equation 4.2, where T_1 is the amateur player's game fitness (1SLA), T_2 is the immediate player's game fitness (RAS) and T_3 is the skilful player's game fitness (MCTS).

$$Fitness = Min(T_3 - T_2, T_2 - T_1) \tag{4.2}$$

T_1, T_2 and T_3 can be derived from equation 4.1. For example, using one set of parameter in a game. If the final score of the three matches are as in Table 4.2, T_1

would be $(\frac{100}{100}) - (\frac{200}{100} + 1000)$ = -1001, T_2 would be $(\frac{400}{100} + 1000) - (\frac{200}{100})$ = 1002 and T_2 would be $(\frac{1000}{100} + 1000) - (\frac{100}{100})$ = 1009.

TABLE 4.2: An example of the score in $T_1, T2$ and T_3

	Player	Enemy
MCTS	1000	100
RAS	400	200
1SLA	100	200

Equation 4.2 is similar to that used for the Physical Travelling Salesman Problem by Perez et. al. [8]. Based on this fitness evaluation, the aim of the algorithms is to maximize the smallest gap between final scores of each game in the order $T_3 > T_2 > T_1$, which would result in the maximum skill-depth.

4.2 GVGAI Game Rules & Space

As discussed in details in 3.2.1 and 3.2.2, the VGDL game description files of all games must be modified first to support game parameter tuning. This involves changing the file class (on the first line) from *BasicGame* to *GameSpace*, and adding a *ParameterSet* section. Moreover, as our score trend functions (details in 4.2.4) are all increasing functions excepts the shifted sigmoids, but with different increasing rates in each area, we added a new rule to provide rich search space where the interesting solutions can be found.

4.2.1 Seaquest

In the original GVGAI Seaquest, all *holes* sprites (which are the NPC's spawn points) can spawn an NPC from the beginning of the game until the end. Since we wanted to vary the score increasing rate, we had to restrict this by adding new interaction sets that will allow the game to restrict these spawning to only some period of the game, and put this as an evolve-able parameter in the space. This caused a change in *SpriteSet* block and *InteractionSet* as depicted in Figure 4.1 and Figure 4.2. There were two main modifications highlighted in blue and

bold text fonts. The blue statements introduced four new sprite types into the game, all of which are the 'non-spawning' types of *holes*. In order, *nholeS* is a non-spawning type of *sharkhole*, *nholeW* is for *whalehole*, *nholeND* is for *normaldiverhole* and *nholeOD* corresponds with *oftendiverhole*. It is possible to group all these 'non-spawning' holes together using hierarchical feature of VGDL, but it would not facilitate the further usage in the *InteractionSet* because we still need to manually assign the statements line by line in all cases, as shown in Figure 4.2. The first four blue lines introduce the transformation of all *nholeS* to *sharkhole*, all *nholeW* to *whalehole*, all *nholeND* to *normaldiverhole* and all *nholeOD* to *oftendiverhole* when the time step is equal to DELAY, which will be set based on the given game parameter. The rest of blue lines do the transformation back into 'non-spawning' holes for all excepts *sharkhole* as the *shark* and *pirana* provide smaller magnitude of reward compared with the *whale* (set by the WHALESCORE parameter).

The bold texts were another main modification for the *ParameterSet* section, with each value will be replaced by the corresponding values with the same name from the *ParameterSet* section, as given in Figure 4.3. Each line of the *ParameterSet* block is in format *Name* > values=[*possible_values*] string=*description*, while [*possible_values*] can be either *minimum value*:*incremental value*:*maximum value* for numerical values, or *true*:*false* for boolean value. For example, the third line in Figure 4.3 (starts with SHPROB) means that the parameter name SHPROB has four possible values: 0.01, 0.06, 0.11 and 0.16 (because 0.16 + 0.05 = 0.21 > 0.2) and its description is SharkHole_SpawnProb, which is the spawn probability of *sharkhole*.

There were 18 parameters in total, which can be categorized into 6 groups based on their functionality in the game: probability related, health point related, speed related, crew related, non-spawn hole related and score related. Table 4.3 shows search space size of each parameter in Seaquest along with their

```
SpriteSet
    sky > Immovable img=oryx/backLBlue
    water > Immovable img=newset/water2
    saved > Immovable color=LIGHTGREEN
    nholeS > Passive color=RED
    nholeW > Passive color=BLUE
    nholeND > Passive color=BLACK
    nholeOD > Passive color=GREY
    holes > SpawnPoint color=LIGHTGRAY img=newset/whirlpool2
    portal=True
        sharkhole > stype=shark prob=SHPROB
        whalehole > stype=whale prob=WHPROB
        diverhole > stype=diver
            normaldiverhole > prob=DHPROB
            oftendiverhole > prob=OFDHPROB

    moving >
        avatar > ShootAvatar color=YELLOW stype=torpedo
        img=newset/submarine healthPoints=HP limitHealthPoints=MHP
        torpedo > Missile color=YELLOW img=oryx/bullet1
        fish >
            shark > Missile orientation=LEFT speed=SSPEED
            color=ORANGE img=newset/shark2
            pirana > Missile orientation=RIGHT speed=PSPEED
            color=RED shrinkfactor=0.6 img=newset/piranha2
            whale > Bomber orientation=RIGHT speed=WSPEED
            color=BROWN stype=pirana prob=WSPROB img=newset/whale
        diver > RandomNPC color=GREEN speed=DSPEED img=newset/diver1
        cons=DCONS

    crew > Resource color=GREEN limit=CRLIMIT
```

FIGURE 4.1: Modified Seaquest *SpriteSet* game description

description. The total space is considerably huge with about 58 billion combinations. It is obvious that brute force is infeasible to search for the fittest set.

4.2.2 Waves

Similar with Seaquest, Waves *portal*s start spawning *rock*s and *alien*s since the beginning of the game. We modified this by adding a new sprite type called *closePortal* into the game. These *closePortal*s will transform into *portalSlow*s after a delayed time, and transform back before the end of the game. These modifications can be seen in Figure 4.4 and Figure 4.5 for *SpriteSet* and *InteractionSet*

```
InteractionSet

    avatar TIME > subtractHealthPoints timer=TIMERHPLOSS
    repeating=True
    avatar TIME > transformToAll stype=nholeS stypeTo=sharkhole
    nextExecution=DELAY timer=DELAY repeating=False
    avatar TIME > transformToAll stype=nholeW stypeTo=whalehole
    nextExecution=DELAY timer=DELAY repeating=False
    avatar TIME > transformToAll stype=nholeND
    stypeTo=normaldiverhole nextExecution=DELAY timer=DELAY
    repeating=False
    avatar TIME > transformToAll stype=nholeOD stypeTo=oftendiverhole
    nextExecution=DELAY timer=DELAY repeating=False
    avatar TIME > transformToAll stype=whalehole stypeTo=nholeW
    nextExecution=SHUTHOLE timer=SHUTHOLE repeating=False
    avatar TIME > transformToAll stype=normaldiverhole
    stypeTo=nholeND nextExecution=SHUTHOLE timer=SHUTHOLE
    repeating=False
    avatar TIME > transformToAll stype=oftendiverhole stypeTo=nholeOD
    nextExecution=SHUTHOLE timer=SHUTHOLE repeating=False

    EOS avatar diver sky > stepBack
    fish EOS > killSprite
    whale EOS > killSprite
    fish torpedo > killBoth scoreChange=1
    avatar fish > killSprite
    whale torpedo > killBoth scoreChange=WHALESCORE
    avatar whale > killSprite
    avatar sky > addHealthPoints value=HPPLUS
    avatar sky > spawnIfHasMore resource=crew stype=saved
    limit=CRLIMIT spend=CRLIMIT
    saved sky > killSprite scoreChange=10
    avatar diver > changeResource resource=crew
    diver avatar > killSprite
```

FIGURE 4.2: Modified Seaquest *InstructionSet* game description

respectively. The lines highlighted in blue in the *InteractionSet* effects the transformation between *closePortal* and *portalSlow*, which will happen at step DELAY and CLOSE, set by the game parameter set.

In the initial Waves description, the *avatar* has a health point which will allow it to be hit by the enemies multiple times before the game is over. It is still true in this modified game unless the SLIMIT is set to lower than |APEN| or |LASERPEN|. This makes the game can be either one-shot killed or multiple-shot killed depending on the parameter setup.

```
ParameterSet

  #Name of the parameter > values(min, inc, max)/(boolean) descriptive
  string

   SHPROB       >  values=0.01:0.05:0.2      string=SharkHole_SpawnProb
   WHPROB       >  values=0.005:0.02:0.1     string=WhaleHole_SpawnProb
   DHPROB       >  values=0.005:0.01:0.045   string=DiverHole_SpawnProb
   OFDHPROB     >  values=0.05:0.02:0.1      string=DiverHole_Often_SpawnProb
   HP           >  values=9:8:40             string=Initial_Health_Points
   MHP          >  values=10:10:40           string=Max_Health_Points
   HPPLUS       >  values=1:1:4              string=Health_Points_Plus
   SSPEED       >  values=0.05:0.15:0.5      string=Shark_Speed
   WSPEED       >  values=0.05:0.15:0.3      string=Whale_Speed
   PSPEED       >  values=0.05:0.15:0.5      string=Pirana_Speed
   DSPEED       >  values=0.1:0.2:1.0        string=Diver_Speed
   WSPROB       >  values=0.01:0.03:0.1      string=Whale_Spawn_Prob
   DCONS        >  values=1:1:3              string=Consecutive_Moves_Diver
   CRLIMIT      >  values=1:2:8              string=Crew_Limit
   TIMERHPLOSS  >  values=5:5:20             string=Is_GameWon_OnTimeOut
   DELAY        >  values=0:50:200           string=Hole_Delay
   SHUTHOLE     >  values=200:50:400         string=Hole_Close
   WHALESCORE   >  values=5:5:20             string=WHALE_SCORE
```

FIGURE 4.3: Seaquest *ParameterSet* game description

TABLE 4.3: Seaquest's parameter set search space

Type	Name	Description	Size
Speed	SSPEED	Shark's speed	4
	WSPEED	Whale's speed	2
	PSPEED	Piranha's speed	4
	DSPEED	Diver's speed	5
Probability	SHPROB	Shark portal's probability to spawn a shark	4
	WHPROB	Whale portal's probability to spawn a whale	5
	DHPROB	Normal diver portal's probability to spawn a diver	5
	OFDHPROB	Fast diver portal's probability to spawn a diver	3
	WSPROB	Whale's probability to spawn a piranha	4
Health Point	HP	Avatar's initial oxygen amount	4
	MHP	Avatar's maximum oxygen amount	4
	HPPLUS	Avatar's oxygen gained per time step at the sea surface	4
	TIMERHPLOSS	Oxygen amount lost per time step underwater	4
Score	WHALESCORE	Score increased when a whale is shot	4
Diver	DCONS	Number of consecutive tiles that a diver can move per step	3
	CRLIMIT	Maximum diver that the avatar can rescue in one diving	4
Portal	DELAY	The time step that all portals start spawning	5
	SHUTHOLE	The time step that all portals stop spawning	5
Total search space size		5.892×10^{10}	

All evolve-able parameters in Waves are depicted in Figure 4.6. There are 17 parameters in total, which fall into each of the 6 categories: speed-related, cooldown-related, probability-related, score-related, shield-related and portal-related. The parameter search space and description of Waves is given in Table

```
SpriteSet

    background > Immovable img=oryx/space1 hidden=True
    asteroid > Immovable img=oryx/planet

    missile > Missile
        rock > orientation=LEFT speed=RSPEED color=BLUE
        img=oryx/orb3
        sam > orientation=RIGHT color=BLUE speed=SSPEED
        img=oryx/orb1 shrinkfactor=0.5
        laser > orientation=LEFT speed=LSPEED color=RED
        shrinkfactor=0.75 img=newset/laser2_1

    portal >
        portalSlow > SpawnPoint stype=alien cooldown=ACOOLDOWN
        prob=APROB img=newset/whirlpool2 portal=True
        rockPortal > SpawnPoint stype=rock cooldown=RCOOLDOWN
        prob=RPROB img=newset/whirlpool1 portal=True
    closePortal > Passive color=BLACK
    shield > Resource color=GOLD limit=SLIMIT img=oryx/shield2
    avatar > ShootAvatar color=YELLOW stype=sam speed=PSPEED
    img=oryx/spaceship1 rotateInPlace=False
    alien > Bomber color=BROWN img=oryx/alien3 speed=ASPEED
    orientation=LEFT stype=laser prob=ASPROB
```

FIGURE 4.4: Modified Waves *SpriteSet* game description

TABLE 4.4: Waves' parameter set search space

Type	Name	Description	Size
Speed	RSPEED	Rock's speed	5
	SSPEED	Avatar missile's speed	4
	LSPEED	Laser's speed	5
	PSPEED	Avatar's speed	3
	ASPEED	Alien's speed	6
Cooldown	ACOOLDOWN	Alien portal's cooldown	4
	RCOOLDOWN	Rock portal's cooldown	4
Probability	APROB	Alien portal's probability to spawn an alien	2
	RPROB	Rock portal's probability to spawn a rock	6
	ASPROB	Alien's probability to shoot a laser	4
Avatar	SLIMIT	Avatar's maximum health point	5
	SPLUS	Amount of avatar's health point increased when collected a shield	5
Score	APEN	Score lost when the avatar collides with an alien	4
	LASERPEN	Score lost when the avatar was hit by a laser	4
	SREWARD	Score gained when an alien is shot	5
Portal	DELAY	The time step that all portals start spawning	7
	CLOSE	The time step that all portals stop spawning	4
Total search space size		7.741×10^{10}	

4.4. The total space size is in the same order (10^{10}) as Seaquest, with 77 billion possible sets.

```
InteractionSet
    avatar EOS > stepBack
    alien EOS > killSprite
    missile EOS > killSprite

    avatar TIME > transformToAll stype=closePortal stypeTo=portalSlow
    nextExecution=DELAY timer=DELAY repeating=False
    avatar TIME > transformToAll stype=portalSlow stypeTo=closePortal
    nextExecution=CLOSE timer=CLOSE repeating=False

    alien sam > killBoth scoreChange=SREWARD
    sam laser > transformTo stype=shield killSecond=True
    avatar shield > changeResource resource=shield value=SPLUS
    killResource=True

    avatar rock > killIfHasLess resource=shield limit=0
    avatar rock > changeResource resource=shield value=-1
    killResource=True

    avatar alien > killIfHasLess resource=shield limit=0
    avatar alien > changeResource resource=shield value=APEN
    killResource=True

    avatar laser > killIfHasLess resource=shield limit=0
    avatar laser > changeResource resource=shield value=LASERPEN
    killResource=True

    asteroid sam laser > killSprite
    rock asteroid > killSprite
    alien asteroid > killSprite
    laser asteroid > killSprite
    avatar asteroid > stepBack
```

FIGURE 4.5: Modified Waves *InteractionSet* game description

4.2.3 Defender

Similar with Seaquest and Waves, we added two new sprites and four new interactions to restrict *alien* spawning period. These are shown in Figure 4.7 and Figure 4.8 for *SpriteSet* and *TerminationSet* respectively. Two new included sprites were *closeSlow*, which is a passive version of *portalSlow*, and *closeFast* that is for *portalFast*. All *closeSlow* will transform into *portalSlow* when the game time step is equal to DELAY, same with all *closeFast* that will become *portalFast*. The *portal* will then transform back to the passive ones at time step CLOSE.

```
ParameterSet

  #Name of the parameter > values(min, inc, max)/(boolean) descriptive
  string

   RSPEED      >  values=0.45:0.5:2.5      string=Rock_Speed
   SSPEED      >  values=0.5:0.5:2.0       string=Shot_Speed
   LSPEED      >  values=0.1:0.1:0.5       string=Laser_Speed
   ACOOLDOWN   >  values=2:4:14            string=Alien_Cooldown
   RCOOLDOWN   >  values=2:4:14            string=Rock_Cooldown
   APROB       >  values=0.01:0.04:0.07    string=Alien_PortalProb
   RPROB       >  values=0.15:0.05:0.4     string=Rock_SpawnProb
   PSPEED      >  values=0.5:0.5:1.5       string=Avatar_Speed
   ASPEED      >  values=0.05:0.05:0.3     string=Alien_Speed
   SLIMIT      >  values=2:2:10            string=Shield_Limit
   ASPROB      >  values=0.005:0.005:0.02  string=Alien_SpawnProb
   SPLUS       >  values=1:1:5             string=Shield_Plus
   APEN        >  values=-4:1:-1           string=Alien_Collide_Penalty
   LASERPEN    >  values=-4:1:-1           string=Laser_Collide_Penalty
   SREWARD     >  values=1:2:9             string=Alien_Shot_Reward
   DELAY       >  values=0:50:300          string=Open_Portal
   CLOSE       >  values=350:50:500        string=Close_Portal
```

FIGURE 4.6: Waves *ParameterSet* game description

The parameter set description of Defender is given in Figure 4.9. In contrast to Seaquest and Waves, Defender score can be negative as LOSSCITY value will be subtracted from the total score if a *city* is bombarded by a *bomb*. Since the *avatar* cannot be killed (both *alien*s and *bomb*s are harmless to the *avatar*), and there are 68 *city*s in the first level game (which is the only level tested in the experiment, and this is kept fixed) the score can be as negative as 68×-4 (minimum value of LOSSCITY) when the game ends. It is obviously more challenging for the EA to evolve a Defender parameter set that satisfies our target score functions. There are 5 parameter groups: speed-related, probability-related, cooldown-related, score-related and portal-related. Defender parameter space size and description is listed in Table 4.5. The total search space size, which is lower than Waves and Seaquest but still in the same 10^{10} order, is around 10 billions.

```
SpriteSet
    floor > Immovable img=oryx/space1 hidden=True
    city > Immovable color=WHITE img=newset/city1
    avatar > ShootAvatar color=YELLOW ammo=bullet stype=sam
    img=oryx/spaceship1 rotateInPlace=False
    missile > Missile
        sam > orientation=RIGHT color=BLUE img=oryx/bullet1
        bomb > orientation=DOWN color=RED speed=BSPEED
        img=newset/bomb shrinkfactor=0.6
    alien > Bomber orientation=LEFT stype=bomb prob=APROB
    cooldown=ACOOLDOWN speed=ASPEED img=oryx/alien3

    portal > SpawnPoint stype=alien cooldown=PCOOLDOWN invisible=True
    hidden=True
        portalSlow > prob=SLOWPPROB
        portalFast > prob=FASTPPROB

    closeSlow > Passive color=BLACK
    closeFast > Passive color=BLACK
    portalAmmo > SpawnPoint stype=supply cooldown=AMCOOLDOWN
    prob=AMPROB invisible=True

    supply > Missile orientation=DOWN speed=SUPSPEED
    img=oryx/goldsack
    bullet > Resource limit=BLIMIT
    wall > Immovable img=oryx/wall1
```

FIGURE 4.7: Modified Defender *SpriteSet* game description

TABLE 4.5: Defender's parameter set search space

Type	Name	Description	Size
Speed	BSPEED	Bomb speed	5
	ASPEED	Alien speed	5
	SUPSPEED	Supply falling speed	3
Probability	APROB	Alien's probability to shoot a bomb	5
	SLOWPPROB	Slow portal's probability to spawn an alien	5
	FASTPPROB	Fast portal's probability to spawn an alien	2
	AMPROB	Supply portal's probability to spawn a supply	3
Cooldown	ACOOLDOWN	Alien's bomb shooting cooldown	5
	PCOOLDOWN	Alien portal's cooldown	4
	AMCOOLDOWN	Supply portal's cooldown	4
Supply	BLIMIT	Avatar's maximum ammo supply	4
	ADDSUP	Amount of ammo a supply pack contains	5
Score	LOSSCITY	Score lost when a city is destroyed	4
	AREWARD	Score gained when an alien is shot	5
Portal	DELAY	The time step that all portals start spawning	7
	CLOSE	The time step that all portals stop spawning	4
Total search space size		1.08×10^{10}	

```
InteractionSet
    avatar TIME > transformToAll stype=closeSlow stypeTo=portalSlow
    nextExecution=DELAY timer=DELAY repeating=False

    avatar TIME > transformToAll stype=closeFast stypeTo=portalFast
    nextExecution=DELAY timer=DELAY repeating=False

    avatar TIME > transformToAll stype=portalSlow stypeTo=closeSlow
    nextExecution=CLOSE timer=CLOSE repeating=False

    avatar TIME > transformToAll stype=portalFast stypeTo=closeFast
    nextExecution=CLOSE timer=CLOSE repeating=False

    avatar EOS city wall > stepBack

    alien EOS > killSprite
    missile EOS city > killSprite

    city bomb > killSprite scoreChange=LOSSCITY
    city sam alien > killSprite

    supply alien > killSprite
    alien sam > killBoth scoreChange=AREWARD

    supply supply > stepBack
    supply wall city > stepBack pixelPerfect=True
    avatar supply > changeResource resource=bullet value=ADDSUP
    killResource=True
```

FIGURE 4.8: Modified Defender *InteractionSet* game description

4.2.4 Fitness Calculation

To evaluate the fitness value of a parameter set, it would be first injected into the game to set all relevant parameters, and then the game with this setup would be played by a selected controller. During the gameplay, the raw score of every game steps would be recorded. This is then further used in fitness calculation based by comparing each with the target function of that run. All target score functions in the experiments are described next.

```
ParameterSet

  #Name of the parameter > values(min, inc, max)/(boolean) descriptive
  string

   BSPEED       >  values=0.1:0.2:0.9       string=Bomb_Speed
   APROB        >  values=0.01:0.01:0.05    string=Alien_SpawnProb
   ACOOLDOWN    >  values=2:2:10            string=Alien_Cooldown
   ASPEED       >  values=0.2:0.2:1.0       string=Alien_Speed
   PCOOLDOWN    >  values=5:5:20            string=Portal_Cooldown
   SLOWPPROB    >  values=0.05:0.05:0.25    string=SlowPortal_Prob
   FASTPPROB    >  values=0.3:0.2:0.7       string=FastPortal_Prob
   AMCOOLDOWN   >  values=5:5:20            string=Ammo_Cooldown
   AMPROB       >  values=0.05:0.1:0.25     string=Ammo_Prob
   SUPSPEED     >  values=0.05:0.2:0.55     string=Supply_Speed
   BLIMIT       >  values=5:5:20            string=Resource_Limit
   ADDSUP       >  values=1:1:5             string=Resource_Add
   LOSSCITY     >  values=-4:1:-1           string=Lost_City_Penalty
   AREWARD      >  values=1:2:9             string=Alien_Shot_Reward
   DELAY        >  values=0:50:300          string=Open_Portal
   CLOSE        >  values=350:50:500        string=Close_Portal
```

FIGURE 4.9: Defender *ParameterSet* game description

Target Score Functions

All of the selected target functions are positive-definite functions[1] (e.g. $f(x) > 0$ for all $x > 0$). The simplest ones are linear functions, followed by a more complex piecewise linear function, and other 3 non-linear functions. The first non-linear function is shifted sigmoid, which are to-the-left and right shifted versions of the original sigmoid function. Next, a logarithm and exponential functions were included to see if the EA can react differently with the fast increasing rate in different areas of the target functions.

- **Linear**

  A linear function is a continuous function in format $y = cx$, we picked the values c to be either $0.2, 0.4$ or 1. Theoretically, our domain set (time step) is discrete, therefore it is not actually a continuous linear function but a piecewise function in the format shown in equation 4.3, while I^+ is the set of positive integers. For simplicity, we will call the functions with this format 'linear' from this point onwards. The similar annotation

[1] https://www.encyclopediaofmath.org/index.php/Positive-definite_function

is also applied for all target functions. Figure 4.10 shows the three linear functions used in the experiment.

$$f(x) = \begin{cases} cx, & \text{if } x \in I^+ \\ undefined, & \text{otherwise} \end{cases} \tag{4.3}$$

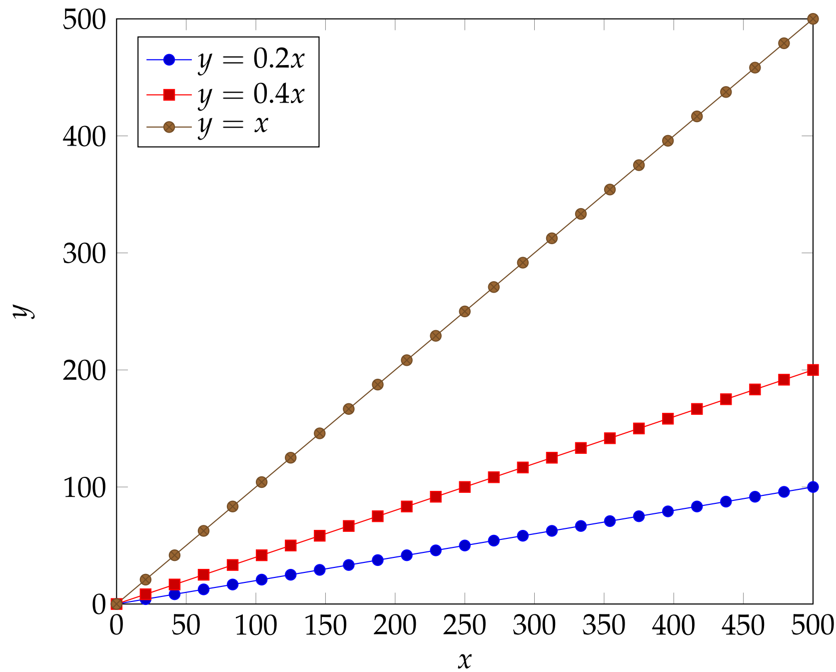


FIGURE 4.10: Linear target functions

As described in 4.2.1, the highest obtainable per step score is 20 for Seaquest, with WHALESCORE is set as 20 and the *avatar* has managed to shoot it in every time step. This requires the parameter WHPROB to be as high as possible. However, the controller must able to avoid colliding with these pool of *whale*s spawn while maintaining the oxygen level. Based on this, it seems possible to find parameter sets that satisfies the $y = x$ linear target function, as scoring 1 per time step in average and trying to survive until the game ends is not challenging in the game with low speed average amount of *fish*. Waves, however, is slightly more difficult with the maximum value of AREWARD is only 9. Although in most settings the

avatar would not lose its life after only a single hit, it is still challenging to avoid all *alien*s and *laser*s and *rock*s while shooting them to maximize the score. Defender is the toughest in this aspect as the score can be negative. This can however be alleviated with LOSSCITY = -1, AREWARD = 9 and considerably low BSPEED and ASPEED, with high ACOOLDOWN. We expected the EA to find good sets of parameters as we believe they are such sets in the search space.

- **Linear Piecewise**

  Piecewise functions refer to all functions that are combined from more than one functions, generally cannot expressed by one equation and need an if-else notation for different intervals. Linear piecewise functions are the piecewise function that all sub-functions are linear. In our second experiment, we defined two linear piecewise target functions, given in equation 4.4 and 4.5. This is clearly plotted in Figure 4.11 and can be seen that the blue graph (represents equation 4.4 is increasing fast in the first 100 timestep, while the red graph (represents equation 4.5) increases slowly until time step 400 and then rapidly afterwards. The objective of defining these two functions is to test if the EA can react with such swift change in increasing rate, in both ways.

$$f(x) = \begin{cases} 2x, & \text{if } x \leq 100 \\ 2(100) + 0.2(x - 100), & \text{otherwise} \end{cases} \tag{4.4}$$

$$f(x) = \begin{cases} 0.2x, & \text{if } x \leq 400 \\ 0.2(400) + 2(x - 400), & \text{otherwise} \end{cases} \tag{4.5}$$

- **Shifted Sigmoid**

  Sigmoid function is an S-shape curve that the simplest format has the y-value between 0 and 1, with $f(0) = 0.5$. The normal sigmoid function is

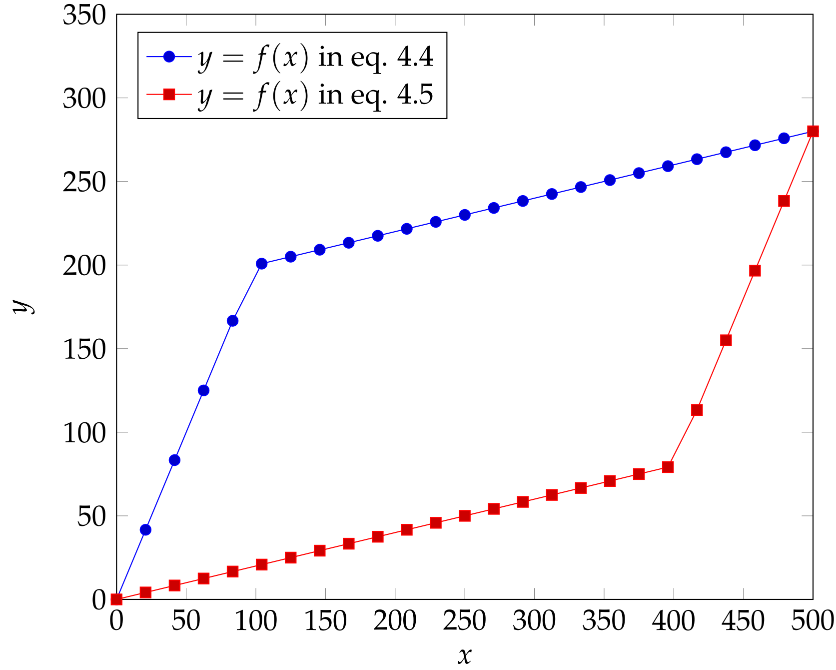


FIGURE 4.11: Linear piecewise target functions

as equation 4.6 and the actual graph is plotted in Figure 4.12.

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{4.6}$$

We modified the original function to be in format:

$$f(x) = K_1\left(\frac{1}{1 + \exp(-\frac{x}{K_2} + K_3)}\right) \tag{4.7}$$

while K_1, K_2, K_3 are constants for shifting the value range. K_1, fixed at 150, is to scale up y to $[0, 150]$, K_2, set at 30, is to restrict the value to be less than 160, and K_3 is for shifting the values along the x-axis (left and right). We set $K_3 = 3$ for a function that will be called *left-sigmoid* from this point, and $K_3 = 12$ for *right-sigmoid*. These left and right shifted sigmoid functions can be visualized in Figure 4.13.

Similar with linear piecewise target functions, the objective of setting these shifted sigmoid functions is to test if the EA can react with the rapid change

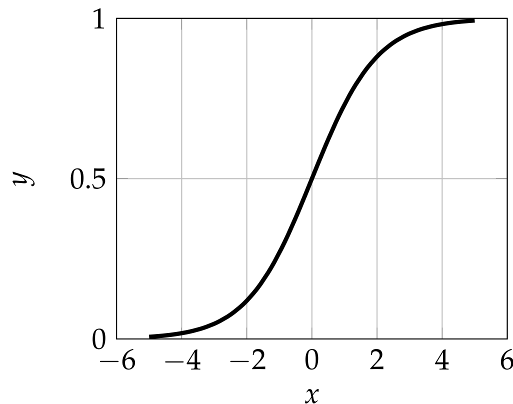


FIGURE 4.12: Sigmoid function

in score increasing rate at different points. Another minor objective is to explore its behaviour when the score is staying constant for a period. These happen after around step 250 to the end of the game for *left-sigmoid* and at the beginning of the game until almost step 200 for *right-sigmoid*.

- **Logarithm and Exponential**

  The last pair of our target functions were a base-2 logarithm and an exponential functions given in equations 4.8 and 4.9 respectively.

$$f(x) = 15 \log_2 x \tag{4.8}$$

$$f(x) = 2^{\frac{x}{70}} \tag{4.9}$$

The function values for $x \in [0, 500]$ are plotted in Figure 4.14. Notice that generally logarithm function does not allow $x \in (\infty, 0]$, but we set $f(0) = 0$ in this graph. It is commonly known that exponential function is among the fastest growing functions (highest order of growth in general) in mathematics. This is the main reason that we have to restrict the maximum time step of most gameplays to 500 in the experiment, because $2^{\frac{500}{70}} \approx 141$ while $2^{\frac{1000}{70}} \approx 19972$ which is clearly unachievable for the current settings.

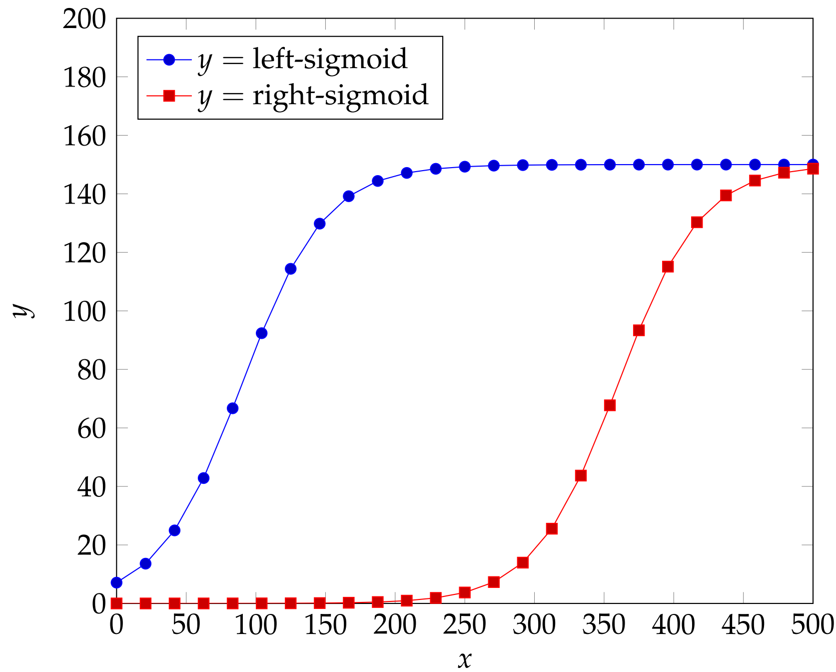


FIGURE 4.13: Shifted sigmoid target functions

> Similar with the piecewise and sigmoid comparison, this pair of logarithm and exponential functions also differs in 'high increasing rate' area, with the logarithm function increases considerably fast in the beginning and slower with higher x, and the exponential function is in the complete opposite style.

Apart from comparing the $f(x)$ values of the functions themselves, their 'increasing rate' (also known as slope, differential) should also be taken into account. As it is possible that the EA might unable to find the parameter sets that give the exact value, but able to achieve the similar score increasing rate. Differential functions of all target score functions are shown in Figure 4.15. It can be seen that the increasing rates of the function in the same groups (or pairs) are alike, while they are all dissimilar across the groups. Linear functions have constant slope functions, which are the value of c in $y = cx$. Linear piecewise function each has two distinct constant slope functions $y = 0.2$ or $y = 2$ according to their definitions. Both shifted sigmoid functions have smooth continuous

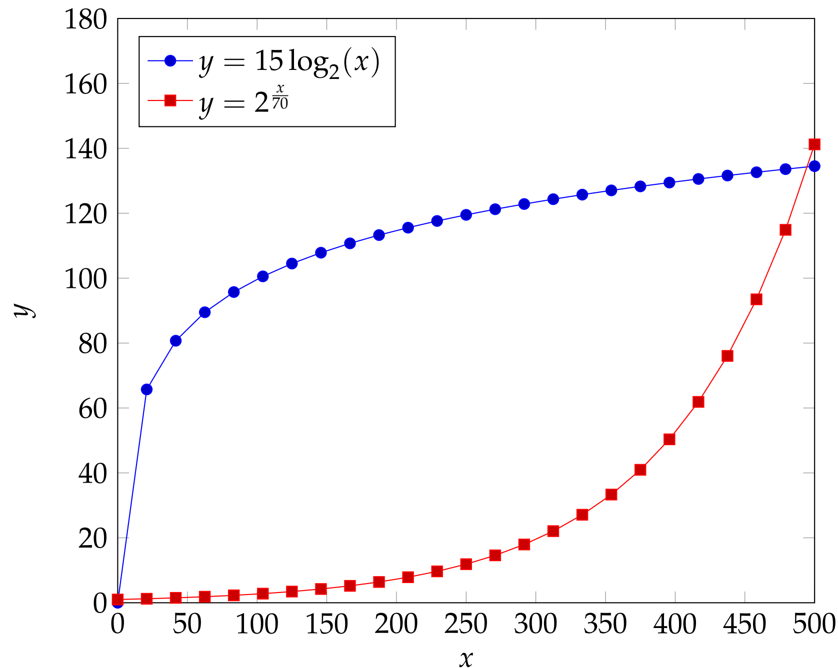


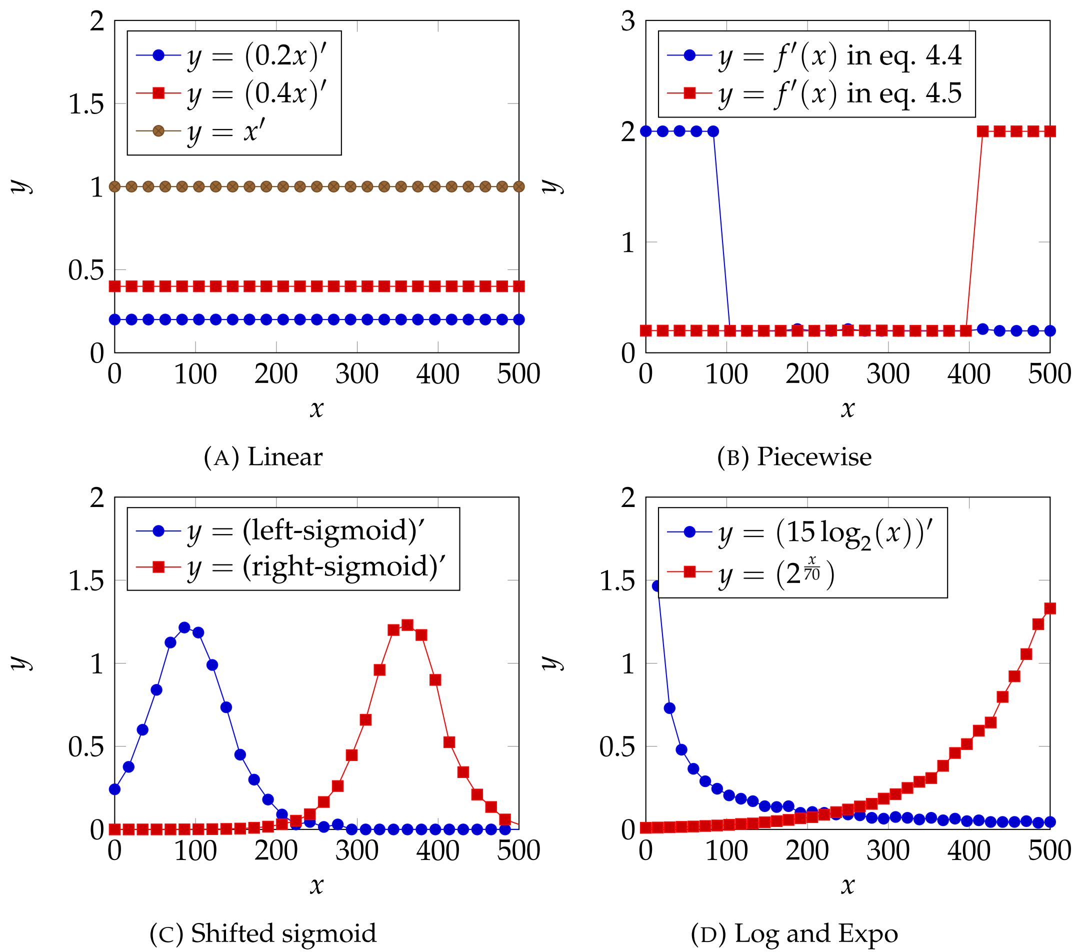
FIGURE 4.14: Logarithm and exponential target functions

bell-shaped curves, which the centre point locates near the point $x = 100$ for *left-sigmoid* and $x = 350$ for *right-sigmoid*. Finally, the logarithm function has a first-quadrant rectangular hyperbola ($xy = 1$) similar-shaped function as its slope, while the exponential function slope is also an exponential function with one order lower (for example, 2^{n-1} is the slope for 2^n).

After the game is finished, the recorded score value of every time step would be compared with the target function of that experiment, then the fitness value is calculated. These procedures are explained in details next.

Loss Calculation & Fitness Value

We used a *Root Mean Square Error (RMSE)* to calculate the total deviation between the obtained score and the target function. Suppose that $\hat{s}$ is a set of real score sequence from time step 1 to the last, and $\hat{y}$ is the set of $f(x)$ values for the selected target function with domain set = $1, 2, ..., n$ where n is the last time step, the RMSE value between $\hat{s}$ and $\hat{y}$ can be computed as follows:



(A) Linear

(B) Piecewise

(C) Shifted sigmoid

(D) Log and Expo

FIGURE 4.15: All target function slope

$$RMSE(\hat{s}, \hat{y}) = \sqrt{\frac{\sum_{i=1}^{n} (\hat{y}_i - \hat{s}_i)^2}{n}} \quad (4.10)$$

This value, however, can have a huge magnitude from a game that $\hat{s}$ and $\hat{y}$ are highly different. We normalized the value by dividing it by $norm = \hat{y}_{max} - \hat{y}_{min}$ to scale down the absolute value of $RMSE(\hat{s}, \hat{y})$. There were two versions of Normalized-RMSE used in our experiment, the normal one (NRMSE) and the biased one (B-NRMSE).

- **Normal Normalized RMSE (NRMSE)**
  The NRMSE between a score set $\hat{s}$ and the selected target set $\hat{y}$ can be calculated by:

$$NRMSE(\hat{s}, \hat{y}) = \frac{RMSE(\hat{s}, \hat{y})}{\hat{y}_{max} - \hat{y}_{min}} \quad (4.11)$$

  where RMSE is depicted in equation 4.10.

- **Biased-Normalized RMSE (B-NRMSE)**
  In this biased version of NRMSE, we penalized the values more in the area that has higher increasing rate. Our initial objective behind this was to trigger the EA to react faster to such areas, expecting that it would help speeding up the target curves fitting. This was done by calculating the difference between each adjacent pair in $\hat{y}$, and called it $sl\hat{o}pe(\hat{y})$:

$$sl\hat{o}pe(\hat{y}) = \{d_i | d_i = \hat{y}_{i+1} - \hat{y}_i\} \qquad \text{for } i = 1, .., |\hat{y}| - 1 \quad (4.12)$$

  Furthermore, we introduced a value called $slopeScale = \frac{1}{max(sl\hat{o}pe(\hat{y}))}$ to normalize these slope values into the ratio between $[0, 1]$ before multiplying them with the square error value calculated. That is, B-NRMSE can be computed as shown in this equation:

$$B\text{-}NRMSE(\hat{s}, \hat{y}) = \sqrt{\frac{\sum_{i=1}^{n}((\hat{y}_i - \hat{s}_i)^2 \times |slopeScale \times \hat{slope}(\hat{y})_i|)}{n}} \tag{4.13}$$

After either NRMSE or B-NRMSE was used to calculate the loss value between $\hat{s}$ and $\hat{y}$, called $Loss(\hat{s}, \hat{y})$, the final fitness value is equal to $1 - Loss(\hat{s}, \hat{y})$. Since the loss value is always at least 0, we know that the maximum possible value of this fitness calculation is 1 and the EA should try to optimize this by maximizing it, which satisfies the current implementation of NTBEA in GVGAI framework.

Fitness Calculation Summary

To explicate the fitness calculation steps employed during the evolution step in the experiment, they are summarized it details as follows:

1. Selects a game *game* to parameterize.
2. Selects a target function to fit, suppose it is called $\hat{t}$.
3. Selects a version of RMSE to be used (either normal or biased one), calls it *Loss*.
4. Uses the EA to select the next parameter set, calls it $\hat{p}$.
5. Constructs a version of *game* using $\hat{p}$.
6. Injects RHEA (see 3.2.3 for details) to play this *game*.
7. Records the score obtained by RHEA for every time step, stored in $\hat{s}$.
8. Calculates $Loss(\hat{t}, \hat{s})$ by:
    (a) Using equation 4.11 if *Loss* is NRMSE.
    (b) Using equation 4.13 if *Loss* is B-NRMSE.

9. Sets fitness of $\hat{p}$ to 1-$Loss(\hat{t}, \hat{s})$.

10. Continues evolution.

Although our possible values of *game* are restricted to only three games and the $\hat{t}$ can be one of the 9 functions described earlier, this approach can be extended to any GVGAI games and any positive-increasing functions.

In this chapter, we clarified in details about all modifications made to the game rules, as well as describing the overall search space. All selected target functions of the second experiment were also presented and illustrated for visualization. Finally the fitness calculation for both experiments were explained deliberately. Next section we present the report of the experiments carried out using these set-ups.

Chapter 5

Experiments & Results

In chapter 1, we formed three main hypotheses as follows:

- **HM1:** NTBEA is more robust to noise in noisy environments than standard hill-climbing evolutionary algorithms.
- **HM2:** NTBEA can be applied to tune game parameters to provide specific pre-defined player score trend, for any players playing the game.
- **HM3:** General Video Game Playing controllers can be used as substitutions for human players in automatic game parameterization.

There were two experiments taken to validate these hypotheses. Both experiments applied NTBEA with general video game agents as representatives of human players to do automatic game parameterization. The first experiment was done for Space Battle Evolved using the game space defined in 4.1 to validate hypothesis HM1 and HM3. The second experiments are for three GVGAI selected games, described in 4.2 for validating hypotheses HM2 and HM3. This chapter includes results from both experiments, starting with Space Battle Evolved, and then GVGAI games.

5.1 Space Battle Evolved

We apply the RMHC, the Biased Mutation RMHC (denoted as B-RMHC) and the N-Tuple Bandit Evolutionary Algorithm (denoted as NTBEA) independently 50

times to evolve game instances, thus 150 games are designed in total. There was one minor hypothesis formed specifically to this experiment:

- **HS1:** The games evolved by NTBEA satisfy human preferences more compared to both RMHC-based algorithms.

Therefore, in this experiment we aimed to validate HM1, HM3 and HS1 hypotheses. 100 game evaluations are allocated to each of the algorithms during the evolution. Each evaluation takes into account the outcomes of three games played by the 1SLA (amateur), RAS (intermediate) and MCTS (skilful) controllers. The sorted average fitness values over 100 evaluations and standard errors are presented in Figure 5.1.

The NTBEA (green markers, denoted as N-Tuple) outperforms both the RMHC and its variant. Moreover, NTBEA is more robust and has a more stable performance (negligible standard error). Among 50 game instances evolved by the NTBEA, only a few of them (very left part in Figure 5.1) have an average fitness below zero. Nevertheless, the lowest average fitness is still much higher than most of the games evolved by both RMHC and B-RMHC. This confirms that the hypothesis HM1 is correct, at least for this Space Battle Evolved game space for this specific optimization task.

Table 5.1 provides the parameters of the games, optimized by the RMHC, the B-RMHC and the NTBEA algorithm, with the highest and lowest average fitness. As seen, NTBEA was more robust than the baseline RMHC in evolving parameters in a simple noisy game environment.

We picked up the games with the highest and lowest average fitness designed by the 3 algorithms and invited two human players to evaluate them. The human players were asked to play this 6 games and provide feedback without being told the fitness level of each game. One screenshot of each of the games is presented in Figure 5.2. These two human players evaluated the games differently, according to their playing preference. Player A cared more about the

TABLE 5.1: Optimized parameters of game instances with the highest or lowest average fitness, designed by three algorithms.

Parameter	Value optimised by different algorithms					
	RMHC		B-RMHC		N-Tuple	
	High	Low	High	Low	High	Low
MISSILE_MAX_SPEED	6	1	10	1	9	10
MISSILE_COOLDOWN	9	5	5	3	2	5
MISSILE_RADIUS	2	10	10	4	4	4
MISSILE_MAX_TTL	140	60	40	80	40	140
GRID_SIZE	4	3	1	1	3	1
BLACKHOLE_CELL(1,1)	0	1	0	1	1	1
BLACKHOLE_CELL(1,2)	0	0	1	1	0	1
BLACKHOLE_CELL(1,3)	0	1	1	0	0	1
BLACKHOLE_CELL(1,4)	1	0	0	0	0	0
BLACKHOLE_CELL(2,1)	0	0	1	1	0	1
BLACKHOLE_CELL(2,2)	1	1	0	1	0	0
BLACKHOLE_CELL(2,3)	1	0	1	0	0	1
BLACKHOLE_CELL(2,4)	1	1	1	0	1	0
BLACKHOLE_CELL(3,1)	1	1	1	0	0	0
BLACKHOLE_CELL(3,2)	1	0	1	0	1	1
BLACKHOLE_CELL(3,3)	1	0	0	0	1	1
BLACKHOLE_CELL(3,4)	1	1	0	0	1	1
BLACKHOLE_CELL(4,1)	1	0	0	1	0	0
BLACKHOLE_CELL(4,1)	1	0	0	0	0	1
BLACKHOLE_CELL(4,3)	1	0	0	1	0	1
BLACKHOLE_CELL(4,4)	0	1	0	0	1	1
BLACKHOLE_RADIUS	200	75	100	100	150	25
BLACKHOLE_FORCE	2	1	3	3	3	1
BLACKHOLE_PENALTY	3	4	0	7	7	8
SAFE_ZONE	20	0	20	20	10	10
BOMB_RADIUS	10	50	20	40	20	20
MISSILE_TYPE	2	1	2	0	2	0
RESOURCE_TTL	400	500	500	500	400	500
RESOURCE_COOLDOWN	200	250	250	200	200	200
ENEMY_ID	0	2	1	0	0	5

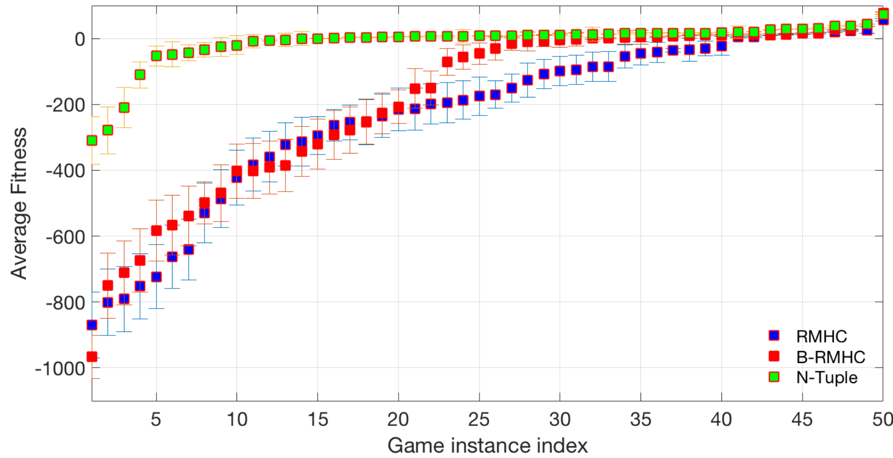
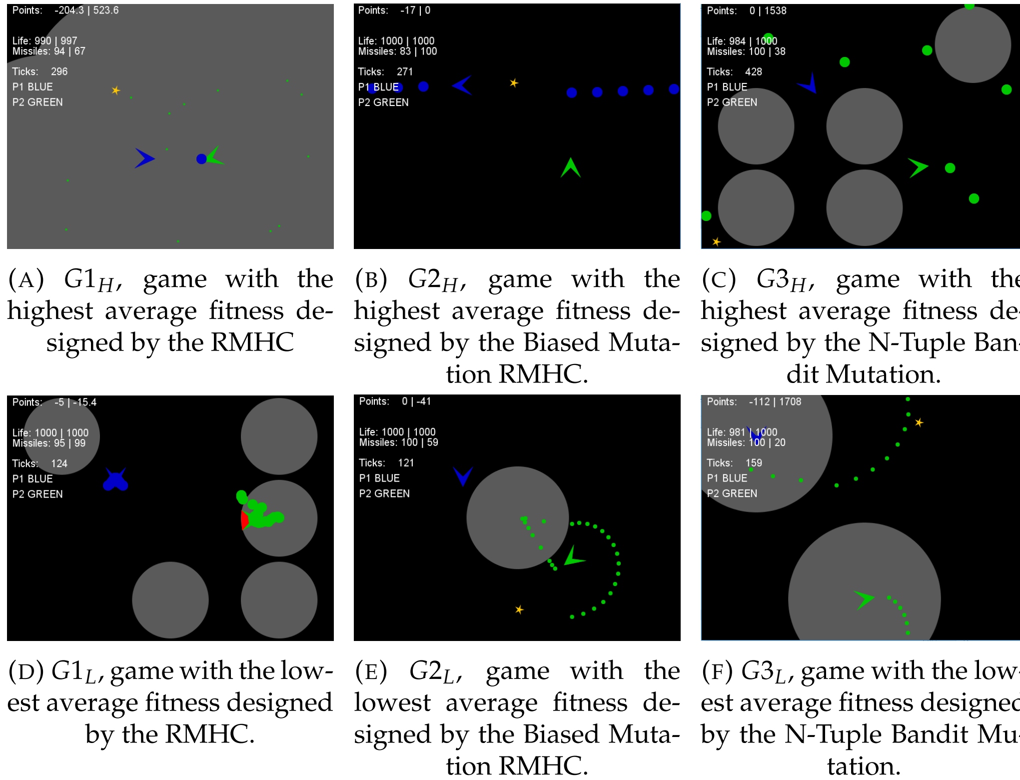


FIGURE 5.1: Sorted average fitness values over 100 evaluations of 50 game instances evolved using three different algorithms

challenging aspect of the game and is attracted more towards uncommon game scenarios; Player B was less easily satisfied and found most of the games boring. Interestingly, though they have ranked the games differently, they both have a preference for the game $G3_H$ (with the highest average fitness value, optimised by NTBEA) and dislike the games $G1_H$ (with the highest average fitness value, optimised by RMHC) and $G2_H$ (with the highest average fitness value, optimised by B-RMHC). The **Player A** ranked the games as $G1_L > G3_H > G1_H > G3_L > G2_H > G2_L$ in terms of challenge/fun, while the **Player B** ranked the same games as $G2_L > G3_H > G3_L > G1_H > G1_L > G2_H$. This shows that NTBEA more successful in evolving Space Battle Evolved parameters that satisfy two human tester preferences in this experiment. This, however, is not sufficient to validate the HS1, but provides a positive outcome. Hence, HS1 is inconclusive in general since there are not enough evidence. Similarly with HM3, we used general video game agents (1SLA and MCTS) to as representatives for human players with different skill-depth, but it is still inconclusive in general.

In this section, we present the results of an experiment that was designed



(A) $G1_H$, game with the highest average fitness designed by the RMHC

(B) $G2_H$, game with the highest average fitness designed by the Biased Mutation RMHC.

(C) $G3_H$, game with the highest average fitness designed by the N-Tuple Bandit Mutation.

(D) $G1_L$, game with the lowest average fitness designed by the RMHC.

(E) $G2_L$, game with the lowest average fitness designed by the Biased Mutation RMHC.

(F) $G3_L$, game with the lowest average fitness designed by the N-Tuple Bandit Mutation.

FIGURE 5.2: Screenshots of the 6 designed games evaluated by human players and their feedback.

to validate two main hypotheses HM1 and HM3 of the research. The task selected was Space Battle Evolved parameterization to find games that best distinguish players based on their skill-depth. Next, the results of optimizing another player-experience task based on score trend using other noisy environments are presented.

5.2 GVGAI Games

As we aimed to validate HM2 and HM3 in this experiment, we ran the NTBEA to evolve game parameter sets for 3 GVGAI games (Defender, Waves and Seaquest), fitting each of 9 target functions (3 linear functions, 2 linear piecewise functions, a logarithm function, an exponential function and 2 shifted sigmoid functions), with two fitness calculation types (NRMSE and B-NRMSE), that is 54 different settings in total. There were 10 games evolved for each distinct setting

and the outcomes were averaged to present the evolution result of such setting. The maximum timestep was fixed at 500 for all gameplay to avoid exploding value for the exponential function ($2^{\frac{1000}{70}} \approx 20000$) that would lead to unachievable score for the current game space. Validation repetition for NTBEA was set at 20 iterations, with 10 neighbour points taken into consideration.

Initially, evaluation budget was set at 10000, but after observing Seaquest logarithmic runs that converged before 2000 generations, we have changed it to 2000 to reduce computation time. However, Defender fitness trend seemed to still be unstable after 2000 generations evolved, therefore we have increased it to 5500 only for this game.

RHEA was selected as player controller during the parameter evolution. Afterwards all evolved games were being validated 10 times each using MCTS as the controller. RHEA population size was set at 20, while the length of the individuals was set to 10. The mutation rate was set at 0.1, meaning that one gene is mutated on average on each individual. For MCTS, its rollout depth was set at 10 to match the length of the RHEA individuals. This is to ensure that our evolved games provided environments for that specific score trend, regardless of the controllers played. Both RHEA and MCTS used the same heuristic function to evaluate states: as per Equation 5.1: the score is used as fitness (resp. reward in MCTS) unless the games end with a victory or loss, in which case the reward is 1000 or -1000 respectively. The C value for the UCB1 equation is $\sqrt{2}$ for both the MCTS agent and the NTBEA. The number of neighbours for NTBEA was set to 100.

$$value(state) = \begin{cases} 1000, & \text{if state result is winning} \\ -1000, & \text{if state result is losing} \\ score, & \text{otherwise} \end{cases} \tag{5.1}$$

These pre-defined parameters are summarized in Table 5.2. Result analysis is

TABLE 5.2: Experiment parameters

Game Specific			Global			
Games	**Budget**	**Max step**	**Evolution Agent**	**Validation Agent**	**Neighbour Size**	**Repetition**
Defender	5500	500	RHEA	MCTS	10	20
Seaquest	10000 for log, 2000 else	500				
Waves	2000	1000 for log, 500 else				

divided into two sections, from the game evolution and validation phase respectively.

5.2.1 Evolving Game Parameters

We measured NTBEA performance in game parameter evolution using three matrices: Average fitness trends over generations, Average score for each time step in 5 generation ranges, and average score difference (slope of score trend) for each time step also in 5 generation ranges. The average fitness trend was used as guidelines to select the last generation to visualize for the score and score slope trend. To clarify, if the fitness values were stable after 500 generations, only the average score and difference from generation 1 to 500 would be shown. This is for ease of observing how the score trends were evolved compared with the actual target functions.

We have compared and analyzed the evolution results in three aspects: between games, between biased and non-biased NRMSE, and between target functions.

Different Games

We have selected Defender, Seaquest and Waves for the experiment because their score system are incremental, which game score can change during gameplays if certain conditions are met. This makes them possible, in theory, for the players to obtain the same or similar score trend with any of our target functions

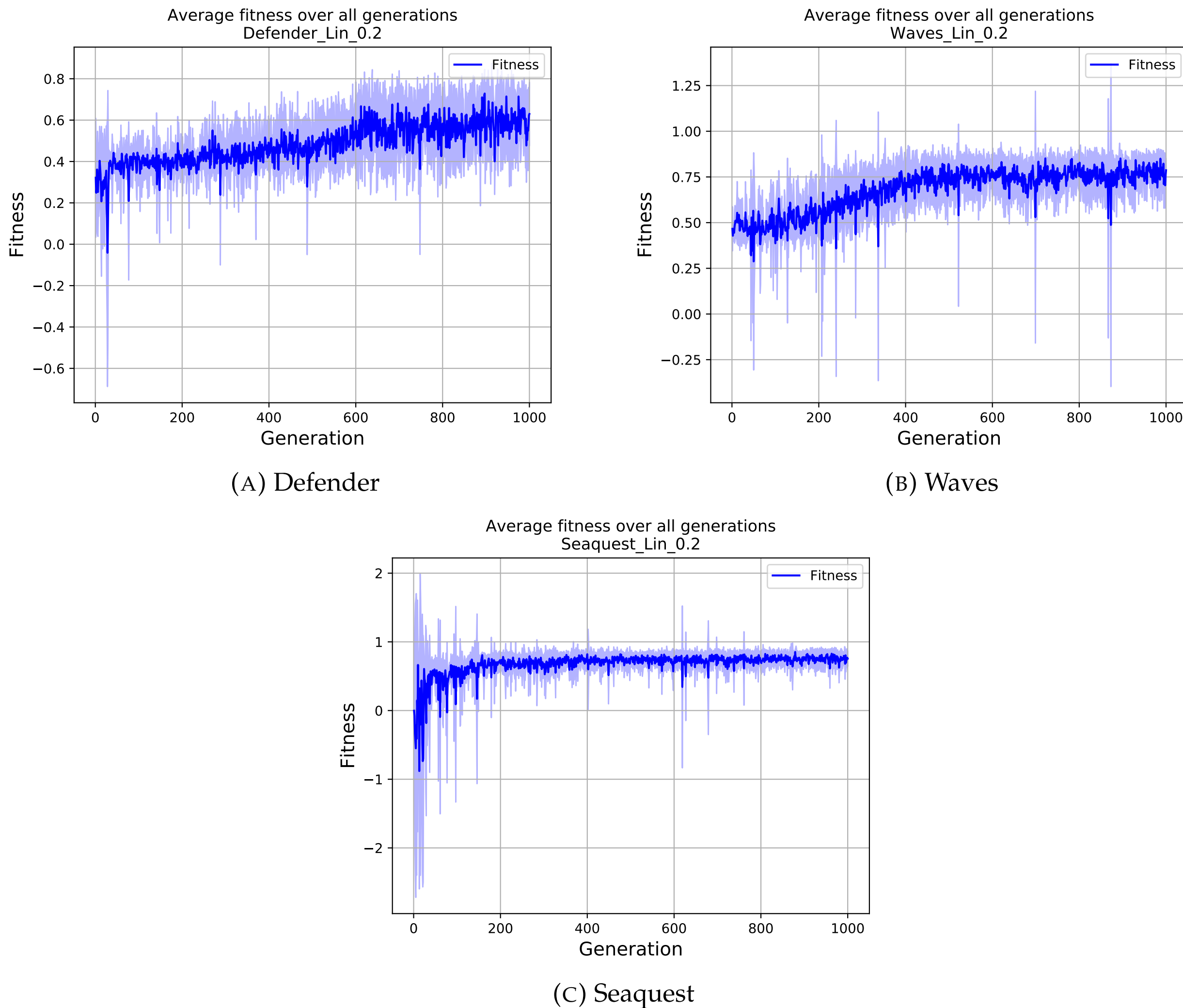
if the parameters are set properly. Our minor hypothesis for this between-game comparison is:

- **HG1:** NTBEA can evolve game parameters to fit the same target functions for any games.

We compared the results between all three games in fitting the same target function and the same loss calculation, beginning with linear function with $m = 0.2$ ($y = 0.2x$) since it was the least challenging function to fit (the m value is considerably small).

The fitness trend, averaged from 10 distinct evolutions, over the first 1000 generations of three games are shown in Figure 5.3. The lighter blue shaded areas indicate standard deviation of the values. It can be seen that the fitness values were stable at generation 1000 for all games, which for Seaquest (Figure 5.3c) it converged before generation 200, while it took at about 500 generations for Waves (Figure 5.3b). Defender fitness, shown in Figure 5.3a, were growing slower than the previous two with about 800 generations to stay non-increased, with significantly higher standard deviation. For this matrix, Seaquest seems to have the least deviate trend, followed by Waves and Defender respectively.

Score and score difference trends also present the similar results, as depicted in Figure 5.4. All target lines are plotted in red. Notice that the final generations shown for each game are different, as they were selected based on their own fitness trend. Seaquest first 50 generation gameplays had a score trend (blue line in Figure 5.4e) that is above the target score line by a significant gap, and managed to find the parameter set that lower the score down to fit the target line more. In contrast, Waves initial parameter set did not provide as high score as the target line, but it showed a promising progress during evolution, with RHEA scored more in the later game set, as can be seen in Figure 5.4c. The score difference trend (Figure 5.4f) also showed the slope curves moved closer to the target line as the game evolved. Defender (Figure 5.4a and 5.4b) was struggling



(A) Defender

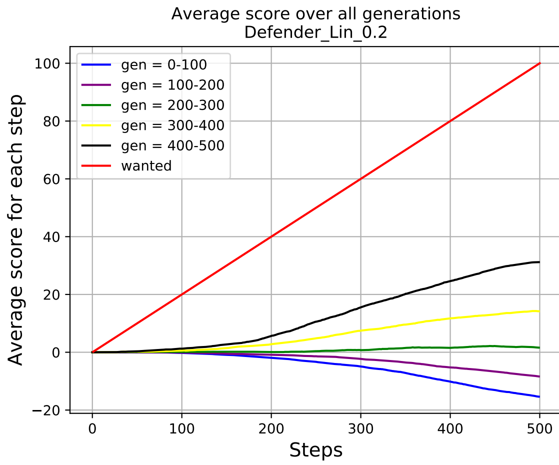
(B) Waves

(C) Seaquest

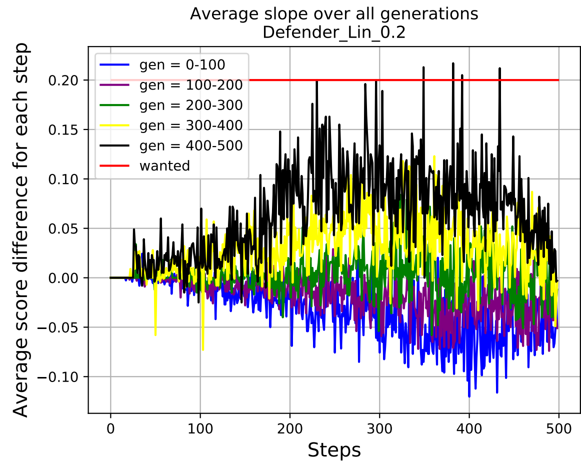
FIGURE 5.3: Average fitness throughout evolutions for $y = 0.2x$ on the games of this study

the most with negative score trend in the beginning, but improved with time and obtained a positive linear-like trend after generation 800.

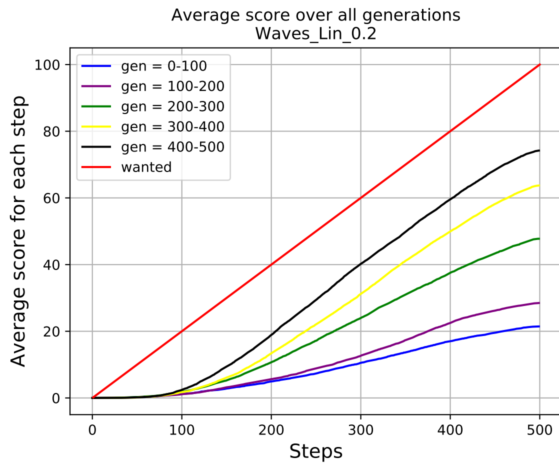
Another example in this between-game comparison is from logarithm function, with NRMSE loss calculation. Fitness trends are plotted in Figure 5.5 while the score and slope trends are given in Figure 5.6a. Similar to the results from $y = 0.2x$ function, Fitness values seem to converge the fastest for Seaquest (5.5c), with only about 200 generations of parameter evolution. This was followed by Waves (5.5b) with about 500 game parameter sets evolved, and lastly Defender (5.5a) after generation 1500.



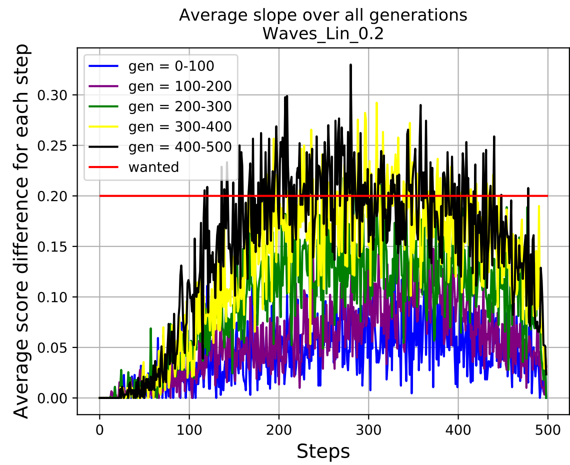
(A) Defender score



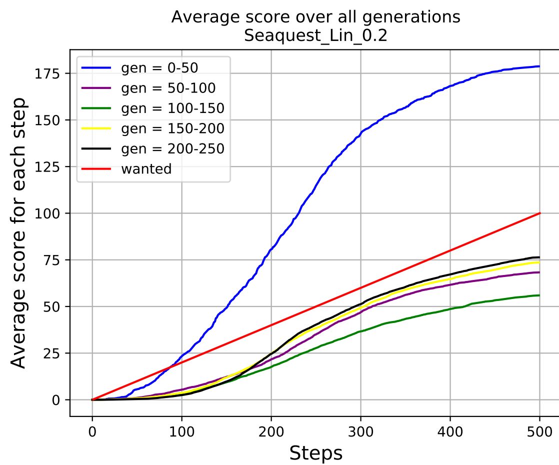
(B) Defender score slope



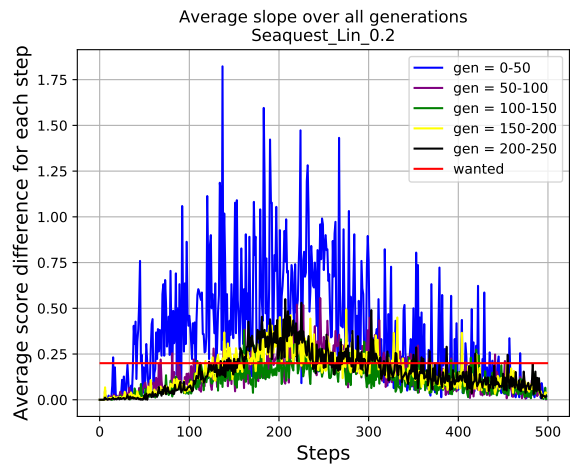
(C) Waves score



(D) Waves score slope

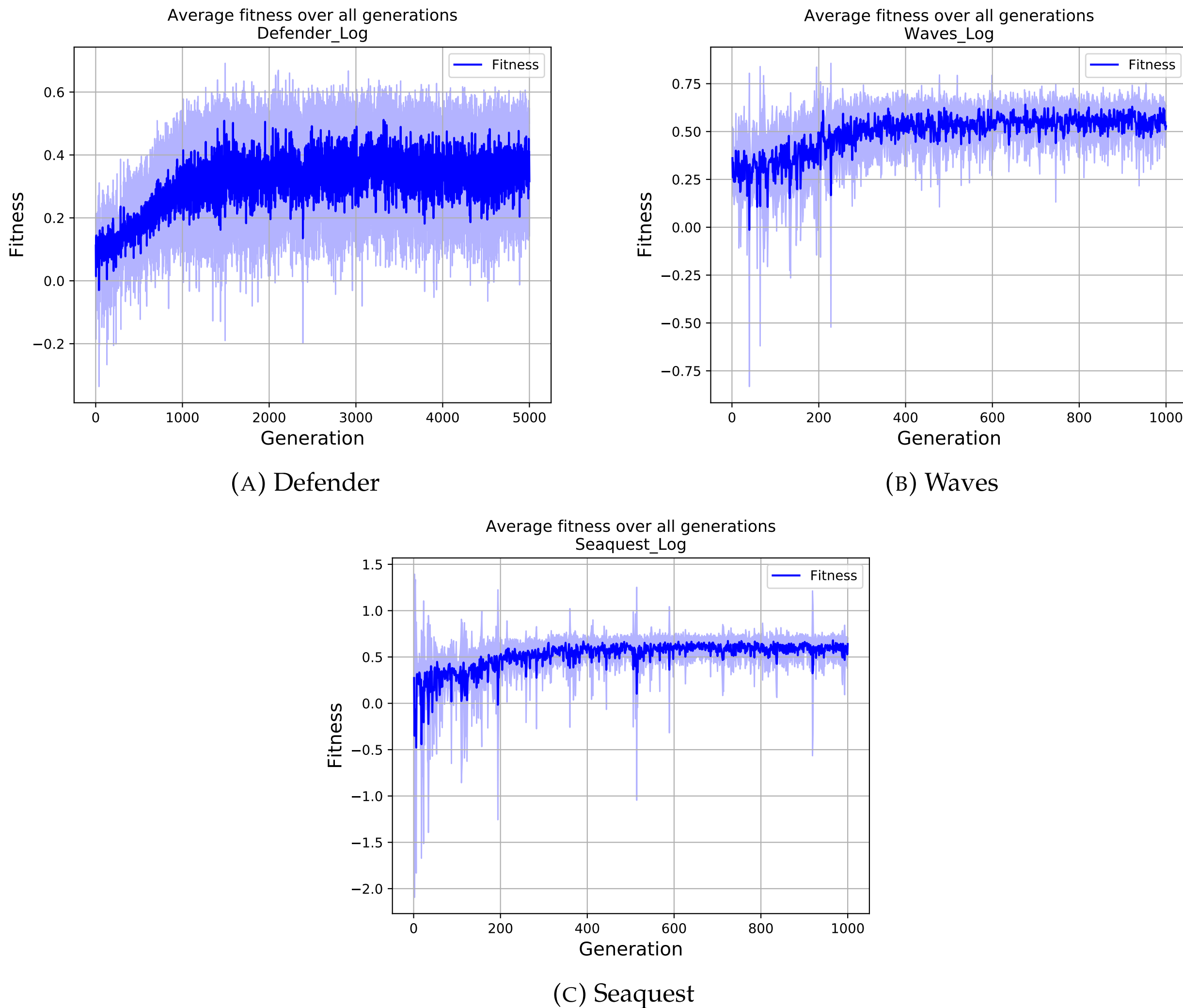


(E) Seaquest score



(F) Seaquest score slope

FIGURE 5.4: Average score trend and score difference throughout evolutions for $y = 0.2x$ on the games of this study. Each plot shows different trends, averages taken at different generation ranges through evolution.



(A) Defender

(B) Waves

(C) Seaquest

FIGURE 5.5: Average fitness throughout evolutions for $y = 15\log_2(x)$ for the games of this study

In score trend, it is difficult to see in Defender (5.6a) whether it is following the trend we want (scoring more in the beginning and less later on), but it is clearer in score difference (5.6b) as the later generation game parameters has higher slope, which the peak point is at around generation 180, and the slope decreases slowly afterwards. The earlier generations, such as those represent in blue lines have clearly lower average score slope. This implies that the EA responded to our fitness function and was trying to adjust parameters accordingly. Waves results (Figure 5.6c for score and 5.6d for score slope) are a bit more noticeable that the average score was increasing faster in the early steps for all

generation ranges. Therefore the EA seemed to be trying to fit the actual values of our target curve, resulting in higher slope in later generations. For Seaquest (5.6e for score and 5.6f for score slope), the EA could find a parameter set with the final score value (at step 500) close to the target function value quite early, therefore it was mostly evolving to fit with the rest of the curve. This is reflected more obviously in the score slope that the peak area has moved from step 250 (blue jagged curve) to around 150 (black jagged curve), and the slope before step 100 is significantly higher in generation 600-750 than in the beginning.

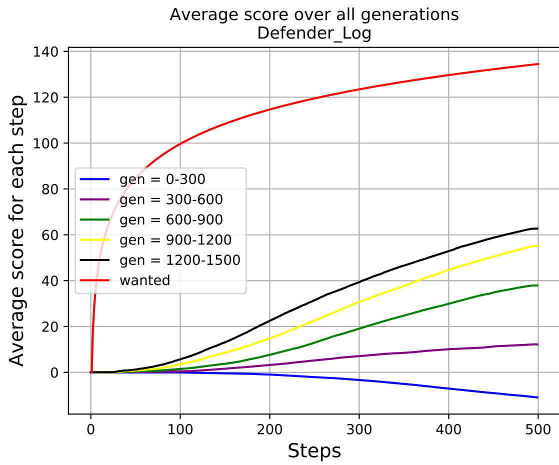
From two sets of examples provided, we can see that NTBEA reacted to our fitness function and attempted to find parameter set that gives the fittest result for all games tested in this experiment. Results from other target functions are also similar to this trend. Although this is not sufficient to confirm that the minor hypothesis HG1 is correct, it provides some evidence in positive direction. For an additional observation, we can see that Defender game space was the most challenging for the EA, while Waves was less difficult and Seaquest was the easiest.

Biasing Loss Calculation

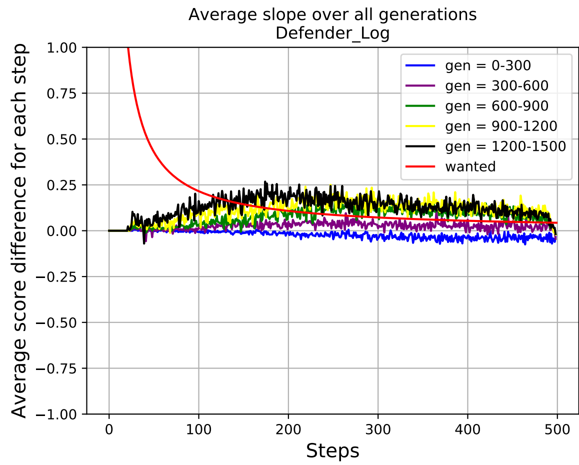
There were two loss calculation functions used in the experiment, Non-biased (NRMSE) and Biased Normalized Root Mean Square Error (B-NRMSE), as described in section 4.2.4. The main difference is that B-NRMSE penalizes errors in the area with higher slope in the target functions more, while normal NRMSE treats errors at all points equally. The minor hypothesis we set for this comparison is:

- **HG2:** Biasing the loss calculation helps NTBEA in our optimization task

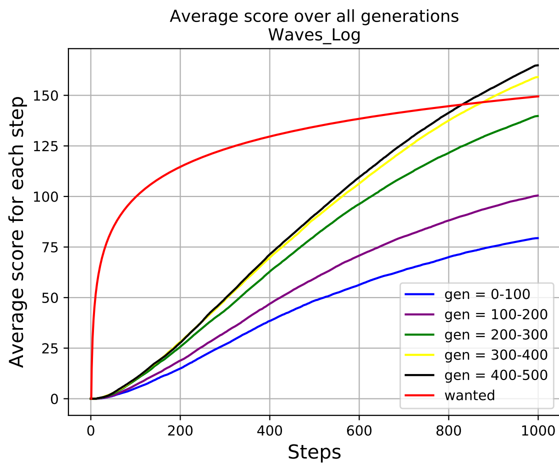
Based on the results, biasing the loss calculation resulted in both positive and negative outcomes in our experiment depending on the target functions. An advantage of biasing the errors is that the EA more actively reacted to those



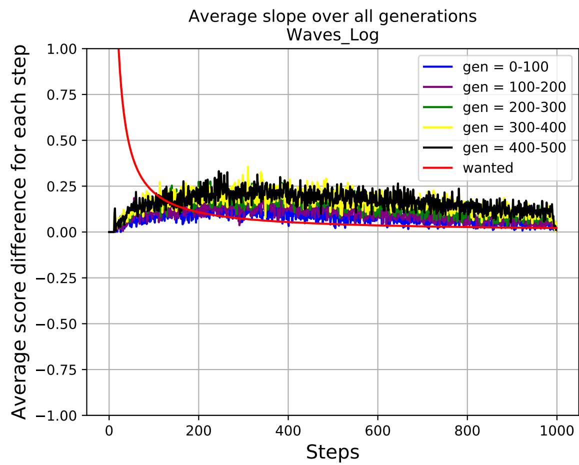
(A) Defender score



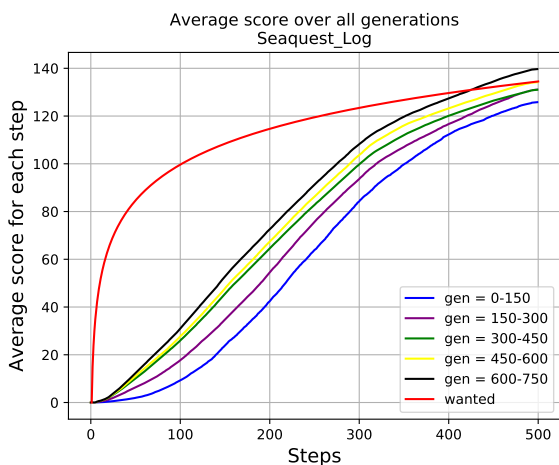
(B) Defender score slope



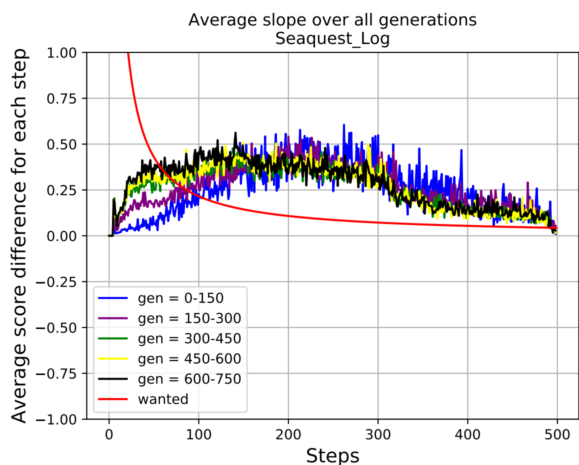
(C) Waves score



(D) Waves score slope

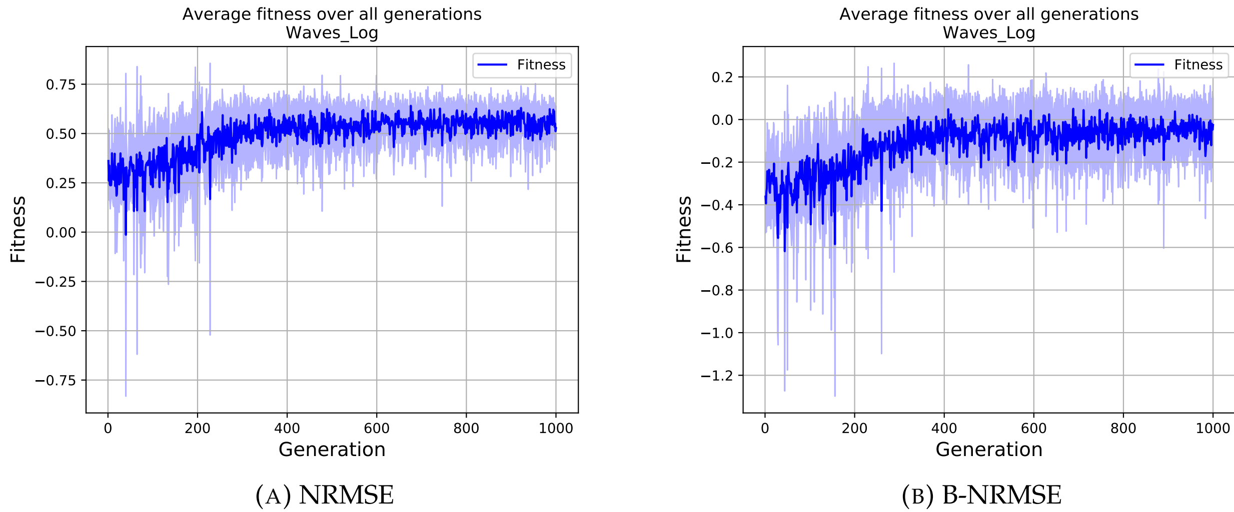


(E) Seaquest score



(F) Seaquest score slope

FIGURE 5.6: Average score trend and score difference throughout evolutions for $y = 15\log_2(x)$, for the games of this study.



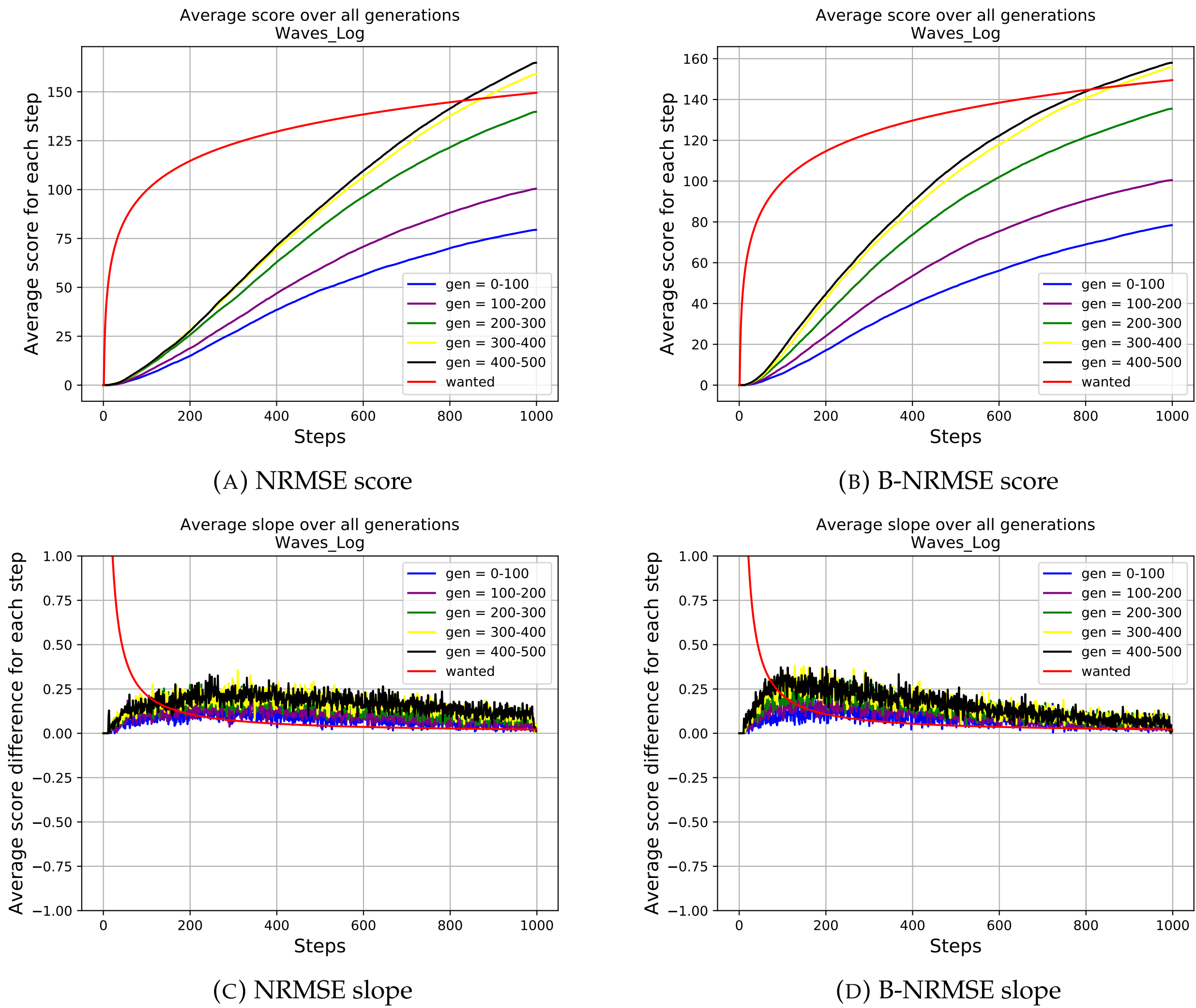
(A) NRMSE (B) B-NRMSE

FIGURE 5.7: Average fitness throughout evolutions for $y = 15\log_2(x)$, normal and biased loss function comparison

areas with higher slope values and found more parameter sets that fit those parts better. One example of this can be seen in Figure 5.7, which shows a comparison between normal and biased NRMSE in Waves for $y = \log_2(15x)$. It can be seen that the black curve (generation 400-500) in the biased version graph (Figure 5.8b) has higher average value at about 90, compared with about 70 in the non-biased graph (Figure 5.8a). This is more obvious in the slope results (Figure 5.8c and Figure 5.8d) that the black jagged curve of the biased graph has higher values at step 100 than the non-biased graph, and decreasing faster afterwards.

On the other hand, a disadvantage of biasing the errors is that it would neglect all errors from the areas with zero slope in the target curve. This can be seen in the results of left-sigmoid curve that the y value always equals to 150 after step 250, hence the slope is 0 from this step onwards. This is shown in Figure 5.9. The average score of normal NRMSE (Figure 5.9a) were restricted to be at most equal to the red curve values, while it is untrue for the biased one (Figure 5.9b). However the slope in biased NRMSE is closer to the target slope than the non-biased one.

Another observed effect of biased error calculation was that it linearly transformed the fitness values down. This is true for all functions, excepts linear



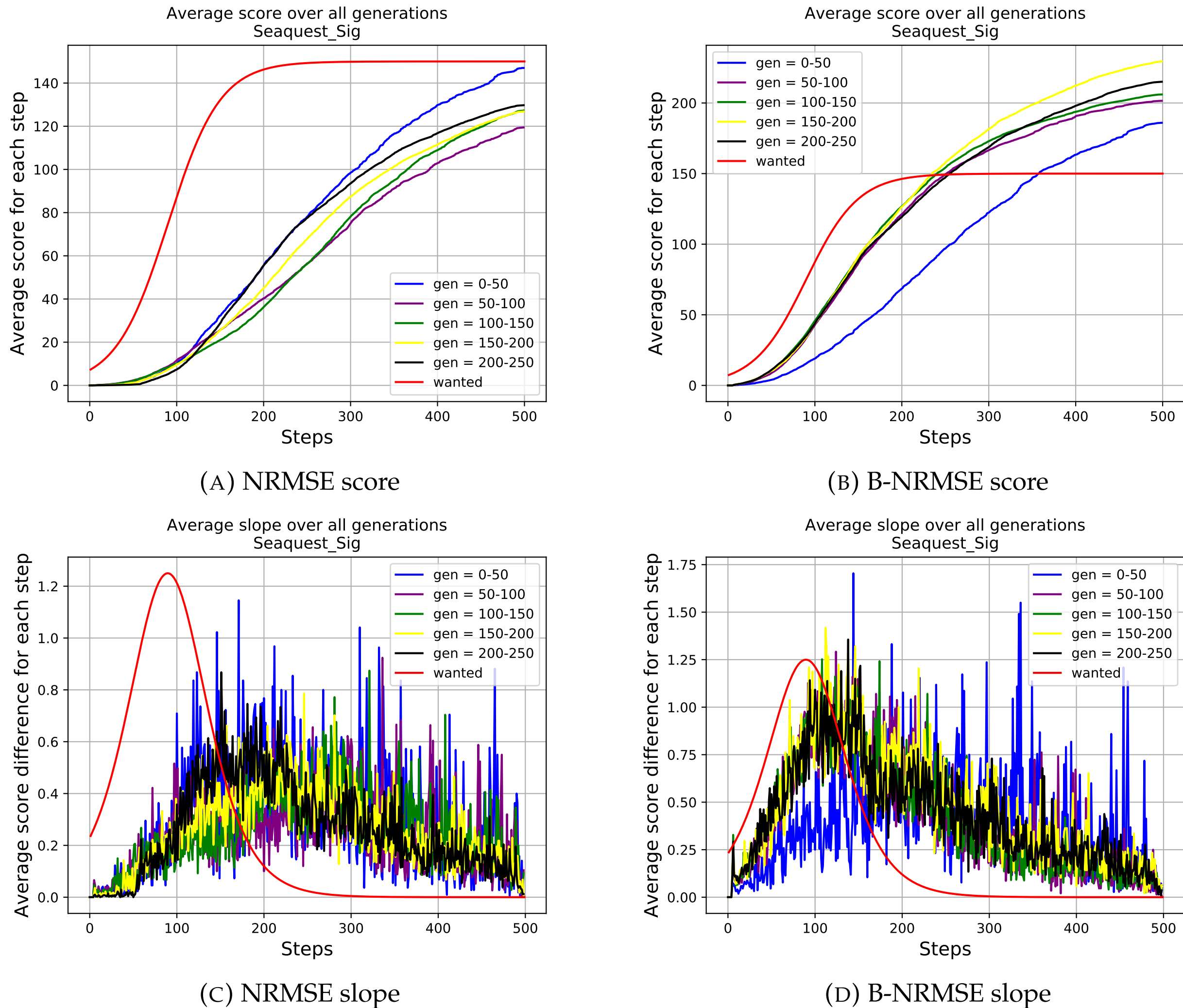
(A) NRMSE score

(B) B-NRMSE score

(C) NRMSE slope

(D) B-NRMSE slope

FIGURE 5.8: Average score trend and score difference throughout evolutions for $y = 15\log_2(x)$, different loss function comparison.



(A) NRMSE score

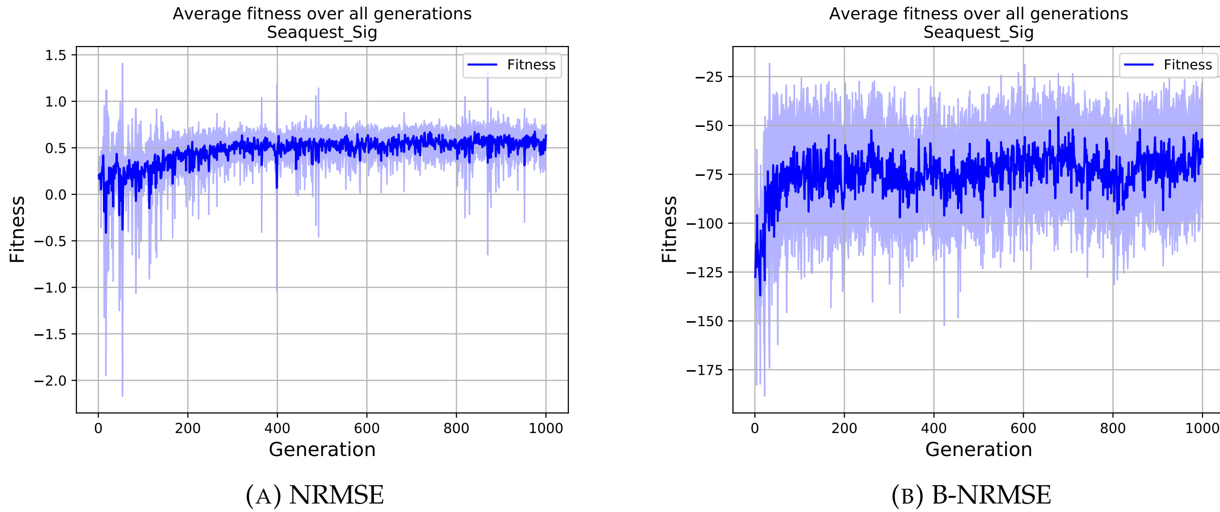
(B) B-NRMSE score

(C) NRMSE slope

(D) B-NRMSE slope

FIGURE 5.9: Average score trend and score difference throughout evolutions for left-sigmoid, for NRMSE and B-NRMSE loss calculation.

functions that have fixed-value slopes. For instance, as given in Figure 5.7, the final average fitness values of normal NRMSE are about 0.5 while they are less than 0 for the biased one. More obvious example can be seen in Figure 5.9, with the final fitness values are 0.5 and -75 for non-biased and biased versions respectively. It is also shown that the biased loss calculation gave significantly higher standard deviation (blue shaded area) in this case, which is also true in others from my observation. Since we observed mixed results from this comparison, HG2 is inconclusive.



(A) NRMSE

(B) B-NRMSE

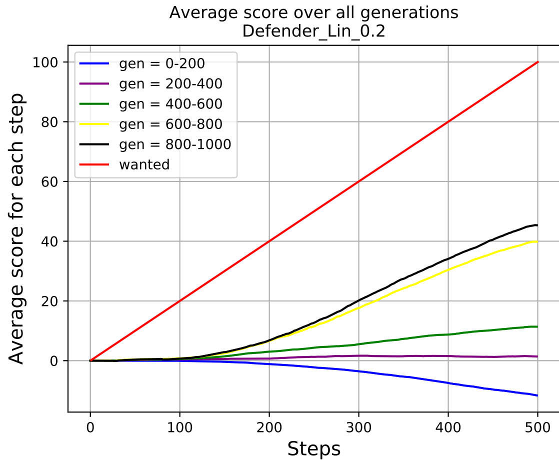
FIGURE 5.10: Average fitness throughout evolutions for $y = \frac{150}{1+\exp(-\frac{x}{20}+12)}$, different loss function comparison

Functions and Parameters

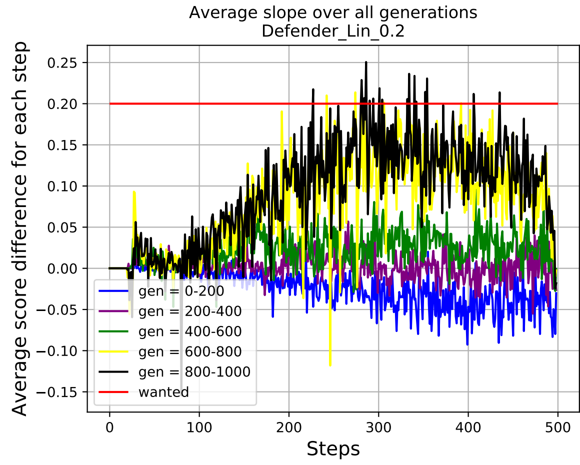
We compared the results from fitting different target functions, by pairing the same types with opposite trends (for instance, higher slope in the beginning and higher slope in the end) together, except linear lines that we compared different m in $y = mx$.

Results from Defender were selected to present linear target function comparison because it was the most challenging game to evolve, as described in 5.2.1. Score and score slope trends are given in Figure 5.11. It is obvious that the target lines with higher slope was more difficult to fit for this game. Nevertheless, the average final score (at step 500) are different, as it is more than 40 for $m = 0.2$, approximately 75 for $m = 0.4$ and slightly more than 100 for $m = 1$. We can infer from this that the EA was affected more by the higher the target slope is and was trying harder to fit it. Average score slope results also confirm this as they have higher peak values with higher m target function.

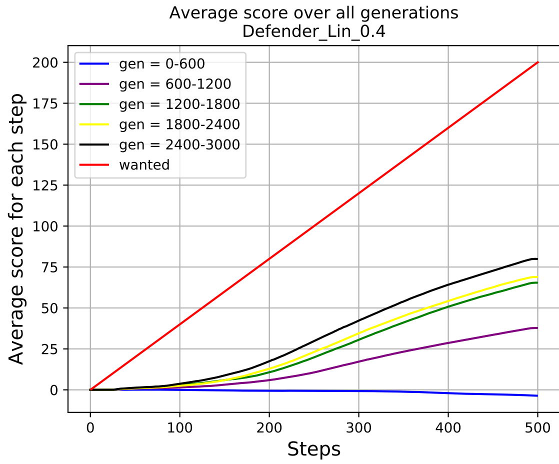
Next, we compare logarithm and exponential function results as they are the inverse function of each other. Waves with B-NRMSE were selected due to



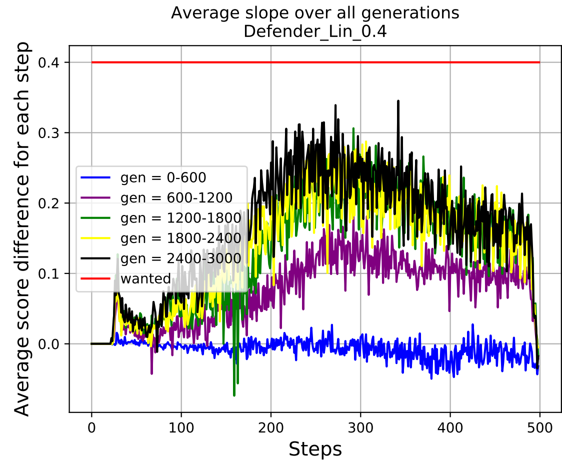
(A) $y = 0.2x$ score



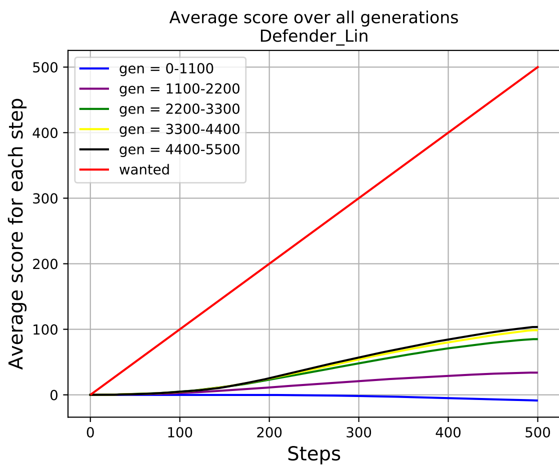
(B) $y = 0.2x$ slope



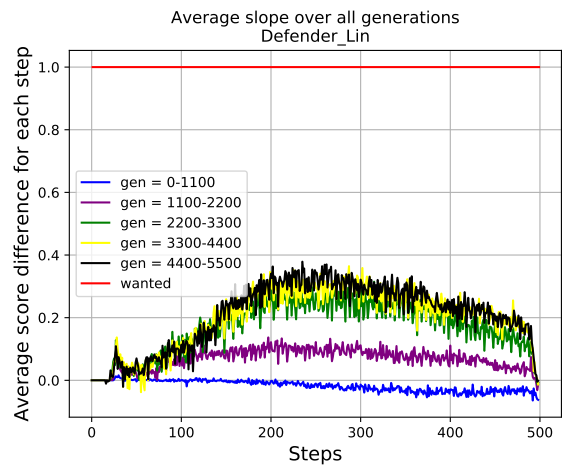
(C) $y = 0.4x$ score



(D) $y = 0.4x$ slope



(E) $y = x$ score



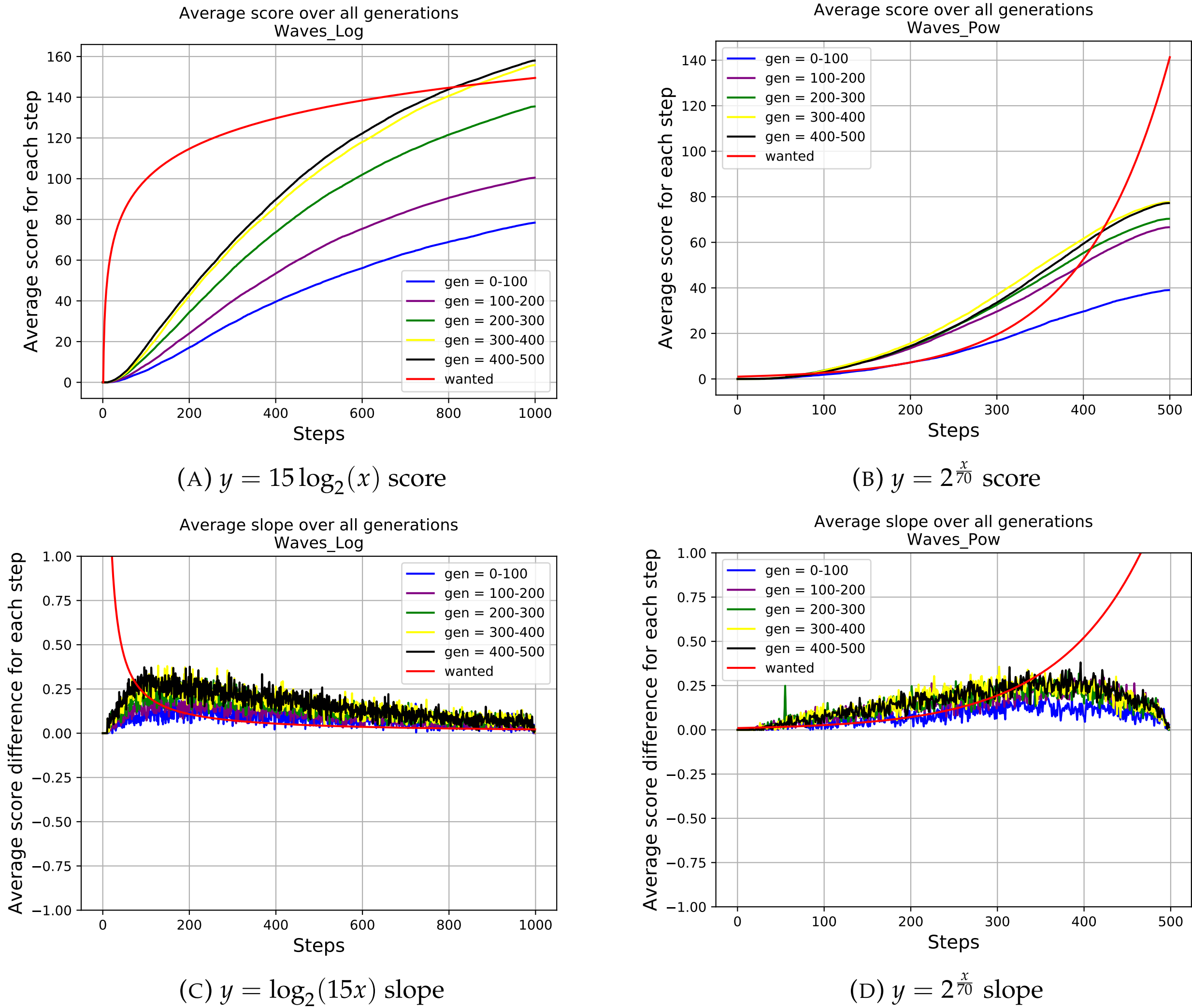
(F) $y = x$ slope

FIGURE 5.11: Average score trend and score difference throughout evolutions for Defender, linear function comparison.

its clearest distinction of the graphics plotted, hence easiest to visualize the differences. The graphs showing average score and slope trends are illustrated in Figure 5.12. It is obvious that the EA was evolving game parameters to fit the given target functions, as the average score values at step 200 for later generations for logarithm function result is significantly higher than the exponential function result. Visualizing average slope trend also confirms this as the peak point of the logarithm result is at around step 100 (for the black line) while for the exponential result it is at around step 400. Based on the results, the EA obviously reacted differently to these two target functions, and producing similar score difference trend to each.

Left and right shifted sigmoid functions, given in Figure 5.13, is compared next. Normal NRMSE Waves results were selected because, again, they are easiest to visualize. The results are similar to those in logarithm and exponential function as the average score in later generation for left sigmoid result is higher than right sigmoid result at step 300. Notice that the blue curves of both results are almost the identical (taking the y-scale into account), pointing out that the initial parameter of both cases were giving the similar score trend (although the initial points in the game space were randomly picked), and the EA managed to evolve games with different trends in the end. This means that the evolutionary algorithm and the approach we are proposing is general and adaptable with various target functions.

Unexpectedly, The most challenging functions to fit were 2-part linear piecewise. B-NRMSE Waves results were selected to show in Figure 5.14. Although the results are of the same trends with those in sigmoid and logarithm-exponential comparisons, the EA could not efficiently react to the section with $m = 2$ in both runs. My assumption is that the m values were probably too high, as we have seen from the linear target function parameter comparison, it was challenging to even fit a linear line with $m = 1$ from the beginning. Further experiment with smaller m for this piecewise setting might confirm this. Another possible



(A) $y = 15\log_2(x)$ score

(B) $y = 2^{\frac{x}{70}}$ score

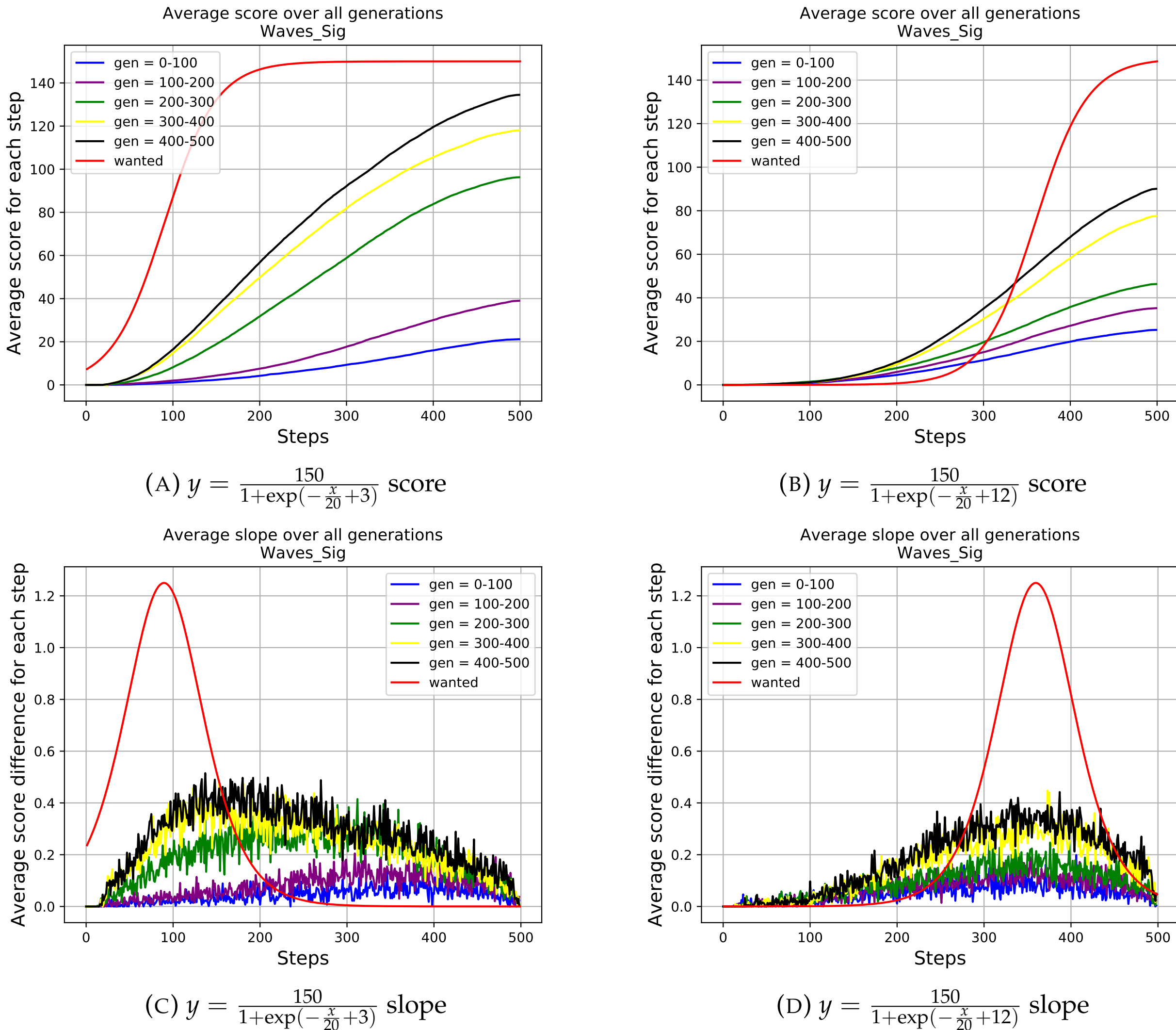
(C) $y = \log_2(15x)$ slope

(D) $y = 2^{\frac{x}{70}}$ slope

FIGURE 5.12: Average score trend and score difference throughout evolutions for Waves, B-NRMSE, logarithm and exponential function comparison

assumption is that the 'change point', which is 100 for the 2-0.2 and 400 for the 0.2-2, were probably need to be closer to the middle step to provide more time to react on it. Modifying this or increasing game playing time in further experiment may confirm this.

In this subsection, I have analyzed the data resulted from the parameter evolution phase using RHEA as the controller to play games. The results were compared between games, between using normal and biased NRMSE as loss calculation function, and between fitting different target functions of the same types.



(A) $y = \frac{150}{1+\exp(-\frac{x}{20}+3)}$ score

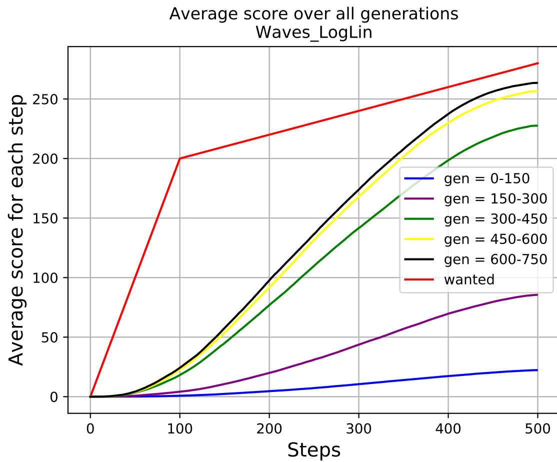
(B) $y = \frac{150}{1+\exp(-\frac{x}{20}+12)}$ score

(C) $y = \frac{150}{1+\exp(-\frac{x}{20}+3)}$ slope

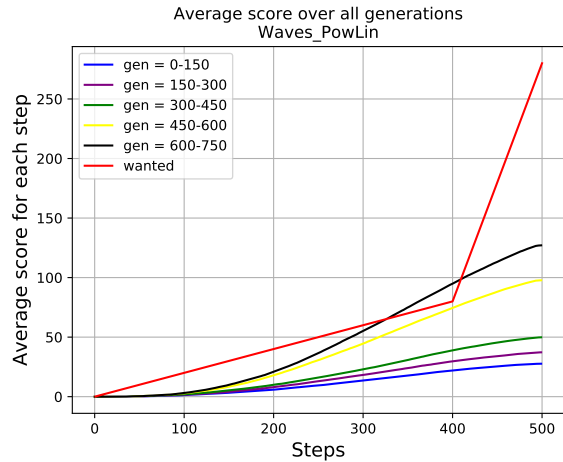
(D) $y = \frac{150}{1+\exp(-\frac{x}{20}+12)}$ slope

FIGURE 5.13: Average score trend and score difference throughout evolutions for Waves, normal NRMSE, shifted sigmoid function comparison

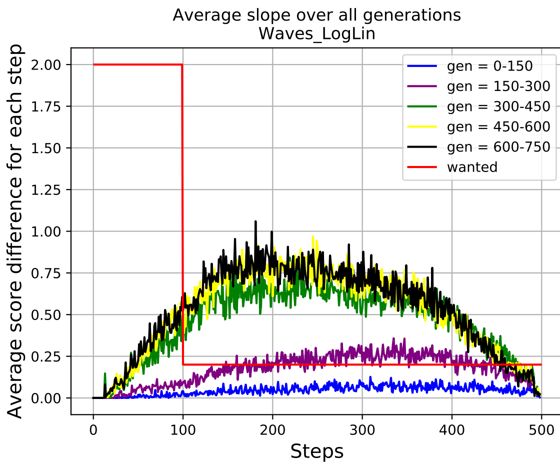
Based on the results, I conclude that the EA can react to the same target function similarly across different games. Biasing the loss calculation resulted in higher sensitivity of the EA, but less robust as it increased standard deviation range. Comparison between different target functions indicates that our NTBEA was able to react to different functions appropriately. Next, some of the evolved parameter sets were taken into validation phase, which is to set the game with such parameters, playtest using another controller, record the results, and analyze the results to confirm whether they provide the environment to achieve such score



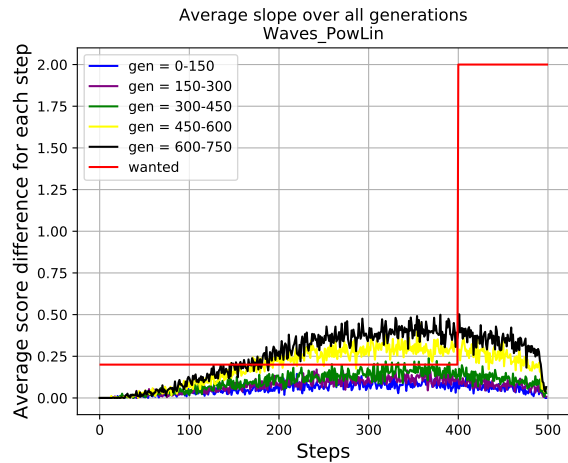
(A) 2-0.2 linear piecewise score



(B) 0.2-2 linear piecewise score



(C) 2-0.2 linear piecewise slope



(D) 0.2-2 linear piecewise slope

FIGURE 5.14: Average score trend and score difference throughout evolutions for Waves, B-NRMSE, 2-part linear piecewise function comparison

trends.

5.2.2 Validating Evolved Games

After game parameter sets had been evolved, the EA reported the best individual on each of the four selection criteria, as mentioned in 3.3.3. We extracted these parameter individuals from every settings in all 10 runs and performed validation playtests using MCTS, 10 times each for one result (more validation may be necessary to reduce the standard error). The score from these playtests

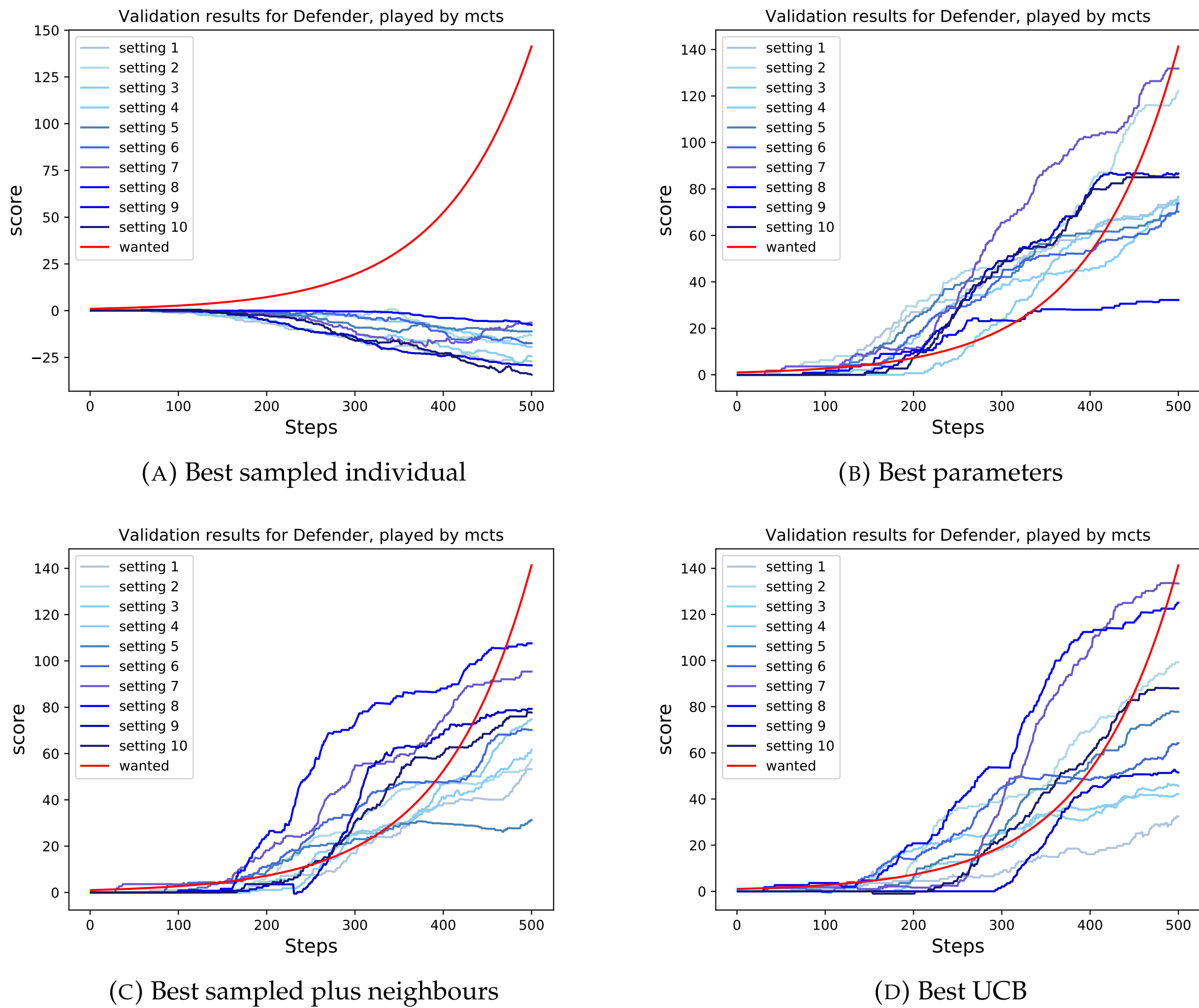
were recorded, averaged for the same parameter set, and plotted along with others in the same evolution configuration. We first compare the results from different selection criteria, then follow with the same comparison criteria with the previous section.

Best Individual Selection Criteria

There are four best individuals selected by the EA for each game parameter evolution. The first one called best *sampled* is the group of parameter that has achieved the highest fitness among all the sampled individuals during evolution. The best *params* is the combination of the parameters with the highest average fitness individually. The best *s+neighbours* is the best set among all sampled individuals with their neighbours (the sets that share all but one parameter values). Finally, the best *UCB* are the best of all sampled points, ranked by UCB values (equation 3.1) instead of only fitness values.

We selected the result from normal NRMSE Defender with exponential target function as it is the best to reflect the differences between each best selection parameter sets. The average score trend of this configuration in validation phase is given in Figure 5.15 and the average slope trend is shown in Figure 5.16. Each blue curve represents the average of playtest results from one game evolved, with the red curve is the target function. It can be seen that the best *sampled* parameters did not provide environment for MCTS to obtain score in the same trend as our target function. The other three best parameter sets achieved more promising performance as most of the games ended in high positive score. For the slope trend, the best *UCB* and the best *s+neighbour* seemed to outperform the other two with clearer higher slope trend in the later steps than the beginning.

To confirm that this performance pattern is also true in other settings, we show the validation results for normal NRMSE Seaquest in linear line $y = x$ as both the game and the function were less challenging to evolve in evolution phase. The average score is given in Figure 5.17 and the score slope is depicted



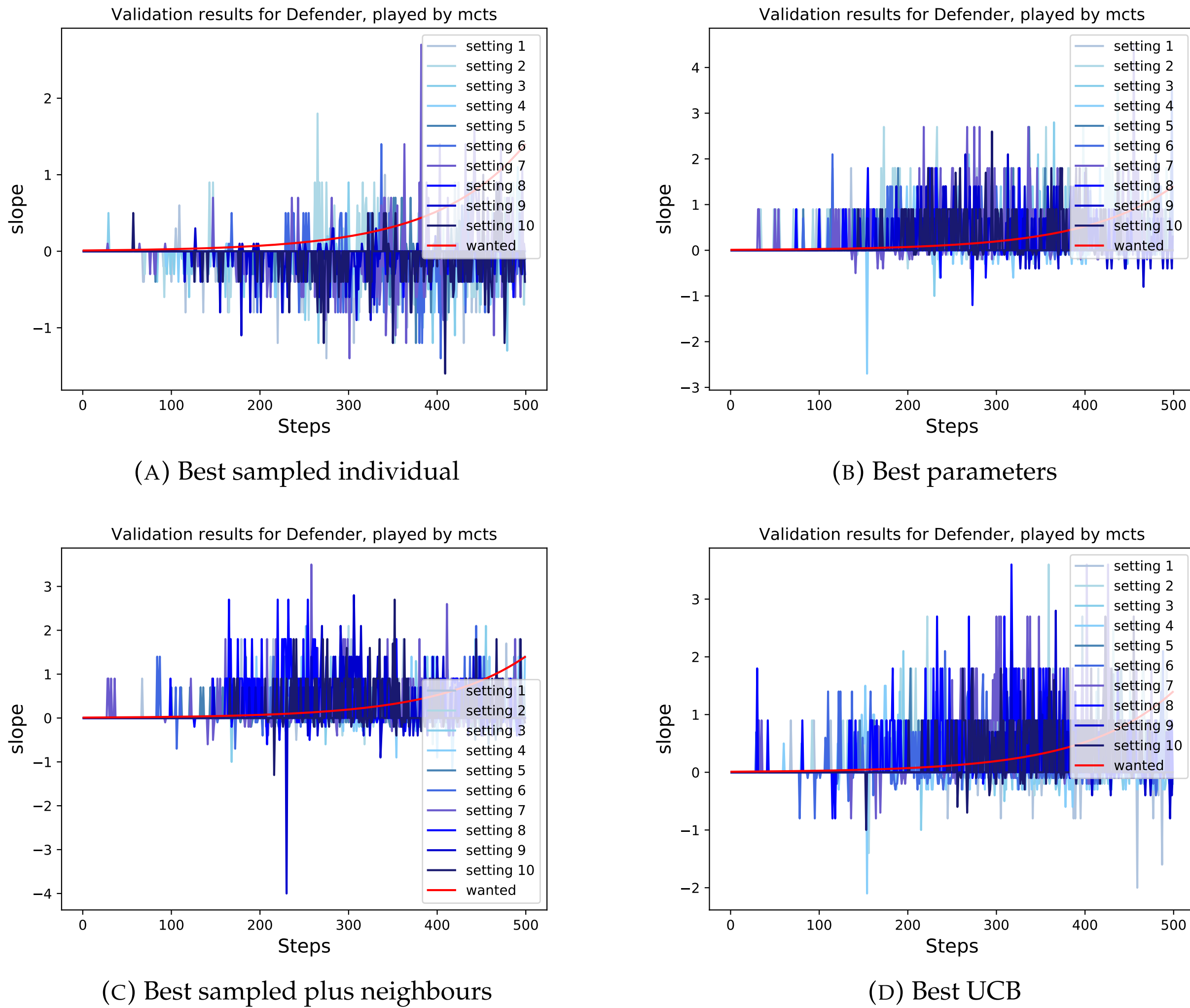
(A) Best sampled individual

(B) Best parameters

(C) Best sampled plus neighbours

(D) Best UCB

FIGURE 5.15: Average score trend in validations for $y = 2^{\frac{x}{70}}$, different best parameter selection comparison



(A) Best sampled individual

(B) Best parameters

(C) Best sampled plus neighbours

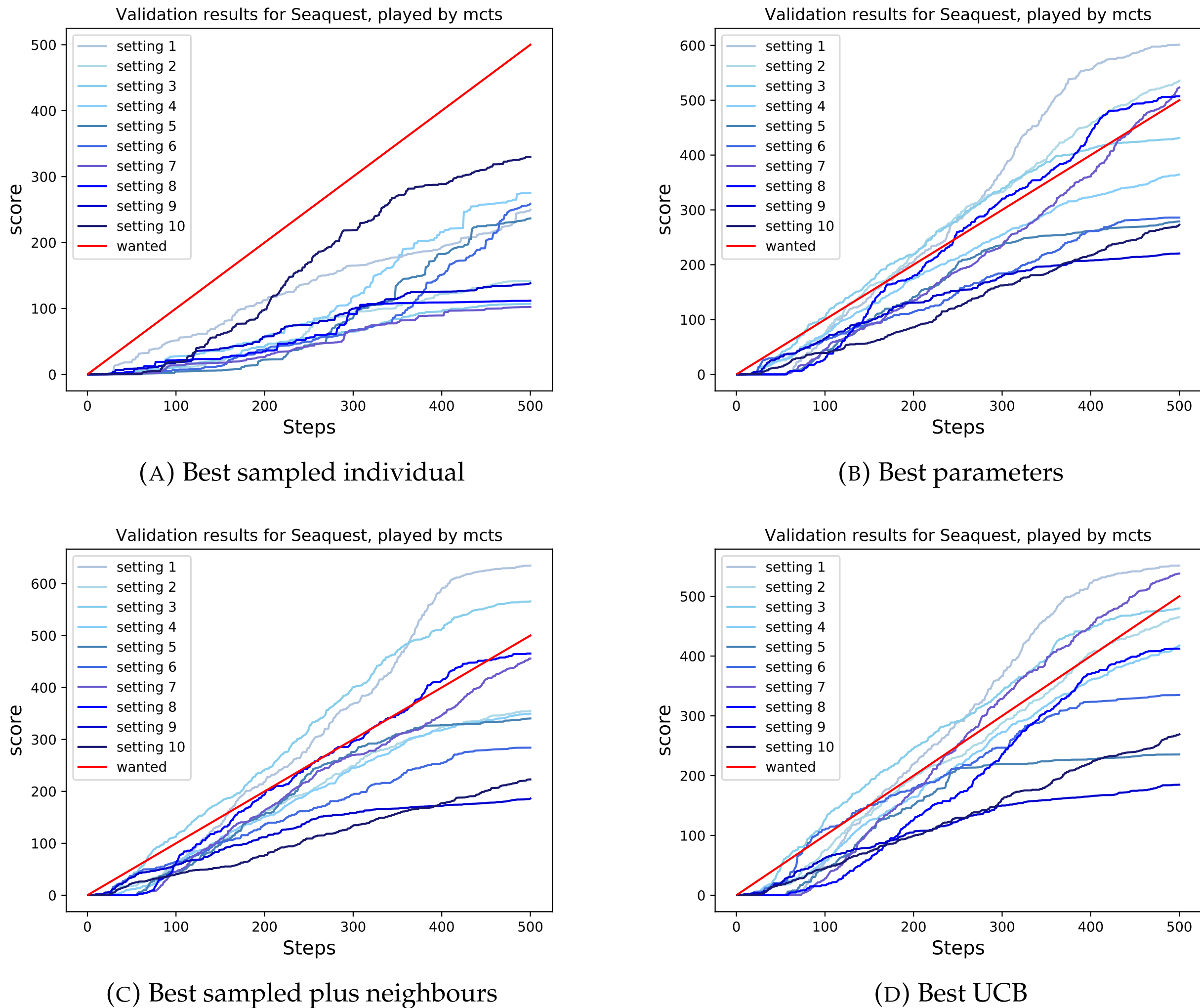
(D) Best UCB

FIGURE 5.16: Average score difference in validations for $y = 2^{\frac{x}{70}}$, different best parameter selection comparison

in Figure 5.18. We can see that all of the best parameter set score trends are similar to the target line except for the best *sampled* individual. Based on this and to keep the document concise, for the rest of this section, only the validation results from the best *UCB* would be selected to show in further comparison.

Different Games

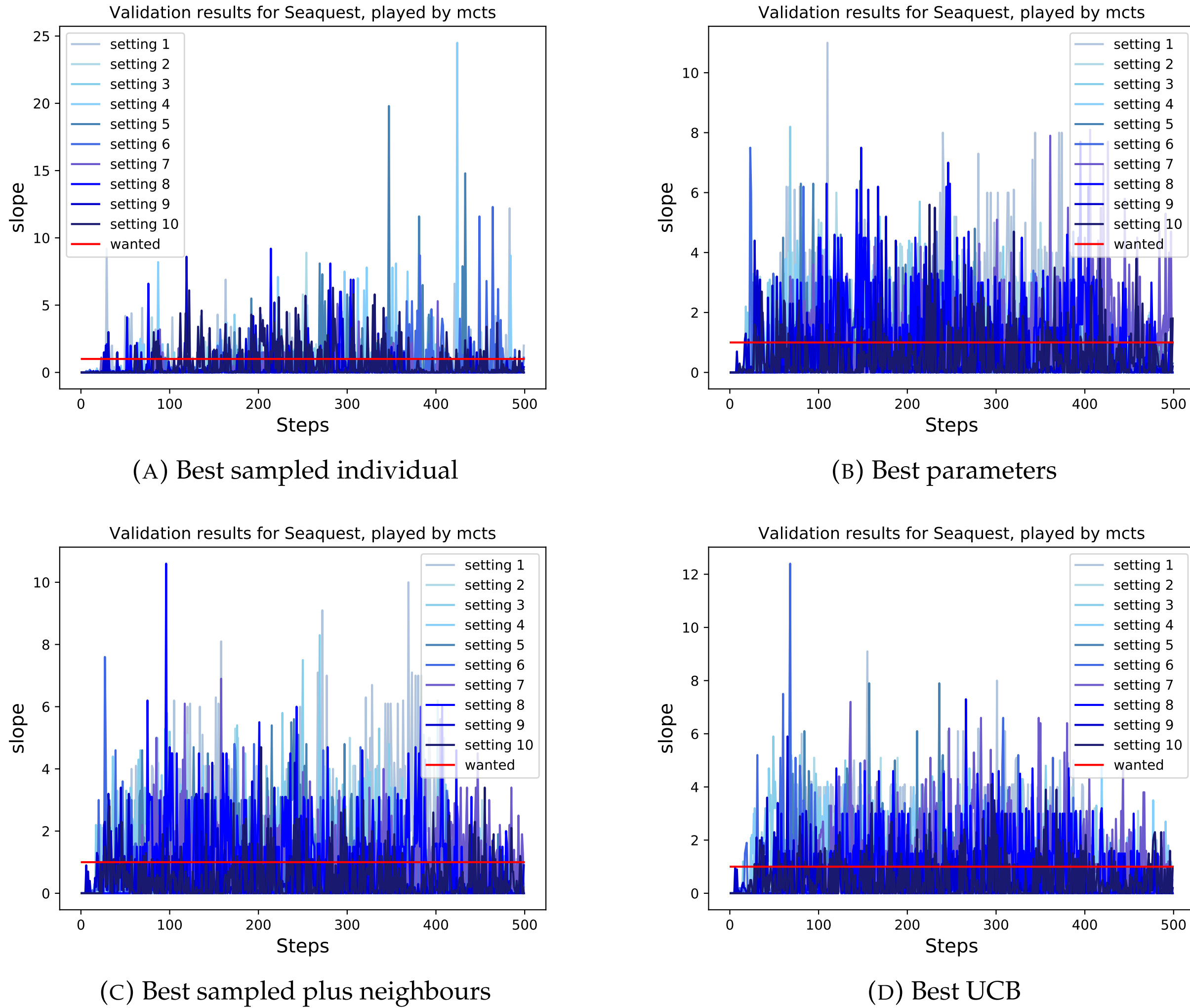
Validation results for the linear target function $y = x$ have been selected for between-game comparison, showed in Figure 5.19. Most of the parameter sets



(A) Best sampled individual
(B) Best parameters
(C) Best sampled plus neighbours
(D) Best UCB

FIGURE 5.17: Average score trend in validations for $y = x$, different best parameter selection comparison

evolved for Defender provided the environment for positive score in average, which is a huge improvement considering that most of the initial parameter set gave negative score, though neither of them able to achieve the average of $m = 1$. Waves and Seaquest evolved parameter sets gave obviously better result in score trend, as most of the lines are close to the target line. In term of score difference (slope), Waves result seems to closer to the target line in average. This is also true for other target function results, as another example for right-sigmoid is given in Figure 5.20. This agrees with the result from evolution



(A) Best sampled individual

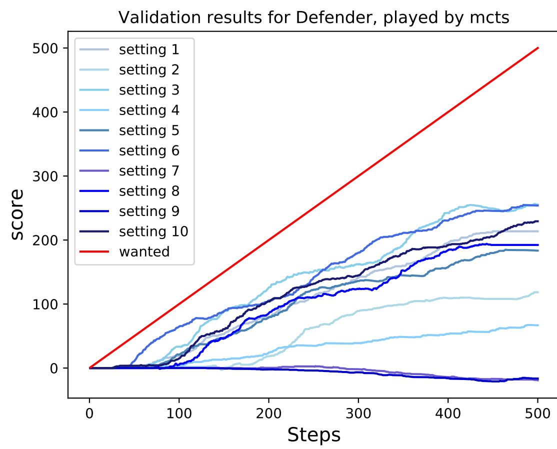
(B) Best parameters

(C) Best sampled plus neighbours

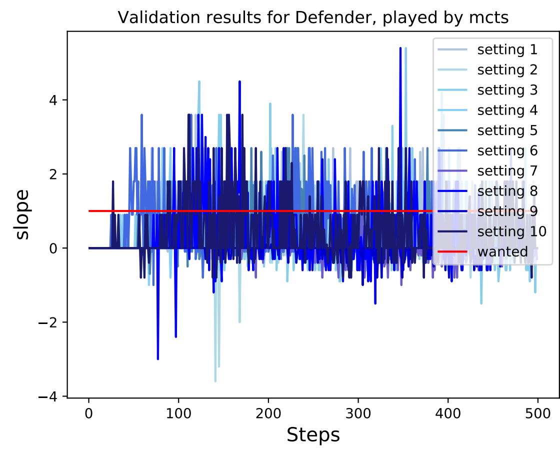
(D) Best UCB

FIGURE 5.18: Average score difference trend in validations for $y = x$, different best parameter selection comparison

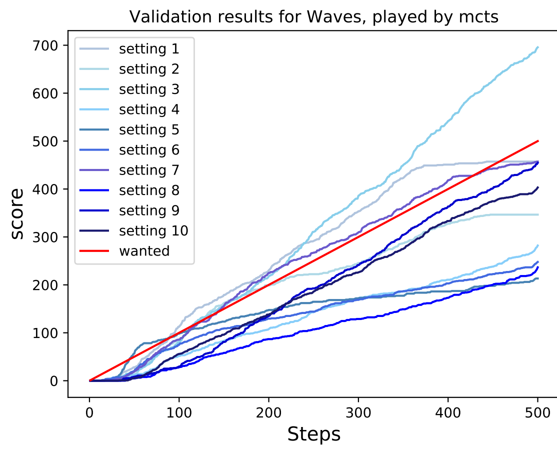
phase that Defender was the most difficult to fit due to its parameter space. Although it is still inconclusive whether Seaquest or Waves are easier to evolve it is obvious from both Figure 5.19 and Figure 5.20 that the NTBEA was capable of tuning parameters for both games, at least in for this function set, with the final outcomes of the evolution still provide the game environment that allows the player to achieve the same score trend as the target functions. This confirms the hypothesis HM2 because two different controllers have similar score trend in average for the same game.



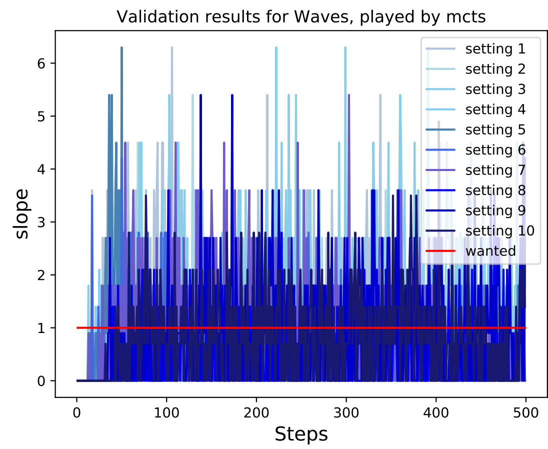
(A) Defender score



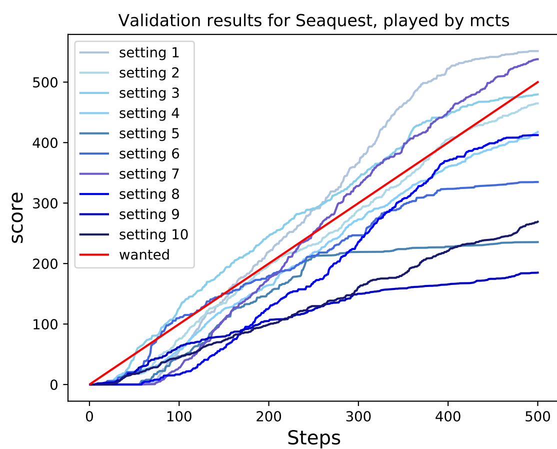
(B) Defender score slope



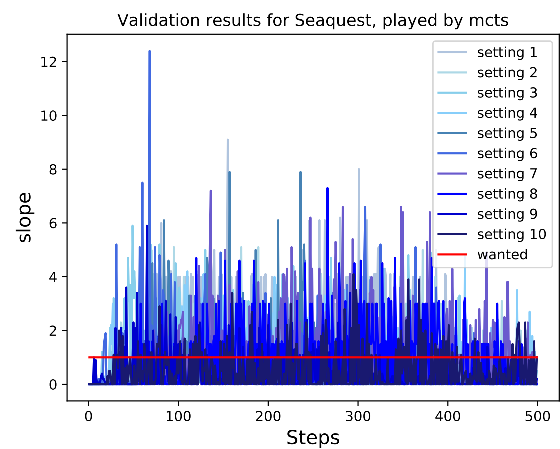
(C) Waves score



(D) Waves score slope

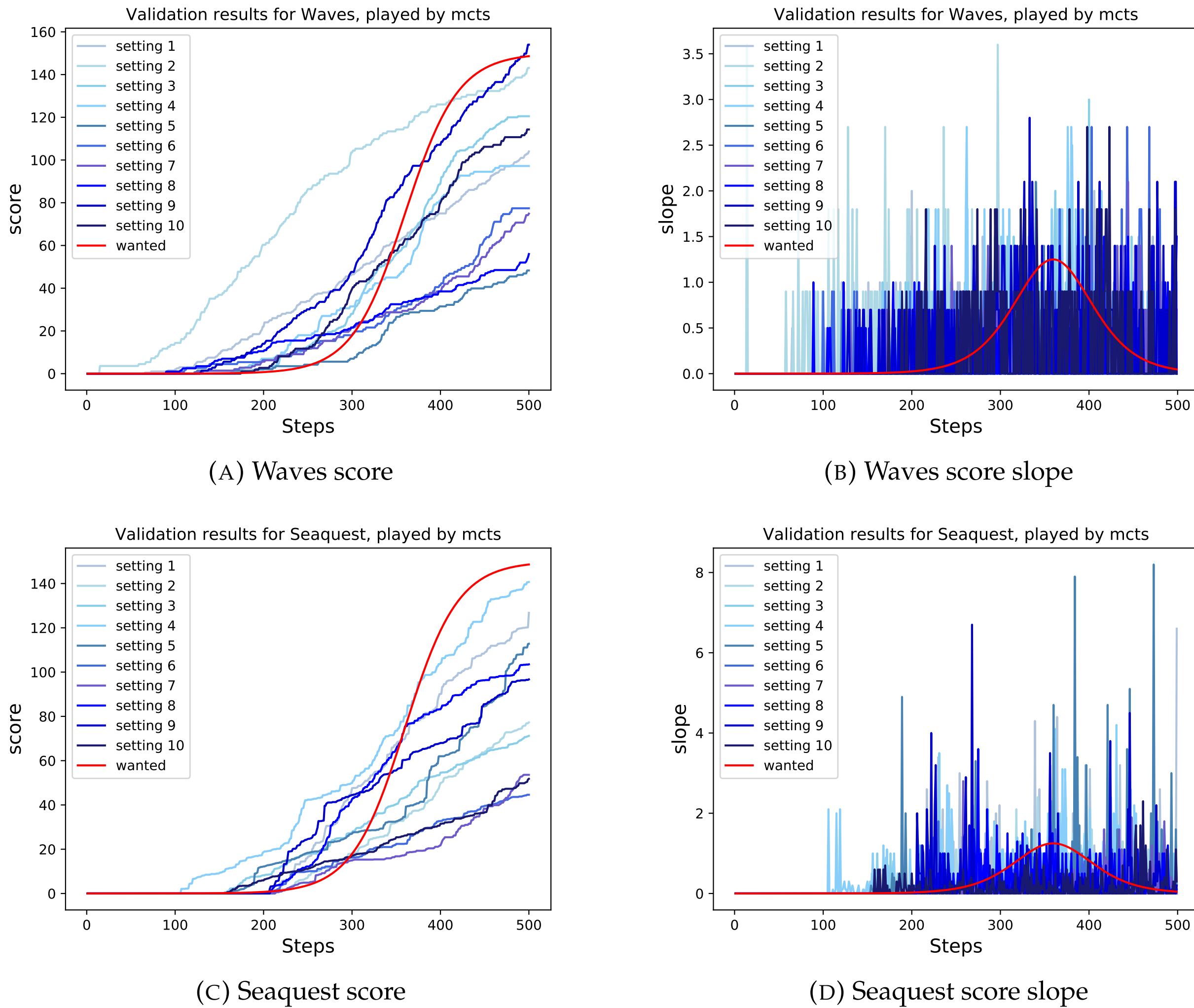


(E) Seaquest score



(F) Seaquest score slope

FIGURE 5.19: Average score trend and score difference during validation for $y = x$ best UCB, on the game of this study



(A) Waves score

(B) Waves score slope

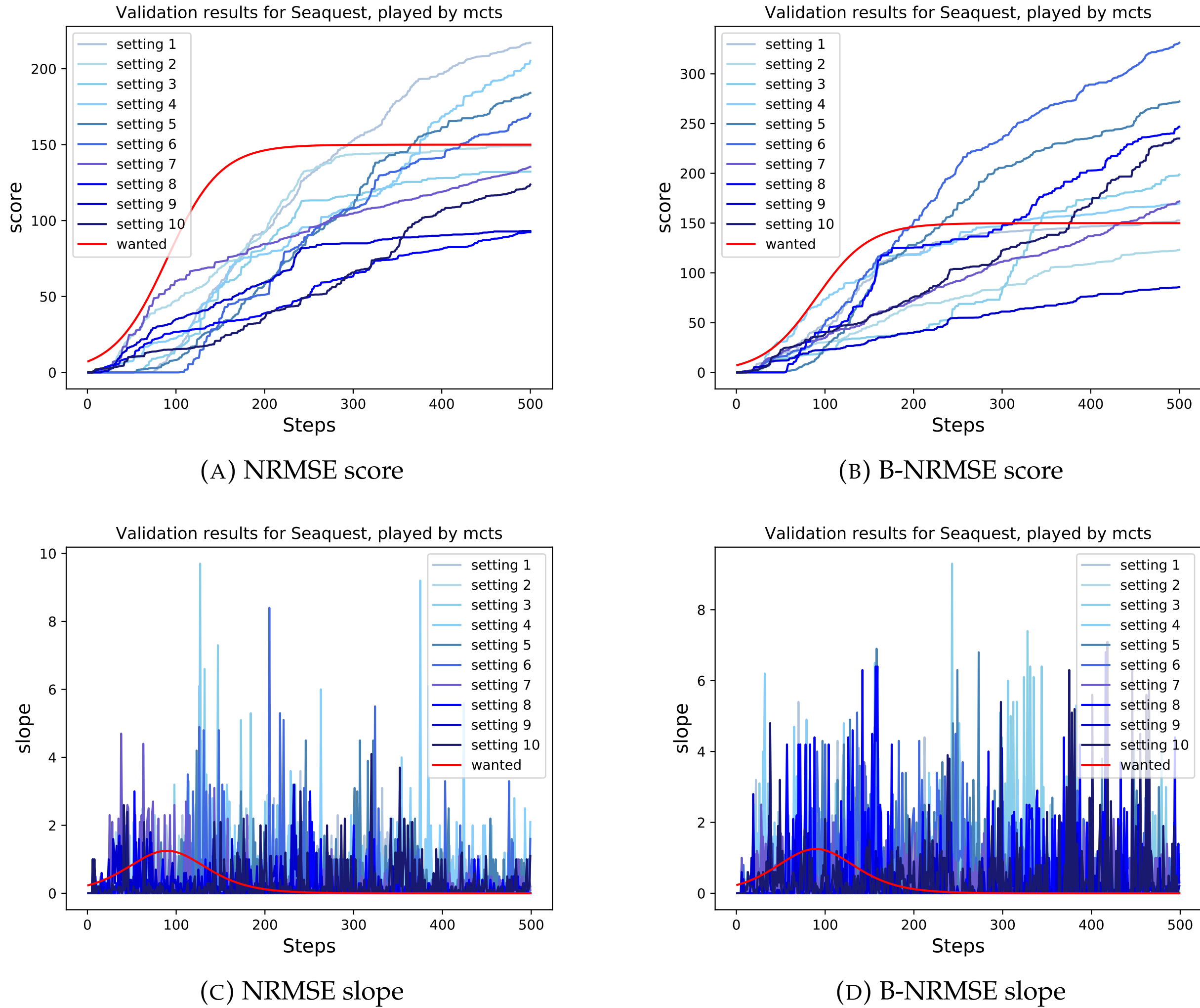
(C) Seaquest score

(D) Seaquest score slope

FIGURE 5.20: Average score trend and score difference during validation for normal NRMSE right-sigmoid best *UCB*, on the game of this study

Normal NRMSE and Biased NRMSE

In 5.2.1 I have clarified that biasing the loss calculation affected the results from fitting shifted sigmoid functions the most. However, it is not as obvious from the validation phase results. A comparison between normal NRMSE and B-NRMSE can be seen in Figure 5.21, which is showing the results from Seaquest. Both score trends (Figure 5.21a for normal NRMSE and 5.21b for B-NRMSE) are similar. This is also true in score difference trends (Figure 5.21c for NRMSE and 5.21d for B-NRMSE) that most of the blue bars in B-NRMSE figure seems to



(A) NRMSE score

(B) B-NRMSE score

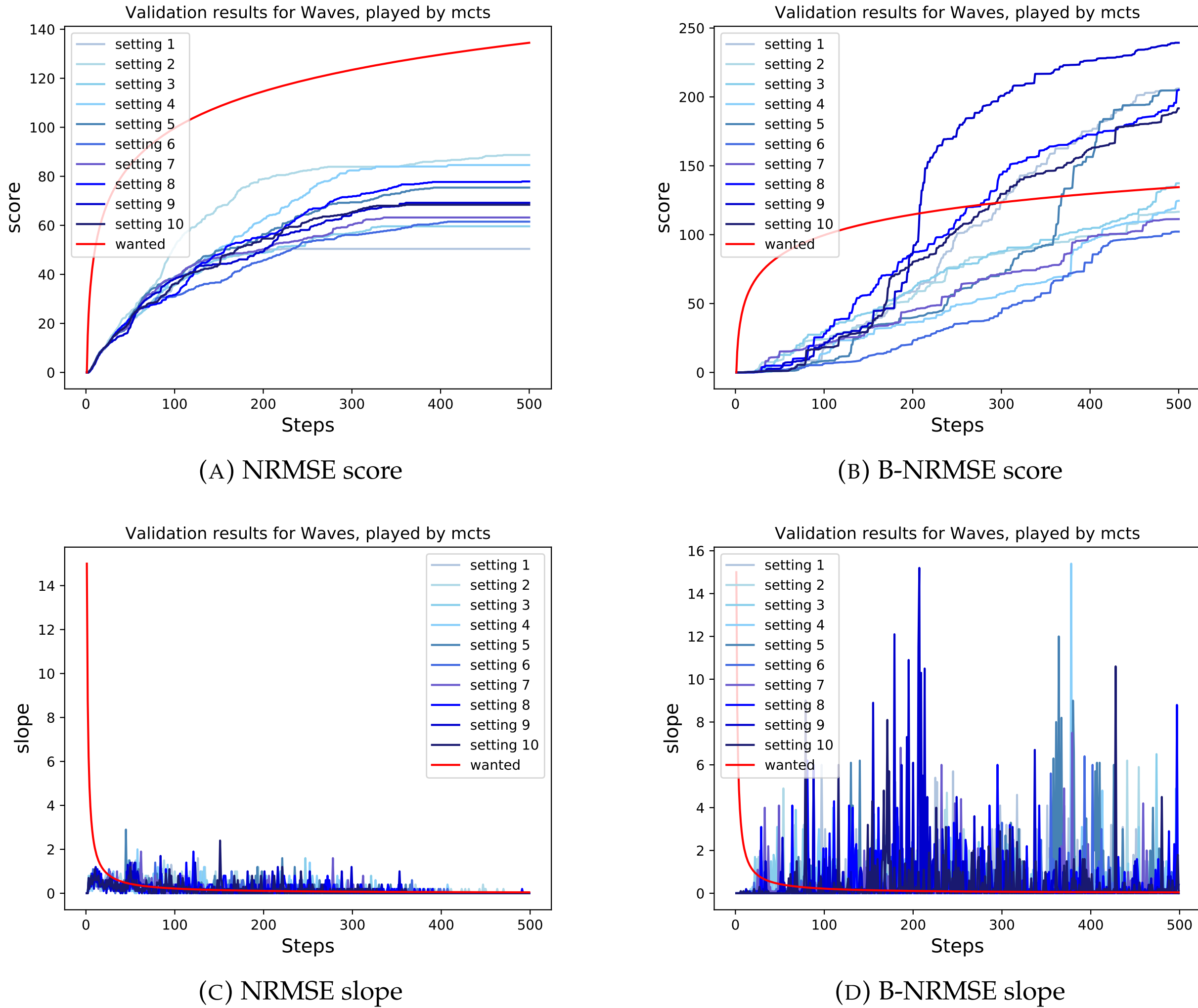
(C) NRMSE slope

(D) B-NRMSE slope

FIGURE 5.21: Average score trend and score difference during validation for Seaquest left-sigmoid best *UCB*, different loss calculation comparison

scatter away from the red curve more than from in NRMSE slope figure.

The more obvious example can be seen from logarithmic validation results of Waves, shown in Figure 5.22. It is clearly observable that all of the curves for normal NRMSE (Figure 5.22a) were restricted by the target curve, while it is untrue for the biased NRMSE (Figure 5.22b) . Score difference graphs (Figure 5.22c for normal NRMSE and 5.22d for biased NRMSE) also show that the slope trends of the ones using normal NRMSE as loss calculation are much more closer to the red curve, that is higher in the beginning and decreasing afterwards. In



(A) NRMSE score

(B) B-NRMSE score

(C) NRMSE slope

(D) B-NRMSE slope

FIGURE 5.22: Average score trend and score difference during validation for Seaquest with, $y = 15\log_2(x)$ best UCB, different loss calculation comparison

contrast, results from the Seaquest game parameters evolved by using B-NRMSE as loss calculator do not follow that trend.

Based on the observed results from the validation phase, I would conclude that biasing the loss value calculation does not improve the performance in this application, hence rejecting the hypothesis HG2. The initial assumption of this was that it would give more priority to the 'more important' areas (the areas with higher slope). However, we have found that it instead reduces the priority of the areas with lower slope values, resulting in higher standard deviation

range, because the EA was less 'restricted' to such areas from the small penalty given.

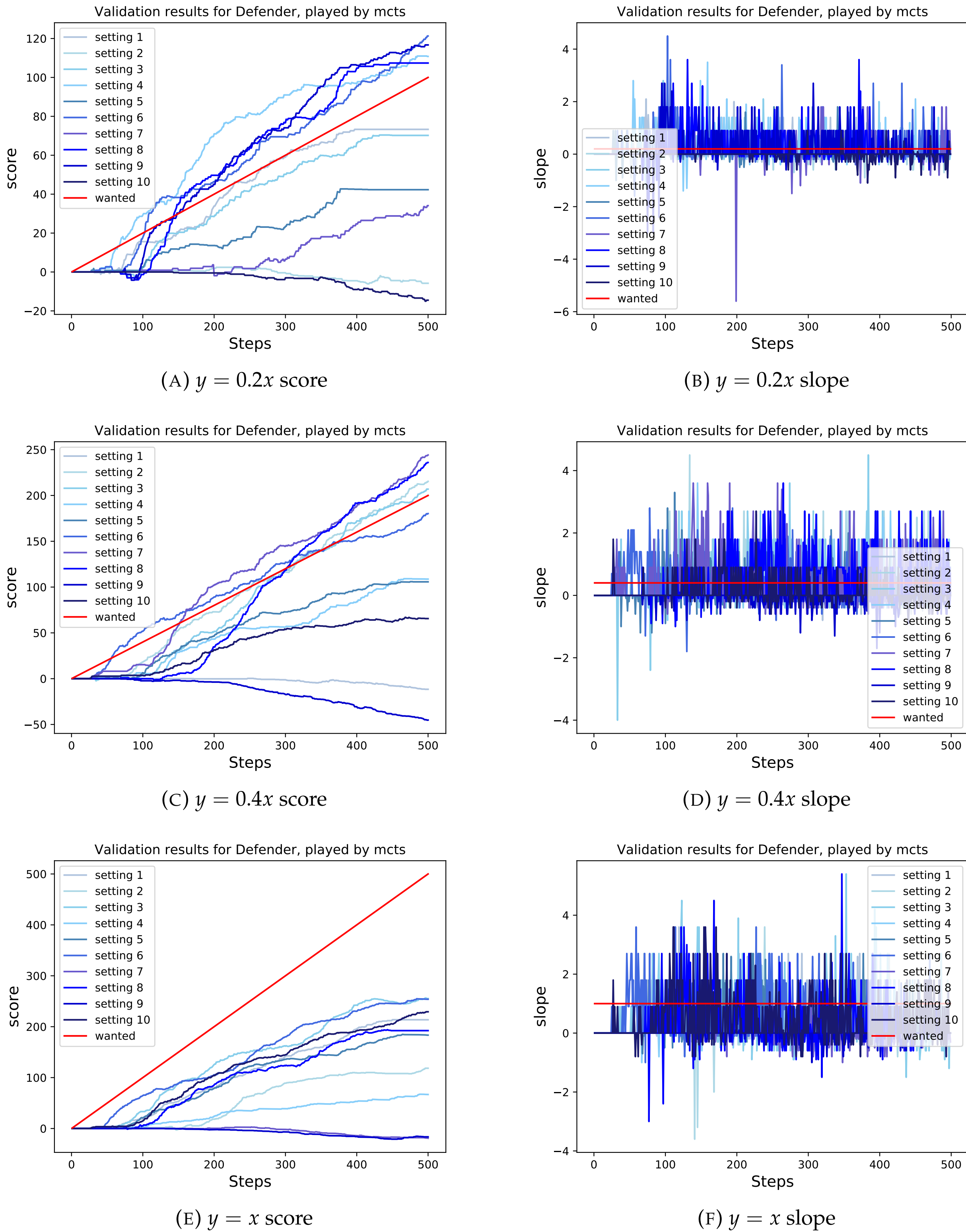
In this subsection, the results of biasing loss calculation for reporting fitness were discussed. Next, I will compare the results between the selected target functions.

Functions and Parameters

Similar to 5.2.1, the functions are divided into four groups based on their increasing behaviours. I selected the result set from only Defender with normal NRMSE as I have discussed earlier that Defender was the most difficult game to evolve, and biasing the loss calculation did not improve the performance. Keeping these two parameters fixed make it easier to compare only the aspect we want, which is the target function.

Figure 5.23 shows the score and slope trends of validation phase for linear functions $y = mx$ with $m = 0.2, 0.4$ and 1. For $m = 0.2$, most of the lines are close to the red line, with two that still have negative average final score. This indicates that the majority of game score throughout the plays follow the target trend. The same trend applies for $m = 0.4$ with fewer lines close to the red line . It can be referred from this that smaller m is easier to fit for this game parameter space. An evidence to confirm this is the result of $m = 1$ (Figure 5.23) that none of the line trends seem to be close to the red line. Moreover, based on the results in Figure 5.23e, it confirms that the $m = 1$ linear score trend is unachievable for the current Defender setting. This would further explain the unsuccessful of fitting with two-part linear piecewise functions later. Nevertheless, since the results are similar to the last generation range in evolution phase, HM1 is confirmed for this target function.

The next function pair is logarithmic and exponential. Figure 5.24 shows the results of average score and score slope for Defender, using normal NRMSE loss calculation. For logarithmic function (Figure 5.24a, all of the curves are



(A) $y = 0.2x$ score

(B) $y = 0.2x$ slope

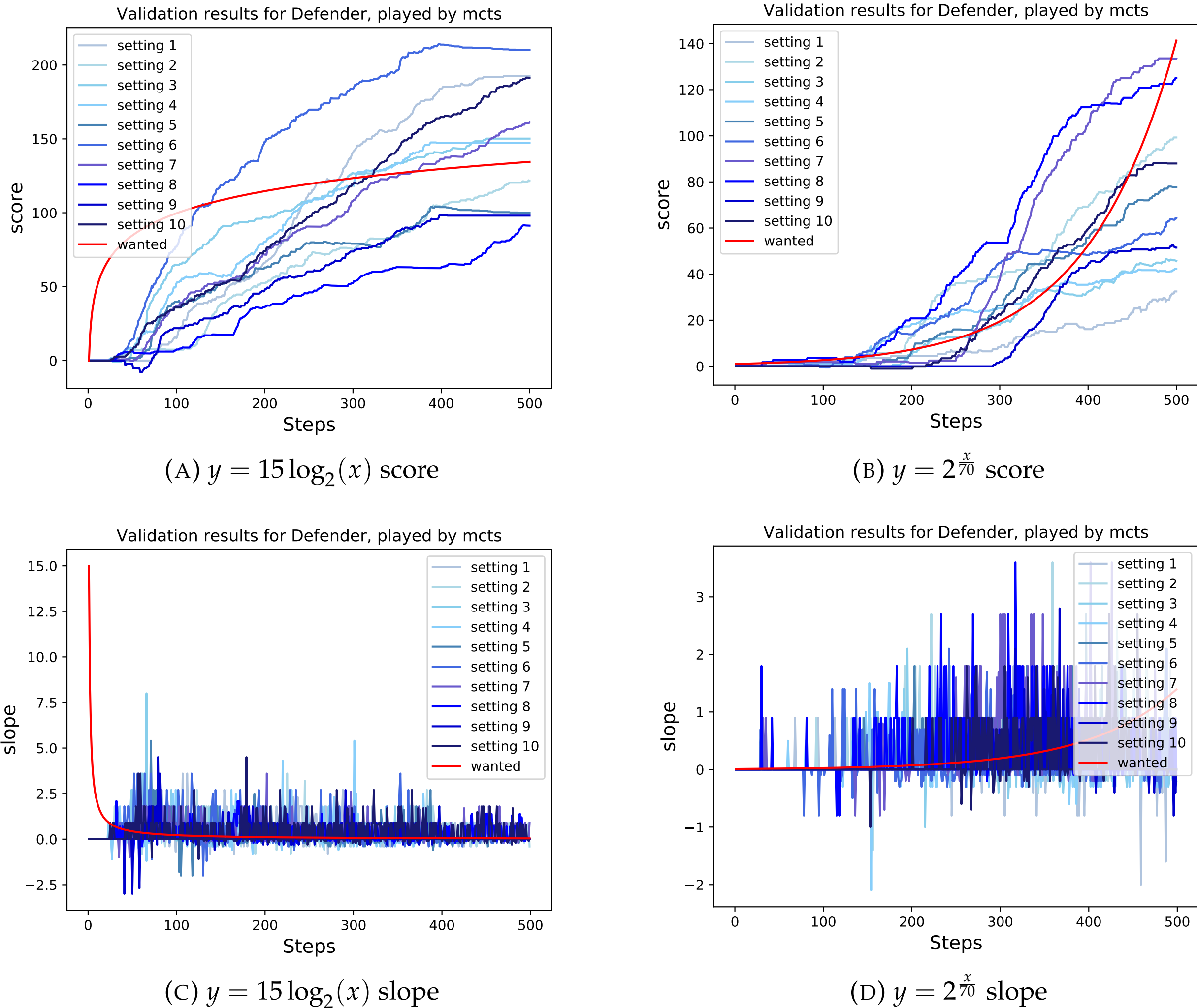
(C) $y = 0.4x$ score

(D) $y = 0.4x$ slope

(E) $y = x$ score

(F) $y = x$ slope

FIGURE 5.23: Average score trend and score difference during validation for normal NRMSE Defender, best *UCB*, linear function comparison



(A) $y = 15\log_2(x)$ score

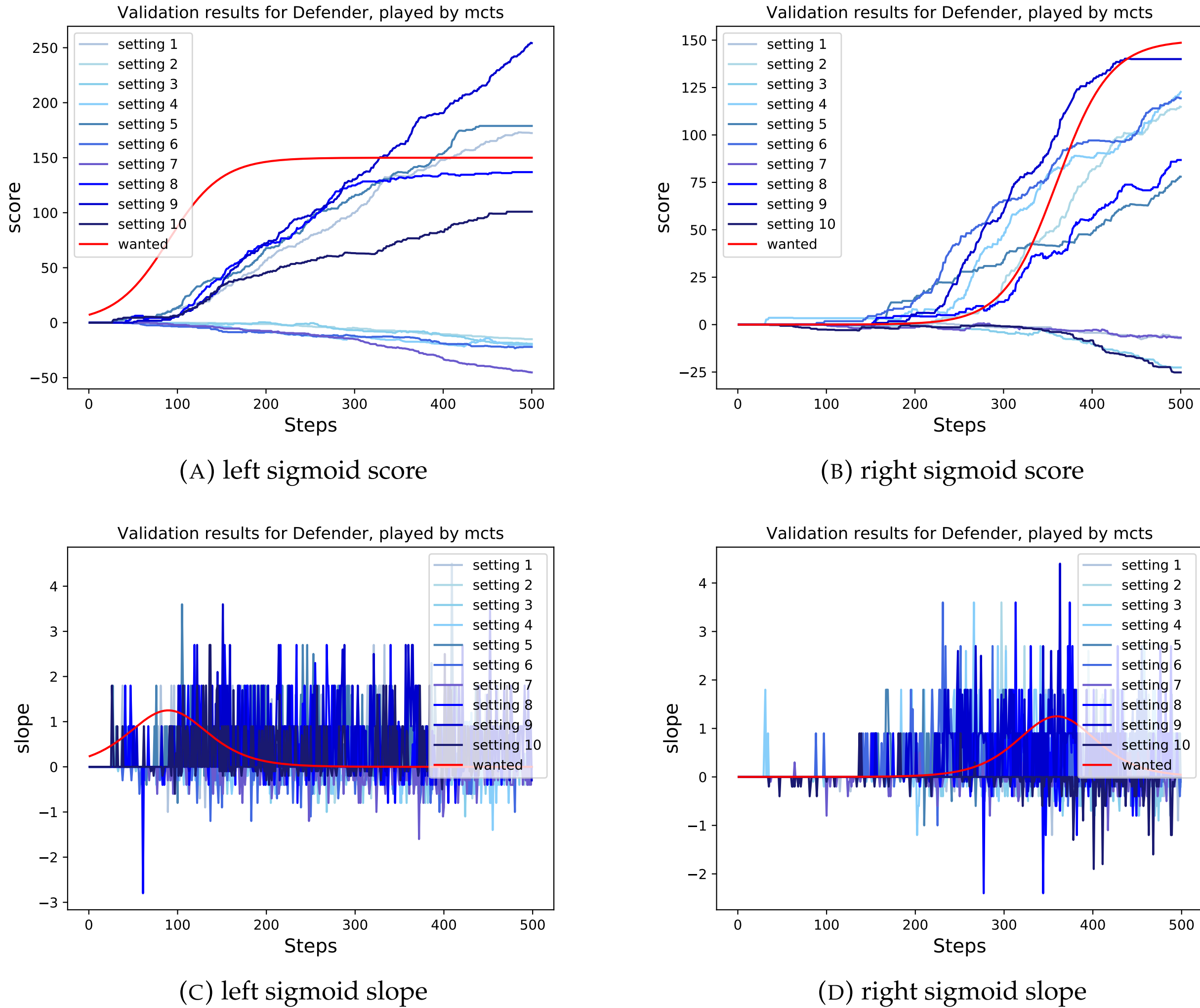
(B) $y = 2^{\frac{x}{70}}$ score

(C) $y = 15\log_2(x)$ slope

(D) $y = 2^{\frac{x}{70}}$ slope

FIGURE 5.24: Average score trend and score difference during validations for normal NRMSE Defender, logarithm and exponential function comparison

completely disjoint until step 100. It is worth noticing that the target values of the few early steps are hardly achievable, as $15\log_2(x) = 15$ when $x = 2$ and almost 35 when $x = 5$. With the decreasing of the increasing rate of the function it is more practical later on. Our initial objectives of selecting this were first to experiment if the EA would be able to find a parameter set that allows the player to score more in the beginning and less afterwards. It is difficult to visualize if we have achieved this as the slope (Figure 5.24c values are more constant-like, although plotting them without some first few steps might clarify



(A) left sigmoid score

(B) right sigmoid score

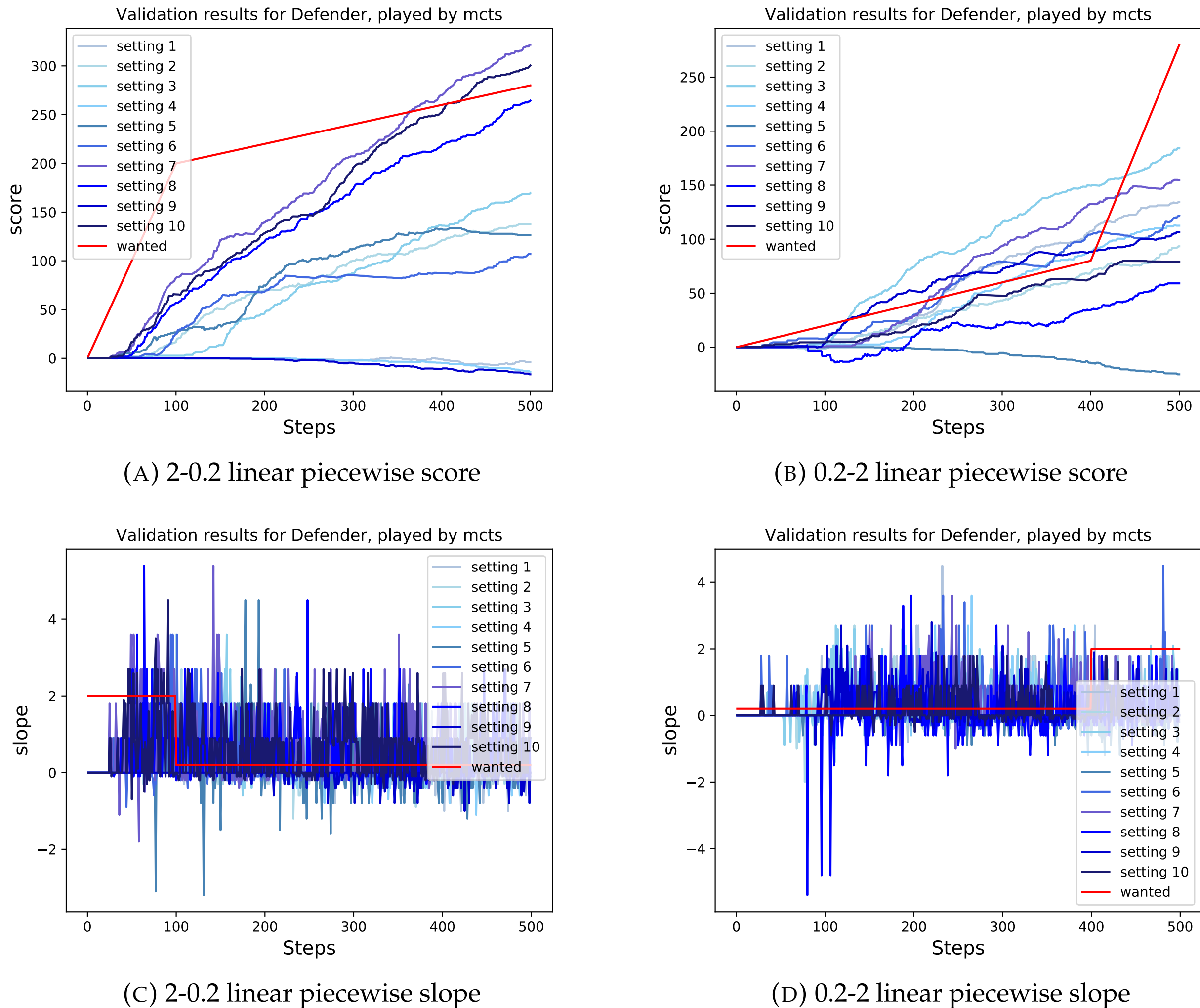
(C) left sigmoid slope

(D) right sigmoid slope

FIGURE 5.25: Average score trend and score difference during validation for normal NRMSE Defender best *UCB*, shifted sigmoid function comparison

this. Nevertheless, observing the raw score results alone can partially confirm HG1 for this pair of functions.

The next function pair is left and right shifted sigmoid, shown in Figure 5.25. We expected that the games evolved to fit the left sigmoid function would provide more score in the beginning and then stop later on. However, based on Figure 5.25a and Figure 5.25c, the score seems increasing in more linearly trend for this game after step 100. The results from fitting right sigmoid functions (Figure 5.25b and Figure 5.25d) look more similar to the target curve. The EA is more



(A) 2-0.2 linear piecewise score

(B) 0.2-2 linear piecewise score

(C) 2-0.2 linear piecewise slope

(D) 0.2-2 linear piecewise slope

FIGURE 5.26: Average score trend and score difference during validations for normal NRMSE Defender, 2-part linear piecewise function comparison

successful in evolving games to fit right shifted sigmoid than left sigmoid. It is possible that it is more challenging to stop providing the score, than to suppress it in the beginning and start giving them out in the later steps.

The last target score pair to compare are two-part linear piecewise functions. The score and slope trend of Defender, using normal NRMSE loss calculation, are given in Figure 5.26. The EA did not respond to the 'shift' point (step 100 for $m = 2 \rightarrow 0.2$ and step 400 for $m = 0.2 \rightarrow 2$) and instead MCTS score trends are linear. This could be caused by several reasons. The first and most probable one

is that the value $m = 2$ is too high, as briefly mentioned earlier in linear function comparison. The EA struggled to fit $y = x$ linear function, therefore it is unlikely to fit $y = 2x$. Further experiments with smaller m would confirm this. Another possible reason is that the 'shift' point might be too close to the beginning and the end of the game, with only 100 steps to react. Modifying this parameter to move the shift points more to the middle of the game might clarify if this is the cause.

5.2.3 Evolved Games

We observed the final parameters evolved by NTBEA in Defender and compared them within the same group of target functions. For linear functions, we compared $m = 0.2$ and 1 and found that the game with $m = 1$ tended to have higher supply limit, with slower bomb speed and alien spawn probability. The portal in $m = 1$ also tended to close faster, probably to prevent more aliens dropping bombs to the city. In right shifted sigmoid final parameter set, supply speed, alien speed and alien spawning probability are higher, while the portal opened later and closed earlier compared with the left-shift version. In logarithmic/exponential comparison, the portal opened and closed later in exponential version, with slower supply speed and faster alien. All games evolved maximized the reward given when an alien is shot while minimized the penalty of a city being bombarded.

Examples of the evolved games in Defender can be found in an online video[1]. This video shows one game from each analyzed set (linear with $m = 2$ and 1, left/right shifted sigmoids and logarithm/exponential). In the videos, it is noticeable the differences in supply speed/spawning rate and numbers of enemies spawned, as well as the lack of enemies at the beginning of the exponential-function-evolved game.

[1] https://youtu.be/GADQLe2TiqI

In this section, the final evolved parameter sets of each setting from the evolution phase were selected and applied to design games. These games were then played by MCTS for validation and the score were recorded and plotted, along with their slope trend. Some of these graphs were selected to compare and analyze in three aspects: between-game comparison, bias and normal normalized root mean square error loss calculation comparison, and between target function comparison. Similar with the evolution phase results, Defender game space was the most difficult to evolve, followed by Waves, and Seaquest respectively. Biasing NRMSE did not provide the better games compared to normal NRMSE. For function and parameters, finding games to fit the functions with higher increasing rates are harder, and the functions which increasing faster in the beginning were more difficult to fit than those increasing later.

In this chapter, the research experiment procedures, parameters and some results for both experiments taken are reported and analyzed in details. In the next chapter, we give the summary and conclusion of this dissertation and discuss the findings of the research taken.

Chapter 6

Summary & Conclusion

The research done in this dissertation falls into the field of Automatic Game Design [1], which involves the usage of Artificial Intelligence techniques to automatically solve game design tasks. We focus on only one task here which is parameter tuning. The scope of this work covers only *Space Battle* and video games from the General Video Game AI framework [10], with three of its games were manually selected for the experiment. We used an evolutionary algorithm called *N-Tuple Bandit Evolutionary Algorithm (NTBEA)* to evolve the selected games' parameters following evolutionary approach. In the first experiment, we aimed to explore the performance of NTBEA and found that it outperformed the well-known RMHC with less standard deviation when applied to a noisy environment. In the second experiment, we aimed to investigate if the NTBEA would be capable of evolving game parameters that provide players with score-hunger behaviour to obtain score in the same trends as pre-defined functions. In this chapter, all of the previous chapter contents, excluding the introduction, are summarized in the *Summary* section and the conclusion of the research done is given in the *Conclusion* section.

6.1 Summary

In Chapter 2, a number of related works have been reviewed to provide the readers with the earlier contributions of the relevant fields and techniques. The

works are categorized into *Automatic Game Design*, *General Video Game AI (GVGAI) framework* and *Evolutionary Algorithms*.

1. In Automatic Game Design, reviewed published literatures are further divided into 4 sub-categories. The first are the early proof-of-concept works, which the authors aimed to investigate the possibility of AI-assisted Game Design. The latter three are more about the tasks that AI techniques can be applied to assist, with the first two are related to auto-generation of game components, which is also known as procedural content generation. They are either focusing on only level/map generations or proposing a complete game generation (level/map and game rules). Last but not least, the research works related to automatic tuning of game parameters from the fixed game space have been explored and discussed, as that is the main task of our research.

2. As one of the domains we selected to do auto-parameterization is GVGAI, related research of the framework are reviewed in 2.2. First, the previous works leading to the origin of the framework are outlined to provide readers with the motivation. Next, the the competition tracks along with their technical detail published works are summarized, with our research would be most related to the game design track. Finally, a few publications that related to game design for GVGAI have been studied and discussed. To the best of my knowledge, those are all of the works in such field until the date that chapter was written.

3. An evolutionary approach was employed to do automatic parameter tuning in our experiments. Therefore evolutionary algorithm literatures were reviewed next. Since the field is popular and the EA nature is widely applicable to various domains, I selected only some important works to give outline of general improvements throughout the years first. Then the

applications of EAs in game domain were explored. Most of the game-designed literatures with evolutionary algorithms would be already mentioned in the game design related work section (2.1), therefore they were not re-mentioned again in 2.3 to avoid unnecessary content duplication.

In Chapter 3, essential background knowledge is provided to the readers with the intention that it would assist them to fully understand the research that this dissertation is presenting. The first necessary knowledge topic involves *Space Battle* game and its variant called *Space Battle Evolved* that was used in this research, then *General Video Game AI (GVGAI) Framework*. The readers have been given a detailed guidance into how the framework is working, along with a comprehensive interpretation of *Video Game Description Language (VGDL)* that is being used as the description language of this framework. The GVGAI implementation details of the selected three games: *Defender*, *Seaquest* and *Waves*, were described, along with the human-language interpretation of each description. Next, the algorithm behind all general game playing agents applied were explained, along with their implementation into GVGAI framework. Another necessary background knowledge that had been described in Chapter 3 is about the selected EA called *N-Tuple Bandit Evolutionary Algorithm (NTBEA)*, and another state-of-the-art EA called *Random Mutation Hill Climber (RMHC)* and its variant designed for this research. NTBEA is an EA for noisy optimization, first proposed in [11], and first applied to Game AI by Prof. Simon Mark Lucas. In that paper only a brief summary of how the algorithm operates was given, while a more detailed explanation was revealed here in 3.3.3.

Chapter 4 clarified the settings and problem definition, in preparation for the experiments. This includes *Space Battle Evolved* game parameter space and the fitness calculation procedures employed in the first experiment. Next, we described game space for the three GVGAI games, and the fitness function calculation of the second experiment. The game space for all three games, defined

in VGDL, were illustrated and all parameters to be tuned by the EA were described. Each game has its defined parameter space in the order of 10^{10}, means there are 10 trillions combinations of possible parameters. Therefore it is obvious that brute-force strategy is infeasible, and some domain specific knowledge is necessary to select an appropriate set. However, as we aimed to propose a general approach, and as the optimization task we are solving is not trivial (i.e. it is not obviously seen or easily deducted how the parameter set should look like), we have decided not to include any game-related knowledge into parameter tuning. Instead, we had introduced new rules for all games, which involve setting the start and end time step that the enemies are allowed to spawn, and included these parameters into the game space. Theoretically, adjusting these parameters would directly affect the score received, as well as its trend, hence there should be at least one suitable setting to fit each of our target function. We were expecting that the EA should be able to notice these parameters and evolve them accordingly.

Chapter 5 presented all aspects of the experiment taken. It started by giving an overview of all different settings along with the control variables and fixed parameters applied. Two experiments, for *Space Battle Evolved* and *GVGAI* are discussed. For Space Battle Evolved, the game were evolved by three evolutionary algorithms: RMHC, biased-RMHC and NTBEA. Based on the results, it can be concluded that NTBEA was more robust than the others and better fulfilled the objective we would like. For GVGAI games, the experiment consisted of two phases: evolution and validation. In evolution phase, the NTBEA were employed to evolve parameter sets for each setting (a specific game, with a specific target score function trend, using a specific loss calculation method), all played by RHEA. Fitness values of each game parameter set (individual) generation were recorded and plotted to observe its growing trend. Step-by-step score of every gameplays for each setting were also recorded and compared for each generation range. The difference of score between all adjacent time steps

(that were usually mentioned as 'slope' or just 'score difference') were also computed from the recorded score and plotted. The evolution phase results were analyzed and compared between three aspects: different games comparison, biasing and non-biasing loss calculation comparison, and comparing between the results from within-group target function. Validation phase involves playing the evolved games with another AI controller, in this case MCTS, and see if the score obtained are the same, or similar, to the target functions. Again, score in every steps were recorded and plotted in comparison with the target function. As the NTBEA can provide more than one best solutions, based on the selection criteria, the results of these criteria were compared and only the best between these were shown in later comparisons. Similar to the evolution phase, the results were analyzed by comparing between-game, between-target functions and between bias & non-bias loss calculation.

6.2 Conclusion

In this section, I give the conclusion from all main and minor hypotheses formed in chapter 1 and chapter 5. There are some inconclusive hypotheses in general cases, but the experiment results in our domains give positive outcomes. These would be labelled as *Inconclusive (+)*. The conclusion for all hypotheses of the first experiment are:

- **HM1: NTBEA is more robust to noise in noisy environments than standard hill-climbing evolutionary algorithms: *Correct***
  The conclusion of this is drawn from the results in 5.1. Figure 5.1 shows that the fitness values for the evolved games from NTBEA had significantly less standard deviation than both versions of RMHC used in the Space Battle Evolved experiment. This means that NTBEA is less susceptible to noise.

- **HM3: General Video Game Playing controllers can be used as substitutions for human players in automatic game parameterization.: *Inconclusive (+)***

  We have applied two general video game controllers (*One Step Look Ahead (1SLA)* as amateur player and *Monte Carlo Tree Search* (MCTS) as skilful player) to do playtests instead of human in parameter evolution task. The positive results from Figure 5.1 and human feedbacks indicate that the approach employed is practical and able to produce good results, although more testers are needed to validate the hypothesis.

- **HS1: The games evolved by NTBEA satisfy human preferences more compared to both RMHC-based algorithms: *Inconclusive (+)***

  Two human players have tested some of the evolved games and provided valuable reviews. Both players preferred the new game evolved using the NTBEA, while they offered mixed opinions on the RMHC games. More testers are needed to validate this hypothesis in general cases.

Next, our conclusions on each hypothesis for GVGAI experiment are:

- **HM2: NTBEA can be applied to tune game parameters to provide specific pre-defined player score trend, for any players playing the game: *Correct***

  The results in the validation phase (5.2.2) are similar to those in the last generation ranges of the same setting in the evolution phase (5.2.1), despite the fact that they were playing by different controllers. This is adequate to conclude that the games evolved using our approach give the same player experience in score trend aspect.

- **HM3: General Video Game Playing controllers can be used as substitutions for human players in automatic game parameterization: *Inconclusive (+)***

  We used RHEA during evolution phase and MCTS during validation phase.

From our observations of some final games, the EA was able to find some sensible parameter values. For example, in the Defender with left-sigmoid target function, the EA adjusted the portal open/close time so it opens and closes earlier than right-sigmoid. This is reasonable since the left-sigmoid function has higher increasing rate in the beginning. It is fair to assume that the EA adjusted this based on the behaviour of RHEA, which is reliably score hunger, and MCTS also produced the similar score trend in validation phase. Therefore, it can be concluded that using general video game agents as substitutions for human players is practical and able to produce good results at least in our domain.

- **HG1: NTBEA can evolve game parameters to fit the same target functions for any games: *Inconclusive (+)***
  As seen from the results in 5.2.1, the average score trends when using the same target function are similar, regardless of the games. This means that our fitness calculation is general enough to apply with the three games tested.

- **HG2: Biasing the loss calculation helps NTBEA in our optimization task: *Incorrect***
  The results from 5.2.1 point out that there are both positive and negative effects of biasing the loss calculation. The advantage is that the EA was more sensitive to the changes and reacted more to them, while the disadvantage is that it neglected the errors in the area that increment rate = 0. However, based on the results from validation phase (5.2.2), B-NRMSE did not have any advantages over NRMSE. Therefore it is sufficient to conclude that biasing the loss calculation does not improve performance of the EA in this optimization task.

Apart from the hypotheses, there are a number of observations noticed in different aspects from the results.

- **Defender is the most challenging game to fit in all target functions**
  It had been formally analyzed earlier that Defender would be difficult to fit with positive increasing functions because of the initial game rules. Most of the games played usually ended up having low final score values, often in negative. Waves and Seaquest did not suffer much as Waves does not penalizes the players via negative score signal as much as Defender and Seaquest does not do it at all apart from when the players have lost.

- **The EA are capable of reacting differently with different target functions**
  We expected that NTBEA should have the ability to react to the different target trends, only via the fitness values given alone. As seen from various examples in 5.2.1, the trend curves of the early generations of each comparison pair/group were similar, but had become more distinctive in the later generations. The validation results in 5.2.2 also confirm this, with most of the comparing examples are obviously dissimilar. This is the evidence that our approach is applicable with varieties of score functions.

- **Later increasing functions are easier to fit**
  Although this is reasonable and should be foreseeable with a careful analysis, we did not realize it until the results were observed. With the game space defined, we expected that the EA would be able to find the specific parameters that directly involve ’opening’ and ’closing’ of enemy portals and tuned them based on only the fitness values, that were provided according to the loss values from the target functions. The functions that increasing later (higher slope area after step 250) need only a single crucial parameter to be tuned, which is the ’opening’. On the other hand, early-growing functions (higher slope area before step 250), require two parameters, the time step for ’opening’ and ’closing’ portals. Based on this, it is sensible why fitting the later-increasing functions are easier.

- **NTBEA reacts similarly to all non-piecewise linear target functions, but showing signs of struggling more with higher m**
- **NTBEA reacts differently to logarithmic and exponential target functions, and able to produce alike score-difference trends with the target function**
- **NTBEA reacts differently to left and right shifted-sigmoid functions, and able to produce alike score-difference trends with the target functions**
- **The EA failed to react with the sudden shifted of the increasing rate of the target functions and could not respond as expected**
  The results from fitting double linear piecewise functions were unsatisfactory, with all of them have shown the behaviour of normal linear functions. Since the value of m for these were defined before the results from single linear functions were observed, it is highly probable that our pre-set double linear piecewise functions were too steep.

In this chapter, I have summarized the contents of all previous sections and concluded the findings deducted from the experimental results. Although the NTBEA has shown a promising ability in evolving game parameters for our both tasks, there are still weak points and improvable aspects spotted. Possible future works are described in the next chapter.

Chapter 7

Future Works

The research done in this project has shown that NTBEA is robust to noisy environment and able to evolve games that provide specific player experience we defined. The first task is evolving *Space Battle Evolved* parameter so it could distinguish players from skill-depth. For the second task, we selected 3 GVGAI games and used NTBEA to evolve parameters that provide specific pre-defined game score trend. There are certain numbers of possible future works visible. Some of them could be performed immediately by analyzing the existing results in other viewpoints, while some others require further experimentations, and extra investigations and preparations are necessary for the rest.

7.1 Further Analysis

There are a couple of possible analysis that could be done with our current results. The first one would be to perform another between-function comparison, but grouped them by whether they are 'early-increasing' or 'later-increasing'. The reason behind this is that it might be interesting to rank the EA performance between these same-increment area functions.

Another interesting comparison to do is evolution-validation results comparison. Specifically, to compare the evolution results from the last few generations and the validation results of the same setting. This might give a clearer picture that MCTS score trends in validation phase are similar to RHEA in the

last few generations. In Chapter 5 they were only compared with the target functions, which some of them are difficult to fit.

Last but not least, the evolved games should have been visualized and played manually to analyze if they do have the characteristics we expected. It is probable that the games evolved to fit the same target functions might share some certain features (e.g. the time step that the enemies are spawned), but it is also possible that EA might find a complete different ways to fit the same function for different games.

7.2 Further Experiment

This section discusses the further experiments that could be done after some trivial modifications. The first one would be to re-run the same experiments with higher number of same-setting samples. The results in Chapter 5 are from 10 distinct evolutions for each setting, similar with the validations that each final games were played 10 times each. Then the outcomes of these were averaged and reported. Increasing the runs would probably reduce the standard deviation and strengthen the conclusion raised in Chapter 6.

Some further experiments could be done with slightly different sets of target functions. As mentioned a couple of times in both Chapter 4 and 5, double linear piecewise functions should be modified. The first adjustment would be to make it less steep, by reducing the higher m value. Another possibly beneficial alteration would be to move the shift point more towards the middle step (250) to give the EA more information.

Experimenting with other target functions, such as middle-shift-sigmoid or some negative functions might also be interesting. Although Seaquest would less likely to achieve that and the EA might end up evolving a very difficult game to minimize the score trend.

Another possible experiment is to evolve the game using different behaviour GVGP agents, as the two we used here are both score-hunger.

7.3 Extensions and Improvements

One main extension that we would like to do is to extend this to other games in GVGAI framework, to validate the generality of the approach. This would, however, require designing the game spaces, which at the moment have to be done manually and carefully for each game.

Another possible improvement that would require further studies and investigations is NTBEA parameter adjustment. There are a few fixed parameters such as the number of neighbour individuals. Tweaking these internal parameters appropriately might significantly improve the EA performance.

Lastly, the fitness calculation is another aspect that could be improved. We used normalized RMSE as it is widely applied in regression analysis loss calculation. It is possible that there is a more suitable fitness calculation method that gives more information to the EA than this.

References

[1] J. Togelius and J. Schmidhuber, "An Experiment in Automatic Game Design", in *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, IEEE, 2008, pp. 111–118.

[2] C. Browne and F. Maire, "Evolutionary Game Design", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 1–16, 2010.

[3] D. Ashlock, "Automatic Generation of Game Elements via Evolution", in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, IEEE, 2010, pp. 289–296.

[4] N. Sorenson and P. Pasquier, "Towards a Generic Framework for Automated Video Game Level Creation", *Applications of Evolutionary Computation*, pp. 131–140, 2010.

[5] M. J. Nelson and M. Mateas, "Towards Automated Game Design", in *Congress of the Italian Association for Artificial Intelligence*, Springer, 2007, pp. 626–637.

[6] M. Cook and S. Colton, "Multi-faceted Evolution of Simple Arcade Games", in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, IEEE, 2011, pp. 289–296.

[7] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'neill, "Evolving Levels for Super Mario Bros using Grammatical Evolution", in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, IEEE, 2012, pp. 304–311.

[8] D. Perez, J. Togelius, S. Samothrakis, P. Rohlfshagen, and S. M. Lucas, "Automated Map Generation for the Physical Traveling Salesman Problem", *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 5, pp. 708–720, 2014.

[9] A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen, "Discovering Unique Game Variants", in *Computational Creativity and Games Workshop at the 2015 International Conference on Computational Creativity*, 2015.

[10] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 229–243, 2016.

[11] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas, "The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement", in *IEEE Proceedings of the Congress on Evolutionary Computation (CEC)*, 2017.

[12] G. A. Miller, "WordNet: A Lexical Database for English", *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[13] H. Liu and P. Singh, "ConceptNet—A Practical Commonsense Reasoning Tool-kit", *BT technology journal*, vol. 22, no. 4, pp. 211–226, 2004.

[14] M. Nelson and M. Mateas, "Recombinable Game Mechanics for Automated Design Support", in *AIIDE*, 2008.

[15] J. M. Thompson, "Defining the Abstract", *Game & Puzzle Design*, vol. 1, no. 1, pp. 83–86, 2015.

[16] N. Shaker, G. N. Yannakakis, and J. Togelius, "Feature Analysis for Modelling Game Content Quality", in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, IEEE, 2011, pp. 126–133.

[17] A. Liapis, H. P. Martínez, J. Togelius, and G. N. Yannakakis, "Adaptive Game Level Creation through Rank-based Interactive Evolution", in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, IEEE, 2013, pp. 1–8.

[18] R. Lara-Cabrera, C. Cotta, and A. J. Fernández-Leiva, "On Balance and Dynamism in Procedural Content Generation with Self-Adaptive Evolutionary Algorithms", *Natural Computing*, vol. 13, no. 2, pp. 157–168, 2014.

[19] A. Liapis, G. N. Yannakakis, and J. Togelius, "Generating Map Sketches for Strategy Games", in *European Conference on the Applications of Evolutionary Computation*, Springer, 2013, pp. 264–273.

[20] M. Shaker, M. H. Sarhan, O. Al Naameh, N. Shaker, and J. Togelius, "Automatic Generation and Analysis of Physics-based Puzzle Games", in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, IEEE, 2013, pp. 1–8.

[21] N. Shaker, M. Shaker, and J. Togelius, "Evolving Playable Content for Cut the Rope through a Simulation-Based Approach", in *AIIDE*, 2013.

[22] E. L. Lawler, J. K. Lenstra, A. R. Kan, D. B. Shmoys, *et al.*, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley New York, 1985, vol. 3.

[23] A. Isaksen, D. Gopstein, and A. Nealen, "Exploring Game Space Using Survival Analysis", in *FDG*, 2015.

[24] J. Liu, J. Togelius, D. Pérez-Liébana, and S. M. Lucas, "Evolving Game Skill-Depth using General Video Game AI Agents", in *IEEE Proceedings of the Congress on Evolutionary Computation (CEC)*, 2017.

[25] D. Perez-Liebana, S. Samothrakis, J. Togelius, S. M. Lucas, and T. Schaul, "General Video Game AI: Competition, Challenges and Opportunities", in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[26] J. Levine, C. Bates Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, "General Video Game Playing", 2013.

[27] B. Goertzel and C. Pennachin, *Artificial General Intelligence*. Springer, 2007, vol. 2.

[28] Y. Björnsson and S. Schiffel, "General Game Playing", *Handbook of Digital Games and Entertainment Technologies*, pp. 23–45, 2017.

[29] M. Genesereth, N. Love, and B. Pell, "General Game Playing: Overview of the AAAI Competition", *AI magazine*, vol. 26, no. 2, p. 62, 2005.

[30] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents", *J. Artif. Intell. Res.(JAIR)*, vol. 47, pp. 253–279, 2013.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level Control through Deep Reinforcement Learning", *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[32] Y. LeCun, Y. Bengio, *et al.*, "Convolutional Networks for Images, Speech, and Time Series", *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[33] C. J. Watkins and P. Dayan, "Q-learning", *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[34] R. D. Gaina, D. Pérez-Liébana, and S. M. Lucas, "General Video Game for 2 Players: Framework and Competition", in *Computer Science and Electronic Engineering (CEEC), 2016 8th*, IEEE, 2016, pp. 186–191.

[35] J. Liu, D. Perez-Liebana, and S. M. Lucas, *The Single-Player GVGAI Learning Framework - Technical Manual*, 2017. [Online]. Available: http://www.liujialin.tech/publications/GVGAISingleLearning_manual.pdf.

[36] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius, “General Video Game Level Generation”, in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, ACM, 2016, pp. 253–259.

[37] A. Khalifa, M. C. Green, D. Perez-Liebana, and J. Togelius, “General Video Game Rule Generation”, in *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, IEEE, 2017, pp. 170–177.

[38] D. Ashlock, D. Perez-Liebana, and A. Saunders, “General Video Game Playing escapes the No Free Lunch Theorem”, in *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, IEEE, 2017, pp. 17–24.

[39] D. H. Wolpert and W. G. Macready, “No Free Lunch Theorems for Optimization”, *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[40] X. Neufeld, S. Mostaghim, and D. Perez-Liebana, “Procedural Level Generation with Answer Set Programming for General Video Game Playing”, in *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, IEEE, 2015, pp. 207–212.

[41] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A Survey of Monte Carlo Tree Search Methods”, *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[42] T. Bäck and H.-P. Schwefel, “An Overview of Evolutionary Algorithms for Parameter Optimization”, *Evolutionary computation*, vol. 1, no. 1, pp. 1–23, 1993.

[43] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT press, 1998.

[44] I. Rechenberg, “Cybernetic Solution Path of an Experimental Problem”, 1965.

[45] ——, “Evolution Strategy: Optimization of Technical Systems by means of Biological Evolution”, *Fromman-Holzboog, Stuttgart*, vol. 104, 1973.

[46] H. Schwefel, *Numerische Optimierung von Computer-modellen Mittels der Evolutionsstrategie. Interdisciplinary Systems Research, 26, 1977*.

[47] L. J. Fogel, A. J. Owens, and M. J. Walsh, "Artificial Intelligence through Simulated Evolution", 1966.

[48] D. B. Fogel, "An Analysis of Evolutionary Programming", in *Proceedings of the First Annual Conference on Evolutionary Programming*, 1992, pp. 43–51.

[49] D. B. Fogel, L. J. Fogel, and J. W. Atmar, "Meta-evolutionary Programming", in *Signals, systems and computers, 1991. 1991 Conference record of the twenty-fifth asilomar conference on*, IEEE, 1991, pp. 540–545.

[50] J. H. Holland, "Outline for a Logical Theory of Adaptive Systems", *Journal of the ACM (JACM)*, vol. 9, no. 3, pp. 297–314, 1962.

[51] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press, 1992.

[52] B. Kazimipour, X. Li, and A. K. Qin, "A Review of Population Initialization Techniques for Evolutionary Algorithms", in *Evolutionary Computation (CEC), 2014 IEEE Congress on*, IEEE, 2014, pp. 2585–2592.

[53] M. L. Mauldin, "Maintaining Diversity in Genetic Search.", in *AAAI*, 1984, pp. 247–250.

[54] D. Gupta and S. Ghafir, "An Overview of Methods Maintaining Diversity in Genetic Algorithms", *International journal of emerging technology and advanced engineering*, vol. 2, no. 5, pp. 56–60, 2012.

[55] K. A. De Jong, "Analysis of the Behavior of a Class of Genetic Adaptive Systems", 1975.

[56] T. Friedrich, P. S. Oliveto, D. Sudholt, and C. Witt, "Analysis of Diversity-preserving Mechanisms for Global Exploration", *Evolutionary Computation*, vol. 17, no. 4, pp. 455–476, 2009.

[57] L. J. Eshelman and J. D. Schaffer, "Preventing Premature Convergence in Genetic Algorithms by Preventing Incest.", in *ICGA*, vol. 91, 1991, pp. 115–122.

[58] D. E. Goldberg, J. Richardson, *et al.*, "Genetic Algorithms with Sharing for Multimodal Function Optimization", in *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, Hillsdale, NJ: Lawrence Erlbaum, 1987, pp. 41–49.

[59] X. Shen, M. Zhang, and T. Li, "A Multi-objective Optimization Evolutionary Algorithm addressing Diversity Maintenance", in *Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on*, IEEE, vol. 1, 2009, pp. 524–527.

[60] H.-G. Beyer, "Evolutionary Algorithms in Noisy Environments: Theoretical Issues and Guidelines for Practice", *Computer methods in applied mechanics and engineering*, vol. 186, no. 2, pp. 239–267, 2000.

[61] Y. Jin and J. Branke, "Evolutionary Optimization in Uncertain Environments-A Survey", *IEEE Transactions on evolutionary computation*, vol. 9, no. 3, pp. 303–317, 2005.

[62] S. Markon, D. V. Arnold, T. Back, T. Beielstein, and H.-G. Beyer, "Thresholding-a Selection Operator for Noisy ES", in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, IEEE, vol. 1, 2001, pp. 465–472.

[63] B. Levitan and S. Kauffman, "Adaptive Walks with Noisy Fitness Measurements", *Molecular Diversity*, vol. 1, no. 1, pp. 53–68, 1995.

[64] S. Rana, L. D. Whitley, and R. Cogswell, "Searching in the Presence of Noise", in *International Conference on Parallel Problem Solving from Nature*, Springer, 1996, pp. 198–207.

[65] T. Ray, "Constrained Robust Optimal Design using a Multiobjective Evolutionary Algorithm", in *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, IEEE, vol. 1, 2002, pp. 419–424.

[66] W. Cedeno and V. R. Vemuri, “On the Use of Niching for Dynamic Landscapes”, in *Evolutionary Computation, 1997., IEEE International Conference on*, IEEE, 1997, pp. 361–366.

[67] D. E. Goldberg and R. E. Smith, “Nonstationary Function Optimization Using Genetic Algorithms with Dominance and Diploidy.”, in *ICGA*, 1987, pp. 59–68.

[68] R. K. Ursem, “Multinational GAs: Multimodal Optimization Techniques in Dynamic Environments”, in *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, Morgan Kaufmann Publishers Inc., 2000, pp. 19–26.

[69] N. Hansen, A. S. Niederberger, L. Guzzella, and P. Koumoutsakos, “A Method for Handling Uncertainty in Evolutionary Optimization with an Application to Feedback Control of Combustion”, *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 1, pp. 180–197, 2009.

[70] A. Syberfeldt, A. Ng, R. I. John, and P. Moore, “Evolutionary Optimisation of Noisy Multi-objective Problems using Confidence-based Dynamic Resampling”, *European Journal of Operational Research*, vol. 204, no. 3, pp. 533–544, 2010.

[71] T. Friedrich, T. Kötzing, M. S. Krejca, and A. M. Sutton, “The Compact Genetic Algorithm is Efficient Under Extreme Gaussian Noise”, *IEEE Transactions on Evolutionary Computation*, vol. 21, no. 3, pp. 477–490, 2017.

[72] S. M. Lucas, J. Liu, and D. Pérez-Liébana, “Efficient Noisy Optimisation with the Sliding Window Compact Genetic Algorithm”, *ArXiv preprint arXiv:1708.02068*, 2017.

[73] S. Risi and J. Togelius, “Neuroevolution in Games: State of the Art and Open Challenges”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 1, pp. 25–41, 2017.

[74] D. B. Fogel, "Using Evolutionary Programing to Create Neural Networks that are capable of Playing Tic-Tac-Toe", in *Neural Networks, 1993., IEEE International Conference on*, IEEE, 1993, pp. 875–880.

[75] ——, *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann, 2001.

[76] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon, "Further Evolution of a Self-Learning Chess Program.", in *CIG*, 2005.

[77] J. B. Pollack, A. D. Blair, and M. Land, "Coevolution of a Backgammon Player", in *Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA: The MIT Press, 1997, pp. 92–98.

[78] D. E. Moriarty and R. Miikkulainen, "Discovering Complex Othello Strategies through Evolutionary Neural Networks", *Connection Science*, vol. 7, no. 3-1, pp. 195–210, 1995.

[79] G. N. Yannakakis and J. Hallam, "A Generic Approach for Generating Interesting Interactive Pac-Man Opponents.", in *CIG*, 2005.

[80] J. Togelius, S. Karakovskiy, J. Koutnik, and J. Schmidhuber, "Super Mario Evolution", in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, IEEE, 2009, pp. 156–161.

[81] Z. Buk, J. Koutnik, and M. Snorek, "NEAT in HyperNEAT substituted with genetic programming", *Adaptive and Natural Computing Algorithms*, pp. 243–252, 2009.

[82] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time Neuroevolution in the NERO Video Game", *IEEE transactions on evolutionary computation*, vol. 9, no. 6, pp. 653–668, 2005.

[83] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies", *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[84] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving Competitive Car Controllers for Racing Games with Neuroevolution", in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, ACM, 2009, pp. 1179–1186.

[85] E. J. Hughes, "Checkers using a Co-evolutionary On-line Evolutionary Algorithm", in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, IEEE, vol. 2, 2005, pp. 1899–1905.

[86] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling Horizon Evolution versus Tree Search for Navigation in Single-player Real-Time Games", in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, ACM, 2013, pp. 351–358.

[87] R. D. Gaina, J. Liu, S. M. Lucas, and D. Pérez-Liébana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing", in *European Conference on the Applications of Evolutionary Computation*, Springer, 2017, pp. 418–434.

[88] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, "Rolling Horizon Evolution Enhancements in General Video Game Playing", in *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, IEEE, 2017, pp. 88–95.

[89] J. Liu, D. Pérez-Liébana, and S. M. Lucas, "Rolling Horizon Coevolutionary Planning for Two-player Video Games", in *Computer Science and Electronic Engineering (CEEC), 2016 8th*, IEEE, 2016, pp. 174–179.

[90] N. Justesen, T. Mahlmann, and J. Togelius, "Online Evolution for Multi-action Adversarial Games", in *European Conference on the Applications of Evolutionary Computation*, Springer, 2016, pp. 590–603.

[91] D. Churchill and M. Buro, "Portfolio Greedy Search and Simulation for Large-scale Combat in StarCraft", in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, IEEE, 2013, pp. 1–8.

[92] C. Wang, P. Chen, Y. Li, C. Holmgård, and J. Togelius, "Portfolio Online Evolution in StarCraft", in *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[93] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a Video Game Description Language", in *Dagstuhl Follow-Ups*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 6, 2013.

[94] T. Schaul, "A Video Game Description Language for Model-based or Interactive Learning", in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, IEEE, 2013, pp. 1–8.

[95] S. M. Lucas, "Learning to play Othello with N-tuple Systems", *Australian Journal of Intelligent Information Processing*, vol. 4, pp. 1–20, 2008.